VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MULTIDISCIPLINARY PROJECT (CO3109)

Assignment

GreenHouse System

Instructor(s): PhD Lê Trọng Nhân, CSE-HCMUT

Team name: Group: 05 - Class: CC01 - Semester 242

Student(s): Trịnh Anh Minh - 2252493
Cao Ngọc Lâm - 2252419
Nguyễn Tấn Bảo Lễ - 2252428
Nguyễn Châu Hoàng Long - 2252444

HO CHI MINH CITY, JUNE 2025

Contents

1 Member List & Workload Distribution

No.	Full name	Student ID	Assignment	Effort
1	Trịnh Anh Minh	2252493	Research, coding and writing report about Frontend Web Application Module, Use Case diagram and details	100%
2	Cao Ngọc Lâm	2252419	Research, coding and writing report about Backend Server Module	100%
3	Nguyễn Tấn Bảo Lễ	2252428	Research, coding and writing report about AI Module, Introduction and Conclusion	100%
4	Nguyễn Châu Hoàng Long	2252444	Research, coding and writing report about Hardware Module and Requirements	100%

2 Introduction

The rapid advancement of information technology and the growing urgency of sustainable agricultural practices have catalyzed the development of intelligent systems that can enhance efficiency, reduce resource consumption, and improve crop yields. In this context, the GreenHouse System project, conducted under the CO3109 Multidisciplinary Project course at the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, presents a comprehensive and integrated solution for smart greenhouse management.

This project addresses the limitations of traditional greenhouse operations by incorporating a synergy of Internet of Things (IoT) technologies, Artificial Intelligence (AI), and modern web development frameworks to create an end-to-end monitoring and control platform. The system architecture is composed of three primary modules: a frontend user interface, a backend server, and an AI-driven prediction engine. These modules are interconnected through real-time communication protocols and services to facilitate seamless data flow, user interaction, and intelligent decision-making.

The frontend module, developed with ReactJS and Tailwind CSS, provides a responsive and intuitive interface for users to interact with the system, monitor real-time sensor data, manage irrigation processes, and visualize historical trends. The backend server, built with Flask and integrated with Adafruit IO via MQTT, acts as the central coordinator, handling sensor data acquisition, storage, and communication with both the frontend and AI modules. The AI module employs a neural network model to predict plant health status based on live sensor readings, enabling proactive and data-driven greenhouse management.

By deploying a system that supports real-time data streaming, historical analysis, and intelligent feedback, the GreenHouse System exemplifies the application of multidisciplinary knowledge to solve practical problems in environmental sustainability. This report elaborates on the requirements, design methodologies, implementation details, and the collaborative efforts that brought this project to fruition.

- **Link Github Repository of our project:** BKGreenHouse

3 Requirements

3.1 Functional Requirements

1. **Environmental Data Collection:**

   The system must collect the following real-time environmental data:

   - Air temperature
   - Air humidity
   - Soil moisture
   - Light intensity

   Data must be updated every 10 seconds.

2. **Real-Time Monitoring:**

   - Users must be able to view current sensor values in real-time.
   - The interface should display updated data automatically without requiring manual refresh.

3. **Pump Control System:**

   The system must allow:

   - Automatic irrigation based on soil moisture threshold.
   - Pump scheduling, allowing users to set a specific time to turn on the pump.
   - Manual interruption, allowing users to stop the pump even while it is scheduled.

4. **Data History and Search:**

   - Users must be able to view historical sensor data.
   - The system should allow filtering/searching historical data by date, data type, etc.

5. **Plant Condition Prediction:**

   The system should analyze collected data to predict the health or condition of plants, based on environmental parameters.

6. **User Authentication and Authorization:**

   - The system must include user authentication.
   - Only authorized administrators can access system functionalities, especially control features.

3.2 Non-Functional Requirements

1. **Performance**

   - Environmental data must be updated every 10 seconds with a UI update delay of less than 1 seconds.
   - Control operations (e.g., turning pump on/off) should respond within 1 second.

2. **Reliability**

- The system must operate reliably 24/7.
- In case of temporary network disconnection, the system should queue or locally buffer data to prevent data loss.

3. **Security**
   - Data must be transmitted and stored securely (e.g., using HTTPS, encryption).
   - Role-based access control must be enforced (e.g., only admins can control the pump).
4. **Usability**
   - The user interface must be easy to understand and operate, even for non-technical users.
   - The system should support access from desktop, mobile, and tablet devices.
5. **Maintainability**
   - The system should support easy updates to firmware (for hardware) and software.
   - APIs and modular code design should allow easy integration and future improvements.

4 Hardware Module

4.1 Introduction

The hardware module plays a crucial role in sensing and collecting environmental data in real time. It consists of multiple sensors, including a temperature and humidity sensor (DHT20), a soil moisture sensor, and a light intensity sensor. These sensors are connected to a microcontroller board, which continuously reads data and sends it to the cloud. Additionally, the system is equipped with a submersible water pump, which can be controlled automatically based on soil moisture levels or manually via remote commands.

All data is transmitted to the Adafruit IO server using the MQTT protocol, allowing real-time monitoring and remote control via an online dashboard. This design enables users to visualize sensor data, track changes over time, and automate irrigation processes effectively. Overall, the hardware module serves as the foundation for building a smart and connected environmental monitoring system that supports automation and real-time interaction.

4.2 Hardware components

4.2.1 Microcontroller Board

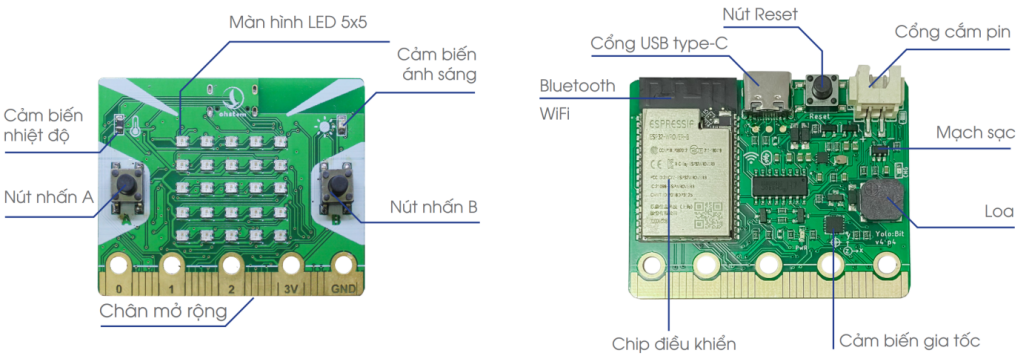


Figure 1: Yolo:Bit Programming Circuit

Yolo:Bit is a compact microcontroller development board designed specifically for educational and IoT (Internet of Things) applications. It is built around the ESP32 chip, which provides integrated Wi-Fi and Bluetooth capabilities, making it ideal for real-time data transmission and wireless control in smart systems.

The board features a 5x5 LED matrix, onboard buttons, a buzzer, and multiple GPIO pins that allow easy connection to sensors and actuators. In this project, Yolo:Bit is responsible for interfacing with environmental sensors (temperature, humidity, soil moisture, light) and controlling the submersible water pump.

Yolo:Bit supports programming in MicroPython, which offers a simple and readable syntax, especially suitable for beginners and rapid prototyping. Thanks to its compact size, built-in communication features, and compatibility with cloud services like Adafruit IO via MQTT protocol, Yolo:Bit serves as the central controller for the entire hardware system.

4.2.2 Sensors

1. **Temperature and Humidity Sensor (DHT20)**

Figure 2: Temperature and Humidity Sensor (DHT20)

- **Working Principle:** The DHT20 is a digital sensor that measures both temperature and relative humidity using a capacitive humidity sensing element and a thermistor. It converts the analog signals from its sensing elements into digital signals using an onboard ADC, providing reliable and calibrated outputs.
- **Connection and Data Acquisition:** The DHT20 communicates via the I2C protocol, requiring only two data lines (SCL and SDA) to interface with the microcontroller. In this project, the sensor is connected directly to the Yolo:Bit board, which reads data at regular intervals and publishes it to the Adafruit IO cloud platform for monitoring.

2. **Soil Moisture Sensor**

Figure 3: Soil moisture sensor

- **Analog/Digital Type:** This project uses an analog soil moisture sensor that detects the volumetric water content of the soil. The sensor outputs a voltage value that varies depending on the moisture level: the wetter the soil, the lower the resistance and the higher the analog output.
- **Irrigation Threshold Setting:** The analog value is read by the microcontroller and compared against a predefined threshold. When the moisture level falls below this threshold, the system triggers the water pump to start irrigation automatically. This ensures that the soil maintains adequate moisture for plant health without human intervention.

3. **Light Sensor**

Figure 4: Light sensor

- **Type Used:** A photoresistor (also known as an LDR - Light Dependent Resistor) is used to detect ambient light levels. Its resistance changes depending on the intensity of light: lower resistance in bright light and higher resistance in darkness.
- **Light Level Monitoring:** The analog output from the photoresistor is read by the microcontroller to determine the surrounding light conditions. This data can be used to analyze plant exposure to sunlight or potentially adjust system behavior based on day/night cycles.

4.2.3 Actuator – Submersible Water Pump

Figure 5: Submersible Water Pump

- **Control Method (Relay):** The system uses a submersible DC water pump to automate irrigation based on soil moisture levels. Since the pump operates at a higher current than the microcontroller can directly provide, it is controlled using a transistor or relay module. In this project, a relay module is used to safely switch the pump on and off by isolating the control circuit (microcontroller) from the power circuit (pump).
- **Manual/Automatic Control Modes:** The pump operates in two modes:

  Automatic Mode: The system continuously monitors the soil moisture sensor. When the moisture level drops below a predefined threshold, the pump is automatically activated to water the plants and is turned off once sufficient moisture is detected.

  Manual Mode: Users can manually control the pump through the Adafruit IO dashboard. By sending a remote control command via MQTT, the pump can be toggled on or off regardless of sensor values, allowing user intervention when needed.

  This dual-mode control system ensures both autonomous operation and remote accessibility, enhancing flexibility and reliability in smart irrigation.

4.3 Circuit Design and Wiring

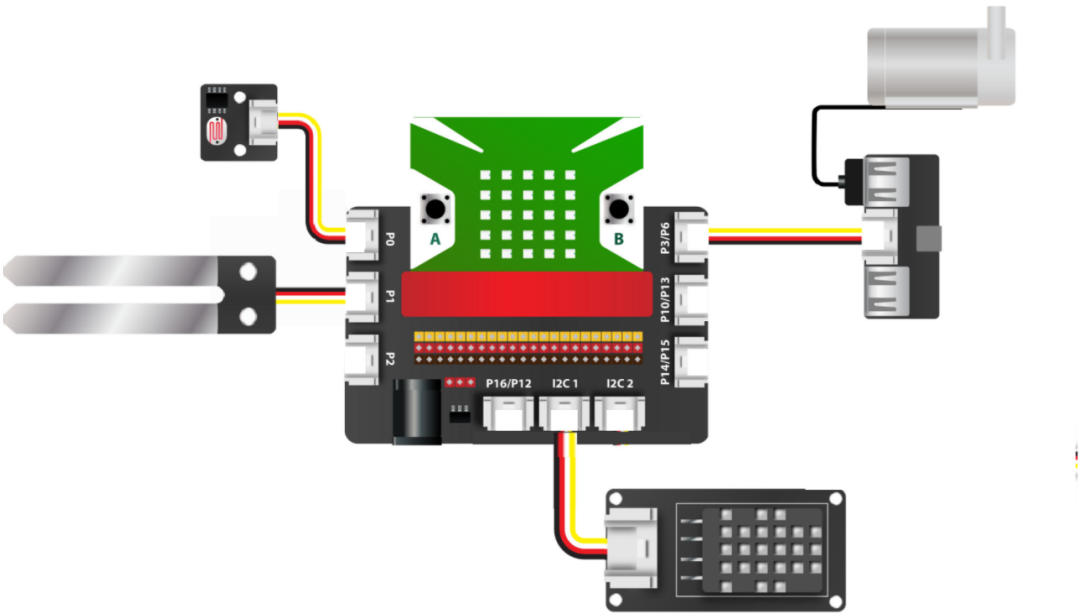


Figure 6: Circuit Design

The hardware circuit is designed around the Yolo:Bit microcontroller, which serves as the central control unit for reading sensor data and controlling the actuator. The system integrates three different types of sensors—temperature and humidity, soil moisture, and light—and one submersible water pump controlled via a relay module. All components are connected to specific ports on the Yolo:Bit board as follows:

- The soil moisture sensor is connected to port P1. It uses an analog signal to measure the water content in the soil. The sensor's output is read as an analog voltage, which is later compared against a predefined threshold to determine whether irrigation is needed.
- The light sensor, which is a photoresistor (LDR), is connected to port P0. It detects ambient light levels by varying its resistance depending on the light intensity. The Yolo:Bit reads the analog signal from this sensor to monitor light conditions for plant growth analysis.
- The DHT20 temperature and humidity sensor is connected to the I2C1 port on the Yolo:Bit. It communicates using the I2C protocol and provides calibrated digital readings of both temperature and relative humidity. This information can be useful for environmental monitoring and decision-making logic in the automation process.
- The submersible water pump is connected to port P3 via a relay module. The relay acts as a switch that allows the microcontroller to control the high-current pump using a low-

current digital signal. When the soil moisture drops below the set threshold, the relay is triggered, activating the pump to irrigate the soil.

Power is supplied to the Yolo:Bit through its standard power port, and the board uses its built-in Wi-Fi capability to connect to the internet. Sensor readings and control commands are transmitted to and from the Adafruit IO platform using the MQTT protocol, enabling real-time monitoring and manual override if needed.

4.4 Data Communication

4.4.1 Cloud Platform – Adafruit IO

Adafruit IO is an easy-to-use cloud service designed for IoT (Internet of Things) applications. It provides real-time dashboards, feeds, and data storage, making it a convenient platform for monitoring and controlling IoT devices over the internet. In this project, Adafruit IO is used to visualize sensor data (temperature, humidity, soil moisture, and light levels) and remotely control the water pump from any internet-connected device.

The system uses **MQTT (Message Queuing Telemetry Transport)**, a lightweight and efficient communication protocol for transmitting data between the Yolo:Bit microcontroller and the Adafruit IO server.

In this project, MQTT topics are used to facilitate two-way communication between the Yolo:Bit microcontroller and the Adafruit IO cloud platform. These topics are categorized into two main types: publishing topics for sending sensor data and subscribing topics for receiving control commands.

Publishing Sensor Data

- temperature: This topic is used to publish real-time temperature readings from the DHT20 sensor.
- humidity: Carries humidity data from the DHT20 sensor.
- soil_moisture: Sends analog values from the soil moisture sensor, representing the water content in the soil.
- light: Publishes light intensity values collected from the photoresistor or light sensor module.

Subscribing to Control Commands

- pump: This topic is used to control the submersible water pump. The microcontroller subscribes to this topic and listens for commands (typically values like 1 to turn on the pump and 0 to turn it off). Users can toggle this value directly from the Adafruit IO dashboard to initiate manual watering.

Additional Topics (Unused/Reserved)

- ai, bkgreenhouse, and topic: These appear to be placeholder or unused topics in the current setup. They may be reserved for future development, such as AI integration, greenhouse grouping, or general-purpose data.

This combination of publishing and subscribing topics allows for full bi-directional communication between the hardware and the cloud, enabling both real-time data visualization and remote device control.

4.4.2 Communication Protocol – MQTT

The communication between the Yolo:Bit microcontroller and the Adafruit IO cloud platform is established using the MQTT (Message Queuing Telemetry Transport) protocol. MQTT is a lightweight, publish-subscribe messaging protocol that is widely used in IoT (Internet of Things) applications due to its simplicity, low bandwidth consumption, and reliable message delivery over unstable networks.

In this project, MQTT enables the bi-directional exchange of data:

- **Sensor readings** are sent (published) from the microcontroller to specific MQTT topics on Adafruit IO.
- **Control commands** (such as turning the pump on or off) are received (subscribed) from Adafruit IO by the microcontroller.

MQTT Setup and Authentication: To connect to Adafruit IO via MQTT, the microcontroller must authenticate using:

- **Username:** The Adafruit IO account username.
- **AIO Key (API Key):** A unique private key generated from the Adafruit IO dashboard.

These credentials are required for every MQTT connection to ensure secure communication between the device and the cloud.

Data Format and Transmission Interval: Sensor data is sent in JSON or plain numeric format, depending on the implementation. Each data point includes the sensor reading and is published every 10 seconds to balance between real-time responsiveness and bandwidth efficiency.

The system ensures that real-time environmental conditions are constantly reflected on the dashboard while minimizing unnecessary network traffic.

Reliability and Efficiency

MQTT is particularly effective in this project because:

- It operates well in **low-power and low-bandwidth** environments like microcontroller-based systems.
- It uses a **persistent connection**, which reduces the overhead of repeated handshakes.
- It supports **QoS (Quality of Service)** levels, although this project primarily uses QoS 0 for speed and simplicity.

The overall communication architecture ensures that the system can be monitored and controlled remotely in real time, even with minimal network resources.

4.5 System Operation Workflow

The hardware module is designed to operate in a continuous cycle that combines periodic sensor readings, wireless data transmission, reception of control commands, and local decision-making logic for automated irrigation. This workflow enables seamless real-time environmental monitoring and remote control of devices via the Adafruit IO cloud platform.

4.5.1 Code

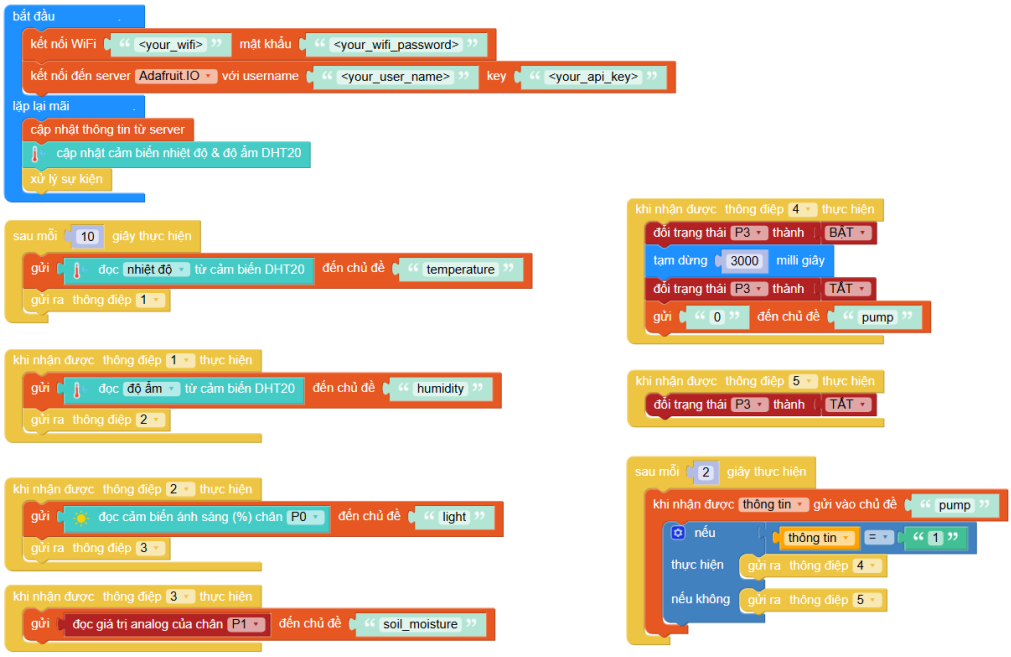


Figure 7: Block-based Programming

The above image illustrates a block-based program developed using the OYSystem platform provided by OhStem Education. This platform utilizes Blockly – a visual programming tool developed by Google – to enable users, especially students and beginners, to create IoT applications through drag-and-drop programming blocks without writing traditional code.

OYSystem supports real-time interaction with hardware such as the Yolo:Bit microcontroller, and provides built-in integration with cloud services like Adafruit IO, making it ideal for building smart systems such as automated plant watering systems.

The program above performs the following key operations:

1. **WiFi and MQTT Connection:**
   - Connects the Yolo:Bit to a Wi-Fi network using a specified password.
   - Establishes a connection with Adafruit IO cloud server using a specific username and API key.
2. **Periodic Sensor Data Publishing:**
   - Every 10 seconds, the program reads the temperature from the DHT20 sensor and publishes it to the temperature topic.
   - Every 10 seconds, the program reads the temperature from the DHT20 sensor and publishes it to the temperature topic.
3. **Sensor Data on Request:** When a control message with a specific code (1, 2, or 3) is received:

- 1 → Reads and sends humidity data to topic humidity.
- 2 → Reads and sends light level to topic light.
- 3 → Reads soil moisture level from analog pin and sends to soil_moisture.

4. **Pump Control Logic:** Every 2 seconds, the Yolo:Bit checks for control information sent to the topic pump:

   If the message is "1" (activate pump), it triggers a sequence:

   - Turns P3 (pump pin) ON → waits 3 seconds → turns it OFF.
   - Sends "0" back to the pump topic as feedback.

   If the message is not "1", it sends a status message 5 to indicate no action.

4.5.2 Periodic Data Reading

At the heart of the system is the Yolo:Bit microcontroller, which interfaces with multiple sensors to collect environmental data. The system reads sensor values at regular intervals, typically every 5 to 10 seconds. The sensors include:

- **DHT20:** Measures ambient temperature and relative humidity. These values are crucial for understanding atmospheric conditions that influence plant health and evaporation rates.
- **Soil Moisture Sensor:** Provides an analog value representing the moisture level in the soil. It helps determine whether the soil is too dry and if irrigation is required.
- **Light Sensor** (photoresistor): Monitors the intensity of ambient light, which is useful for understanding daily sunlight exposure, an important factor in plant photosynthesis.

The data collection process is optimized to be lightweight and fast, ensuring minimal delay and power consumption while maintaining up-to-date readings.

4.5.3 Sending Sensor Data to the Server

Once sensor data is collected, it is formatted (typically as plain text values) and sent to the Adafruit IO server via MQTT. Each type of data is published to a specific MQTT topic under the user's feed namespace, such as:

- temperature
- humidity
- soil_moisture
- light

These MQTT messages are sent over Wi-Fi using the lightweight MQTT protocol, which ensures reliable and efficient communication with minimal overhead. The published data can then be displayed on the Adafruit IO dashboard in the form of graphs, gauges, or numerical values. This allows users to monitor environmental parameters in real time from any device with internet access.

4.5.4 Receiving Control Signals from the Server

In addition to publishing data, the Yolo:Bit microcontroller continuously subscribes to the pump topic to listen for incoming control commands from the server. This feature enables remote actuation of the water pump from the dashboard. For instance:

- A value of "1" indicates the user wants to activate the pump.
- A value of "0" turns the pump off.

This setup allows for manual control of the irrigation system through the web interface. The subscription ensures that even if the system is unattended, users can trigger irrigation remotely when needed.

4.5.5 Automatic Irrigation Logic

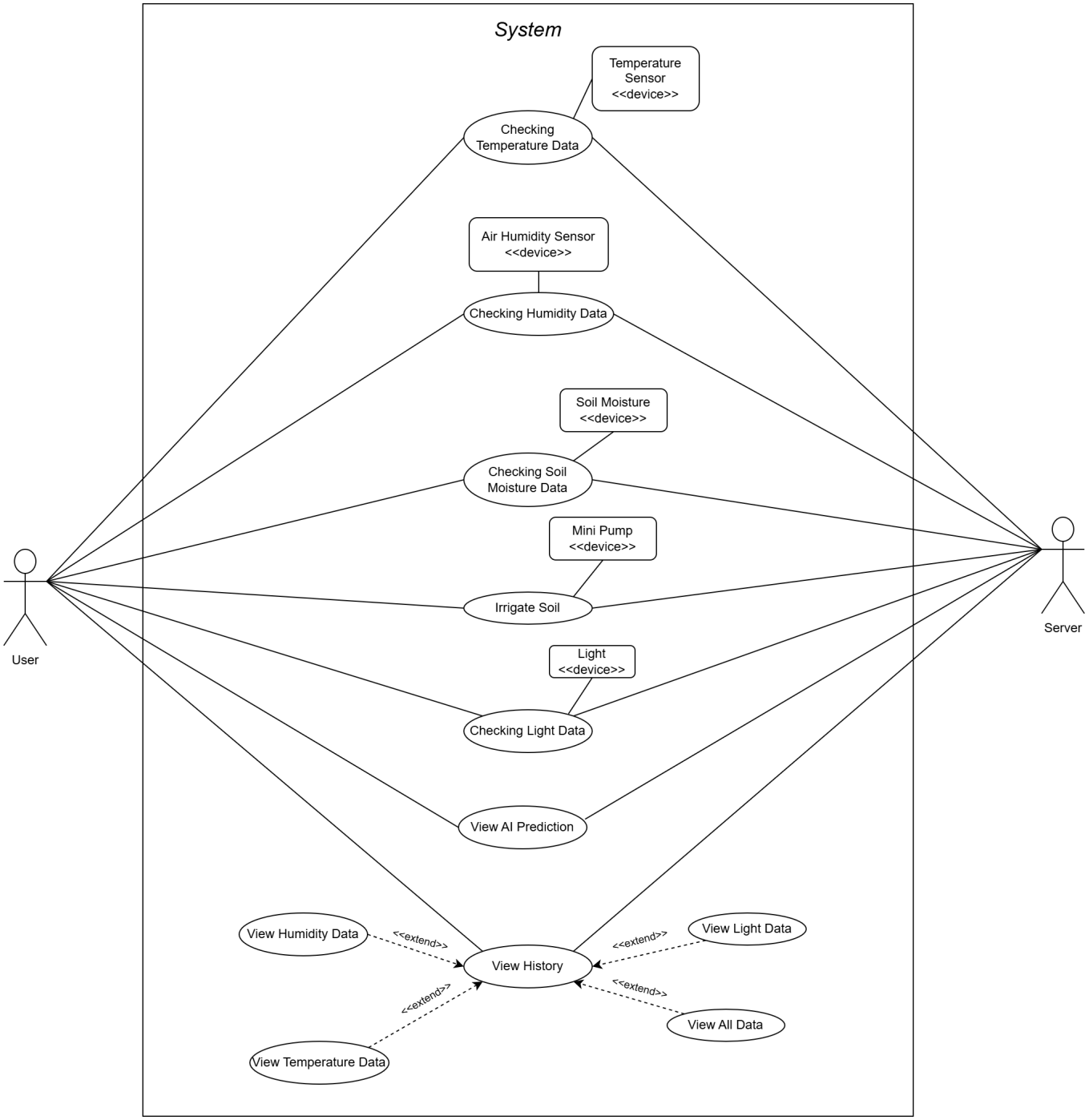
Beyond manual control, the system also implements autonomous irrigation logic directly on the microcontroller. This logic monitors the real-time soil moisture values and compares them to a predefined threshold. The logic works as follows:

- If soil moisture falls below the threshold: The system automatically activates the submersible water pump to irrigate the plants.
- If the soil moisture reaches or exceeds the threshold: The pump is turned off to avoid overwatering and conserve water.

This decision-making occurs locally on the Yolo:Bit, ensuring fast response without depending on continuous cloud access. This approach improves system reliability, especially in unstable network conditions.

The automatic irrigation feature ensures that the plants receive water only when necessary, reducing waste and promoting sustainable agricultural practices. It also minimizes the need for human supervision, making the system ideal for small greenhouses, indoor gardens, and educational purposes.

5 Use Case Diagram of Whole System



6 Use Case Details

6.1 Use Case 1: Login Authentication

ID and Name	UC-1 Login
Date Created	12th June, 2025
Actor	User
Description	The user enters their email and password to log into the Green House System. Upon successful authentication, the system redirects the user to the system dashboard.
Trigger	The user selects the "Đăng nhập" (Log In) button on the login form.
Preconditions	**PRE-1.** The user has a registered email and password. **PRE-2.** The system is online and functional.
Postconditions	**POST-1.** The user is granted access to the dashboard interface. **POST-2.** The login session is established for the authenticated user.
Normal Flow	**1.0. Log In Process** 1. The user opens the Green House System login page. 2. The system displays a login form with fields for email and password. 3. The user enters a valid email and password. 4. The user optionally checks “Ghi nhớ đăng nhập” (Remember me). 5. The user clicks the “Đăng nhập” button. 6. The system verifies the email and password: • If valid, proceed to step 7. • If invalid, go to Alternative Flow 1.1. 7. The system grants access and redirects the user to the dashboard.
Alternative Flows	**1.1. Invalid Credentials** 1. The system shows an error message: “Sai email hoặc mật khẩu” (Invalid email or password). 2. The user is prompted to re-enter login information.
Exceptions	**1.1 E1 – System Error / No Server Response** System displays an error message and prompts the user to try again later.

6.2 Use Case 2: Irrigation System

ID and Name	UC-2 Control Irrigation System
Date Created	12th June, 2025
Actor	User, server, device (mini pump)
Description	The user can remotely control the irrigation system of the greenhouse by setting a desired watering duration and activating or deactivating the water control system.
Trigger	The user clicks the "Start Watering" button at section of control watering system.
Preconditions	**PRE-1.** The user is logged into the system. **PRE-2.** The Water Control System is currently INACTIVE. **PRE-3.** The selected watering time is within the valid range (5–300 seconds).
Postconditions	**POST-1.** The watering process is executed for the selected duration. **POST-2.** The system returns to the INACTIVE state after watering completes.
Normal Flow	**2.0. Control Irrigation System Process** 1. The user logs into the system and navigates to the Device Controls section. 2. The user sees the current status of the watering system (ACTIVE or INACTIVE). 3. The user sets the watering time using: • Increment/Decrement the time buttons (+ / -) • Or one of the quick-select buttons for setting the time (10s, 30s, 60s, 120s) 4. The user clicks the "Start Watering" button to activate the irrigation system for the selected time. 5. The system updates the status to ACTIVE and start the watering process. 6. After the set duration, the system automatically turns off the water and updates the status to INACTIVE.
Alternative Flows	**2.1. Watering in Progress** 1. If the system is already running, the "Start Watering" button is disabled until the current session ends. 2. The system shows a message: “No data available.”
Exceptions	**2.1 E1 – IOT server is unreachable** • The system shows a failure message and remains INACTIVE.

6.3 Use Case 3: View Data History

ID and Name	UC-3 View Data History
Date Created	12th June, 2025
Actor	User, server
Description	The user selects a type of data (Temperature, Air Humidity, Soil Moisture, or All) and optionally picks a specific date to view the historical environmental data (highest and lowest) stored in the system.
Trigger	The user navigates to the "History" tab in the system menu.
Preconditions	**PRE-1.** The user is logged in to the system. **PRE-2.** Historical data is available in the database.
Postconditions	**POST-1.** The user sees the data history (if available) filtered by type and/or date. **POST-2.** The system returns the highest and lowest data records that match the filter.
Normal Flow	**3.0. View Historical Data Process** 1. The user clicks on the "History" tab. 2. The system displays the "Data History" page with data filters. 3. The user selects a data type: • Temperature • Air Humidity • Soil Moisture • All 4. The user optionally chooses a date using the date picker. 5. The user clicks the “Search” button. 6. The system retrieves and displays the historical data in a table: • Date • Highest Data • Lowest Data 7. The user reviews the displayed results.
Alternative Flows	**3.1. No Data Available** 1. The system finds no data matching the filters. 2. The system shows a message: “No data available.”

Exceptions	**3.1 E1 – Invalid Date Format** • System displays an error message and prompts the user to try again later. **3.2 E2 – Server Error** • The system cannot fetch data due to server issues. • System displays an error message and prompts the user to try again later.

7 Frontend Web Application Module

7.1 Introduction

The Frontend Module of the GreenHouse System provides an intuitive and responsive user interface that enables users to monitor and control greenhouse operations seamlessly. Developed using modern web technologies such as ReactJS, JavaScript, TailWind CSS, the frontend acts as a bridge between users and the backend services. This module is designed to display real-time environmental data, system statuses, and AI-driven insights about plant conditions received from the backend via Server-Sent Events (SSE). It also allows users to interact with various IoT components (e.g., setting the watering system) through a clean and accessible interface.

7.2 Welcome Page

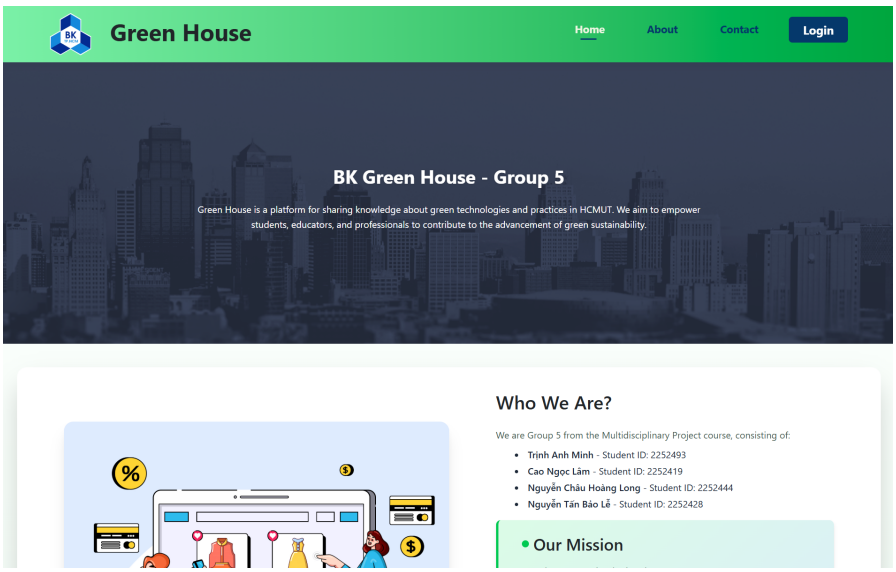


Figure 8: Welcome Page

The Home page of the Green House system offers a clear overview of the platform's mission and vision. On this page, users can learn about the purpose of Green House: a platform to share knowledge on green technologies and sustainable practices at HCMUT. Get to know the development team (Group 5), including basic information about each member. People can understand the mission of the project - to support students, educators, and the community in promoting environmental awareness and sustainable development.

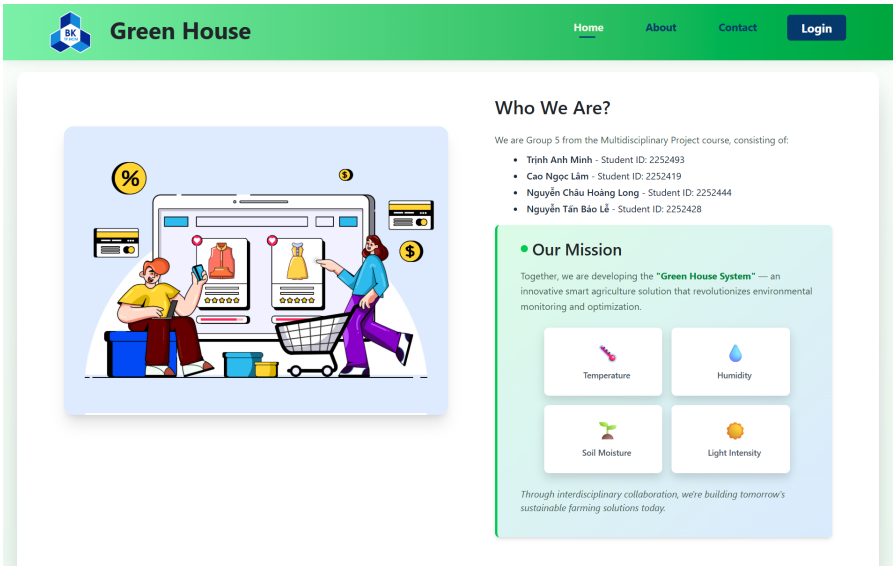


Figure 9: About our team

7.3 Authenticate Page

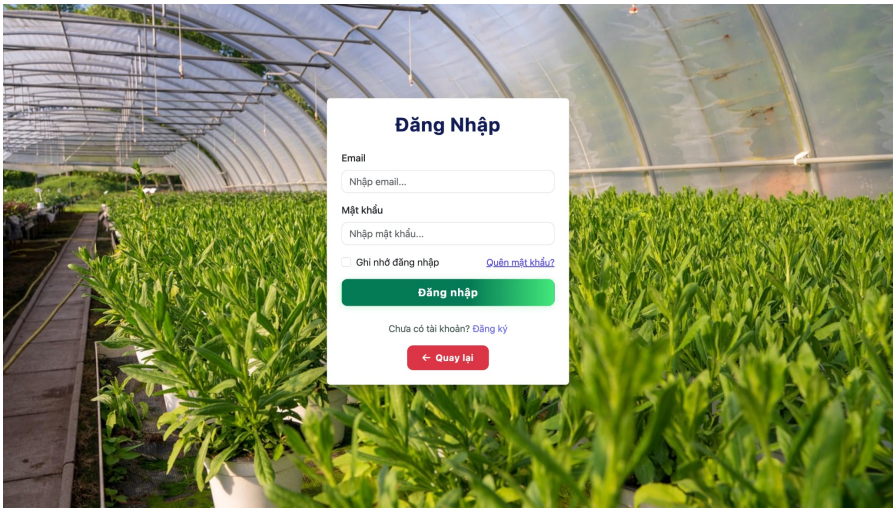


Figure 10: Login Authentication Page

The Authenticate pages includes the following components:

- "Email" field: Users input their email address (e.g., greenhouse@gmail.com).
- "Password" field: Users enter their password.
- Additional options include a "Remember login" checkbox and a "Forgot password?" link to assist users during the login process.
- A green "Login" button to confirm the entered information.
- Upon successful login, a "Login successful!" notification is displayed, indicating that the user has been authenticated.
- Upon successful login, a "Login successful!" notification is displayed, indicating that the user has been authenticated.
- If the login information is incorrect, the system will implie a return to the login page for re-entry of credentials and an error message prompting correction.
- If the user lacks an account or chooses not to proceed, the "Don't have an account? Register" link directs them to a registration page, while the "Back" button allows a return to the homepage.

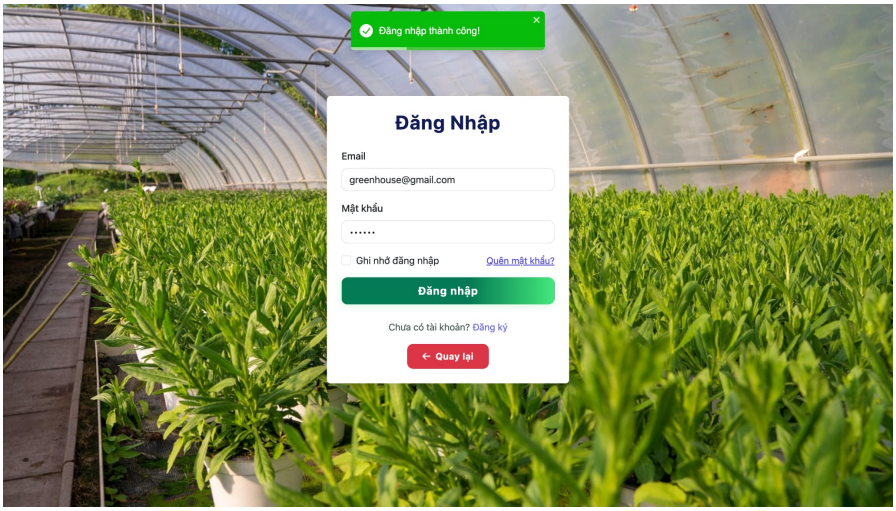


Figure 11: Login Authentication Successfully Page

After login success, a notification will be toasted on the top of the page and then the browser will navigate to the home page after around 3 seconds.

7.4 Home page

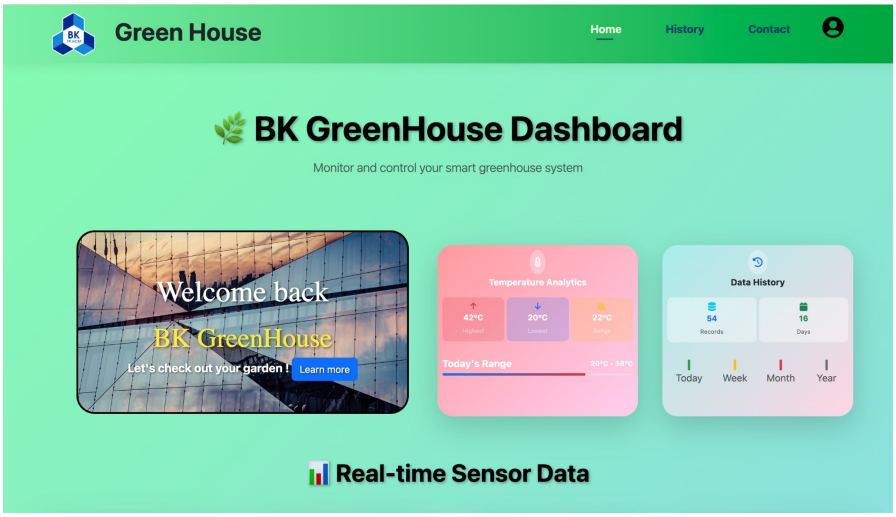


Figure 12: BK GreenHouse Dashboard

The home page contains summary data for temperature analytics, presenting real-time metrics such as the highest (42°C), lowest (20°C), and average (27°C) temperatures, alongside a visual range indicator. Additionally, it includes summary data for historical records, accessible across various time frames (Today, Week, Month, Year) with corresponding data points (54 records, 16 days).

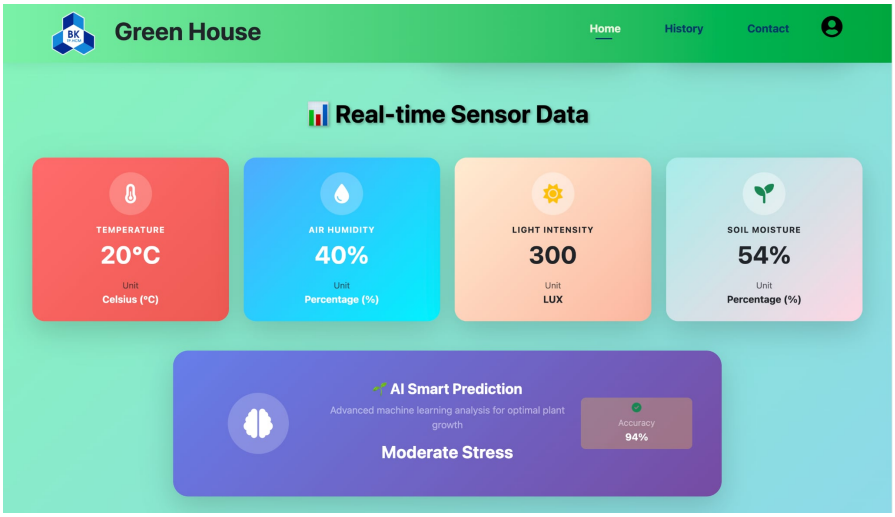


Figure 13: Real time sensor data and AI Prediction

The Real-time Sensor Data section provides a continuous overview of live sensor inputs, ensuring effective monitoring and control of the smart greenhouse system. They are the continuous data collected from the Iot sensor devices, transfer through the Adafruit IoT server to the Backend server and send to the Web client. In addition, there is an AI Smart Prediction section, which is took from the AI Module prediction to optimize plant growth.

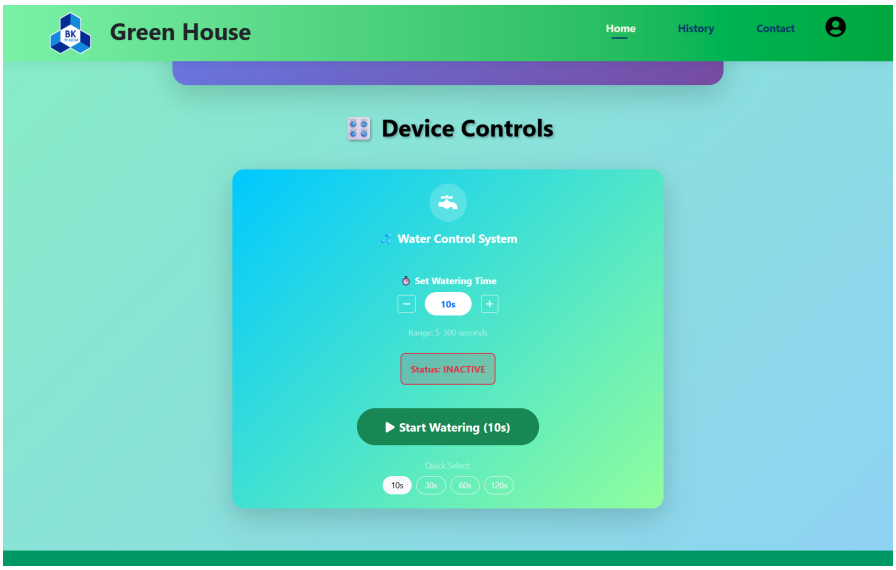


Figure 14: Water Control System Section

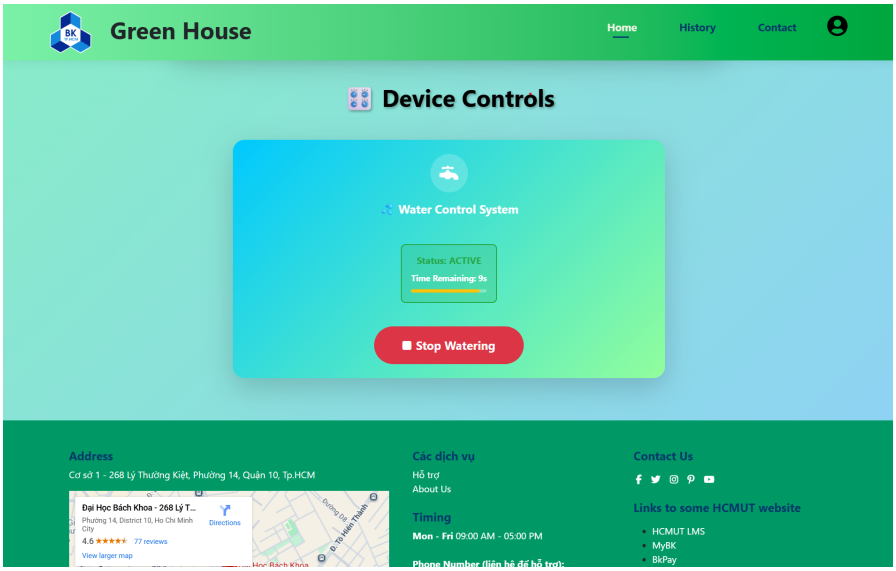


Figure 15: Irrigation system is working

The Device Controls section of the dashboard, as depicted in Figure 7 and Figure 8, contains the Water Control System interface. This section allows users to manage the watering system, featuring options to set watering time with a default of 10 seconds (adjustable within a range of 5 to 1000 seconds) and display the current status (e.g., "Inactive" or "Active"). The interface includes a "Start Watering (10s)" button to initiate the process and a prominent "Stop Watering" button to halt it, ensuring precise control.

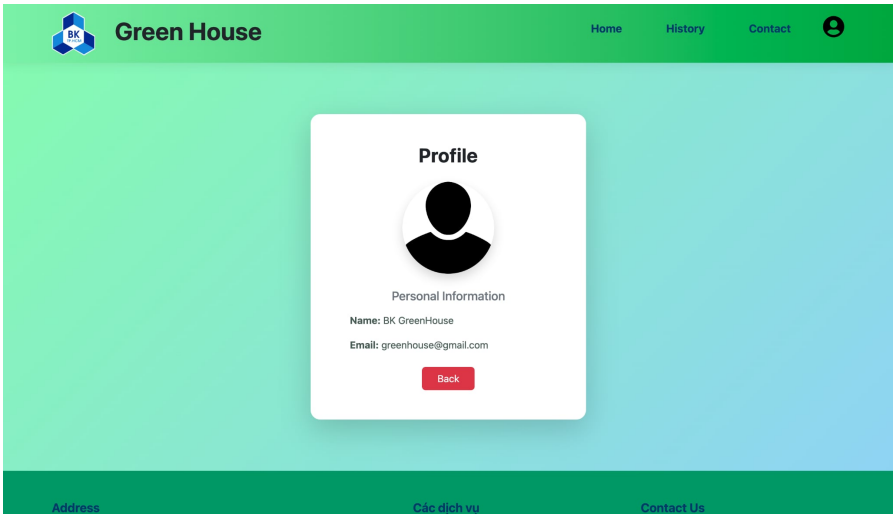


Figure 16: User Profile

In order to view the current profile, click into the provide section in the avatar dropdown. The Profile page contain the name and email of the current admin. It also have a button to redirect to the Dashboard page.

7.5 History Screen

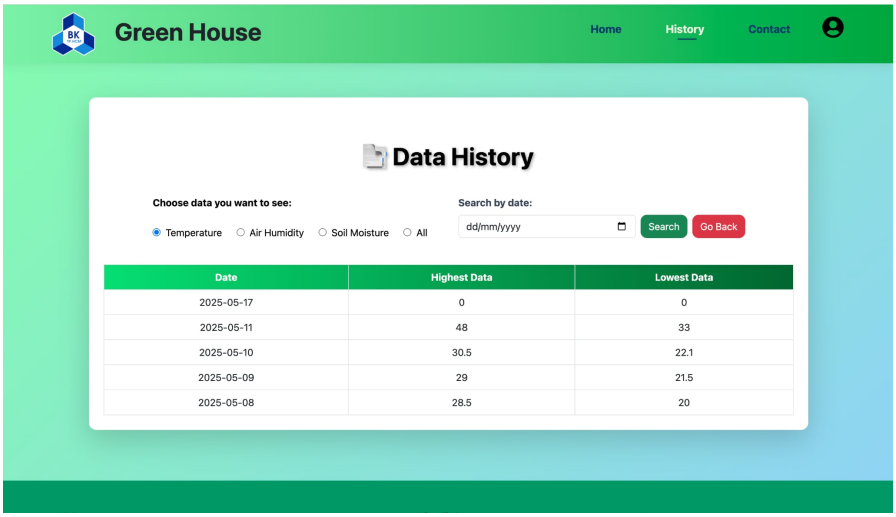


Figure 17: Data History Page

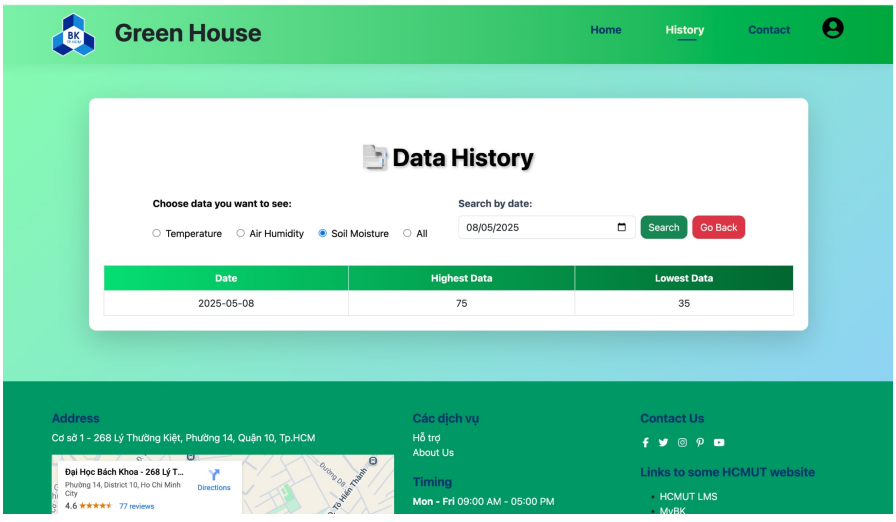


Figure 18: Choosing Soil Moisture Data History

The History Screen, as illustrated in Figures 10 and 11, contains the Data History page, allowing admin to review historical data by selecting specific data types (Temperature, Air Humidity, Soil Moisture, or All) and searching by date, with a default search option available. The data table displays key metrics, including the date, highest data value, and lowest data value for the selected parameter, as shown with examples such as 2025-05-17 (0 to 0), 2025-05-11 (48 to 33), and 2025-05-09 (29 to 21.5) for temperature. Figure 11 demonstrates the interface when filtering for Soil Moisture data, showcasing a specific entry for 2026-08-08 with a highest value of 75 and a lowest value of 33.

8 Backend Server Module

8.1 Introduction

The Backend module of the GreenHouse System serves as the central hub for data processing, communication, and integration among the IoT devices, AI module, and Front End interface. Built using Flask, a lightweight Python web framework, the backend is responsible for receiving real-time sensor data from Adafruit IO, processing and storing it, integrating AI-driven tree condition analysis, and delivering updates to the frontend via Server-Sent Events (SSE).

8.2 Objectives

The backend module aims to achieve the following:

- **Data Acquisition**: Retrieve real-time environmental data (e.g., temperature, humidity, soil moisture) and AI-generated tree condition data from Adafruit IO.
- **Data Processing**: Store and process incoming data for efficient retrieval and analysis.
- **Real-Time Communication**: Push processed sensor data and AI insights to the React FrontEnd using SSE for real-time updates.
- **System Integration**: Ensure seamless interaction between IoT devices, the AI module, and the Frontend.

8.3 System Architecture

The backend is a Flask application integrated with Adafruit IO via MQTT and REST APIs, a SQLite database for storage, and an SSE endpoint for frontend updates. Key components include:

- **Flask Server**: Manages routes, processes data, and serves the SSE stream.
- **Adafruit IO Client**: Subscribes to MQTT feeds for sensor and AI data.
- **SQLite Database**: Stores daily records of environmental data.
- **SSE Endpoint**: Delivers real-time updates to the React frontend.

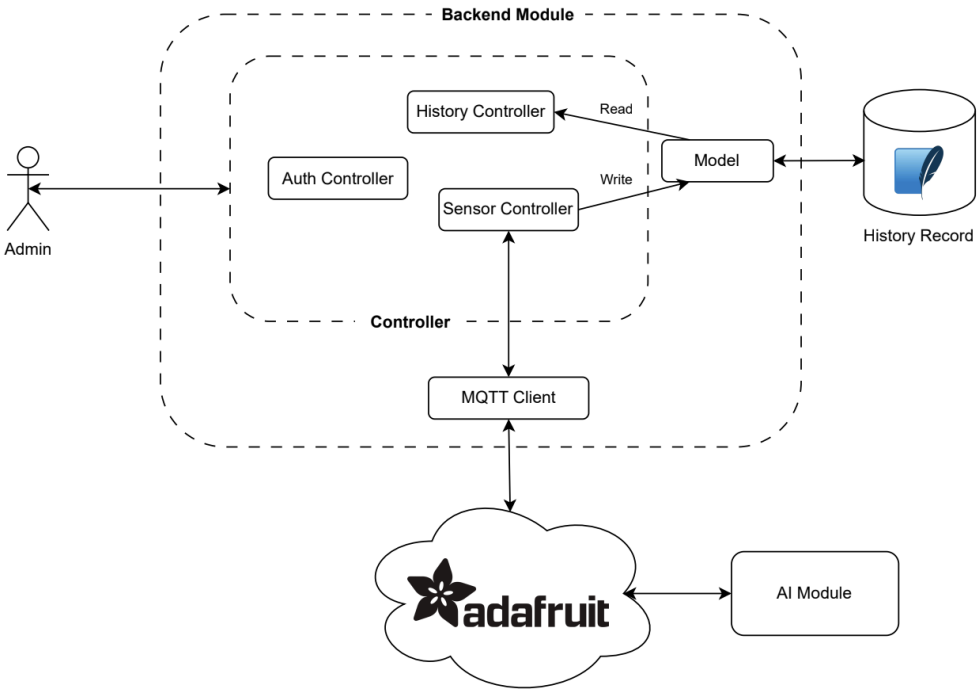


Figure 19: Backend Module Diagram

Data flows as follows:

1. IoT sensors publish real-time environmental data (e.g., temperature, humidity, soil moisture) to Adafruit IO feeds via MQTT.
2. The AI module retrieves sensor data from Adafruit IO and updates tree condition feeds with analysis results.
3. The `SensorController` in the backend processes the data and updates the SQLite database with new or adjusted daily records.
4. Processed data is streamed to the React frontend in real time via Server-Sent Events (SSE) for dynamic monitoring.
5. For requests from the React client to read historical data, the `HistoryController` handles the retrieval of data from SQLite.

8.4 Database

The database component of the GreenHouse System is designed to store and manage environmental data efficiently using SQLite, a lightweight and serverless database engine. This section outlines the database structure and its integration with the backend.

```
-- Create the Daily Records table
CREATE TABLE IF NOT EXISTS daily_records (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date TEXT UNIQUE DEFAULT (DATE('now')),
    max_temperature REAL DEFAULT 0,
    min_temperature REAL DEFAULT 0,
    max_soil_moisture REAL DEFAULT 0,
    min_soil_moisture REAL DEFAULT 0,
    max_humidity REAL DEFAULT 0,
    min_humidity REAL DEFAULT 0
)
```

The database is initialized with a primary table named `daily_records`, which is designed to aggregate daily environmental data collected from IoT sensors and processed by the backend. This table captures essential metrics such as temperature, soil moisture, and humidity, storing both the maximum and minimum values observed each day to provide a comprehensive picture of greenhouse conditions. The structure is optimized for efficiency and scalability, with the following key attributes:

1. `id`: An auto-incrementing integer serving as the primary key to uniquely identify each record.
2. `date`: A text field with a unique constraint, defaulting to the current date using the DATE('now') function, ensuring that each day's data is recorded only once.
3. `max_temperature` and `min_temperature`: Real number fields to store the highest and lowest temperature readings, defaulting to 0 if no data is available.
4. `max_soil_moisture` and `min_soil_moisture`: Real number fields to track the range of soil moisture levels, also defaulting to 0.
5. `max_humidity` and `min_humidity`: Real number fields for recording the daily range of humidity levels, initialized to 0.

9 AI Module

9.1 Introduction

The AI module of the GreenHouse System serves as the central hub for processing real-time sensor data and delivering AI-driven plant health predictions. Built using Python and integrated with Adafruit IO's MQTT client, the backend receives sensor data (ambient temperature, soil moisture, light intensity, and humidity) from IoT devices, processes it using a pre-trained TensorFlow/Keras neural network model, and logs predictions to a CSV file. The system ensures robust data handling by incorporating error checking, data scaling with a pre-trained StandardScaler, and label encoding for consistent plant health status outputs, enabling seamless integration with external interfaces or monitoring systems.

9.2 Objectives

The AI module aims to achieve the following:

- **Real-Time Sensor Data Collection**: Subscribe to Adafruit IO feeds to collect real-time environmental data, including ambient temperature, soil moisture, light intensity, and humidity, from IoT devices.
- **AI-Driven Plant Health Prediction**: Utilize a pre-trained TensorFlow/Keras neural network model to process sensor data and predict plant health status with confidence scores, ensuring accurate and reliable assessments.
- **Data Preprocessing and Consistency**: Apply a pre-trained StandardScaler and LabelEncoder to normalize sensor inputs and encode health status outputs, maintaining consistency with the model's training phase.
- **Data Logging and Monitoring**: Log sensor inputs and predicted plant health statuses with timestamps to a CSV file (data_log.csv) for historical analysis and system monitoring.
- **Robust Error Handling**: Implement error checking for invalid sensor data and prediction failures to ensure system stability and graceful recovery from issues.
- **Real-Time Result Publishing**: Publish AI-predicted plant health statuses to an Adafruit IO output feed, enabling integration with external systems or user interfaces.
- **Scalable and Secure Operation**: Use environment variables for secure credential management and maintain a lightweight, scalable architecture for processing sensor data in real time.

9.3 System Architecture

The diagram illustrates an "AI Module" at the heart of an IoT system designed for processing sensor data and making ML predictions, then publishing those predictions.

1. **IoT Devices to Adafruit IO Feeds (Input):**
    - **Action**: "publish"
    - **Description**: Various IoT devices (e.g., sensors collecting environmental data) transmit their raw data.

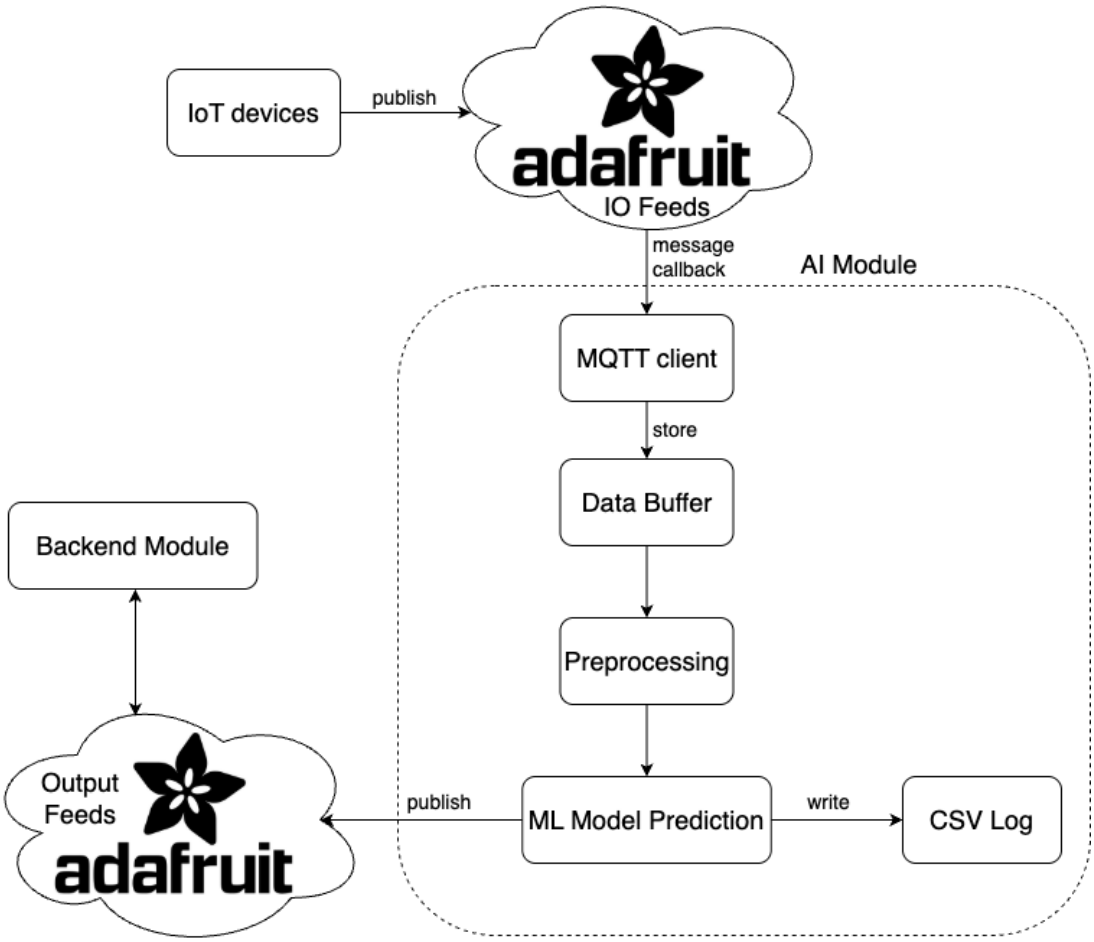


Figure 20: AI Module Diagram

- **Protocol**: This data is sent to the Adafruit IO cloud platform. Based on previous discussions, this "publish" action would typically occur via MQTT or HTTP POST requests.
- **Destination**: The raw sensor data lands in designated Adafruit IO Feeds, acting as input channels for the system.

2. **Adafruit IO Feeds to AI Module (Input Consumption):**

- **Action**: "message callback"
- **Description**: The "AI Module" actively monitors the input Adafruit IO Feeds.
- **Protocol**: This is primarily handled by an MQTT client within the AI Module. The client is subscribed to these input feeds. When new data arrives on a subscribed feed, the MQTT client receives it via a "message callback" mechanism.

3. **Inside the AI Module (Processing Pipeline): MQTT Client to Data Buffer:**
    - **Action**: "store"
    - **Description**: The data received by the MQTT client from the Adafruit IO feeds is temporarily stored in a "Data Buffer." This is crucial when multiple sensor readings are required for a single prediction. The buffer waits until all necessary data points are collected.
4. **Data Buffer to Preprocessing:**
    - **Action**: Data flow
    - **Description**: Once all required raw sensor values are buffered, they are passed to a "Preprocessing" step. This step prepares the data for the ML model, often involving normalization, scaling, or feature engineering.
5. **Preprocessing to ML Model Prediction:**
    - **Action**: Data flow
    - **Description**: The preprocessed data is then fed into the "ML Model Prediction" component. This is where the trained machine learning model performs inference to generate a prediction (e.g., plant health status).
6. **ML Model Prediction to CSV Log:**
    - **Action**: "write"
    - **Description**: The raw input data, along with the generated prediction, is written to a "CSV Log" file. This provides a persistent record of the data and the model's output for analysis or debugging.
7. **AI Module to Adafruit IO Output Feeds (Output Publication):**
    - **Action**: "publish"
    - **Description**: The prediction result from the ML model is sent back to the Adafruit IO cloud.
    - **Protocol**: Similar to the IoT devices, the AI Module uses its MQTT client to "publish" the prediction result.
    - **Destination**: This prediction goes to a dedicated Adafruit IO Output Feed, making the model's inference available to other parts of the system.
8. **Adafruit IO Output Feeds to Backend Module:**
    - **Action**: Data flow
    - **Description**: Other components, like a "Backend Module" (which might be a web application, a mobile app, or another IoT service), can then consume the predictions from the Adafruit IO Output Feeds. This allows for displaying results, triggering alerts, or further automated actions.

9.4 Source code

9.4.1 Connection and real-time streaming

The code represents the "AI Module" from above diagram, specifically designed for real-time inference of plant health using sensor data streamed via Adafruit IO. It acts as the operational brain of our IoT system.

```
import sys
import time
import csv
import os
from datetime import datetime

import pandas as pd
import numpy as np
from tensorflow.keras.models import load_model
from sklearn.preprocessing import StandardScaler
import joblib
from Adafruit_IO import MQTTClient
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Adafruit IO credentials and feed IDs
AIO_FEED_ID = os.getenv("AIO_FEED_ID").split(",") # Input topics
AIO_USERNAME = os.getenv("AIO_USERNAME")
AIO_KEY = os.getenv("AIO_KEY")
AIO_RESULT_FEED = os.getenv("AIO_RESULT_FEED") # Output topic

# Load saved ML model, scaler, and label encoder
model = load_model("plant_health_model.h5")
scaler = joblib.load("scaler.save")
label_encoder = joblib.load("label_encoder.save")

# Buffer for storing incoming sensor values
sensor_data = {}

# Callback when connected
def connected(client):
    print("Connected to Adafruit IO")
    for feed in AIO_FEED_ID:
        client.subscribe(feed)
        print(f"Subscribed to {feed}")

# Callback when subscription succeeds
def subscribe(client, userdata, mid, granted_qos):
    print("Subscription successful")

# Callback when disconnected
def disconnected(client):
    print("Disconnected from Adafruit IO")
    sys.exit(1)
```

```
# Callback for incoming messages
def message(client, feed_id, payload):
    print(f"Data from {feed_id}: {payload}")
    try:
        sensor_data[feed_id] = float(payload)
    except ValueError:
        print(f"Invalid data from {feed_id}: {payload}")
        return

    # When all sensor data is available, predict
    if all(feed in sensor_data for feed in AIO_FEED_ID):
        try:
            # Ensure the order matches the model input
            print('here')
            input_data = [sensor_data[feed] for feed in AIO_FEED_ID]
            input_df = pd.DataFrame([input_data], columns=["Ambient_Temperature",
                "Soil_Moisture", "Light_Intensity", "Humidity"])
            input_scaled = scaler.transform(input_df)

            probs = model.predict(input_scaled)[0]
            predicted_idx = np.argmax(probs)
            predicted_label = label_encoder.inverse_transform([predicted_idx])[0]
            confidence = probs[predicted_idx]

            print(f"Predicted Health: {predicted_label} ({confidence:.2%})")

            # Publish prediction to Adafruit
            client.publish(AIO_RESULT_FEED, predicted_label)

            # Log the result
            with open("data_log.csv", "a", newline="") as f:
                writer = csv.writer(f)
                writer.writerow([datetime.now()] + input_data + [predicted_label])

            # Reset for next batch
            sensor_data.clear()

        except Exception as e:
            print("Error during prediction:", e)

# Configure and start the MQTT client
client = MQTTClient(AIO_USERNAME, AIO_KEY)
client.on_connect = connected
client.on_disconnect = disconnected
client.on_message = message
client.on_subscribe = subscribe

client.connect()
client.loop_background()

# Keep the script alive
print("Listening for sensor data...")
```

```
while True:
    time.sleep(1)
```

Key Responsibilities and Components:

1. **Environment Setup & Configuration:**
    - Secure Credential Handling: Utilizes python-dotenv to load API keys and usernames from a `.env` file, preventing hardcoding sensitive information directly into the script. This is crucial for security and portability.
    - Adafruit IO Integration: Defines input `AIO_FEED_IDs` (a comma-separated list of MQTT topics for sensor data) and an `AIO_RESULT_FEED` (the MQTT topic for publishing predictions).
2. **ML Model and Preprocessing Assets Loading:**
    - Pre-trained Model: Loads the `plant_health_model.h5` Keras model, which is the core of the prediction logic.
    - Scaler: Loads the `scaler.save` (a StandardScaler object) used for feature scaling. This ensures that incoming raw sensor data is transformed consistently with how the model was trained.
    - Label Encoder: Loads the `label_encoder.save` (a LabelEncoder object) to convert numerical model outputs back into human-readable plant health status labels (e.g., 0 -> "Healthy", 1 -> "Unhealthy").
3. **Data Ingestion and Buffering (MQTT Client):**
    - MQTT Client: Instantiates `Adafruit_IO.MQTTClient` to connect to the Adafruit IO broker.
    - Subscription: Upon connection, it subscribes to all specified input `AIO_FEED_IDs`. This means it actively listens for new data published by the IoT sensor devices.
    - Message Callback (message function): This is the heart of the data reception.
    - It captures incoming sensor data `(feed_id, payload)`.
    - Data Buffer (`sensor_data` dictionary): It stores individual sensor readings (e.g., temperature, humidity) in a dictionary. This is essential because the ML model requires all four sensor values simultaneously for a single prediction, but these values might arrive at slightly different times from different feeds.
    - Prediction Trigger: The `if all(feed in sensor_data for feed in AIO_FEED_ID):` condition ensures that the prediction routine only runs once data for all required sensor inputs has been received and buffered. This prevents incomplete predictions.
4. **Prediction Pipeline:**
    - Data Assembly: Combines the buffered sensor data into a Pandas DataFrame in the correct order, matching the model's expected input features.

- Preprocessing: Applies the loaded `scaler.transform()` to scale the input data. This is crucial for models trained on scaled data.
- Inference: `model.predict()` executes the pre-trained neural network on the scaled input to obtain prediction probabilities.
- Result Interpretation: `np.argmax()` and `label_encoder.inverse_transform()` are used to convert probabilities into a final predicted plant health label and its corresponding confidence.

5. **Output and Logging:**
   - Publishing Prediction: `client.publish(AIO_RESULT_FEED, predicted_label)` sends the predicted plant health status to the designated Adafruit IO output feed, making the model's inference available to other services (like a backend module or dashboard).
   - Local Logging: Appends the timestamp, raw input data, and the prediction to a `data_log.csv` file. This provides a local history for debugging, auditing, or further offline analysis.
6. **Continuous Operation:**
   - `client.loop_background()`: Runs the MQTT client in a separate thread, allowing the script to continuously listen for messages without blocking.
   - `while True: time.sleep(1)`: Keeps the main script alive indefinitely.

Overall Function: This script serves as the "AI Module" that orchestrates the flow of sensor data from IoT devices, processes it through a machine learning model, and then publishes the intelligent insights back into the IoT ecosystem. It demonstrates a robust, real-time edge inference setup using MQTT for communication.

9.4.2 Plant Health Classifier

This code is responsible for training and evaluating the machine learning model used in the inference module. It covers the entire lifecycle of developing the plant_health_model.h5, scaler.save, and label_encoder.save files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Step 1: Load and inspect dataset
filtered_df = pd.read_csv("filtered_plant_data.csv")
print("Dataset overview:\n", filtered_df.describe())
```

```
print("\nClass distribution:\n", filtered_df["Plant_Health_Status"].value_counts())

# Step 2: Define features and labels (include Humidity)
X = filtered_df[["Ambient_Temperature", "Soil_Moisture", "Light_Intensity", "Humidity"]]
y = filtered_df["Plant_Health_Status"]

# Step 3: Encode class labels
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
class_names = label_encoder.classes_

# Step 4: Split dataset
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded
)

# Step 5: Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 6: Build the model (input shape changed to 4)
model = Sequential([
    Dense(128, input_shape=(4,), activation='relu'),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(len(class_names), activation='softmax')
])

# Step 7: Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
     metrics=['accuracy'])

# Step 8: Train the model
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history = model.fit(
    X_train_scaled, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)

# Step 9: Evaluate the model
y_pred_prob = model.predict(X_test_scaled)
y_pred_classes = np.argmax(y_pred_prob, axis=1)

accuracy = accuracy_score(y_test, y_pred_classes)
report = classification_report(y_test, y_pred_classes, target_names=class_names)
```

```
print(f"\nAccuracy: {accuracy:.4f}")
print(f"\nClassification Report:\n{report}")

# Step 10: Plot confusion matrix
cm = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
    yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Step 11: Predict function with probability
def predict_plant_health(ambient_temp, soil_moisture, light_intensity, humidity,
    thresholding=True):
    input_df = pd.DataFrame([[ambient_temp, soil_moisture, light_intensity, humidity]],
                            columns=["Ambient_Temperature", "Soil_Moisture",
                                "Light_Intensity", "Humidity"])

    input_scaled = scaler.transform(input_df)
    probs = model.predict(input_scaled)[0]
    predicted_class_idx = np.argmax(probs)
    predicted_label = class_names[predicted_class_idx]
    confidence = probs[predicted_class_idx]

    print(f"\nPrediction confidence scores:")
    for label, prob in zip(class_names, probs):
        print(f"- {label}: {prob:.2%}")

    # Optional rule-based override
    if thresholding and (ambient_temp > 50 or ambient_temp < 0 or soil_moisture < 10):
        print("Unusual input detected. Overriding prediction to 'Unhealthy'.")
        return "Unhealthy"

    return predicted_label

model.save("plant_health_model.h5")

# Save scaler
import joblib
joblib.dump(scaler, "scaler.save")

# Save label encoder
joblib.dump(label_encoder, "label_encoder.save")

print("Model, scaler, and label encoder saved successfully.")
# Example prediction
# result = predict_plant_health(10.0, 60, 200, 55.0)
# print(f"\nPredicted Plant Health Status: {result}")
```

Key Responsibilities and Components:

1. **Data Loading and Initial Exploration:**
    - Dataset Loading: Reads `filtered_plant_data.csv` into a Pandas DataFrame.
    - Basic Inspection: Prints descriptive statistics (`describe()`) and class distribution (`value_counts()`) to understand the dataset's characteristics and potential class imbalances.
2. **Feature Engineering and Label Encoding:**
    - Feature Selection: Defines X (features) as `"Ambient_Temperature"`, `"Soil_Moisture"`, `"Light_Intensity"`, and `"Humidity"`. This explicitly sets the input dimensions for the model.
    - Label Definition: Defines y (labels) as `"Plant_Health_Status"`.
    - Label Encoding: `LabelEncoder` is used to convert categorical plant health status labels (e.g., "Healthy", "Unhealthy") into numerical representations (e.g., 0, 1). This is necessary for neural networks. `class_names` stores the mapping for later interpretation.
3. **Data Splitting and Scaling:**
    - Train/Test Split: `train_test_split` divides the dataset into training (80%) and testing (20%) sets. `stratify=y_encoded` ensures that the proportion of each plant health class is maintained in both training and test sets, which is important for balanced evaluation.
    - Feature Scaling: `StandardScaler` is fitted only on the training data (`fit_transform`) and then used to transform both training and test data (`transform`). This standardizes feature values, preventing features with larger scales from dominating the learning process.
4. **Neural Network Model Definition:**
    - Sequential Model: Uses `tf.keras.models.Sequential` to build a simple feedforward neural network.
    - Layers:
        - `Dense(128, input_shape=(4,), activation='relu')`: The input layer with 128 neurons and ReLU activation, taking 4 features as input.
        - `Dropout(0.3)`: A dropout layer for regularization to prevent overfitting by randomly setting 30% of input units to 0 at each update during training.
        - `Dense(64, activation='relu')`: A hidden layer with 64 neurons and ReLU activation.
        - `Dropout(0.2)`: Another dropout layer.

- `Dense(len(class_names), activation='softmax')`: The output layer with a number of neurons equal to the number of plant health classes. softmax activation is used for multi-class classification, outputting probabilities for each class.

5. **Model Compilation and Training:**
   - Compilation: `model.compile()` configures the model for training:
     - `optimizer='adam'`: A popular and effective optimization algorithm.
     - `loss='sparse_categorical_crossentropy'`: Appropriate loss function for multi-class classification with integer-encoded labels.
     - `metrics=['accuracy']`: Tracks accuracy during training.
   - Early Stopping: EarlyStopping callback monitors `val_loss` and stops training if validation loss doesn't improve for a certain number of epochs `(patience=10)`, restoring the best weights. This prevents overfitting and saves training time.
   - Training: `model.fit()` trains the model on `X_train_scaled` and `y_train`, with a validation split for monitoring performance on unseen data during training.
6. **Model Evaluation:**
   - Prediction: `model.predict()` generates probability predictions on the unseen `X_test_scaled` data.
   - Metrics: Calculates and prints:
     - `accuracy_score`: Overall classification accuracy.
     - `classification_report`: Detailed metrics (precision, recall, f1-score) for each class, providing insights into per-class performance.
   - Confusion Matrix: Generates and plots a confusion matrix using seaborn and matplotlib. This visual tool helps to understand where the model is making errors (e.g., misclassifying "unhealthy" as "healthy").
7. **Model Persistence (Saving):**
   - `model.save("plant_health_model.h5")`: Saves the trained Keras model, including its architecture and learned weights.
   - `joblib.dump(scaler, "scaler.save")`: Saves the fitted StandardScaler object. This is crucial so that new, incoming data can be scaled identically before prediction.
   - `joblib.dump(label_encoder, "label_encoder.save")`: Saves the fitted LabelEncoder. This ensures that the numerical outputs from the model can be correctly mapped back to their original plant health labels during inference.
8. **Example Prediction Function (for testing/demonstration):**
   - `predict_plant_health()`: A utility function included to demonstrate how to use the saved model and scaler for making new predictions.

- Rule-based Override (Thresholding): This function includes a practical rule-based override. If sensor inputs are extreme (e.g., temperature > 50°C or < 0°C, very low soil moisture), it can override the ML model's prediction to "Unhealthy," adding a layer of robustness for outlier conditions.

Overall Function: This script is the "workbench" where the machine learning model is crafted. It takes raw historical data, preprocesses it, trains a neural network, evaluates its performance, and finally saves all the necessary components for deployment in the real-time inference system (the first file).

10 Conclusion

The GreenHouse System represents a significant step toward the realization of intelligent and sustainable agriculture through the integration of cutting-edge technologies. The project has successfully delivered a fully functional prototype that combines real-time sensor monitoring, user interaction, and AI-powered plant health prediction into a cohesive and scalable platform. Each component of the system has been carefully designed and implemented to ensure modularity, maintainability, and real-world applicability.

From a technical perspective, the project demonstrates the effective use of Flask for lightweight server-side processing, ReactJS for building dynamic user interfaces, and TensorFlow/Keras for implementing robust machine learning models. The use of MQTT and Adafruit IO as communication middleware illustrates the practical application of cloud-based services in IoT systems. Additionally, the employment of server-sent events (SSE) ensures low-latency data updates and enhances user experience through near-instantaneous feedback.

Beyond its technical accomplishments, the GreenHouse System has served as an invaluable learning experience for the development team, promoting interdisciplinary collaboration across fields such as software engineering, machine learning, embedded systems, and cloud computing. The rigorous design process, coupled with systematic testing and validation, has reinforced best practices in software development and system integration.

Looking forward, the system offers several avenues for future enhancement, including automated irrigation decision-making based on AI outputs, support for additional sensor types, integration with mobile platforms, and the implementation of advanced analytics for long-term crop health trends. Furthermore, the project can serve as a foundational model for academic research, educational tools, or pilot programs in smart agriculture initiatives.

In conclusion, the GreenHouse System stands as a testament to the practical application of multidisciplinary knowledge in addressing critical challenges in environmental monitoring and agricultural sustainability. The outcomes of this project not only meet the academic objectives of the CO3109 course but also contribute meaningfully to the broader vision of intelligent and eco-friendly agricultural systems.