Sensors Substitution using AI for Agriculture Soil Moisture Monitoring

BY

TAY KAI SHENG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 202

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Dr Teoh Shen Khang who has given me this bright opportunity to engage in this development-based project. It is my first step to establish a career in IT field. Besides that, they have given me a lot of guidance to complete this project. When I was facing problems in this project, the advice from them always assists me in overcoming the problems. A million thanks to you.

ABSTRACT

This project focuses on the growing trend of the Internet of Things (IoT) and Machine Learning (ML) in precision agriculture, specifically sensor substitution using AI for agriculture soil moisture monitoring. Traditional soil moisture sensors face challenges such as environmental degradation and maintenance costs, leading to the need for a more reliable and scalable solution. This project aims to develop an AI-powered soil moisture prediction system that enhances irrigation management by utilizing temperature and humidity data instead of direct soil moisture readings.

The system consists of IoT hardware (ESP32 microcontroller and DHT22 sensor), a cloud-based web application, and a trained machine learning model. The collected sensor data is sent to a real-time monitoring dashboard, where users can view live data trends and change to developer mode when data collection is needed for new plants. The AI model which is trained using ensemble method which contains random forest regressor and gradient boosting regressor to process the collected information to predict soil moisture levels and detect anomalies, providing smart irrigation recommendations.

The key novelty in this project is eliminating the need for direct soil moisture sensors through reliable AI estimation, integration of developer mode triggering for ESP32 via backend control and a modular dashboard design for visualizing data and database integration. The experimental results show promising accuracy in soil moisture predictions and support the efficient irrigation decision-making.

The systems improve the scalability, maintainability and also cost-effectiveness in smart farming, contributing toward AI-driven agriculture and better plant monitoring management.

Area of Study (Maximum 2): Internet of Things, Machine Learning

Keywords (Maximum 5): IoT in data collection, Web monitoring application, Data management in web application, Machine learning in agriculture, AI-based soil moisture prediction.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

β	beta
Ω	Ohm (resistance)

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
GPIO	General Purpose Input Output
HTML	Hyper Text Markup Language
IP	Internet Protocol
IOT	Internet of Things
KNN	K Nearest Neighbors
LSTM	Long Short-Term Memory
MLP	Multi-Layer Perceptron
RAM	Random Access Memory
SVM	Support Vector Machine

Chapter 1
Introduction

In this chapter, the summary of research, covering its history, inspirations, contributions, and thesis statement will be presented. Precision agriculture has benefited greatly from the rapid development of sensor technology, IoT, and data analytics. Accurate soil moisture monitoring is essential to maximize irrigation, conserve water, and increase crop yields. However, environmental factors like rust often cause standard soil moisture sensors to fail, resulting in uneven data collection and high maintenance costs. By leveraging artificial intelligence to predict soil moisture levels based on temperature and humidity data, this work, titled "Sensor Replacement for Agricultural Soil Moisture Monitoring Using Artificial Intelligence," aims to address these issues. By doing so, the need for fragile soil moisture sensors is reduced and provide a more reliable and affordable option for farmers and other agricultural professionals.

1.1 Problem Statement and Motivation

Nowadays, agriculture is becoming one of the most important sectors in the world, and farmers are struggling to optimize irrigation due to faulty soil moisture sensors. Traditional soil moisture sensors can be damaged by the environment which require expensive maintenance, and have accuracy problems like rust, which can result in inaccurate reading [1]. Due to these restrictions, precision agriculture suffers, leading to either over or insufficient irrigation, which has a direct impact on crop productivity and water conservation initiatives [2].

This project seeks to address this problem by developing an AI-based soil moisture prediction system that uses temperature and humidity data rather than direct soil moisture readings. The system can reduce the need for physical sensors and improve irrigation management by accurately predicting soil moisture levels using machine learning algorithms [3]. The goal of the project is to provide farmers and agricultural experts with a reliable, scalable, and affordable alternative to improve crop health, resource efficiency, and sustainability of modern agricultural methods.

1.2 Objectives

In this project, the main goal of this project is to create an Internet of Things (IoT) based soil moisture monitoring system that uses machine learning to predict soil moisture levels based on external variables such as humidity and temperature. The system will collect real-time data using an ESP32 microcontroller, DHT22 sensor. The ESP32 device will support two operation mode which is developer mode for data collection to train new updated AI model as it will trigger the esp32 to collect soil moisture data with the soil moisture sensor connected, another mode which is user mode is used for regular monitoring by collecting only temperature and humidity and the soil moisture data will be predicted after the data is sent to the system.

The second objective is to build a backend server using the Go programming language, serving as the core system for managing data flow between the IoT device, database, and web platform. This server is also integrated with a PostgreSQL database to store sensor readings for historical analysis, logging, and data management to form a foundation for trend observation and system scalability. The backend also includes the machine learning pipeline, where collected developer-mode data is used to retrain and update the AI model. The server will expose RESTful APIs to support functions such as data collection, developer/user mode switching, anomaly detection, and machine learning model interaction. The backend will ensure data integrity, scalability, and efficient communication across system components.

The third objective is to develop a user-friendly, web-based monitoring platform hosted on Firebase. This platform will allow users to visualize live sensor data, track historical trends, and receive notifications when anomalies are detected. By integrating the backend APIs, the dashboard will offer seamless access to all system features and provide real-time insights into environmental conditions for effective soil monitoring.

This project will not cover the development of an automated watering system, integration with commercial smart home platforms, or advanced AI-driven plant health diagnostics beyond soil moisture prediction.

1.3 Project Scope and Direction

This project aims to develop an IoT-based soil moisture monitoring system with machine learning integration to predict soil moisture levels based on temperature and humidity readings. The final deliveries will include both hardware and software components:

- Hardware – An ESP32 microcontroller, DHT22 sensor to collect real-time environmental data for indoor plant monitoring, specifically for Hebe andersonii and snake plant.
- Software – A web-based dashboard that allows users to monitor real-time sensor readings, access historical data, and receive alerts when abnormal conditions are detected. The system will also feature an AI-based soil moisture prediction model, which will analyse collected data and provide insights to improve plant care efficiency.

The project will focus on data collection, visualization, and AI-driven prediction, ensuring a user-friendly and effective solution for plant monitoring.

1.4 Contributions

Our project presents a brand-new approach to soil moisture monitoring by substituting traditional sensors with AI-driven predictions, a concept that holds immense potential for agriculture and plant care. Soil moisture sensors, while widely used, are prone to degradation, corrosion, and inaccuracies over time, leading to high maintenance costs and inefficiencies [4]. With the advancements in AI technology, replacing physical sensors with machine learning models trained on environmental data offers a cost -effective, reliable, and sustainable solution.

This project becomes important in today's agricultural world as farmers and indoor plant enthusiasts are facing challenges with sensor accuracy and operational costs [5]. By using AI models to estimate soil moisture levels based on temperature and humidity data, the problem of relying on hardware components will be reduced, increasing the lifespan of monitoring systems and reducing the frequency of repairs. The system's capacity to produce accurate prediction without direct soil contact distinguishes it as a unique and scalable solution for smart farming, hydroponics, and urban agriculture.

Beyond financial savings, this study supports resource conservation and precision agriculture, therefore guiding irrigation plans and avoiding overwatering. In the future, this AI-driven technology can change agricultural monitoring in every part by making it more accessible, efficient, and environmentally friendly. Once AI improved, sensor substitution may become a normal routine for the next generation of smarter, more flexible, and highly sustainable agricultural systems to show the evolution of AI.

1.5 Report Organization

This report is organized into 7 chapters. Chapter 1 provides introduction of the final year project. Chapter 2 provides a literature review, the content included discussing existing studies on IoT-based soil monitoring systems, the limitations of physical sensors, and the role of AI in predictive analytics for agriculture. After that, Chapter 3 outlines the proposed method which focuses on the development of an AI-based soil moisture prediction model, the integration of IoT for real-time data collection, and the implementation of a web-based monitoring application. Next, Chapter 4 presents the preliminary work which includes initial experiments, model training, and evaluation of AI performance in predicting soil moisture levels. Chapter 5 concludes the study by summarizing the key findings, discussing challenges encountered, and suggesting future research directions for improving AI-driven soil monitoring systems. Chapter 6 focuses on the critical phase of system evaluation, detailing the testing procedures, presenting performance results, reflecting on the challenges overcome, and providing a thorough evaluation against the initial objectives. Finally, Chapter **7** concludes the report by summarizing the project's key findings and achievements and offers a set of forward-looking recommendations for future enhancements that could expand upon the work completed in this project.

Chapter 2
Literature Review

2.1 Review of Technologies

2.1.1 Hardware Platform

The hardware platform chosen for this project is the ESP32 microcontroller, which serves as the core component of the IoT-based soil moisture monitoring system.

Two sensors are connected to the ESP32 which is a DHT22 sensor and a capacitive soil moisture sensor. The DHT22 is a low-cost digital sensor capable of measuring temperature and humidity with a high degree of accuracy and reliability. The soil moisture sensor is selectively activated based on the ESP32's operating mode. In developer mode, the sensor collects real soil moisture data to be used for training and updating the AI prediction model. In user mode, to preserve the sensor and reduce hardware dependency, the ESP32 only collects temperature and humidity data, while soil moisture is predicted via the machine learning model deployed on the backend. All the systems are developed using laptop Legion 5. Table 3.2 shows the specification of the laptop, make sure to meet the requirements for smooth development procedure. Figure 1 below shows the ESP32 attached with sensors.

Table 2.1 Specifications of microcontroller

Description	Specifications
Model	ESP32
Processor	Dual-core Tensilica Xtensa LX6 microprocessor
Memory	520 KB SRAM
Wireless Connection	Wi-Fi 2.4 GHz / Bluetooth v4.2 BR/EDR
Development Support	Arduino IDE

Table 2.2 Specifications of laptop

Description	Specifications
Model	Legion 5 15IAH7H
Processor	Intel Core i7-12700H
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX 3060
Memory	16GB DDR5 RAM
Storage	1TB Seagate HDD

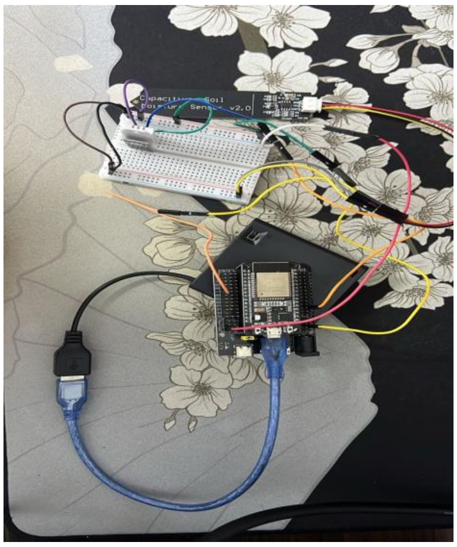


Figure 1: ESP32 attached with sensors

2.1.2 Firmware / OS

The firmware for the ESP32 microcontroller is developed using Arduino framework, which is an open-source platform for programming various types of embedded systems. Ther is a lot open-source libraries used such as WiFi.h, HTTPClient.h, Adafruit_ssensor.h, DHT.h, and ArduinoJson.h. These libraries handle important features like sensor interfacing, data serialization, and HTTP communication to ensure that the transmission of data moves smoothly to the backend server. The reason of choosing Arduino as the framework as its strong community support, various open-source libraries, ease of debugging and compatibility with the ESP32 board.

The development of the entire system including frontend, backend is conducted with VS Code IDE. VS Code IDE supports various code languages and extension which applicable for my backend (Go) and frontend (react, typescript, html). It also supports a Git-based workflow that integrates with GitHub, Google Cloud Platform (GCP), and Firebase through a CI/CD pipeline. This setup ensures the automated deployments, consistent testing, and a streamlined version control across the project components. Arduino is used for firmware development and flashing to the ESP32 board. Figures 2 and 3 below show the icon for both IDE.

Figure 2: Arduino IDE

Figure 3: VS Code IDE

2.1.3 Database

The database component of this project acts as an important role in managing, storing, and retrieving sensor data and system settings for real-time and historical analysis. The chosen database system is PostgreSQL, which is a powerful open-source relational database known for its reliability, scalability and robustness. In this project, the PostgreSQL is hosted on Azure AWS, which offer a better performance, easy integration with backend system and security secure for data in cloud.

The database schema is designed to support multiple features of the system, for example user's sensor data logging, developer mode state for esp32 management, geolocation tracking and user account management. The temperature, humidity and soil moisture that collected by the esp32 device will be stored in the SensorData table, along with metadata like timestamps, user IDs, and anomaly flags. This structure enables trend analysis and supports AI model training through collected environmental data.

Apart from that, the DeveloperModeSetting table is used to manage the activation period of developer mode, which control the esp32 to collect full sensor data (humidity, temperature and soil moisture) for model retraining. This configuration is cached in memory upon application startup and kept synchronized with the database using go mutex locks to ensure thread-safe operation across concurrent API requests.

Additionally, the database includes tables such as DeviceLocation, which logs geolocation data obtained via the Google Geolocation API based on surrounding Wi-Fi access points, and User, which stores user credentials and roles to manage access control within the system. All data interactions are handled using **GORM** (Go ORM), which simplifies object-relational mapping, query abstraction, and schema migration in the Go-based backend server. Figure 4 below shows the logo of PostgreSQL.

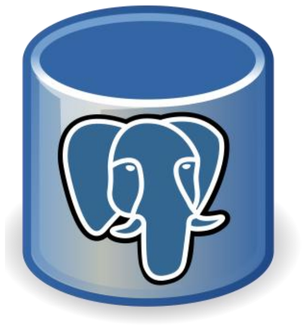


Figure 4: PostgreSQL

2.1.4 Programming Language

This project used multiple language software stack to support various types of components such as embedded systems, backend services, machine learning pipeline and the frontend interface. Each language is selected based on their strengths and compatibility which enhance the quality of the final deliverable systems.

At the embedded system level, the ESP32 microcontroller is programmed using C/C++ within the Arduino framework. Arduino libraries such as WiFi.h, HTTPClient.h, Adafruit_Sensor.h, DHT.h, and ArduinoJson.h are used for handling sensor communication, Wi-Fi connectivity, and JSON formatting. The choice of C/C++ ensures low-level control, efficient memory usage, and real-time responsiveness and these is all IoT devices need. Figure 5 below shows the icon of C/C++.

Figure 5: C/C++

For the backend server, it is developed using Go programming language (Golang) and in form of RESTFULAPI. Go is known for its performance, built-in concurrency support, clean syntax, making it well-suited for building scalable API servers and background services. The servers use the Gin framework to provide features like routing, middleware support, and rendering. The server also uses the GORM library for ORM integration with PostgreSQL and handles tasks such as data storage, anomaly detection and developer mode logic. Figure 6 below shows the icon of Golang.

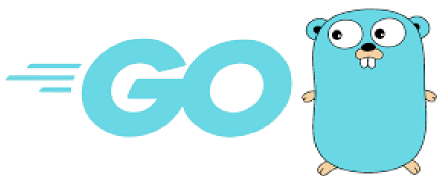


Figure 6: Go Programming Language (Golang)

Apart from that, the frontend is developed using React.js framework with TypeScript. Typescript is a superset of JavaScript, adds static typing which improves code maintainability and reduces runtime errors. React facilitates the development of a responsive web application, allowing real time data visualization, user interaction, and system control through API integrations. Figures 7 and 8 below show the icon of react framework and typescript.

Figure 7: React Framework

Figure 8: TypeScript

2.1.5 Algorithm

The core algorithm used in this project for soil moisture prediction is the Random Forest Regressor and Gradient Boosting Regressor, a robust ensemble learning method widely recognized for its high accuracy and resistance to overfitting, especially in small-to-medium datasets. This model is trained using historical environmental data (temperature, humidity, timestamp features, and calculated soil moisture changes) to predict soil moisture without direct sensor readings in user mode.

The machine learning pipeline is implemented in Python using the Scikit-learn library. The raw dataset, collected during developer mode, includes sensor data with timestamped readings. A series of preprocessing steps are applied to enhance model accuracy:

- Timestamp conversion into numerical format and extracted components such as hour, minute, and day of week
- Soil moisture change rate, calculated using the .diff() method to capture short-term variations

Evaluation metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R^2 Score are used to validate the model's effectiveness. On test data, the model demonstrates strong correlation between predicted and actual soil moisture values.

The trained model and scaler are deployed in a lightweight Flask API server, which dynamically loads models based on the plant name (e.g. *Hebe andersonii*), extracts feature from incoming JSON requests, applies scaling, and returns predicted soil moisture. This modular structure allows plant-specific model updates and seamless integration with the Go backend, which calls the Flask server for real-time predictions.

Overall, this algorithm replaces the need for real-time soil moisture sensors in user mode, reduces hardware reliance, and enables intelligent soil condition monitoring with minimal energy and maintenance requirements.

2.1.6 Summary of the technologies review

The system incorporates different technologies to provide smart agriculture solutions with a reliable and intelligent soil moisture monitoring system. The hardware level uses the ESP32 microcontroller as the main IoT unit together with a DHT22 temperature and humidity sensor as well as a capacitive soil moisture sensor. These sensors operate in two different modes: In developer mode, the ESP32 gathers full environmental data for AI model training. During user mode, only temperature and humidity readings are taken, with soil moisture estimation performed by a machine learning model on the server.

ESP32 firmware is programmed in C/C++ using the Arduino framework, which includes standard libraries like WiFi.h, HTTPClient.h, and ArduinoJson.h. The development environment is based on Visual Studio Code (VS Code), which has support for backend development in Go, frontend work in React and TypeScript, as well as for GitHub, Firebase and Google Cloud Platform (GCP) for version control and CI/CD deployments.

The backend server is written in the Go programming language with the Gin web framework and GORM supporting ORM-based operations against a PostgreSQL database on Azure AWS. It handles API endpoints to log sensor data, switch developer mode, identify anomalies, and talk to the AI model.

The machine learning model, having been executed using Python and Scikit-learn, employs a Random Forest Regressor that predicts soil moisture from environmental parameters. Preprocessing includes parsing the timestamp, identification of changes in the soil moisture, and feature scaling using StandardScaler. The model and scaler are deployed on a Flask server and are dynamically selected depending on plant type (e.g., Hebe andersonii).

The frontend, developed in React and TypeScript, is the graphical user interface that offers real-time monitoring, historical visualization, and system control. It is deployed on Firebase, leveraging the backend APIs to fetch live updates and communicate with.

They collectively create a robust, modular, and intelligent system that can reduce reliance on physical sensors while maintaining high accuracy for soil moisture measurement via AI prediction.

2.2 Review of Existing Systems

2.2.1 ARIMA-Based Sensor Fault Tolerance in IoT Monitoring

The challenge of sensor reliability in remote monitoring applications has been already studied in previous research. The advancement of low-power sensor technology and energy harvesting techniques has been a primary focus to improve sensor node reliability and lifespan. According to previous work, various fault-tolerant mechanisms and robust hardware designs have been proposed to enhance the durability of sensor nodes, aiming to reduce failures and operational disruptions. Moreover, data analytics techniques such as autoregressive integrated moving average (ARIMA) models have been explored to improve data reliability by predicting and mitigating sensor failures [6]. Figure 9 below shows the flowchart of the estimation process.

Complementing these techniques, cloud computing architectures have enabled more advanced fault detection strategies. By correlating data streams from multiple sensor nodes, cloud-based systems can validate and substitute sensor readings in real time, thus increasing system resilience and enhancing data integrity. However, these systems often rely on the availability of multiple redundant nodes, which may not be feasible for small-scale or cost-constrained deployments. Figure 9 below shows the flowchart of the estimation process.

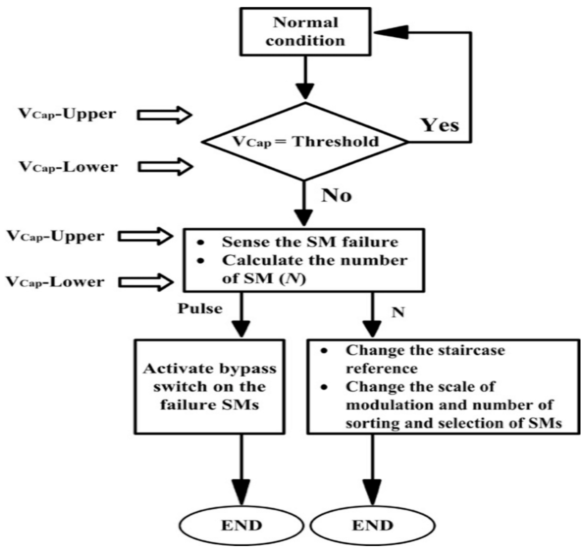

Normal condition
VCap-Upper
VCap-Lower
VCap = Threshold
Yes
No
VCap-Upper
VCap-Lower
Sense the SM failure
Calculate the number of SM (N)
Pulse
N
Activate bypass switch on the failure SMs
Change the staircase reference
Change the scale of modulation and number of sorting and selection of SMs
END
END

Figure 9: Flowchart of The Estimation Process (ARIMA)

Additionally, cloud computing has played a significant role in enhancing sensor availability and data accuracy. Researchers have investigated correlations between data from multiple sensor nodes to detect anomalies and compensate for sensor failures. The integration of cloud-based data processing with IoT sensor networks has enabled real-time decision-making and improved the robustness of remote monitoring systems.

2.2.2 SmartFarm AI - IoT-Based Smart Irrigation System

The integration of IoT in agriculture has given rise to smart irrigation platforms such as SmartFarm AI, which combine sensor technology and rule-based logic for automated irrigation scheduling. These systems leverage real-time monitoring and data-driven decision-making to improve water efficiency, reduce labor costs, and enhance crop yields [7]. For instance, some studies have introduced IoT-driven irrigation scheduling systems that regulate watering based on real-time soil moisture data, significantly reducing water wastage while improving crop productivity. SmartFarm AI systems are effective in reducing water usage and increasing crop yield. They often include mobile or web interfaces to allow farmers to remotely monitor soil conditions and configure thresholds. However, they heavily depend on the continuous operation of soil moisture sensors, which may degrade over time due to corrosion, salinity buildup, or harsh environmental exposure. Figure 10 below shows the diagram of IoT and AI in agriculture.

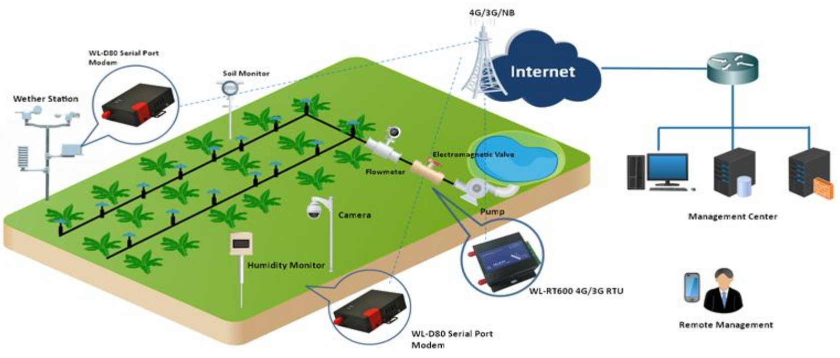


Figure 10: Smart Irrigation System Diagram

2.2.3 AI-Driven Soil Moisture Prediction Systems

Machine learning techniques have been widely employed to predict soil moisture levels using environmental parameters such as temperature and humidity. Research has demonstrated that models like Random Forest, Decision Tree, Support Vector Machine (SVM), k-Nearest Neighbours (KNN), and Naïve Bayes can effectively predict soil moisture using sensor data [8]. Furthermore, studies have explored the potential of transfer learning to improve soil moisture prediction accuracy. By fine-tuning pre-trained models with localized sensor data, researchers have achieved better predictive performance in regions with limited historical data [9]. Figure 11, 12, 13, 14 will show different types of AI model diagrams.

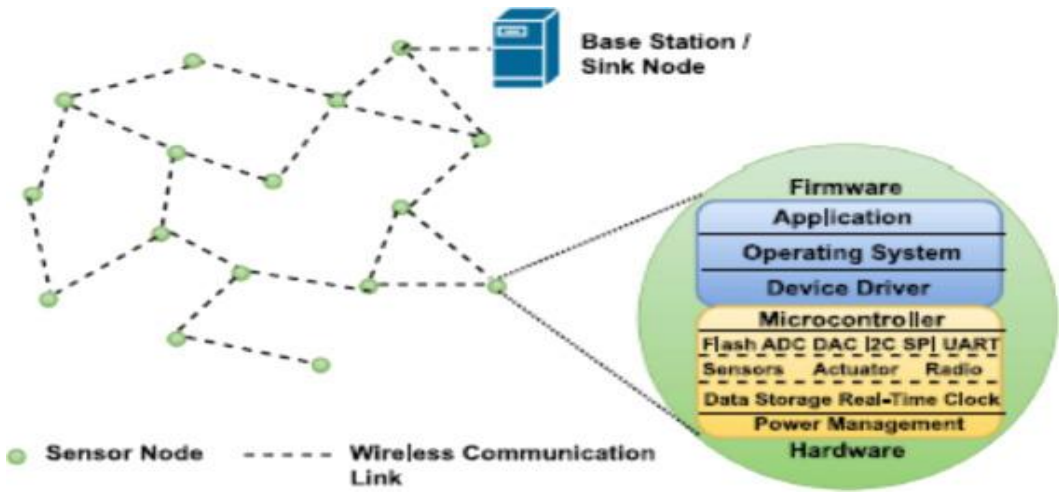


Figure 11: Diagram of Sensor Network

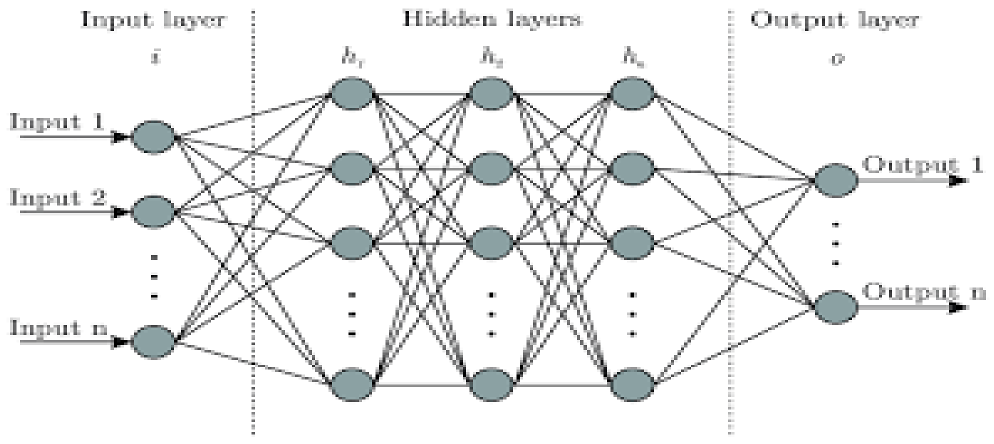


Figure 12: Neural Network Architecture

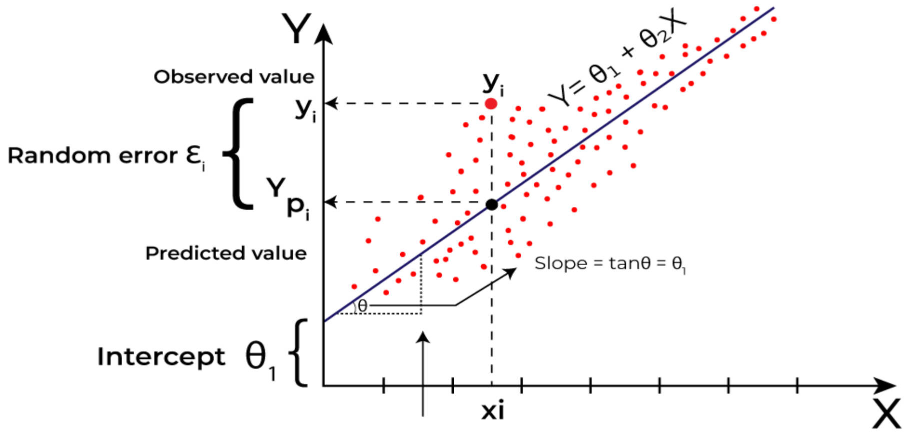

Y
Observed value
y_i
Random error ε_i
Y_{p_i}
Predicted value
Intercept θ_1
$Y = \theta_1 + \theta_2 X$
Slope = tanθ = θ_1
θ
xi
X

Figure 13: Linear Regression

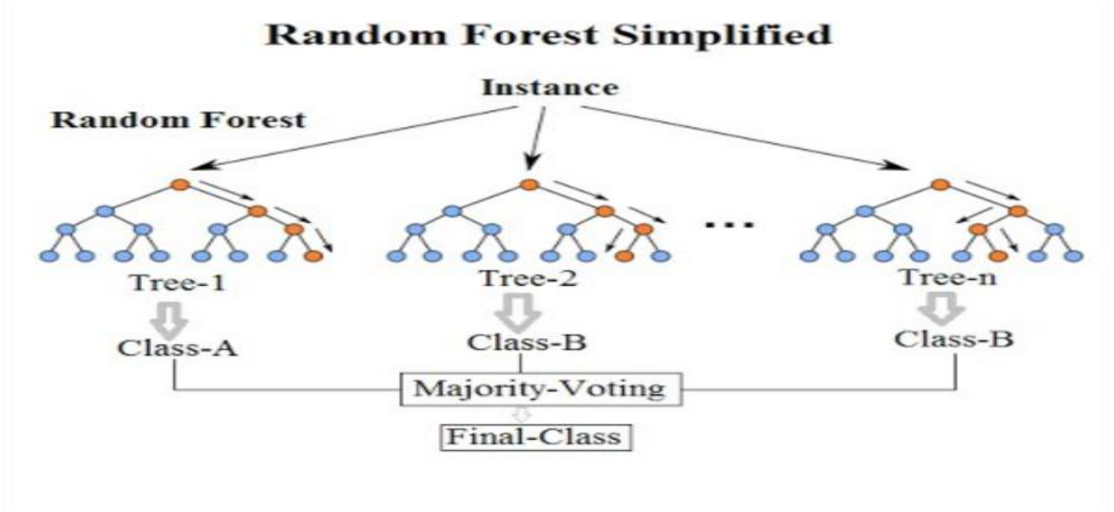


Figure 14: Random Forest

Despite these advancements, challenges such as sensor reliability, model generalization, and data dependence persist, necessitating further research and innovation. AI integration in agriculture, particularly through neural networks like Multi-Layer Perceptron (MLP), has shown promising results in handling complex classification tasks [10]. For instance, hybrid models combining MLP with optimization techniques, such as the firefly algorithm, have demonstrated superior accuracy in predicting soil moisture levels compared to traditional methods [11]. Besides, many AI-driven soil monitoring solutions still rely heavily on continuous sensor data. The failure or inaccuracy of a single sensor can affect the entire system's reliability. This project aims to overcome this challenge by developing an AI-based substitution model, allowing temperature and humidity sensors to estimate soil moisture levels even when soil moisture sensors fail. Figure 15 below shows the relationship between soil moisture, temperature and humidity. This proved that there is a relationship between them.

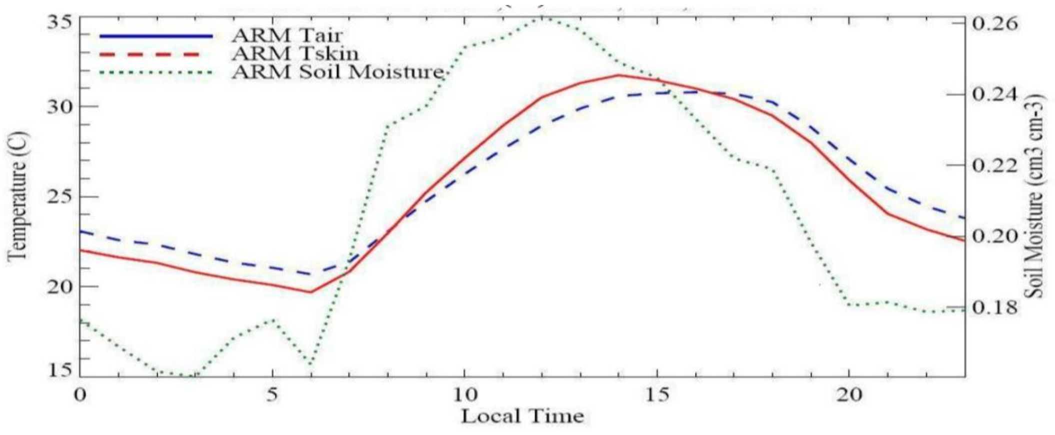


Figure 15: Relationship soil moisture (green) between temperature (blue) and humidity (red) [25]

2.2.4 AI-Driven Soil Moisture Prediction Systems

Another significant advancement in soil monitoring systems is the use of wideband radar technology combined with machine learning models to estimate soil moisture in a non-invasive manner. In the study [12], researchers developed a system that leverages wideband radar to capture electromagnetic reflections from the soil, which are then analyzed using predictive models such as Partial Least Squares (PLS) regression and Random Forest. This approach eliminates the need for traditional in-soil moisture probes, reducing hardware degradation and maintenance requirements.

The system achieved high accuracy, with coefficients of determination (R^2) approaching 0.89 across various soil conditions, demonstrating strong generalization. While the hardware involved is more complex and costly compared to low-cost IoT sensors, this solution highlights the growing trend of sensor substitution using AI and signal processing techniques. It aligns with the objective of the current project, which aims to estimate soil moisture using environmental parameters such as temperature and humidity, thus minimizing reliance on physical moisture sensors. Although this radar-based method relies on specialized equipment, both systems share the goal of improving soil moisture estimation through non-traditional, intelligent sensing strategies that increase system reliability and scalability in agricultural monitoring. Figure 16 below shows the experimental setup.

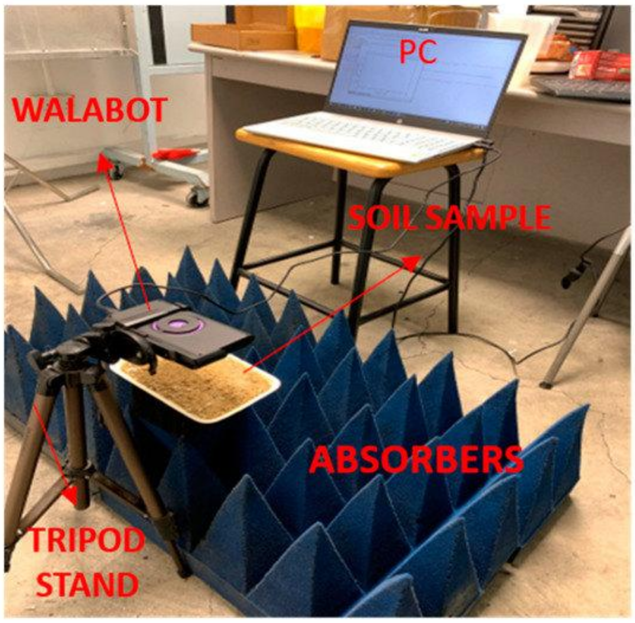


Figure 16: Experimental Setup for Experimental Setup for AI-Driven Soil Moisture Prediction Systems

2.2.5 RF Backscatter Tag for Low-Cost Soil Moisture Sensing

Previous work about RF Soil Moisture Sensing System using battery-powered RF backscatter tags and ultra-wideband transceivers to non-invasively measure soil moisture [13]. These passive tags, deployed at depths of up to 75 cm, communicate via backscatter signals and achieve accuracy within 0.01–0.03 cm^3/cm^3 of ground truth comparable to commercial sensors while offering projected battery lifetimes up to 15 years. This system addresses common issues in soil sensing such as sensor maintenance, power supply, and deployment costs. Its minimalist hardware design and longevity make it ideal for scalable agricultural monitoring. Although the sensing modality differs (RF vs. environmental sensors), the goal to accurately estimate soil moisture while minimizing hardware dependency closely parallels the aim of using AI-based substitution with minimal physical soil sensors.

2.2.6 Wi-Fi Chipless Tag Soil Moisture Detection

In another low-cost, wireless sensing solution SoilTAG is developed to detect soil moisture levels using battery-Free Wi-Fi tag [14]. This system uses chipless passive Wi-Fi tags embedded in soil that alter their reflection signature based on moisture content. The tags generate signal variations detectable up to 13 m away via standard Wi-Fi readers. SoilTAG achieves approximately 2 percentage-point resolution at close distance and about 5 percentage-point accuracy at longer ranges. This non-invasive design eliminates the need for onboard power or soil probes, prioritizing maintenance-free operation.

2.2.7 Summary of the Existing System

The review of existing systems highlights diverse approaches to addressing challenges in agricultural monitoring, particularly soil moisture estimation and sensor reliability. For the first system, which employs ARIMA-based fault tolerance, focuses on enhancing data integrity by predicting and correcting sensor failures through statistical modeling and cloud-based validation. While effective in improving robustness, it often relies on sensor redundancy, which may not suit small-scale deployments.

Second, the SmartFarm AI, demonstrates the benefits of real-time monitoring and automation in smart irrigation. It combines soil moisture sensors and rule-based decision systems to optimize water usage. However, its heavy reliance on continuous sensor functionality poses a limitation in harsh or long-term environments.

Third system explores AI-driven soil moisture prediction models using environmental variables like temperature and humidity. Machine learning techniques such as Random Forest, SVM, and hybrid neural networks have shown promising accuracy, especially when enhanced through transfer learning. Nonetheless, these models still face challenges with generalization and sensor dependency for initial data.

The fourth introduces a non-invasive radar-based sensing system combined with machine learning. This method offers high accuracy in soil moisture prediction without embedding sensors into the soil, thus extending hardware lifespan. However, its cost and complexity limit its accessibility compared to simpler IoT solutions.

The fifth and sixth system demonstrate passive, non-invasive technologies that achieve soil moisture sensing without embedded probes, significantly reducing hardware degradation and maintenance. They reinforce the concept of sensor substitution whether via RF backscatter or Wi-Fi reflections—aligning with your project's strategy of using ML with auxiliary environmental data for prediction. While their technologies differ, they validate the broader approach of trading hardware complexity for intelligent signal-based inference, emphasizing reliability, low power usage, and long-term deployment.

Overall, these systems collectively emphasize the importance of data-driven approaches, sensor resilience, and the growing role of AI in precision agriculture. The current project aligns with these trends by proposing an AI-based substitution model that estimates soil moisture using temperature and humidity data, offering a cost-effective, scalable, and sensor-resilient solution.

Chapter 3

System Methodology/Approach

This chapter explains the system development methodology and presents the design diagrams that illustrate the structure and workflow of the project. The system is designed to substitute traditional soil moisture sensors using AI predictions based on temperature and humidity data collected by the DHT22 sensor, integrated with ESP32. A web-based dashboard and cloud backend are used for real-time monitoring and AI integration.

3.1 System Design Diagram/Equation

The overall system design integrates IoT hardware, a cloud-based backend, an AI prediction model, and a web-based frontend interface. It consists of three main layers:

- Hardware Layer: ESP32 with DHT22 and Soil Moisture Sensor
- Backend Layer: Golang server with PostgreSQL and Flask-based AI model
- Frontend Layer: React.ts dashboard for real-time monitoring

3.1.1 System Architecture Diagram

Figure 17 below shows the interaction between hardware, backend, AI model, and front end:

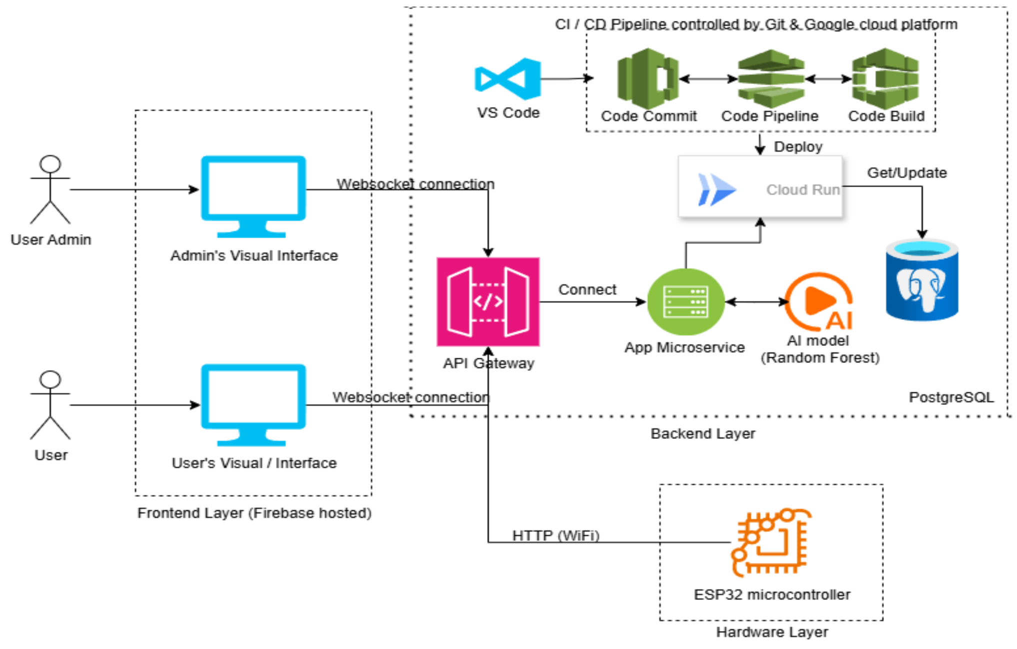


Figure 17: System Architecture Diagram showing interaction between hardware (ESP32), backend microservice (Go & Flask), and frontend (Firebase-hosted React app), including CI/CD integration.

This system architecture diagram describes a full three-tier application-based IoT solution capable of real-time data processing and decision-making. The front-end layer is essentially a web application hosted on Firebase that provides an interface for the admin user and an interface for the end user; with data transfer using WebSocket connections, this allows for real-time execution and communication with the end user. The back end of the solution is based on Google Cloud Platform and features an API Gateway that manages messages between the application microservices, communicates predictions from an AI model (using the Random Forest algorithm), and uses a PostgreSQL database for persistent data storage. The hardware layer essentially consists of ESP32 microcontrollers operating as edge computing nodes that communicate both ways with the cloud services using HTTP/Wi-Fi protocols, although it could just as easily implement using other edge devices and cloud services so long as it adheres to the same method of data transfer.

Finally, throughout all three layers of this application architecture exists a robust CI/CD process that is integrated with both Git to control versions and Google Cloud Platform to automate code deployment and operations, with continuous integration and operate at scalable level on cloud services. The architecture presented is a current representation of the methodology to IoT system deployment; it has unrestricted capabilities leveraging data from edge computing on a cloud-based intelligent AI system, and automated DevOps practices for an enterprise-level, scalable, smart and maintainable solution.

3.1.2 Use Case Diagram and Description

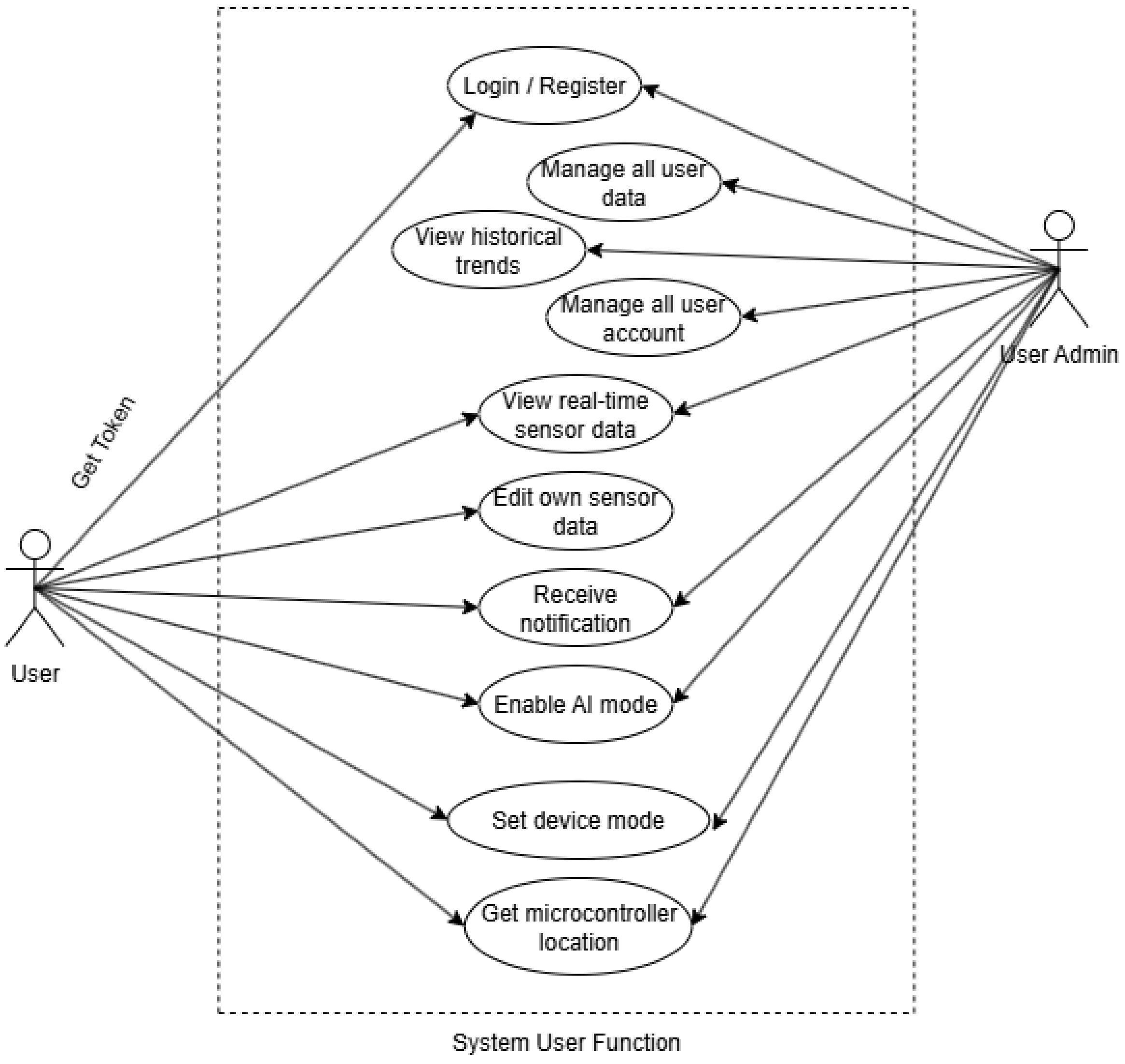


Figure 18: Use Case Diagram illustrating interactions between human users (Admin and User) and system components (ESP32 microcontroller).

Table 3.1: Use Case Descriptions for User, Admin, and ESP32 System Functions.

Use Case	**Actor(s)**	**Description**
Login / Register	User, admin	Authenticates the entity (human or device) to access the system with a valid token.
View real-time sensor data	User, admin	Display current sensor readings via WebSocket.
View historical trends	User, admin	Shows past data in a chart / graph format for analysis during developer mode.
Enable AI mode	User, admin	Turns on machine learning-based prediction when soil moisture prediction is needed.
Set device mode	User, admin	Switches esp32 between developer and user modes remotely.
Get microcontroller location	User, admin	Fetches the device's estimated location, either manually set or system estimated.
Receive notification	User, admin	Receive real-time alerts for abnormal conditions or AI-mode activation.
Edit own sensor data	User	Allow editing or removing previously submitted data.
Manage all user data	Admin	Grants access all users' sensor records for moderation or correction or deletion.
Manage all user account	Admin	Admin can update, activate, or delete user accounts.

3.1.3 Activity Diagram

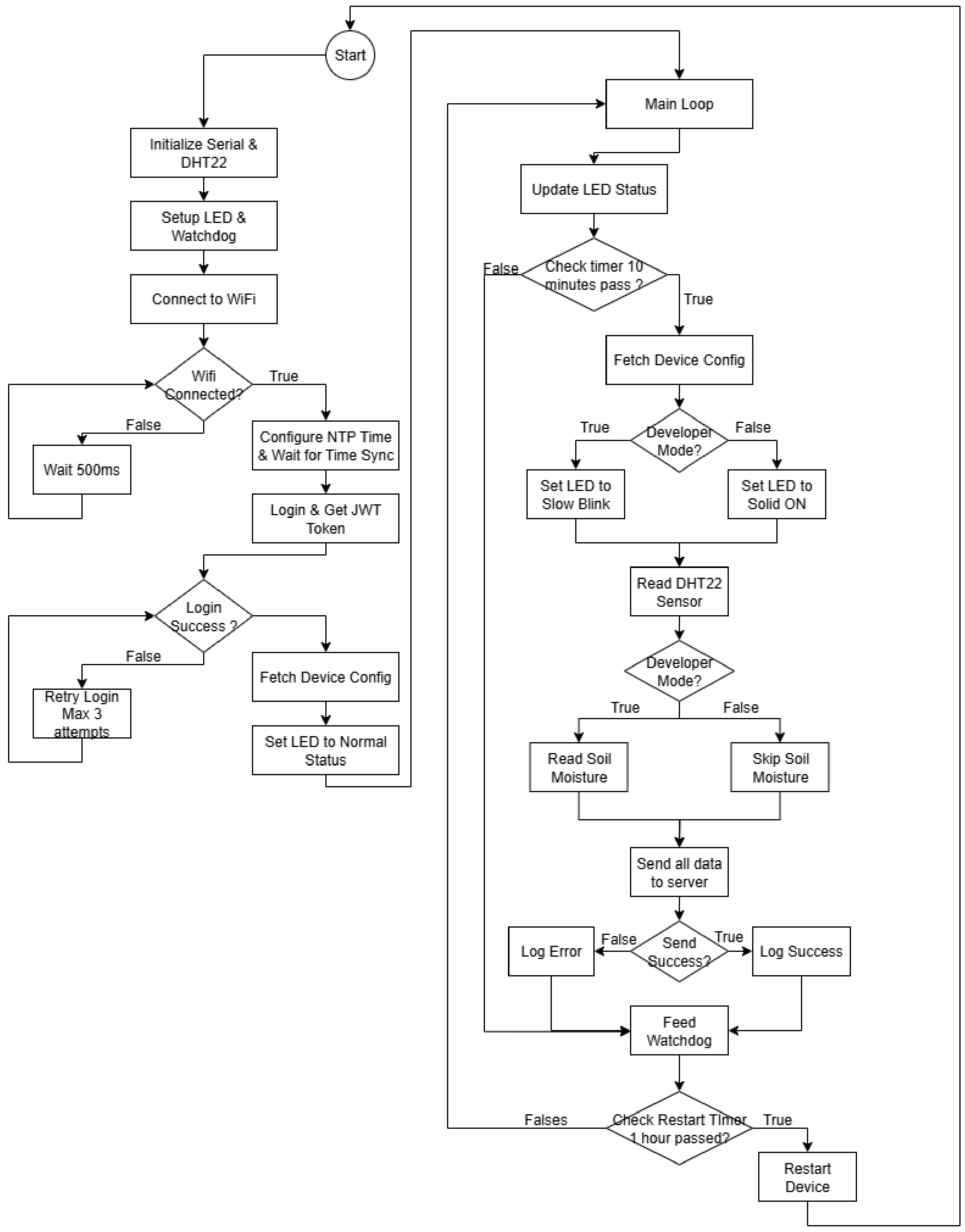


Figure 19: Activity diagram for ESP32 microcontroller

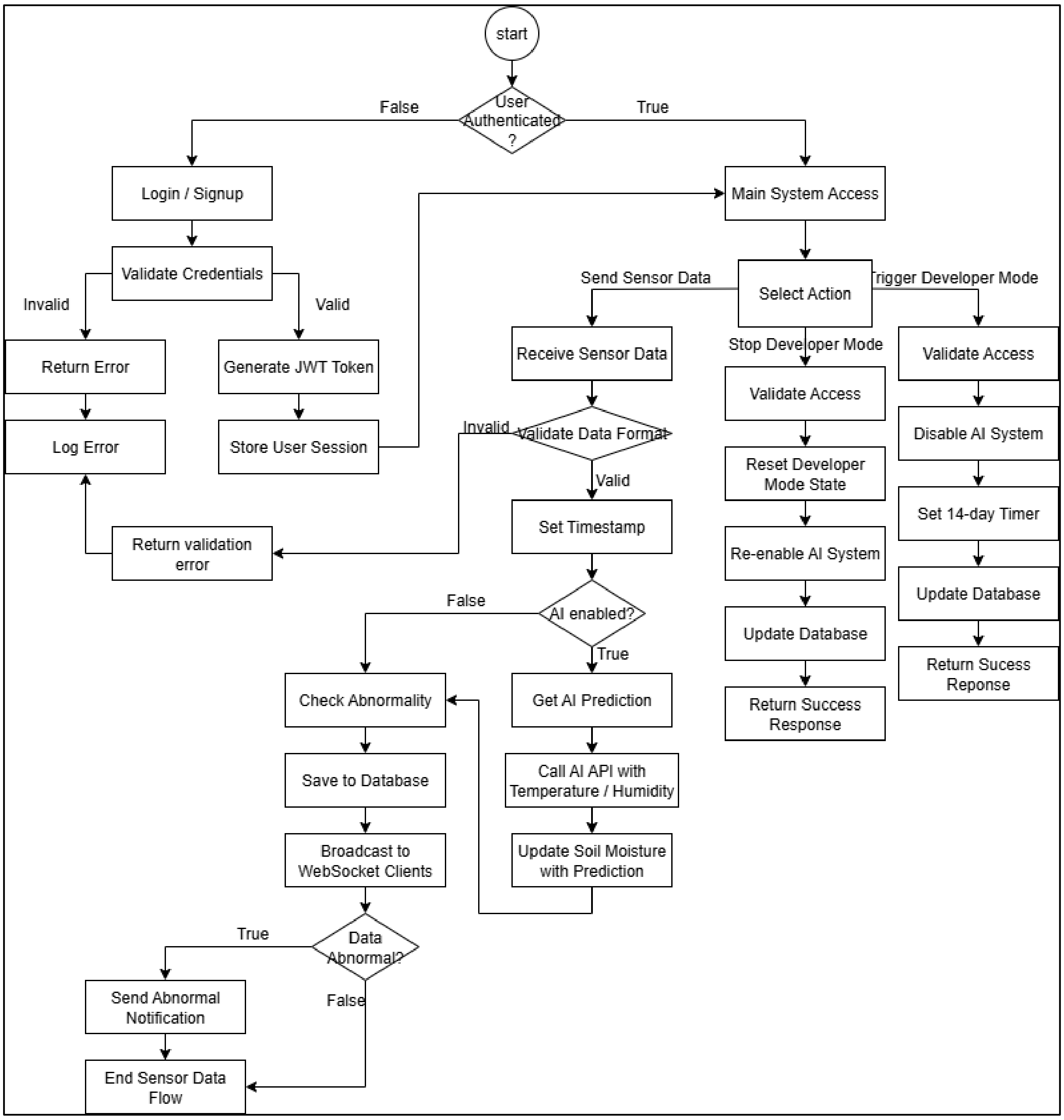


Figure 20: Activity diagram for backend part 1

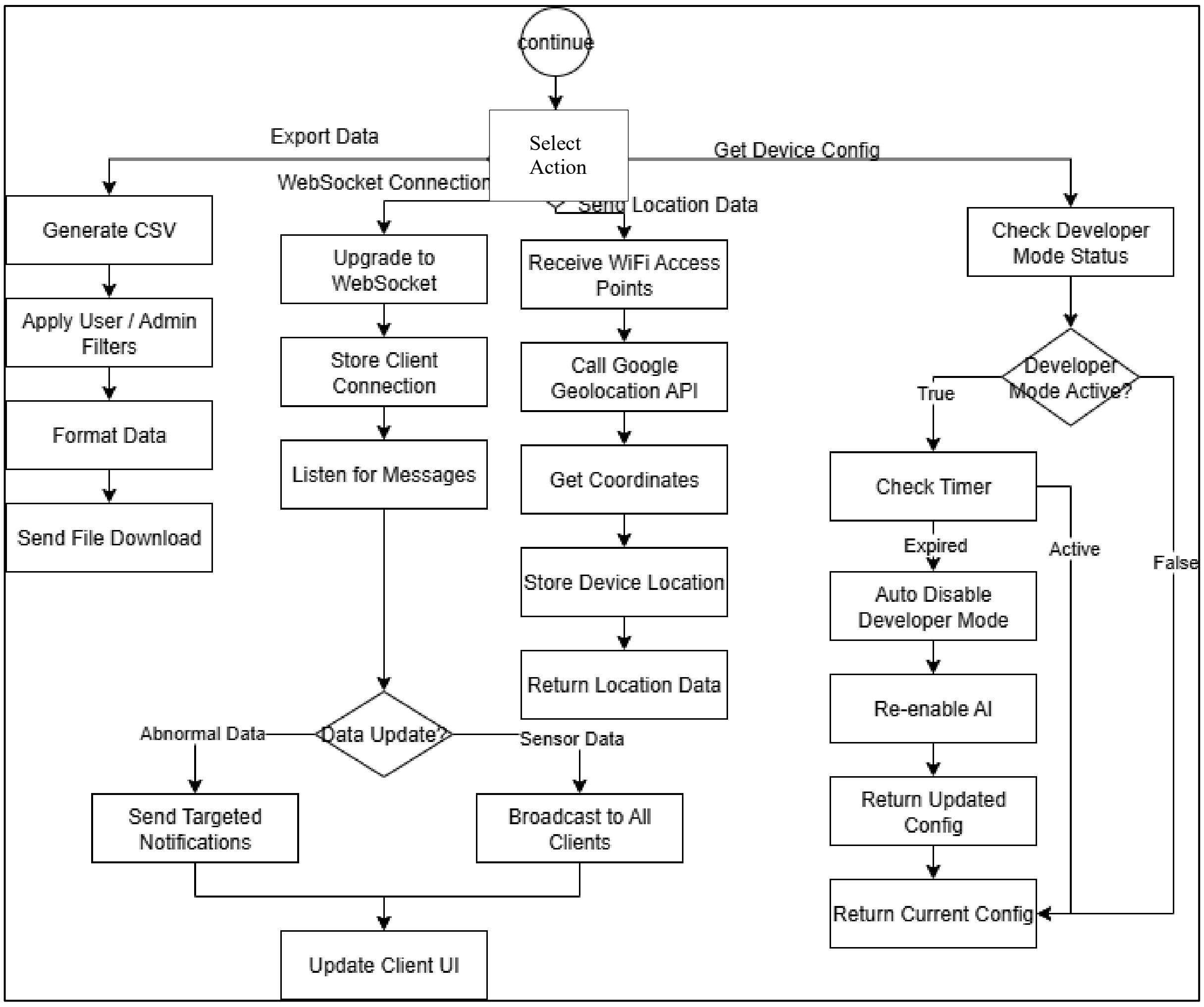


Figure 21: Activity diagram for backend part 2

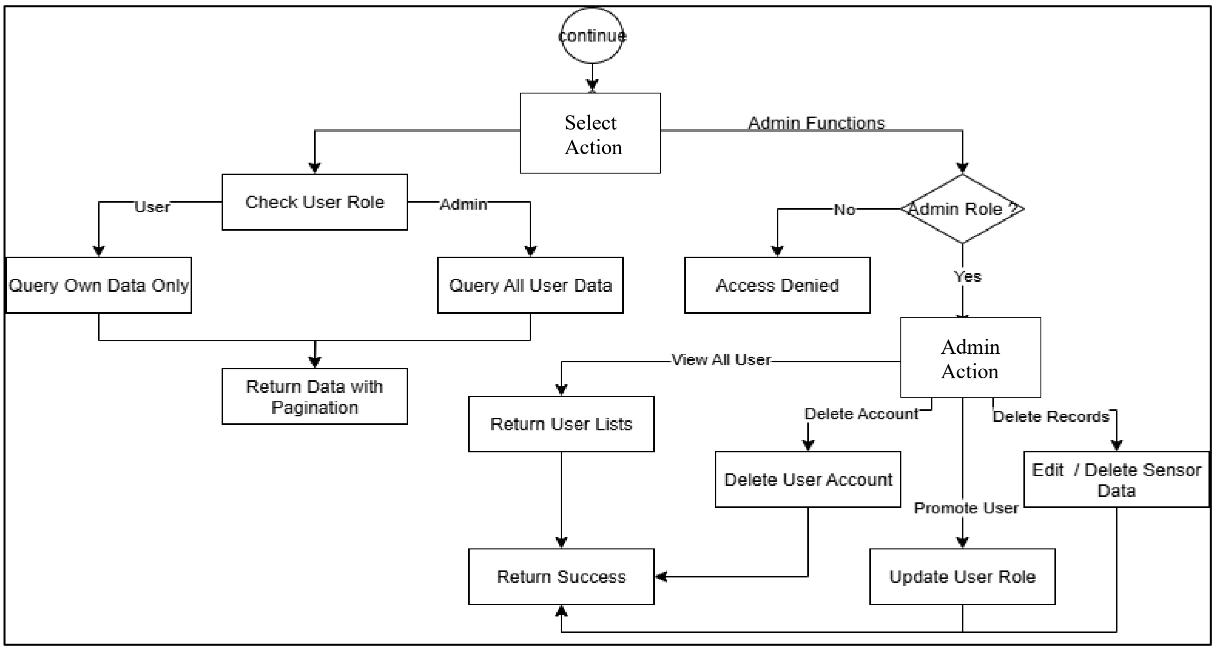


Figure 22: Activity diagram for backend part 3

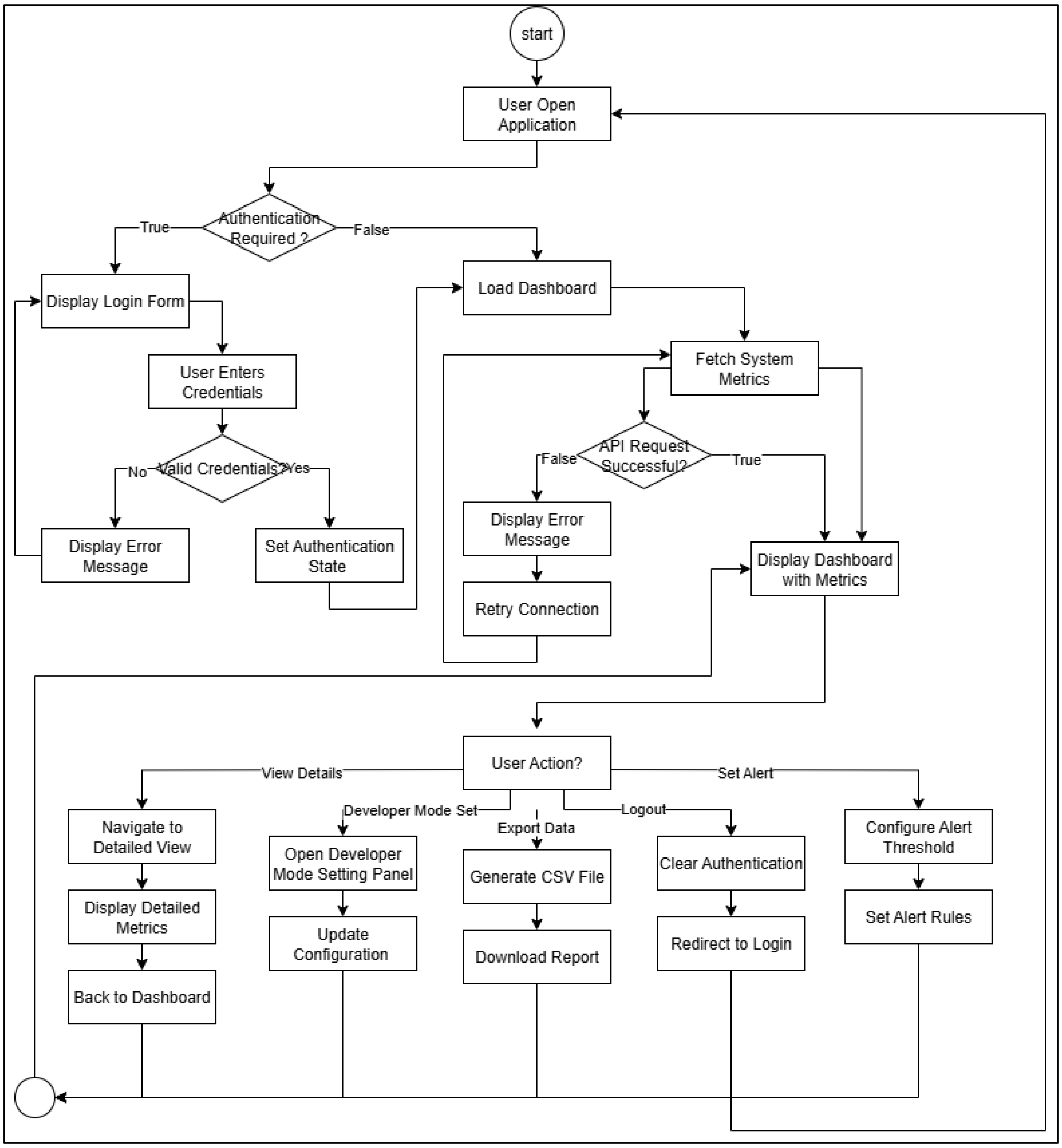


Figure 23: Activity diagram for web application

Chapter 4
System Design

4.1 System Block Diagram

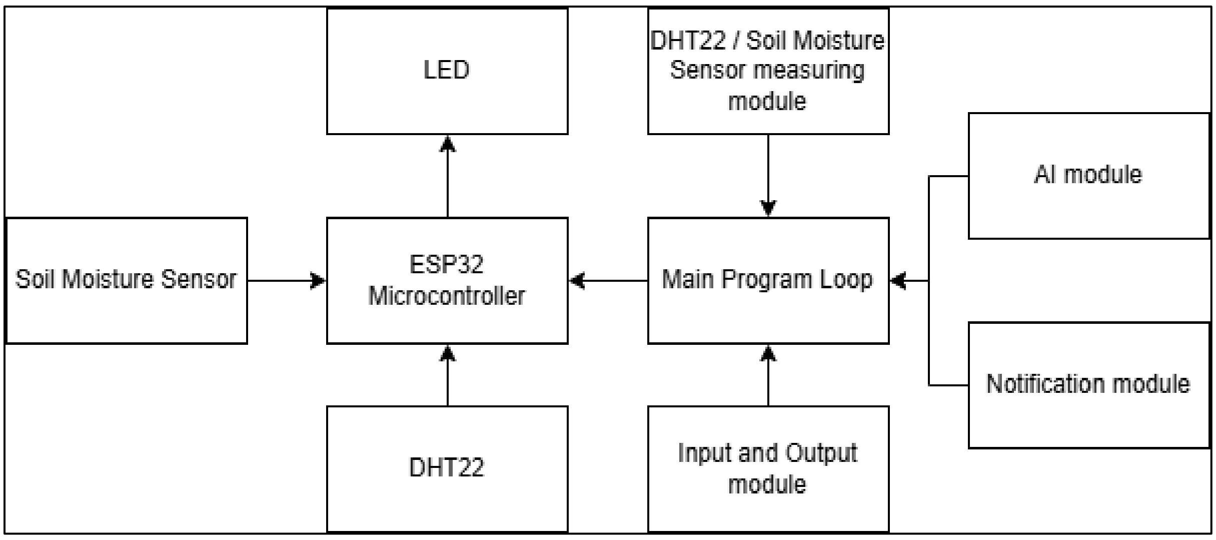


Figure 24: System Block Diagram

The system block diagram consists of an ESP32 microcontroller serving as the central processing unit that coordinates all system components. The ESP32 interfaces with multiple sensor modules including a soil moisture sensor and a DHT22 temperature/humidity sensor, which provide environmental monitoring capabilities.

The main program loop acts as the system's control centre, receiving data from the DHT22/Soil Moisture Sensor measuring module and managing the overall system operation. This central loop coordinates with several key modules: an Input and Output module that handles system interactions, an AI module for intelligent data processing and decision-making, and a Notification module for user alerts and communication.

Visual feedback is provided through an LED indicator connected directly to the ESP32, allowing for immediate status indication. The modular design ensures efficient data flow between components, with the ESP32 microcontroller managing sensor data acquisition, processing through the AI module, and triggering appropriate responses via the notification system.

This architecture enables real-time environmental monitoring with intelligent analysis capabilities, making it suitable for applications such as smart agriculture, greenhouse automation, or environmental monitoring systems where soil moisture and atmospheric conditions need to be continuously tracked and managed.

4.2 System Components Specifications

4.2.1 IoT Hardware Layer

1. ESP32 microcontroller
    - Model: ESP32-WROOM-32
    - Function: WiFi-enabled MCU to read sensor data and transmit it to the backend server.
    - Programming: Arduino IDE using C/C++

Table 4.2 Specifications of microcontroller

Description	Specifications
Model	ESP32
Processor	Dual-core Tensilica Xtensa LX6 microprocessor
Memory	520 KB SRAM
Wireless Connection	Wi-Fi 2.4 GHz / Bluetooth v4.2 BR/EDR
Development Support	Arduino IDE

2. DHT22 sensor
    - Model: DHT22 (AM2302)
    - Sensor type: Digital temperature and humidity sensor
    - Interface: Single-wire digital interface
    - Pin assignment: GPIO 4
    - Voltage: 3.3V – 6.0V
3. Soil moisture sensor
    - Type: Capacitive/Resistive analog sensor
    - Interface: Analog input (ADC)
    - Pin assignment: GPIO 32 (12-bit ADC)
    - Operating voltage: 3.3V - 5V
    - Activation: Developer mode only
4. Status indicators
    - LED: Built-in LED (GPIO 2)
    - Serial monitor: 115200 baud rate debugging

- Status modes:
    - Fast blink (200ms): Connecting
    - Slow blink (1000ms): Developer Mode
    - Solid ON: Normal Operation

4.2.2 Backend Layer

1. Hardware infrastructure
    - Platform: Google Cloud Run (serverless)
    - CPU: Auto-scaling
    - Network: Global load balancing
2. Software stack
    - Programming language: Go (Golang)
    - Framework: Gin web framework
    - Version: Go 1.19
    - Concurrency: Goroutines for high-performance handling
    - HTTP server: HTTP/2 support
    - WebSocket support: Real-time bidirectional communication
3. Database specifications
    - Database: PostgreSQL
    - Version: 14
    - Hosting: Microsoft Azure SQL Database
    - Backup: Automated daily backups
    - Performance: Optimized for time-series data
4. API endpoints
    - Public routes (No Authentication Required)
        - User registration: POST /signup - Create new user account
        - User authentication: POST /login - JWT token generation
        - AI toggle: POST /toggle-ai - Enable/disable AI predictions globally
    - Protected routes (Requires JWT Authentication)
        - ❖ Real-time connection: GET /ws - WebSocket endpoint for live data streaming
        - ❖ Promote admin: POST /promote-admin - Elevate user to admin role

- ❖ Promote user: POST /promote-user - Change admin back to user role
- ❖ Get users: GET /users - Retrieve all registered users
- ❖ Get profile: GET /profile - Get current user profile information
- ❖ Delete user account: DELETE /admin/delete-user/:user_id - Admin delete user
- ❖ Data ingestion: POST /sensor-data - Receive sensor data from ESP32
- ❖ Get history: GET /history - Retrieve historical sensor data
- ❖ Update record: PUT /update/:id - Modify specific sensor record
- ❖ Delete record: DELETE /delete/:id - Remove specific sensor record
- ❖ Delete all records: DELETE /delete/all - Remove all sensor data
- ❖ Delete user records: DELETE /delete/my-records - Remove current user's records
- ❖ Delete specific user records: DELETE /delete/user/:user_id - Admin remove user's data
- ❖ Download CSV: GET /download-csv - Export sensor data as CSV file
- ❖ Get device config: GET /device-config/esp32-001 - Retrieve device settings
- ❖ Stop developer mode: POST /device-config/esp32-001/stop-dev - Disable soil sensor
- ❖ Trigger developer mode: POST /device-config/esp32-001/trigger-dev - Enable soil sensor
- ❖ Set location: POST /location - Receive GPS coordinates from ESP32
- ❖ Get location: GET /get-location/:device_id - Retrieve device location
- ❖ Abnormal count: GET /abnormal-count - Get count of anomalous readings
- ❖ Abnormal history: GET /abnormal-history - Retrieve anomalous data records
- ❖ Train AI model: POST /train-model – Send CSV data and plant_name to flash server for ai training.
- ❖ Get available AI model: GET /models – Fetch available AI model from flask server

- Technical specifications
  - ❖ Protocol: HTTPS with TLS 1.3
  - ❖ Authentication: JWT bearer token (except public routes)
  - ❖ Rate limiting: 100 requests/minute per device

- ❖ Data format: JSON request/response
- ❖ CORS: Configured for cross-origin requests
- ❖ Middleware: Authentication middleware for protected routes

4.2.3 Frontend Layer

1. Technology stack
    - Framework: React.js 18
    - Language: TypeScript
    - State management: React hooks (usestate, useeffect)
    - HTTP client: Fetch API
    - Real-time: WebSocket client
    - Authentication: JWT token handling
2. UI components
    - Dashboard: Real-time sensor data visualization
    - Data management: Configuration interface
    - Responsive design: Mobile and desktop compatible
    - Notifications: Real-time alerts and status updates
3. Performance specifications
    - Load time: <3 seconds initial loads
    - Real-time updates: <1 second latency (WebSocket)
    - Browser support: Chrome, Firefox, Safari, Edge
    - Mobile support: IOS Safari, Chrome Mobile

4.2.4 AI Model Layer

1. Machine learning stack
    - Framework: Python flask
    - ML library: Random Forest / Gradient boosting
    - Python version: 3.10+
    - Dependencies: scikit-learn, pandas, numpy
2. Model specifications
    - Input features: Temperature, humidity, timestamp, plant_type

- Output: Predicted Soil moisture percentage
- Training data: Historical sensor correlations
- Accuracy: 85%+ prediction accuracy
- Response time: <200ms per prediction
- Hosting: Cloud Run (docker Implemented)

3. API specifications
    - Predict soil moisture: /predict – Use temperature, humidity to predict soil moisture
    - Train AI: /train-model – Post CSV file and plant_name to train the ai model the specific plant_name.
    - Get models: /models – Get the available model in the python flask server and checks its results.

4.2.5 Communication Protocols

1. Device-to-Server communication
    - Protocol: HTTPS (TLS 1.3)
    - Authentication: JWT bearer tokens
    - Data format: JSON
    - Compression: gzip encoding
    - Timeout: 30 seconds per request
    - Retry logic: Exponential backoff (3 attempts)
2. Real-time communication
    - Protocol: WebSocket
    - Heartbeat: 30-second ping/pong
    - Message format: JSON
    - Broadcast: Server-to-multiple clients
    - Fallback: HTTP polling method used if WebSocket fails

4.2.6 System Requirements

1. Power requirement
    - ESP32: 3.3V @ 240mA peak
    - DHT22: 3.3V @ 2.5mA max

- Soil sensor: 3.3V @ 35mA typical (analog)
- Total power: <5W during operation

2. Network requirement
    - Bandwidth: 1KB / 10min per device (minimal)
    - Latency: <500ms for real-time updates
    - Connectivity: 2.4GHz WiFi
    - Range: 50 – 100m depending on environment

4.3 Circuit and Components Design

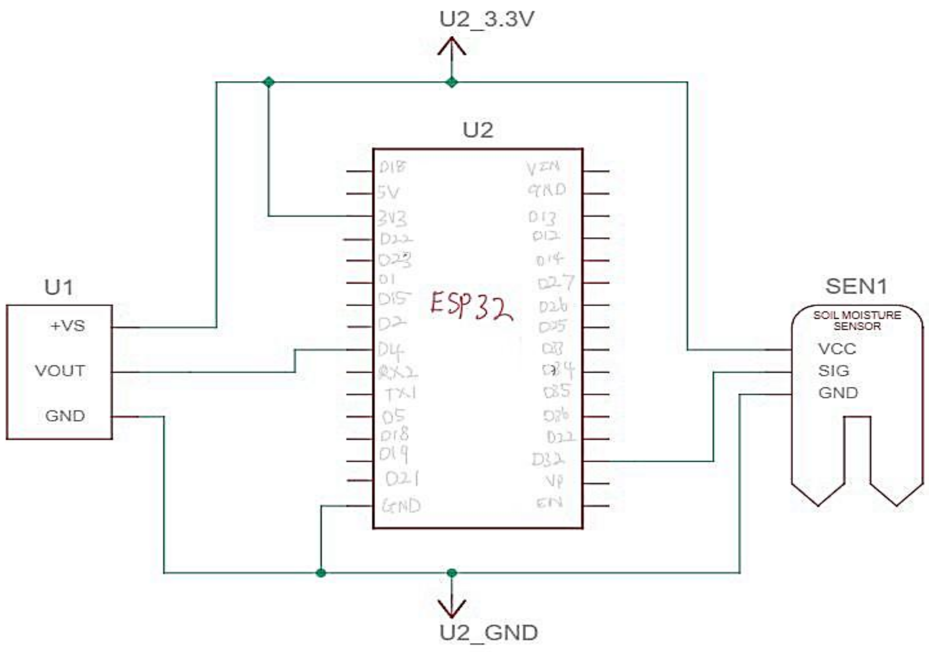


Figure 25: Schematic Diagram of Microcontroller

4.3.1 Circuit Description

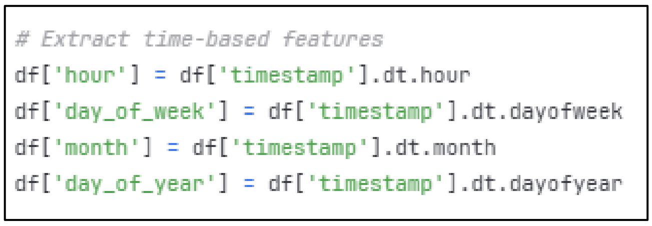
- U1 – DHT22 temperature & humidity sensor
    - +VS: Connected to 3.3V power supply (U2_3.3V).
    - VOUT: Connected to a GPIO pin on the ESP32 to read temperature and humidity data (U2_D4).
    - GND: Connected to the ESP32's ground (U2_GND).
- U2 – ESP32 microcontroller
    - Powered by USB adapter.

    - DHT22 data from the VOUT pin via GPIO (U2_D4).
    - Soil moisture data from the SIG pin of the soil moisture via another GPIO pin (U2_D32).
- SEN1 – Soil moisture sensor
    - VCC: Connected to 3.3V power line
    - SIG: Connected to a GPIO pin on the ESP32 for reading analog (U2_D32).

4.4 AI Model Training Logic and Design

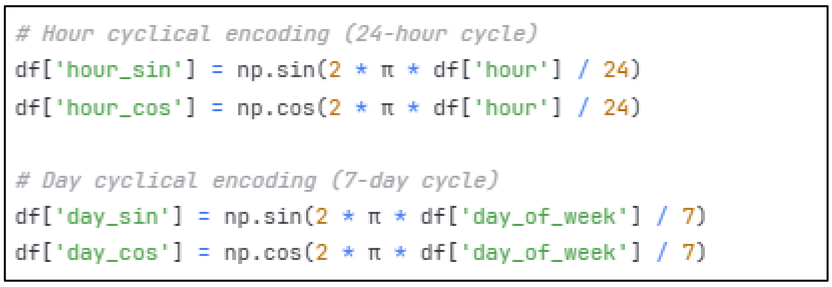
4.4.1 Data Pipeline Architecture

The data input required columns included timestamp, temperature, humidity and soil_moisture. The minimum data points are 50 records and there is a function for automatic validation for missing columns and insufficient data. Now let's move to the data preprocessing pipeline. There are several processes needed to transform raw sensor data into meaning features which are temporal feature extraction, cyclical encoding, interaction features, temporal aggregation features and lag features. Figure 26, 27, 28, 29 and 30 shows the code for temporal feature extraction, cyclical encoding, interaction features, temporal features aggregation features and lag features.



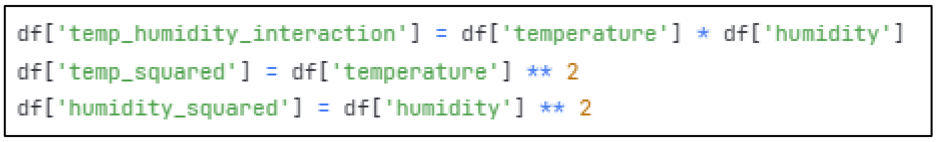
```
# Extract time-based features
df['hour'] = df['timestamp'].dt.hour
df['day_of_week'] = df['timestamp'].dt.dayofweek
df['month'] = df['timestamp'].dt.month
df['day_of_year'] = df['timestamp'].dt.dayofyear
```

Figure 26: Temporal Feature Extraction



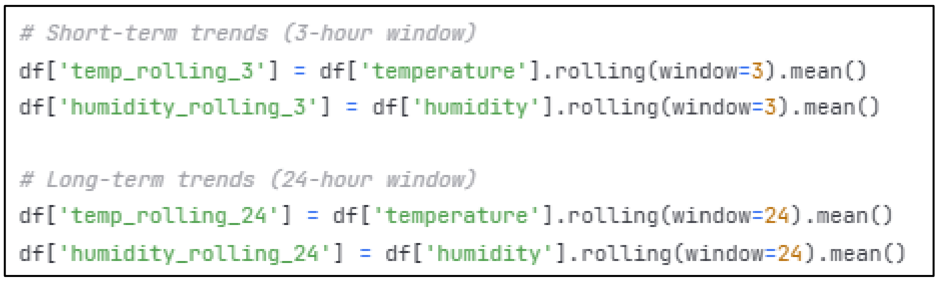
```
# Hour cyclical encoding (24-hour cycle)
df['hour_sin'] = np.sin(2 * π * df['hour'] / 24)
df['hour_cos'] = np.cos(2 * π * df['hour'] / 24)

# Day cyclical encoding (7-day cycle)
df['day_sin'] = np.sin(2 * π * df['day_of_week'] / 7)
df['day_cos'] = np.cos(2 * π * df['day_of_week'] / 7)
```

Figure 27: Cyclical Encoding



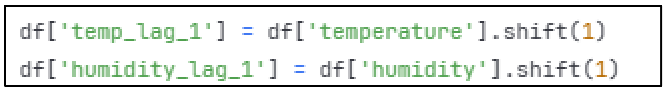
```
df['temp_humidity_interaction'] = df['temperature'] * df['humidity']
df['temp_squared'] = df['temperature'] ** 2
df['humidity_squared'] = df['humidity'] ** 2
```

Figure 28: Interaction Features



```
# Short-term trends (3-hour window)
df['temp_rolling_3'] = df['temperature'].rolling(window=3).mean()
df['humidity_rolling_3'] = df['humidity'].rolling(window=3).mean()

# Long-term trends (24-hour window)
df['temp_rolling_24'] = df['temperature'].rolling(window=24).mean()
df['humidity_rolling_24'] = df['humidity'].rolling(window=24).mean()
```

Figure 29: Temporal Feature Aggregation Features



```
df['temp_lag_1'] = df['temperature'].shift(1)
df['humidity_lag_1'] = df['humidity'].shift(1)
```

Figure 30: Lag Features

The purpose of temporal feature extraction is to divide the timestamp received into several forms which is hour, day of week, month and day of year to indicate the timestamp more details and specific to the AI. The cyclical encoding is to convert the time-based feature to cyclical representation to capture periodic patterns. After that the interaction features are to create composite features that capture relationship between variables. The temporal aggregation feature is to roll averages to capture the recent trends by retrieving out 3-hour window and 24-hour window of data. Lastly, the lag feature is used to capture temporal dependencies by using previous time step values.

4.4.2 Machine Learning Model Architecture

The algorithm selected for the machine learning model is ensemble methods. This is because ensemble method combines multiple individual models to create a stronger, more accurate predictor. It is like asking multiple experts for their opinion and then combining their answer to get a better result than any single expert could provide. There are 2 main types of ensemble method used which are random forest regressor and gradient boosting regressor.

The parameters tuned for random forest are:

- n_estimators: [100, 200] - Number of trees in the forest
- max_depth: [10, 20, None] - Maximum depth of each tree
- min_samples_split: [2, 5] - Minimum samples required to split a node
- min_samples_leaf: [1, 2] - Minimum samples required at a leaf node
- max_features: ['sqrt', 'log2'] - Number of features to consider at each split

The parameters tuned for gradient boosting are:

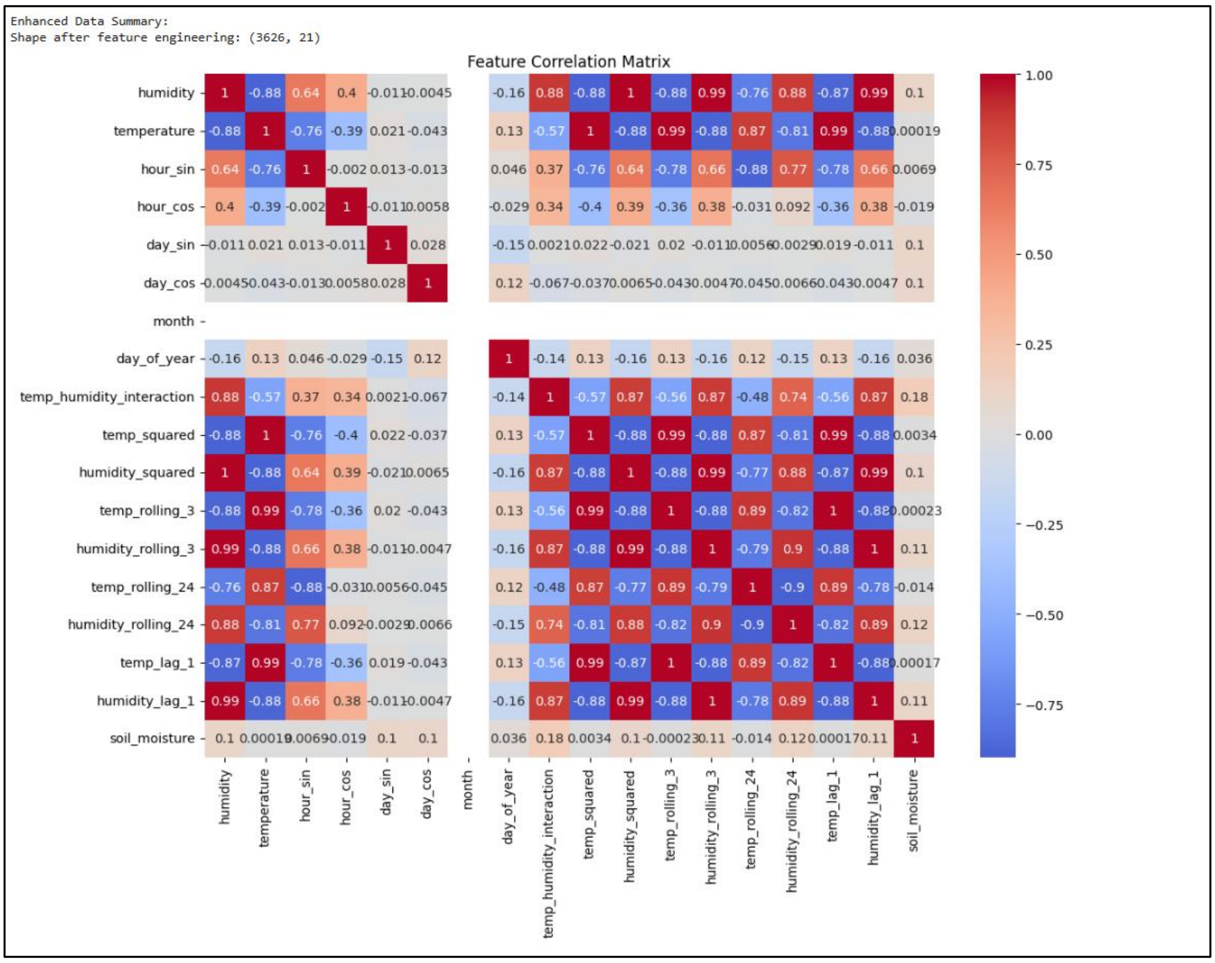
- n_estimators: [100, 200] - Number of boosting stages

- max_depth: [3, 5] - Maximum depth of individual trees (kept shallow)
- learning_rate: [0.1, 0.2] - How much each model contributes
- subsample: [0.8, 1.0] - Fraction of samples used for fitting

The reason for choosing these 2 models is because random forest regressor can handle non-linear relationships, robust to outliers and provide feature importance while gradient boosting regressor can build models sequentially, each learning from previous mistake and each model focuses on the hardest-to-predict examples and able to handle complex patterns.

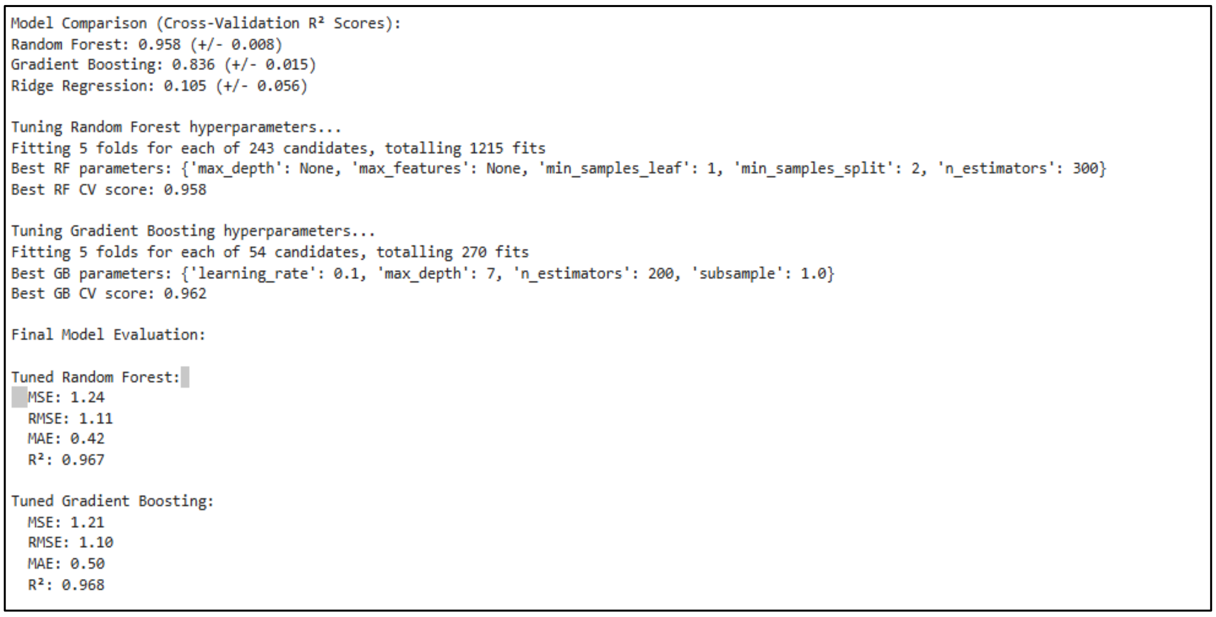
Hyperparameter optimization is required for machine learning model training. Hyperparameter are the setting or configuration of a machine learning algorithm that you must set before training begins. After parameter tuning, cross validation will be implemented for each combination by splitting training data into 3 folds, train models on 2folds, test on 1 fold and repeat them 3 times (each fold used as test once). After that, calculate the average R2 score across 3 folds. This average R2 represents the parameter combination's performance. The system will filter out the best results of the model.

Overall, the flow of the training process will be like the first process is data validation, second is preprocessing, third is feature engineering, fourth is train-test split, fifth is hyperparameter tuning, sixth is model selection, seventh is the evaluation and the last step is model persistence. The system will automatically select the model with the highest R2 score on the test set and also display performance metrics like MSE (mean squared error), RMSE (root mean squared error), MAE (mean absolute error) and R2 score. Figure 31, 32, 33 and 34 below shows one of the results when running through the training process.



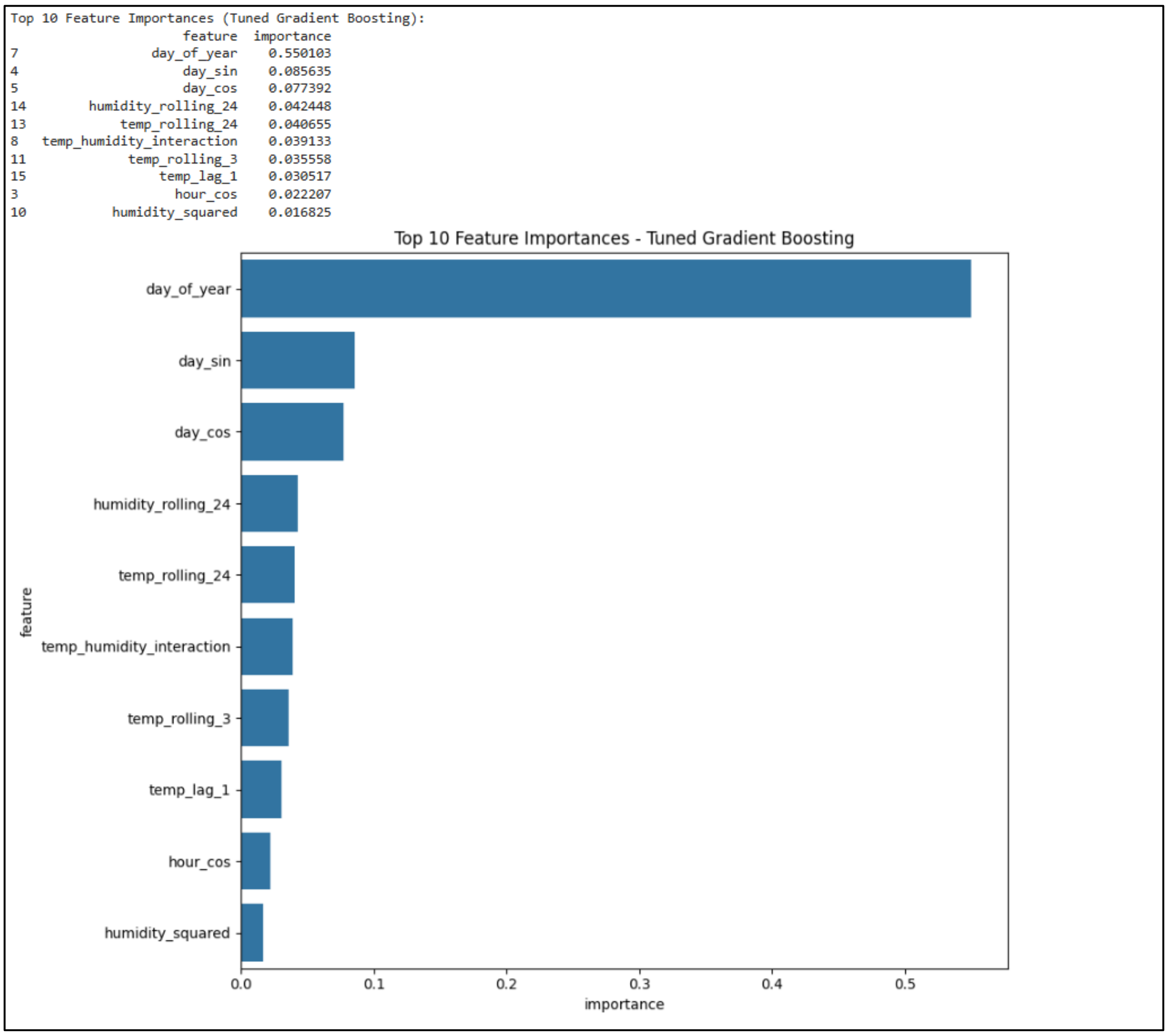
```
Enhanced Data Summary:
Shape after feature engineering: (3626, 21)
```

Figure 31: Feature Correlation Matrix Heatmap



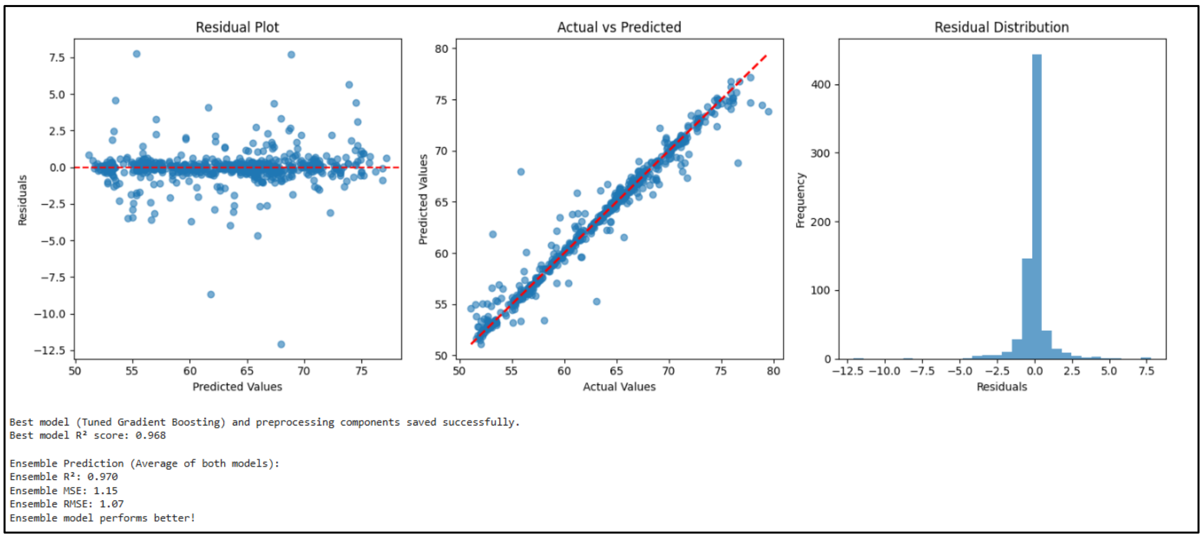
```
Model Comparison (Cross-Validation R² Scores):
Random Forest: 0.958 (+/- 0.008)
Gradient Boosting: 0.836 (+/- 0.015)
Ridge Regression: 0.105 (+/- 0.056)

Tuning Random Forest hyperparameters...
Fitting 5 folds for each of 243 candidates, totalling 1215 fits
Best RF parameters: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 300}
Best RF CV score: 0.958

Tuning Gradient Boosting hyperparameters...
Fitting 5 folds for each of 54 candidates, totalling 270 fits
Best GB parameters: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200, 'subsample': 1.0}
Best GB CV score: 0.962

Final Model Evaluation:

Tuned Random Forest:
  MSE: 1.24
  RMSE: 1.11
  MAE: 0.42
  R²: 0.967

Tuned Gradient Boosting:
  MSE: 1.21
  RMSE: 1.10
  MAE: 0.50
  R²: 0.968
```

Figure 32: Hyperparameter Tuning and Model Comparison Results



```
Top 10 Feature Importances (Tuned Gradient Boosting):
                     feature  importance
7                day_of_year    0.550103
4                    day_sin    0.085635
5                    day_cos    0.077392
14          humidity_rolling_24    0.042448
13              temp_rolling_24    0.040655
8    temp_humidity_interaction    0.039133
11               temp_rolling_3    0.035558
15                   temp_lag_1    0.030517
3                    hour_cos    0.022207
10            humidity_squared    0.016825
```

Figure 33: Feature Importance Analysis for Gradient Boosting Model



```
Best model (Tuned Gradient Boosting) and preprocessing components saved successfully.
Best model R² score: 0.968

Ensemble Prediction (Average of both models):
Ensemble R²: 0.970
Ensemble MSE: 1.15
Ensemble RMSE: 1.07
Ensemble model performs better!
```

Figure 34: Model Performance Diagnostic Plots

Figure 31 about feature correlation matrix heatmap is displaying relationships between all 21 engineered features and the target variable (soil_moisture) from 3,626 data samples. Strong correlations (0.88-0.99) are evident among related feature groups such as humidity variables, temperature measurements, and their corresponding rolling averages, validating feature engineering effectiveness. The weak individual correlations with soil_moisture (-0.16 to 0.18) indicate that predictive power emerges from complex feature interactions rather than simple linear relationships.

Figure 32 is the comprehensive comparison of model performance showing cross-validation R^2 scores for Random Forest (0.958 ± 0.008), Gradient Boosting (0.836 ± 0.015), and Ridge Regression baseline (0.105 ± 0.056). The results display optimal hyperparameters found through grid search optimization, with final model evaluation metrics confirming Gradient Boosting as the best performer ($R^2 = 0.968$, RMSE = 1.10, MAE = 0.50).

Figure 33 shows the top 10 most important features ranked by their contribution to soil moisture prediction accuracy. The day_of_year feature dominates with 55% importance, indicating strong seasonal patterns in soil moisture behavior. Temporal cyclical features (day_sin, day_cos) and environmental rolling averages (humidity_rolling_24, temp_rolling_24) constitute the remaining top contributors, demonstrating the effectiveness of engineered temporal and environmental features.

While figure 34 is the three-panel diagnostic visualization showing (left) residual plot with randomly distributed errors around zero indicating good model fit, (center) actual vs predicted values plot demonstrating strong linear correlation ($R^2 = 0.968$) with points closely following the perfect prediction line, and (right) residual distribution histogram showing near-normal error distribution centered at zero, confirming model validity and unbiased predictions.

Chapter 5
System Implementation

5.1 Hardware Setup

This section details the physical assembly of the hardware components used for the IoT data collection device. The setup is designed to be straightforward, using a breadboard to manage power distribution and connections between the ESP32 microcontroller and the sensors.

The core of the setup is the ESP32 DevKitC V4 board. To power the other components, the ESP32's 3.3V and GND pins are connected to the positive and negative power rails of the breadboard, respectively. This allows the breadboard to act as a central power hub for the sensors. Figure 35 below shows the hardware setup overview. After connecting all components, open Arduino IDE and paste the written source code located at Appendix I. Compile and transmit to the microcontroller. The following components were connected to the ESP32:

- DHT22 Sensor: This sensor is responsible for measuring temperature and humidity. Its VCC and GND pins are connected to the breadboard's power rails. The crucial data pin is connected directly to digital pin D4 on the ESP32.
- Capacitive Soil Moisture Sensor: This sensor provides soil moisture readings during "developer mode." Like the DHT22, it is powered from the breadboard's 3.3V and GND rails. Its analog signal pin is connected to pin D32 on the ESP32, which is an Analog-to-Digital Converter (ADC) pin capable of reading the sensor's output voltage.
- Status LED: The built-in blue LED on the ESP32 board is used to provide visual feedback on the device's operational status, such as when it is connecting to Wi-Fi or operating in a specific mode.

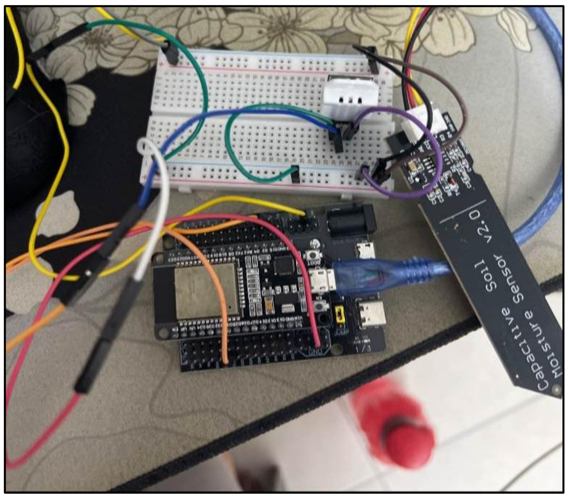


Figure 35: Hardware Setup Overview

5.2 Software Setup

This section details all the software tools required to develop and deploy the Sensor Substitution using AI for Agriculture Soil Moisture Monitoring system. It includes development environments, programming tools, and command-line utilities used throughout the project, along with download links and basic installation guidance.

5.2.1 Arduino IDE

- Purpose: Programming the ESP32 microcontroller and uploading sensor code.
- Download link: https://www.arduino.cc/en/software
- Installation notes:
    - Install the ESP32 board package via the Boards Manager.

5.2.2 Visual Studio Code (VS Code)

- Purpose: Code editor for backend (Go), frontend (React), and AI (Python) codebases.
- Download link: https://code.visualstudio.com/
- Recommended extensions:
    - Python
    - Go
    - PostgreSQL
    - Docker
    - GitHub Copilot

5.2.3 GitHub CLI

- Purpose: Version control, CI/CD automation, and repository management.
- Download link: https://cli.github.com/
- Installation command (Windows):
    - winget install –id GitHub.cli

5.2.4 Google Cloud CLI

- Purpose: Deploying services (Flask, Go backend) to Google Cloud Run
- Download link: https://cloud.google.com/sdk/docs/install
- Setup:
    - gcloud init
    - gcloud auth login
    - gcloud config set project [PROJECT_ID]
- Use cases: Access google cloud services securely and efficiently.

5.2.5 Docker CLI

- Purpose: Containerization of AI services for consistent deployment.
- Download link: https://www.docker.com/products/docker-desktop/
- Basic usage:
    - docker build -t my-container-name .
    - docker run -p 8080:8080 my-container-name

5.2.6 Node.js and npm CLI

- Purpose: Frontend development and dependency management.
- Download link: https://nodejs.org/
- Recommended version: LTS version (Node 18.x or Node 20.x)
- Basic commands:
    - npm install
    - npm run dev
    - npm run build

5.2.7 Firebase CLI

- Purpose: Firebase is used for hosting the frontend web app.
- Download link: https://firebase.google.com/docs/cli
- Install command (via npm): npm install -g firebase-tools
- Initialize firebase command:
    - firebase login
    - firebase init

5.2.8 Go CLI

- Purpose: Developing and running the backend REST API server.
- Download Link: https://go.dev/dl/
- Environment setup:
    - Add go binary path /bin to system's path.
    - Initialize Go module with command below:
        - go mod init my-backend-name
        - go mod tidy
        - go run main.go

5.2.9 Python CLI

- Purpose: Training machine learning models and running the Flask AI server.
- Download link: https://www.python.org/downloads/
- Recommended version: Python 3.10+

5.2.10 Postman Desktop

- Purpose: API testing for the backend restful api server.
- Download link: https://www.postman.com/

5.3 Setting and Configuration

This section outlines the configuration steps required to ensure all components which are the hardware, software, backend, frontend, and AI are properly connected and operational. These configurations are critical for replicating the system and ensuring compatibility across platforms.

5.3.1 ESP32 Microcontroller (Arduino IDE)

- Board manager setup:
  - In Arduino IDE, go to Preferences, add this URL https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json to additional board manager URLs.
  - Install ESP32 by Espressif Systems from board manager.
- Board selection:
  - Tools – Board: Select ESP32 Dev Module
  - Tools – Port: Select the correct COM port (e.g. COM3)
- Libraries installed:
  - DHT sensor library
  - Adafruit unified sensor
  - WiFi.h, HTTPClient.h, ArduinoJson.h, esp_task_wdt.h

5.3.2 Google Cloud Configuration

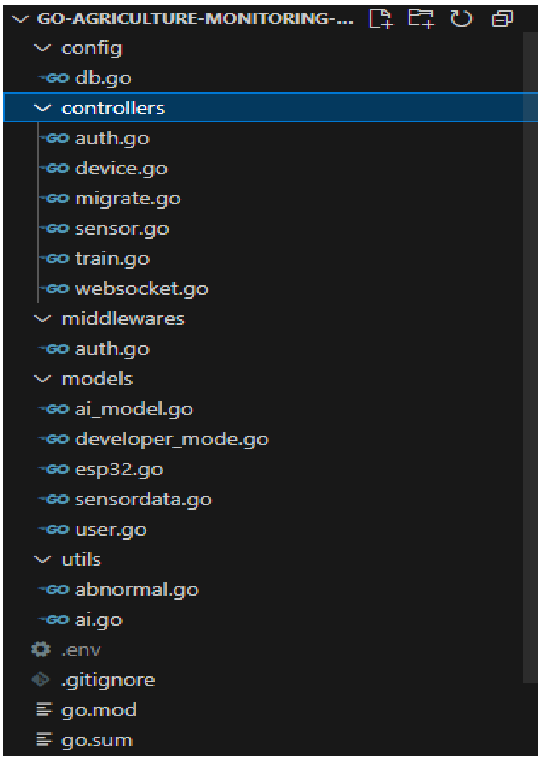
- Authentication (command):
  - gcloud auth login
  - gcloud config set project [PROJECT_ID]

5.3.3 Firebase Setting and Configuration for Frontend

- Create a firebase project at https://console.firebase.google.com/.
- Enable hosting.
- In react project, run command:
  - firebase init hosting
- Deploy the frontend to firebase by using command:
  - npm run build
  - firebase deploy

5.3.4 Backend Server (Go)

- Folder structure: Figure 36 below shows the folder structure of backend server.



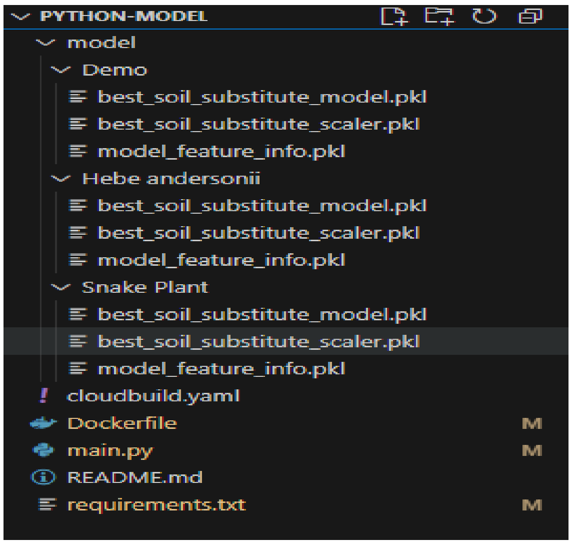
```
GO-AGRICULTURE-MONITORING-...
  config
    db.go
  controllers
    auth.go
    device.go
    migrate.go
    sensor.go
    train.go
    websocket.go
  middlewares
    auth.go
  models
    ai_model.go
    developer_mode.go
    esp32.go
    sensordata.go
    user.go
  utils
    abnormal.go
    ai.go
  .env
  .gitignore
  go.mod
  go.sum
```

Figure 36: Go Backend Folder Structure

- Configuration file (.env):
    - DATABASE_URL=postgresql://username:password@name.postgres.database .azure.com:port/postgres
    - AI_URL=https://python-model-738775168875.asia-southeast1.run.app/predict
    - PYTHON_TRAINING_SERVICE_URL=https://python-model-738775168875.asia-southeast1.run.app
    - GOOGLE_API_KEY=google-map-api-key
- Run command (use port 8080): go run main.go

5.3.5 AI Server (Python + Flask)

- Model folder structure: Figure 37 below shows the flask server folder structure.



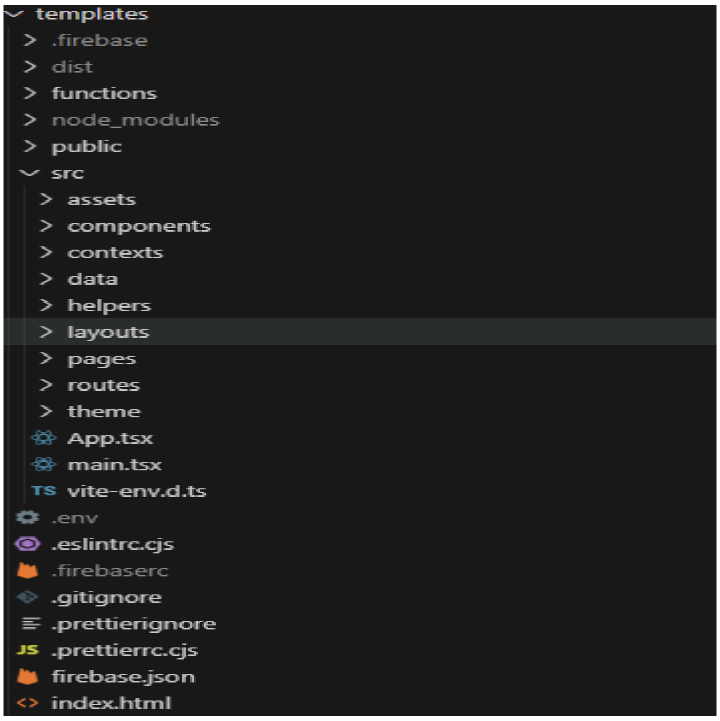
```
PYTHON-MODEL
  model
    Demo
      best_soil_substitute_model.pkl
      best_soil_substitute_scaler.pkl
      model_feature_info.pkl
    Hebe andersonii
      best_soil_substitute_model.pkl
      best_soil_substitute_scaler.pkl
      model_feature_info.pkl
    Snake Plant
      best_soil_substitute_model.pkl
      best_soil_substitute_scaler.pkl
      model_feature_info.pkl
  cloudbuild.yaml
  Dockerfile                         M
  main.py                            M
  README.md
  requirements.txt                   M
```

Figure 37: AI Server Folder Structure

- Environment setup:
  - Run command: pip install -r requirements.txt
  - Run command (use port 5000): python main.py

5.3.6 Frontend Web App (React + TypeScript)

- Folder structure: Figure 38 below shows the frontend folder structure.



```
templates
  .firebase
  dist
  functions
  node_modules
  public
  src
    assets
    components
    contexts
    data
    helpers
    layouts
    pages
    routes
    theme
    App.tsx
    main.tsx
    vite-env.d.ts
  .env
  .eslintrc.cjs
  .firebaserc
  .gitignore
  .prettierignore
  .prettierrc.cjs
  firebase.json
  index.html
```

Figure 38: Frontend Folder Structure

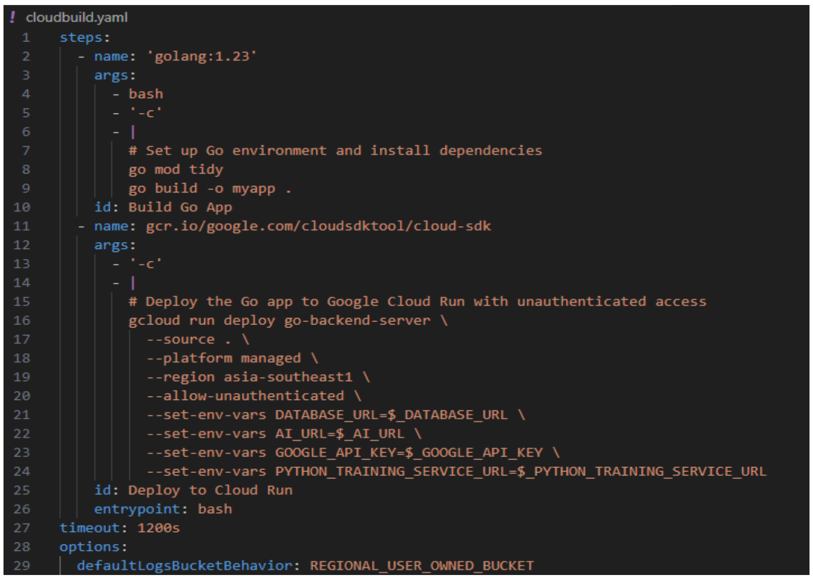
- Configuration file (.env):
  - VITE_API_URL= https://go-backend-server-738775168875.asia-southeast1.run.app
  - VITE_GOOGLE_MAPS_API_KEY=google-map-api-key
  - VITE_API_WS_URL= wss://go-backend-server-738775168875.asia-southeast1.run.app/ws
- Run dev server command:
  - npm install
  - npm run dev

5.3.7 Azure PostgreSQL Database Configuration

- Cloud provider: Microsoft azure
- Database type: PostgreSQL flexible server
- Steps:
  1. Create PostgreSQL server on azure portal.
  2. Configure firewall rules to allow GCP IP ranges.
  3. Create a database.
  4. Create a user and password.
- Connection string format:
  - postgresql://<username>:<password>@<host>.postgres.database.azure.com:5432/<database_name>
- Environment variables in Go Backend:
  - DATABASE_URL=postgresql://username:password@host.postgres.database.azure.com:5432/database_name

5.3.8 Google Cloud Platform Deployment Configuration

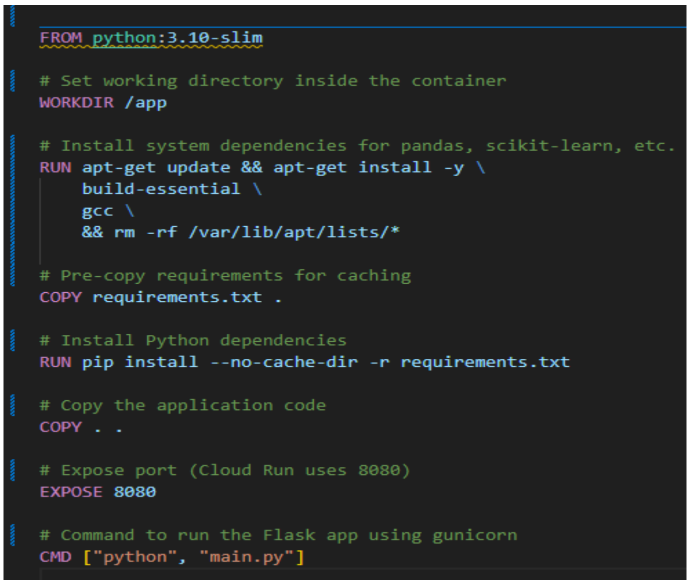
- Go backend deploy to cloud run (YAML configuration): Figure 39 below shows the cloudbuild.yaml example.



```yaml
cloudbuild.yaml
steps:
  - name: 'golang:1.23'
    args:
      - bash
      - '-c'
      - |
        # Set up Go environment and install dependencies
        go mod tidy
        go build -o myapp .
    id: Build Go App
  - name: gcr.io/google.com/cloudsdktool/cloud-sdk
    args:
      - '-c'
      - |
        # Deploy the Go app to Google Cloud Run with unauthenticated access
        gcloud run deploy go-backend-server \
          --source . \
          --platform managed \
          --region asia-southeast1 \
          --allow-unauthenticated \
          --set-env-vars DATABASE_URL=$_DATABASE_URL \
          --set-env-vars AI_URL=$_AI_URL \
          --set-env-vars GOOGLE_API_KEY=$_GOOGLE_API_KEY \
          --set-env-vars PYTHON_TRAINING_SERVICE_URL=$_PYTHON_TRAINING_SERVICE_URL
    id: Deploy to Cloud Run
    entrypoint: bash
timeout: 1200s
options:
  defaultLogsBucketBehavior: REGIONAL_USER_OWNED_BUCKET
```

Figure 39: cloudbuild.yaml of Go Backend Server

- Deployment command for Go backend deploy to cloud run:
  - gcloud run services replace cloudbuild.yaml
- Flask AI server deploy to cloud run via Docker.
  - Dockerfile for python flask: figure 40 below is the contain inside Dockerfile.



```dockerfile
FROM python:3.10-slim

# Set working directory inside the container
WORKDIR /app

# Install system dependencies for pandas, scikit-learn, etc.
RUN apt-get update && apt-get install -y \
    build-essential \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Pre-copy requirements for caching
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application code
COPY . .

# Expose port (Cloud Run uses 8080)
EXPOSE 8080

# Command to run the Flask app using gunicorn
CMD ["python", "main.py"]
```

Figure 40: Dockerfile for Python Flask

  - Build and deploy command:

    gcloud builds submit --tag gcr.io/[PROJECT_ID]/python-model

    gcloud run deploy python-model \

     --image gcr.io/[PROJECT_ID]/python-model \

     --platform managed \

    --region asia-southeast1

5.4 System Operation

5.4.1 System Access and User Authentication

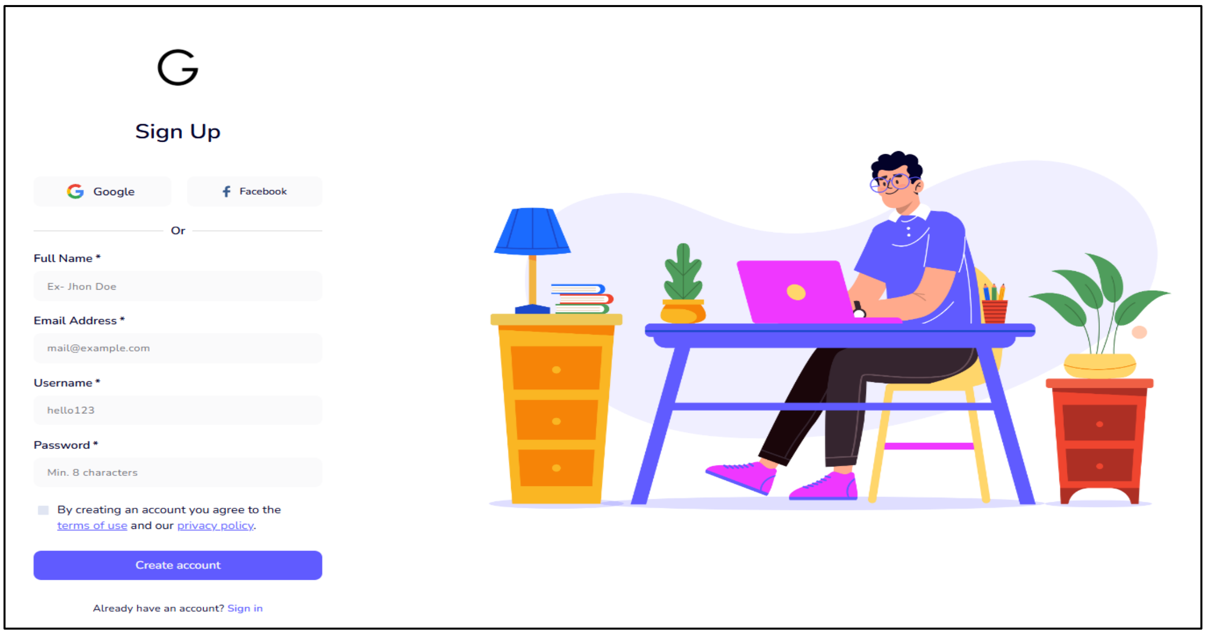


Figure 41: GoMonitor SignUp Interface

The GoMonitor system can be accessed via:

https://fyp-backend-bd5cc.web.app/base/auth/signup

The figure 41 above is the GoMonitor system signup interface. It allows users to register their own account. The system supports user authentication and provides access to monitoring and managing the functionalities for plant data collection and AI-powered predictions for soil moisture. After signing up, the user will be able to sign in using the successful signup account. Figure 42 below is the GoMonitor sign in interface. For demo purposes, please sign in using username: staging and password: 12345 for running test.

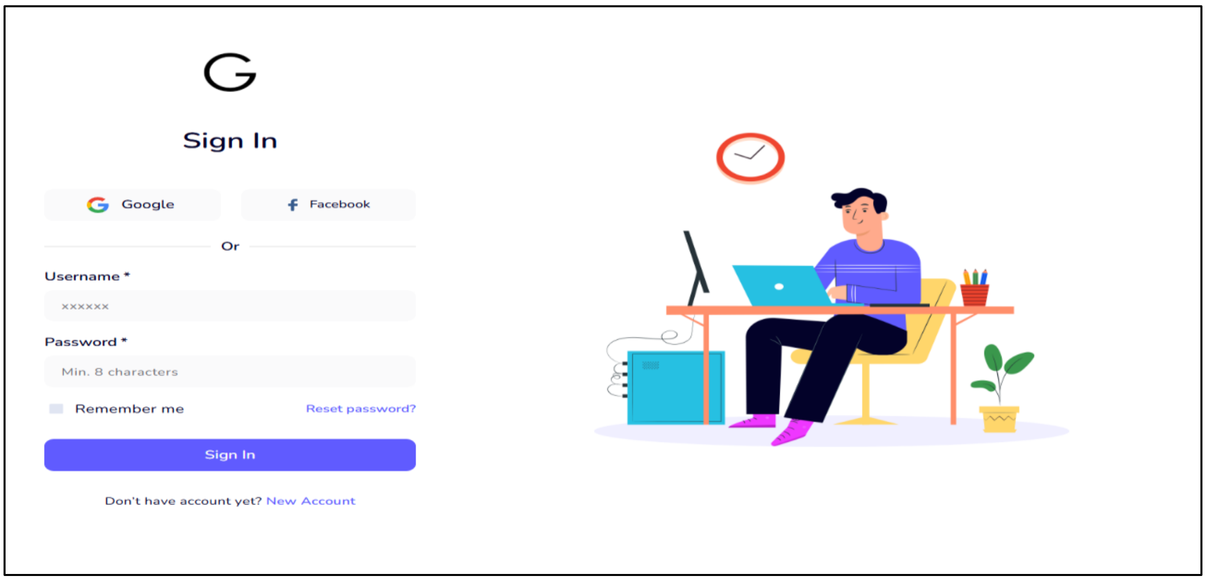
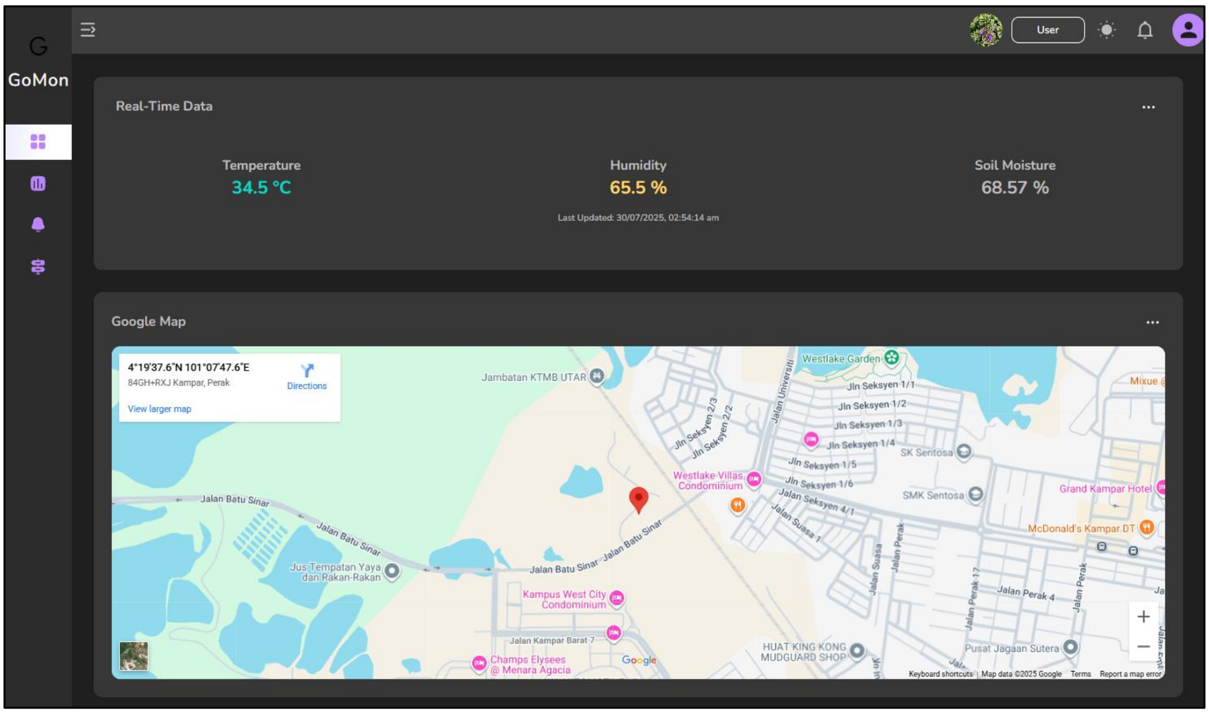


Figure 42: GoMonitor SignIn Interface

5.4.2 Main Dashboard Overview



Real-Time Data

Temperature
34.5 °C

Humidity
65.5 %

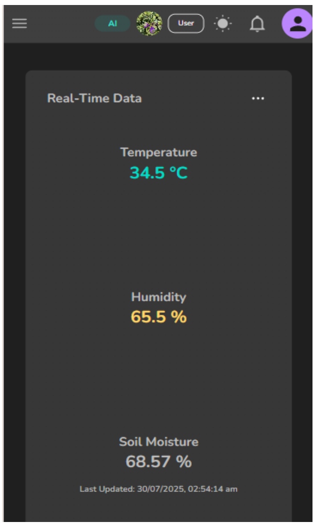
Soil Moisture
68.57 %

Last Updated: 30/07/2025, 02:54:14 am

4°19'37.6"N 101°07'47.6"E
84GH+RXJ Kampar, Perak

Figure 43: GoMonitor Main Dashboard Overview

Figure 43 above shows the main dashboard overview of the system. The main dashboard provides an overview of real time data and the microcontroller's current location. The real-time data is fetched using WebSocket which has no latency. The top bar has the selection of user mode / developer mode, shifting the theme between light mode and dark mode, AI toggle button for plant model, and recent alerts and notifications. There are 4 options available at the sidebar, which are dashboard, management, notification and guidelines page. This web application is also available in mobile friendly. Figure 44 below shows the view on a mobile website.



Real-Time Data

Temperature
34.5 °C

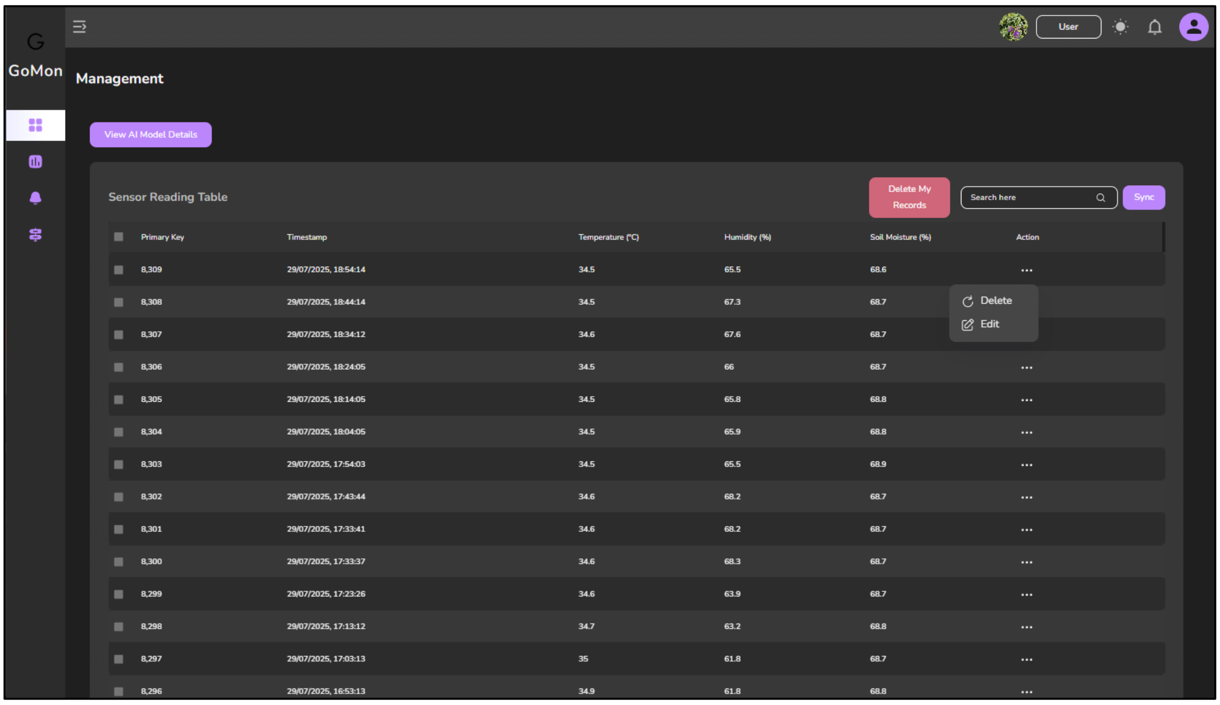
Humidity
65.5 %

Soil Moisture
68.57 %

Last Updated: 30/07/2025, 02:54:14 am

Figure 44: GoMonitor Main Dashboard Overview in Mobile

5.4.3 Management Page Overview

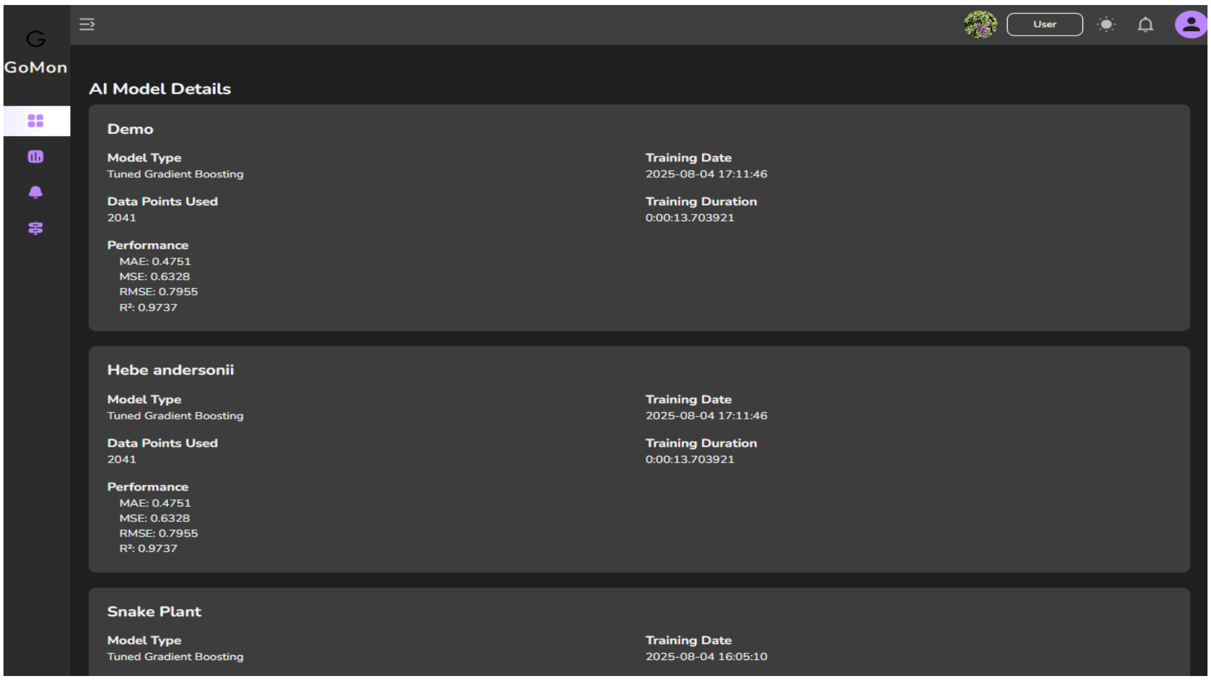


Sensor Reading Table

Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action
8,309	29/07/2025, 18:54:14	34.5	65.5	68.6	···
8,308	29/07/2025, 18:44:14	34.5	67.3	68.7	
8,307	29/07/2025, 18:34:12	34.6	67.6	68.7	
8,306	29/07/2025, 18:24:05	34.5	66	68.7	···
8,305	29/07/2025, 18:14:05	34.5	65.8	68.8	···
8,304	29/07/2025, 18:04:05	34.5	65.9	68.8	···
8,303	29/07/2025, 17:54:03	34.5	65.5	68.9	···
8,302	29/07/2025, 17:43:44	34.6	68.2	68.7	···
8,301	29/07/2025, 17:33:41	34.6	68.2	68.7	···
8,300	29/07/2025, 17:33:37	34.6	68.3	68.7	···
8,299	29/07/2025, 17:23:26	34.6	63.9	68.7	···
8,298	29/07/2025, 17:13:12	34.7	63.2	68.8	···
8,297	29/07/2025, 17:03:13	35	61.8	68.7	···
8,296	29/07/2025, 16:53:13	34.9	61.8	68.8	···

Figure 45: Management Page Interface

Figure 45 above shows the management page interface which is showing a clickable button "View AI Model Details" and a sensor reading table which have the feature of delete all data, delete chose data, and edit the data. There is a sync button allowing users to click to get the newest data, also known as refresh sensor reading table button. When users click the "View AI Model Details" button, the webpage will direct user to another page which have been shown at figure 46 below. It will show the available ai model and their performance, and train duration.



AI Model Details

Demo

Model Type
Tuned Gradient Boosting

Training Date
2025-08-04 17:11:46

Data Points Used
2041

Training Duration
0:00:13.703921

Performance
MAE: 0.4751
MSE: 0.6328
RMSE: 0.7955
R²: 0.9737

Hebe andersonii

Model Type
Tuned Gradient Boosting

Training Date
2025-08-04 17:11:46

Data Points Used
2041

Training Duration
0:00:13.703921

Performance
MAE: 0.4751
MSE: 0.6328
RMSE: 0.7955
R²: 0.9737

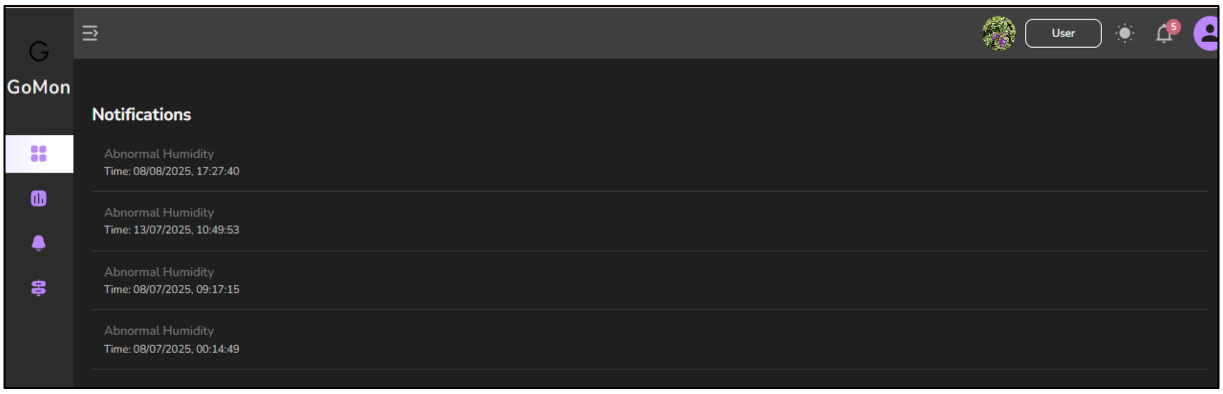
Snake Plant

Model Type
Tuned Gradient Boosting

Training Date
2025-08-04 16:05:10

Figure 46: AI Model Details Page

5.4.4 Notification Page Overview



Notifications

Abnormal Humidity
Time: 08/08/2025, 17:27:40

Abnormal Humidity
Time: 13/07/2025, 10:49:53

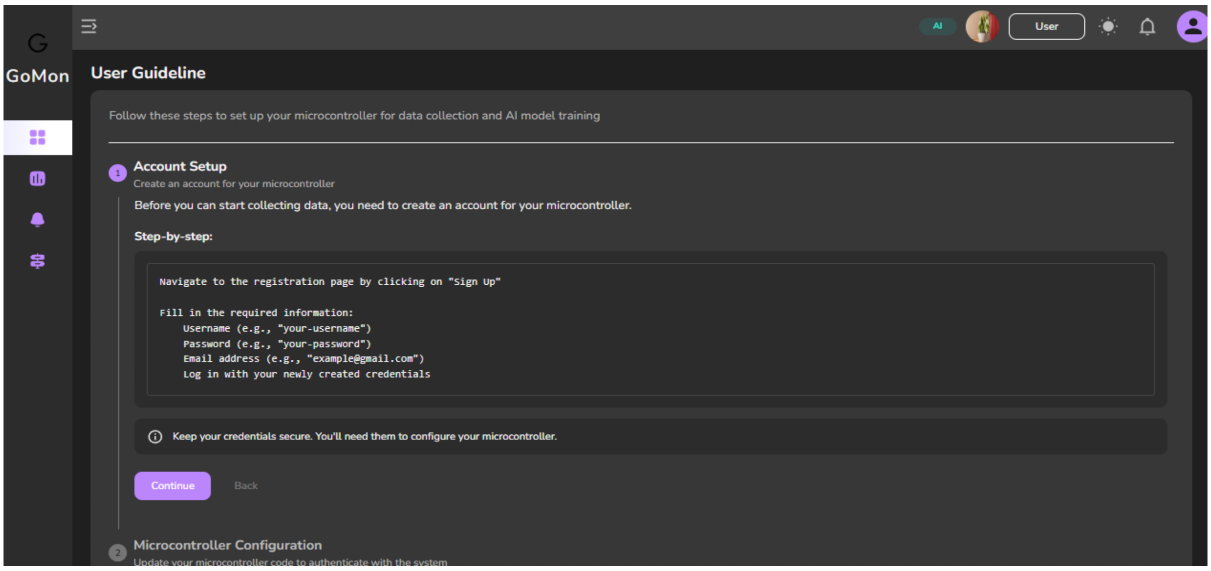
Abnormal Humidity
Time: 08/07/2025, 09:17:15

Abnormal Humidity
Time: 08/07/2025, 00:14:49

Figure 47: Notification Page

This is the notification page of the system, which will jump red alert on the top bar as the figure 47 above shows when the system received abnormal temperature or humidity value. The value of temperature will detect as abnormal when it is lower than 10 degrees Celsius or higher than 50 degrees Celsius, while the humidity is lower than 20 percent or higher than 90 percent. This feature is to alert users to check whether there are sensor malfunctions, or the climate change occurs suddenly. It provides the users with taking fast actions on it when receiving notification and alerts. When users click on the notification icon on the sidebar or top bar, it will direct to this notification page where users will be able to check which data is abnormal and what time the data is collected for further diagnosis.

5.4.5 Guideline Page Overview



User Guideline

Follow these steps to set up your microcontroller for data collection and AI model training

1 **Account Setup**
Create an account for your microcontroller

Before you can start collecting data, you need to create an account for your microcontroller.

Step-by-step:

```
Navigate to the registration page by clicking on "Sign Up"

Fill in the required information:
    Username (e.g., "your-username")
    Password (e.g., "your-password")
    Email address (e.g., "example@gmail.com")
    Log in with your newly created credentials
```

Keep your credentials secure. You'll need them to configure your microcontroller.

2 Microcontroller Configuration
Update your microcontroller code to authenticate with the system

Figure 48: Guideline Page

Figure 48 above is the guideline page overview; the purpose of this page is to teach user how to set up their microcontroller to connect with system and use the features. All setup procedures are given.

5.4.6 Real-time Monitoring & User Mode Operation

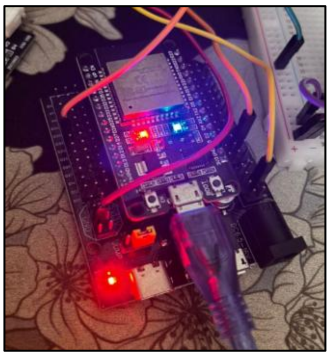


Figure 49: Microcontroller Status

For the real-time monitoring, the microcontroller will be setup with prepared code using Arduino IDE for collecting data and sent to backend server, after that the frontend will retrieve data from the backend server. Figure 49 above shows the microcontroller status by blue LED, when LED is blinking, it means that the system for the specific account is currently in developer mode, when it is not blinking, it means the system is currently in user mode. When the system is in user mode, the microcontroller will only send temperature and humidity data to backend and the soil moisture will be predicted by the AI model. Figure 50 below shows the users how to toggle the AI model for specific plant ON. When users click on the photo on the top bar, this popup modal will come out.

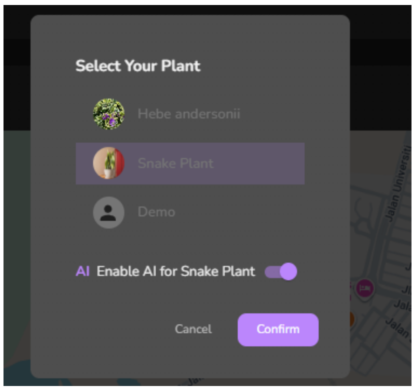
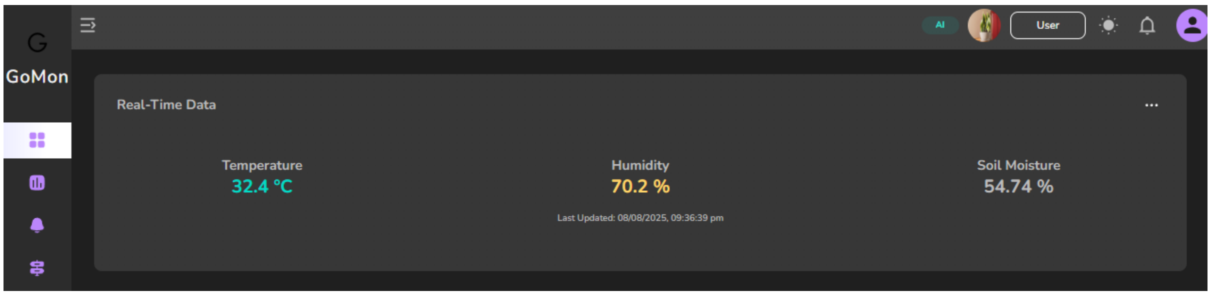


Figure 50: AI Plant Model Selection



Real-Time Data

Temperature 32.4 °C

Humidity 70.2 %

Soil Moisture 54.74 %

Last Updated: 08/08/2025, 09:36:39 pm

Figure 51: Main Dashboard Operations

After the AI model is chosen, users will notice there is an AI icon at the top bar shown, which means AI is enabled in user mode, and when the microcontroller sends data to the backend, the soil moisture will be predicted, and the front end will render out the real-time data using WebSocket. Figure 51 above shows the results.

5.4.7 Developer Mode & AI Model Training Operation

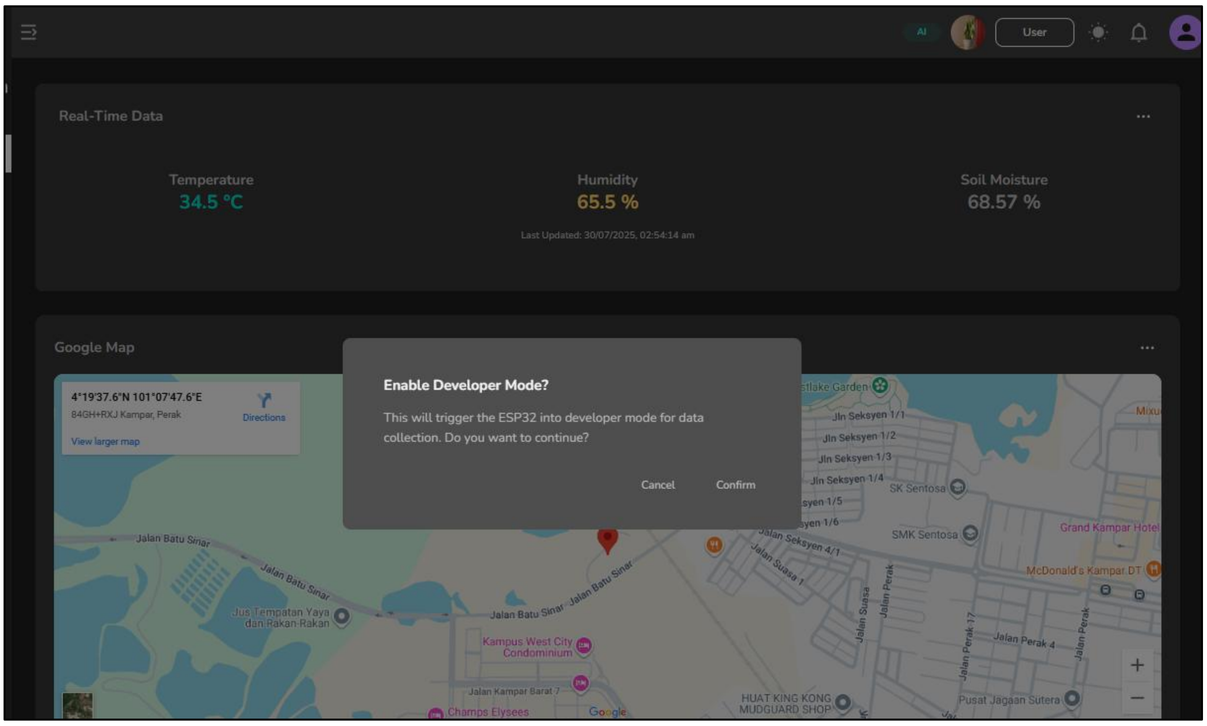

User
Real-Time Data
Temperature
34.5 °C
Humidity
65.5 %
Soil Moisture
68.57 %
Last Updated: 30/07/2025, 02:54:14 am
Google Map
Enable Developer Mode?
This will trigger the ESP32 into developer mode for data collection. Do you want to continue?
Cancel
Confirm

Figure 52: Developer Mode Enable Confirmation

For the users to enable developer mode, just click on the “user” button once and this confirmation note will be shown to ask confirmation of trigger ESP32 into developer mode. Figure 52 above shows the screenshot of the confirmation. After confirmation, the dashboard will become different, figure 53, 54, and 55 shows the new view of the dashboard in developer mode.

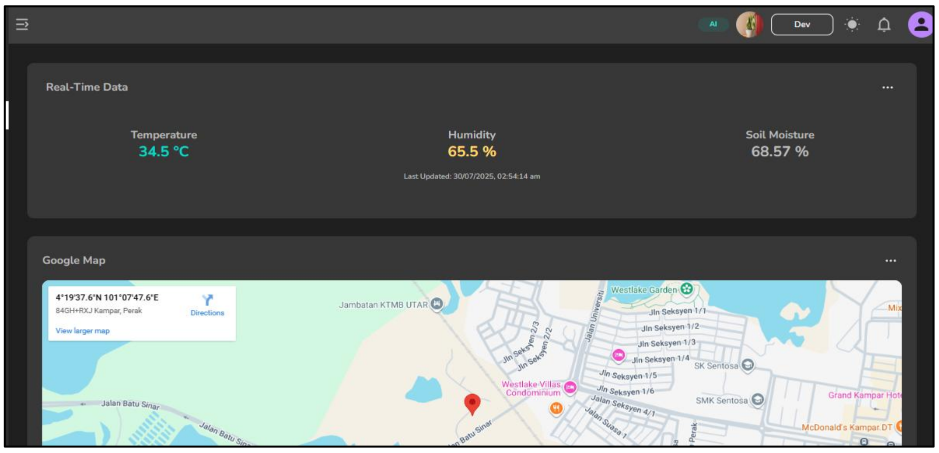
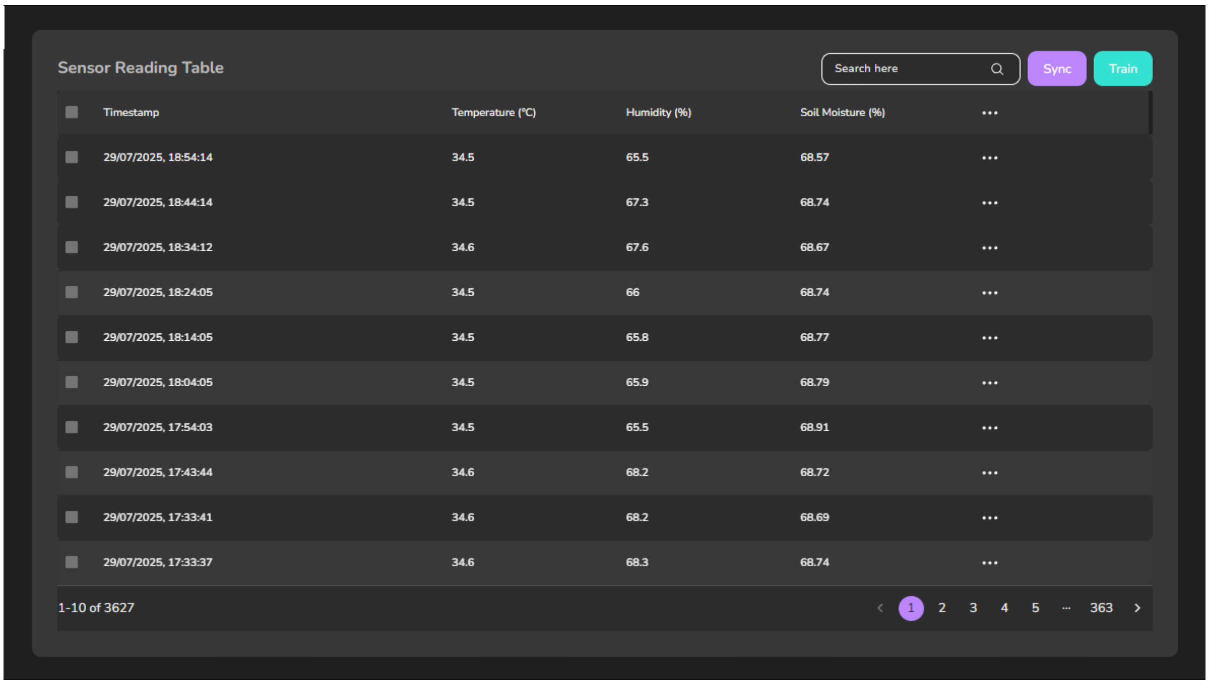


Figure 53: Dashboard Overview in Developer Mode View 1



Sensor Reading Table

Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	···
29/07/2025, 18:54:14	34.5	65.5	68.57	···
29/07/2025, 18:44:14	34.5	67.3	68.74	···
29/07/2025, 18:34:12	34.6	67.6	68.67	···
29/07/2025, 18:24:05	34.5	66	68.74	···
29/07/2025, 18:14:05	34.5	65.8	68.77	···
29/07/2025, 18:04:05	34.5	65.9	68.79	···
29/07/2025, 17:54:03	34.5	65.5	68.91	···
29/07/2025, 17:43:44	34.6	68.2	68.72	···
29/07/2025, 17:33:41	34.6	68.2	68.69	···
29/07/2025, 17:33:37	34.6	68.3	68.74	···

1-10 of 3627

Figure 54: Dashboard Overview in Developer Mode View 2

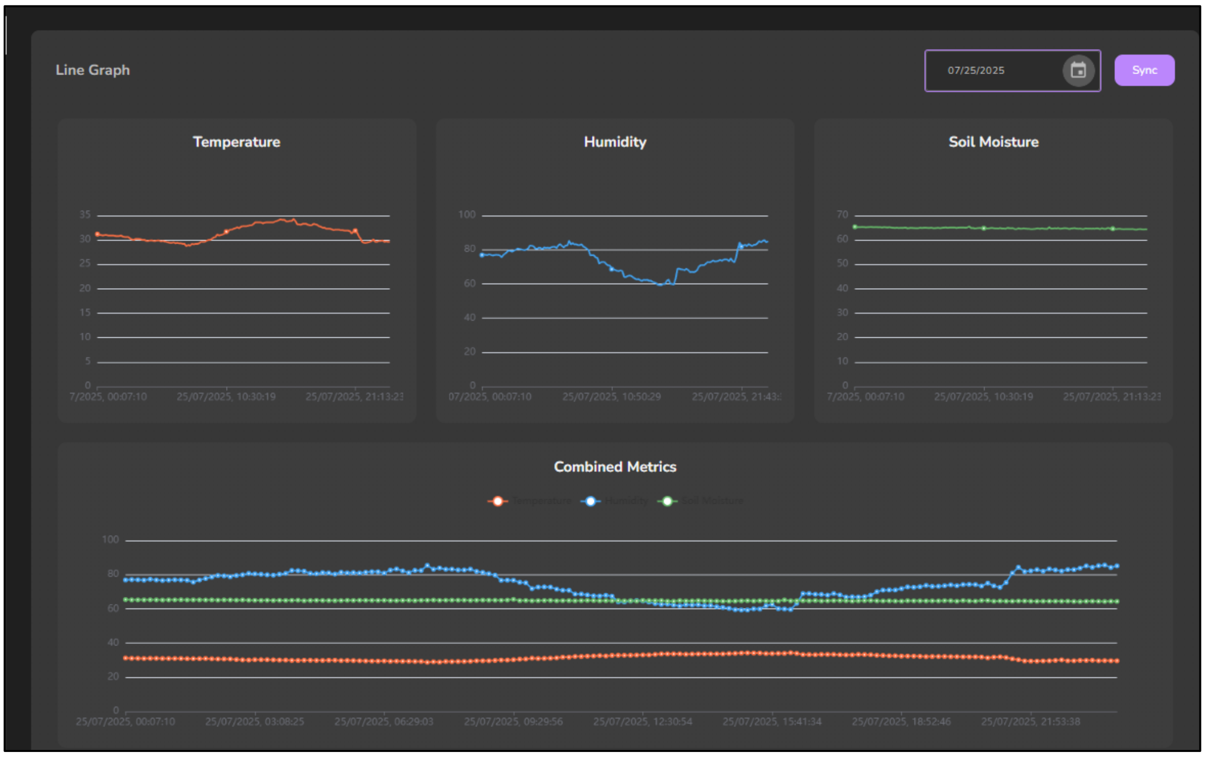
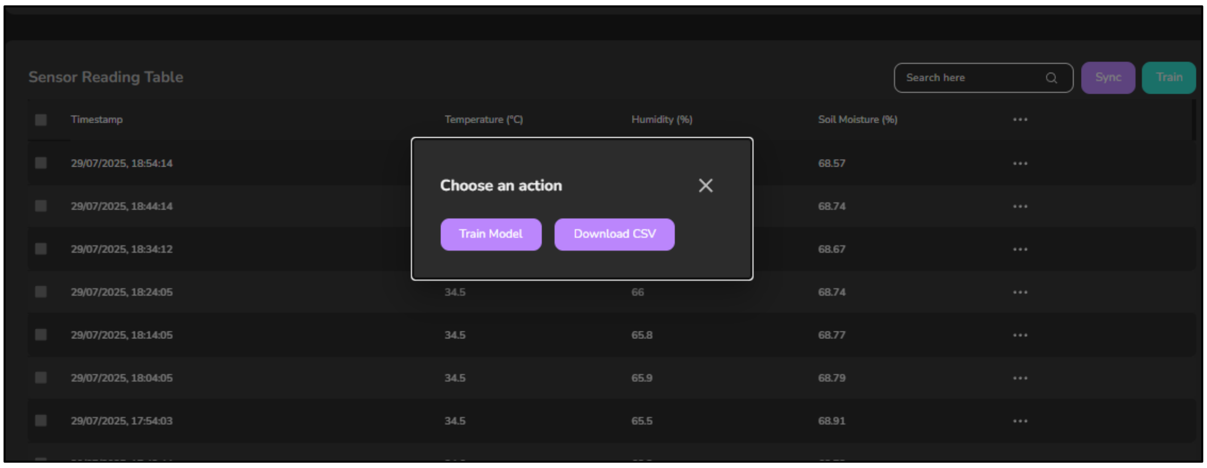


Figure 55: Dashboard Overview in Developer Mode View 3

The purpose of having developer mode view for users is to enable advance feature for data collection and model training. When the system is in developer mode, it will trigger the microcontroller into developer mode too where it will send temperature, humidity and soil moisture to the backend for data collection and further model training. The developer mode will on for 14 days and automatically closed as datapoints for 14 days data collection should be enough to train a strong ai model. Figure 55 above shows the data visualization features in the system. Users able to visualize the line chart for each temporal data analysis, real-time gauge displays, historical trend analysis and customizable time range selection. The button of "sync" is used to refresh the data and fetch the newest data. Figure 54 above shows the historical data for the microcontroller collected data and there is a button "train" to train the data. Figure 56 below shows the overview of the train model page when the "train" button clicked.


Sensor Reading Table

Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	•••
29/07/2025, 18:54:14			68.57	•••
29/07/2025, 18:44:14			68.74	•••
29/07/2025, 18:34:12			68.67	•••
29/07/2025, 18:24:05	34.5	66	68.74	•••
29/07/2025, 18:14:05	34.5	65.8	68.77	•••
29/07/2025, 18:04:05	34.5	65.9	68.79	•••
29/07/2025, 17:54:03	34.5	65.5	68.91	•••

Figure 56: Train Button Clicked Overview

When the "train" button clicked by users, there will be two options which is train model and download csv. If users click download csv, the system will generate a csv file and download to the users' device as shown at figure 57 below. While if the users click "train model" button, another popup modal will show to users to choose which plant model users want to train using the account's csv data as shown at figure 58 below.

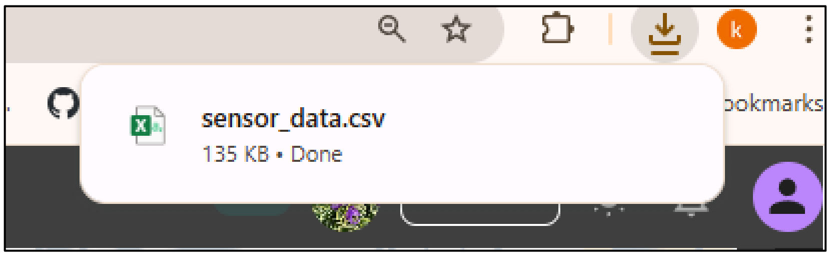


Figure 57: Download CSV Button Clicked Overview

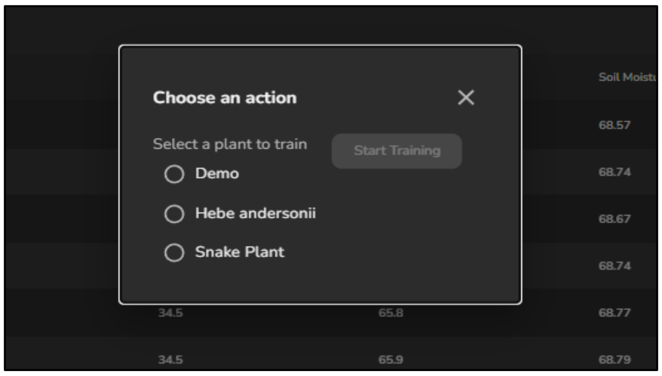


Figure 58: Train Model Button Clicked Overview

When users clicked "train model" button, the system will show current available plant option for users to choose to train their own plant ai model using their collected data. As the current account, the data is collected by using snake plant, so users will need to choose snake plant and click the "start training" button. Figure 59 and 60 below shows the response of the frontend. Figure 57 below shows the users that the ai model is in training process and loading. Figure 58 below shows the result of the successful trained ai model.

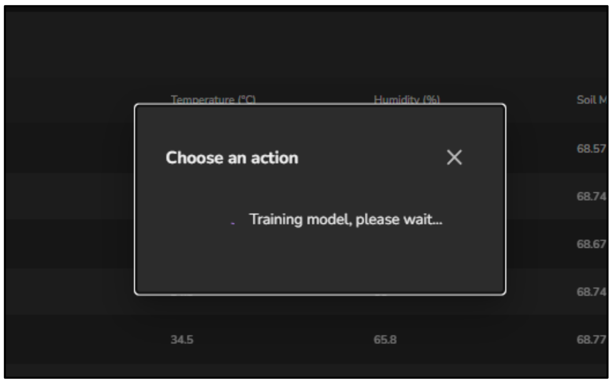
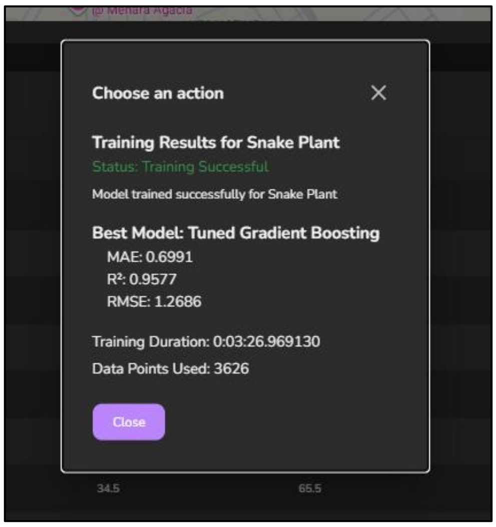


Figure 59: Start Training Button Clicked Overview


Choose an action

Training Results for Snake Plant

Status: Training Successful

Model trained successfully for Snake Plant

Best Model: Tuned Gradient Boosting

MAE: 0.6991

R²: 0.9577

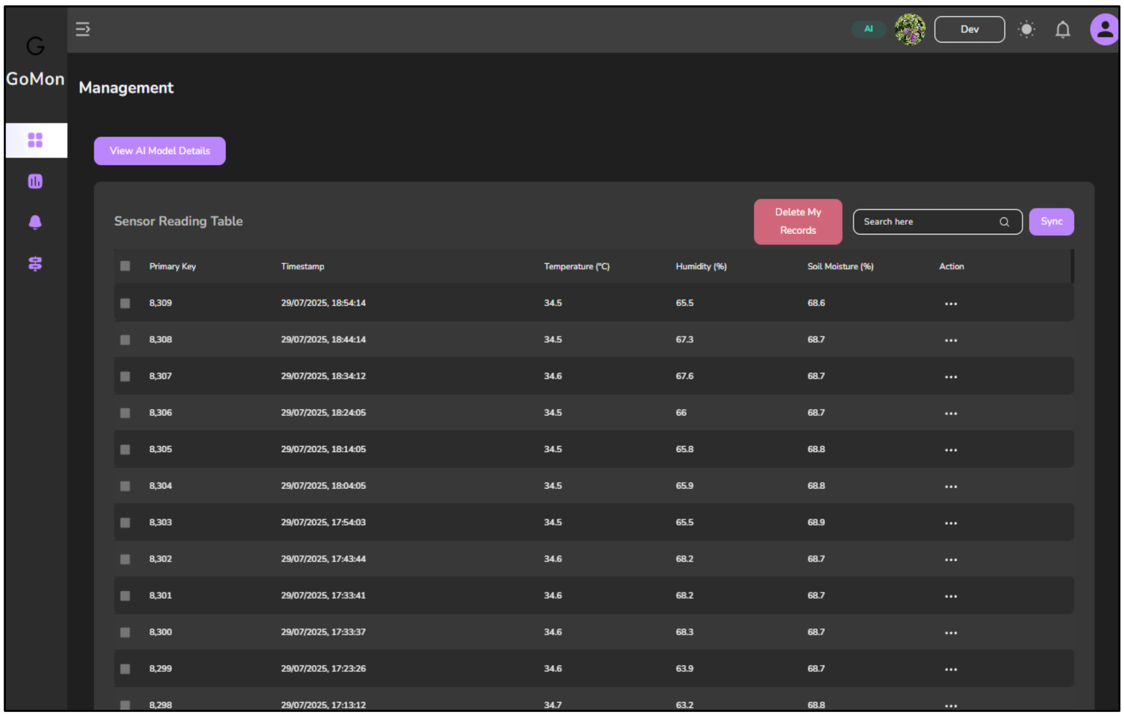
RMSE: 1.2686

Training Duration: 0:03:26.969130

Data Points Used: 3626

Figure 60: AI Model Successful Trained Overview

5.4.8 Data Management Operation

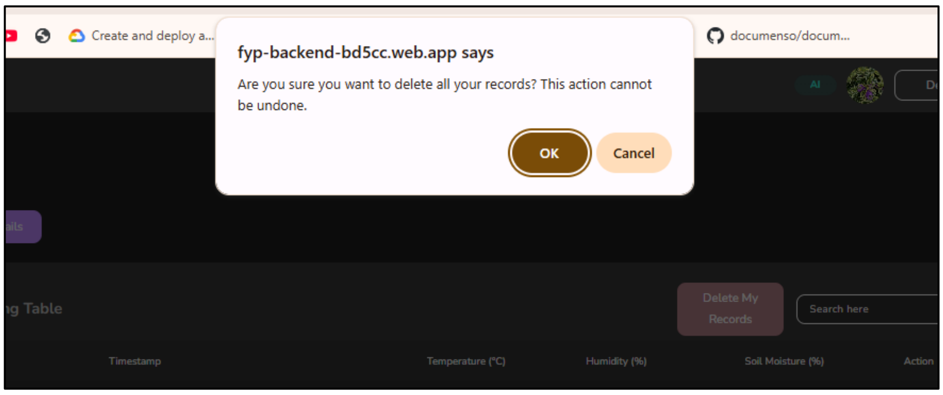


Sensor Reading Table

Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action
8,309	29/07/2025, 18:54:14	34.5	65.5	68.6	...
8,308	29/07/2025, 18:44:14	34.5	67.3	68.7	...
8,307	29/07/2025, 18:34:12	34.6	67.6	68.7	...
8,306	29/07/2025, 18:24:05	34.5	66	68.7	...
8,305	29/07/2025, 18:14:05	34.5	65.8	68.8	...
8,304	29/07/2025, 18:04:05	34.5	65.9	68.8	...
8,303	29/07/2025, 17:54:03	34.5	65.5	68.9	...
8,302	29/07/2025, 17:43:44	34.6	68.2	68.7	...
8,301	29/07/2025, 17:33:41	34.6	68.2	68.7	...
8,300	29/07/2025, 17:33:37	34.6	68.3	68.7	...
8,299	29/07/2025, 17:23:26	34.6	63.9	68.7	...
8,298	29/07/2025, 17:13:12	34.7	63.2	68.8	...

Figure 61: Data Management Overview Interface

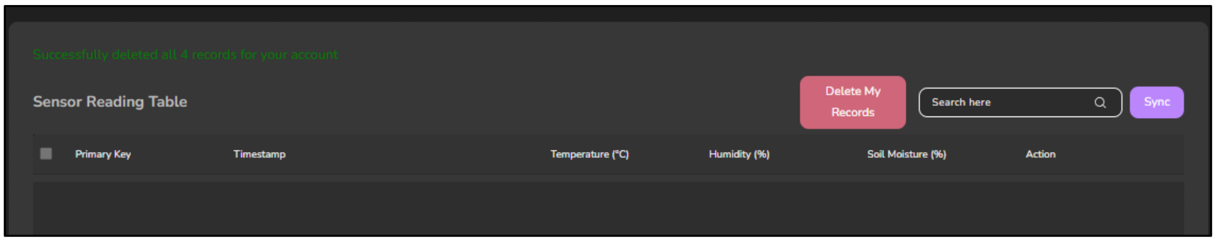
Besides real-time monitoring, this system also supports data management features. When users click on the side bar management page, the frontend will direct users to this data management page as shown as figure 61 above. When users click on the "Delete My Records", the system will delete all the records for current account. Figures 62 and 63 below show the results.



fyp-backend-bd5cc.web.app says

Are you sure you want to delete all your records? This action cannot be undone.

Figure 62: Confirmation of Delete All Records

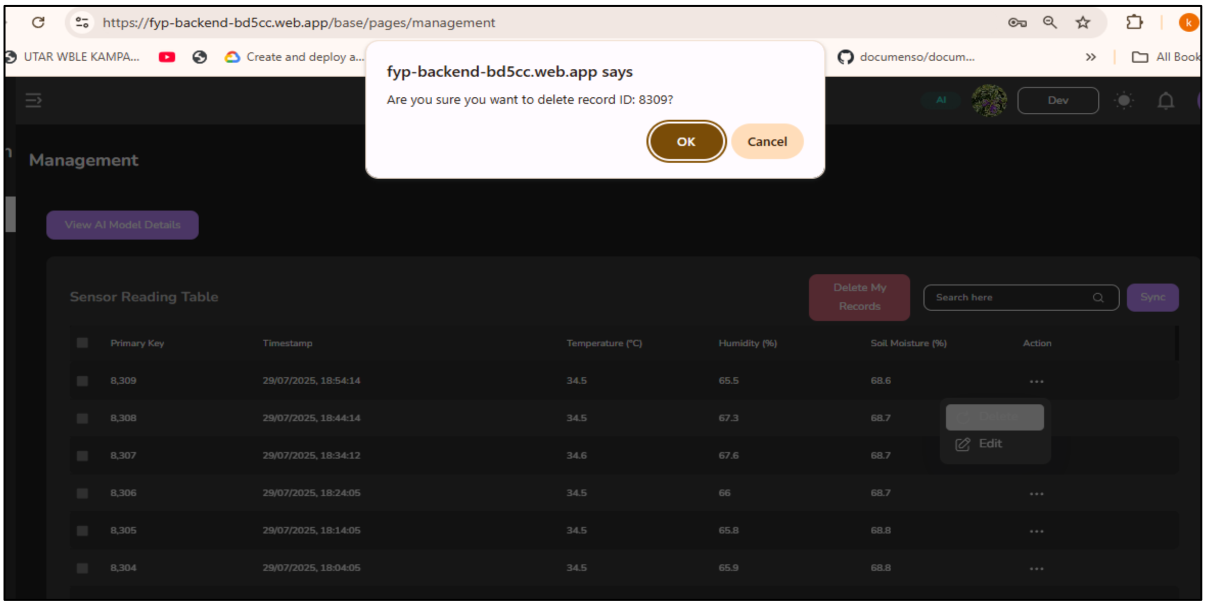


Successfully deleted all 4 records for your account

Sensor Reading Table

Figure 63: All Records Successful Deleted Overview

Instead of deleting all records, the system also allows users to delete the selected data or modify it. Figure 64 below will show how to delete the selected data. Users can click on the Action (…) to choose to delete or edit for the selected data. When users click on the delete button, it will prompt a message to users to confirm performing delete for the selected data with showing record ID.


https://fyp-backend-bd5cc.web.app/base/pages/management

fyp-backend-bd5cc.web.app says

Are you sure you want to delete record ID: 8309?

OK Cancel

Management

View AI Model Details

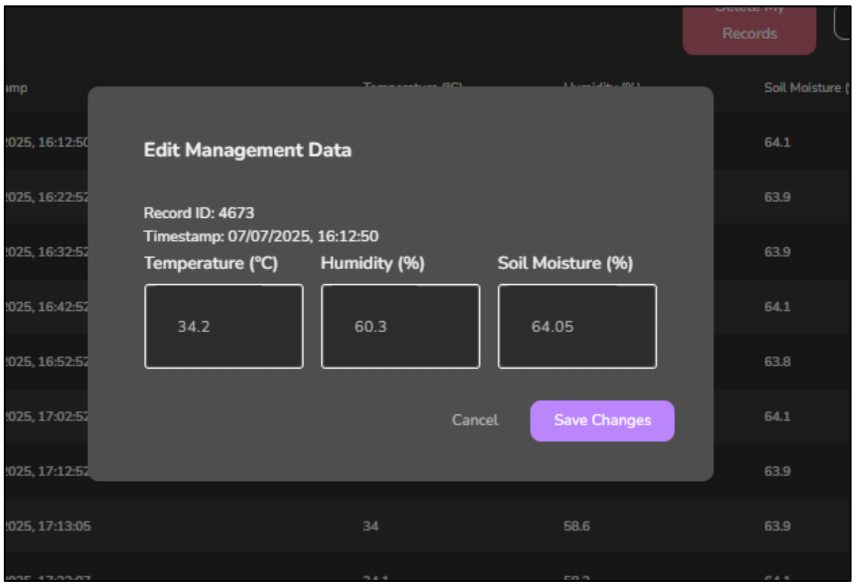
Sensor Reading Table

Delete My Records | Search here | Sync

Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action
8,309	29/07/2025, 18:54:14	34.5	65.5	68.6	···
8,308	29/07/2025, 18:44:14	34.5	67.3	68.7	Delete
8,307	29/07/2025, 18:34:12	34.6	67.6	68.7	Edit
8,306	29/07/2025, 18:24:05	34.5	66	68.7	···
8,305	29/07/2025, 18:14:05	34.5	65.8	68.8	···
8,304	29/07/2025, 18:04:05	34.5	65.9	68.8	···

Figure 64: Selected Records Delete Overview

While for the users to edit the data, figure 65 below shows the overview of the edit data page when users clicked on edit for selected data. There is a popup modal that came out when users clicked on edit button. Users are allowed to modify the temperature, humidity and soil moisture value. When it is modifieds just click on the save changes button and the data will be saved.


Edit Management Data

Record ID: 4673
Timestamp: 07/07/2025, 16:12:50

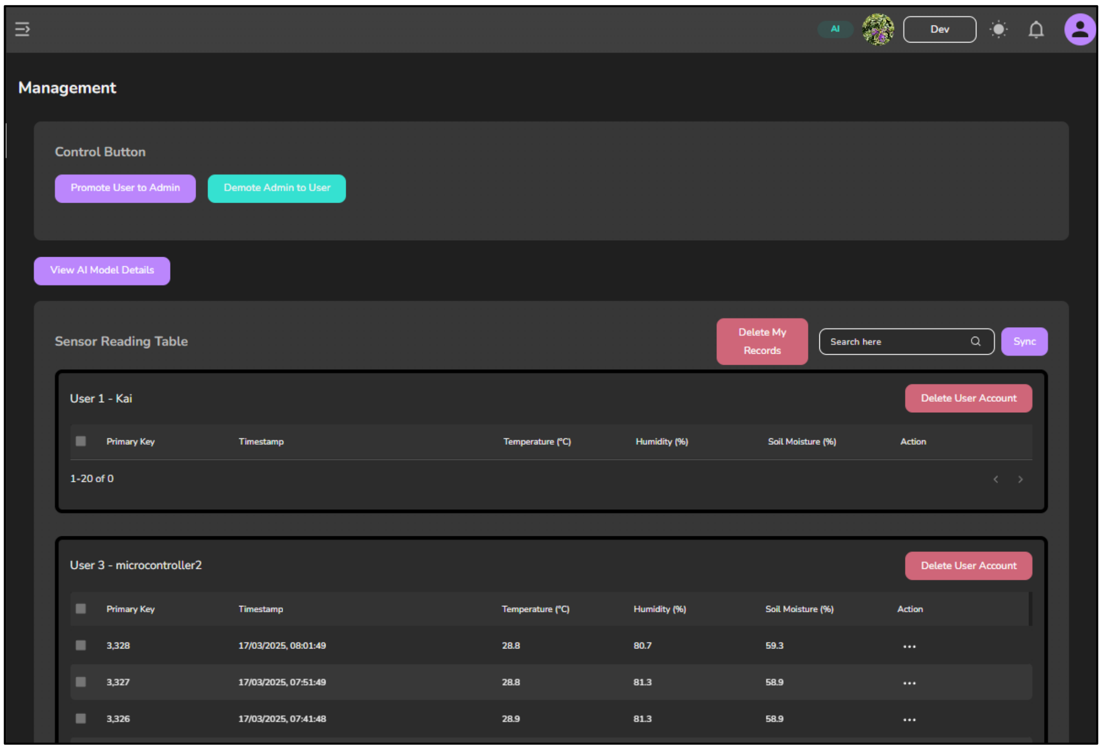
Temperature (°C)	Humidity (%)	Soil Moisture (%)
34.2	60.3	64.05

Cancel | Save Changes

Figure 65: Selected Records Edit Overview

5.4.9 System Administration and Settings

This system is also designed with administrative features. For example, there is user permission management that was divided into user and admin role. Figures 66 and 67 will show the difference between user and admin role. For admin roles, it able to see all available accounts' data collection table, have permission to edit, delete them and able to perform functionality like promote or demote user role to admin role or delete the entire account.



Management

Control Button

Promote User to Admin | Demote Admin to User

View AI Model Details

Sensor Reading Table

Delete My Records | Search here | Sync

User 1 - Kai

Delete User Account

	Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action

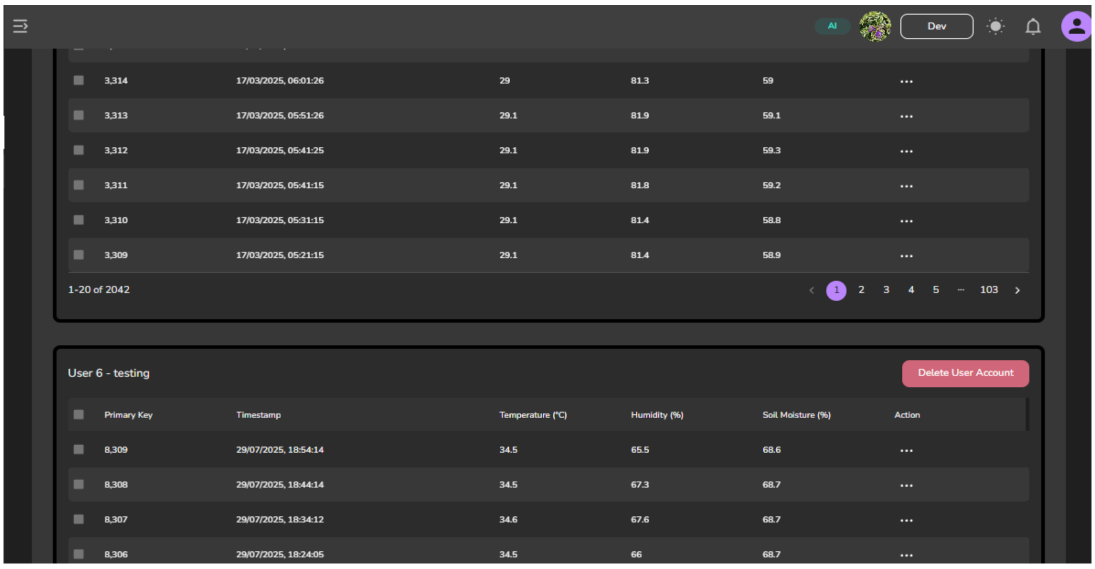
1-20 of 0

User 3 - microcontroller2

Delete User Account

	Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action
■	3,328	17/03/2025, 08:01:49	28.8	80.7	59.3	...
■	3,327	17/03/2025, 07:51:49	28.8	81.3	58.9	...
■	3,326	17/03/2025, 07:41:48	28.9	81.3	58.9	...

Figure 66: Management Page Overview for Admin Part 1



■	3,314	17/03/2025, 06:01:26	29	81.3	59	...
■	3,313	17/03/2025, 05:51:26	29.1	81.9	59.1	...
■	3,312	17/03/2025, 05:41:25	29.1	81.9	59.3	...
■	3,311	17/03/2025, 05:41:15	29.1	81.8	59.2	...
■	3,310	17/03/2025, 05:31:15	29.1	81.4	58.8	...
■	3,309	17/03/2025, 05:21:15	29.1	81.4	58.9	...

1-20 of 2042 ‹ 1 2 3 4 5 ··· 103 ›

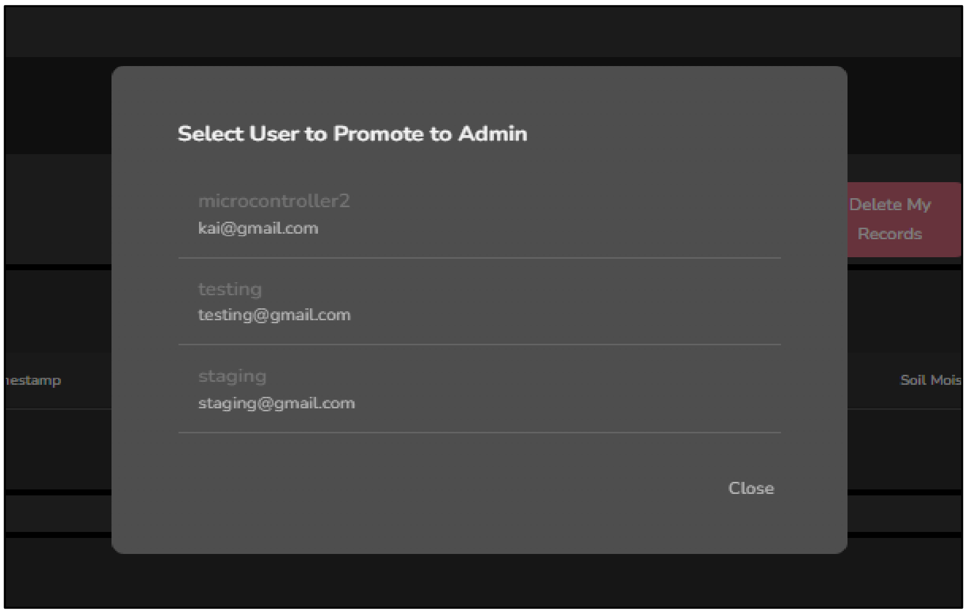
User 6 - testing

Delete User Account

	Primary Key	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)	Action
■	8,309	29/07/2025, 18:54:14	34.5	65.5	68.6	...
■	8,308	29/07/2025, 18:44:14	34.5	67.3	68.7	...
■	8,307	29/07/2025, 18:34:12	34.6	67.6	68.7	...
■	8,306	29/07/2025, 18:24:05	34.5	66	68.7	...

Figure 67: Management Page Overview for Admin Part 2

When the promote user to admin button is clicked, a popup modal will show current available user role account to admin to choose which account admin want to promote. When admin clicks on it, the clicked account will become an admin role. Figure 68 below shows the overview of the results.



Select User to Promote to Admin

microcontroller2
kai@gmail.com

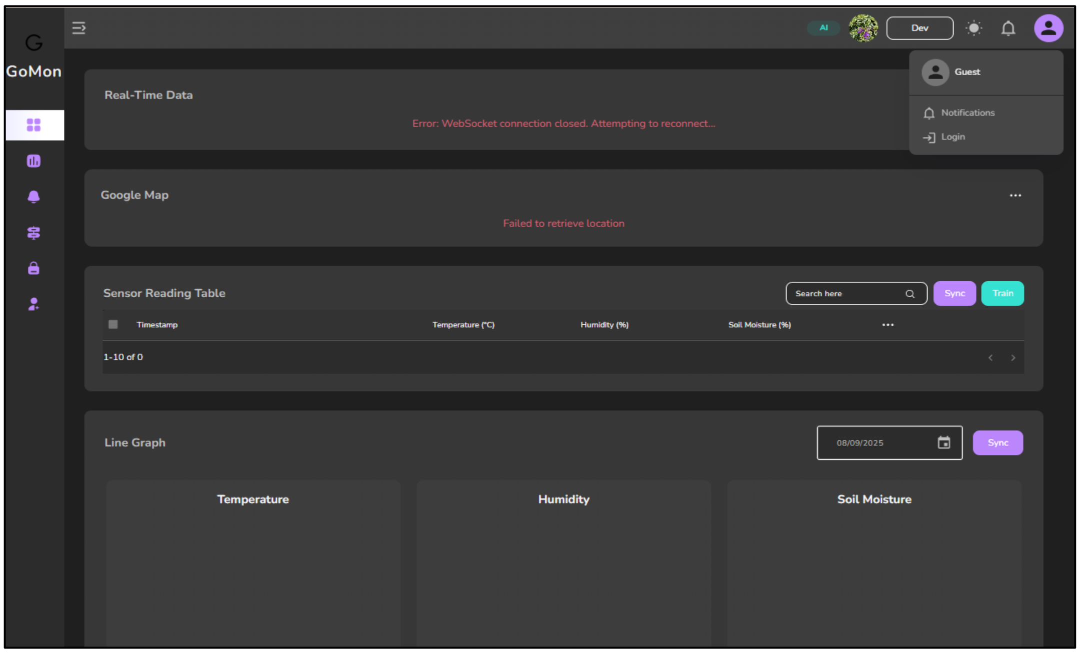
testing
testing@gmail.com

staging
staging@gmail.com

Figure 68: Promote User to Admin Overview

5.4.10 Error Handling and Validation

Figure 69 below shows the response from the front end when it is in guest mode, empty data will be read and showing errors on the main dashboard. Figure 70 below shows the response from frontend when guest wanted to check available ai model list at the management page.



Real-Time Data

Error: WebSocket connection closed. Attempting to reconnect...

Google Map

Failed to retrieve location

Sensor Reading Table

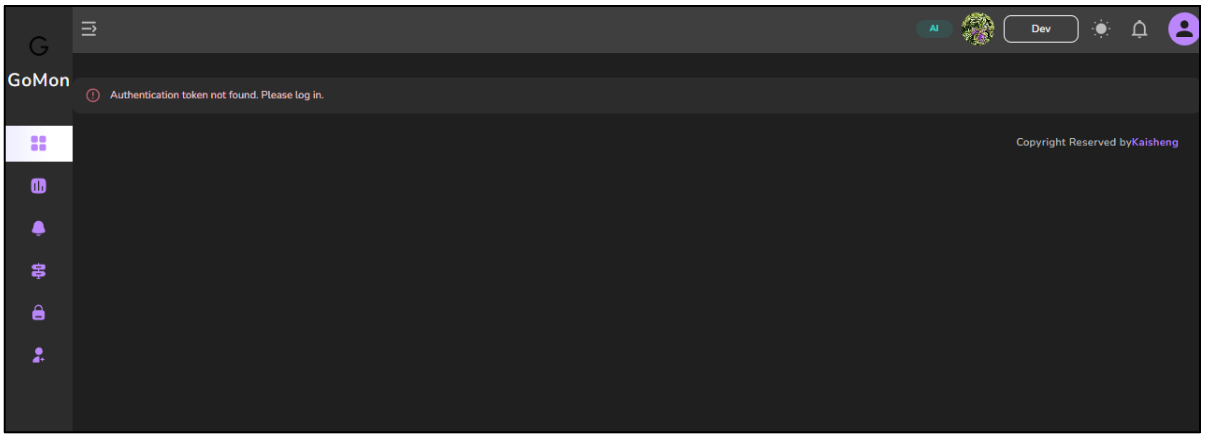
	Timestamp	Temperature (°C)	Humidity (%)	Soil Moisture (%)

1-10 of 0

Line Graph

08/09/2025

Figure 69: Main Dashboard Overview in Guest

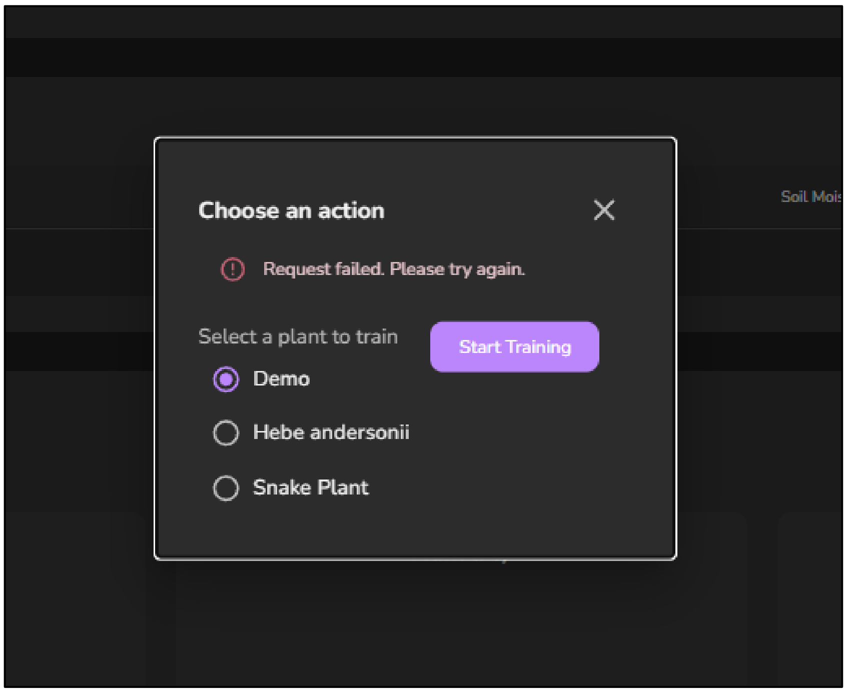


Authentication token not found. Please log in.

Copyright Reserved byKaisheng

Figure 70: AI Model List Overview in Guest

Besides, when users try to train an ai model using an empty data account, it will show request failure as there is not enough data in the csv file, and it will be rejected by the system. Figure 71 below shows the response result.



Choose an action

Request failed. Please try again.

Select a plant to train

Start Training

[✓ Demo]

Hebe andersonii

Snake Plant

Figure 71: Train Model Rejected Overview

5.5 Implementation of Issues and Challenges

5.5.1 Hardware and Firmware Integration

The physical layer, centred around the ESP32 microcontroller, posed the initial set of hurdles. First is the sensor reliability and power management problem. While the project aims to replace soil moisture sensors, the initial data collection phase relied on them heavily. Ensuring the capacitive soil moisture sensor provided consistent and accurate readings was challenging. Furthermore, implementing robust power management and a watchdog timer on the ESP32 was necessary to ensure long-term stability and prevent system freezes, requiring multiple code revisions to achieve reliable, autonomous operation.

Another hurdle is the Wi-Fi connectivity and data transmission. Establishing stable Wi-Fi communication between the ESP32 and the backend was a significant challenge. The firmware needed sophisticated error handling, including an automatic reconnection mechanism with an exponential backoff, to manage intermittent network disruptions. Data was formatted as JSON and sent via HTTPS POST requests, and ensuring the lightweight ESP32 could handle the overhead of TLS encryption and JSON serialization without performance degradation required careful optimization

5.5.2 Backend and Database Complexity

The backend, which is built with Go and PostgreSQL, serves as the system's core and presents its own integration and performance challenges. The challenges included concurrent data handling and real-time communication. The backend needed to manage simultaneous HTTP requests from the IoT device and WebSocket connections from multiple web clients. Implementing a thread-safe system in Go to handle state changes, such as switching between "developer" and "user" modes, required careful use of concurrency patterns (goroutines and mutex locks) to prevent race conditions.

Besides, there is cross-service integration needed for the backend. The Go backend had to communicate seamlessly with three external services which is the PostgreSQL database hosted on Azure, the flask ai server for soil moisture prediction and the google geolocation api. Managing authentication tokens, API keys, a different data formats across these services was complex. Ensuring low latency, especially when the Go service had to wait for a prediction from the Flask API before responding to the client, required efficient code and network configurations.

Automating the deployment of the Go backend and flask ai service to google cloud run using a CI/CD pipeline was a significant undertaking. Writing the cloudbuild.yaml and dockerfiles, managing environment variables securely, and configuring firewall rules for the Azure database to accept connections from google cloud services were intricate and error-prone steps.

5.5.3 AI Model Development and Deployment

The AI component was the most innovative but also one of the most complex parts of the project. The success of the ai model depend entirely on the quality of the data collected in "developer mode". The key challenge was in feature engineering which will transform the raw data (temperature, humidity, timestamp) into meaning features that could predict the soil moisture value accurately. This involved creating cyclical features for time, interaction features, and rolling averages. The initial models had lower accuracy that required experimentation with hyperparameter tuning for both the random forest and gradient boosting models to achieve the desired performance which is (R2 score > 0.9).

Dynamic model loading and scalability is also a challenge for the ai model development. The Flask API was designed to dynamically load different trained models based on the plant type (e.g., Hebe andersonii, Snake Plant) specified in the API request. Implementing this required a structured folder system for the model files (.pkl) and a robust loading mechanism. Containerizing this flask application with docker and deploying it on a serverless platform like cloud run was necessary to ensure it could scale independently of the main Go backend.

5.5.4 Frontend and User Experience

The React and TypeScript frontend aimed to provide a user-friendly interface but faced challenges in handling real-time data and managing complex application states. Implementing a performant dashboard that could visualize a continuous stream of data via WebSockets without causing the user interface to lag was a primary challenge. This required efficient state management using React hooks and careful rendering logic to ensure smooth updates for charts and gauges. Managing user authentication with JWT tokens and dynamically changing the user interface based on the user's role (user vs. admin) and the device's mode (developer vs. user) added significant complexity to the frontend state. Ensuring consistent and secure user experience across different states and modes requires a well-designed component architecture and rigorous testing.

5.6 Concluding Remark

Chapter 5 has detailed the complete journey of transforming the conceptual design of the "Sensors Substitution using AI" system into a tangible, operational prototype. The implementation phase successfully integrated a diverse stack of modern technologies, from low-level firmware on the ESP32 microcontroller to sophisticated cloud-native services. The detailed walkthrough of the hardware setup, software tools, and precise configurations demonstrates the successful creation of a cohesive and functional ecosystem.

The system's operation was brought to life, showcasing a seamless user experience through the GoMonitor web application. Key functionalities are including secure user authentication, real-time data streaming via WebSocket, and the dynamic switching between User Mode and Developer Mode were all successfully implemented and demonstrated. A significant achievement was the operationalization of the AI model training pipeline, allowing users to initiate the training of a machine learning model directly from the web interface using their own collected data.

Despite the successes, this chapter also acknowledged the significant implementation challenges that were overcome. These ranged from ensuring hardware reliability and stable connectivity to managing the complexities of a microservices-based backend architecture involving Go, Python, and multiple cloud platforms. The successful resolution of these issues underscores the robustness of the final implementation.

In summary, this chapter confirms that the project has progressed from architectural diagrams to a fully functional, end-to-end system. The hardware is collecting data, the backend is processing it, the AI is making predictions, and the frontend is providing an intuitive interface for users to interact with it all. The system now stands as a proof-of-concept, ready for the rigorous evaluation and performance analysis that will be detailed in Chapter 6.

Chapter 6
System Evaluation and Discussion

6.1 System Testing and Performance Metrics

This section describes how the system was tested to make sure it is working correctly and efficiently. The focus of the testing was on the backend API, which is the core of the entire system. The testing method was to send HTTP requests to the deployed API endpoints and then check the responses. This was done to measure two key things: the correctness of the output and the performance of the server. The API testing was performed using the live backend URL: https://go-backend-server-738775168875.asia-southeast1.run.app.

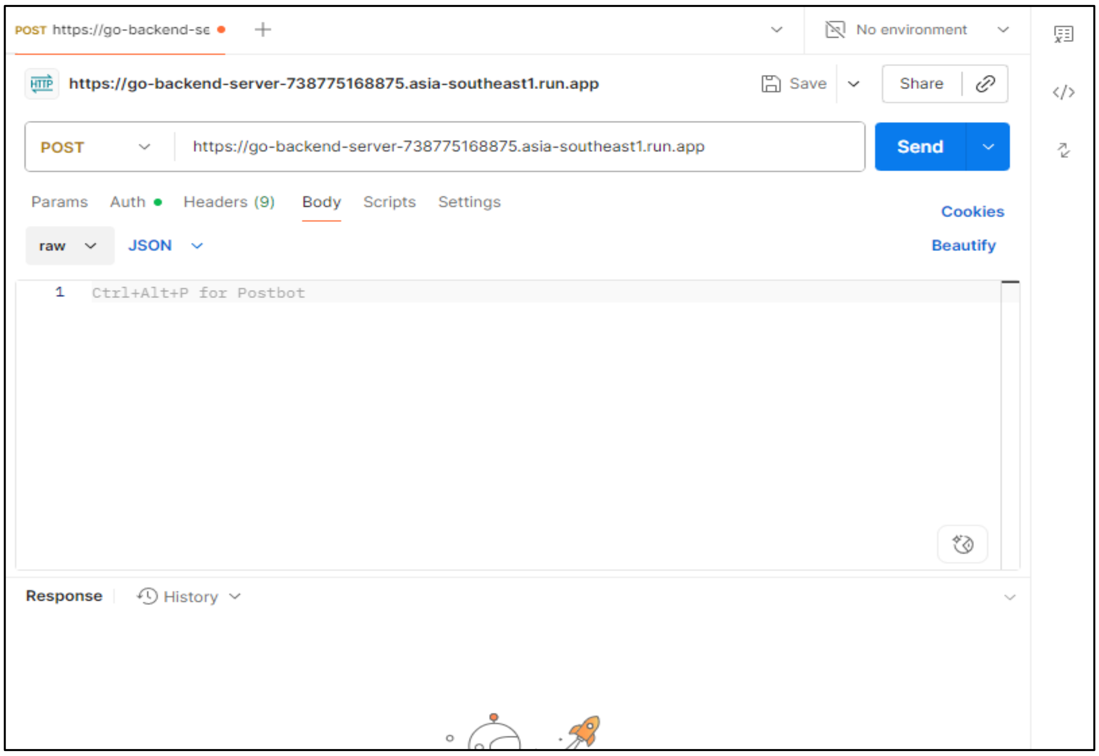
For each important API route, a request was sent, and the following metrics were recorded which is the correctness of output and response time. Correctness of output was the most important test. For every API request, the response from the server was checked to see if it was correct. For example, when using the /login route, the test checked if a valid user received a success message and a token. When using the /sensor-data route, the test checked if the data was saved in the database. This confirms that the system's logic is working as expected.

Besides, the response time measures how fast the system is. The response time is the duration from when an API request is sent to when the server sends back a complete response. This was measured in milliseconds (ms). A fast response time means the system is efficient and provides good user experience. The average response time will be measured by sending 10 requests at the same time using postman. By using these two metrics, it was possible to get a clear picture of both the functionality and performance of the system.

6.2 Testing Setup and Result

6.2.1 Testing Setup

This section describes the setup used for testing and present the results that were collected. For the testing setup, all tests were performed on the deployed system to simulate a real-world usage environment. The hardware used to send the API request is legion 5 laptop. The computer processor is Intel Core i7-12700H and memory is 16GB DDR5 RAM. While the software used for testing is Postman. Postman is used to create and send the HTTP request and to view the responses. The backend Go application was deployed and running on Google Cloud Run, and the database was hosted on Azure PostgreSQL. Figure 72 below shows the setup of the postman for testing. The http request must send to the live backend URL: https://go-backend-server-738775168875.asia-southeast1.run.app and add /end-point at the end of the URL.


POST https://go-backend-se

No environment

https://go-backend-server-738775168875.asia-southeast1.run.app

Save Share

POST https://go-backend-server-738775168875.asia-southeast1.run.app Send

Params Auth Headers (9) Body Scripts Settings Cookies

raw JSON Beautify

Ctrl+Alt+P for Postbot

Response History

Figure 72: Postman Setup

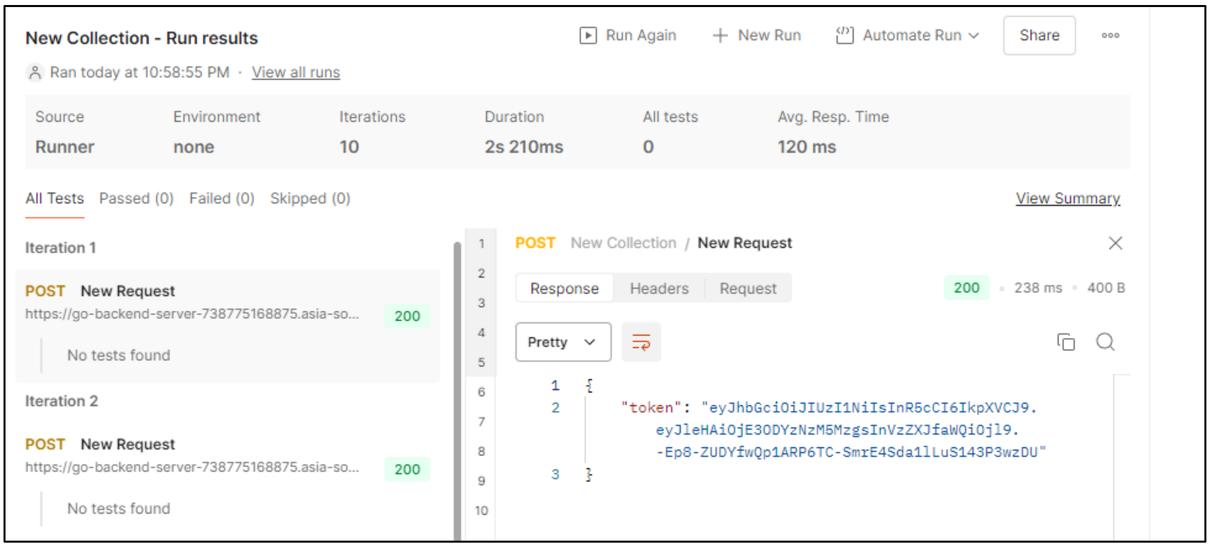
6.2.2 Testing Result

The table 6.1 below shows the result for the api endpoints testing results. Each test was run 10 times to get an average response time.

Table 6.1 Testing Result for API Endpoints

API Endpoint	HTTP Method	Description	Expected Outcome	Actual Outcome	Average Response Time
/login	POST	Authenticates a user with correct credentials.	Returns a JWT token.	Success. Received token.	120ms
/sensor-data (with AI)	POST	Receives and saves sensor data (temperature & humidity) from the ESP32 and sends to AI model for prediction.	Returns a success message. Data saved in the database.	Success. Data saved correctly.	97ms
/sensor-data (without AI)	POST	Receives and saves sensor data (temperature, humidity & soil moisture) from the ESP32	Returns a success message. Data saved in the database.	**Success.** Data saved correctly.	52ms

/history	GET	Fetches the historical sensor data for a user.	Returns a JSON array of all sensor data records.	Success. Received all records.	62ms
/device-config/esp32-001	GET	Fetch the current mode of esp32 (user or developer).	Returns a JSON with developer mode start time if true and details.	Success. Received the details.	57ms
/device-config/esp32-001/trigger-dev	POST	Enables developer mode for the ESP32.	Returns a success message. Device mode is updated.	Success. Mode updated.	71ms
/device-config/esp32-001/stop-dev	POST	disables developer mode for the ESP32.	Returns a success message. Device mode is updated.	Success. Mode updated.	67ms
/abnormal-history	GET	Fetches the abnormal historical sensor data for a user.	Returns a JSON array of all abnormal sensor data records.	Success. Received all records.	41ms
/delete/:id	DELETE	Delete the sensor data according to id.	Returns a success message. Record deleted.	Success. Record deleted	34ms
/models	GET	Lists all available trained AI models.	Returns a JSON array with details of each AI model.	Success. Received model list.	45ms
/train-model	POST	Starts the AI model training process with user data.	Returns a message that training start successfully and receive the training result	Success. Training started.	154157 ms



New Collection - Run results

Run Again | New Run | Automate Run | Share

Ran today at 10:58:55 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	2s 210ms	0	120 ms

All Tests | Passed (0) | Failed (0) | Skipped (0) | View Summary

Iteration 1

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 2

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

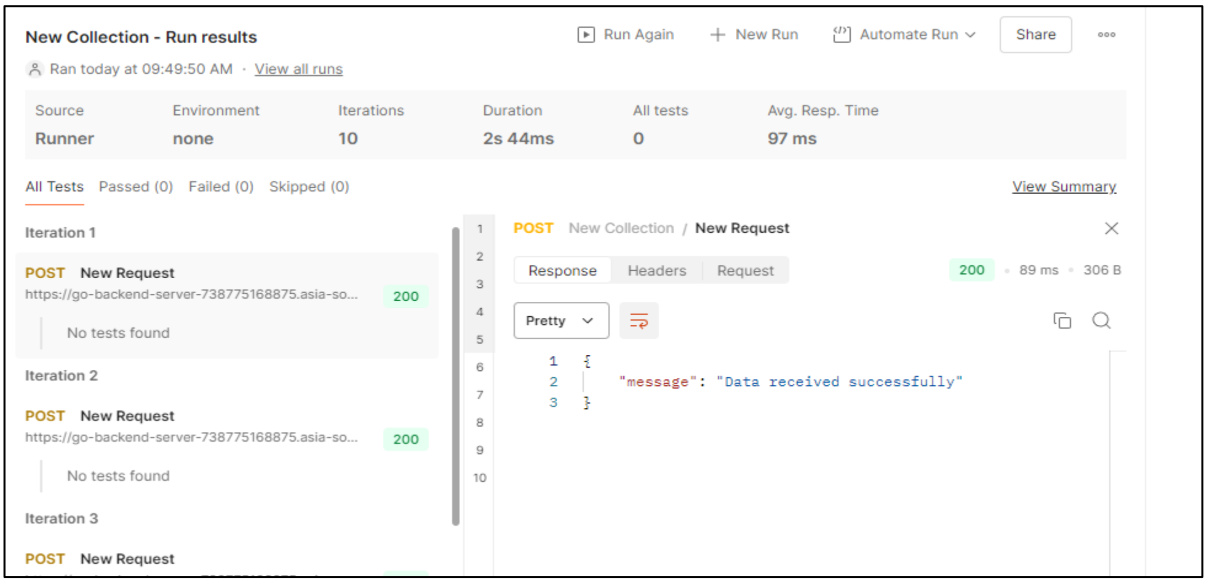
POST New Collection / New Request

Response | Headers | Request — 200 · 238 ms · 400 B

Pretty

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJleHAiOjE3ODYzNzM5MzgsInVzZXJfaWQiOjl9.
        -Ep8-ZUDYfwQp1ARP6TC-SmrE4Sda1lLuS143P3wzDU"
}
```

Figure 73: Testing Result for POST /login



New Collection - Run results

Run Again | New Run | Automate Run | Share

Ran today at 09:49:50 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	2s 44ms	0	97 ms

All Tests | Passed (0) | Failed (0) | Skipped (0) | View Summary

Iteration 1

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 2

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 3

POST New Request

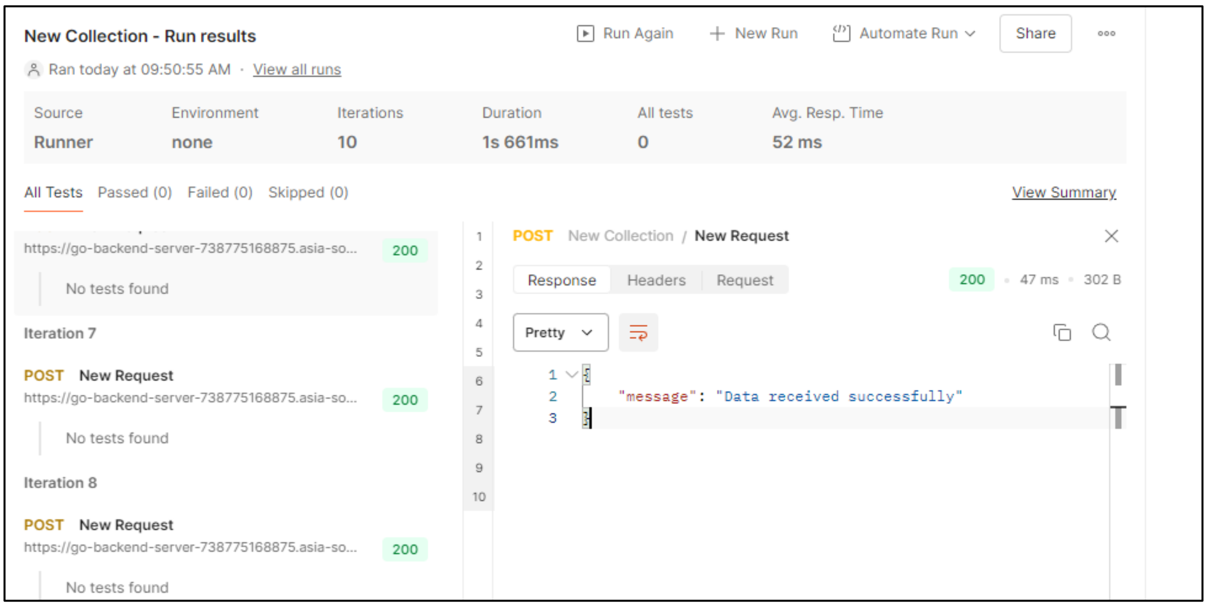
POST New Collection / New Request

Response | Headers | Request — 200 · 89 ms · 306 B

Pretty

```
{
    "message": "Data received successfully"
}
```

Figure 74: Testing Result for POST /sensor-data (with AI)



New Collection - Run results

Run Again | New Run | Automate Run | Share

Ran today at 09:50:55 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 661ms	0	52 ms

All Tests | Passed (0) | Failed (0) | Skipped (0) | View Summary

https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 7

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 8

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

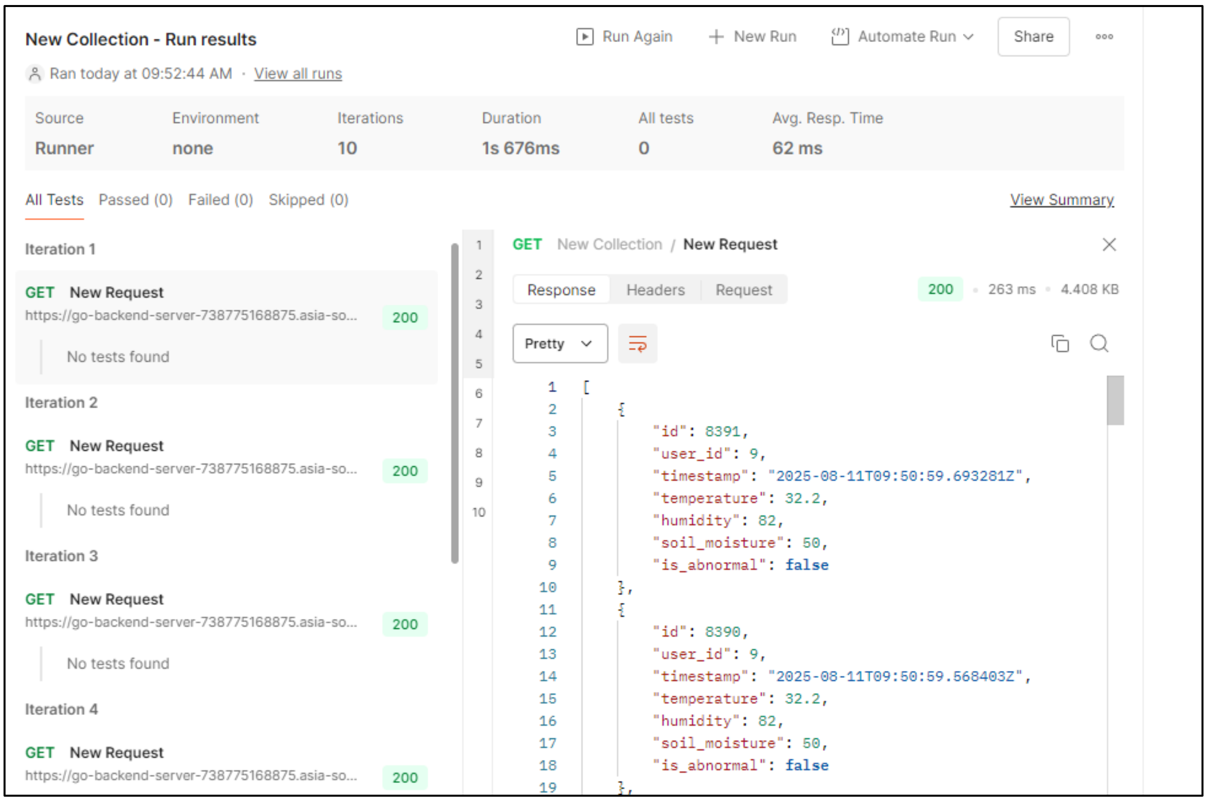
POST New Collection / New Request

Response | Headers | Request — 200 · 47 ms · 302 B

Pretty

```
{
    "message": "Data received successfully"
}
```

Figure 75: Testing Result for POST /sensor-data (without AI)


New Collection - Run results

Run Again · New Run · Automate Run · Share

Ran today at 09:52:44 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 676ms	0	62 ms

All Tests · Passed (0) · Failed (0) · Skipped (0) · View Summary

Iteration 1
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 2
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 3
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 4
GET New Request https://go-backend-server-738775168875.asia-so... 200

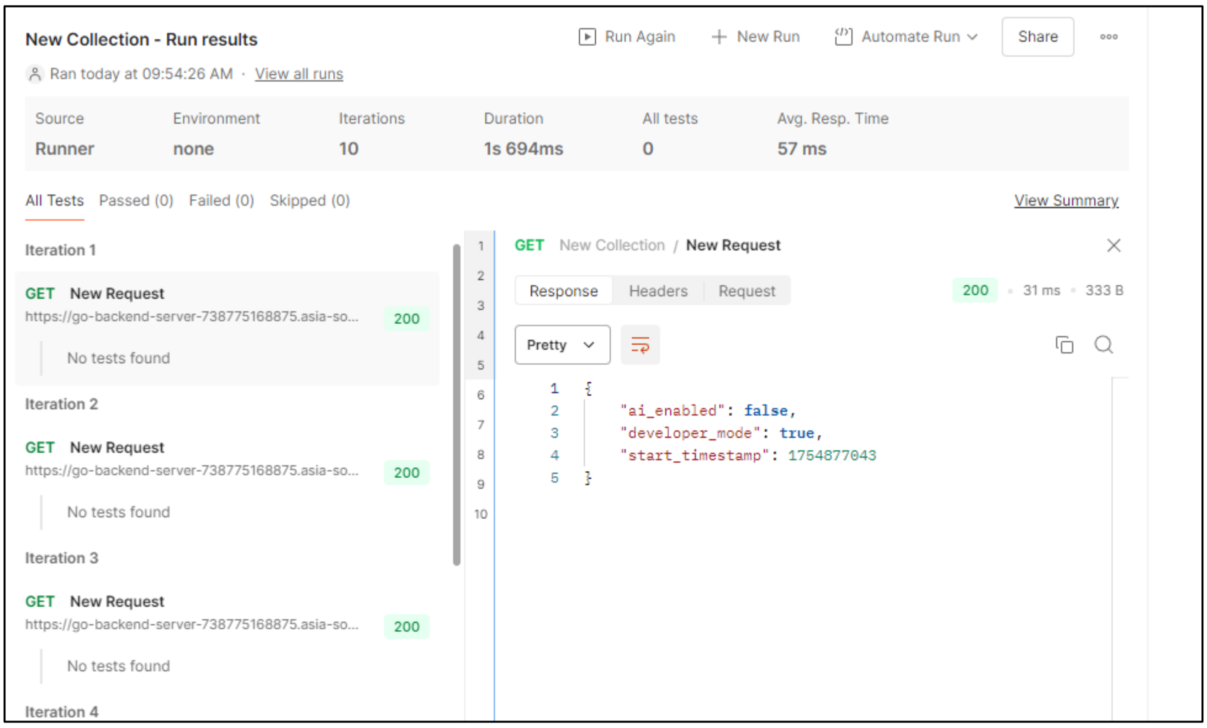
GET New Collection / New Request

Response · Headers · Request — 200 · 263 ms · 4.408 KB

Pretty

```
[
    {
        "id": 8391,
        "user_id": 9,
        "timestamp": "2025-08-11T09:50:59.693281Z",
        "temperature": 32.2,
        "humidity": 82,
        "soil_moisture": 50,
        "is_abnormal": false
    },
    {
        "id": 8390,
        "user_id": 9,
        "timestamp": "2025-08-11T09:50:59.568403Z",
        "temperature": 32.2,
        "humidity": 82,
        "soil_moisture": 50,
        "is_abnormal": false
    },
```

Figure 76: Testing Result for GET /history


New Collection - Run results

Run Again · New Run · Automate Run · Share

Ran today at 09:54:26 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 694ms	0	57 ms

All Tests · Passed (0) · Failed (0) · Skipped (0) · View Summary

Iteration 1
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 2
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 3
GET New Request https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 4

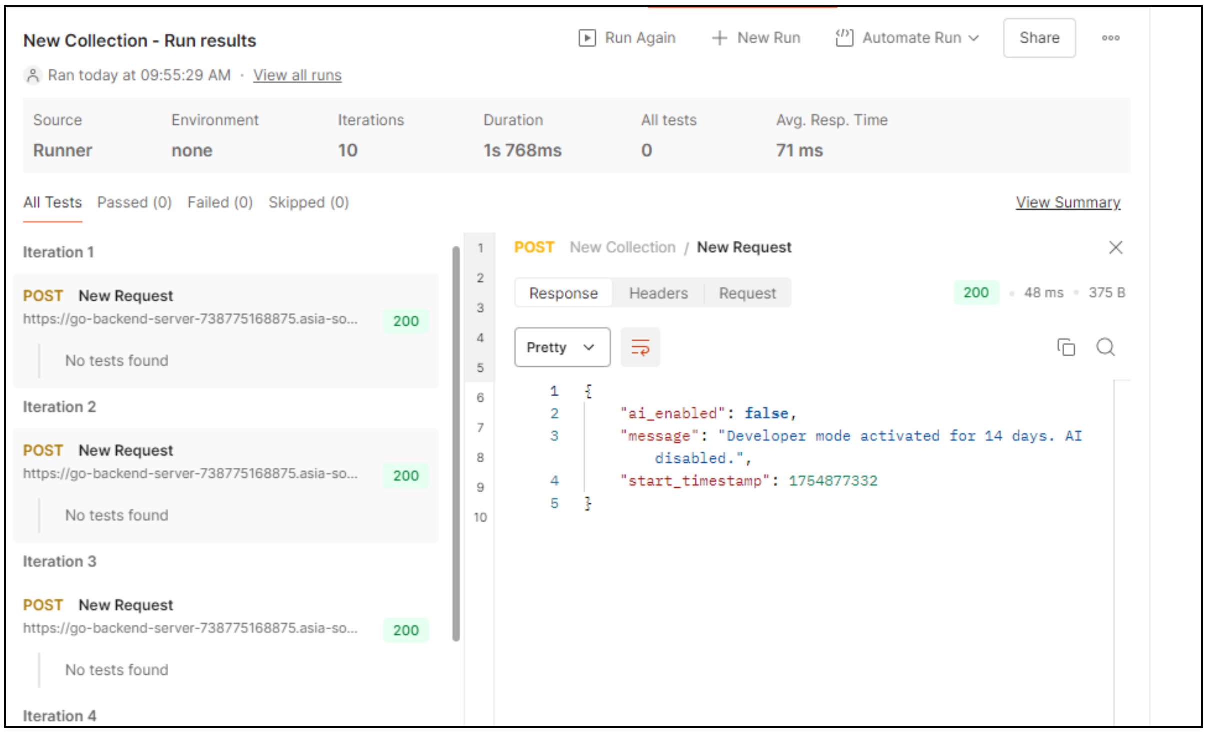
GET New Collection / New Request

Response · Headers · Request — 200 · 31 ms · 333 B

Pretty

```
{
    "ai_enabled": false,
    "developer_mode": true,
    "start_timestamp": 1754877043
}
```

Figure 77: Testing Result for GET /device-config/esp32-001



New Collection - Run results

Ran today at 09:55:29 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 768ms	0	71 ms

All Tests Passed (0) Failed (0) Skipped (0)

Iteration 1

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 2

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

Iteration 3

POST New Request
https://go-backend-server-738775168875.asia-so... 200
No tests found

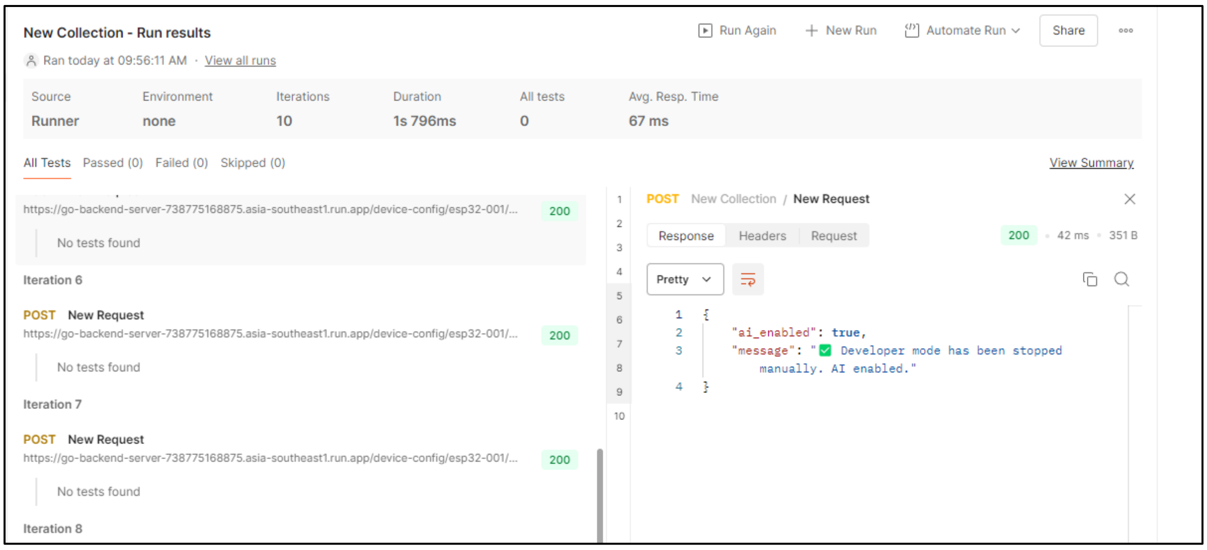
Iteration 4

POST New Collection / New Request

Response 200 · 48 ms · 375 B

```
{
    "ai_enabled": false,
    "message": "Developer mode activated for 14 days. AI
        disabled.",
    "start_timestamp": 1754877332
}
```

Figure 78: Testing Result for POST /device-config/esp32-001/trigger-dev



New Collection - Run results

Ran today at 09:56:11 AM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 796ms	0	67 ms

All Tests Passed (0) Failed (0) Skipped (0)

https://go-backend-server-738775168875.asia-southeast1.run.app/device-config/esp32-001/... 200
No tests found

Iteration 6

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/device-config/esp32-001/... 200
No tests found

Iteration 7

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/device-config/esp32-001/... 200
No tests found

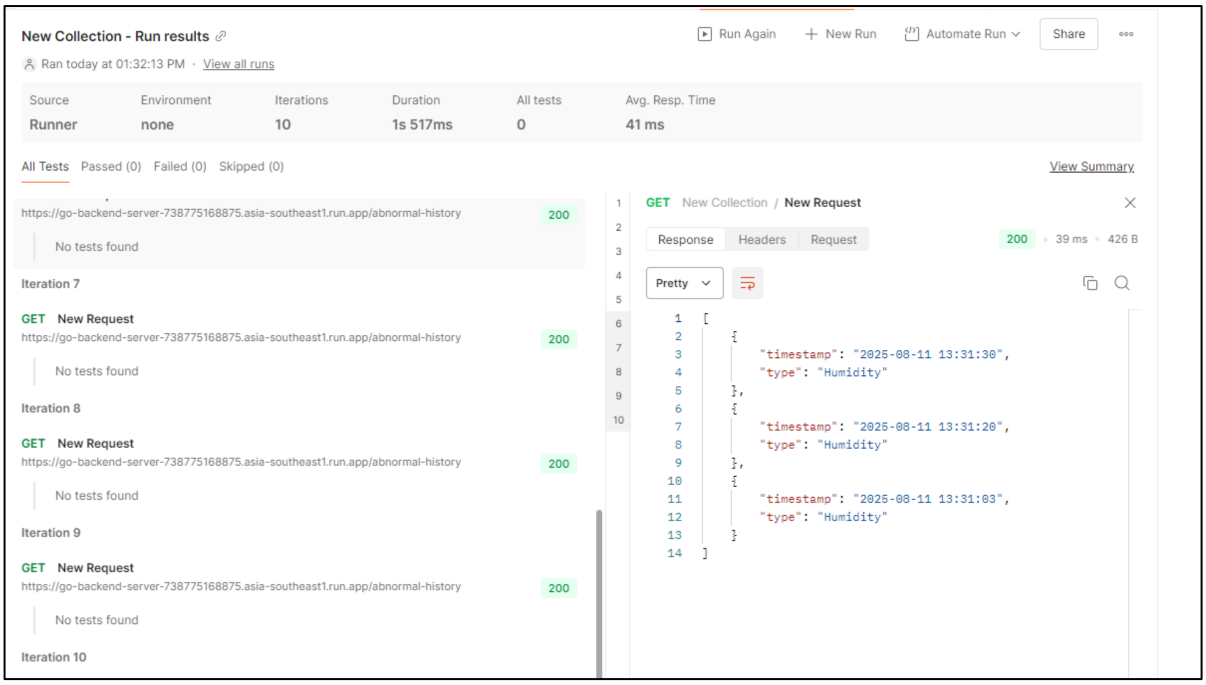
Iteration 8

POST New Collection / New Request

Response 200 · 42 ms · 351 B

```
{
    "ai_enabled": true,
    "message": "✅ Developer mode has been stopped
        manually. AI enabled."
}
```

Figure 79: Testing Result for POST /device-config/esp32-001/stop-dev



New Collection - Run results

Run Again | New Run | Automate Run | Share

Ran today at 01:32:13 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 517ms	0	41 ms

All Tests | Passed (0) | Failed (0) | Skipped (0) | View Summary

https://go-backend-server-738775168875.asia-southeast1.run.app/abnormal-history 200
No tests found

Iteration 7
GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/abnormal-history 200
No tests found

Iteration 8
GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/abnormal-history 200
No tests found

Iteration 9
GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/abnormal-history 200
No tests found

Iteration 10

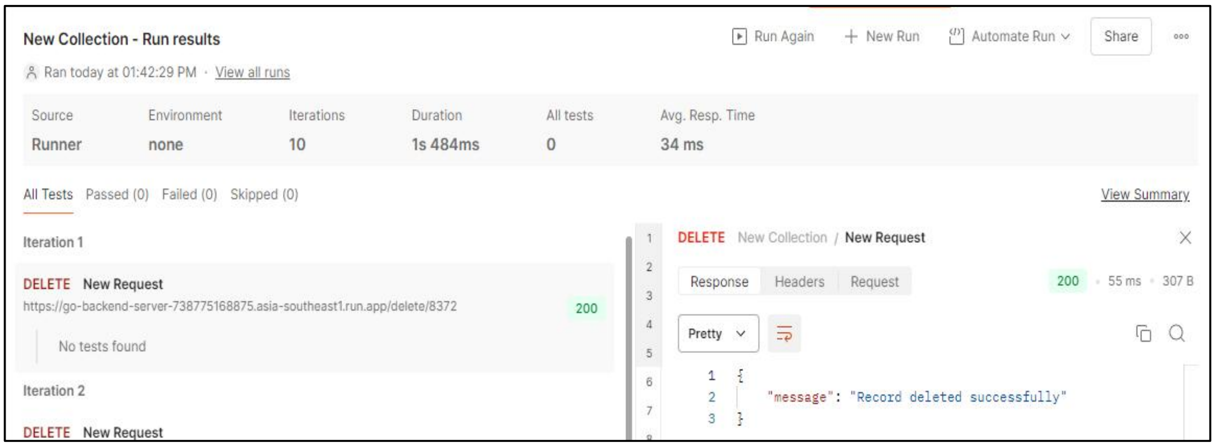
GET New Collection / New Request

Response | Headers | Request — 200 · 39 ms · 426 B

Pretty

```
[
    {
        "timestamp": "2025-08-11 13:31:30",
        "type": "Humidity"
    },
    {
        "timestamp": "2025-08-11 13:31:20",
        "type": "Humidity"
    },
    {
        "timestamp": "2025-08-11 13:31:03",
        "type": "Humidity"
    }
]
```

Figure 80: Testing Result for GET /abnormal-history



New Collection - Run results

Run Again | New Run | Automate Run | Share

Ran today at 01:42:29 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 484ms	0	34 ms

All Tests | Passed (0) | Failed (0) | Skipped (0) | View Summary

Iteration 1
DELETE New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/delete/8372 200
No tests found

Iteration 2
DELETE New Request

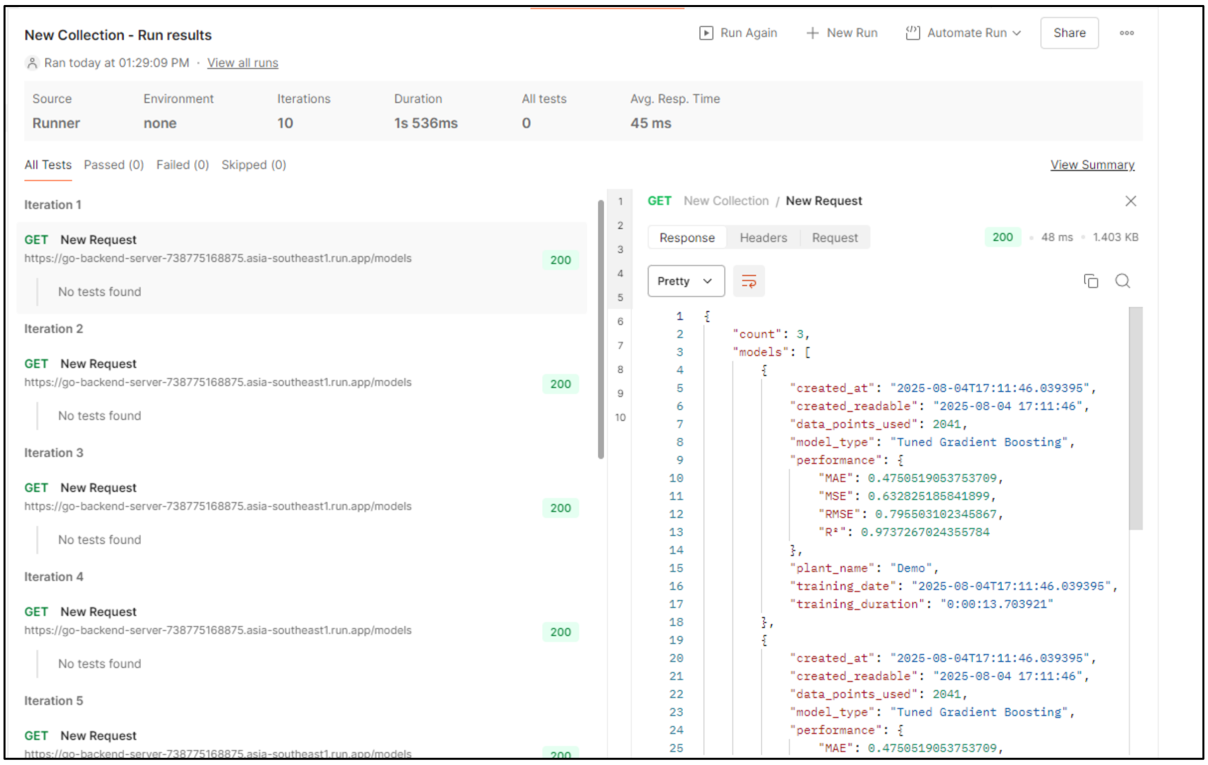
DELETE New Collection / New Request

Response | Headers | Request — 200 · 55 ms · 307 B

Pretty

```
{
    "message": "Record deleted successfully"
}
```

Figure 81: Testing Result for DELETE /delete/:id



New Collection - Run results

Ran today at 01:29:09 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	1s 536ms	0	45 ms

All Tests Passed (0) Failed (0) Skipped (0) View Summary

Iteration 1

GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/models 200
No tests found

Iteration 2

GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/models 200
No tests found

Iteration 3

GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/models 200
No tests found

Iteration 4

GET New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/models 200
No tests found

Iteration 5

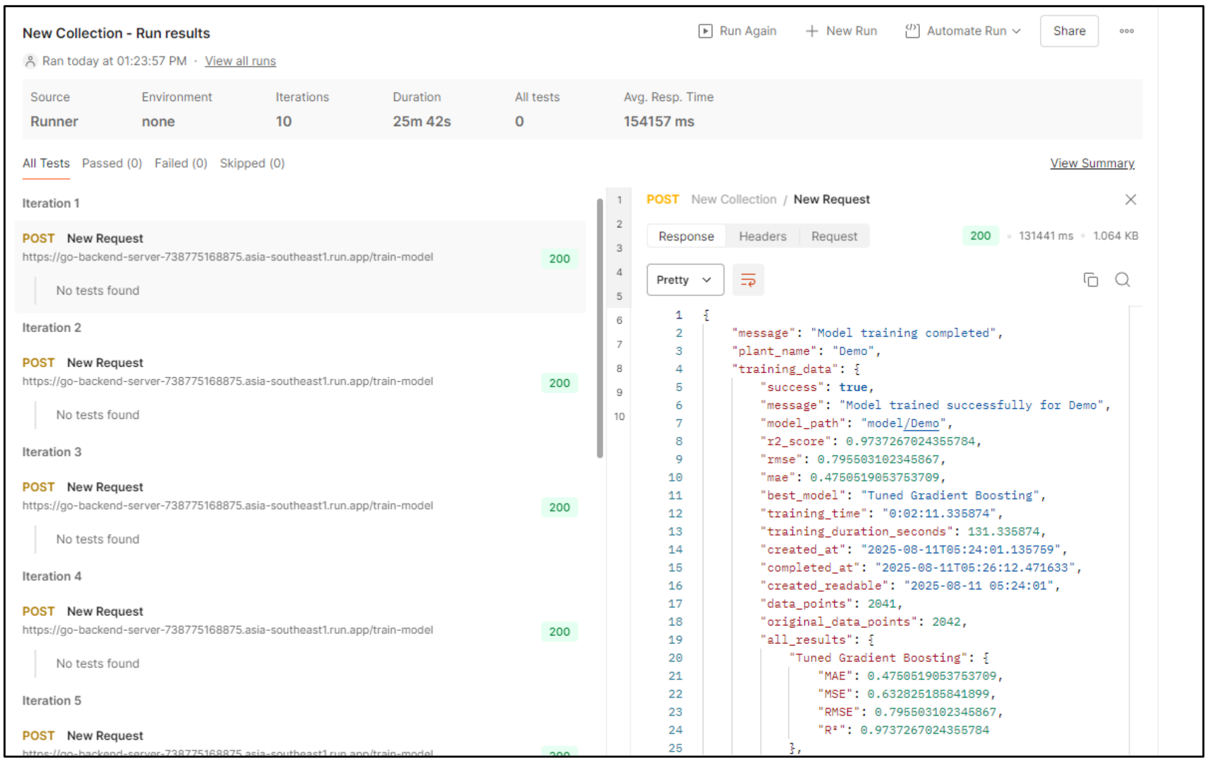
GET New Request

GET New Collection / New Request

Response Headers Request 200 · 48 ms · 1.403 KB

```
{
    "count": 3,
    "models": [
        {
            "created_at": "2025-08-04T17:11:46.039395",
            "created_readable": "2025-08-04 17:11:46",
            "data_points_used": 2041,
            "model_type": "Tuned Gradient Boosting",
            "performance": {
                "MAE": 0.4750519053753709,
                "MSE": 0.632825185841899,
                "RMSE": 0.795503102345867,
                "R²": 0.9737267024355784
            },
            "plant_name": "Demo",
            "training_date": "2025-08-04T17:11:46.039395",
            "training_duration": "0:00:13.703921"
        },
        {
            "created_at": "2025-08-04T17:11:46.039395",
            "created_readable": "2025-08-04 17:11:46",
            "data_points_used": 2041,
            "model_type": "Tuned Gradient Boosting",
            "performance": {
                "MAE": 0.4750519053753709,
```

Figure 82: Testing Result for GET /models



New Collection - Run results

Ran today at 01:23:57 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	10	25m 42s	0	154157 ms

All Tests Passed (0) Failed (0) Skipped (0) View Summary

Iteration 1

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/train-model 200
No tests found

Iteration 2

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/train-model 200
No tests found

Iteration 3

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/train-model 200
No tests found

Iteration 4

POST New Request
https://go-backend-server-738775168875.asia-southeast1.run.app/train-model 200
No tests found

Iteration 5

POST New Request

POST New Collection / New Request

Response Headers Request 200 · 131441 ms · 1.064 KB

```
{
    "message": "Model training completed",
    "plant_name": "Demo",
    "training_data": {
        "success": true,
        "message": "Model trained successfully for Demo",
        "model_path": "model/Demo",
        "r2_score": 0.9737267024355784,
        "rmse": 0.795503102345867,
        "mae": 0.4750519053753709,
        "best_model": "Tuned Gradient Boosting",
        "training_time": "0:02:11.335874",
        "training_duration_seconds": 131.335874,
        "created_at": "2025-08-11T05:24:01.135759",
        "completed_at": "2025-08-11T05:26:12.471633",
        "created_readable": "2025-08-11 05:24:01",
        "data_points": 2041,
        "original_data_points": 2042,
        "all_results": {
            "Tuned Gradient Boosting": {
                "MAE": 0.4750519053753709,
                "MSE": 0.632825185841899,
                "RMSE": 0.795503102345867,
                "R²": 0.9737267024355784
            },
```

Figure 83: Testing Result for POST /train-model

6.3 Project Challenges

Throughout the development of this project, several significant challenges were faced. These difficulties spanned the entire system, from the physical hardware and its connection to the internet, to the complex backend integration, the demanding AI model development, and the real-time frontend user interface.

The first major challenge was related to the hardware and its connectivity. The core of the data collection system, the ESP32 microcontroller, had to be programmed to be extremely reliable. Ensuring it could maintain a stable Wi-Fi connection over long periods to consistently send data was difficult. The firmware, written in C++/Arduino, needed robust error-handling logic to automatically reconnect to the network if the connection dropped. Furthermore, the ESP32 had to efficiently format the sensor readings into JSON, a text-based format, and send them securely over HTTPS, which required careful memory management on the resource-constrained device.

A second significant challenge was the integration of the complex backend system. The project's architecture was not a single application but a collection of different services that needed to communicate perfectly. The Go backend server running on Google Cloud Run had to manage requests from the ESP32, interact with the Python Flask API for AI predictions, and connect to the PostgreSQL database hosted on a completely different cloud platform, Microsoft Azure. Making these separate systems, built with different languages and hosted on different clouds, work together required careful configuration of network rules, firewalls, and secure management of API keys and credentials.

Perhaps the most demanding challenge of the project was the development of the artificial intelligence model. The main objective was to create a model accurate enough to completely replace a physical soil moisture sensor. This was not a simple task. It required extensive work in collecting high-quality data from the ESP32's "developer mode." After collecting the data, a process of complex feature engineering was needed to create meaningful information from basic temperature and humidity readings. This involved creating new features based on time, such as the day of the year or cyclical representations of the hour, to help the model understand temporal patterns. Many experiments were needed to train, test, and fine-tune the Random Forest and Gradient Boosting models to finally achieve a high level of prediction accuracy.

Finally, developing a responsive and user-friendly frontend presented its own set of challenges. The main requirement for the web dashboard was to display sensor data in real-time. This was achieved using WebSockets to stream data directly from the server to the browser. Implementing this feature required careful and advanced state management within the React application. The code had to be written efficiently to handle a constant flow of new data, updating charts and values instantly without causing the user's browser to slow down, lag, or crash. Ensuring this smooth, real-time experience was a key challenge in delivering a polished final product.

6.4 Objectives Evaluation

This section evaluates the project's outcome against the three main objectives set out at the beginning to determine if all initial requirements were successfully met. The first objective was to create an Internet of Things (IoT) based soil moisture monitoring system that uses machine learning to predict soil moisture levels, featuring an ESP32 with "developer" and "user" modes. This objective was fully achieved. An IoT device using an ESP32 microcontroller and a DHT22 sensor was successfully built to collect real-time data. The system successfully supports two operation modes, which can be controlled via the API to switch between collecting full data for training in developer mode and predicting soil moisture in user mode. The core goal was realized as the system can substitute the physical soil moisture sensor with an AI prediction.

The second objective was to build a backend server using the Go programming language, integrated with a PostgreSQL database, to manage all data flow and expose a RESTful API. This objective was also fully achieved. A robust backend server was developed using Go and successfully deployed. It effectively manages all system functions through a comprehensive set of RESTful API endpoints that handle everything from user authentication to data management and AI model training. The server's successful integration with a PostgreSQL database for data storage and the Flask AI service for the machine learning pipeline demonstrates a complete and scalable data management system.

The third objective was to develop a user-friendly, web-based monitoring platform on Firebase for visualizing data and tracking trends. This objective was fully achieved as well. A user-friendly web platform was developed using React and hosted on Firebase. The platform successfully visualizes live sensor data using a real-time dashboard, allows users to track historical trends with interactive charts, and includes a notification system for alerts. The platform successfully integrates with all backend APIs to provide a seamless user experience for effective soil monitoring.

In conclusion, all three primary objectives of the project were met successfully. The result is a complete, end-to-end system that covers all requirements from hardware data collection and AI-based prediction to real-time monitoring on a web-based platform.

6.5 Concluding Remark

In summary, this chapter has provided a thorough and systematic evaluation of the "Sensors Substitution using AI for Agriculture Soil Moisture Monitoring" system, confirming its successful implementation and operational readiness. The tests on the API showed that the system is not only working correctly but is also quick and responsive, which is great news for anyone using the web dashboard.

Furthermore, this chapter has reflected on the significant project challenges that were encountered during the development lifecycle. These obstacles, which included ensuring stable hardware connectivity, integrating a complex multi-service backend architecture, and developing a highly accurate AI prediction model, were all successfully navigated. The ability to overcome these technical hurdles is a testament to the robust design and careful implementation of the project.

Ultimately, this chapter confirms that the project has hit all the targets we set out to achieve. We have a complete system that does what it was designed to do which is use AI to predict soil moisture without needing a physical sensor. Now that we know the system is solid, we can move on to the final chapter to wrap everything up and think about what could come next.

Chapter 7
Conclusion and Recommendations

This final chapter brings the project, "Sensors Substitution using AI for Agriculture Soil Moisture Monitoring," to its close. It will begin by summarizing the entire project, from the initial problem statement through to the final evaluated system, to reflect on the achievements and outcomes. Following the conclusion, this chapter will provide a series of recommendations for future work, outlining potential enhancements and new directions that could build upon the foundation established by this research.

7.1 Conclusion

This project was started to solve a significant and practical problem in modern agriculture which is the unreliability and cost associated with traditional soil moisture sensors. These sensors are prone to degradation from environmental factors, leading to inaccurate data and inefficient water use. The core mission of this project was to design, develop, and validate an innovative system that replaces these physical sensors with an intelligent, AI-driven prediction model, thereby creating a more robust, cost-effective, and scalable solution for soil moisture monitoring.

Throughout this project, a complete, end-to-end system was successfully built, integrating multiple modern technologies. The journey began with the development of the hardware layer, where an ESP32 microcontroller and a DHT22 sensor were configured to act as a reliable IoT data collection node. A key innovation at this stage was the implementation of two distinct operating modes: a "developer mode" for gathering comprehensive training data (including actual soil moisture readings) and a "user mode" where the device relies solely on temperature and humidity to predict soil moisture using AI model.

The heart of the system is the sophisticated backend architecture. A high-performance server was developed using the Go programming language, which seamlessly manages data flow between the IoT device, a PostgreSQL database hosted on Microsoft Azure, and a dedicated AI prediction service. The backend was designed to be scalable and secure, exposing a comprehensive set of RESTful APIs to handle everything from user authentication and data management to remote device control. The successful integration of these disparate services, running on different cloud platforms, stands as a major technical achievement of this project.

The most innovative component of this project is the AI-powered sensor substitution. A machine learning pipeline was developed in Python, utilizing an ensemble of Random Forest and Gradient Boosting models to predict soil moisture levels with a high degree of accuracy ($R^2 > 0.96$). The success of this model, which relies only on temperature and humidity data, validates the central hypothesis of this project: that a physical sensor can be effectively replaced by an intelligent algorithm. The system's ability to allow users to trigger the retraining of this model with their own data ensures that it can be adapted to new plants and different environmental conditions.

Finally, a user-friendly, web-based monitoring platform was developed using React and TypeScript and hosted on Firebase. This frontend provides users with a powerful interface to visualize real-time data via WebSocket, track historical trends, manage their devices, and receive notifications about abnormal conditions. The dashboard successfully brings all the system's capabilities together into a single, cohesive user experience.

In conclusion, this project has successfully met all of its initial objectives. It has delivered a fully functional proof-of-concept that demonstrates the viability of using AI to substitute physical sensors in agricultural monitoring. The final system is a testament to the power of integrating IoT, cloud computing, and machine learning to solve real-world problems, offering a significant contribution to the field of precision agriculture.

7.2 Recommendation

While the current system is a successful proof-of-concept, there are numerous opportunities for future work that could enhance its capabilities, improve its performance, and broaden its applicability. The following recommendations are proposed for future development.

First, the AI model's predictive power could be further improved by incorporating additional environmental variables. While the current model achieves high accuracy with temperature and humidity, integrating more sensors into the ESP32 device, such as a light intensity sensor (LDR) and a barometric pressure sensor (BMP280), could capture more nuanced conditions that influence soil moisture, like evaporation from sunlight. Retraining the AI model with these new features could increase its accuracy and make it more robust across a wider range of environments.

A second recommendation is to implement a fully automated watering system. The current system provides excellent monitoring but still requires manual intervention for irrigation. The logical next step is to close this loop by integrating a relay module and a small water pump with the ESP32. The backend could then be enhanced with a feature allowing users to set a target soil moisture threshold. When the AI predicts moisture levels have dropped below this threshold, the server would automatically command the ESP32 to activate the pump, transforming the system from a passive monitor into an active, intelligent irrigation controller. Third, developing a dedicated native mobile application would provide superior user experience. Although the web dashboard is mobile-friendly, a native app for iOS and Android, built with a framework like React Native or Flutter, could leverage device-specific features such as push notifications for more immediate and reliable alerts. It could also use the phone's GPS to simplify the process of tagging the location of new devices during setup.

Finally, exploring more advanced AI and machine learning models could yield even better performance. While the current ensemble model is effective, experimenting with time-series forecasting models like LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit) networks could capture more complex temporal patterns in the sensor data. Additionally, unsupervised learning models could be used for more advanced anomaly detection, helping to identify subtle deviations that might indicate sensor malfunction or the early onset of plant distress. By pursuing these recommendations, the foundation laid by this project could be built upon to create an even more powerful and impactful solution for the future of smart agriculture.

REFERENCES

[1] Benbrook, Charles M., and Ag BioTech InfoNet. "Rust, resistance, run down soils, and rising costs–Problems facing soybean producers in Argentina." *AgBioTech InfoNet, Technical Paper* 8 (2005): 26. Available: https://www.greenpeace.org/static/planet4-netherlands-stateless/2018/06/rust-resistance-run-down-soi.pdf

[2] Wang, T., Jin, H. & Sieverding, H.L. Factors affecting farmer perceived challenges towards precision agriculture. *Precision Agric* 24, 2456–2478 (2023), doi:10.1007/s11119-023-10048-2

[3] D. Danikovich, "Machine Learning in Agriculture," EffectiveSoft. https://www.effectivesoft.com/blog/machine-learning-in-agriculture.html (accessed Feb. 25 / 2025).

[4] "Moisture sensor corrosion," *Arduino Forum*, Aug. 09, 2017. https://forum.arduino.cc/t/moisture-sensor-corrosion/474537 (accessed Feb. 26, 2025).

[5] J. Blalock, "The Issues Facing Modern Agriculture," *Cropler.io*, Oct. 09, 2024. https://www.cropler.io/blog-posts/the-issues-facing-modern-agriculture

[6] B. Y. Ooi, W. L. Beh, W. K. Lee, and Shervin, "Using the Cloud to Improve Sensor Availability and Reliability in Remote Monitoring," *IEEE Transactions on Instrumentation and Measurement*, vol. 68, no. 5, pp. 1522–1532, May 2019, doi: https://doi.org/10.1109/tim.2018.2882218.

[7] R. Ullah *et al.*, "EEWMP: An IoT-Based Energy-Efficient Water Management Platform for Smart Irrigation," *Scientific Programming*, vol. 2021, pp. 1–9, Apr. 2021, doi: https://doi.org/10.1155/2021/5536884.

[8] Srinivasa Rao Burri, D. K. Agarwal, N. Vyas, and R. Duggar, "Optimizing Irrigation Efficiency with IoT and Machine Learning: A Transfer Learning Approach for Accurate Soil Moisture Prediction," Jul. 2023, doi: https://doi.org/10.1109/wconf58270.2023.10235220.

[9] Q. Li, Z. Wang, W. Shangguan, L. Li, Y. Yao, and F. Yu, "Improved daily SMAP satellite soil moisture prediction over China using deep learning model with transfer learning," *Journal of Hydrology*, vol. 600, p. 126698, Sep. 2021, doi: https://doi.org/10.1016/j.jhydrol.2021.126698.

[10] D. R. Vincent, N. Deepa, D. Elavarasan, K. Srinivasan, S. H. Chauhdary, and C. Iwendi, "Sensors Driven AI-Based Agriculture Recommendation Model for Assessing Land Suitability," *Sensors*, vol. 19, no. 17, p. 3667, Jan. 2019, doi: https://doi.org/10.3390/s19173667.

[11] V. Sinitsin, O. Ibryaeva, V. Sakovskaya, and V. Eremeeva, “Intelligent bearing fault diagnosis method combining mixed input and hybrid CNN-MLP model,” *Mechanical Systems and Signal Processing*, vol. 180, p. 109454, Nov. 2022, doi: https://doi.org/10.1016/j.ymssp.2022.109454.

[12] A. Uthayakumar, M. P. Mohan, E. H. Khoo, J. Jimeno, M. Y. Siyal, and M. F. Karim, “Machine Learning Models for Enhanced Estimation of Soil Moisture Using Wideband Radar Sensor,” *Sensors*, vol. 22, no. 15, p. 5810, Aug. 2022, doi: https://doi.org/10.3390/s22155810.

[13] C. Josephson, B. Barnhart, S. Katti, K. Winstein, and R. Chandra, “RF Soil Moisture Sensing via Radar Backscatter Tags,” *arXiv.org*, 2019. https://arxiv.org/abs/1912.12382 (accessed Jul. 16, 2025).

[14] W. Jiao, J. Wang, Y. He, X. Xi, and X. Chen, “Detecting Soil Moisture Levels Using Battery-Free Wi-Fi Tag,” *arXiv.org*, 2022. https://arxiv.org/abs/2202.03275 (accessed Jul. 16, 2025).

APPENDIX

Source Code (Repository)

Web Application Repository: https://github.com/Kaisheng328/agriculture-monitoring-ui

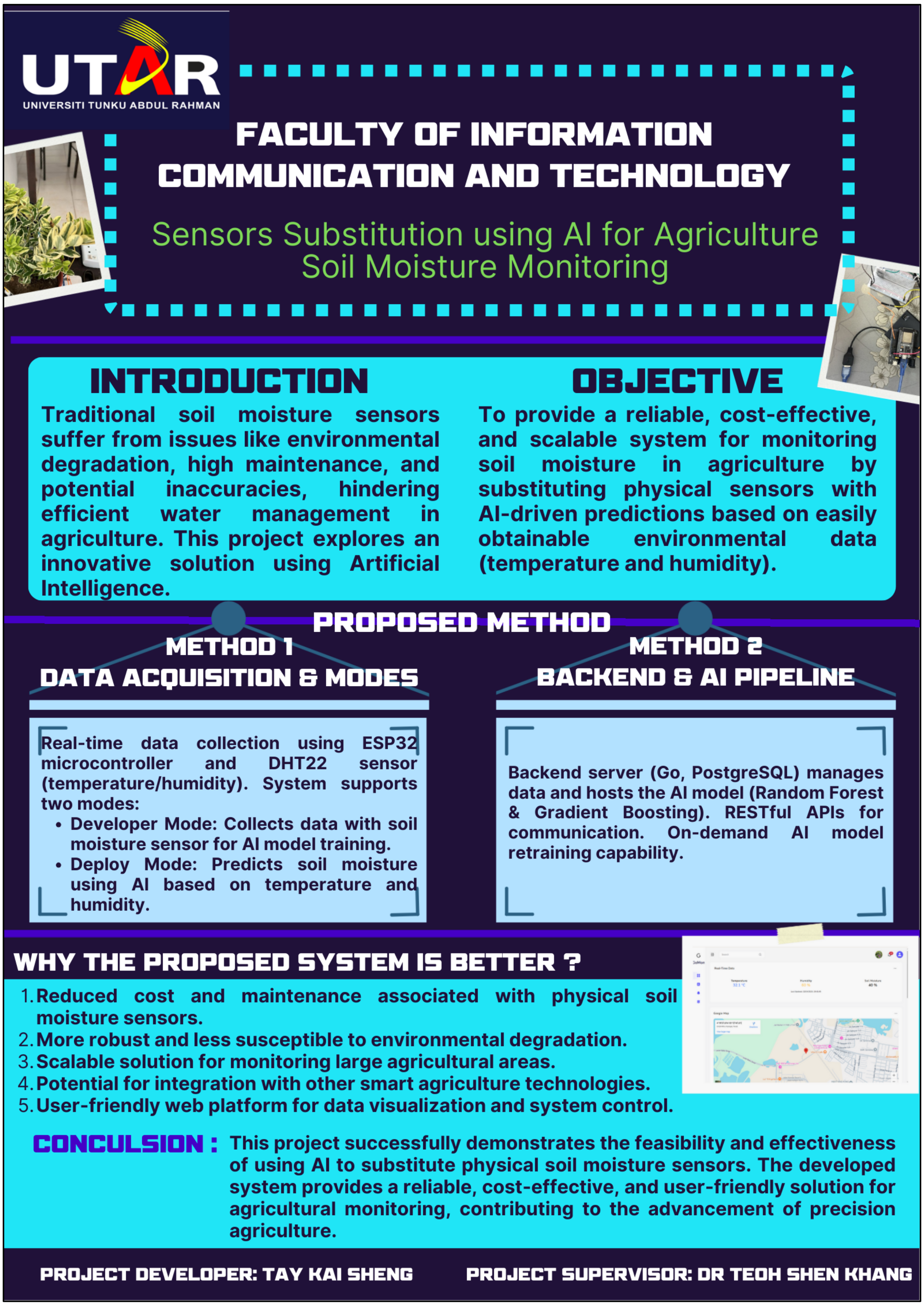
Go Backend Repository: https://github.com/Kaisheng328/go-agriculture-monitoring-backend

Python Flask Repository: https://github.com/Kaisheng328/flask-ai-model

Source Code (Microcontroller)

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>
#include <DHT_U.h>
#include <ArduinoJson.h>
#include <esp_task_wdt.h>

#define WIFI_SSID "wifiID"
#define WIFI_PASSWORD "wifiPassword"
#define DATA_URL "https://go-backend-server-738775168875.asia-southeast1.run.app/sensor-data"
#define LOGIN_URL "https://go-backend-server-738775168875.asia-southeast1.run.app/login"
#define CONFIG_URL "https://go-backend-server-738775168875.asia-southeast1.run.app/device-config/esp32-
001"

#define DHTPIN 4          // Pin connected to the DHT22 sensor
#define DHTTYPE DHT22     // Specify the sensor type (DHT22)
#define SOIL_PIN 32       // Pin connected to the soil moisture sensor (analog)
#define LED_PIN 2 // Built-in or external LED

enum LedState {
  STATUS_CONNECTING,
  STATUS_NORMAL,
  STATUS_DEVELOPER
};

LedState currentLedState = STATUS_CONNECTING;
unsigned long lastLedToggle = 0;
bool ledOn = false;

bool developerMode = false;
time_t developerStartTime = 0;
String jwtToken = "";
DHT dht(DHTPIN, DHTTYPE);

const unsigned long checkInterval = 600000; // 10 minutes
const unsigned long restartInterval = 3610000; // Optional hourly restart
const unsigned long developerModeDuration = 14UL * 24 * 60 * 60; // 14 days in seconds
// const unsigned long developerModeDuration = 5UL * 60; // 5 minutes in seconds

unsigned long previousMillis = 0;
unsigned long lastRestartMillis = 0;

void setup() {
  Serial.begin(115200);
  currentLedState = STATUS_CONNECTING;
  dht.begin();
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW);
  const esp_task_wdt_config_t wdt_config = {
    .timeout_ms = 10000,      // 10 seconds timeout
    .idle_core_mask = 1 << 0, // Core 0 (typical for single-core task watchdog)
    .trigger_panic = true     // Panic if not fed in time
  };

  esp_task_wdt_add(NULL);  // Add current task (loop task) to watchdog

  // Initialize pins
  pinMode(SOIL_PIN, INPUT);
  // Connect to Wi-Fi
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected to Wi-Fi");
  configTime(0, 0, "pool.ntp.org", "time.nist.gov");  // NTP for Unix time
  waitForTimeSync();
  loginAndGetToken(); // Get JWT token at startup
  fetchDeviceConfig();
  sendSensorData(); // Take first reading immediately
  previousMillis = millis();
  lastRestartMillis = millis();
}
void updateLED() {
  unsigned long now = millis();

  switch (currentLedState) {
    case STATUS_CONNECTING:
      if (now - lastLedToggle > 200) { // Fast blink
        ledOn = !ledOn;
        digitalWrite(LED_PIN, ledOn ? HIGH : LOW);
        lastLedToggle = now;
      }
      break;

    case STATUS_DEVELOPER:
      if (now - lastLedToggle > 1000) { // Slow blink
        ledOn = !ledOn;
        digitalWrite(LED_PIN, ledOn ? HIGH : LOW);
        lastLedToggle = now;
      }
      break;

    case STATUS_NORMAL:
      digitalWrite(LED_PIN, HIGH); // Solid ON
      break;
  }
}

void checkWiFiConnection() {
  if (WiFi.status() != WL_CONNECTED) {
    Serial.println("⚠️ WiFi lost. Reconnecting...");
    WiFi.disconnect();
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    unsigned long startAttemptTime = millis();

    while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime < 15000) {
      delay(500);
      Serial.print(".");
    }

    if (WiFi.status() != WL_CONNECTED) {
      Serial.println("\n❌ WiFi reconnection failed, restarting...");
      ESP.restart();
    } else {
      Serial.println("\n✅ Reconnected to WiFi");
    }
  }
}

void waitForTimeSync() {
  Serial.print("⏳ Syncing time via NTP");
  time_t now = time(nullptr);
  unsigned long start = millis();

  while (now < 100000 && millis() - start < 30000) { // wait up to 30 seconds
    delay(500);
    Serial.print(".");
    now = time(nullptr);
  }

  if (now >= 100000) {
    Serial.println("\n✅ Time synced");
  } else {
    Serial.println("\n⚠️ Time sync failed. Continuing anyway.");
    // Optional: continue without restarting
  }
}

void loginAndGetToken() {
  if (WiFi.status() == WL_CONNECTED) {
    for (int attempt = 0; attempt < 3; attempt++) {
      HTTPClient http;
      http.begin(LOGIN_URL);
      http.addHeader("Content-Type", "application/json");

      String loginPayload = "{\"username\": \"testing\", \"password\": \"12345\"}";
      int httpResponseCode = http.POST(loginPayload);

      if (httpResponseCode == 200) {
        String response = http.getString();
        DynamicJsonDocument doc(512);
        deserializeJson(doc, response);
        jwtToken = doc["token"].as<String>();
        Serial.println("✅ Login successful");
        http.end();
        currentLedState = STATUS_NORMAL;
        return;
      }

      http.end();
      Serial.println("❌ Login failed (attempt " + String(attempt + 1) + ")");
      delay(2000 * (attempt + 1));  // Exponential backoff
    }
    Serial.println("❌ Login failed after multiple attempts, restarting...");
    ESP.restart();
  }
}

void fetchDeviceConfig() {
  if (WiFi.status() == WL_CONNECTED && jwtToken != "") {
    HTTPClient http;
    http.begin(CONFIG_URL);
    http.addHeader("Authorization", "Bearer " + jwtToken);
    int code = http.GET();

    if (code == 200) {
      String response = http.getString();
      DynamicJsonDocument doc(512);
      DeserializationError error = deserializeJson(doc, response);
      if (!error) {
        bool requestedMode = doc["developer_mode"];
        time_t start = doc["start_timestamp"];
        time_t now = time(nullptr);

        if (requestedMode && now - start <= developerModeDuration) {
          developerMode = true;
          developerStartTime = start;
        } else {
          developerMode = false;
        }

        Serial.println("🔁 Mode check → Developer mode: " + String(developerMode));
      }
    } else {
      Serial.println("❌ Config fetch failed: " + String(code));
    }
    currentLedState = developerMode ? STATUS_DEVELOPER : STATUS_NORMAL;
    http.end();
  }
}

void sendSensorData() {
  float temperature = dht.readTemperature();
  float humidity = dht.readHumidity();

  if (isnan(temperature) || isnan(humidity)) {
    Serial.println("⚠️ Failed to read DHT22");
    return;
  }

  float soilMoisturePercent = -1;
  int soilValue = -1;

  if (developerMode) {
    soilValue = analogRead(SOIL_PIN);
    soilMoisturePercent = (100.0 * (4095 - soilValue)) / 4095.0;
    Serial.println("Soil value (analog): " + String(soilValue));
  }

  if (WiFi.status() == WL_CONNECTED && jwtToken != "") {
    HTTPClient http;
    http.begin(DATA_URL);
    http.addHeader("Content-Type", "application/json");
    http.addHeader("Authorization", "Bearer " + jwtToken);

    String payload = "{ \"temperature\": " + String(temperature) +
                     ", \"humidity\": " + String(humidity);
    if (developerMode) {
      payload += ", \"soil_moisture\": " + String(soilMoisturePercent);
    }
    payload += "}";

    int httpCode = http.POST(payload);
    if (httpCode > 0) {
      Serial.println("✅ Data sent: " + payload + "Response: " + http.getString());
    } else {
      Serial.println("❌ Send error: " + String(httpCode));
    }
    http.end();
  }
}

void restartDevice() {
  Serial.println("🔄 Restarting ESP32...");
  delay(1000);
  ESP.restart();
}

void checkRestartCondition() {
  if ((millis() - lastRestartMillis) >= restartInterval) {
    lastRestartMillis = millis();
    restartDevice();
  }
}

void loop() {
  updateLED();
  unsigned long now = millis();
  if ((now - previousMillis) >= checkInterval) {
    previousMillis = now;
    fetchDeviceConfig();
    sendSensorData();
  }
  esp_task_wdt_reset(); // Feed the watchdog
  checkRestartCondition();
}
```

POSTER



UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION COMMUNICATION AND TECHNOLOGY

Sensors Substitution using AI for Agriculture Soil Moisture Monitoring

INTRODUCTION

Traditional soil moisture sensors suffer from issues like environmental degradation, high maintenance, and potential inaccuracies, hindering efficient water management in agriculture. This project explores an innovative solution using Artificial Intelligence.

OBJECTIVE

To provide a reliable, cost-effective, and scalable system for monitoring soil moisture in agriculture by substituting physical sensors with AI-driven predictions based on easily obtainable environmental data (temperature and humidity).

PROPOSED METHOD

METHOD 1
DATA ACQUISITION & MODES

Real-time data collection using ESP32 microcontroller and DHT22 sensor (temperature/humidity). System supports two modes:

- Developer Mode: Collects data with soil moisture sensor for AI model training.
- Deploy Mode: Predicts soil moisture using AI based on temperature and humidity.

METHOD 2
BACKEND & AI PIPELINE

Backend server (Go, PostgreSQL) manages data and hosts the AI model (Random Forest & Gradient Boosting). RESTful APIs for communication. On-demand AI model retraining capability.

WHY THE PROPOSED SYSTEM IS BETTER ?

1. Reduced cost and maintenance associated with physical soil moisture sensors.
2. More robust and less susceptible to environmental degradation.
3. Scalable solution for monitoring large agricultural areas.
4. Potential for integration with other smart agriculture technologies.
5. User-friendly web platform for data visualization and system control.

CONCLUSION : This project successfully demonstrates the feasibility and effectiveness of using AI to substitute physical soil moisture sensors. The developed system provides a reliable, cost-effective, and user-friendly solution for agricultural monitoring, contributing to the advancement of precision agriculture.

PROJECT DEVELOPER: TAY KAI SHENG
PROJECT SUPERVISOR: DR TEOH SHEN KHANG