Irrigation System with Hardware and GUI Applications

ESE 498 Capstone Design Project Formal Report

Client: Washington University Green House: *Ben Wolf - (wolfbenjamin@wustl.edu)*

Title: *Greenhouse manager for the Jeanette Goldfarb Plant Growth Facility*

Advisor: Dorothy Wang - (dorothyw@wustl.edu)

Title: *Senior Lecturer Electrical & Systems Engineering*

Engineers: Kylee Bennett - (b.jessica1@wustl.edu)

David Young - (d.m.young@wustl.edu)

Chengyu Li - (lichengyu@wustl.edu)

Washington University in St. Louis
McKelvey School of Engineering

Abstract

Greenhouse irrigation can be both time-consuming and inconsistent. Therefore, we aim to design and implement an irrigation system capable of detecting when plants need water, thus providing a consistent and sufficient amount of water to them. To achieve this goal, we will integrate hardware and software components, drawing on concepts from circuits, Python programming, control systems, sensors and actuators, and embedded systems communication (MQTT). Our system will include (but is not limited to) a Raspberry Pi as the main controller, a water reservoir, a water level sensor, solenoid valves to manage water flow, capacitive moisture sensors, multiplexers and demultiplexers to handle signals, an ESP32 to gather and transmit data to the Pi, and a GUI for manual control. Ultimately, we aim to deliver a scalable irrigation solution that automates plant watering in a controlled environment.

Introduction

A. Background

Plants, whether grown indoors or outdoors, contribute significantly to both aesthetic appeal and functionality in human life. Initially, this project was conceived as a small-scale automatic watering system designed for individual plant owners. The goal was to ensure the success and prosperity of houseplants by automating their irrigation needs. However, as the project evolved and the search for sponsorship began, it became evident that the concept had broader applications, particularly in agriculture and greenhouse management.

Greenhouses provide controlled environments for plant growth, offering protection from external weather conditions and optimizing factors such as temperature, humidity, and light exposure. However, traditional irrigation methods in greenhouses require significant manual labor and water resources, often leading to inefficiencies and inconsistent watering practices. A well-designed automated irrigation system can address these challenges by ensuring precise water distribution, conserving resources, and enhancing crop health.

The proposed irrigation system aims to integrate advanced sensing and control mechanisms to create a scalable and efficient solution. By incorporating real-time monitoring and remote accessibility, the system will not only reduce labor-intensive tasks but also improve agricultural productivity. The potential to expand the system's capabilities with additional environmental sensors further underscores its relevance in modern greenhouse automation.

B. Problem Statement

Effective irrigation is crucial for maintaining healthy plant growth, yet many existing methods rely on manual intervention or inefficient automated systems that do not

adapt to real-time soil conditions. Overwatering and underwatering remain common issues in greenhouse environments, often leading to resource wastage and poor plant health.

Currently, the Wash U greenhouse's watering is not fully automation; however, even if a system was in place most options would not provide users with real-time insights or remote-control capabilities, limiting their ability to optimize irrigation schedules based on changing environmental conditions. To be specific, when doing some research on IOT irrigations systems and automaton in greenhouses, there were not too many companies found implementing this technology. In fact, many companies in larger agricultural spheres only focus on the water distribution using timers rather than the health and temperament of the plants' soil. This can be quite problematic, as the whole purpose of our design is to focus on the plant's overall health. For example, most of the Argus Titan systems only use timers and controls to water greenhouse plants [4].

This project seeks to address these limitations by developing a greenhouse irrigation system that integrates soil moisture sensors, automated water regulation, and remote monitoring features. By implementing a progressive design approach—starting from a basic wired system and advancing to a fully networked, remote-accessible version—the project will deliver a practical and scalable solution for greenhouse operators. The ultimate goal is to improve water conservation, reduce manual labor, and enhance overall plant health through precise and adaptive irrigation management.

C. Aims/Objectives

The objective of this project is to design a robust and scalable greenhouse irrigation system that can automatically detect soil moisture levels, regulate water supply based on plant needs, and provide users with remote monitoring and control options. The ultimate goal is to create an innovative solution that not only saves labor and water resources but also improves crop health through precise and timely irrigation. The initial implementation supports 3 plants. By using modular components (sensors, valves, and control points), the system can grow beyond 3 plants. Each additional plant simply requires an extra moisture sensor and a dedicated control valve.

To achieve this, the project is organized into multiple versions that build upon one another.

1. Version 01 (Wired Setup)

- Uses a Raspberry Pi and basic scripts to manage watering schedules.
- Relies on a wiring moisture sensor directly to the Pi to measure soil moisture for one or more plants.
- Employs a wired valve system that opens and closes based on the sensor reading.

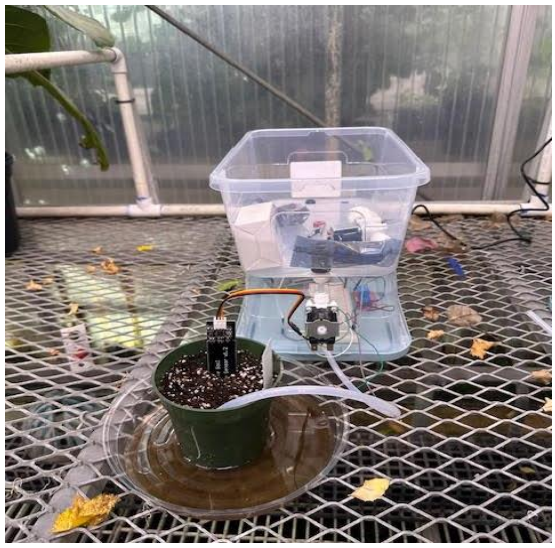


Figure 01 - Version 01

2. Version 02 (Wireless Sensor)

- Replaces the wired moisture sensor with a wireless module to allow flexible placement around the greenhouse.
- Uses MQTT communication protocol to send moisture data to the Raspberry Pi, which then connects to Wi-Fi for broader network access.
- Maintains the same in and out valves for water flow control.

Figure 02 - Version 02

3. Version 03 (Incorporating reservoir monitoring)

- Receives signals from the wireless moisture sensor module, forwarding them to the controller.

- Uses a multiplexer and relay system to open/close the solenoid valves. This approach supports expansion to multiple valves for multiple plants.
- Monitors reservoir water level using a simple float valve.
- Connects to water hose to refill reservoir.
- Ensures each component (sensor, controller, valve) communicates smoothly to simplify troubleshooting and future upgrades.

Figure 03 - Image of front and float valve

Figure 04 - Version 03

4. Version 04 (Conceptual Advanced Control & GUI)

- Explores more sophisticated estimation and control algorithms (e.g., PID control, machine learning–based predictions).
- Integrates additional sensors—temperature, humidity, CO_2—to optimize irrigation further and potentially automate other greenhouse functions (ventilation, heating, etc.).
- May incorporate a secondary reservoir specifically for fertilized water or automated fertilizer mixing if required.
- Graphical User Interface (GUI) accessible over the network:
    - Displays real-time moisture readings and reservoir water levels.
    - Allows users to set custom watering thresholds and schedules.

- Sends alerts (e.g., email, text message, or app notifications—configurable based on user preference) when moisture levels or reservoir levels are out of the desired range.

In Versions 01–03, a soil moisture sensor serves as the main input to determine when to water. Each plant site will eventually have its own sensor for independent moisture readings. Future expansions (Version 04 and beyond) may include sensors for temperature, humidity, CO_2, and other environmental parameters. Additionally, Version 04 could implement a GUI system that would allow the user to remotely monitor the irrigation system. These extra inputs will help the system make more precise decisions about watering and overall greenhouse climate control. Each plant or group of plants can be connected to its own valve for precise water delivery.

At minimum, the system has:

- In Valve: Controls water flow into a mixing reservoir.
- Out Valve: Drains or diverts excess water from the mixing reservoir when needed.
- Mixing Reservoir: Currently, the reservoir is used to hold water before it's distributed to plants. Fertilizer can be added manually for now, but a future version could introduce a secondary reservoir or automated fertilizer mixing system.

Below is the "Block Diagram of the Wireless Greenhouse Irrigation System (Version 03/04)", illustrating data flow and how each component interconnects. The diagram begins with the VNC/GUI, where users can remotely monitor real-time data—such as soil moisture and reservoir levels—and adjust system settings. These commands travel through the Wi-Fi module, which relays the information to the controller (often a Raspberry Pi or similar device). The controller uses these user-defined thresholds, along with sensor readings, to decide when and how much water to deliver.

Moisture data originates in the soil, where a wireless sensor measures water content. This sensor transmits information via the data collection module to the controller. Once the controller has evaluated soil moisture and reservoir levels, it issues commands through a multiplexer, which coordinates signals for multiple outputs, and finally a relay. The relay, in turn, energizes the valves. An "out valve" supplies water from the reservoir to the plants, while the "in valve" refills the reservoir when needed. This setup ensures precise and automatic irrigation control, with the user able to oversee and adjust the entire process through a straightforward interface.

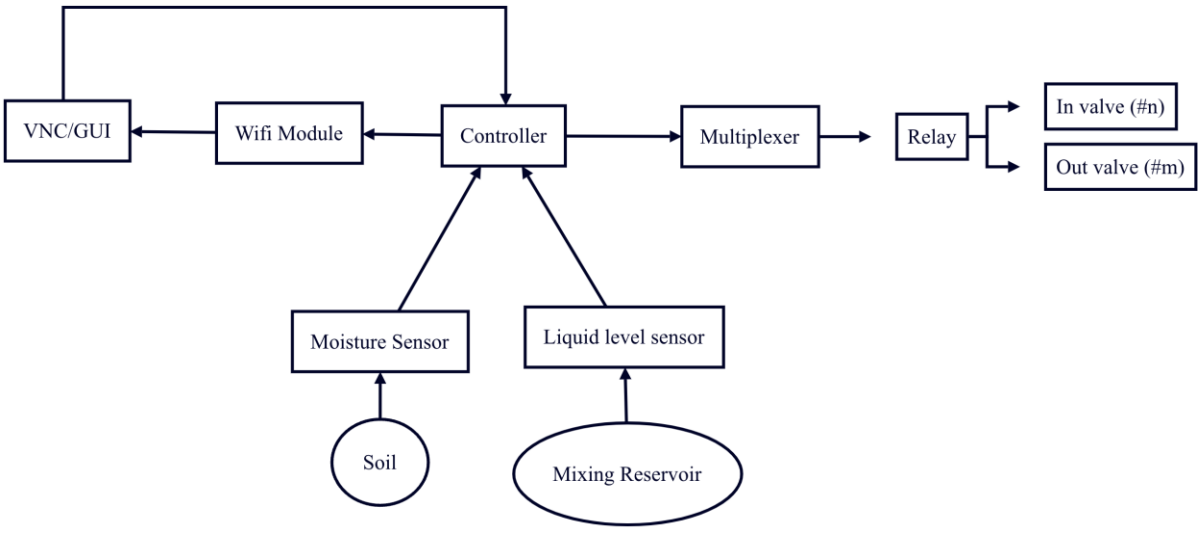


Figure 05 – Block Diagram of the Wireless Greenhouse Irrigation System (Version 03/04)

Version 03 serves as a cornerstone in demonstrating the synergy between hardware components and software applications. In this iteration, a Wi-Fi module is integrated with the Raspberry Pi, enabling the transmission of real-time moisture data from a wireless sensor.The controller processes the incoming information and makes decisions regarding water flow, employing a multiplexer and relay to open or close valves. One valve supplies water into the mixing reservoir (in valve), while another drains or diverts water when needed (out valve). A liquid level sensor in the reservoir provides continuous feedback on available water, ensuring that both irrigation and water replenishment occur automatically.

On the software side, the user interacts with the system through a graphical user interface (GUI) that can be accessed over a network connection. This GUI provides real-time visuals of soil moisture trends, reservoir water levels, and other critical data. Users can also modify irrigation thresholds, set watering schedules, and receive alerts if the system detects anomalies such as unexpectedly high or low moisture readings. By centralizing control and monitoring functions within the Raspberry Pi, Version 04 ensures smooth communication among all system components, making it significantly easier to troubleshoot issues and implement future enhancements.

Methods/ Version Updates History

This project evolved into four main versions, each focusing on improvements in hardware design, sensor integration, and remote data management. Version 01 introduced a simple wired prototype for automated irrigation. A single Raspberry Pi 4 served as the central

controller, reading moisture levels from a wired soil sensor connected via an MCP3008 ADC. The Pi's SPI pins were enabled through the 'raspi-config' interface, allowing the system to convert the analog sensor outputs into digital values. A single in-line solenoid valve, powered through a 5 V relay module, opened or closed based on soil moisture thresholds. This valve drew water from a reservoir bucket (Figure 06) and delivered it to the plant via a ¼-inch tube. The initial setup featured minimal physical protection against leaks or corrosion (Figure 07), although electronics were kept on top of a plastic enclosure to mitigate risk.

Figure 06 - Tub Used for Reservoir

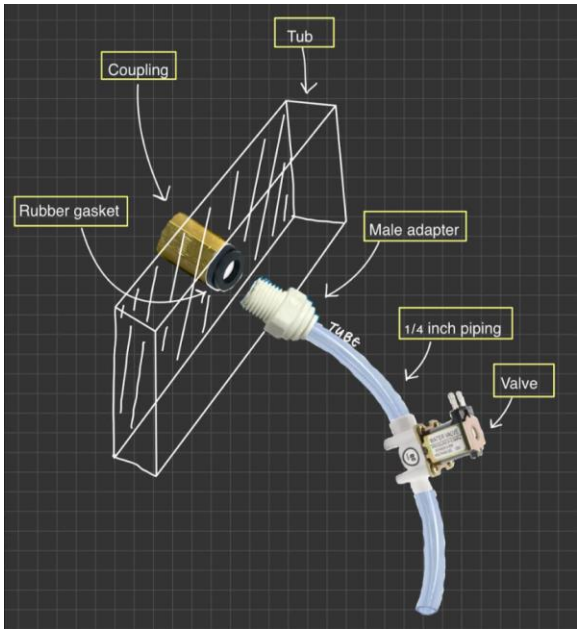
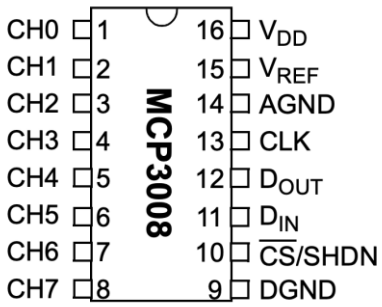


Figure 07 - Image of Initial Irrigation System (Version 01)

A build of materials (BOM) for Figure 07 has been provided in Figure 25 of the Appendix. When the soil was dry the moisture sensor then sent a signal to the raspberry pi which allowed the relay to open the water valve. Once the soil was fully moist, the valve was then closed.

Version 01's wiring instructions are given below:
MCP 3008 ADC to PI



Power the MCP3008

- MCP3008 VDD → Pi 5.0 V
- MCP3008 VREF → Pi 5.0 V (so the ADC range is 0–5.0 V)
- MCP3008 AGND + DGND → Pi GND

Connect SPI pins

- MCP3008 CLK → Pi SCLK (GPIO 11 on a standard 40-pin header)
- MCP3008 DOUT → Pi MISO (GPIO 9)
- MCP3008 DIN → Pi MOSI (GPIO 10)
- MCP3008 CS/SHDN → Pi CE0 (GPIO 8)

Sensor to MCP3008

GND → Pi ground (same ground as MCP3008)
VCC → Pi 5.0 V
Sensor 1 AOUT → MUX C0
Sensor 2 AOUT → MUX C1
··· and so on, up to 8 sensors.

Relay and Valve to Demux

DC+ → 5.0 V
DC- → Pi ground
IN → one of the Y outputs on the 74154, e.g. Y0 (Pin 4 on 74154)

COM → 9 V battery (+)
NO → Valve's positive terminal
Valve's other terminal → 9 V battery (−)

The control software for Version 01 comprises multiple Python modules for sensor reading, data logging, and actuator control, coordinated by a central main.py script. To make the raw moisture readings more interpretable, the system first measures the sensor's output when it is completely dry (no moisture contact), treating this reading as x_{dry}. It then measures the output when the sensor is fully saturated with water (e.g., wrapped in a wet paper towel), designating that as x_{wet}. x_{wet}. An example of these measurements on an ESP32 platform yields $x_{dry} \approx 3385.14$ and $x_{wet} \approx 1604$. Using these extremes, we convert any intermediate reading ($x_{measured}$) into a percentage:

$$\text{readInterpret} = [\ (x_{dry} - x_{measure})/(x_{dry} - x_{wet})\ ] * 100\ \ [\%]$$

Thus, readInterpret approaches 100 % in the presence of abundant moisture, and it nears 0 % when conditions are very dry. Figure 08 illustrates how sensor readings increase when moisture is placed closer to the sensor's base—indicating that the sensor length is "weighted," with the base contributing more to the final reading than the tip. Consequently, we place the sensor so that the principal moisture source (e.g., the core root zone) is near the base, giving more consistent measurements. From these tests, we define a plant's "golden percent" moisture target. Whenever the sensor reading falls below this threshold, an automatic watering algorithm instructs a relay to open the solenoid valve for a brief soak period (commonly 2 s), thus restoring adequate soil moisture.

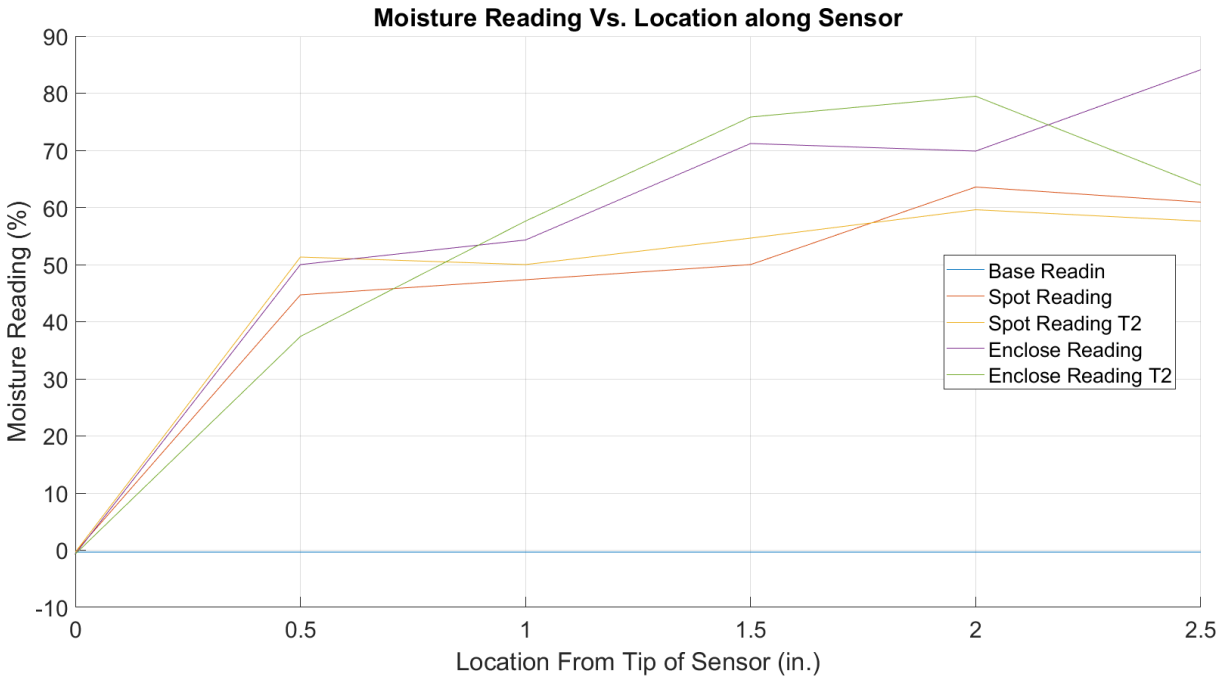


Figure 08 - Percentage of the max moisture level vs. Location of applied Moisture

Beyond calibration, we verify that while absolute sensor readings differ among hardware (e.g., Arduino, Raspberry Pi, ESP32), they remain consistent within the same platform. We also confirm that capacitive sensors are more durable than their resistive counterparts, which often corrode over time. Users in online communities suggest extending sensor life by waterproofing the PCB with epoxy, nail polish, or liquid electrical tape, potentially increasing operating lifespan to around two years. Separately, we encounter network challenges when attempting to run our ESP32 devices on the university's secure Wi-Fi, as these networks typically require credentials beyond a simple SSID and password. Although we manage to whitelist some ESP32 boards on the open network, others remain incompatible—a complication that lies outside our project's core objectives.

Finally, to interface with the Raspberry Pi, the sensor's analog output (0 V to ~5 V) goes through an MCP3008 ADC, which provides 10-bit digital readings. The Pi computes the

moisture percentage from these readings and compares it against the dryness threshold (e.g., 80 %). If the threshold is exceeded, the relay energizes, opening a solenoid valve that delivers water from a reservoir via a ¼ inch tube. After a short soak time, the valve closes, and the event is logged in a CSV file. Over multiple days of testing, researchers observe that brief, frequent watering episodes (i.e., bottom-watering) effectively maintain stable soil moisture in potted plants. However, as the scope of experiments widens to larger greenhouse spaces and multiple plants, the limitations of a single sensor–single valve model become apparent, prompting further iterations to improve scalability and flexibility.

Version 02 introduced wireless soil moisture sensors to remove the physical tangle of wires and allow flexible sensor placement around the greenhouse. This shift was crucial for test scenarios with multiple plants, such as corn grown in 11-inch pots and sensitive plants like tobacco (MBs) or fast-growing Brassicas. Each wireless sensor module was built around an ESP32 board, which periodically measured soil moisture (either via a capacitive probe or a similar analog sensor) and published the reading to an MQTT broker running on the Raspberry Pi. A new script, typically named sensors.py, subscribed to the "esp32/#" MQTT topic, decoded the moisture readings, and made them available for real-time decisions. This approach required the Pi and all ESP32 modules to share the same Wi-Fi network. Battery life and wireless range emerged as practical considerations—especially for large greenhouses—leading the team to standardize efficient sleep/wake cycles in the ESP32 code, and to ensure local Wi-Fi coverage was stable. Despite these challenges, the major advantage was clear: multiple pots or flats could be instrumented without stringing additional wires from the Pi to each container, reducing setup time and complexity. More details regarding data collection can be found in the next section.

By Version 03, the system needed to manage more than one or two valves, since different plant species or experimental groups demanded distinct watering schedules or fertilizer regimens. To enable multi-valve control while minimizing the Pi's GPIO usage, a CD74HC4067 multiplexer/demultiplexer was integrated. The Pi connected to S0–S3 for channel selection and drove a single SIG pin, which, when set HIGH, would open whichever valve corresponded to the selected channel. To handle the relatively higher power demands of the valves (often running at 9 V or 12 V), each valve coil was switched by a 5 V relay module or transistor circuit, with the multiplexer outputs triggering each relay input. This design permitted up to 16 valves under a single SIG line, which significantly simplified wiring for an expanding greenhouse. Physically, researchers also replaced the clear plastic containers with opaque or dark bins to inhibit algae growth, and they used ¼-inch grommets with lubricant to prevent cracks or leaks when routing tubing through enclosure walls. All electronics sat at the top of each box, above any accidental spills or condensation, thereby protecting the circuitry from corrosive fertilizers or moisture.

On the software side, Version 03 added a Flask-based web application for real-time monitoring and control. The Pi's app.py launched a local server that displayed live sensor data (transmitted from the ESP32 modules) using Server-Sent Events. Users could open a simple webpage on a laptop, tablet, or smartphone and see each pot's moisture reading, the status of each valve, and historical logs. A set of on-screen buttons allowed manual overrides, letting the user open or close a specific valve at will. Data logging expanded to track each sensor reading (time, pot ID, moisture level) and each actuator event (time, valve ID, duration). Over time, these logs provided insights into daily water usage and potential correlations between moisture readings and plant growth metrics like leaf count or biomass.

Finally, Version 04 outlined the conceptual framework for more sophisticated control. Here, additional sensors would measure greenhouse conditions such as temperature, humidity, and CO_2. Combining these data with the soil moisture readings would enable more advanced watering algorithms, potentially driven by predictive models or PID controllers. The architecture also supported the idea of maintaining a second reservoir or fertilizer injection system, allowing plants to receive water or nutrient solutions as needed. Future expansions would integrate additional notifications—like emails or text alerts—triggered whenever moisture or reservoir levels moved outside desired ranges. This version represented an aspirational vision for a highly automated greenhouse, wherein each plant or set of plants could be individually monitored and watered, freeing the research team from manual checks and ensuring more consistent plant care.

Through these iterative versions, the project addressed real-world greenhouse challenges, such as preventing leaks, minimizing fertilizer-related corrosion, improving sensor placement, and allowing remote monitoring. Each new iteration of hardware and software built upon the lessons learned from the previous one. In this way, the system advanced from a single wired sensor/valve prototype to a multi-sensor, networked irrigation controller capable of scaling to larger greenhouse applications.

Data Collection

The most important feature that our team would like our project to accomplish is the ability to transport water from a reservoir, through a specific waterway, to a potted plant. This water reservoir can be a large bin with one end for the water inlet and the other end for the water outlet. For the inlet of our reservoir, we plan to have tap water. The water outlet of our reservoir will consist of 3 different pipes/hoses (for our small-scale prototype), each leading to a different potted plant.

Each outlet pipe is fitted with a 12 V DC solenoid valve. To switch these valves automatically from the Raspberry Pi, we route each one through a relay connected to an external 12 V supply (the Pi itself can deliver only 3.3 V logic and limited current). The relay control lines are driven through a 16-channel analog multiplexer/demultiplexer (74HC4067 [1]), which minimizes GPIO usage; if more channels are needed, we can cascade additional 74HC4067s or select a larger multiplexer.

The following figure shows the wiring diagram for the Raspberry pi, relays, boost converter, and solenoid valves.

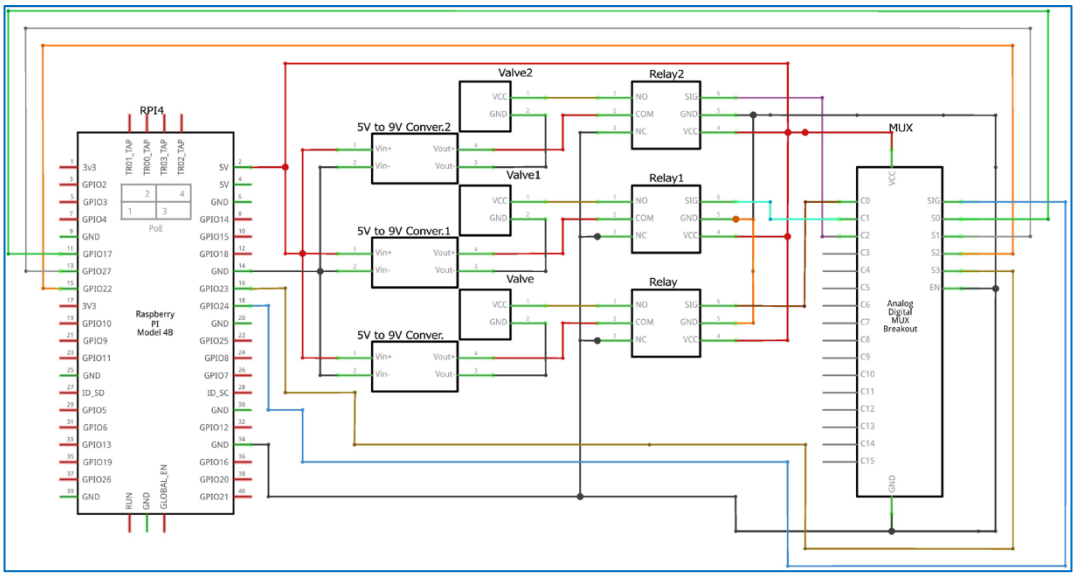


Figure 09 - Wiring Diagram for Water Control Module

The next feature is data gathering and wireless communication with the Raspberry Pi. While the system's core task is watering plants, it also needs to detect when plants require watering. To achieve this, we used three capacitive moisture sensors [2]. We chose capacitive sensors over resistive ones because fertilized water, which contains salts, can distort the readings of resistive sensors.

Each capacitive sensor has three pins: VCC, GND, and AOUT. The VCC pin connects to 3.3V from the ESP32, the GND pin connects to the ESP32's ground, and the AOUT pin outputs the analog moisture signal to be read. The sensor and its three-pin wiring are shown in figure 10 [2].

.

Figure 10 - 3 Wired Capacitive Moisture Sensor

Since our prototype features three potted plants, we used three sensors. To manage their readings efficiently, all three sensors are connected to a single analog input channel on the ESP32 using a multiplexer (MUX). This strategy, similar to how we control valves using a demultiplexer, keeps wiring neat and conserves ESP32 pins. Figure 11 shows the MUX/demux module [1].

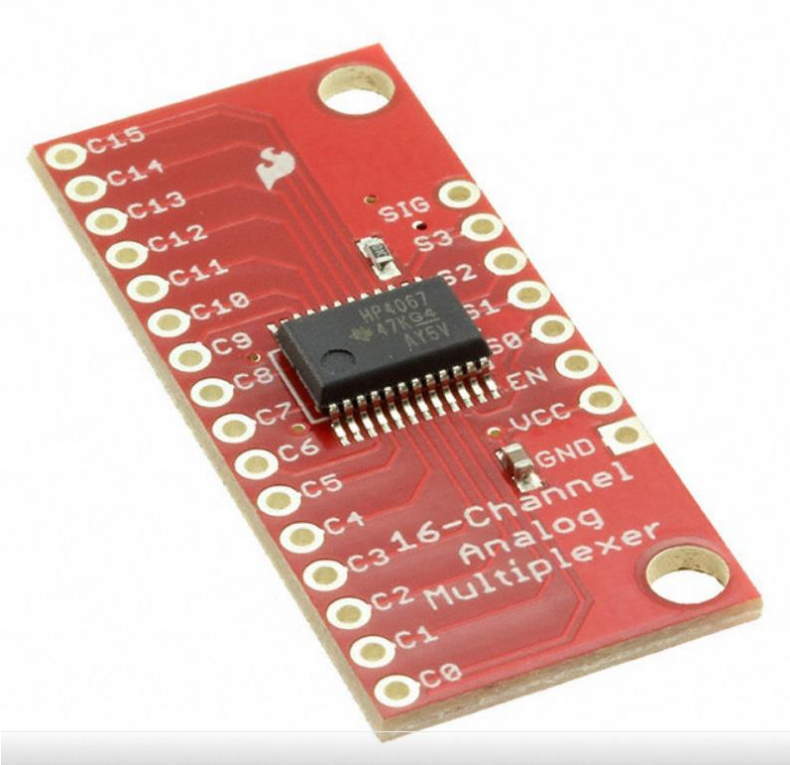


Figure 11 - MUX/deMUX Close-up View

All MUX/deMUX systems have three main parts: **inputs**, **outputs**, and **select signals**. A MUX has multiple inputs and a single output; a demux has a single input and multiple outputs. The MUX selects one input (e.g., c0 to c15) to pass to the output (**SIG**), based on four select signals (**s0 to s3**), which are controlled by the ESP32. The number of select lines is based on the number of inputs using:

$$n = \log_2 X$$

where n is the number of select lines, and X is the number of inputs. For 16 inputs, we need 4 select signals. These lines form a binary number with s3 as the most significant bit and s0 as the least. For instance, selecting input c4 (binary 0100) requires: s3 = 0, s2 = 1, s1 = 0, s0 = 0.

The final component of the data collection module is the ESP32 microcontroller. We selected the ESP32 over an Arduino because, while they share a development environment (Arduino IDE), the ESP32 supports Wi-Fi and Bluetooth communication. Initially, the ESP32 was powered via USB-A from a laptop, allowing us to program and monitor outputs. In the final build, it was powered independently via a 5V, 10W wall adapter. Once programmed, the ESP32 runs autonomously and communicates with the Raspberry Pi. Figure 12 shows an ESP32 board [5].

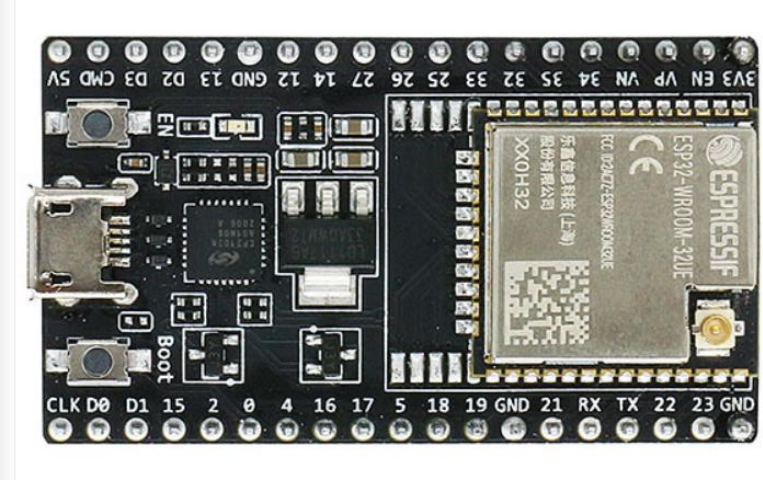


Figure 12 - ESP32UE Board

The first step in building the data collection system was to test the moisture sensors. We connected VCC to 3.3V, GND to ground, and AOUT to an analog-to-digital converter (ADC) pin on the ESP32. Because the ESP32 cannot process analog signals directly, the onboard ADC was used to convert them into digital values. These pins are listed in the ESP32 datasheet [5]. During testing, we found that sensor readings vary by hardware. The same sensor gives different raw values on an Arduino Uno versus an ESP32, even with the same power input. However, swapping one sensor for another of the same model yielded nearly identical results on the same device — meaning recalibration isn't required for sensor replacement on the same platform.

The capacitive moisture sensor was measured to be 3.75 inches in length. We determined that the maximum penetration depth into the soil, measured from the tip, would be 2.5 inches. To assess how the sensor responds to moisture at different positions, we conducted tests to determine whether the location of applied moisture affects the sensor's readings or if the output remains consistent regardless of placement. Understanding which part of the sensor is most sensitive to moisture is crucial, as it allows us to account for potential inaccuracies in the moisture readings. If the sensor's response is weighted toward a particular region, our readings may not accurately reflect the true soil moisture content, requiring adjustments to the watering algorithm to increase or decrease water delivery accordingly. To investigate this, we conducted two tests: spot measurement and enclosure measurement.

The spot measurement test involved using a wet paper towel to make contact with specific areas along the moisture sensor. At each location, we recorded the sensor's reading and then moved to the next position until we had collected data along the entire length of the sensor.

The enclosure measurement test consisted of wrapping the sensor in a wet paper towel, starting from the tip (the pointy end) up to a specific length. We recorded the sensor's reading, then continued wrapping further along the length, taking measurements at each step until the entire sensor was wrapped in the wet paper towel. Each of these tests was performed twice, resulting in a total of four data sets. To maintain consistency, we recorded the moisture sensor's readings at 0.5-inch intervals. Specifically, in each test, we started at the tip of the sensor, took a measurement, then moved up 0.5 inches, taking additional measurements until reaching 2.5 inches from the tip. The Matlab plot and the discussion of the enclosure and spot test can be found in the next section.

To convert raw readings into an interpretable format, we measured values for both saturated and dry soil. Using those, we defined:

$$readInterpret = \left[\frac{x_{dry} - x_{measure}}{x_{dry} - x_{wet}}\right] \cdot 100[\%]$$

Since dry readings produce higher values than wet readings, this equation maps any moisture level to a percentage between 0% (wet) and 100% (dry).

The ESP32 code was developed in two stages:

Stage 1: Reading Moisture Data

The loop section of our code contains three blocks — one per sensor. Each block calls a moisture reading function, which:

- Accepts the sensor channel
- Reads raw data
- Converts it to a percentage using the equation above
- Averages three readings to reduce noise
- Optionally prints the raw and processed values

The delay time function's purpose was to adjust the frequency the ESP32 would read from a specific sensor, that is, the ESP32 would not enter one of the three blocks if time since the ESP32 read from a specific sensor till the present time was not greater than the delay time. The original plan was to use this function to reduce the ESP32's power consumption by sampling a specific moisture sensor less frequently if that sensor was reading a higher moisture percent. For example, if we want a maximum delay to be 1 hour and the minimum delay to sample one of the three sensors was to be 5 seconds, then we would enter these numbers into the appropriate variables. The delay time function would then use an equation similar to how we obtained the moisture percentages and calculate a delay time that is equal to or between the max and min delay times based on how high the moisture percentage was. Thus, if sensor 2 was reading a high moisture percent, then the ESP32 would skip reading from sensor 2 and only read from sensor 1 and 0, assuming both sensors 1 and 0 have low moisture percentages and thus a small delay time. Although this feature is in the code, we decided against using this feature, because the ESP32 consumes around 78.32 mW, which is not a lot. In our final system, we decided to keep the delay to read from each sensor to around 5 seconds, meaning that the ESP32 would just sequentially read from each sensor with a period of around 5 seconds.

Stage 2: Wireless Communication

The ESP32 was connected to both:

- Wi-Fi, and
- An MQTT server hosted on the Raspberry Pi.

Using standard libraries (e.g., `WiFi.begin()`), the ESP32 connects to the same network and subscribes to the MQTT server. If disconnected, it attempts to reconnect automatically.
Each sensor block also includes a publish function. It sends the moisture percentage and sensor ID to a topic on the MQTT server. The Raspberry Pi, being subscribed to that topic, receives and logs this data. Each message is prefixed by the sensor ID — for example, `"02 52"` indicates sensor 2 reports 52% moisture.

Figure 13 shows a photo of the first prototype of the data gathering module where we only used one moisture sensor.

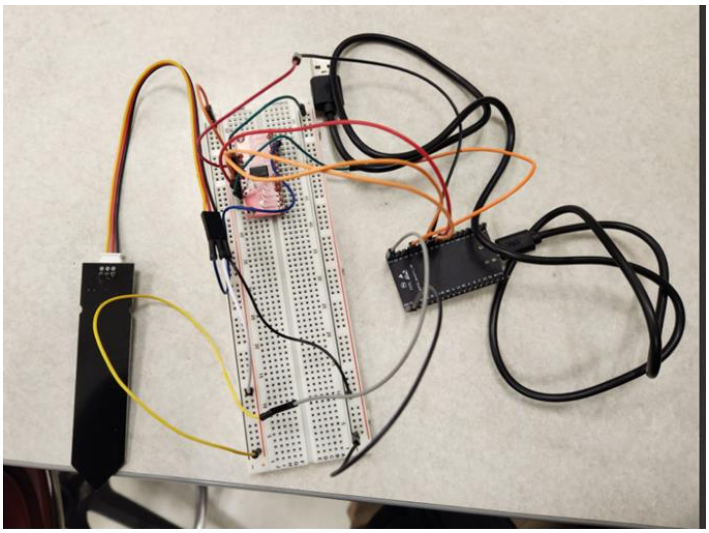


Figure 13 - First Prototype of the Data Gathering Module

The following Figure shows the wiring diagram for our data collection module.

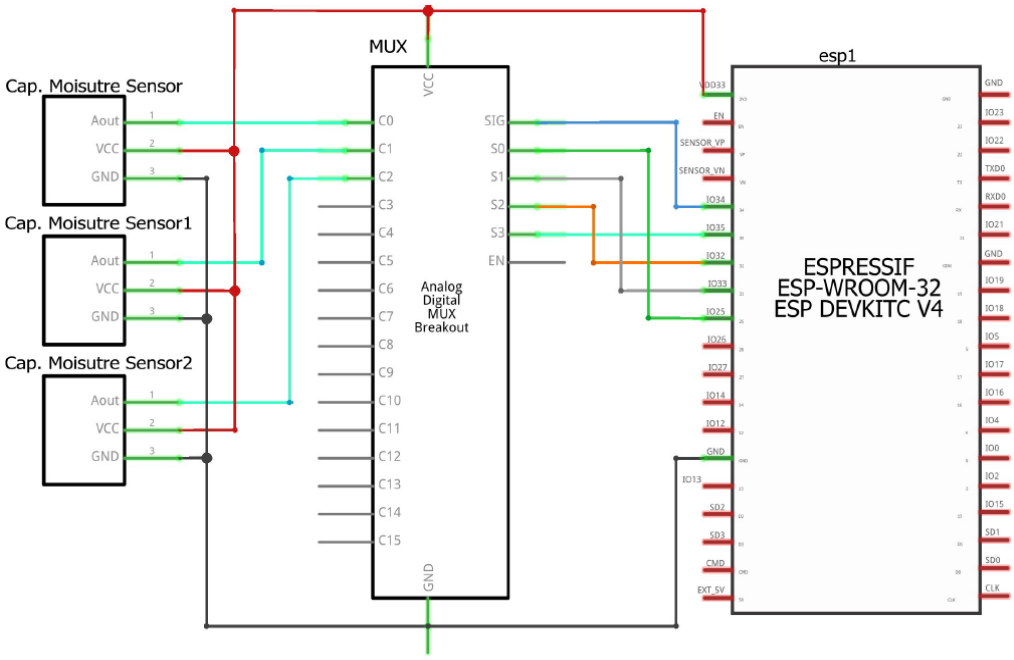


Figure 14 - ESP32 Wiring Diagram

The third feature we aim to implement is an automated system that determines which plant needs water and delivers the appropriate amount accordingly. The Raspberry Pi will host the algorithm responsible for assessing moisture levels and making watering decisions. While specific details—such as manual scheduling and maximum moisture thresholds for different plants—will be incorporated based on the available development time, the general flow of the algorithm will be as follows:

- Create an array that holds the moisture data and the valve control of a specific plant.
- A closed loop P controller:
    - Determine if the moisture level of a plant is above a threshold set by the user.
    - If the moisture level is below the threshold, gather an amount of water into the reservoir from either tap or fertilized water, drain the reservoir, sending all of the water to the plant by opening the correct valve.
    - If moisture is really low send lightly more water (Proportional control).
    - Wait and see if the moisture level rises above the threshold. If not, repeat the watering process until the desired moisture level.
- If no plants need watering, keep reading the sensor data till the plant's moisture level falls below the threshold.

Figure 15 shows the block diagram that tells the general function of our system and how it ties together.

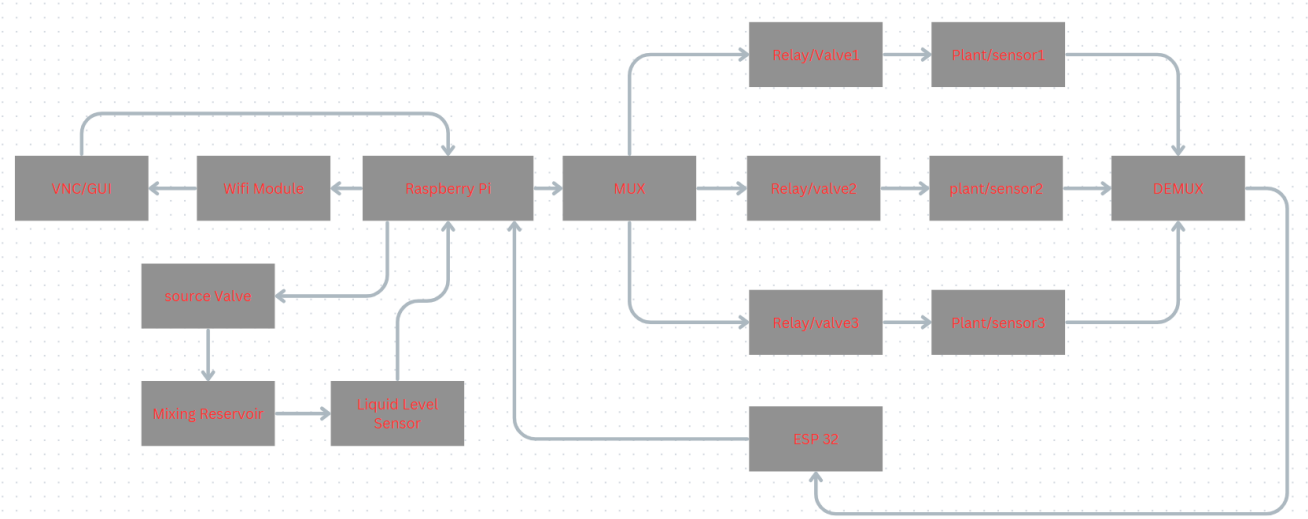


Figure 15 - Block Diagram of How the System Components Work Together

Results

Figures 17 show the system's final design with the electronic hardware soldered onto Perfboards. **Figure 16** shows an image of the soldered data collection module where this package includes the ESP32 WROOM E model and MUX. From this package, we have two main inputs: wires extend to the three moisture sensors and a USB B micro wire to provide 5V with 10W of power to the module.

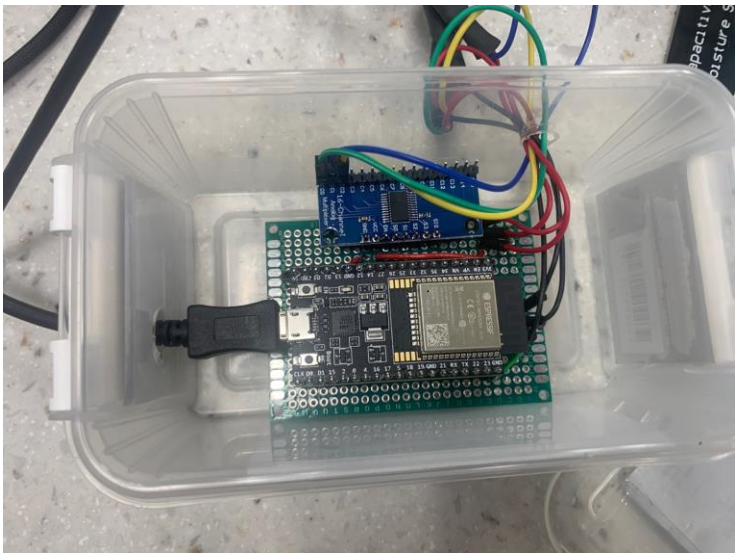


Figure 16 - Soldered Data Collection Module

Figure 17 shows the soldered hardware of hardware components of the irrigation system. The Perfboard contains wires to the Raspberry Pi, a MUX, as well as wires to the relays and

boost converters. This soldered package has an input of 5.1V with a minimum of 3A for power and output control signals to the solenoid valves.

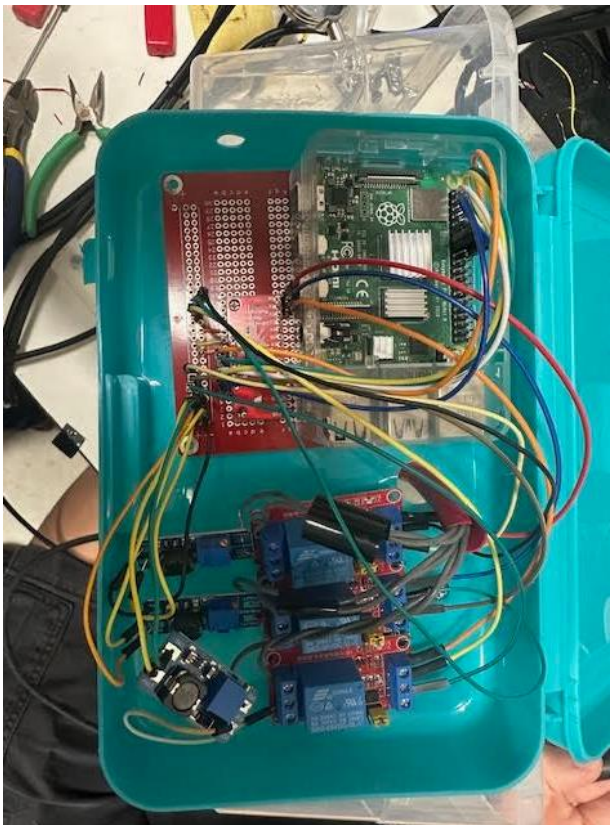


Figure 17 - Inside Hardware of Irrigation System

Figures 18, and **19** both showcase the tomato plants in a state of two extremes, one being overwatered and the other when is left to dry out. When looking at this data it is clear to see that the moisture sensor has a linear relationship between the moisture reading and the time that the plants are not being watered, assuming that the evaporation rate is constant which it probably is because the greenhouse keeps the environment at set parameters. This information tells that a tomato plant dehydrates linearly and that if the plant dehydrated quadratically or with a higher order, then this would be concerning as plants that need water may die quickly if our system is to malfunction and not deliver water.

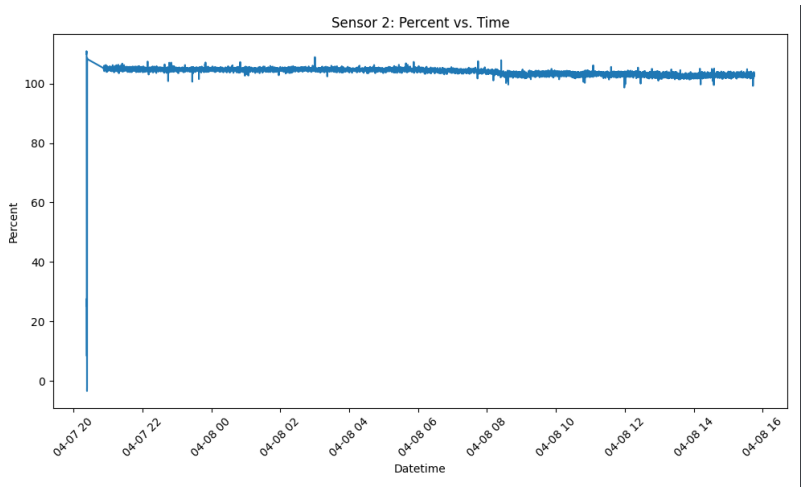


Figure 18 - Data Collected from an Overwatered plant

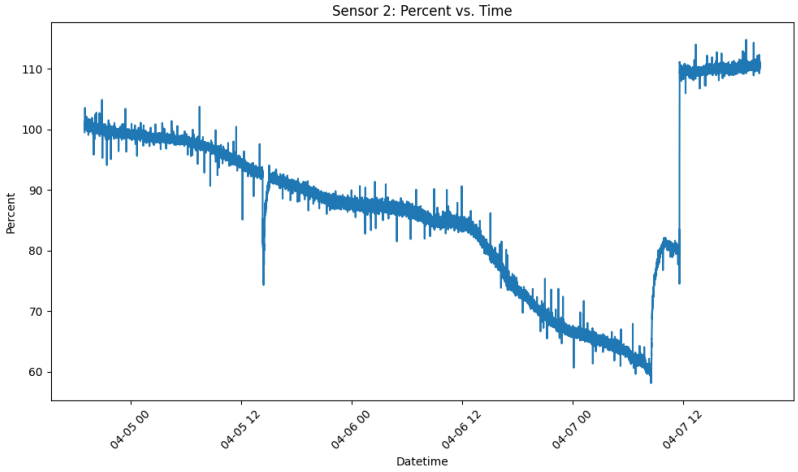


Figure 19 - Data Collected from a non-watered plant

To make sure the tomato plants achieve optimal watering, Benjamin Wolf was consulted and informed us of the ideal moisture level of a tomato plant which we were then able to conclude was within a 80%-75% moisture level . With that in mind, the system was programmed to keep an 80% moisture level in the tomato plants at all times.

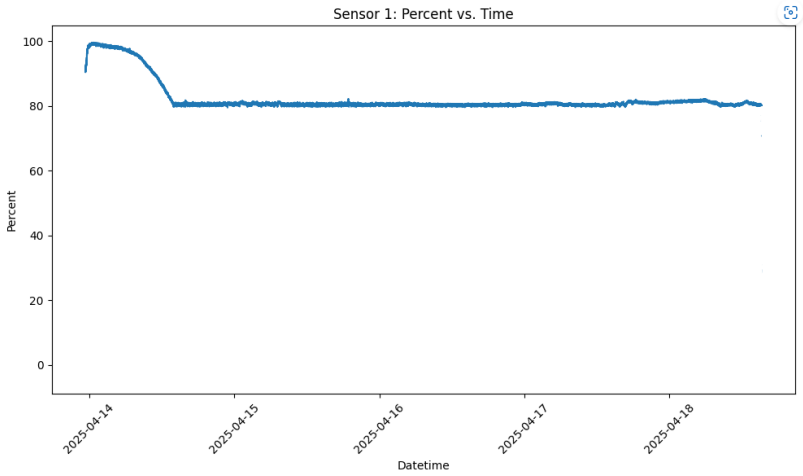


Figure 20 - Proof 80% threshold for ideal plant watering was achieved

Figure 20 not only showcases that our irrigation system was able to keep the target moisture level of around 80% for a prolonged period of time, but it also highlights the fact that the plant was healthy and alive at the end of its week cycle.

Discussion

A. Interpret Results

As seen in **Figure 20 a**bove, our system operated seamlessly. This data proves that not only are the moisture sensors reading correctly but they are also communicating smoothly to the controller. Furthermore, this also indicates that multiplexer systems used to operate the valves were working effortlessly.

Furthermore, we can say that our system is able to provide the appropriate amount of water to sustain a tomato plant over a prolonged period of time. This means that in regard to tomato plants, we would be able to provide a scalable, automated watering system which can rid the need to manually water each tomato plant. Tomato plants were an ideal standard to test our system against, as a tomato plant dramatically shows if it is underwatered or overwatered within a short amount of time. This meant that to sustain a tomato plant, our system had to keep each plant at a strict moisture percentage, demonstrating the functionality and cooperation of our data collection module and water actuation module of our system.

B. Unforeseen Results/ Minor Roadblocks

While working on the project the team faced many roadblocks. Once roadblock dealing with the weight of the plants and the other dealing with placement of the moisture sensor. As seen in **Figure 21** when beginning this design, we were alerted that the tomato plant had fallen over, which negatively impacted the moisture level readings as seen in **Figure 22**. This not only caused the readings to be off, but it also stopped the watering process;

thus, making the moisture level fall below the desired 80% threshold. Once aware of this issue we were advised to place wights in the plants, which ultimately helped.

Figure 21 - Fallen and dried out tomato plant

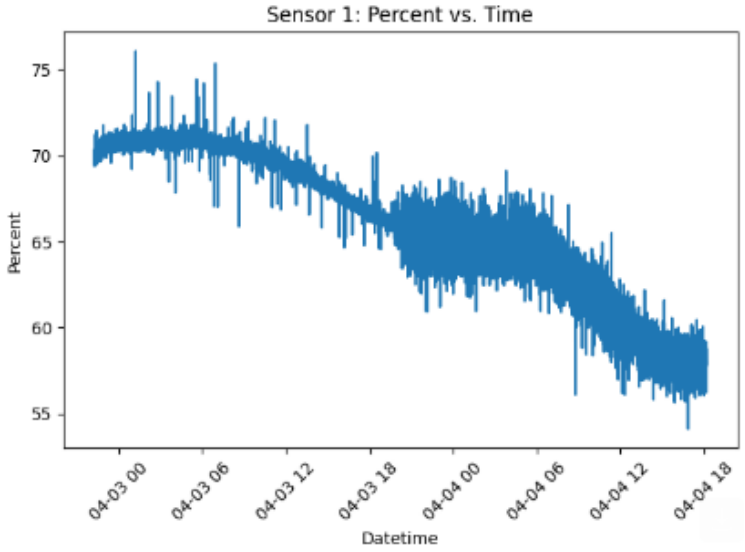


Figure 22 - Fallen tomato plant sensor data

Another issue that we faced happened during our final design, Version 03. When going to check in on the tomato plants we noticed that one of the plants, while reading a high moisture level was in fact dry. The reason being that the moisture sensor was too close to where the valve tube was inserted into the plant, By having the moisture sensor so close to where the water was being delivered to the plant it was not able to tell the actual moisture level of the soil because it itself was being watered on directly. To fix this issue, we then placed the moisture sensors on the

opposite side of the pot away from where it was being watered that way, so that gather more accurate readings.

C. Future Applications

While our system was only tested to sustain a tomato plant, our system's code can be modified to sustain other types of plants as well. Currently, our system is meant to keep its plant at a constant moisture level, however, other plants such as succulents will perish if held at a constant moisture percentage. To sustain these different plants, we could test the upper and lower limits of moisture percentage that the plant prefers, and then, we can test to figure out how many times a day the plant is to be watered.

In an email sent from Dr. Wolf, he mentioned that he noticed the soil of one of our plants was really dry, however, our system was able to provide just enough water to the plant so that it was healthy. Using our system, Dr. Wolf saw that our project could be potentially used in drought studies where our system would hold a plant in a state that simulates a mild drought. Because a research group in WashU is performing drought studies on plants, our system may be beneficial to their research. Currently, we are happy to provide information to the research team in regard to our system's hardware and software.

D. Satisfied Objectives

Overall, when looking at our results, it is clear that the objectives listed for Version 03 were met and ran as expected.

- ✓ Receives signals from the wireless moisture sensor module, forwarding them to the controller.
- ✓ Uses a multiplexer and relay system to open/close the in and out valves. This approach supports expansion to multiple valves for multiple plants.
- ✓ Monitors reservoir water level using a simple float valve sensor.
- ✓ Ensures each component (sensor, controller, valve) communicates smoothly to simplify troubleshooting and future upgrades.
- ✓ Waters and sustains Tomato Plants.

Conclusion

In this project, we successfully designed, built, and tested a prototype greenhouse irrigation system capable of maintaining the moisture levels of multiple plants automatically. Each incremental "version" of our design—from the initial wired sensor prototype (Version 01) to the semi-wireless (Version 02) and multiplexer-driven (Version 03)—demonstrated an increasing degree of sophistication in both hardware and software integration. Through capacitive moisture sensors, we obtained reliable soil readings, relaying these data to a Raspberry Pi via MQTT for real-time decision-making. By actuating 12 V solenoid valves

(powered through relays and multiplexers/demultiplexers), the Pi could selectively water specific plants, thereby preventing under- or overwatering.

We validated our approach by sustaining tomato plants at their ideal moisture level (~ 80 %) over extended periods, confirming the effectiveness of our sensor calibration and closed-loop control. Network challenges, placement of sensors in pots, and dealing with large or top-heavy plants all presented hurdles that informed later design refinements. Despite these setbacks, the project achieved its main objectives, including modular code, logged sensor data, and robust valve actuation. Although a fully polished remote GUI (Version 03) and advanced control concepts (Version 04) remain partially implemented, the groundwork is laid for future enhancements.

Overall, our results show that a scalable, automated irrigation system is not only feasible but also well-aligned with modern greenhouse management needs. The design can be extended to accommodate additional valves, sensors, or environmental parameters (e.g., temperature, humidity, or CO_2). In the long term, this platform could also incorporate predictive algorithms or integrate seamlessly with existing greenhouse automation tools. By reducing manual watering tasks and ensuring a consistent water supply tailored to each plant's needs, our system stands to improve resource efficiency, plant health, and the overall productivity of greenhouse operations.

Deliverables

A. Design and Documentation

We proposed to include system architecture diagrams, hardware schematics, and detailed software documentation for Versions 01, 02, and 03. Each version's documentation would specify wiring layouts, sensor configurations, and controller scripts or firmware settings, ensuring clarity for any team member or end-user setting up the system.

In the end, we were able to present the wiring diagrams, and the code needed to replicate our final product. While we will not present the wiring or the code needed to replicate the system in the different prototype iterations, our final system builds off prototype iterations, meaning it would be trivial if one wanted to build earlier versions of our system.

B. Functional Prototypes

We proposed that each version of the system would be demonstrated with a physical working model. These prototypes would showcase how soil moisture data is collected, processed, and used to trigger water flow. Any modifications or improvements introduced in later versions—such as wireless sensor capability—would be clearly illustrated in the final assemblies.

In the end, we were able to build and test prototype version 01 – 03 where we accomplished the main purpose of each version. Version 04 was not built as it was a prototype that we could build if given extra time. A feature that we tried to complete as a part of Version 04 was the GUI. In the end, we were able to put some work into GUI,

however, we did not have enough time to build it to a point where we could remotely adjust our system's settings.

C. User-Friendly GUI

We proposed a polished graphical interface (in Version 04) to be delivered to allow real-time monitoring and control. Through this interface, users would be able to view sensor readings, set irrigation thresholds, and manage water flow over any device with internet access. The GUI's design would emphasize ease of use, providing intuitive dashboards and alert systems.

In the end, we planned the layout of the GUI (draft seen **Figure 26** and **27** of the appendix) and implemented it with some code which yielded the results seen in **Figure 23**. The Code can be seen in **Figure 28** and **Figure 29** in the appendix**.** While we were able to build a good foundation for the GUI, we were unfortunately unable to complete this feature due to time constraints.

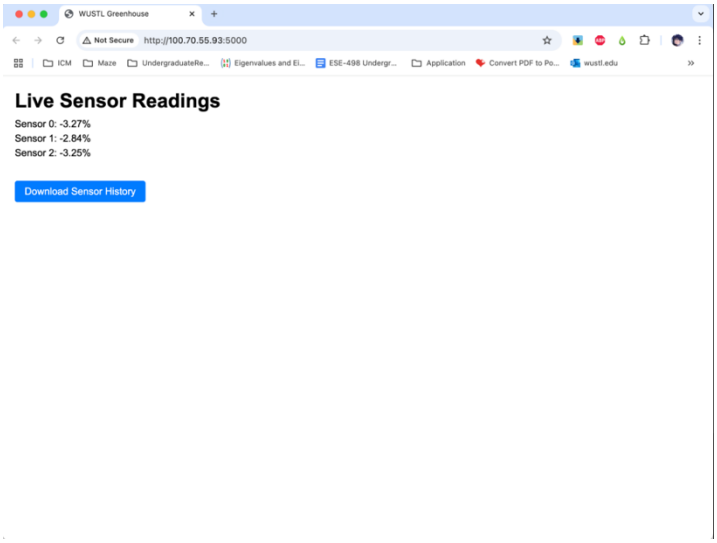


Figure 23 - Image of the GUI in Web browser

Schedule/Timeline

Figure 9 shows the Gantt chart of the project timeline in our proposal.

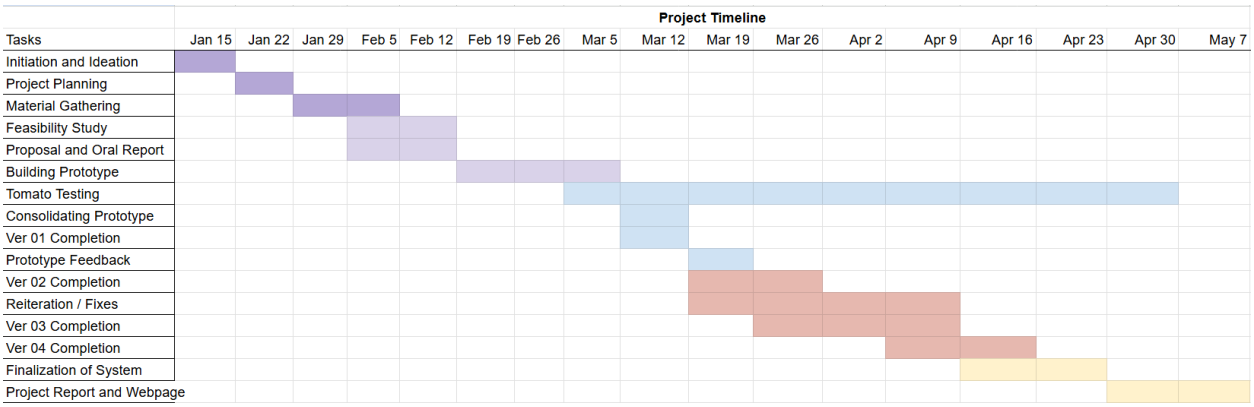


Figure 24 – Gantt Chart of Project Timeline

In retrospect, our team was able to adhere to the schedule detailed in the Gnatt chart above. While we were able to complete each of the versions of the system, excluding version 4, near their stated dates, we did not account for extra time that would be needed to create detailed documentation for each version. Of all of the tasks seen in the chart, the "Tomato Testing" task is something we wished we could have redone. Because the growth cycle of a Tomato plant is around 10 weeks, we hoped that we could implement our system to sustain the tomato plant throughout all 10 weeks. The problem was that our system was not able to water any kind of plant until around the completion of Ver 01 of our system. In retrospect, it would have been ideal if we could have ideated and built our first prototype faster than we did so that we would be able to see the full effects of our system on the tomato plants.

Contributions:
Chengyu:

- Proposal
- Version 01 Control Code
- Version 02 Control Code
- Version 03 Control Code
- Version 04 GUI Prototype Code
- Final Report

Kylee Bennett :

- Build Prototypes (Version 01,02,03)
- Worked on Initial GUI
- Soldering Hardware
- Made Poster
- Final Report

David:

- Proposal
- Data gathering Module
- Soldering hardware onto Perfboard.
- Hardware Wiring Diagrams
- Final Report

Appendix

Items	Quanty	Cost
Water Level Sensor		
1/4" Valuves	1	$8.99
Power supply (12V, 0.46A)	1	
1/2" Valuves	1	$9.99 for 2
Power supply (12V 0.42A)	1	
100ft 1/4' '1/2" Tubes	2	$11.99/14.99
1 buckets	1	$1.99
Rasberry Pi	1	
Epoxy	2	$11.96
Capacitive moisture sensors	5	$9
MUX/DEMUX	1	
DEMUX Input Vcc: 2 to 6V; input V: 0 to Vcc; output V: 0 to Vcc Good for raspberry pi	1	$1.12
A to D chip for reading moisture sensor (MCP3008)	1	$3.12
ESP32 microcontroller for Wireless stuff	1	$10
USB A to USB micro B for ESP32 don't buy. Have in lab	1	4.99%
Raspberry pi Relay for Solenoid valves	1	10.99 for 2

Figure 25 – BOM

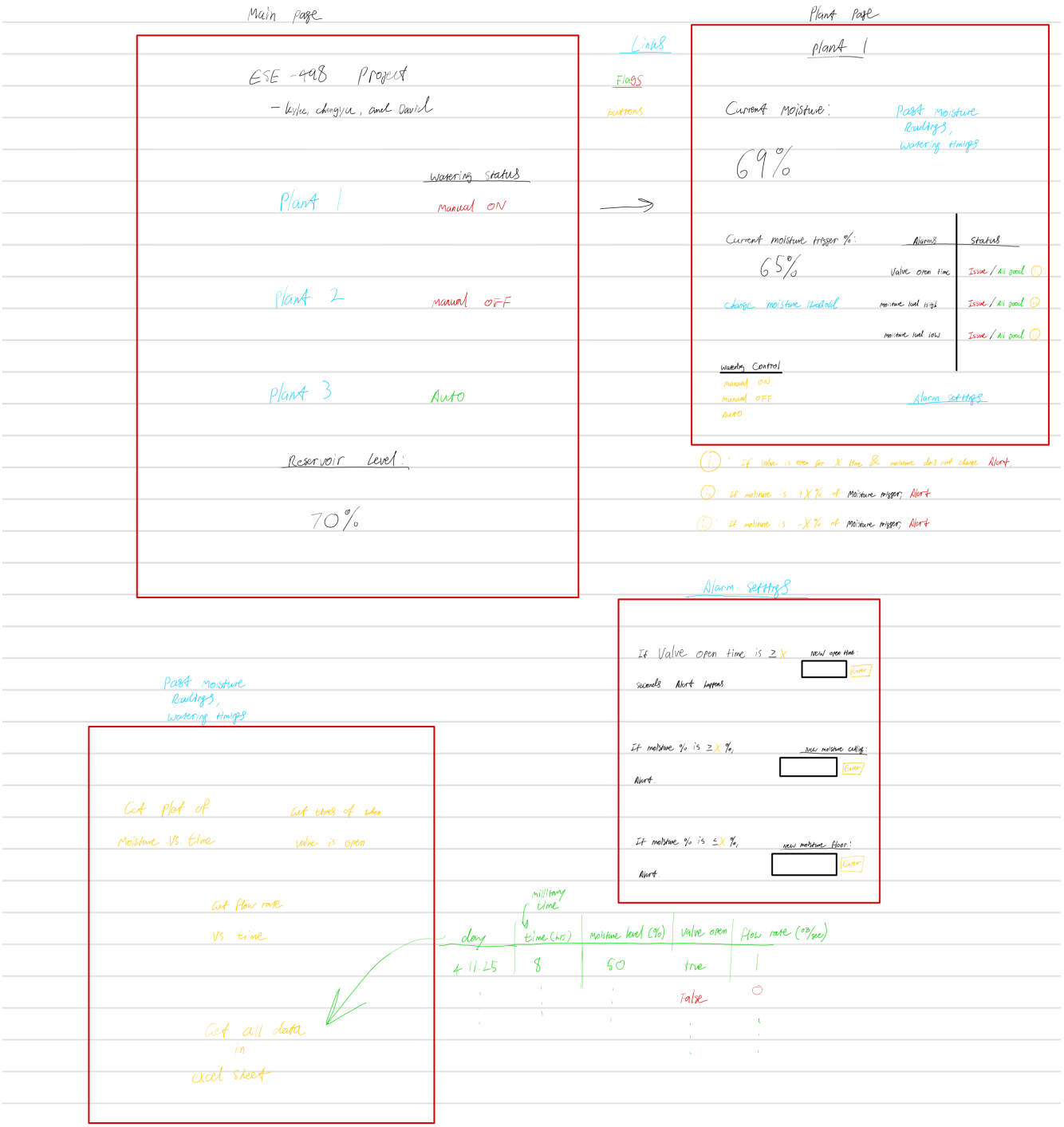


Figure 26 - GUI Layout Part 1/2

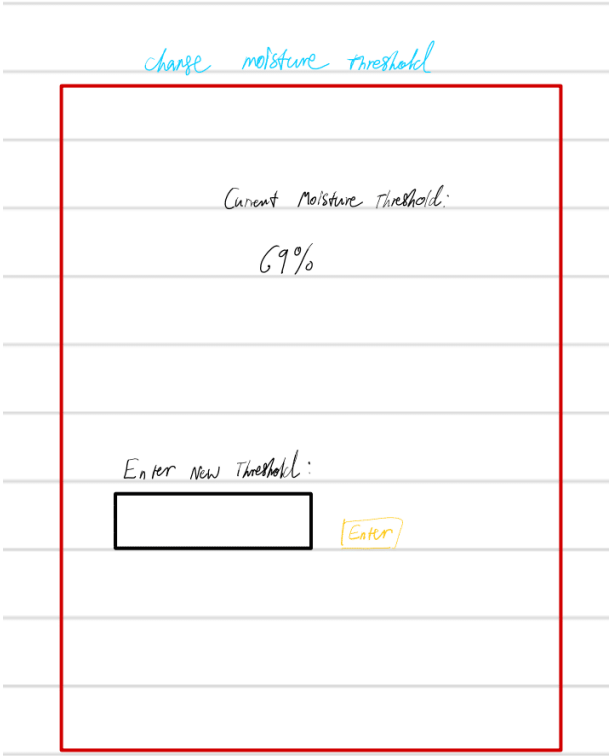


Figure 27 - GUI Layout Part 2/2

Figure 28 - GUI Code

templates > index.html > ...

```html
<!DOCTYPE html>
<html>
<head>
    <title>WUSTL Greenhouse</title>
    <style>
      body { font-family: Arial, sans-serif; margin: 20px; }
      h1  { margin-bottom: 10px; }
      .sensor-box { margin: 5px 0; }
      .download-button {
        display: inline-block;
        padding: 8px 16px;
        background: #007BFF;
        color: #fff;
        text-decoration: none;
        border-radius: 4px;
        margin-top: 20px;
      }
      .download-button:hover {
        background: #0056b3;
      }
    </style>
</head>
<body>
    <h1>Live Sensor Readings</h1>

    <!-- Sensor placeholders: update them in real-time via SSE -->
    <div class="sensor-box" id="sensor0">Sensor 0: No data yet</div>
    <div class="sensor-box" id="sensor1">Sensor 1: No data yet</div>
    <div class="sensor-box" id="sensor2">Sensor 2: No data yet</div>

    <!-- Download CSV file -->
    <p>
      <a class="download-button" href="/download-sensor-history">
        Download Sensor History
      </a>
    </p>

    <script>
    // Create an EventSource to listen to /sensor-updates
    const evtSource = new EventSource("/sensor-updates");

    evtSource.onmessage = function(event) {
        // event.data is a JSON string, e.g. {"0": 44.7, "1": 33.1}
        let readings = JSON.parse(event.data);
        for (const sensorId in readings) {
```

Figure 29 - HTML

References

[1] 74HC4067 Eval board. MUX/DEMUX.
Available: https://www.digikey.com/en/products/detail/sparkfun-electronics/BOB-09056/5673767. [Accessed: Feb. 24, 2025].

[2] Capacitive Soil Moisture Sensor Module (5-pack). [Online].
Available: https://www.amazon.com/dp/B07SYBSHGX. [Accessed: Feb. 11, 2025].

[3] Reddit user. *Capacitive Soil Moisture Sensor Gives Out After About a Month?* Reddit, 25 Aug. 2020.
Available: https://www.reddit.com/r/arduino/comments/ii2ab0/capacitive_soil_moisture_sensor_gives_out_after/.

[4] *Argus Titan System*. Argus.
Available: https://www.arguscontrols.com/en/products/titan/. [Accessed: Feb. 27, 2025].

[5] "ESP32-DEVKITC-32UE – ESP32-WROOM-32UE 4MB Flash Evaluation Board." DigiKey Electronics, Espressif Systems.
Available: https://www.digikey.com/en/products/detail/espressif-systems/ESP32-DEVKITC-32UE/12091813. [Accessed: Apr. 23, 2025].

[6] Bedean 12V ¼" Inlet Water Solenoid Valve. [Online].
Available: https://www.amazon.com/dp/B07NWCQJK9. [Accessed: Feb. 11, 2025].

[7] HiLetgo 5pcs Capacitive Soil Moisture Sensor Module. [Online].
Available: https://www.amazon.com/dp/B07KCGYQVD. [Accessed: Feb. 11, 2025].