GVUS capstone:  
Project greenthumb

A Semi-Automated IoT Gardening System

Khiem H. Nguyen

11/6/2025

Table of Contents

[Introduction 2](#_Toc214401973)

[Requirements 4](#_Toc214401974)

[Capstone Requirements 4](#_Toc214401975)

[Stakeholder Requirements 4](#_Toc214401976)

[Stakeholder: Professor Fredericks 4](#_Toc214401977)

[Stakeholder: Jay von Rosen 5](#_Toc214401978)

[Stakeholder: Khiem Nguyen (Me) 5](#_Toc214401979)

[Design and Implementation 7](#_Toc214401980)

[Embedded Hardware and Application 7](#_Toc214401981)

[Embedded System + Circuitry 7](#_Toc214401982)

[IoT Implementation and IoT Cloud Service 8](#_Toc214401983)

[Embedded Application 8](#_Toc214401984)

[Frontend Service: Web Application 9](#_Toc214401985)

[Backend Service: RESTful API w/ Agentic Chat Service and Adafruit Service 10](#_Toc214401986)

[LLM 11](#_Toc214401987)

[Deployment 12](#_Toc214401988)

[Results 12](#_Toc214401989)

[Challenges 27](#_Toc214401990)

[Embedded Hardware 27](#_Toc214401991)

[Embedded Application 27](#_Toc214401992)

[Frontend Service 28](#_Toc214401993)

[Backend Services 28](#_Toc214401994)

[Cloud Services 29](#_Toc214401995)

[Others 30](#_Toc214401996)

[Proposed Future Improvement 30](#_Toc214401997)

[Embedded Hardware 30](#_Toc214401998)

[Embedded Application 30](#_Toc214401999)

[Frontend Service 31](#_Toc214402000)

[Backend Service 31](#_Toc214402001)

[Conclusion 32](#_Toc214402002)

# Introduction

A functional semi-automated IoT gardening system was proposed and built for the capstone project. Fondly nicknamed Project Greenthumb, it is both an academic and a personal endeavor.

The academic goal of the capstone project is for students to develop a functional, high-end application that draws on and incorporates multiple topics or concepts learned throughout the graduate program.

The personal goals of this project are creating an application or system that 1) incorporate some personal hobbies or interests, 2) has a practical and functional use and purpose, 3) allows me to explore and learn new challenging and enjoyable programming skills and concepts not taught in the graduate program and learned at work as a mean to stay competitive, relevant, and flexible in the job market, and 4) let me have fun.

Gardening was selected for this project because it's a lifelong hobby from childhood and merges my love for science and biology. Gardening also has many benefits; it promotes self-reliance, can offset food scarcity and insecurity by providing a steady source of sustenance and even income if those products are sold, generates green space, provides therapy, and promotes mental well-being.

However, some people may not have the time, knowledge, or ability to tend a plant, let alone a garden. Thus, I wanted to build an open-sourced, semi-automated IoT gardening system that could help grow plants with minimal human involvement to address the issues stated and make gardening and its benefits more accessible to people. The gardening system is intended to be low-cost, customizable, repairable, and non-proprietary, unlike the commercial alternatives. And while the gardening system won't be production-ready by the end of the capstone project, it is still functional and usable, serves as a sound proof of concept, and lays the groundwork for future enhancements.

To achieve the objective of building a semi-automated gardening system, I had to explore and learn about embedded and IoT design, basic circuitry, the MQTT protocol and services, and agentic design and orchestration; all of which were neither taught in my courses at GVSU nor used at work.

The gardening system is comprised of three components: an embedded hardware component and two software components. The embedded and hardware components consist of a small plastic container to house the plant and the water supply, a microcontroller, multiple sensors, a pump, and an LED growlight. One of the software components is a responsive frontend web application, built with Angular and PrimeNg UI library, as a means to retrieve and display sensor data information from the system and provide a chat messenger for users to communicate and handle conversations with an LLM that can answer questions relevant to the project, the sensor data, or general plant information. The other software component is a backend RESTful API microservice built using ASP.NET that connects to two different cloud platform to retrieve data, processes it, and send it to the frontend application.

In total, the gardening system uses four cloud platforms for its services. The two cloud platforms the backend service communicates with are 1) Microsoft Azure to access OpenAI's affordable LLM Model, o3-mini, hosted on it, and 2) Adafruit IO to access the sensor feed stored in its database. The third cloud platform is Google Firebase, which enables user authentication and authorization for the frontend application. The last cloud platform used was Google Cloud to host the frontend application and the backend service, making it publicly accessible.

And while not implemented, given the scope and time constraints of the capstone project, I explored the feasibility of running the local models. I concluded that a paid cloud platform and AI services were not needed for this project, and that local databases and LLMs could be used instead.

The stakeholders of the capstone project are my advisor, Professor Fredericks, who reviewed and approved the scope and proposed features in the project and options for my design choices; my boss, Jay Van Rosen of JvR Enterprises, who approved and encouraged me to do some research and design on agentic solution, Semantic Kernel SDK, Kernel Memory, and Retrieval Augmented Generation (RAG) on company time; and myself, for the reasons stated above.

The project did not use any formal software engineering methodology. Still, the methodologies that best fit the software development process were the Exploratory and Iterative Development models, which relied heavily on prototyping and building small, throwaway applications to build knowledge of various topics where I lacked experience and to refine the project's scope and feasibility based on the knowledge gained. This was especially true for the embedded and agentic aspects of the project.

# Requirements

The capstone project has many objectives and requirements. These requirements were defined by the capstone's purpose and requested by the project's stakeholders.

## Capstone Requirements

As previously mentioned, the capstone project requires students to develop a functional, high-end application that draws on and incorporates multiple topics or concepts learned throughout the graduate program.

## Stakeholder Requirements

The capstone project has three stakeholders: my advisor, Professor Fredericks; my boss, Jay von Rosen of JvR Enterprises; and myself. Each stakeholder proposed features for the gardening system and its design requirements.

### Stakeholder: Professor Fredericks

My advisor, Professor Fredericks, reviewed and approved the scope and proposed features of the capstone project, as well as the options for my design choices. Through our discussion over the summer, we agreed that if the semi-automated IoT gardening system were to be approved as a capstone project, it should meet and fulfill the capstone requirements, be complete, functional, and polished.

The capstone project should include an embedded component or system that collects data from plants via sensors and provides a means to act on that data. To make the gardening system IoT-enabled, the data and information should be able to be exchanged between the embedded system and an external system over the internet.

The gardening system should also have a UI to display the collected sensor data; for this requirement, a frontend web application with a dashboard page or several dashboard pages was proposed. This dashboard component of the web application will be one of the page’s key features.

Professor Fredericks also proposed that the web application include a mechanism to authenticate users and authorize them to access the application. For this to occur, the web application should consist of pages for user login, logout, and signup, as well as a backend service to manage and maintain user identities and accounts.

### Stakeholder: Jay von Rosen

My boss, Jay Van Rosen of JvR Enterprises, approved and encouraged me to research and design agentic solutions, the Semantic Kernel SDK, Kernel Memory, and Retrieval Augmented Generation (RAG) on company time, and supported my decision to incorporate some of those features into my project.

We discussed and agreed on several features for the gardening system. We proposed that the frontend web application include a chat messenger component that can communicate with an LLM. To ensure the LLM returns responses for prompts related to the gardening system or plant information, a RESTful agentic backend system should be implemented.

The plant information provided by the backend service could be formatted so it can be manually entered as threshold values for the embedded system.

Jay also requested that the agentic backend system analyze and/or summarize the sensor feed data.

### Stakeholder: Khiem Nguyen (Me)

Besides my personal objectives for the capstone project, I also had additional design requirements for the semi-automated gardening system that were not proposed or requested by the other stakeholders.

#### Embedded System and Growing Setup

From my years of gardening experience, I learned that a plant's growth and well-being depend on several factors: enough water to maintain an ideal soil moisture level, an adequate amount of (full-spectrum) light in terms of both duration and intensity, a suitable growing medium and nutrients, and an environment with the proper temperature or humidity.

So, in terms of sensors, I want ones that can measure most of those requirements. I want to use sensors to measure environmental temperature and humidity, the light intensity the plant receives, and soil moisture. Also, I want to find a way to measure the plant's growth over time and its water supply level.

Along with sensors, the gardening system should have a means to pump water from the water supply to the plant when the soil moisture level is below a minimum threshold. I also want the gardening system to have a grow light to supplement the plants with additional light when light intensity falls below a minimum threshold for a set duration.

The capstone gardening system will use commercially available, inexpensive hardware components to ensure the embedded component of the project is affordable, modular, upgradable, replaceable, and repairable, unlike many commercial automated gardening systems. Many readily available commercial automated gardening systems are expensive, costing hundreds of dollars, and are unrepairable, with proprietary parts and supplies that may not be commercially available for purchase. To offset this, I plan to utilize commercially available components from popular resellers, such as Amazon and Adafruit.

Also, from experience, I knew that for this capstone project, I want to grow my plants in soil rather than a hydroponic system. Growing plants in soil is a lot cheaper and requires less maintenance than in a hydroponic system. In a hydroponic system, nutrient supplies are more expensive, water must be distilled and oxygen-rich, and the entire system must be kept hygienic and cleaned often.

For the project, I decided to grow cat grass. It sprouts quickly and has a 1-2 week growing period. So, the gardening system can undergo multiple growing sessions and be refined if needed.

#### Frontend

The web application will have four main pages: the landing page, the dashboard page, the chat page, and the auth page for logins, logouts, and sign-ups. The landing page will highlight the web application's main features. The dashboard page will display sensor feed data in a graph and a table. The chat page will include a simple, usable chat messenger. The chat history should persist only for the duration of the user's session and can be cleared if desired.

The auth page(s) should include a means for users to log in, log out, and sign up. The user's authentication should persist for the duration of their login. The user is no longer authenticated after logging out. If the user is not signed in, they can only view the auth page. If the user is signed in, they can access all pages.

Besides being complete, functional, and polished, the web application must also be user-friendly and intuitive. The web application must have a consistent color palette and theme throughout the website. The web application must also be responsive and work on various screen sizes.

#### Backend

The backend service should be RESTful and have endpoints that are publicly accessible. The backend services should have three main functionalities: 1) To retrieve feed sensor data from an external source and process and format it to be usable and readable once it is passed to the frontend application, 2) To take a user prompt related the capstone gardening system or plant information and provide a tailored response related to the prompt, and 3) ensure the agentic component of the backend service has the guardrails to only answers questions relevant to the project and provide a response accordingly for irrelevant prompts or errors.

# Design and Implementation

The design and implementation of the semi-automated gardening system can be broken into three major subsystems: The embedded system, the frontend web application, and the backend application.

To compensate for my limited knowledge and background in Embedded Systems and Design, C++ Programming Language, IoT Principles and Design, and Agentic Orchestration, I began my capstone project a few months before the Fall 2025 academic semester.

 Shortly after Professor Fredericks approved the capstone project idea, I began R&D, prototyping, and design for the embedded component in July 2025. Then I started a similar process for the agentic orchestration system in August 2025.

In my short time doing R&D, prototyping, and building a throwaway application, I expanded my knowledge base, developed reusable code templates for my capstone project to shorten development time, and refined my approach to designing and architecting my application.

## Embedded Hardware and Application

### Embedded System + Circuitry

For the embedded system, a microcontroller is needed. It is the brain of the embedded component in the gardening system. The microcontroller is a small computer with an integrated circuit, RAM, processor, and I/O pins (IBM) required to connect the sensors and associated devices physically. The microcontroller can be programmed to perform functional logic to capture sensor data and communicate with external services via various communication modules and protocols, such as Wi-Fi/Internet or Bluetooth.

For the capstone project, I was split between the ESP32 and the Arduino Uno R4. I ultimately selected the Arduino Uno R4 over the ESP32 because it is 1) user-friendly, 2) has a lot of official documentation, 3) can supply both 5v AND 3.3v of power, which allows more flexibility in terms of sensors and devices I can use in the gardening system, and 4) has the I/O peripherals pre-soldered with female pin headers.

The microcontroller was also paired with a mini-breadboard, which was used to build the electronic circuits for the various sensors and devices (pump and growlight) without soldering. Jumper wires, the breadboard, and splicing connectors were used to connect and complete the circuitry. Only a few components were soldered, such as the growlight’s wires, after I removed its internal circuit board. It was soldered with a tin-based and lead-free solder.

The decision to use a breadboard and leave most components unsoldered was intentional, given the faultiness of the sensors and wires, to make the system easier to replace, and also to ensure it remains easily configurable and modifiable.

In terms of sensors, four types were used in the project. The DHT11 Sensor measures an environment’s temperature and humidity. The photoresistor can measure the light intensity of an environment. The soil conductivity sensor can measure soil moisture content. The ultrasonic sensor, which is typically used to measure distance by emitting a sonar signal and detecting it, measures the time it takes to travel to and from the sensor. This ultrasonic sensor was used to measure both the plant height and the water supply level in a container in the project.

All analog sensors were calibrated, and the readings were mapped accordingly to the upper and lower bounds of the target ranges.

### IoT Implementation and IoT Cloud Service

To connect the embedded system to the internet, make it IoT-enabled, and send data to a remote database, the Adafruit IO Cloud Service was used. AdafruitIO serves as an MQTT broker and a cloud database.

Several built-in classes and 3rd-party libraries were used in the Arduino code to establish a connection to the AdafruitIO Cloud Service. The "WiFiServer" class was used to set up and establish a WiFi connection to the local home router. The "WiFiSSLClient" library and class were required and used to establish an SSL (Secure Socket Layer) connection to the Adafruit IO server. AdafruitIO requires an SSL certificate to establish a secure, encrypted connection to their server.

The "Adafruit\_MQTT\_Client" is a library and class used to establish and manage network connections and communications between the Arduino and Adafruit IO (MQTT broker). The "Adafruit\_MQTT" library contains all of the MQTT logic that allows the board to publish data to and subscribe from the Adafruit IO (MQTT broker).

### Embedded Application

The embedded system was coded in C++ in both Visual Studio Code and the Arduino IDE, and the compiled code was sent to the microcontroller via the Arduino IDE and a USB connection. The embedded application has a bunch of variables and class objects at the top of the program that are declared, initialized, and/or instantiated. It is at the top of the program that the sensor threshold variables were established and that the WiFi Server, WiFi SSL Client, Adafruit MQTT Client, and various Arduino Task Timer objects were instantiated.

The embedded application also has two main functions that are called automatically at runtime: setup() and loop(). The setup function is called once at the start of the program, and the loop function is executed continuously after it.

The setup() function in my embedded application was used to 1) initialize serial communication, 2) setup and establish the WiFi connection to the router, 3) registers Adafruit IO's CA certificate, 4) setup the pins connection and mode for the sensors, 5) request test readings from all of the sensors and serializing the value, and 5) instantiate any class object that may was only initialize at the top of the program.

The loop() function in my embedded application was used to periodically 1) capture sensor readings and serialize them, 2) publish those sensor data to the Adafruit IO, and 3) manage the grow light and pump trigger events and activation duration, all on different concurrent timers.

The embedded application utilized three custom classes that I built: "ArduinoTimerTask," "Adafruit\_Helper," and "Unit\_Converter."

The custom class "ArduinoTaskTimer" was created to streamline the process of creating, checking, and resetting a timer. The class provides a reusable, cleaner code base. It allows the application to have multiple concurrent timers to trigger and fulfill various events, such as publishing to the broker at independent times, subscribing from the broker, and retrieving sensor readings.

The custom class, "Adafruit\_Helper," was created to streamline, extend, and abstract away all the logic and setup needed to connect to, publish to, and subscribe to the MQTT broker.

The custom class "Unit\_Converter" was created to provide a means to mainly convert units of time from hours, minutes, and seconds to milliseconds, and back. However, it is not limited to units of time, and has one variable that can be used to help convert between units of volume.

## Frontend Service: Web Application

The frontend web application was built using the Angular Framework. Angular was chosen over React because it has many built-in libraries and features that reduce the need for third-party libraries that may not be maintained.

Angular supports several control flow syntaxes that help users conditionally render elements in components. It also has a built-in system that manages data states in services and components (Signals) and detects changes within an element to trigger re-rendering (ChangeDetectionRef). Classes and services can be dependency-injected into an element to preserve data states and share signals among other courses via the component's class constructor or the Inject function.

The PrimeNG UI Component Library, the PrimeFlex CSS Utility Library, and Tailwind CSS Utility Framework were all used to save time and avoid building complex components and CSS styles from scratch. Free UI blocks from the PrimeBlock UI Block library were used as placeholder templates for some components' layouts. These layouts were later updated and reconfigured to show the desired content and data, and implemented with logic.

The benefit of using the Prime UI libraries is that they all share a unified theming architecture and a set of design tokens (or defined CSS properties). The properties assigned to these tokens can be overridden to apply style changes across the application when a custom theme is used.

Each main page in the web application is its own component. Some pages, such as the chat, dashboard, and auth page components, had custom child components built for them.

The chat page included a customized chat container and chat form components. The chat container displays all chat messages between the chat completion agent (assistant) and the user. The chat form provides users with a text box and buttons to type, submit, or clear the chat logs. The chat is only stored per user session.

The auth page has three components. Each has templates and handles user login, logout, and sign-up. They render one at a time, depending on whether some conditions are met.

The web application will also have Firebase's auth library installed and will access and interact with Firebase Authentication services to handle user authentication during sign-in, sign-out, and sign-up. Depending on the user's authentication state, they can either access all pages in the application or only the auth page. Routes are protected with Angular's built-in Auth Guard feature.

The application also has four internal services. Two of the services connect to the backend service's endpoints, one for the Adafruit IO feed data and the other for the Chat Completion agent's response. The other two are shared services that can be injected into various components to share common data, such as auth state, light or dark mode state, and more.

## Backend Service: RESTful API w/ Agentic Chat Service and Adafruit Service

The backend service was built with ASP.NET in Visual Studio. The backend service follows the Controller-Service-Repo architecture. There are no functional repo classes, but there are two primary internal controllers and services in this application: the Adafruit Controller and Service, and the Agentic Chat Completion Controller and Service. The controller abstracts the services' logic and exposes the application to the internet. Each controller also has its own custom CORS policy, and the backend application is integrated with Swagger UI. The services house the majority of the business logic that makes up the backend.

The backend application has various support classes that these two services rely on, such as factory, helper, HTTP client, mapper, orchestration, registry, plugin, model, domain, and DTO classes.

The Adafruit Chat Service calls methods on the Adafruit HTTP Client to fetch feed data from Adafruit IO HTTP API endpoints and process it for the frontend application or view.

The Agentic Chat Completion Service uses Microsoft's Semantic Kernel SDK to generate custom AI Agents and place them in a Handoff Orchestration to respond to user prompts. After the handoff, orchestration responds; the Chat Editor agent then edits and streamlines the response before sending it back to the user.

AI Agents are software objects that use AI from an LLM to process information, complete tasks, and respond to prompts autonomously and independently of users, while also having the capability to coordinate with users. Agents can be assigned specific roles and return specific outputs, provided with a system prompt or a role description and output description during instantiation. Agents can also be assigned plugins. Plugins are a collection of Kernel Functions, programmatic algorithms wrapped in a method, that are used to perform specific tasks and provide a specified return.

It should be noted that every agent requires a kernel to be instantiated. A kernel is the core element of the Semantic Kernel ecosystem and serves as a container that houses all necessary services and tools, such as plugins and functions, and LLM/AI services, that agents can utilize to perform an assigned task. However, these features needed to be added to the kernel manually. For this project, a single master kernel was created in the custom, static KernelFactory class. Each agent clones and uses a copy of the master kernel and registers the plugin(s) required for its specific needs.

Orchestration is a process that drives agents to collaborate and coordinate with others toward a shared objective. The orchestration pattern used in the Agentic Chat Service is the Handoff Orchestration. The Handoff Orchestration takes a user prompt, analyzes it, and passes it to an agent who can best provide a response based on the prompt's relevance to the agent's role and purpose.

The agents created for the handoff orchestration are the Chat Moderator agent, the Adafruit Feed agent, the Project Info agent, and the Plant Info agent. The Adafruit feed agents have kernel functions that point to methods in the Adafruit Feed Service. These methods use an HTTP Client to reach out to the Adafruit HTTP API to retrieve feed data from various sensors; these agents will answer questions associated with the sensor feed data and analyze and summarize it. The Chat Moderator agent acts as the relay operator in the Handoff Orchestration. It is the first agent to receive the user prompts, and based on the prompt's context, it will determine which agent can best respond. The Project Info agents will respond and provide details related to the semi-automated plant gardening system.

The Chat completion service also has another agent independent from the orchestration, the Chat Editor agent, which takes the response generated by the Handoff Orchestration, edits and streamlines it, and returns it to the user.

## LLM

Microsoft Azure was used to host OpenAI's LLM Model, o3-mini, which was used as a service to generate chat message responses. The decision to use and host the model on Azure was based on its affordability and accessibility. The o3-mini LLM model is one of the cheaper models that can be hosted in the cloud, and since it's on the cloud, it's accessible to anyone with the proper authentication. A publicly accessible model is essential because I want my deployed applications to have access to it. That said, if other people want to use a local or free LLM model instead, it is possible with Microsoft's Semantic Kernel. I have successfully swapped it with a locally run LLM, Ollama's TinyLlama, for one of my prototypes. Microsoft's Semantic Kernel streamlined the process. All it takes is to register the locally deployed Ollama Chat Completion Service in the kernel, then provide the model ID and local endpoint to the model.

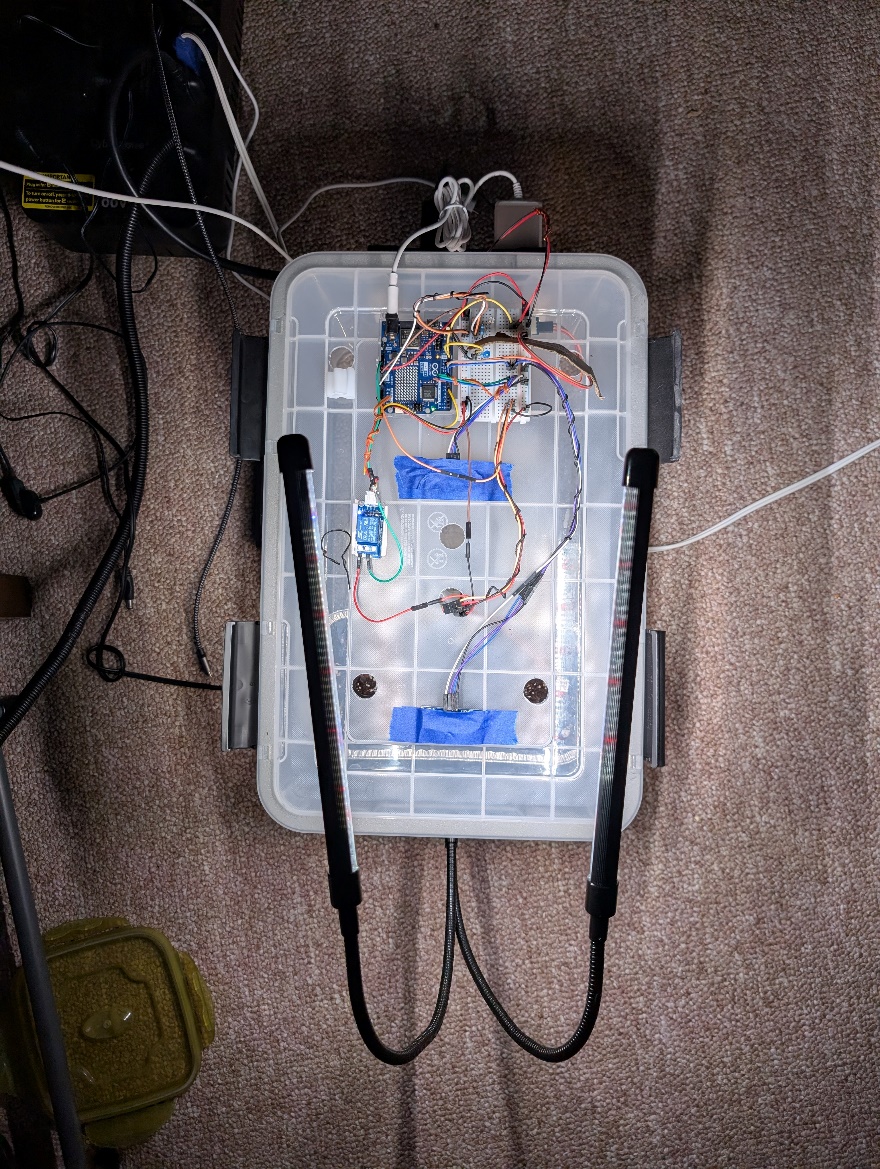
## Deployment

For deployment, Google Cloud Platform was used to host the frontend and backend applications via Google App Engine, Cloud Run, and Artifact Registry. The front application was built on my computer using the Angular CLI build command, then copied into a directory on Google Cloud's Shell along with a .yaml configuration file, and deployed to Google Cloud's App Engine using the Google Shell CLI.

The backend application took more effort. I had to create a Docker image of my application, push it to Google Cloud Platform's Artifact Registry, and then, from Google Cloud's Shell, use the image to add it to a container in Google Cloud Run with a Google Shell CLI command. It may sound easy, but I first had to create a Dockerfile for the application in the source folder and configure it to include the necessary files. I then had to make a Docker image of the application on my computer using a Docker CLI command in PowerShell. Tag the image so that it is compatible with Google Cloud Platform's Artifact Registry. Download the Google Cloud SDK, log in to my GCP account, configure Google to allow Docker access to the account, and then I can finally push the image to Google Cloud's Artifact Registry.

# Results

After months of prototyping and development, the embedded system, including both hardware and software components, was largely completely at the end of September. The IoT enabled gardening system was able to talked to and publish sensor data to Adafruit IO cloud service at that time.



**Photo 1:** Semi-automated gardening system with the top on and growlight activated.   
At the start of the first growing session.



**Photo 2:** Semi-automated gardening system with the top off showing the container for the plant and water supply.

To put the embedded system to the test, I filled a container with soil and cat grass seeds and left it undisturbed while I went on vacation at the end of September for an entire week.

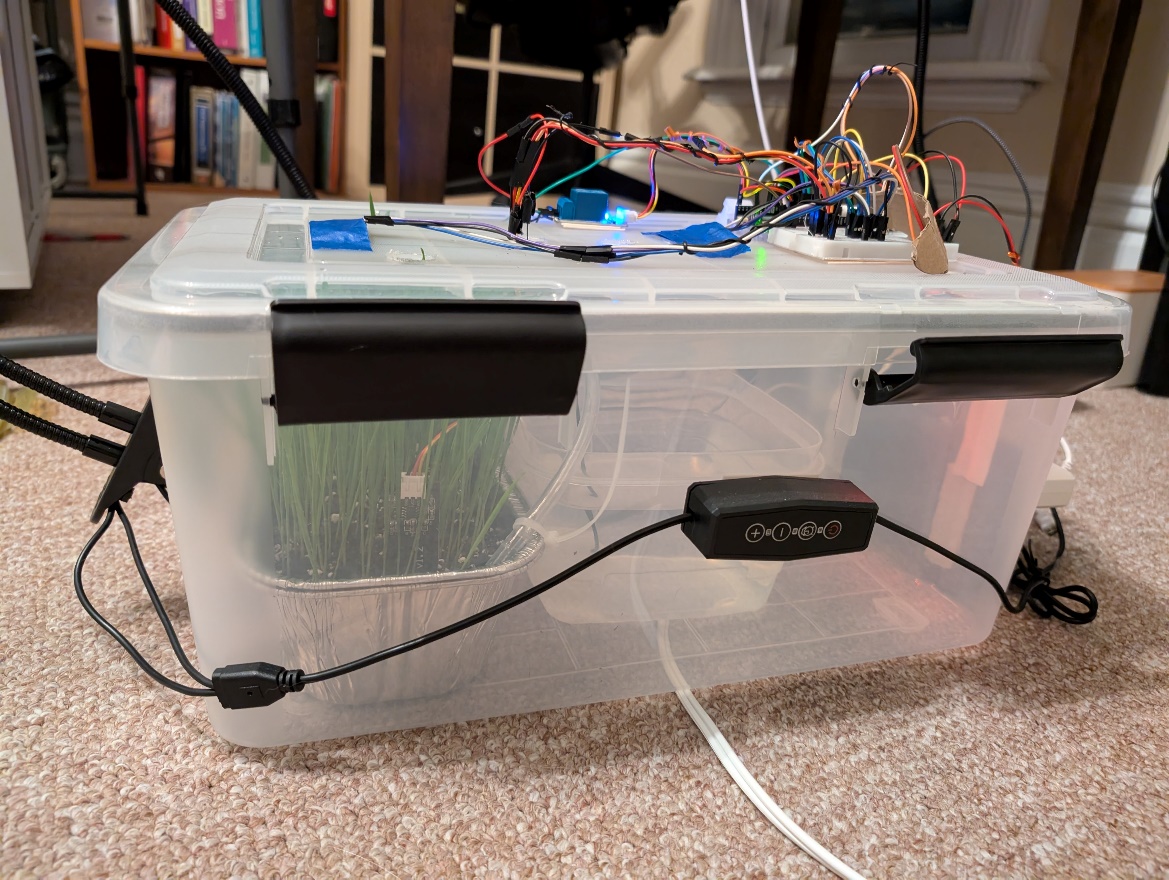


**Photo 3:** Close up of the potted container inside the gardening system. Top soil layered with cat grass.

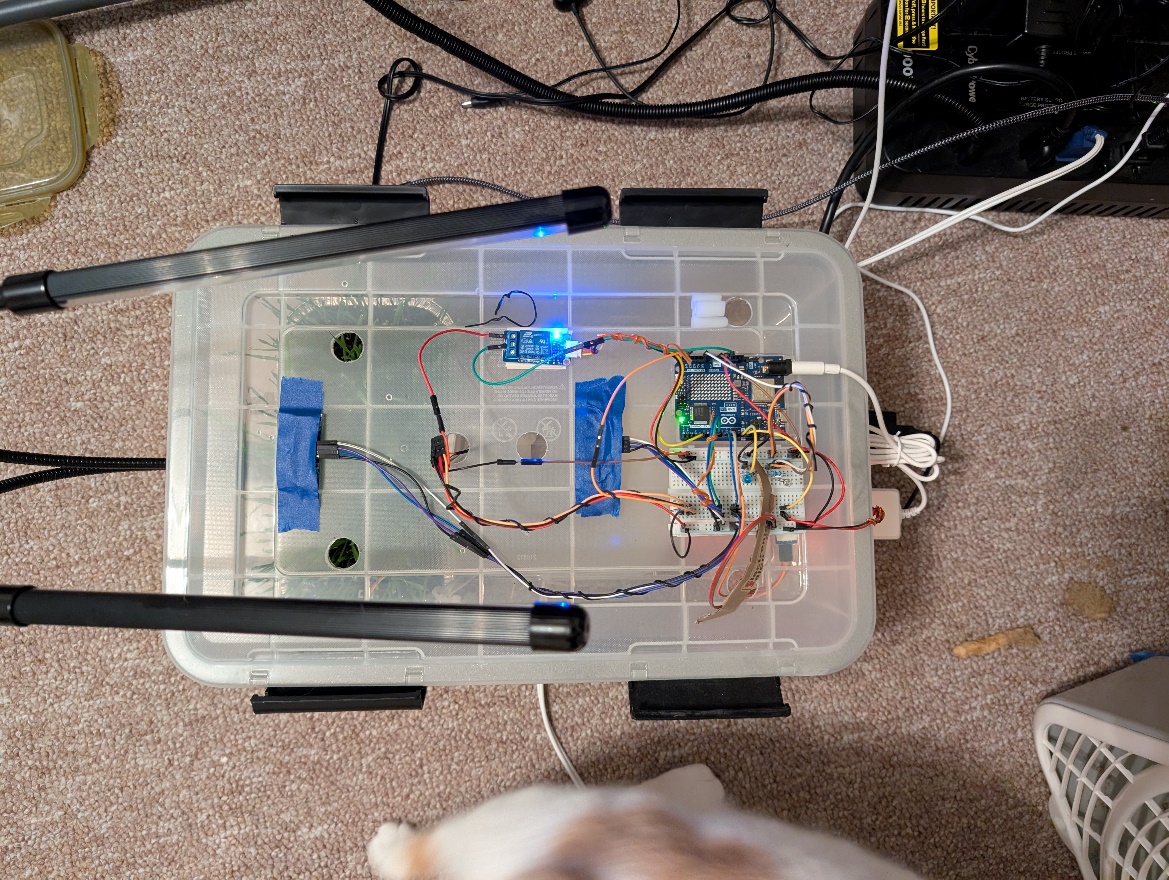


**Photo 4:** A top view of the potted container inside the gardening system. Top soil layered with cat grass.

A week after returned, I saw that the gardening system has successively grown cat grass, autonomously.



**Photo 5:** A side view of the cat grass growing inside the gardening system.

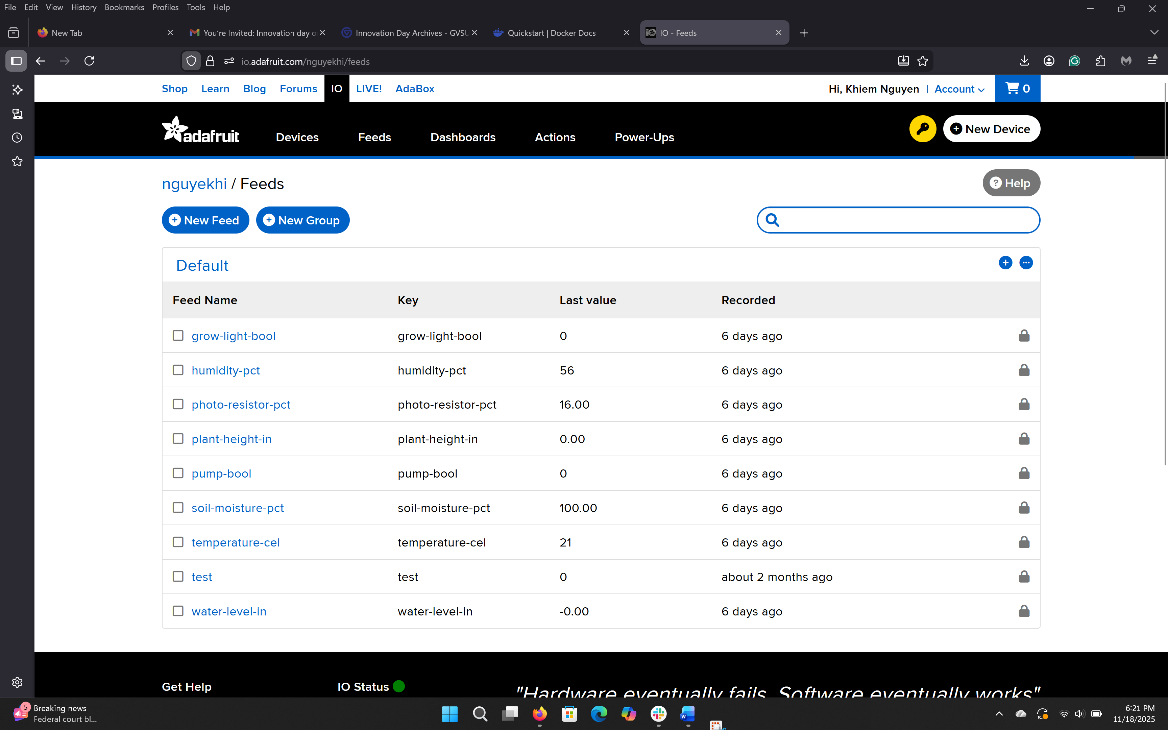


**Photo 6:** A top view of the cat grass growing inside the gardening system.

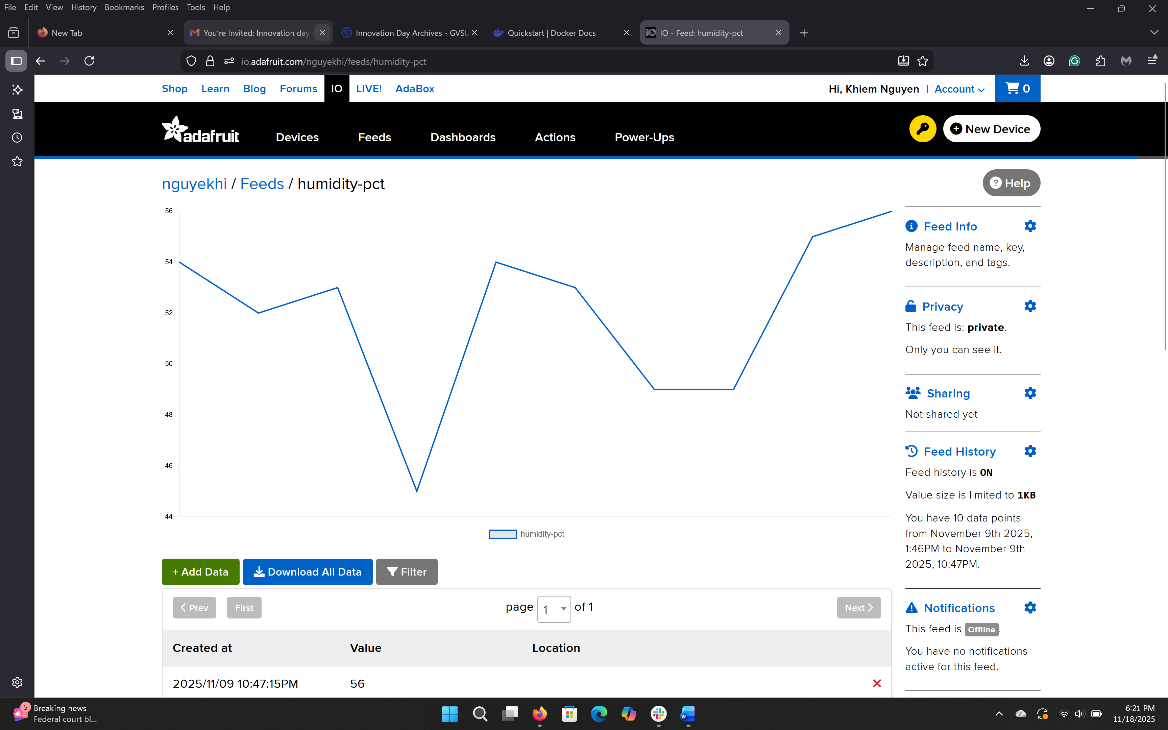


**Photo 7:** My cat, Sammy, enjoying the harvest from the gardening system’s first growing session.

Checking Adafruit IO’s cloud service, I confirmed that service had logged and recorded all of the sensor data published to it into their respective feed database while I was away. Unfortunately, the free version of Adafruit IO only has a 30-day retention policy, so data from the initial growing session has since been deleted. However, my gardening system is still connected to Adafruit IO and still publishes data to it.

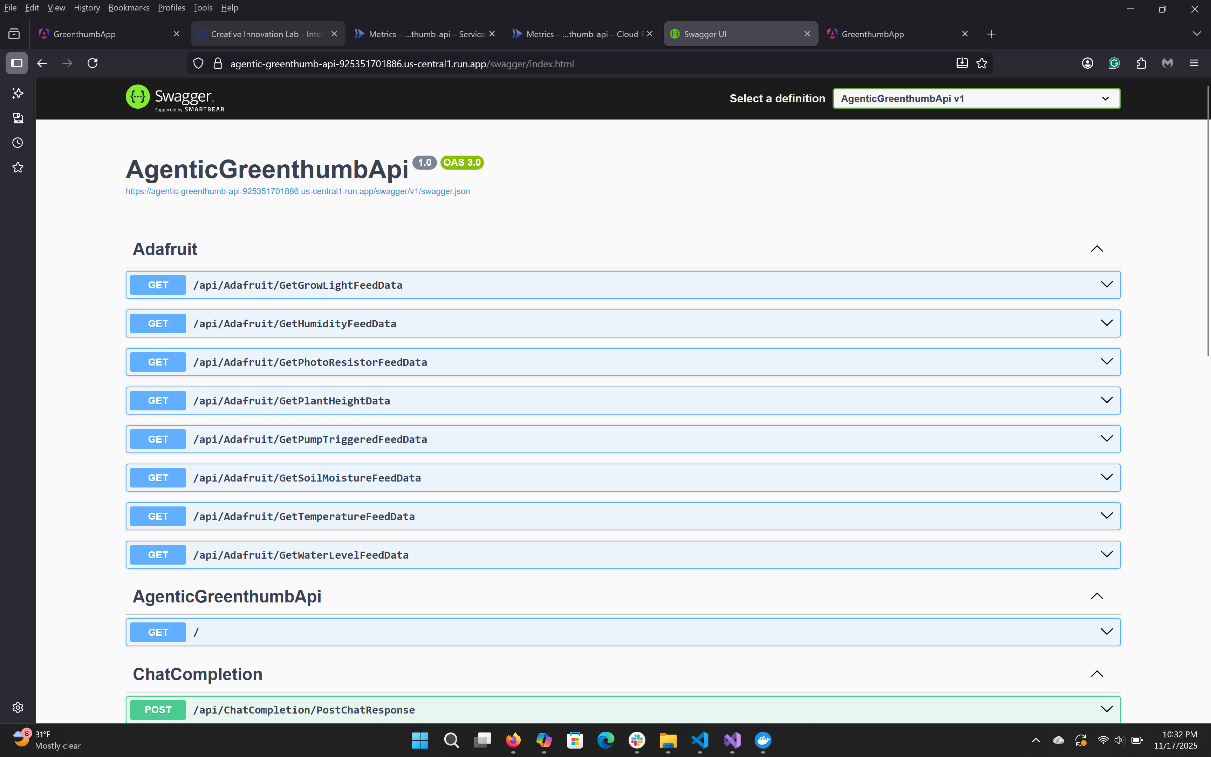


**Photo 8:** Adafruit IO Feed Databases

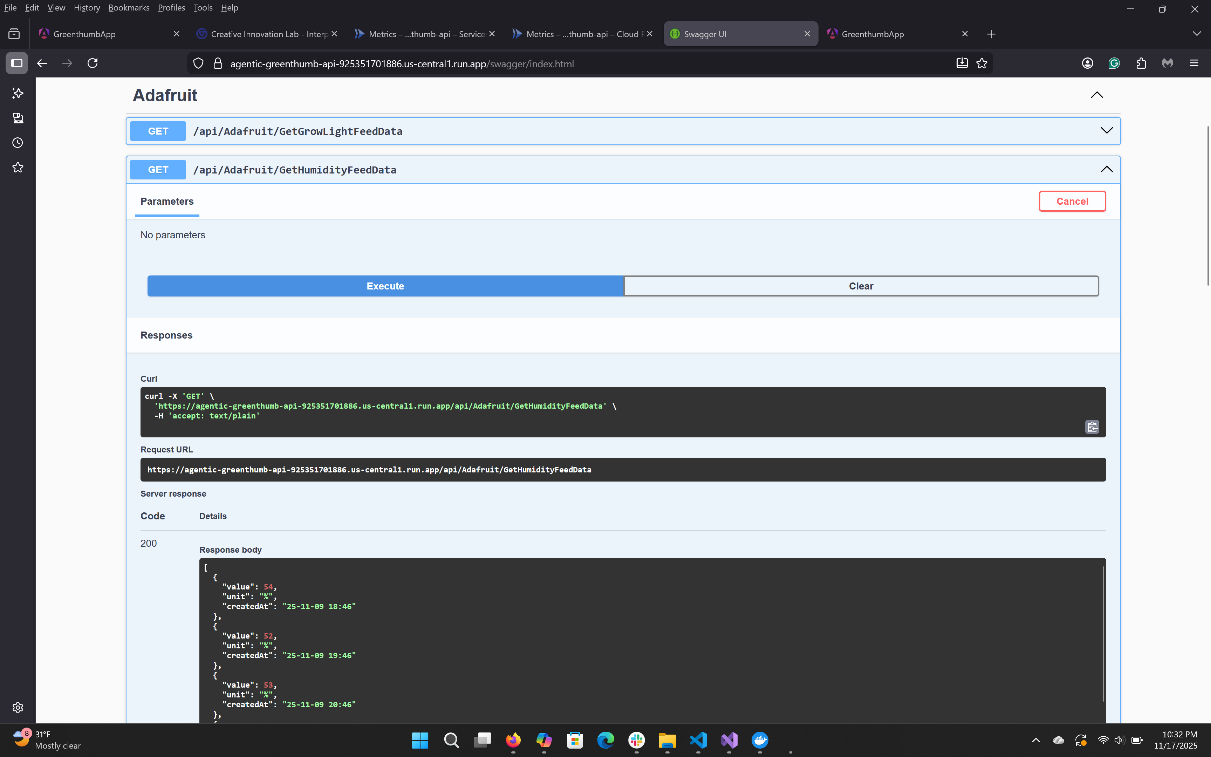


**Photo 9:** Adafruit IO’s Humidity Feed Dashboard

Development of the both backend and frontend applications for the gardening system did not really begin until after the completion of the embedded system, towards the start of October. Completion of both components of the gardening system was completed, and deployed onto Google Cloud Platform by mid-November. However, the result a functional, complete, and aesthetically pleasing application.

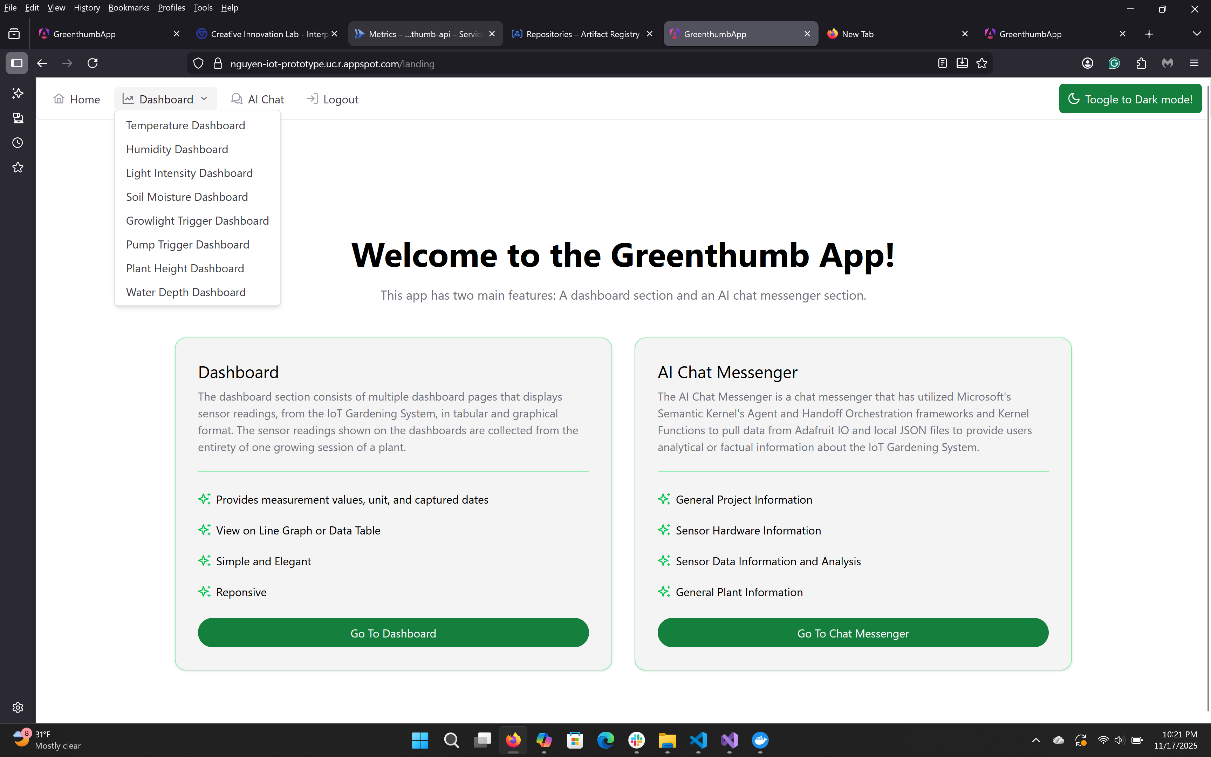


**Photo 10:** Screenshot of the Swagger UI and endpoints of the backend application

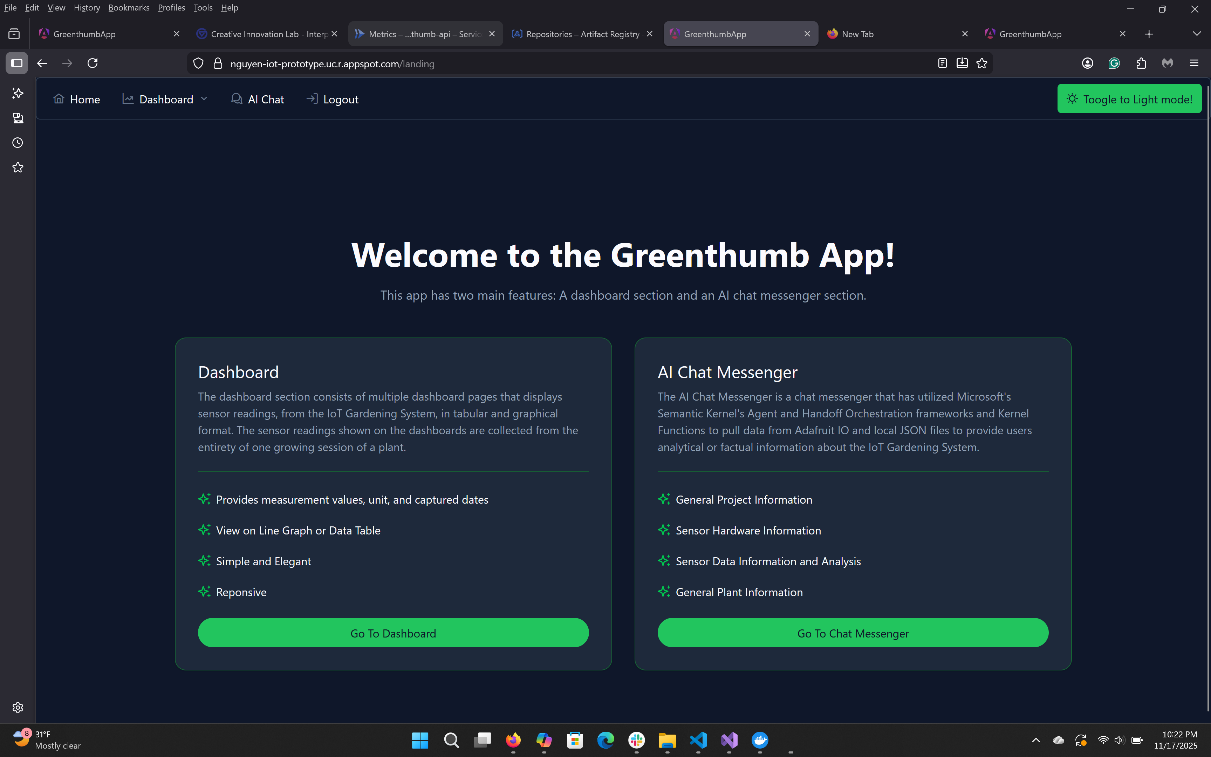


**Photo 11:** Screenshot of the Swagger UI and one of the backend application’s endpoint being tested.

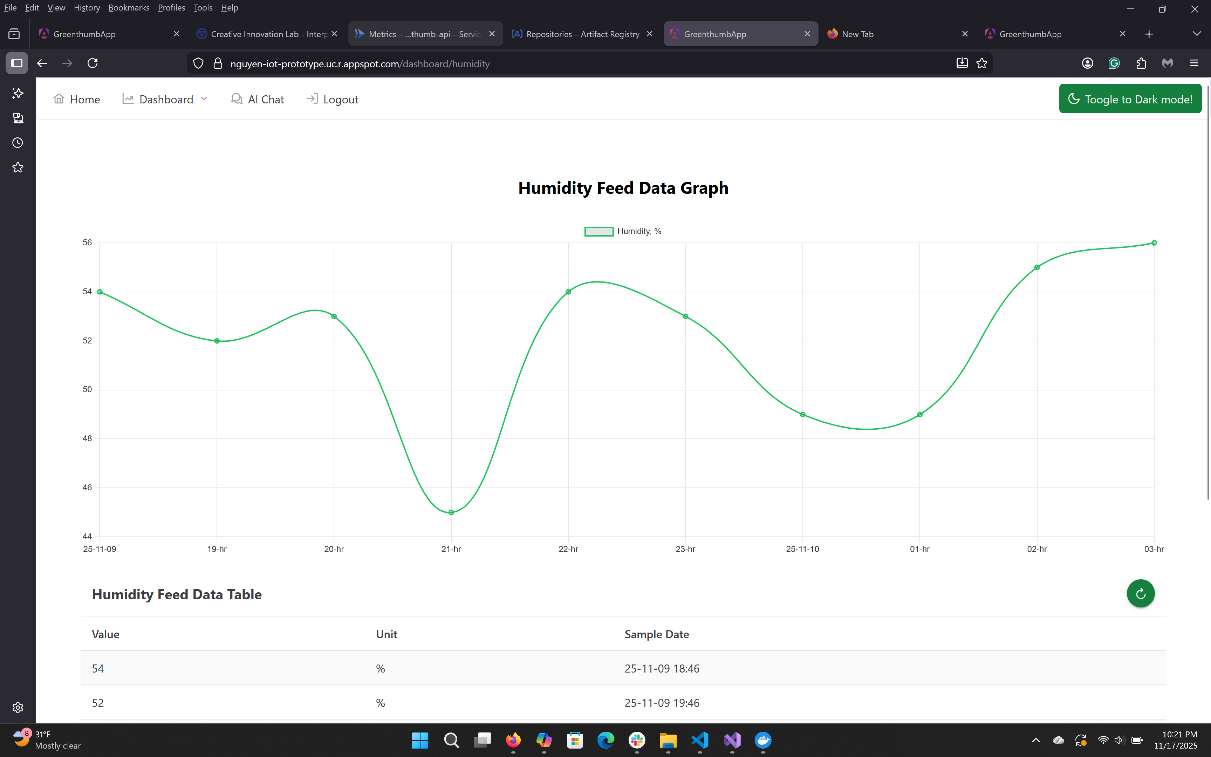
The frontend application for Project Greenthumb is feature complete. The application has a landing page, a set of dashboards, a chat messenger page, and an auth page. The complete set of pages are only available when the user is logged in, else they can only view the auth page. The frontend application also has a light and dark mode that user can toggle in-between and is responsive and can adjust to different screens.



**Photo 12:** The landing page of the frontend application. Light mode.



**Photo 13:** The landing page of the frontend application. Dark mode.



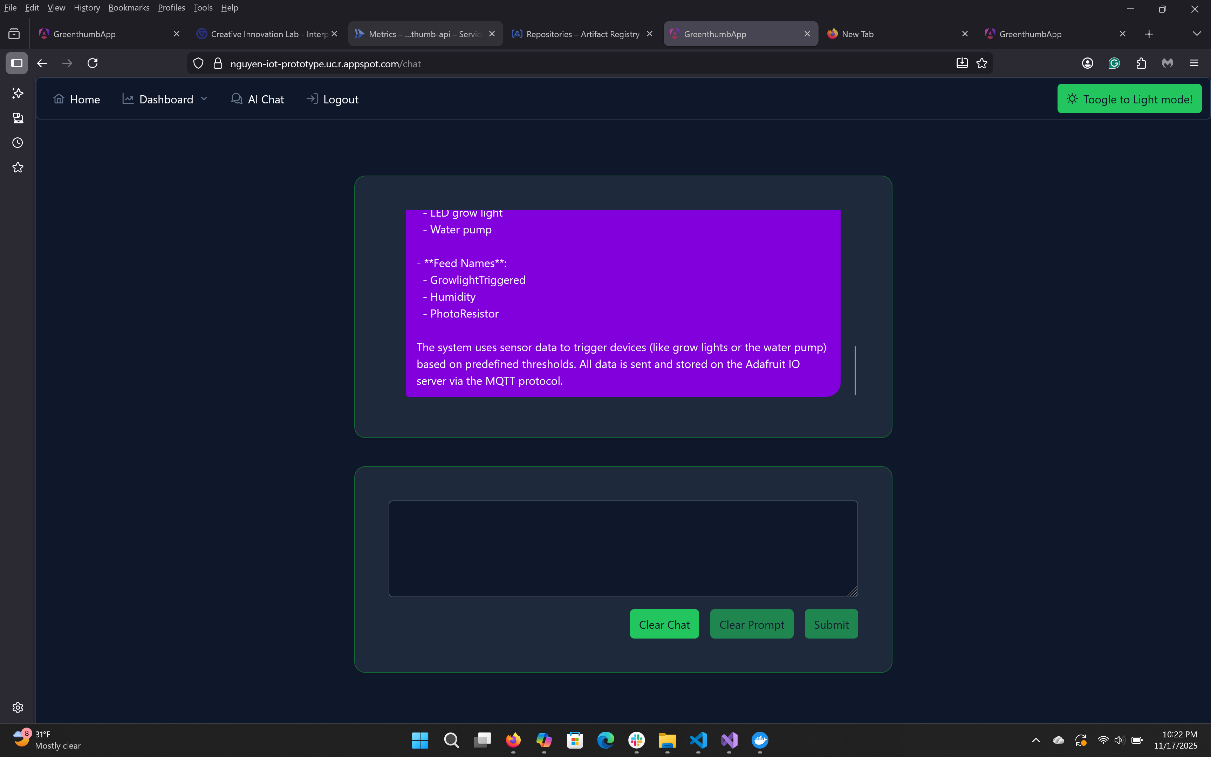
**Photo 14:** The humidity feed dashboard page of the frontend application. Light mode.



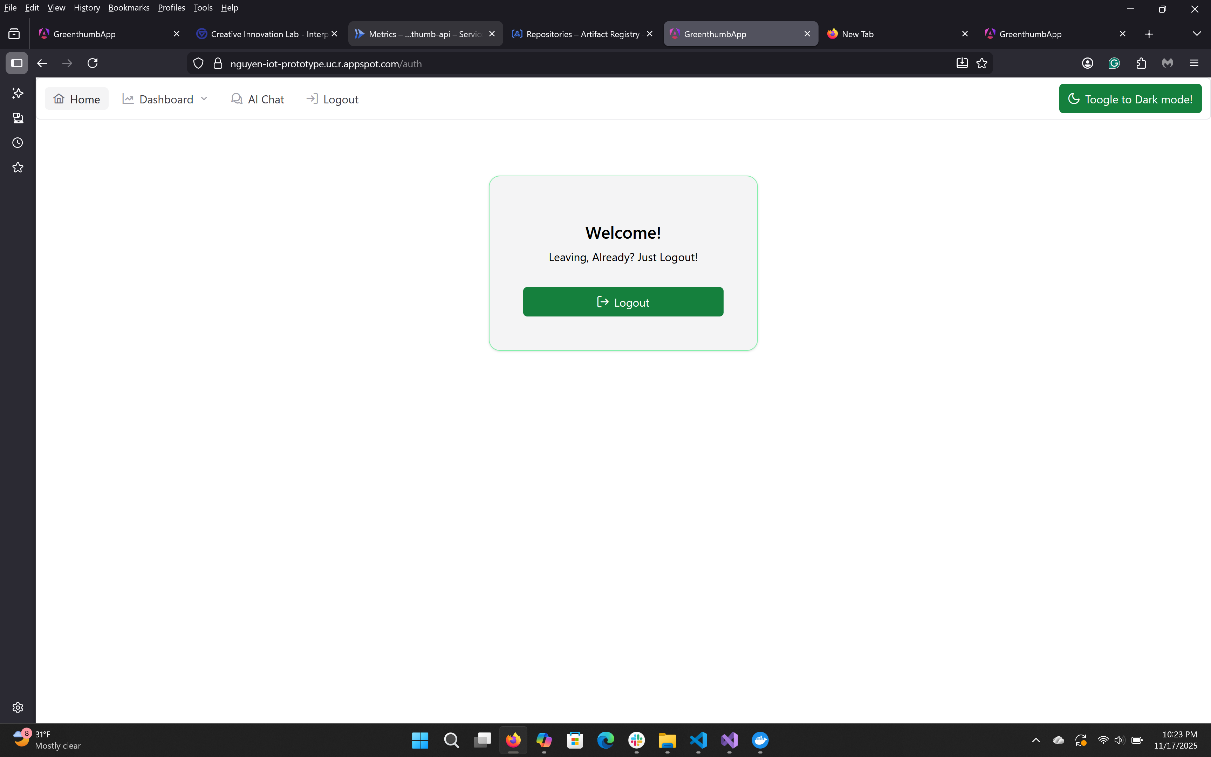
**Photo 15:** The humidity feed dashboard page of the frontend application. Dark mode.



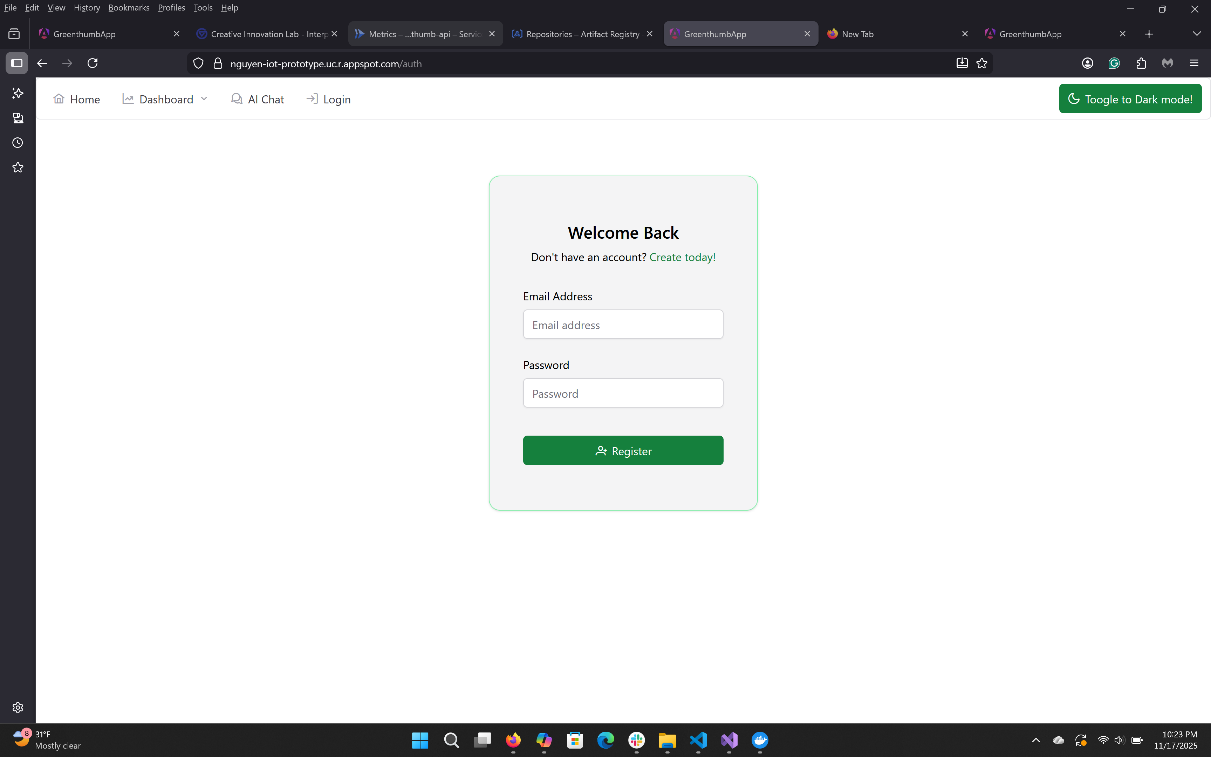
**Photo 16:** The chat messenger page of the frontend application. Light mode.



**Photo 17:** The chat messenger page of the frontend application. Dark mode.



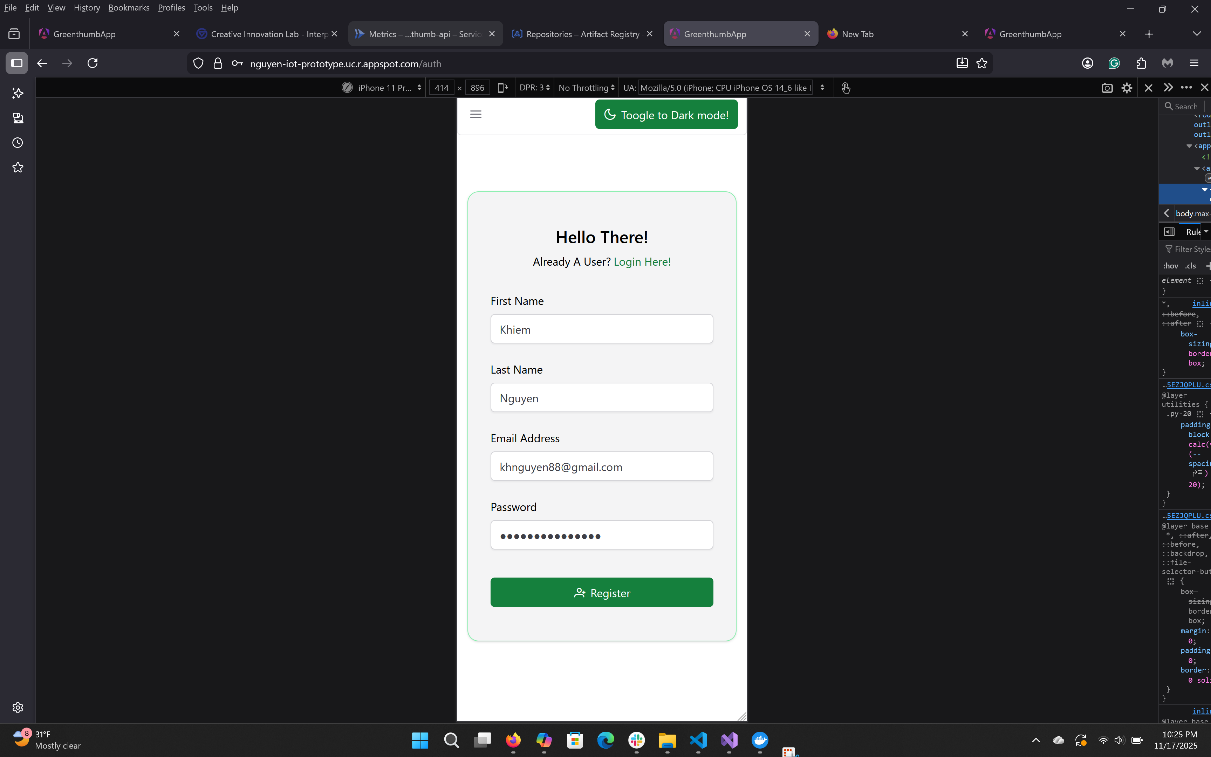
**Photo 18:** The auth page of the frontend application, during the logout view. Light mode.



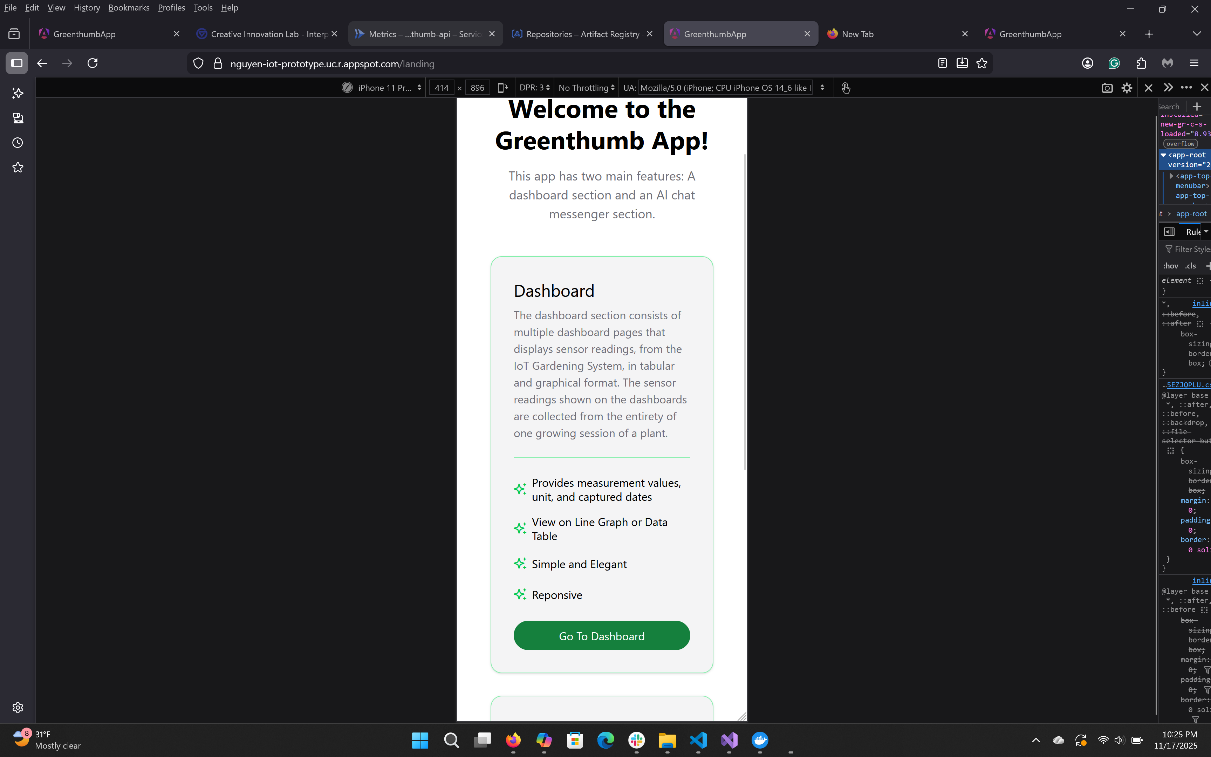
**Photo 19:** The auth page of the frontend application, during the login view. Light mode.



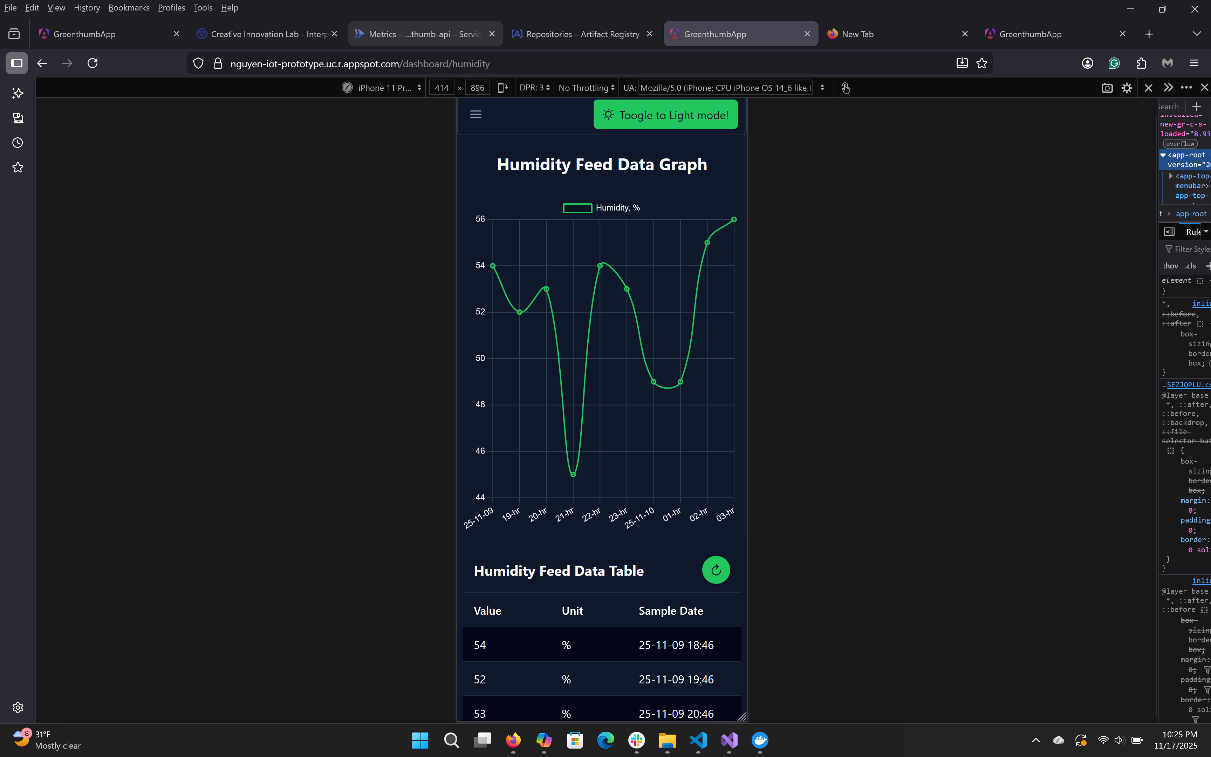
**Photo 20:** The auth page of the frontend application, during the signup view. Light mode.



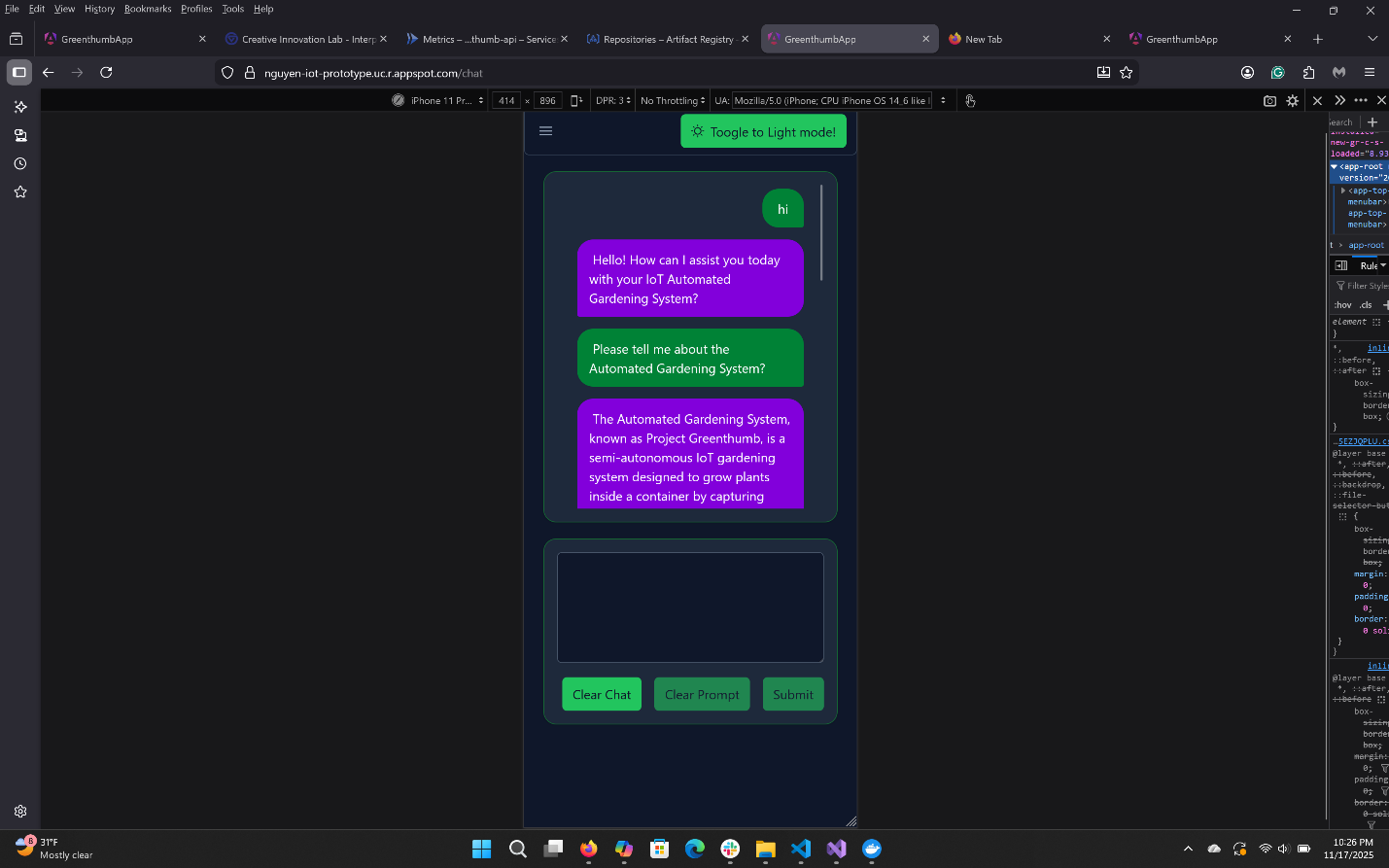
**Photo 21:** The auth page of the frontend application. Simulated on mobile. Light mode.



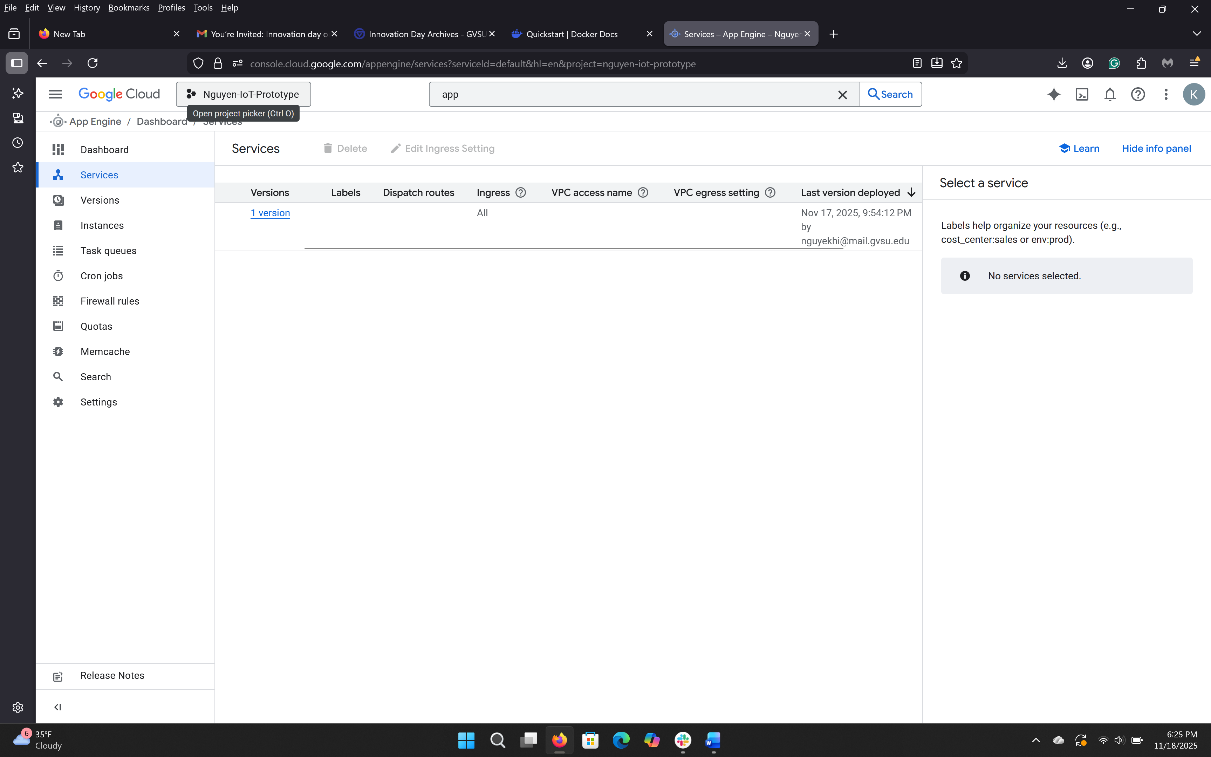
**Photo 22:** The land page of the frontend application. Simulated on mobile. Light mode.



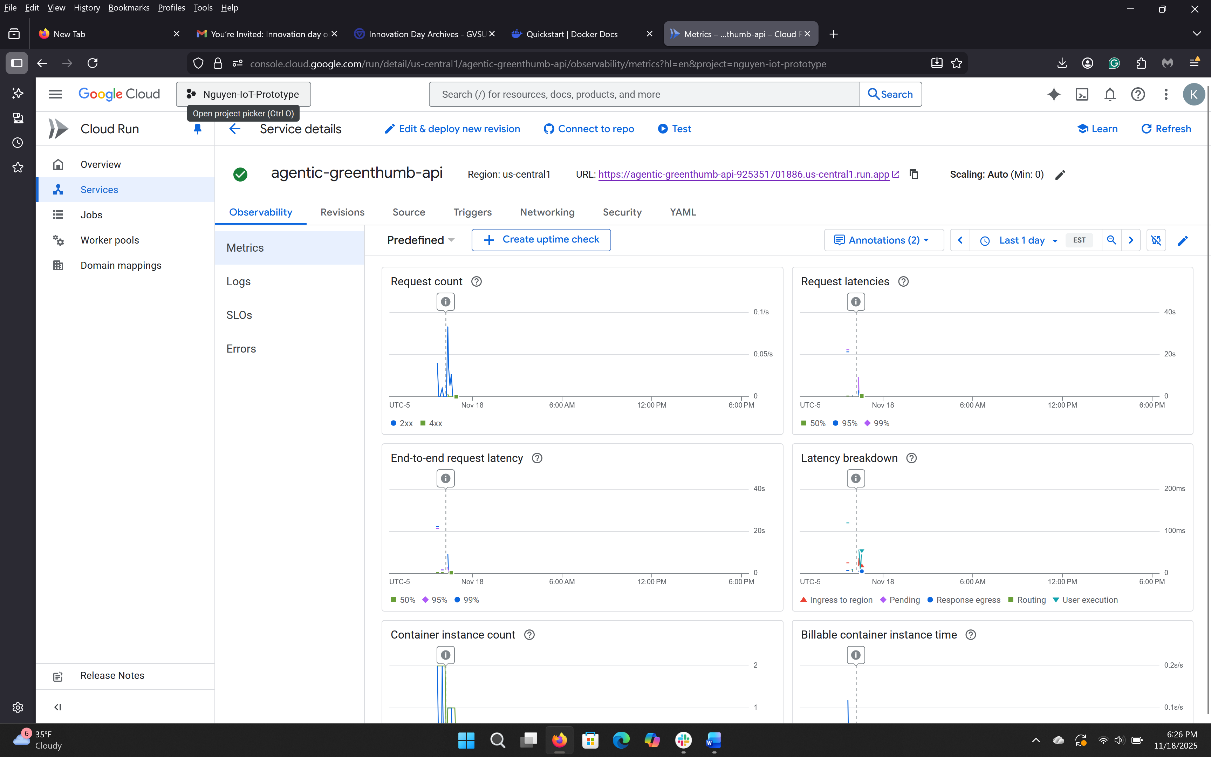
**Photo 23:** The dashboard page of the frontend application. Simulated on mobile. Dark mode.



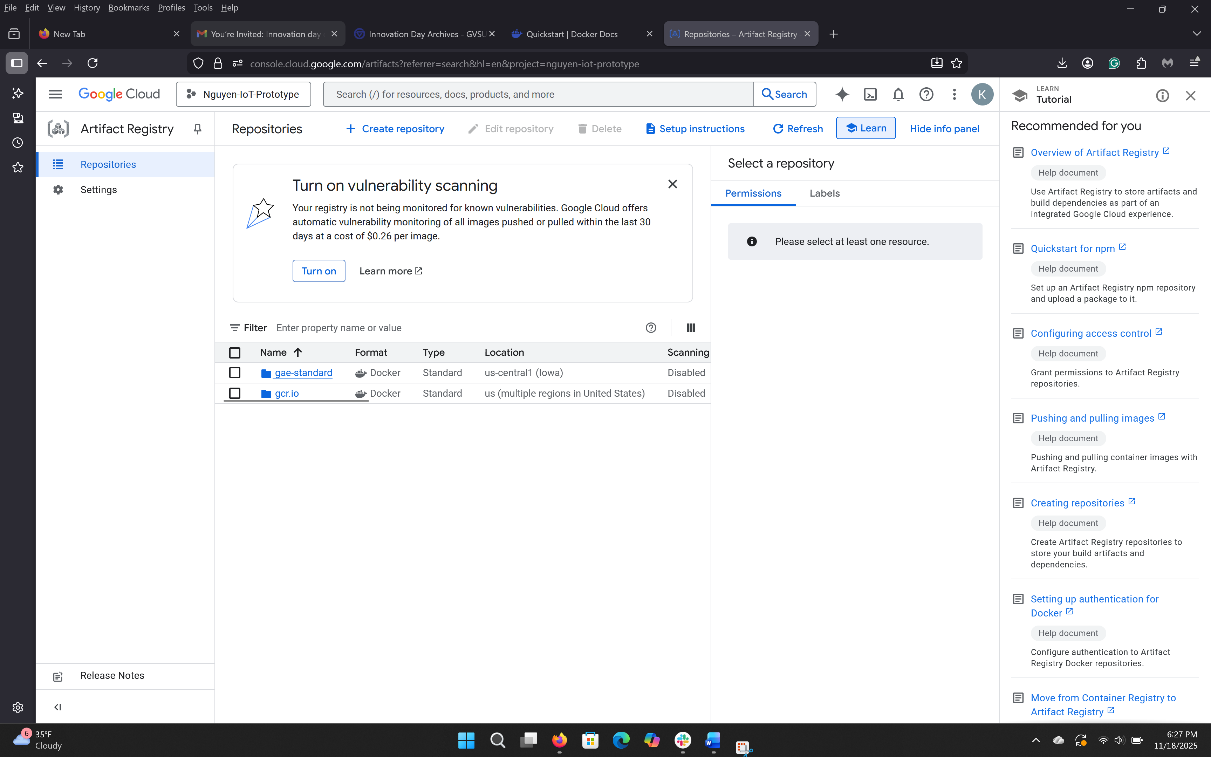
**Photo 24:** The chat messenger page of the frontend application. Simulated on mobile. Dark mode.



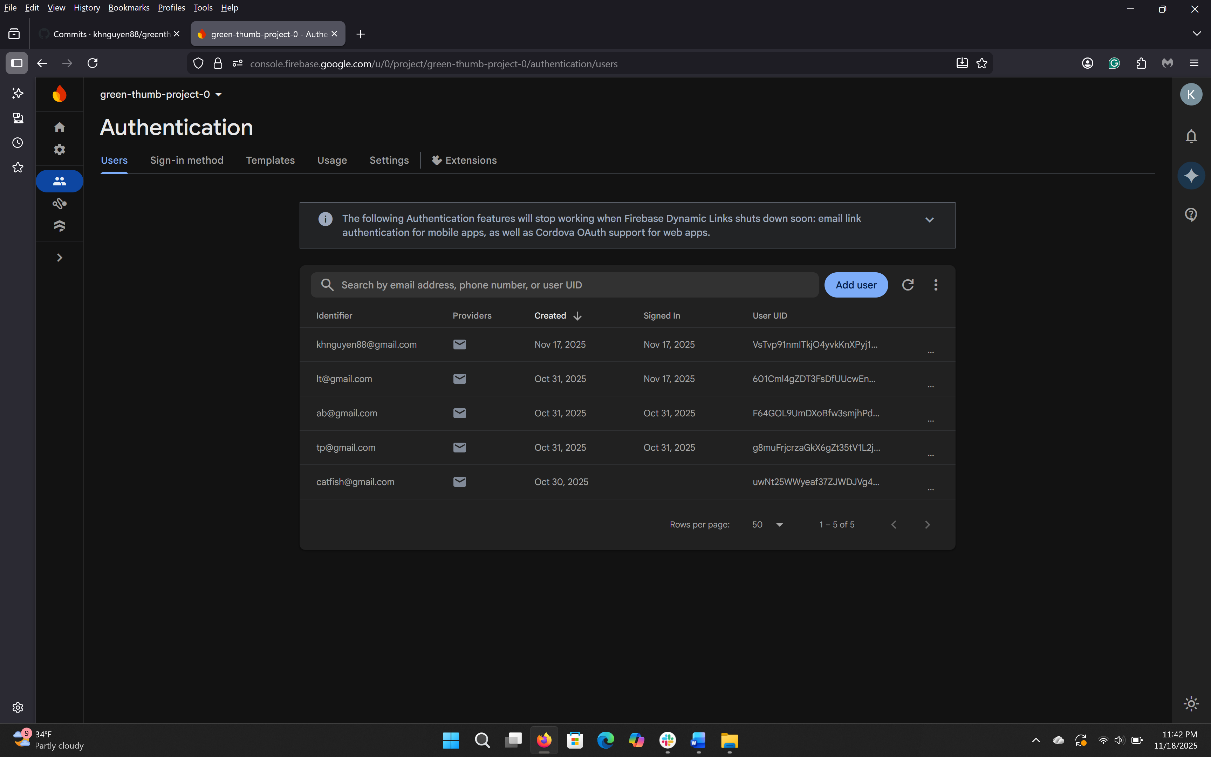
**Photo 25:** Dashboard of GCP’s App Engine showing the frontend application being hosted.



**Photo 26:** Dashboard of GCP’s Cloud Run showing the backend application being hosted.



**Photo 27:** Dashboard of GCP’s Artifact Registry showing an image of the backend application.



**Photo 28:** Dashboard of Google Firebase’s Authentication showing a list of registered user.

# Challenges

## Embedded Hardware

The general challenge of the hardware component of the embedded system was 1) trying to pick up and learn basic circuitry in a short period, 2) debugging and figuring out the root issues with sensors that were not, and 3) figuring out sensors, circuitry components, and devices that were compatible with the Arduino and its needs.

While I had dabbled with the Arduino in the past, it's been 10 years since I last used it, so I had to relearn a lot of the basics.  Fortunately, microcontroller boards like the Arduino and ESP32 are popular devices with plenty of quick-start guides, documentation, forums, schematics, and other resources online, so it was simply a matter of reading up on them. These include the basic circuitry relevant to that microcontroller. It was fun learning and using Ohm's Law or the Voltage Divider equation for some devices (LED light) and sensors (Photoresistor).

The biggest challenge I face is dealing with sensors that either don't give any readings or give inconsistent readings. The point source of failure could be a myriad of things, and ultimately, I had to build my own internal checklist, go through it, and perform a process of elimination to figure out the root cause of the issue. The issue could have been a faulty sensor, incorrect wiring, a bad connection between the pins and connectors at the breadboard, a faulty microcontroller or wires, a faulty breadboard, a faulty resistor, wrong code, and more. In my case, I dealt with all of these issues across multiple sensors, including ultrasonic sonar, photoresistors, the water pump, and the grow light. It was initially hard since I had no idea where to start, but as I ran into more issues, understood some common root causes, and used a checklist, it got easier.

## Embedded Application

For the application side of the embedded system, the challenges I face were 1) trying to understand, learn, and capitalize on the basic features of C++, 2) obtaining C++ support for VS Code; 3) writing code that is works within and outside of Arduino IDE; and 4) writing a clean sketch (main application) file for the Arduino.

 Of all the challenges faced with the embedded application, understanding and learning C++ was one of the hardest. While I am familiar with programming in higher-level languages such as Typescript, JavaScript, Python, and C#, C++ felt different because it was a bit more archaic and lower-level. I had to learn and use pointers and references very often in my application, especially when passing pointers and references to objects as parameters to my helper methods. The concept of header and concrete/source files was something I never had to concern myself with in other higher-level languages, and the way override or populate methods inherited from a header or parent classes required a different setup; for example, the scope resolution operators (::) are needed to specify which method the class came from even if your referenced the parent class. There were a few more nuisances, such as memory allocation, especially when initializing an array, “string” and “char” data types, and casting values that I had to get used to as well.

Another challenge I faced was finding an environment conducive to writing and managing multiple C++ files simultaneously. Visual Studio Code does not support C++ out of the box. It requires additional extensions and libraries, and even then, I was unable to figure out how to get Visual Studio Code to recognize the Arduino Library found natively recognized by Arduino’s IDE, so ultimately, my workflow became writing blind in Visual Studio Code, and then compiling and debugging the application with Arduino IDE. It was a hassle, but it was better than managing multiple files in Arduino’s IDE.

The last challenge was learning to parse and separate code from my central sketch and source (.ino) file, as I gained a better understanding of C++ and my workflow. Luckily, prototyping other sample code helped me understand how I wanted to structure the embedded application and what code I needed to parse into a separate class to keep the main sketch file somewhat clean.

## Frontend Service

There were two challenges I had to overcome in developing the frontend application.

The first and biggest challenge was understanding and implementing the PrimeNG theming system, which governs the global CSS styles of the components used in the application. The library's new theming method utilizes CSS variables, each carrying a predefined set of style properties. Styles are linked to design tokens, basically properties in a theming TypeScript object that a user can create to override one of Angular's available base themes.

The issue is that Angular's documentation is inadequate and does not provide the complete list of available tokens and their associated CSS variables that users can override. The features are hidden behind a paid Theme Designer's Figma Extension, so I spent more time figuring out this feature than programming any other custom components in the application. This process was vastly different from previous versions of PrimeNg, which allowed for custom styles in the global stylesheet or individual component stylesheets to override PrimeNg theming.

The other challenge was learning and understanding Angular's new state management feature, signals, how it works with Angular's change detection mechanism, and how to use it in the application. Change Detection is a system that ensures the application UI View reflects the component's internal state, as defined by its property values. It turns out to be very easy and is very similar to React's "useState" and "useContext". Before this project, I used behavior for state management, which could be retired with the addition of signals.

## Backend Services

During the development of the backend services, I ran into three major hurdles.

The first major hurdle was understanding the tools and features available in Microsoft's Semantic Kernel and Kernel Memory SDKs, learning how to use them in small prototype applications, and selecting and defining a viable architecture for them in the capstone's backend application.

While there are tons of resources and sample code for both SDKs, most cover only a small portion of the SDK's features, and even then, they become irrelevant as Microsoft updates and makes changes to these pre-release packages at a rapid pace. The best resources for learning about these two SDKs were Microsoft's online documentation, GitHub repository, and learning modules. Even then, I still had to slowly dissect some of the features on my own to fully appreciate and understand them.

Through the learning module exercises and prototyping, I was able to play and experiment with a good chunk of the features provided by Semantic Kernel and Kernel Memory. I selected the features relevant to this capstone's backend service and proposed a rudimentary approach to incorporate them into the architecture.

The second challenge I ran into while developing the backend application was learning to implement CORS for an ASP.NET web application. I've done it with Flask and Node.js backend applications before, but never with an ASP.NET backend application. Luckily, there were many resources available online from Microsoft and Stack Overflow, and the process was relatively simple.

The last challenge I ran into while developing the backend application was finding a viable way to deploy it to Google Cloud Run. For a React and Angular application, it is as simple as uploading the build folder along with a YAML deployment file onto Google Cloud's shell environment and running a few CLI commands to deploy it on App Engine. This process did not work for the ASP.NET application.

Remembering that Cloud Run effectively builds an application image and runs it in a container, I looked up, found, and used the process for building and publishing the backend application as a Docker image. Then, I pushed that image to Google's Artifact Registry Service, pulled it from the registry, and deployed it to Google Cloud Run using the Google Cloud Shell CLI. This process required me to build a Dockerfile, specify the application's required files and directories in the image, install the Google Cloud SDK, and grant Docker access to my GCP project. Luckily, resources and tutorials for those were readily available online.

## Cloud Services

I encountered no significant challenges with Google Cloud Platform, Microsoft Azure, Google Firebase, or Adafruit IO Service. Having taken the Mobile Application Development and Cloud Application Development courses, I was able to comfortably navigate and utilize the services in Google Cloud Platform and Google Firebase. I was able to learn and navigate Microsoft Azure thanks to my familiarity with Google Cloud Platform.

There were only minor challenges navigating around Adafruit IO Service, and Google Firebase was trying to figure out how to use their respective SDKs to connect and communicate with the cloud services. However, both have some good documentation online, and Google Firebase is particularly popular; there are no third-party resources or tutorials for it.

## Others

Another challenge was allocating time to work on the project outside of work, home, and parental responsibilities. Fortunately, I was able to find time in the evenings after my toddler had gone to sleep, but there were instances when plans were thwarted by sleep regression or an irregular sleep schedule due to illness. Luckily, I had given myself a reasonable scope and sufficient slack time to complete the project and deliverables a few weeks before the deadlines.

# Proposed Future Improvement

While the application is usable, many improvements could enhance the user experience, improve configurability, and extend the usability of the IoT gardening system components to other projects as well.

## Embedded Hardware

The one improvement proposed for the embedded system's hardware component is to replace the ultrasonic sonar sensors used to measure plant height and water supply level with other sensors. The ultrasonic sonar sensors were determined to be too inaccurate and unreliable for their readings to be used for analysis. Proposed alternatives include a water metering sensor, a LiDAR sensor, and a photo camera. That said, support for the ultrasonic sonar sensors will remain, as they are a cheap option for other users.

## Embedded Application

For improvements on the application side of the embedded system, the focus is mainly on refining, cleaning up, and streamlining the existing code, and adding new sensor capabilities.

These improvements include separating existing code into new custom helper classes and methods for the sensors currently used by the embedded system.

The second improvement proposed is to refine the concurrent timers and frequency at which data is published to Adafruit IO, especially when a device such as a pump or grow light activates and deactivates.

And while the MQTT subscription feature was successfully implemented in a prototype, it was not used in the final embedded application. Another improvement to the embedded application is to enable the subscribe feature and allow users to manually adjust parameters, such as sensor threshold trigger values, grow light activation duration, water pump activation duration, general growing information and description, and flags to turn on or off individual sensor readings or publications.

The last improvement I want to make is to add new sampling, handling, and processing logic for new sensors, such as a water metering device, LiDAR sensor, and a photo camera, so users can import and use them in the code whenever needed. I wanted to support these sensors because I found the Ultrasonic Sonar sensor to be wildly inaccurate and unreliable for distance measurement.

## Frontend Service

A future improvement to the frontend application is the addition of two settings pages: an Embedded System Settings page and a User Settings page.

The Embedded System Settings page will provide users with the options to 1) toggle on and off sensors used in the embedded system, 2) configure the grow light and water pump settings, such as sensor threshold values and activation duration, and 3) provide general gardening information.

The Users Settings page will allow users to customize the UI of their application. For example, users can choose 1) their preferred lighting mode, 2) the primary theme color, and 3) the ability to turn the chat messenger on or off.

## Backend Service

There are several improvements proposed for the backend. One significant improvement proposed for the backend is to rework the agent and agentic orchestration setup and instantiation process in the Chat Completion Service to make it extensible and configurable. At the moment, all of the agent's descriptions and instructions are hard-coded in their designated registry class and instantiated in the classes that use them. Similarly, my Chat Handoff Orchestration is a wrapper around Semantic Kernel's Handoff Orchestration class and explicitly specifies the type of agent passed in and used to set up and instantiate the Handoff Orchestration.

Microsoft's Semantic Kernel SDK allows agents to be created from YAML template files, so I would use that in the future to set up multiple YAML files and instantiate all of my agents through that method in a custom factory class. I could even use the YAML file to adjust each agent's prompt execution settings, which are used to specify an AI model I want to run (local or remote) and fine-tune a model response. This would abstract away the need to create or remove a custom registry class anytime someone wants to add or remove an agent.

Similarly, Semantic Kernel's Handoff Orchestration allows for params of Agent objects to be passed into its parameter, meaning the parameter could accept either individual agents or an array of agents, so I could generate an array of agents and pass a collection of agents into the wrapper and instantiate the Handoff Orchestration in a more extensible way. To make my orchestration wrapper extensible, I may also include an interface that other child orchestration wrappers can inherit, including common methods shared amongst them.

Another consideration and design change is to add a Data(base) service to the backend in a way that makes it extensible and configurable based on a user's application settings. The idea is to create a generic data service interface that child data service classes can inherit. At the start of the application, it can select and use the preferred database service based on the user's settings. This will be a considerable undertaking; some services, such as Local SQL Server, In-Memory, Firebase's Firestore, and local storage, have different architectures, and each handles CRUD actions differently.

I also want to create an endpoint that can handle image processing for the new sensor or device I plan to incorporate into the embedded system in the future. This way, the backend service can run locally and the features can be made available to others for free, while the service can be deployed to the cloud as a serverless function or a Cloud Run application.

Lastly, I would update the application's settings (configuration) file to make the agentic feature of my model optional. If people do not want to use the AI chat features, we can turn them off. I also want to extend the application settings file and my custom Kernel Factory class to give users the option to include either a local LLM service (free), a remote LLM model service (paid), or both models in the master kernel based on their preference.

# Conclusion

Overall, Project Greenthumb, the semi-automated IoT gardening system built for the Capstone Project, was successful.

It met the capstone requirements by implementing responsive and good UI/UX design in my frontend application and RESTful Principles for my backend services and application, learned in both the Mobile Application Development and Web Architecture courses, and the utilization of various cloud services for authentication, data storage, and web hosting, many of which were taught in Cloud Application Development.

I also met all of the business requirements from the stakeholders and my personal requirements. While improvements could be made to the capstone project, the current iteration meets all business requirements, is functional, and is polished.

The semi-automated IoT gardening system can independently grow plants with a short growing period (2 weeks), such as cat grass, from seed to maturity, with no human intervention. Refilling the water supply and amending the soil would be the only intervention required if users were to grow plants with a longer growing period.

In addition, in the short time I worked on this capstone project, I learned C++ and basic circuitry for programming, and I created a functional gardening system. I learned about the MQTT protocol and how to use the Adafruit IO Cloud Service and its MQTT API to publish and subscribe to sensor data in their feed database. And for AI Services, I learned and used Microsoft's Semantic Kernel Software Development Kit (SDK) to develop a RESTful API that leverages Multiple Specialized Agents and Agentic Orchestration to provide guardrails and responses for specific topics within a prompt during conversational interactions with an LLM.

The gardening system, which is IoT-enabled, can send sensor feed data to Adafruit IO database services, which are then retrieved and processed by the backend services whenever the frontend application requests the data. The frontend application is implemented with the user in mind; it is responsive, works across various screen sizes, and includes the requested features, such as an AI-powered chat messenger, a set of dashboard pages to display data from every feed type, and an auth page for user sign-up, login, and logout.

A copy of the application is deployed to the cloud and can be accessed anywhere with an internet connection, but the application can also run locally.

Despite the challenges I encountered during the application's implementation phase, I managed to overcome them and complete the project. I learned ways to improve it from a user, maintainability, and reusability standpoint to ensure it has a long and active lifecycle, and I intend to continue improving and providing support for the application long after the Capstone course is completed.