GVSU CAPSTONE: PROJECT GREENTHUMB

A Semi-Automated IoT Gardening System

Khiem H. Nguyen

11/6/2025

Table of Contents

Introduction

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

The academic goal of this capstone project was for students to create a functional, high-end application that integrates multiple topics and concepts learned throughout the graduate program.

The personal goals of this project included creating an application or system that 1) incorporated a personal hobby or interest, 2) had a practical and functional use and purpose, and 3) provided an opportunity to explore and learn new programming skills and concepts beyond the scope of the current graduate program's curriculum and professional responsibilities.

Gardening was selected for this project because it had been a lifelong hobby and aligned with a personal interest in science and biology. The objective was to build an open-sourced, semi-automated IoT gardening system that could help grow plants with minimal human involvement. The gardening system was designed to be low-cost, customizable, repairable, and non-proprietary. Although the gardening was not expected to be production-ready by the end of the capstone project, it was still functional and usable, served as a sound proof of concept, and laid the foundation for future improvements.

To achieve the objective of building a semi-automated gardening system, new concepts in embedded and IoT design, basic circuitry, the MQTT protocol and services, and agentic design and orchestration were explored and learned. None of those subjects were covered in the graduate program's curriculum, nor used at work.

The gardening system was composed of three components: an embedded hardware component and two software components. The hardware consisted of a small plastic container to house the plant and the water supply, a microcontroller, multiple sensors, a pump, and an LED grow light. One of the software components was a responsive frontend web application, built with Angular and the PrimeNg UI library, which served to retrieve and display sensor data from the system and provide a chat interface for users to communicate and handle conversations with an LLM that could answer questions relevant to the project, the sensor data, or general plant information. The other software component was a backend RESTful API built with ASP.NET that connected to two different cloud platforms to retrieve data, process it, and send it to the frontend application.

In total, the gardening system utilized four cloud platforms for its services. The two cloud platforms the backend service communicated with were 1) Microsoft Azure, which hosted and provided access to OpenAI's affordable LLM model, o3-mini, and 2) Adafruit IO, which supplied the sensor feed stored in its database. The third cloud platform was Google Firebase, which enabled user authentication and authorization for the frontend application. The last cloud platform used was Google Cloud Platform, which hosted both the frontend and backend applications, making them publicly accessible.

The stakeholders of the capstone project included GVSU professor and advisor, Professor Fredericks, who reviewed and approved the scope and proposed features of the project as well as the design

choices; Jay Von Rosen, boss and owner of JvR Enterprises, who approved and encouraged research and design on agentic solutions, Semantic Kernel SDK, Kernel Memory, and Retrieval Augmented Generation (RAG) during company time; and myself, the project contributor, for the reasons stated above.

The project followed a hybrid software developmental process that combined the Exploratory and Iterative Developmental models. The initial part of the process relied heavily on prototyping and the development of throwaway applications to build the knowledge on concepts and topics that were never covered in the graduate program's academic curriculum or used at work, such as embedded design and circuitry, IoT, and agentic design. The exploratory process helped refine the project's scope and feasibility based on the knowledge gained and contributed to the initial plans of the capstone 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: the capstone advisor, Professor Fredericks; the owner of JvR Enterprises, Jay von Rosen; and myself. Each stakeholder proposed features for the gardening system and its design requirements.

Stakeholder: Professor Fredericks

Capstone advisor Professor Fredericks reviewed and approved the scope and proposed features of the capstone project. During discussions held over the summer, a consensus was reached that the project needed to meet specific capstone requirements and be complete, functional, and polished.

Professor Fredericks established a set of requirements for the project, which included the incorporation of an embedded component designed to collect plant data via sensors and act on it. The gardening system was required to be IoT-enabled, facilitating the exchange of data and information between the embedded system and an external system via the internet.

The gardening system was also required to feature a user interface (UI) that displayed the collected sensor data; for this purpose, a frontend web application with several dashboard pages was proposed. This dashboard component of the web application was identified as a key requirement feature.

Additionally, Professor Fredericks recommended that the web application include a mechanism for user authentication and authorization. To achieve this, the web application was to consist of pages for user login, logout, and signup, accompanied by a backend service to manage and maintain user identities and accounts.

Stakeholder: Jay von Rosen

Several features for the gardening system were discussed and agreed upon. It was proposed that the frontend web application include a chat messenger component that can communicate with a large language model (LLM). To ensure the LLM delivered relevant responses to prompts about the gardening system or plant information, a RESTful agentic backend and guardrails were deemed necessary.

Discussions with Jay von Rosen reached a consensus on several features for the gardening system. The recommendation for the frontend web application included a chat messenger page that could engage with the LLM. This approach aimed to ensure the LLM provided accurate responses related to the gardening system or plant information through the proposed RESTful agentic backend.

The plant information supplied by the backend service could be formatted for manual entry as threshold values for the embedded system.

Stakeholder: Khiem Nguyen (Me)

In addition to the personal objectives for the capstone project, other design requirements for the semi-automated gardening system were identified that had not been proposed or requested by the other stakeholders.

Embedded System and Growing Setup

For plants to grow correctly, they require sufficient water, an ideal soil moisture level, adequate full-spectrum light, a suitable growing medium and nutrients, and an environment with the proper temperature or humidity.

The embedded system needed to incorporate sensors to measure ambient temperature and humidity, the amount of light received by a plant, and soil moisture content. Additionally, a method to monitor the plant's growth over time and its water supply level was desired. The gardening system was expected to feature a mechanism that would pump water from the supply to the plant when the soil moisture level fell below a minimum threshold. Furthermore, it was essential for the system to include a grow light to provide additional illumination when the light intensity dropped below a specified threshold for a defined duration.

The proposed capstone gardening system utilized commercially available, inexpensive hardware components to ensure that the embedded system remained affordable, modular, upgradable, replaceable, and repairable.

It was determined that plants within this gardening system would be grown in soil rather than in a hydroponic system. Growing plants in soil was recognized as more cost-effective and requiring less maintenance than hydroponic setups, which demanded more expensive supplies and a sanitary environment with oxygen-rich, distilled water.

The plant selected for this project was cat grass, due to its ability to germinate and sprout quickly, with a growth period of 1 to 2 weeks. This choice facilitated multiple growing sessions, allowing the gardening system to be refined as needed.

Frontend

The web application was designed to include 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 highlighted the application's primary features. The dashboard page displayed sensor feed data in both graph and table formats. The chat page incorporated a simple, user-friendly chat messenger, and the chat history persisted only for the duration of the user's session, with the option to clear it if desired.

The auth page(s) provided users with the capability to log in, log out, and sign up. User authentication remained active throughout their login session, and upon logging out, the user was no longer authenticated. If users were not signed in, they could only access the auth page. Conversely, signed-in users had access to all application pages.

In addition to being complete, functional, and polished, the web application was designed to be user-friendly and intuitive. It maintained a consistent color palette and theme throughout the site, ensuring a cohesive user experience. The application was also responsive, enabling functionality across various screen sizes.

Backend

The backend application was proposed to be RESTful, with publicly accessible endpoints. The backend services had three main functionalities: 1) retrieved feed sensor data from an external source and processed and formatted it to be usable and readable once it was passed to the frontend application, 2) took a user prompt related to the capstone gardening system or plant information and provided a tailored response related to the prompt, and 3) ensured the agentic component of the backend service had the guardrails to only answer questions relevant to the project and provided a response accordingly for irrelevant prompts or errors.

Design and Implementation

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

To compensate for the limited knowledge and background in Embedded Systems and Design, the C++ programming language, IoT Principles and Design, and Agentic Orchestration, the capstone project begun a few months before the Fall 2025 academic semester.

Shortly after Professor Fredericks approved the capstone project idea, research and development, prototyping, and design for the embedded component started in July 2025. A similar process for the agentic orchestration system followed in August 2025.

In the short time spent on research and development, prototyping, and building throwaway applications, the knowledge base expanded, reusable code templates were developed to shorten development time, and the project plan and scope was developed and refined.

Embedded Hardware and Application

Embedded System + Circuitry

For the embedded system, a microcontroller was needed. It served as the brain of the embedded component in the gardening system. The microcontroller was a small computer with an integrated circuit, RAM, processor, and I/O pins (IBM) required to connect the sensors and associated devices physically (Schneider & Smalley, 2025). The microcontroller was programmed to perform functional logic. The microcontroller captured sensor data and communicated with external services via various communication modules and protocols, such as Wi-Fi/Internet or Bluetooth (Schneider & Smalley 2025).

For the capstone project, there was a choice between the ESP32 and the Arduino Uno R4. Ultimately, the Arduino Uno R4 was selected over the ESP32 because it was 1) user-friendly, 2) had many official documents, 3) could supply both 5v and 3.3v of power, which allowed more flexibility in terms of sensors and devices used in the gardening system, and 4) had the I/O peripherals pre-soldered with female pin headers (Arduino).

The microcontroller was paired with a mini-breadboard, which was used to build the electronic circuits for the various sensors and devices (pump and grow light) 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 grow light's wires, after the internal circuit board was removed. The embedded system 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. The use of a breadboard ensured the embedded system remained easily configurable and modifiable.

In terms of sensors, four types were used in the project. The DHT11 sensor measured an environment's temperature and humidity. The photoresistor measured the light intensity of an environment. The soil conductivity sensor measured soil moisture content. The ultrasonic sensor, which typically measured distance by emitting a sonar signal and detecting its return, measured the time it took 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, enable IoT, and send data to a remote database, the Adafruit IO Cloud Service was utilized. AdafruitIO served as an MQTT broker and a cloud database.

Several built-in classes and third-party libraries were incorporated into the Arduino code to establish a connection to the AdafruitIO Cloud Service. The "WiFiServer" class was employed to set up and establish a WiFi connection to the local home router (Arduino). The "WiFiSSLClient" library and class were required and utilized to establish an SSL (Secure Socket Layer) connection to the Adafruit IO server (Realtrek). An SSL certificate was necessary for AdafruitIO to establish a secure, encrypted connection to its server.

The "Adafruit_MQTT_Client" library and class were used to establish and manage network connections and communication between the Arduino and the Adafruit IO (MQTT Broker) (Ada, Cooper & Cooper, 2024). The "Adafruit_MQTT" library contained all the MQTT logic that enabled the board to publish data to and subscribe from Adafruit IO (Ada, Cooper & Cooper, 2024).

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 contained several variables and class objects at the top of the program that were declared, initialized, and/or instantiated. At the top of the program, the sensor threshold variables were established, and the WiFi Server, WiFi SSL Client, Adafruit MQTT Client, and various Arduino Task Timer objects were instantiated.

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

The setup() function in the embedded application was used to 1) initialize serial communication, 2) set up and establish the WiFi connection to the router, 3) register Adafruit IO's CA certificate, 4) set up the

pins connection and mode for the sensors, 5) request test readings from all of the sensors and serialize the values, and 6) instantiate any class objects that were only initialized at the top of the program. The loop() function in the embedded application was used to periodically 1) capture sensor readings and serialize them, 2) publish those sensor data to Adafruit IO, and 3) manage the grow light and pump trigger events along with activation duration, all on different concurrent timers.

The embedded application utilized three custom classes: "ArduinoTimerTask," "Adafruit_Helper," and "Unit_Converter."

The custom class "ArduinoTaskTimer" was created to streamline the creation, checking, and resetting of timers. The class provided a reusable, cleaner code base and allowed 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 the logic and setup required to connect to, publish to, and subscribe to the MQTT broker.

The custom class "Unit_Converter" was created to provide a means to convert units of time from hours, minutes, and seconds to milliseconds, and back. However, it was not limited to units of time and included one variable that could be used to help convert between units of volume.

Frontend Web Application

The frontend web application was built using the Angular Framework. Angular was chosen over React because it had many built-in libraries and features that reduced the need for third-party libraries that might not have been maintained.

Angular supported several control flow syntaxes that helped users conditionally render elements in components. It also had a built-in system that managed data states in services and components (Signals) and detected changes within an element to trigger re-rendering (ChangeDetectionRef). Classes and services could be dependency-injected into an element to preserve data states and share signals among other components with either the component's class constructor or Angular’s Inject function.

The PrimeNG UI Component Library, the PrimeFlex CSS Utility Library, and the Tailwind CSS Utility Framework were all utilized 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 display the desired content and data and were implemented with logic.

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

Each main page in the web application functioned as 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 displayed all chat messages between the chat completion agent (assistant) and the user. The chat form provided users with a text box and buttons to type, submit, or clear the chat logs. The chat was only stored per user session.

The auth page had three components. Each component had its own templates and collectively, handled user's login, logout, and sign-up. They rendered one at a time, depending on whether certain conditions were met.

The web application also included and utilized the Firebase Auth library, which allowed access to and interaction with Firebase Authentication services to handle user authentication during sign-in, sign-out, and sign-up. Depending on the user's authentication state, they could either access all pages in the application or only the auth page. Routes were protected using Angular's built-in Auth Guard.

The application had four internal services. Two of the services connected 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 were shared services that could be injected into various components to share common data, such as auth state, light or dark mode state, and more.

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

The backend service was built with ASP.NET in Visual Studio. The backend service followed the Controller-Service-Repo architecture. There were no functional repo classes, but there were two primary internal controllers and services in this application: the Adafruit Controller and Service, and the Agentic Chat Completion Controller and Service. The controller abstracted the services' logic and exposed the application to the internet. Each controller also had its own custom CORS policy, and the backend application was integrated with Swagger UI. The services housed the majority of the business logic that made up the backend.

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

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

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

AI Agents were software objects that used AI from an LLM to process information, complete tasks, and respond to prompts autonomously and independently of users, while also being able to coordinate with users (Microsoft, 2024). Agents could be assigned specific roles and return specific outputs, provided with a system prompt or a role description and output description during instantiation (Microsoft, 2024). Agents could also be assigned plugins. Plugins were a collection of Kernel Functions, programmatic algorithms wrapped in a method, that were used to perform specific tasks and provide a specified return.

Every agent required 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 (Microsoft 2025). For this project, a single master kernel was created in the custom, static KernelFactory class. Each agent cloned and used a copy of the master kernel and registered the plugin(s) required for its specific needs.

Orchestration was a process that drives agents to collaborate and coordinate with others toward a mutual objective (Microsoft 2025). The orchestration pattern used in the Agentic Chat Completion Service was the Handoff Orchestration. Handoff orchestration was a process used to get a group of agents to coordinate with one another to find an agent who could best fulfill a user request and assign responsibility to them (Microsoft 2025). The handoff orchestration identified the agent responsible for handling the request by evaluating each agent based on their role, responsibilities, and tools, and selecting the one that best matches the prompt or request's context (Microsoft 2025).

The agents created for the handoff orchestration were the Chat Moderator agent, the Adafruit Feed agent, the Project Info agent, and the Plant Info agent. The Adafruit feed agents had kernel functions that pointed to methods in the Adafruit Feed Service. These methods used an HTTP Client to reach out to the Adafruit HTTP API to retrieve feed data from various sensors; these agents answered questions associated with the sensor feed data and analyzed and summarized it. The Chat Moderator agent acted as the switch or relay operator in the Handoff Orchestration. It was the first agent to receive user prompts, and based on the prompt's context, it determined which agent could best respond to the prompt. The Project Info agents responded and provided details related to the semi-automated plant gardening system.

The Chat Completion Service also had another agent independent from the orchestration, the Chat Editor agent, which took the response generated by the Handoff Orchestration, edited and streamlined it, and returned it to the user.

LLM

Microsoft Azure was used to host OpenAI's LLM Model, o3-mini, which generated chat message responses. The decision to host the model on Azure was motivated by its affordability and accessibility. The o3-mini LLM model represented one of the more economical options available for cloud hosting, making it accessible to any user with the appropriate authentication. A hosted model proved essential for deployed applications requiring online access. For users who prefer to use a local or free LLM model,

Microsoft's Semantic Kernel provides this option through its extensions. In one prototype, the model was successfully swapped for a locally run LLM, specifically Ollama's TinyLlama. The process was streamlined by Microsoft's Semantic Kernel, which required registering the locally deployed Ollama Chat Completion Service in the kernel and providing the model ID and local endpoint.

Deployment

For deployment, Google Cloud Platform (GCP) was utilized to host the frontend and backend applications via Google App Engine, Cloud Run, and Artifact Registry. The frontend application was built with the Angular CLI, then copied to a directory on Google Cloud Shell along with a YAML configuration file, and deployed to Google Cloud App Engine using the Google Cloud Shell CLI.

The backend application required more effort. A Docker image of the application was created and pushed to Google Cloud Platform's Artifact Registry. Then, from Google Cloud Shell, the image was added to a container in Google Cloud Run using the Google Cloud Shell CLI.

To create the Docker image, a Dockerfile for the application was developed and configured to include the necessary files. A Docker image of the application was built using a Docker CLI command in PowerShell and was assigned a specific tag recognized by Google Cloud Platform's Artifact Registry. To push the image to GCP's Artifact Registry, the Google Cloud SDK was downloaded and logged in, and Docker was configured and granted access to the GCP account.

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.

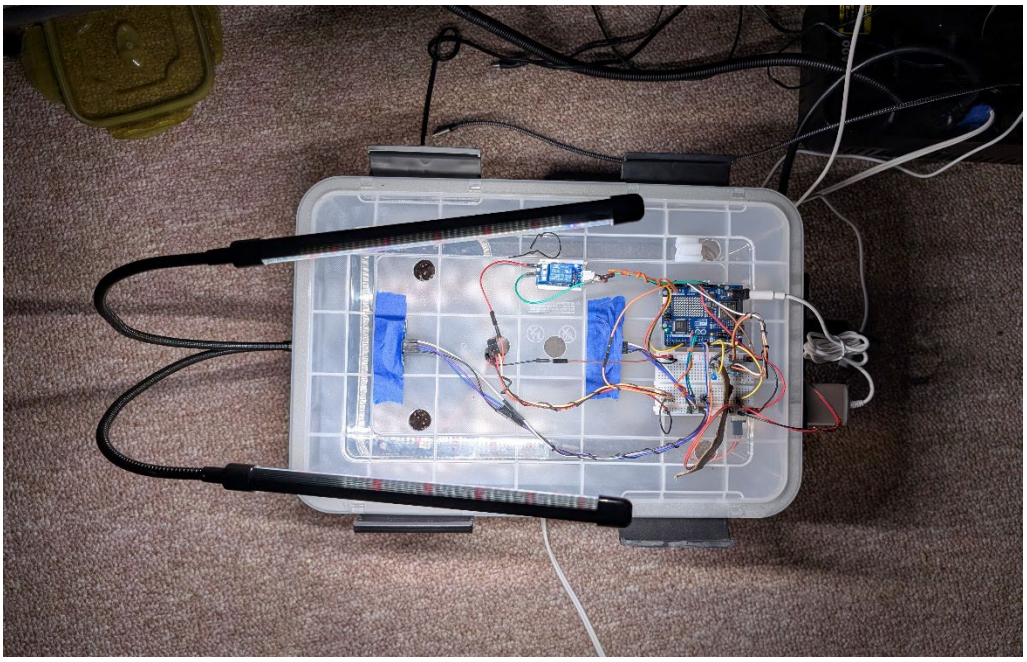

Photo 1: Semi-automated gardening system with the top on and growlight activated. At the start of the first growing session.

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, a container was filled with soil and cat grass seeds and left undisturbed at the end of September for an entire week.

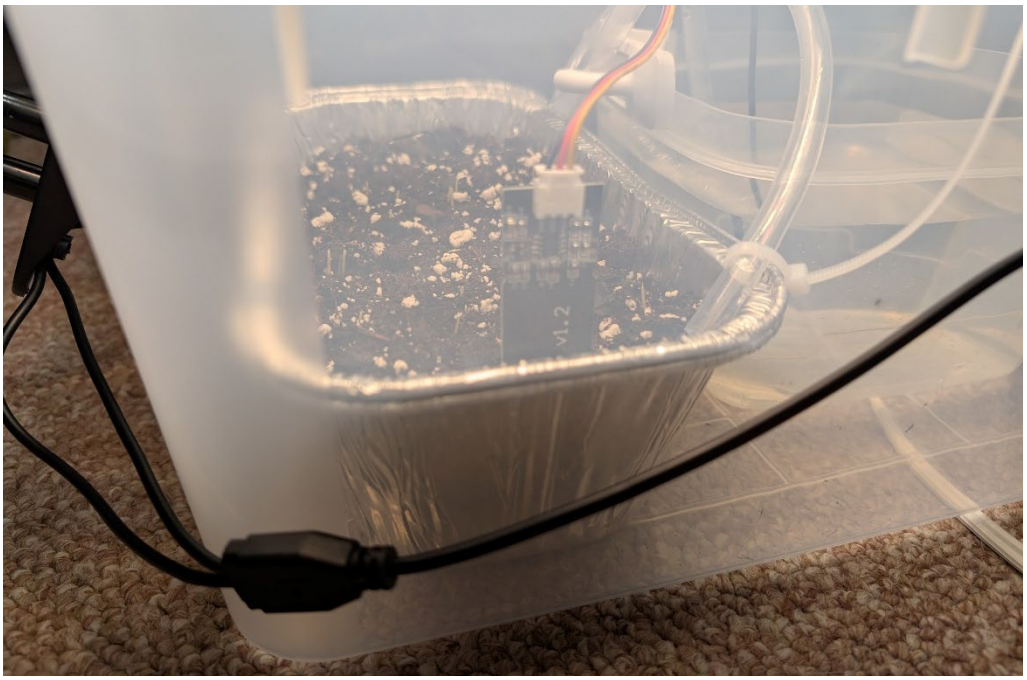

Photo 3: Close up of the potted container inside the gardening system. Top soil layered with cat grass.

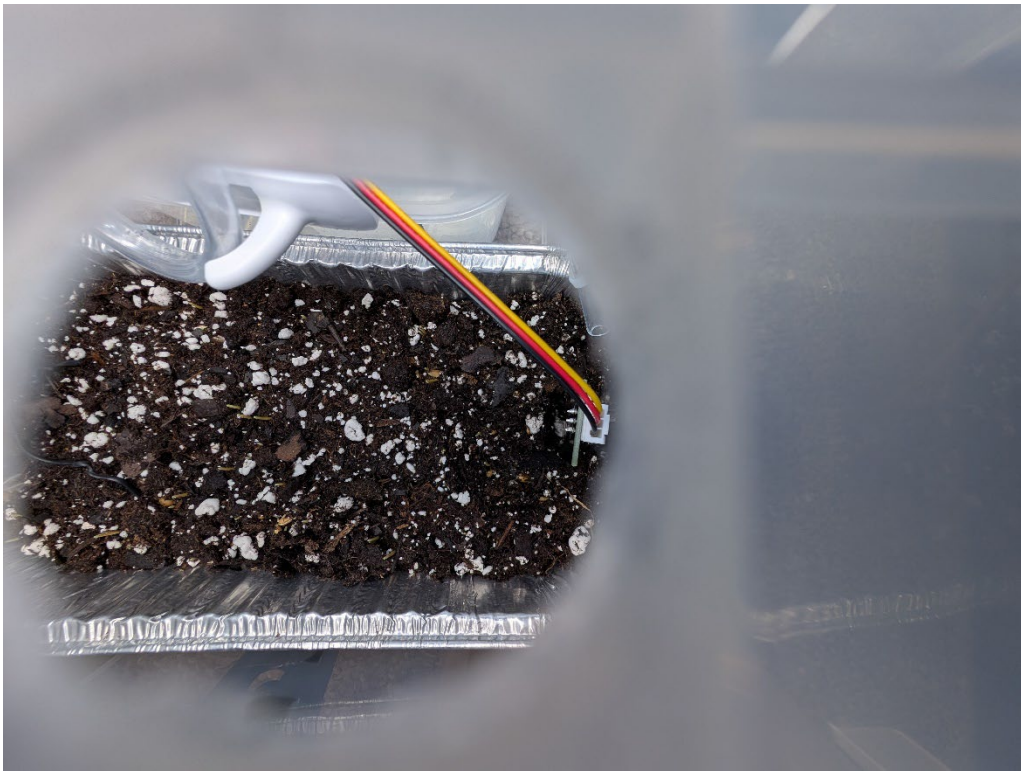

Photo 4: A top view of the potted container inside the gardening system. Top soil layered with cat grass.

When the gardening system was revisited a week later, it had successively grown cat grass, autonomously.

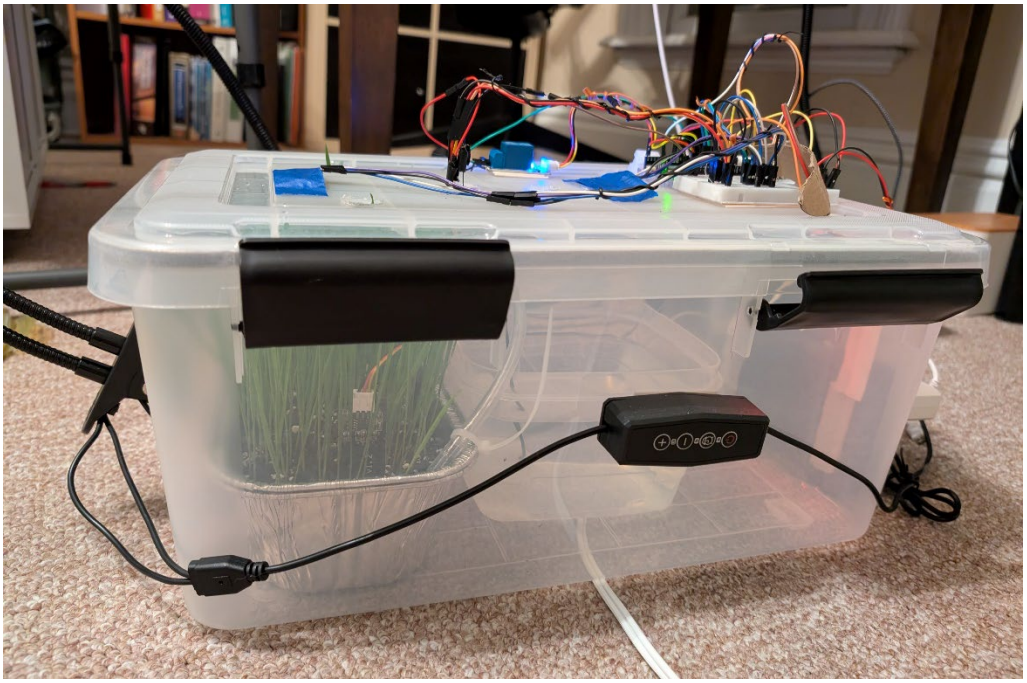

Photo 5: A side view of the cat grass growing inside the gardening system.

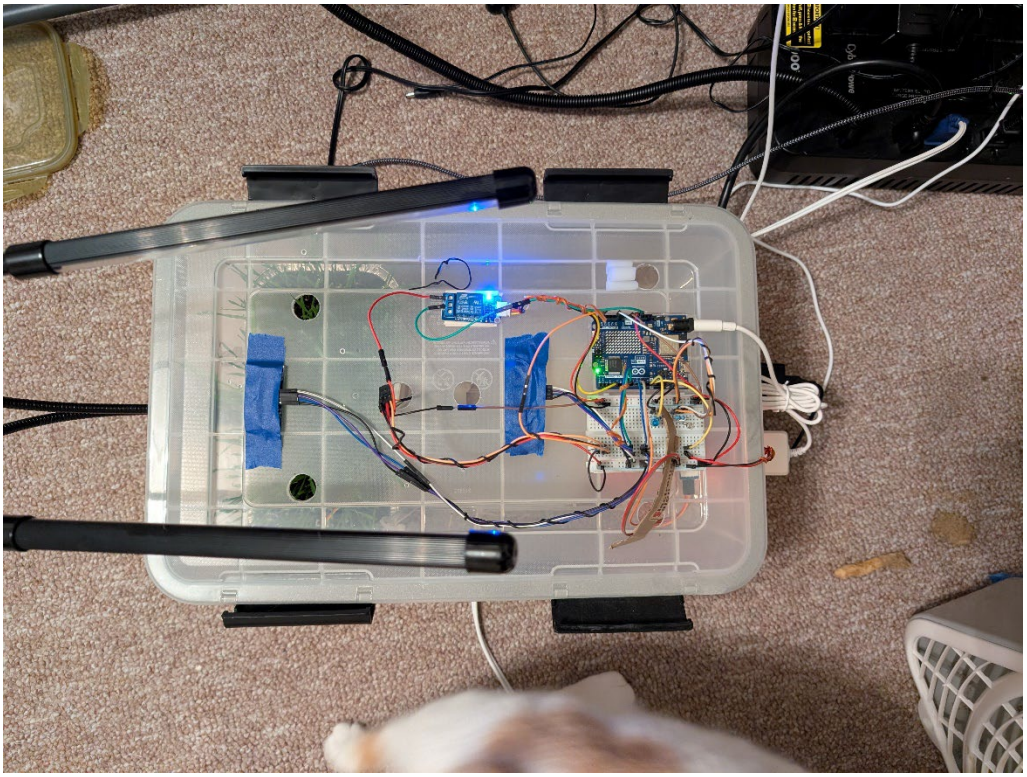

Photo 6: A top view of the cat grass growing inside the gardening system.

Photo 7: Sammy, enjoying the harvest from the gardening system's first growing session.

Checking Adafruit IO's cloud service confirmed that the service had logged and recorded all of the sensor data published to it into their respective feed database during the week without human interference.

Unfortunately, the free version of Adafruit IO had only enforced a 30-day retention policy, so data from the initial growing session had since been deleted. However, the gardening system remained connected to Adafruit IO and continued to publish data to it.

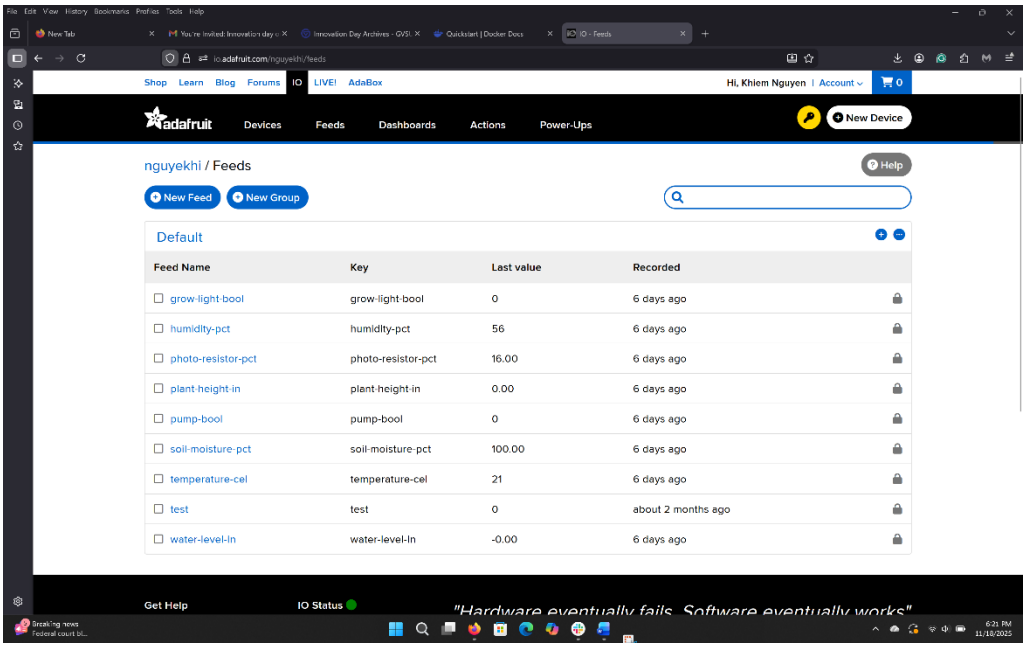

Photo 8: Adafruit IO Feed Databases

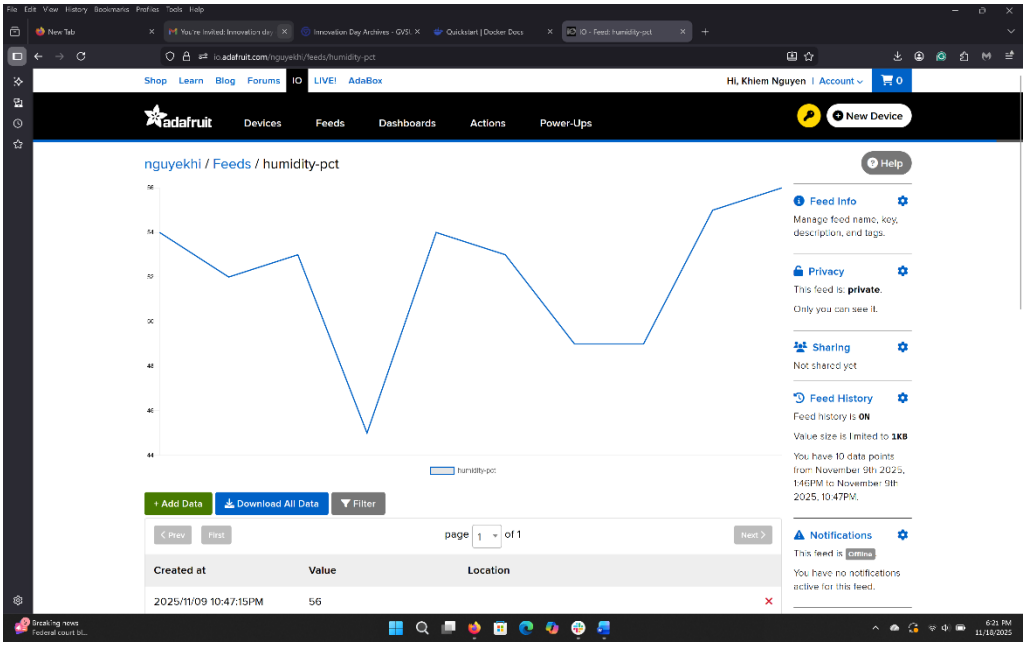

Photo 9: Adafruit IO's Humidity Feed Dashboard

Development of both the 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 achieved and deployed onto Google Cloud Platform by mid-November. However, the result was a functional, complete, and aesthetically pleasing application.

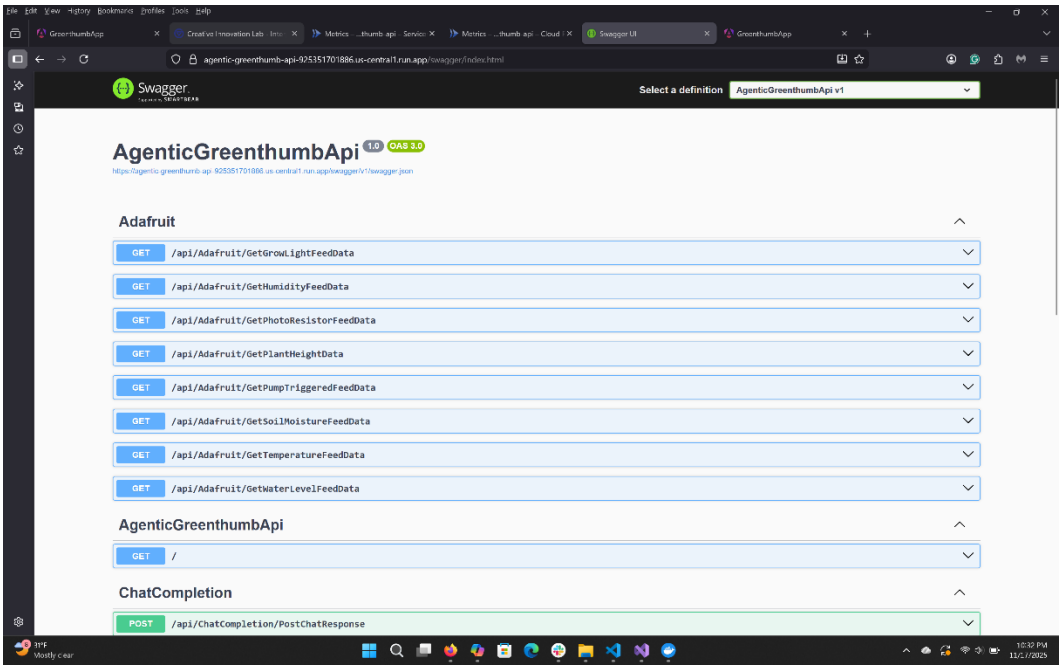

Photo 10: Screenshot of the Swagger UI and endpoints of the backend application

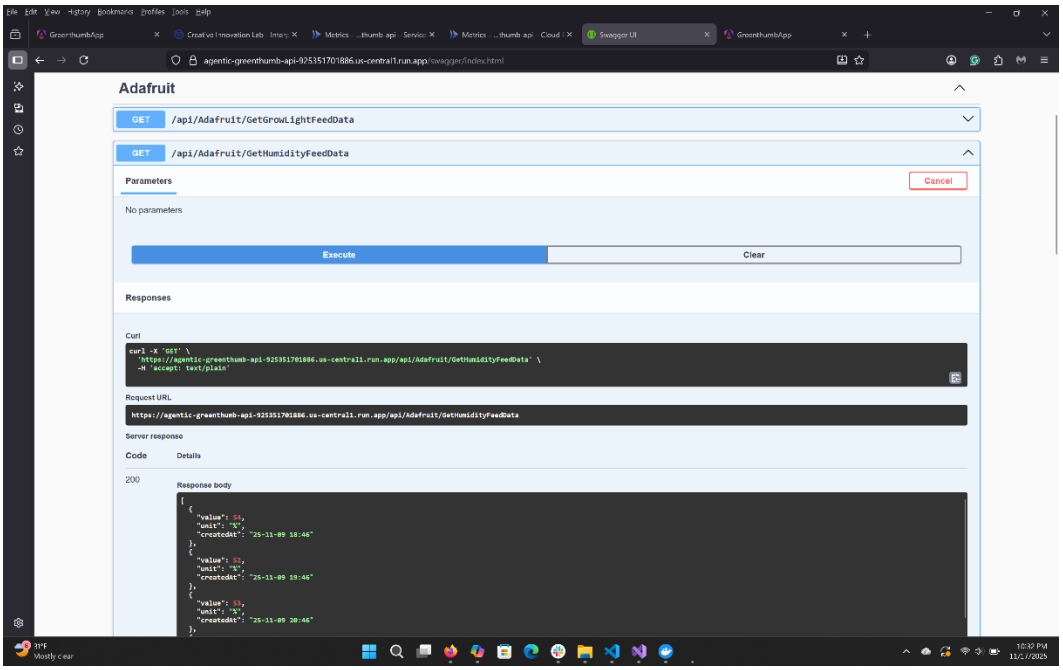

Photo 11: Screenshot of the Swagger UI and one of the backend application's endpoint being tested. The frontend application for Project Greenthumb is feature complete.

The application had a landing page, a set of dashboards, a chat messenger page, and an auth page. The complete set of pages was only available when the user was logged in; otherwise, only the auth page could be viewed. The frontend application also included a light and dark mode that the user could toggle between and was responsive, adjusting to different screens.

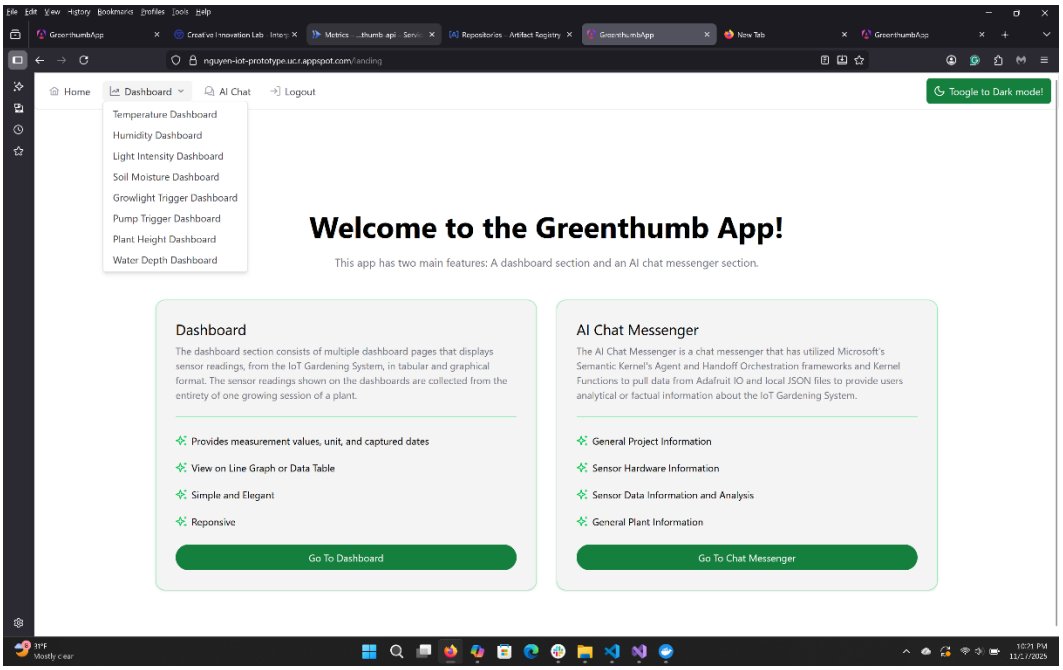

Photo 12: The landing page of the frontend application. Light mode.

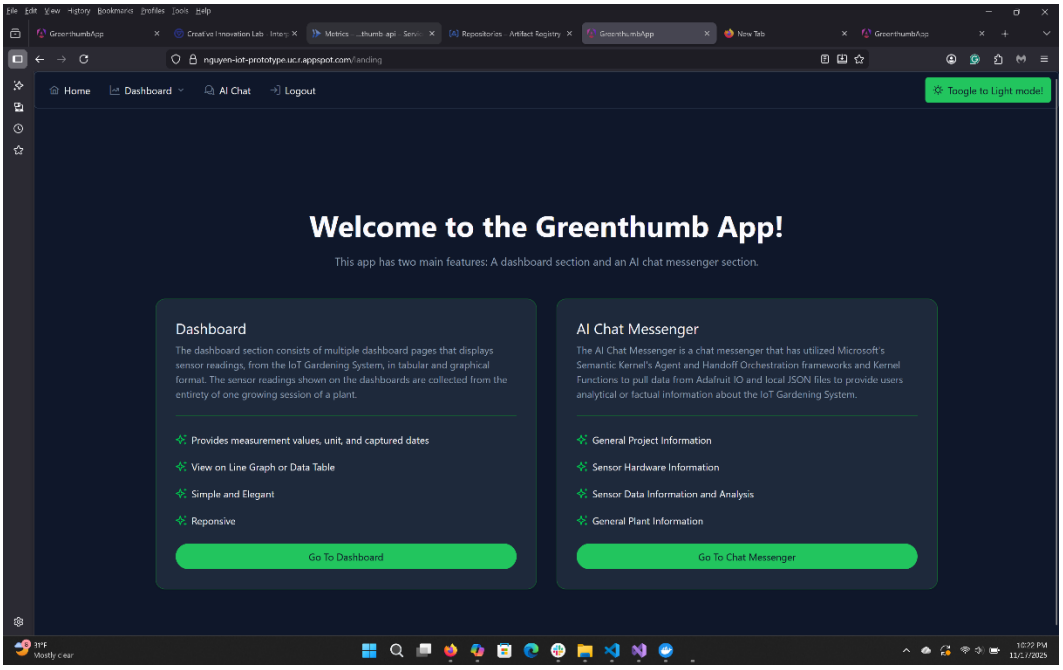

Photo 13: The landing page of the frontend application. Dark mode.

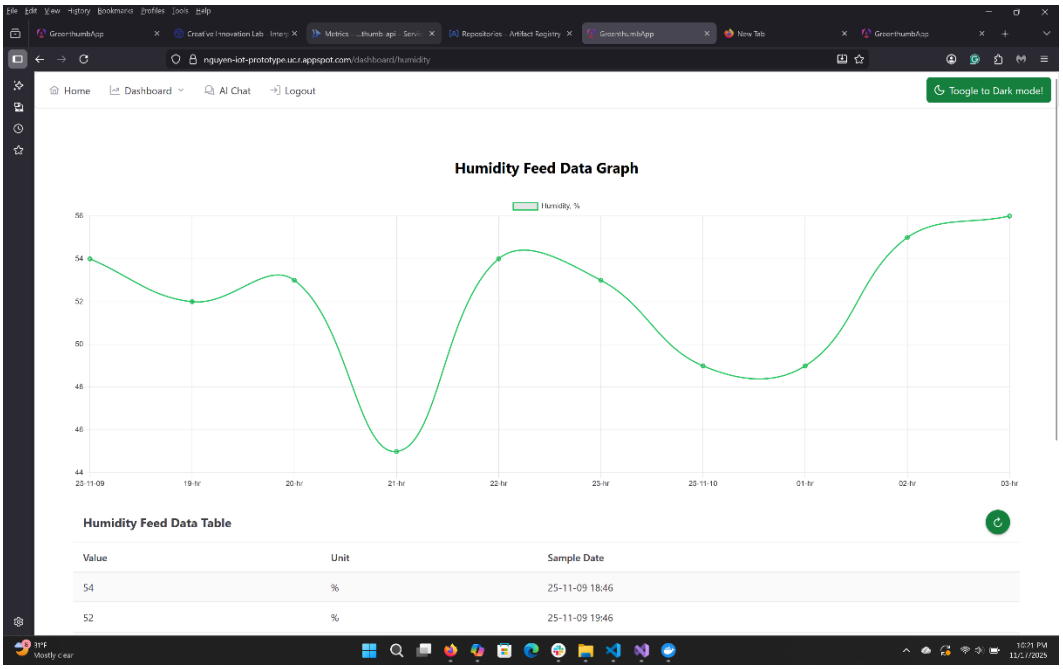

Photo 14: The humidity feed dashboard page of the frontend application. Light mode.

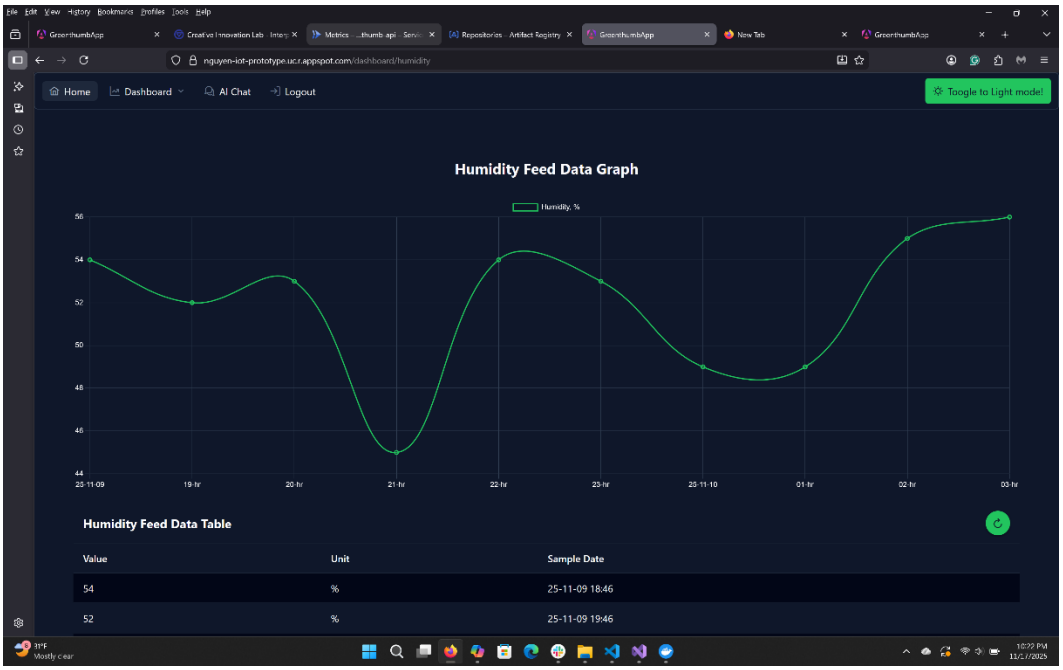

Photo 15: The humidity feed dashboard page of the frontend application. Dark mode.

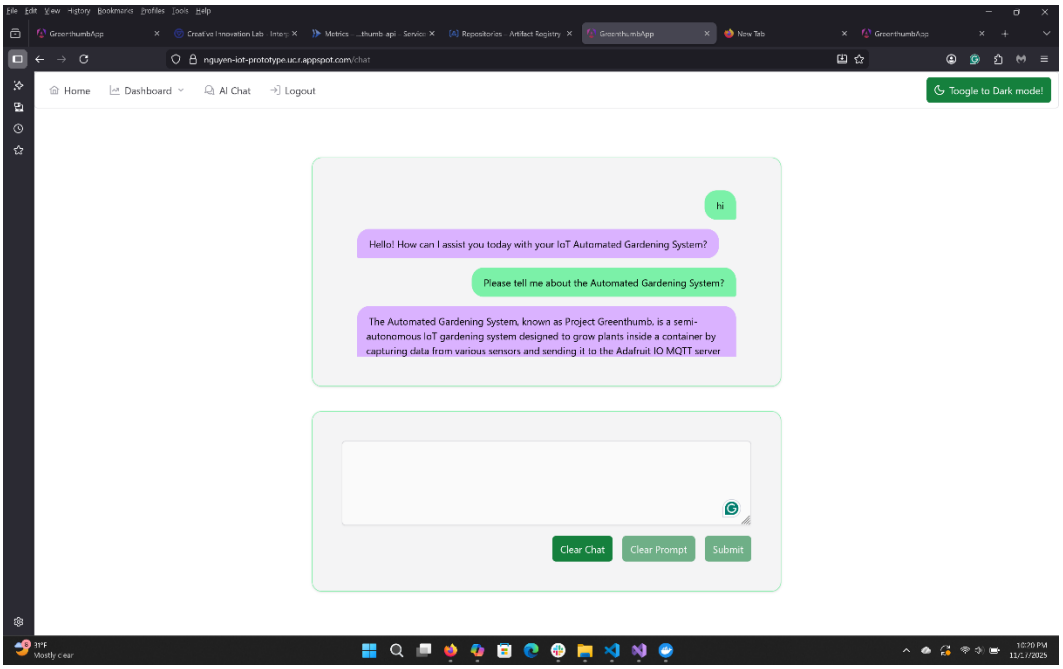

Photo 16: The chat messenger page of the frontend application. Light mode.

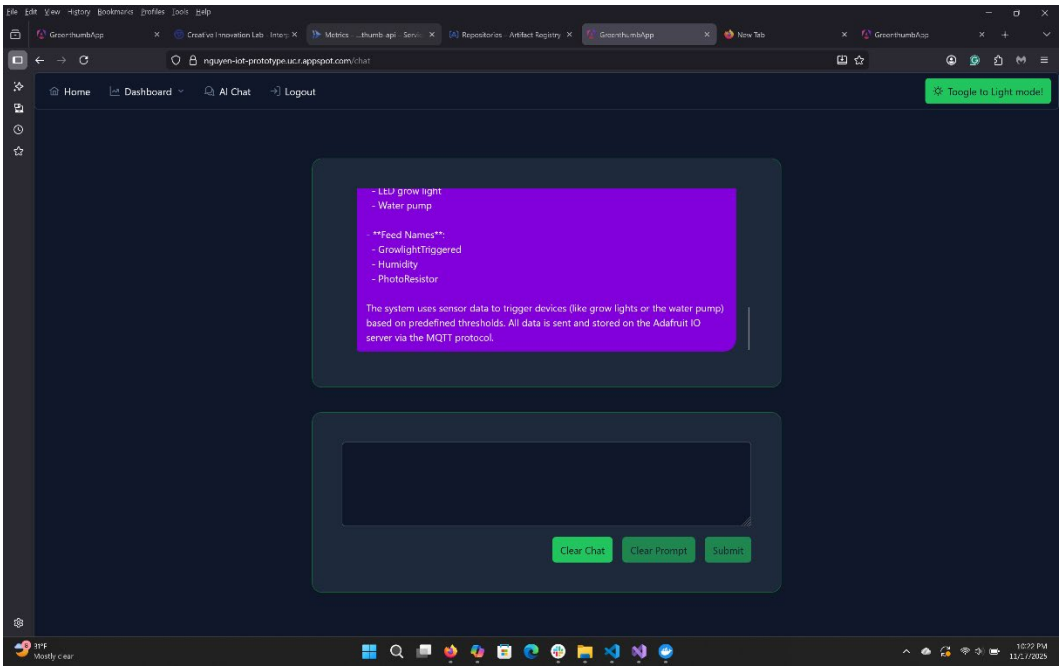

Photo 17: The chat messenger page of the frontend application. Dark mode.

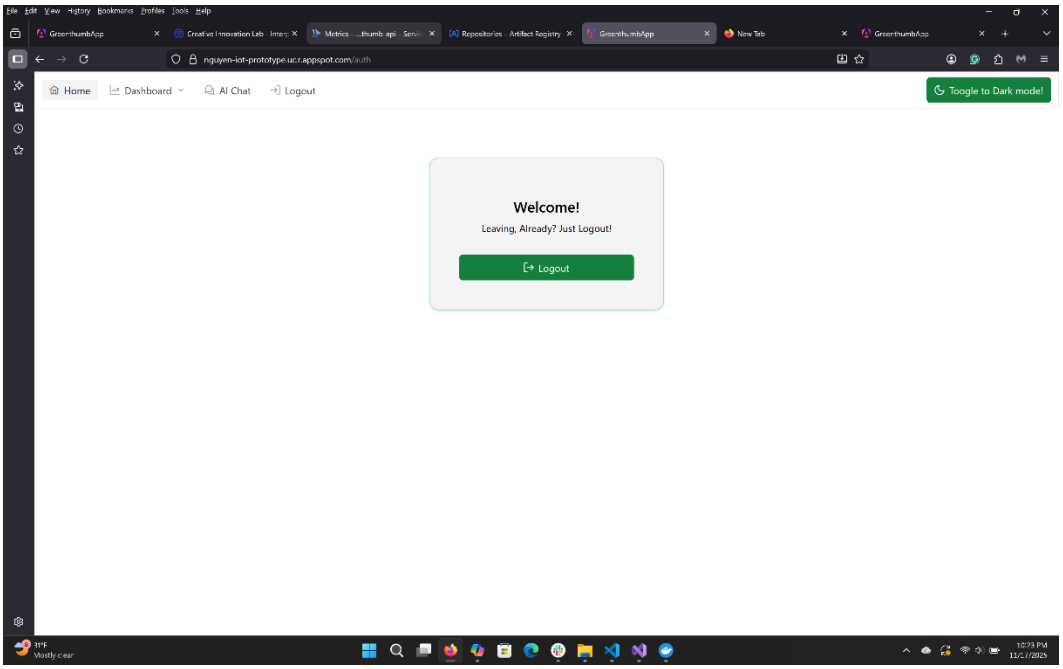

Photo 18: The auth page of the frontend application, during the logout view. Light mode.

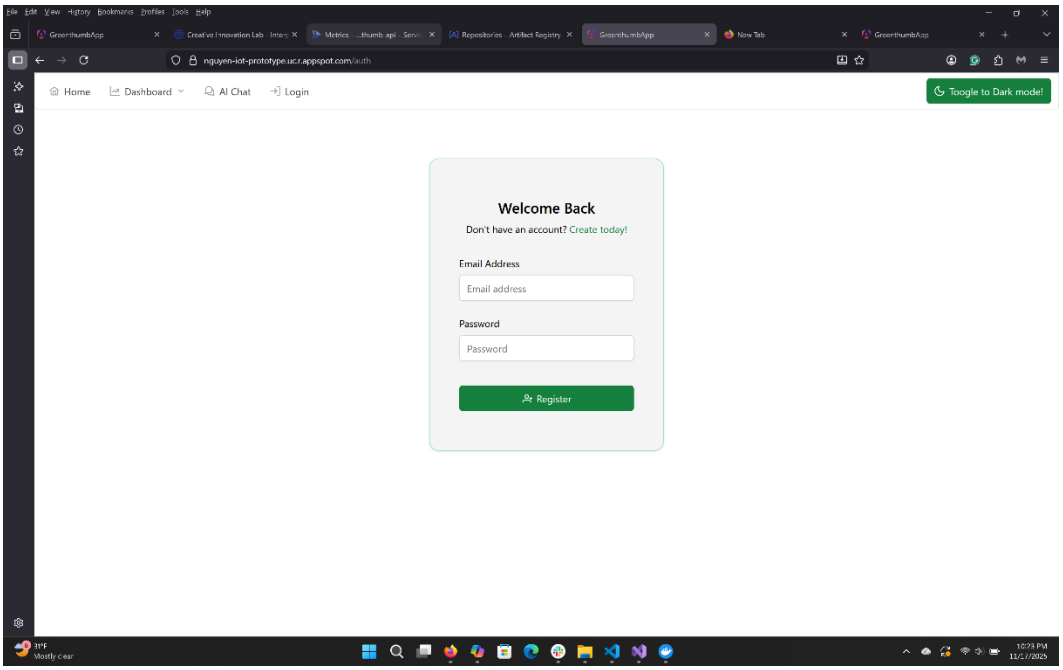

Photo 19: The auth page of the frontend application, during the login view. Light mode.

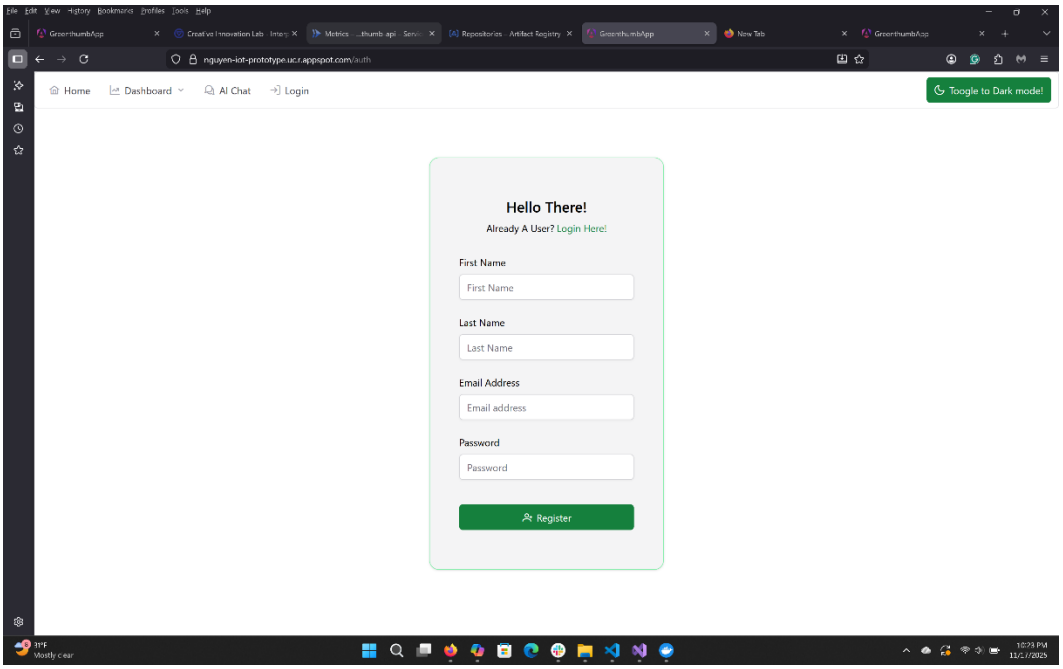

Photo 20: The auth page of the frontend application, during the signup view. Light mode.

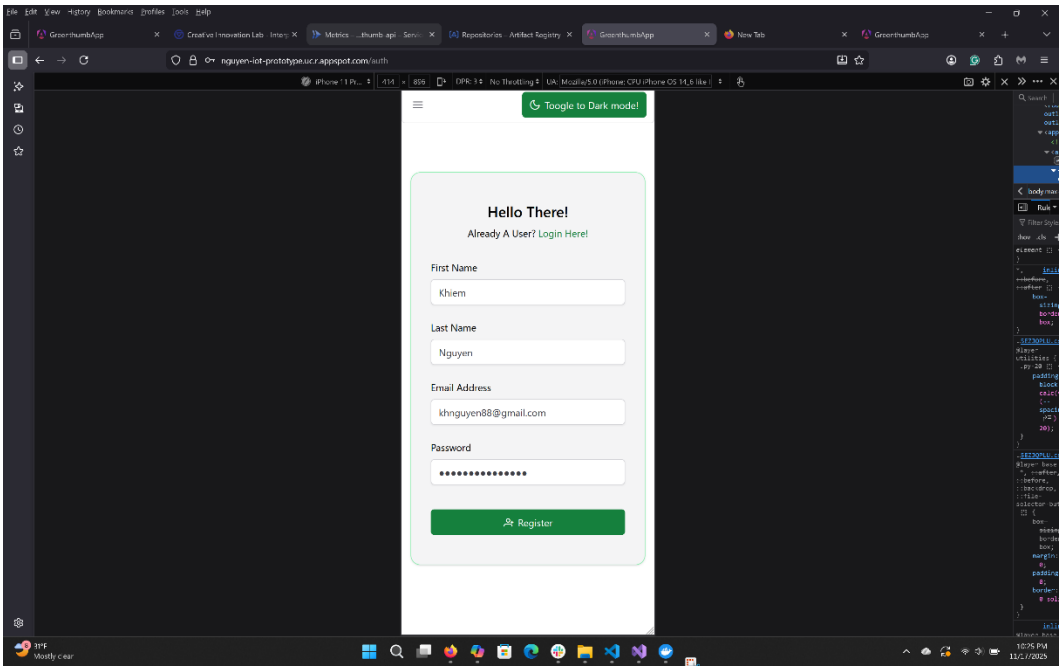

Photo 21: The auth page of the frontend application. Simulated on mobile. Light mode.

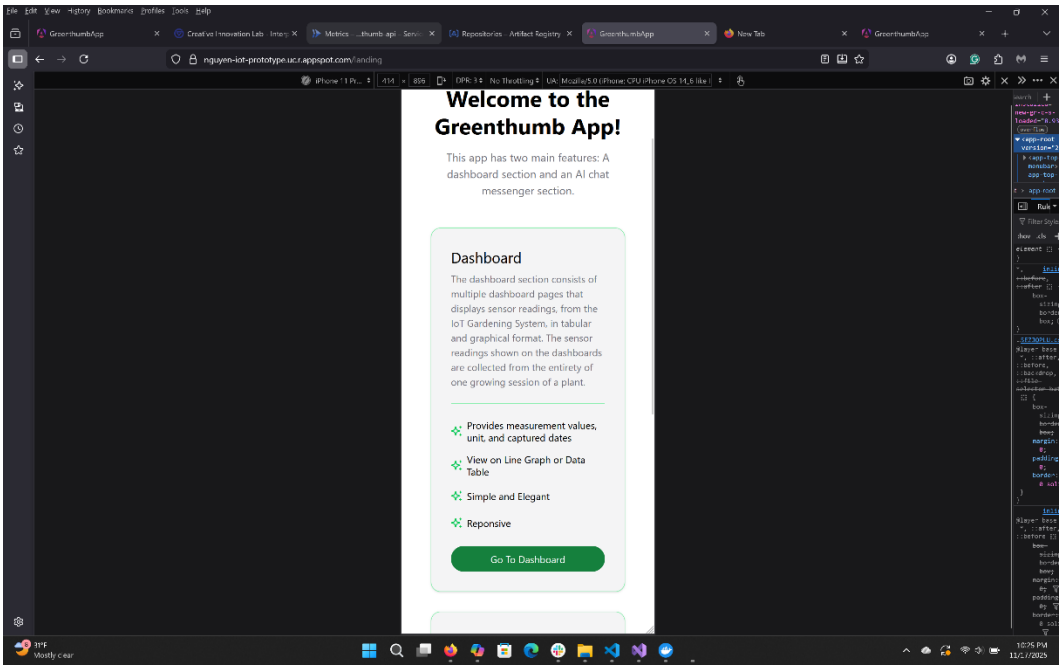

Photo 22: The land page of the frontend application. Simulated on mobile. Light mode.

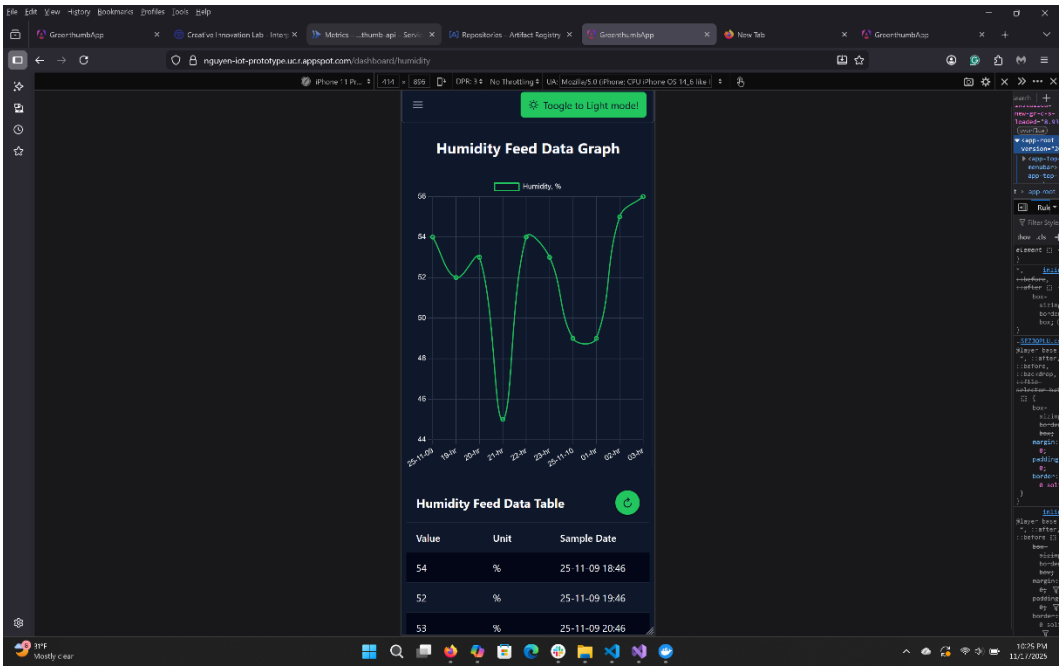

Photo 23: The dashboard page of the frontend application. Simulated on mobile. Dark mode.

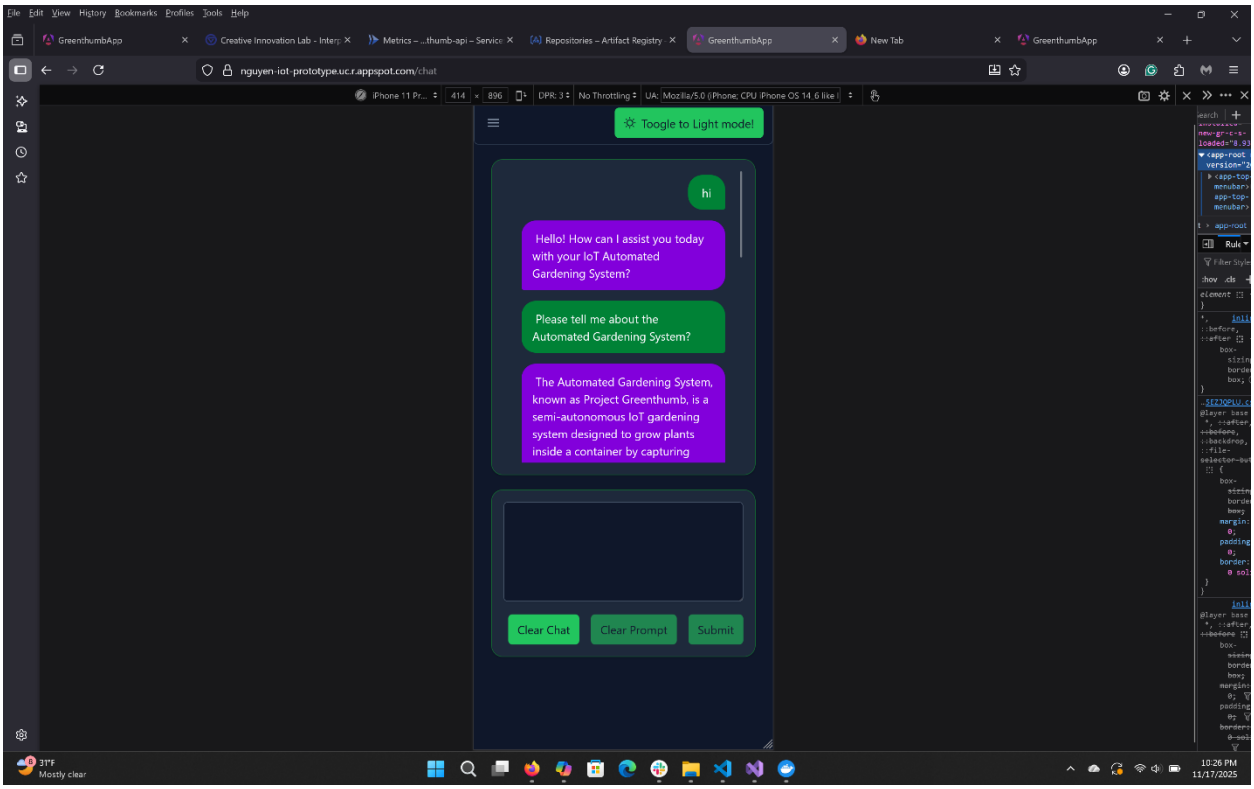

Photo 24: The chat messenger page of the frontend application. Simulated on mobile. Dark mode.

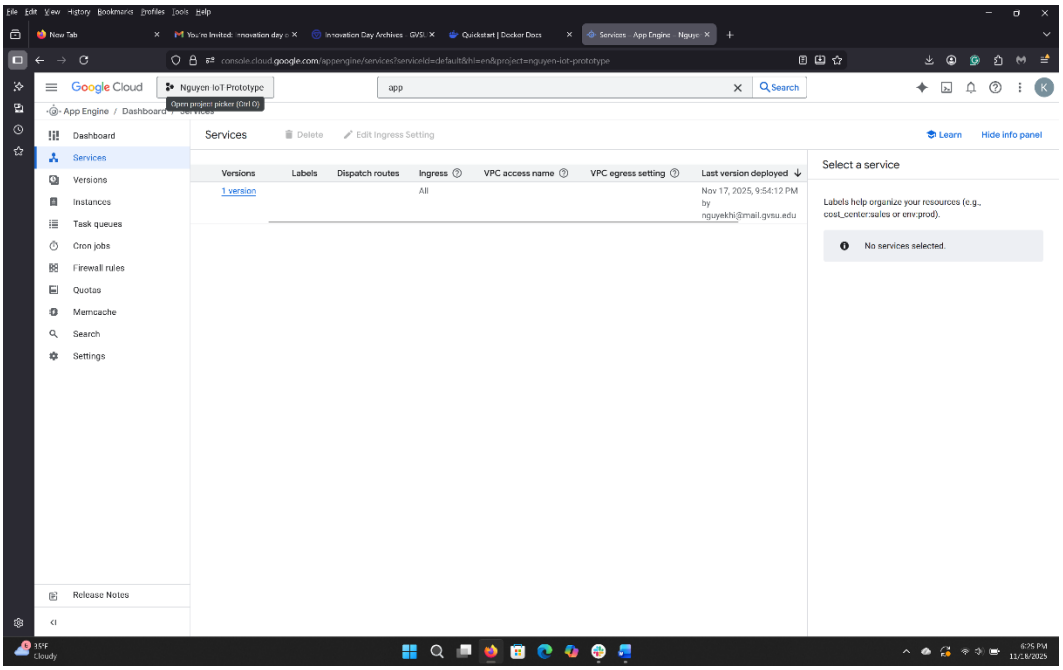

Photo 25: Dashboard of GCP's App Engine showing the frontend application being hosted.

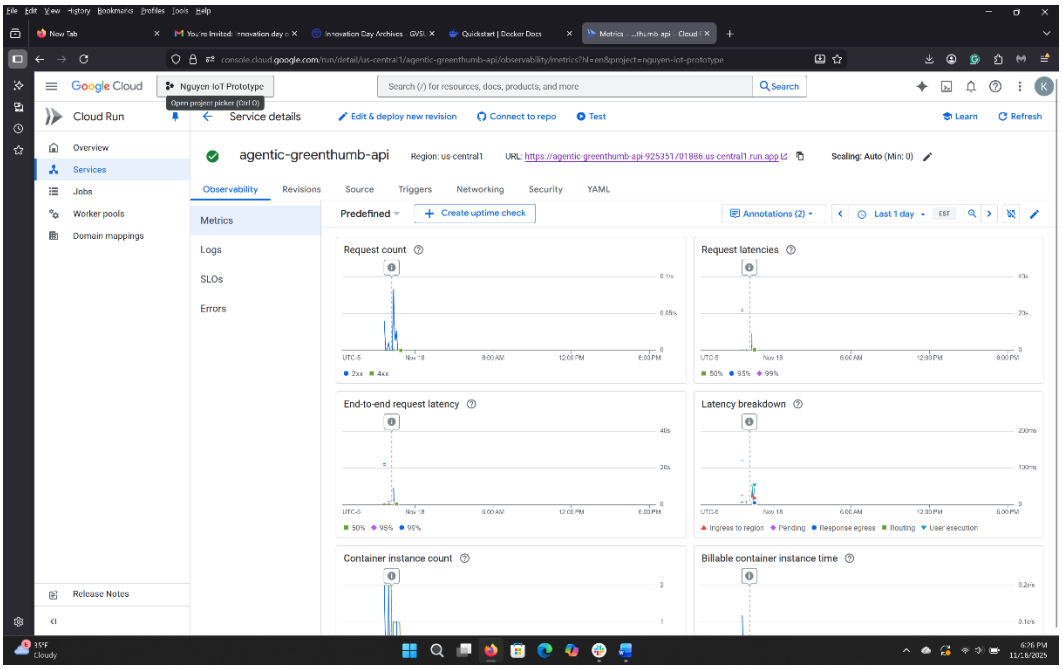

Photo 26: Dashboard of GCP's Cloud Run showing the backend application being hosted.

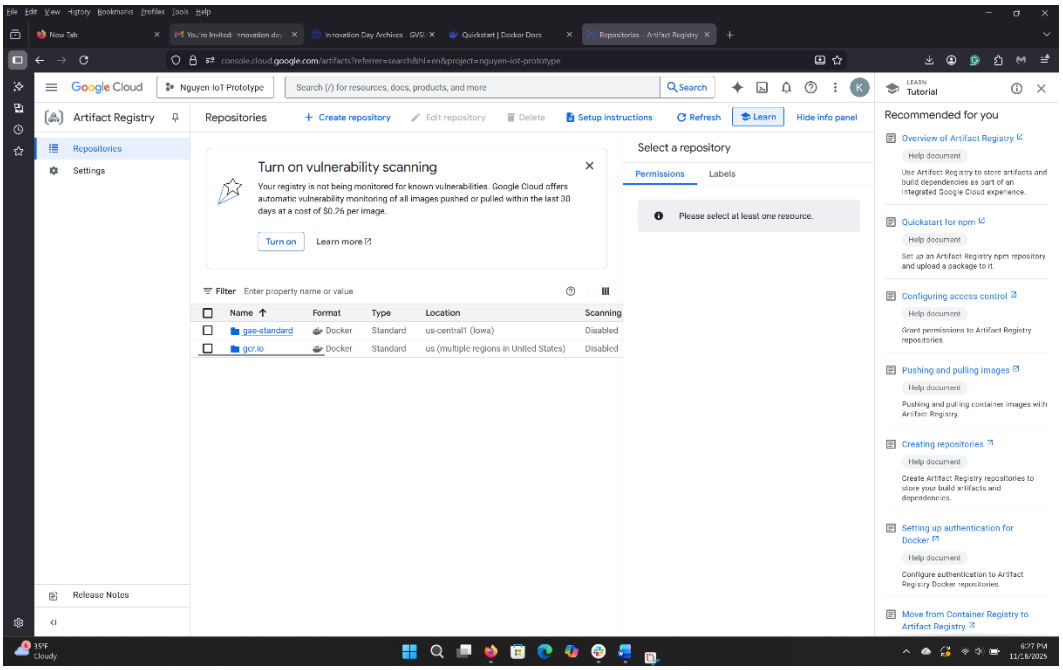

Photo 27: Dashboard of GCP's Artifact Registry showing an image of the backend application.

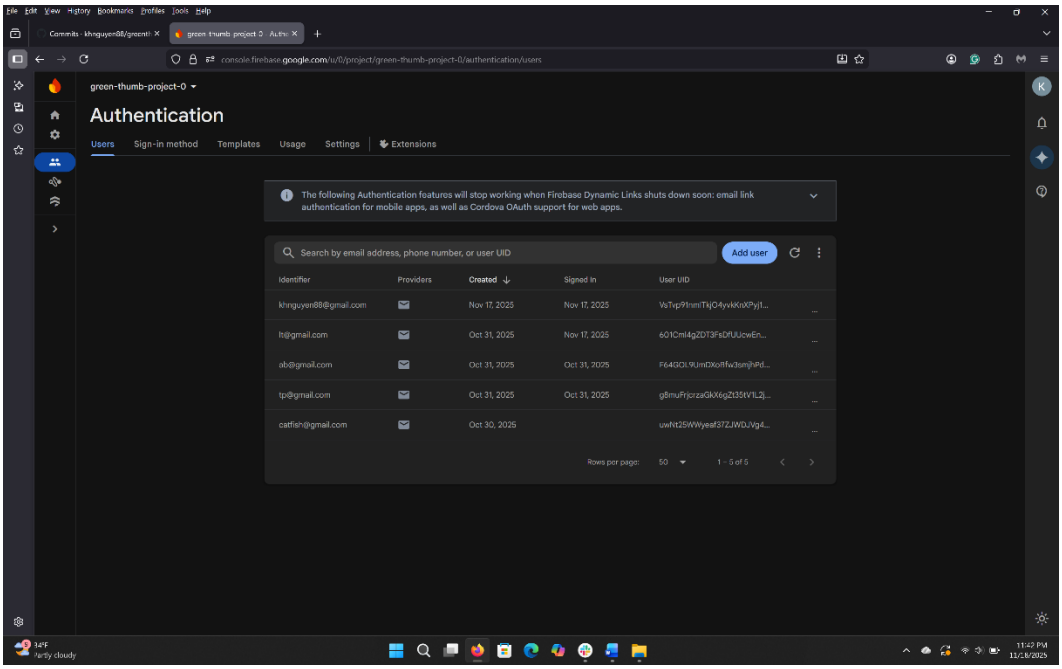

Photo 28: Dashboard of Google Firebase's Authentication showing a list of registered user.

Challenges

Embedded Hardware

The general challenge of the hardware component of the embedded system involved two primary tasks: 1) learning and applying basic circuitry within a limited timeframe, and 2) debugging and determining the root causes of erroneous sensor readings.

Fortunately, microcontroller boards such as Arduino and ESP32 were widely used devices with plenty of guides, documentation, forums, schematics, and other resources online. These resources facilitated the necessary knowledge acquisition for the capstone project and effectively addressed the first challenge.

The second task, however, proved to be more difficult to overcome. Numerous instances occurred in which the sensors provided either faulty or no readings, with various potential issues contributing to inaccurate results. This led to the development and implementation of an internal checklist to assist in the process of elimination and identify root causes. Potential problems included defective sensors, incorrect wiring, poor connections between pins and connectors on the breadboard, malfunctioning microcontrollers or wires, faulty breadboards, defective resistors, erroneous code, and more. These issues arose across multiple sensors, including ultrasonic sonar, photoresistors, the water pump, and the grow light.

Embedded Application

For the application side of the embedded system, the challenges faced included 1) understanding, learning, and capitalizing on the basic features of C++, 2) obtaining C++ support for VS Code, 3) writing code that worked within and outside of the Arduino IDE, and 4) creating a clean sketch (main application) file for the Arduino.

Of all the challenges encountered with the embedded application, understanding and learning C++ proved to be one of the hardest. While familiarity existed with programming in higher-level languages such as Typescript, JavaScript, Python, and C#, C++ felt different due to its more archaic and lower-level nature. Pointers and references needed to be used frequently in the embedded application, especially when passing them as parameters to helper methods. The concept of header and concrete/source files was not a concern in other higher-level languages, and the overriding or populating of methods inherited from header or parent classes required a different setup; for example, scope resolution operators (::) were necessary to specify which method the class originated from, even when referencing the parent class. Additional nuisances included memory allocation, especially when initializing an array, as well as understanding “string” and “char” data types, and casting values that needed to be adapted to.

The second challenge involved finding an environment conducive to writing and managing multiple C++ files simultaneously posed another challenge. Visual Studio Code did not support C++ out of the box and required additional extensions and libraries. Even after these extensions were installed, there was

difficulty in getting Visual Studio Code to recognize the Arduino Library that was natively recognized by Arduino's IDE. Consequently, the workflow involved writing in Visual Studio Code with incomplete IntelliSense, followed by compiling and debugging the application with Arduino IDE. Though a hassle, it was preferable to managing multiple source files within Arduino's IDE.

The final challenge involved learning to parse and separate code from the central sketch and source (.ino) file. This process was accomplished while gaining a better understanding of C++ and refining the workflow. Fortunately, prototyping other sample code assisted in determining how to structure the embedded application and identifying which code needed to be parsed into a separate class to maintain some cleanliness in the main sketch file.

Frontend Application

Two challenges were overcome during the development of the frontend application.

The first and biggest challenge involved understanding and implementing the PrimeNG theming system, which governed the global CSS styles of the components used in the application. The library's new theming method utilized CSS variables, each carrying a predefined set of style properties. Styles were linked to design tokens, essentially properties in a theming TypeScript object that a user created to override one of Angular's available base themes.

The issue was that Angular's documentation was inadequate and did not provide the complete list of available tokens and their associated CSS variables that users could override. The features were hidden behind a paid Theme Designer's Figma Extension. The framework's limited documentation resulted in more time being spent on figuring out this feature than on 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 involved learning and understanding Angular's new state management feature, signals, how it worked with Angular's change detection mechanism, and how to use it in the application. Change Detection was a system that ensured the application's UI view reflected the component's internal state, as defined by its property values. It turned out to be very easy and was similar to React's "useState" and "useContext." Prior to this project, behavior had been used for state management, which could be retired with the addition of signals.

Backend Application

During the development of the backend services, three major hurdles were encountered.

The first major hurdle involved 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 were numerous resources and sample codes for both SDKs, most only covered a small portion of the SDK's features, and even then, they became irrelevant as Microsoft updated and made changes to these pre-release packages at a rapid pace. The best resources for learning about these two SDKs included Microsoft's online documentation, GitHub repository, and learning modules. Even with those resources, some features had to be carefully explored to fully appreciate and understand them.

Through the learning module exercises and prototyping, a significant portion of the features provided by Semantic Kernel and Kernel Memory was explored and experimented with. The features relevant to this capstone's backend service were selected, and a rudimentary plan was proposed to incorporate them into the architecture.

The second challenge involved learning to implement CORS for an ASP.NET web application. Implementation had been done previously with Flask and Node.js backend applications, but never with an ASP.NET backend application. Fortunately, many resources were available online from Microsoft and Stack Overflow, and the process proved to be relatively simple.

The final challenge experienced during the development of the backend application was finding a viable way to deploy it to Google Cloud Run. For a React and Angular application, the process was 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. However, this process did not work for the ASP.NET application.

Recognizing that Cloud Run effectively builds an application image and runs it in a container, the process for building and publishing the backend application as a Docker image was researched and utilized. The image of the backend application was then pushed to Google's Artifact Registry Service, pulled from the registry, and deployed to Google Cloud Run using the Google Cloud Shell CLI. This process required building a Dockerfile, specifying the application's required files and directories in the image, installing the Google Cloud SDK, and granting Docker access to the GCP project. Fortunately, resources and tutorials for these tasks were readily available online.

Cloud Services

No significant challenges were encountered with Google Cloud Platform, Microsoft Azure, Google Firebase, or Adafruit IO Service. Having completed the Mobile Application Development and Cloud Application Development courses, navigating and utilizing the services in Google Cloud Platform and Google Firebase was accomplished with ease. Learning and navigating Microsoft Azure was facilitated by familiarity with Google Cloud Platform.

Only minor challenges arose while navigating the Adafruit IO Service and Google Firebase Authentication Service. The main difficulty involved figuring out how to use their respective SDKs to connect and communicate with the cloud services. However, both offered good documentation online, and Google Firebase was particularly popular; there were no third-party resources or tutorials available for it.

Proposed Future Improvement

While the application was usable, many improvements could have enhanced the user experience, improved configurability, and extended the usability of the IoT gardening system components to other projects as well.

Embedded Hardware

The proposed improvement to the embedded system's hardware component involves replacing the ultrasonic sonar sensors currently used to measure plant height and water supply level with alternative sensors. The ultrasonic sonar sensor readings were found to be too inaccurate and unreliable for analysis. Proposed alternatives include a water metering sensor, a LiDAR sensor, and a camera. However, support for ultrasonic sonar sensors remains, as they are a cost-effective option for other users.

Embedded Application

Improvements on the application side of the embedded system focus primarily on refining, cleaning up, and streamlining the existing code, as well as adding new sensor capabilities.

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

The second proposed improvement involves refining the concurrent timers and optimizing the frequency of data publication to Adafruit IO, particularly when devices such as pumps or grow lights are activated and deactivated.

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

The last proposed improvement includes adding new sampling, handling, and processing logic for new sensors, such as a water metering device, LiDAR sensor, and a photo camera. This enhancement ensures that users can easily import and use these sensors in the code as needed. Support for these sensors is pursued because ultrasonic sonar sensors were observed to be highly inaccurate and unreliable for distance measurement.

Frontend Application

An improvement proposed for the frontend application involves adding two settings pages: an Embedded System Settings page and a User Settings page.

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

The User Settings page is designed to offer users UI customizations. For instance, users can select 1) their preferred lighting mode, 2) the primary theme color, and 3) the ability to turn the chat messenger on or off.

Backend Application

Several improvements were proposed for the backend. One significant improvement proposed involves reworking the agent and agentic orchestration setup and instantiation process in the Chat Completion Service to make it extensible and configurable. Currently, all agent descriptions and instructions were hard-coded in their designated registry class and instantiated in the classes that use them.

Microsoft's Semantic Kernel SDK allows agents to be created from YAML template files. This feature will enable the setup of multiple YAML files and the instantiation of all agents through that method in a custom factory class. The YAML files can also adjust each agent's prompt execution settings, specifying whether the AI model runs locally or remotely and fine-tuning the model response. This approach abstracts away the need to create or remove a custom registry class whenever someone wants to add or remove an agent.

Similarly, Semantic Kernel's Handoff Orchestration enables parameters of Agent objects to be passed into its parameter, allowing it to accept either individual agents or an array of agents. This capability facilitates the generation of an array of agents to pass into the constructor, enabling more extensible instantiation of the Handoff Orchestration. This feature was not used in the gardening system's backend application, but would be implemented as an improvement in the future.

Another consideration and design change involves adding a Data service to the backend that is extensible and configurable based on user application settings. The goal is to create a generic data service interface that child data service classes can inherit. At application startup, it will select and use the preferred database service based on the user's settings. This task would be a substantial undertaking, since services such as Local SQL Server, In-Memory, Firebase's Firestore, and local storage have distinct architectures, each handling CRUD operations differently.

An endpoint is also planned to manage image processing for a new sensor or device intended for integration into the embedded system. This approach allows the backend service to run locally, making features available to others for free while enabling deployment to the cloud as a serverless function or Cloud Run application.

Lastly, the application's settings (configuration) file will be updated to make the model's agentic feature optional. If users do not wish to utilize the AI chat features, they can turn those off. The application settings file and the custom Kernel Factory class will also be extended to allow users to include either a local LLM service (free), a remote LLM model service (paid), or both models in the master kernel, based on their preferences.

Conclusion

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

It met the capstone requirements. The semi-automated IoT gardening system was responsive, with a user-friendly UI/UX design in the frontend application, and adhered to RESTful principles for the backend services and application, as learned in both the Mobile Application Development and Web Architecture courses. Additionally, it utilized various cloud services for authentication, data storage, and web hosting, many of which were covered in the Cloud Application Development course.

All business requirements from the stakeholders and personal requirements were met. While improvements could have been made to the capstone project, the current iteration fulfilled all business requirements, performed well, and appeared polished.

The semi-automated IoT gardening system independently grew plants with a short growing period, such as cat grass, from seed to maturity, without any human intervention. Refilling the water supply and amending the soil were the only interventions required if users intended to grow plants with a longer growing period.

In addition, during the course of the project, C++ and basic circuitry were learned, resulting in a functional embedded system and gardening system. Knowledge of the MQTT protocol was gained, including how to use the Adafruit IO Cloud Service and its MQTT API to publish and subscribe to sensor data in their feed database. For AI services, Microsoft's Semantic Kernel Software Development Kit (SDK) was utilized to develop a RESTful API that leveraged multiple specialized agents and agentic orchestration to provide guardrails and responses for specific topics during conversational interactions with a large language model. And while not implemented, given the scope and time constraints of the capstone project, the feasibility of running local models was explored. It was 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 by using one of Semantic Kernel's extensions to connect to the local Ollama LLM.

The gardening system, once IoT-enabled, successfully sent sensor feed data to the Adafruit IO database service, which was then retrieved and processed by the backend application whenever the frontend application requested it. The frontend application was designed with the user in mind; it was responsive, compatible across various screen sizes, and included desired features such as an AI-powered chat

messenger, a set of dashboard pages to display data from every feed type, and an authentication page for user sign-up, login, and logout.

Copies of the applications were deployed to the cloud and could be accessed from anywhere with an internet connection, but the application could also run locally.

Despite the challenges encountered during the application's implementation phase, these were successfully overcome, leading to project completion. Improvements were identified from a maintainability and reusability standpoint to ensure the project had a long, active lifecycle, with the intention of continuing to enhance and support the application long after the Capstone course was completed.

References

Schneider, J., & Smalley, I. (2025, November 17). *What is a microcontroller?*. IBM. https://www.ibm.com/think/topics/microcontroller

Arduino. *Arduino Uno R4 WIFI*. Arduino Online Shop. (n.d.). https://store-usa.arduino.cc/products/uno-r4-wifi

Arduino. (n.d.). *WiFiServer*. Arduino Docs. https://docs.arduino.cc/language-reference/en/functions/wifi/server/

Realtrek. (n.d.). *Realtek IOT/Wi-Fi MCU Solutions*. Realtek IoTWiFi MCU Solutions. https://www.amebaiot.com/cn/rtl8195-arduino-api-wifisslclient/

Ada, Lady, Cooper, J., & Cooper, T. (2024, March 08). *MQTT, Adafruit IO & You!*. Adafruit Learning System. https://learn.adafruit.com/mqtt-adafruit-io-and-you/intro-to-adafruit-mqtt

Microsoft. (2024, June 24). *Introduction to semantic kernel*. Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/overview/

Microsoft. (2025, April 16). *Understanding the kernel in Semantic Kernel*. Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/concepts/kernel?pivots=programming-language-csharp

Microsoft. (2025, May 6). *Semantic Kernel agent framework*. Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/?pivots=programming-language-csharp

Microsoft. (2025, July 21). *Semantic kernel agent orchestration*. Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-orchestration/?pivots=programming-language-csharp

Microsoft. (2025, July 21). *Handoff agent orchestration*. Handoff Agent Orchestration | Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-orchestration/handoff?pivots=programming-language-csharp