Connected Farm: Final Design Report

Dominic Bembry, Andrew Bonsted, Levi Hochstetler, Gabriel Johnson, Katie Moffitt

May 2, 2025

COSC 402 – Computer Science Senior Design
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996

dbembry@vols.utk.edu
abonsted@vols.utk.edu
lhochste@vols.utk.edu
gjohns57@vols.utk.edu
dmoffit1@vols.utk.edu

Technical Advisor: Josh Lothian

Executive Summary

Problem Statement

Farmers need a reliable, scalable, and cost-effective solution to address these challenges and improve their overall farming operations. Connected Farm to satisfy this need by providing real-time data insights, analytics, alerts, and visualizations to enhance farm efficiency and sustainability at an affordable cost.

Proposed Solution

The design of the Connected Farm system involves four aspects: sensors, networking, backend, and frontend.

Each sensor consists of an Arduino Mega connected to a temperature and humidity sensor, a soil moisture sensor, a water level sensor, and an ultrasonic sensor. A 9V battery serves as the module's power source. An ESP8266-01s Wi-Fi module transmits the collected data back to the backend server. These components can be easily acquired at a relatively low price to reduce the barrier of entry for farmers considering IoT monitoring.

The data that the sensors collect needs to be periodically transmitted to the backend server. This is handled via Wi-Fi since it is simple to set up and a well-established solution to the problem of wirelessly transmitting data. The data transmitted from the sensors then needs to be received by a server to be operated on and stored in a database. Since setting up server hardware is expensive and complicated, Connected Farm takes advantage of AWS cloud services to provide a simple, low upfront cost way for farmers to set up the system.

On the software side, the backend will be built with Node.js using the Express.js library. This backend provides API calls that allow both the sensors and the frontend to interact with the databases. After receiving a request, the backend will either place data in or get data from one of the two databases depending on the type of data. The readings from the sensors will be placed in an InfluxDB timeseries database since it offers faster selection of data over intervals of time. All other data such as configuration of sensors, and user data will be stored in an AWS Relational Database Service database.

The frontend, which presents the data collected from the sensors to users through various plots and charts, will tie the entire project together. Since farmers need to access this data on the go, the frontend will be built for mobile using React Native as well as various libraries and tools selected to create a professional, compelling, and visually appealing design.

Final Thoughts

Connected Farm offers a cost effective and simple solution to enable information-based farming with cheap, readily available sensor components, cloud hosting to keep upfront costs down, and a robust and intuitive software companion app.

Table of Contents

List of Figures

List of Tables

Problem Definition & Background

Agriculture has played and will continue to play undeniably fundamental roles in maintaining a successful society. These roles include, but are not limited to, providing a population with sustenance, supporting a country's cultural cuisine, and providing jobs for millions of people. In 2023, United States farmers contributed around $223 billion dollars to the U.S. GDP [1] with farmers producing $276.7 billion dollars' worth of cash crop receipts and $249.6 billion dollars' worth of animal and animal product receipts [2]. A major problem is introduced as global populations increase. Agriculture producers are expected to maintain or increase their output with each passing year to directly combat food availability concerns. In the U.S., the population is estimated to see a 4% growth from 332.6 million people in 2020 to 346.1 million in 2030, which is on par or greater than the entire population of the European Union and many developed countries including the United Kingdom, China, and Japan [3]. Due to the necessity of a quality food supply, all levels of society are impacted in case of a food shortage, including food producers.

To address the needs of a growing population, agricultural practices have evolved over time with the development and implementation of modern technologies. Agricultural experts have identified four major "revolutions" that each coincide with several significant technological advancements. A broad timeline for each agricultural revolution is outlined below in **Figure 1** [4]. Farmers have achieved greater efficiency and yields after each milestone period. However, this has not been the case for the environment. As the world experiences more adverse effects from climate change overall, there has been a greater cultural focus on minimizing the ecological footprint of while maintaining steady production efficiency and yield.

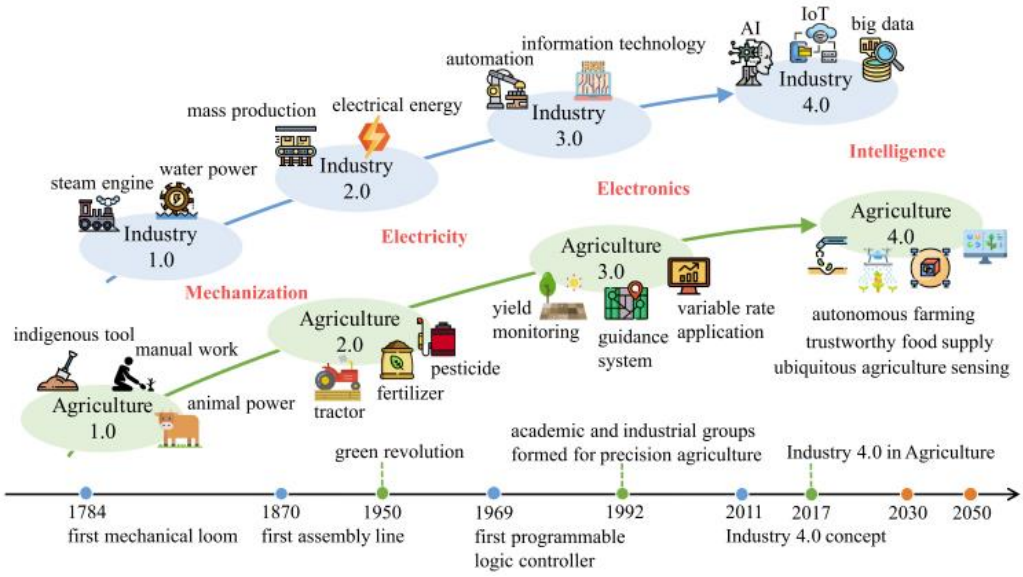

Figure 1 – Roadmap of Agricultural Revolutions Over Time (adapted from [4])

With farmers as the cornerstones of food production, they are in constant search of tools that may enhance their farm's performance while also complying with environmental regulations. The current "revolution" we live in, dubbed "Agriculture 4.0," has focused on meeting these needs by

equipping farms with reliable, real-time Internet of Things (IoT) systems coupled with software that receives the data and communicates with the devices. The IoT permits small, digital devices to communicate with one another and other remote resources via the internet. In an agricultural context, the IoT can be applied to crop data collection and recording, livestock monitoring, process controlling, holistic farm management, and many other useful functionalities. Farmers can use these "smart farms" to have more precise control over their farm's inputs and outputs and can operate some parts of their farms from remote settings. Advanced smart farm systems must involve some form of data collection through a multitude of sensors (environmental features, animal vitals, crop images, etc.) These sensors are connected to a computing module that can transmit each sensor's data to a remote server through communication to the internet (Wi-Fi, LoRa, Zigbee, etc.). After the smart farm data reaches the cloud server, more intensive computing functions can be performed on the data to provide insights, analysis, and forecasts on the farm's current performance.

Ideally, such smart farm systems and software tools should be cost effective, easy to install and configure, straightforward to use remotely and on-site, and furnished with extensive resources for proper usage. These factors play a large role in how users may wish to adopt and take advantage of the IoT backed smart systems. In addition to farmers and their use for such data, there are many other groups that may find use for this information. Such people may include policy makers, consumers, and environmentalists. The data that will be collected and the insights that will be drawn from it may very well be used to drive policy in areas such as soil conservation, erosion prevention, food safety, and even climate change [5]. This data may also be of interest to consumers who have concerns about where their products have come from and under what conditions they were grown. There are also several government agencies that might find this data useful in evaluations, such as soil conservationists, state-level agricultural departments, and agricultural commissions. Having consistent records of soil health and other farm metrics would be beneficial for these customers.

Agriculture 4.0 smart farm systems have already seen extensive work in commercial, open source, and research applications. Community oriented solutions like ThingsBoard [6] and OpenRemote [7] provide suites of IoT management, infrastructure configuration, and visualization tools for both free and subscription-based customers. Examples of larger companies that contract custom smart farm services for a specific customer include Monnit, CropTracker, and Telit. Recent literature contains multiple survey studies articles concerning the leading technologies used in smart farming and precision agriculture, as well as common practices within the field itself [4], [8], [9]. Practical implementations of smart farm systems with wireless sensor capabilities [10] AI-powered insights [11], [12] have demonstrated the feasibility and usefulness of such systems. These existing works contain complementary ideas that address and focus on serving smart farm customers' needs with appropriate tools and technologies.

Requirement Specification

The Connected Farm aims to create a scalable network of sensors and an intuitive mobile application that will allow farmers and other users to access real-time information about the state of their farm. This information would include insights on soil quality by measuring moisture levels,

temperature, and porousness with the use of sensors. We have determined several primary requirements of the Connected Farm in **Table 1** through connecting with members of the agricultural community and performing intendent research. While we believe all these initial requirements are important, we have ranked them from highest priority to lowest priority.

Table 1 - Connected Farm Requirements

1. Software interaction with smart farm backend and hardware infrastructure a. Users need to be able to visualize farm status and insights b. Software must be easy to use in the field c. Data should be pulled down from servers quickly (low latency user experience)
2. Reliable IoT hardware infrastructure a. Must be energy efficient, durable, and compact b. Must be easy to install
3. Cloud solution for data accumulation (compared to farm-hosted servers) a. Must be horizontally scalable b. Must be cost-effective and low latency
~~4. ML powered system knowledge~~ ~~a. Machine learning models for moisture prediction, growth conditions, etc.~~ b. ~~Reports on soil quality and general "farm health"~~

Daniel Owens, a farmer and soil conservationist in middle Tennessee, emphasized the importance of soil quality in agriculture and the performance of produce. Farmers must regularly conduct tests to measure the quality of their soil based on things such as temperature, infiltration rates, and mineral content. As both a farmer and member of the Tennessee Soil and Water Conservation Commission Service, his opinion was that soil monitoring would be very beneficial to farmers and would provide the necessary information to make informed decisions about daily farm maintenance such that the yield of the farm might be positively impacted. For example, additional insights from information such as soil temperature may alert a farmer to the loss of cover crops in a particular region of their farm. Daniel states that cover crops not only regulate the temperature of the soil, but that they ensure the nutrient content of the soil to increase biodiversity and allows the soil to go longer periods of time without needing to be tilled, which helps to prevent significant erosion. The Connected Farm system will prioritize collecting and displaying real-time information and key insights based on the collected data with the intent to inform farmers on the current state of their farm.

Dr. Hao Gan, an assistant professor of biosystems engineering here at the University of Tennessee, provided our team with technical advice regarding the sensors and collection of data. Dr. Gan's research is focused on improving food production using "precision agriculture", which aligns with the goals of our project [13]. Several of his publications are centered around technology and how various types of technology can be used to improve crop yield [14]. Given his research focuses and experience in precision agriculture, his feedback was crucial to the decisions made regarding the hardware specifications of our product.

When identifying the requirements for the Connected Farm, it was important to consider how the system could gather useful information with the least number of resources to keep the energy and

maintenance requirements low. This was determined based on the feedback from both Dr. Gan and Daniel Owens: it was their advice that the system be easily maintained and low-cost energy-wise due to the remote locations of the sensor system and the wide area that they may be placed in. The system must be beneficial to a farmer such that it saves them time, and any time spent maintaining the system is unnecessary labor that does not relate to their product. For these reasons, the system was designed to remain untouched for long periods of time. Dr. Gan specifically noted the benefits of solar-powered systems coupled with long-term battery banks as a power source for sensor clusters. However, we could not fully achieve this design requirement due to financial and time constraints. Regardless, our solution must demand a properly contained sensor environment made from materials that can withstand the elements.

For our software component, it was decided that there should be strong emphasis on ensuring the flow of the user interface is easy to use and intuitive. This requirement was determined by feedback given by farmers who expressed dissatisfaction with technology that is difficult to use and stated that they would prefer to not have technology if there was any learning curve. With this feedback it was determined that the mobile application must be a simple design that models other commonly used apps for navigation flow and general functionality. Our software product will be in the form of a mobile application to allow for easier engagement with the Connected Farm system while in the field. This suggestion was also brought forward by Daniel Owens as well and was commonly seen as an option in both the ThingsBoard and OpenRemote IoT systems.

According to the University of Tennessee's Institute of Agriculture Soil, Plant, and Pest Center, large farms should gather approximately twenty samples from different locations across the farm to gain comprehensive insight from their soil analysis [15]. Therefore, the Connected Farm aims to have smaller components that obtain data from smaller sites across a larger area. This project will focus on perfecting a single unit in such a way that it can easily scale with any desired number of modules up to twenty. The system should be able to accurately measure the state of the soil, communicate that data to the cloud, and then present that data in an intuitive way such that the user can gain valuable insights from it about the state of their farm in the surrounding 1000 square feet [15]. Having our product connected to the cloud allows users to only worry about setting up and using their Connected Farm system without needing their own computing resources.

In an ideal system, the hardware modules would be able to communicate data using long-range network protocols such as LoRa WAN rather than directly connecting to a remote Wi-Fi transceiver module. However, due to budget and time restraints it was advised by Dr. Gan that the Connected Farm system simply uses UTK Wi-Fi to reduce both monetary and time costs. For this reason, the hardware module will include an ESP8266 microcontroller that will connect the system to the internet via Wi-Fi.

Beyond our three main requirements for Connected Farm, we had planned in COSC 401 to implement one more product requirement relating to machine learning (ML). ML and artificial intelligence (AI) play key roles in the current Agriculture 4.0 landscape, and our team had ambitions to recreate and develop insights about farm health and plant growth conditions based on our sensor data. However, we had to remove it as a requirement from this project for time and budgetary purposes. A lack of resources forced us to use a fewer number of lower quality sensors, which only

amount to four ML features at the final stage in the project. We determined that we could not deliver a product that had reliable or desirable ML features, so we unfortunately decided to remove this requirement from the scope of this semester's work. We intend to propose future work dedicated to considering ML design and implementation given time and additional funds.

Technical Approach

Our technical approach was split up into two main project spaces, the hardware space and the software space. In the hardware space, we worked with microcontrollers and sensors to collect data. In the software space, we built a cross-platform mobile application, communicated the data through wireless networking, and built a backend interface for the sensor module and mobile app to interact with. An overview of the entire project space can be seen below in **Figure 2**.

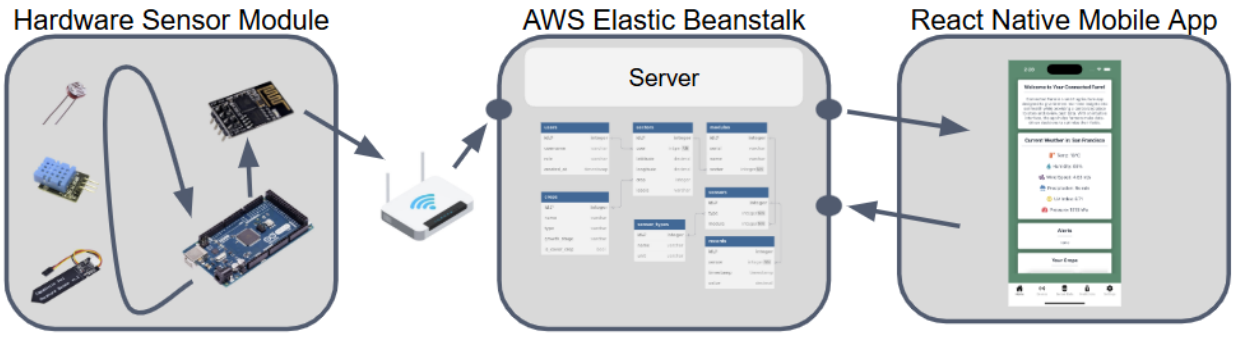

Figure 2 - Connected Farm Flow Diagram

Hardware Approach

With all our group members having a background in computer science, we wanted to undertake a partially hardware-oriented project to gain some experience in how software and hardware can connect. Seeking guidance on the hardware aspect, at the beginning of the Fall 2024 semester, our group met with Dr. Hao Gan after researching his work with the Smart Agriculture Lab in the Department of Biosystems Engineering and Soil Science at UT. We have most of our hardware components from an Arduino starter kit that Dr. Gan generously loaned to us for this semester. In that Arduino kit, there was an Arduino Uno, Arduino Mega, a breadboard, and the various sensors we would need to get started. Those sensors include a temperature and humidity sensor, an ultrasonic sensor, a water level sensor, and a soil moisture sensor.

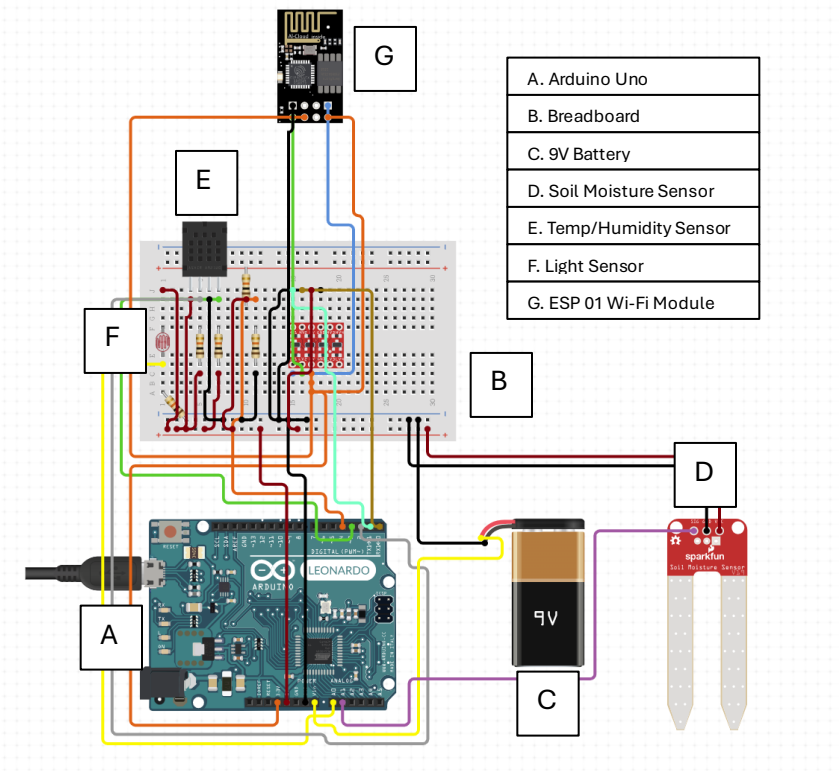

Figure 3 - Arduino Design for Connected Farm

Connecting to Wi-Fi

From the beginning of this project, we knew our sensor module had to be able to wirelessly send data to our backend. This was a pivotal part of our project as we wanted a hands-off system that didn't require any human interaction to transfer that data. After analyzing our options, we chose to use Wi-Fi over LoRaWAN and Zigbee (discussed further in Design Concepts, Evaluation, & Selection). There was a need for an additional hardware component to work with the Arduino board on collecting and sending the data. After conducting research on various hardware parts, we found the ESP8266 ESP-01 Wi-Fi module that allowed us to connect to a router and send data to our backend. The ESP-01 is a cheap, lightweight module that comes with its own firmware, but it acts like a computer, and we can program it separately from the Arduino.

Initial development for the ESP-01 consisted of utilizing the built-in AT commands as part of its firmware. These AT Commands allowed us to connect to Wi-Fi using the Serial Monitor in the Arduino IDE but allowed us to only leverage limited functionality from the ESP-01's full capabilities. Realizing we could upload our own C++ code onto the ESP-01 module and bypass the default AT interface was a breakthrough that allowed our final product to be developed.

The program for the ESP-01 and the Arduino board can be seen in **Figure 4**. For our final product, when the ESP-01 module is connected within our sensor module, being powered by the battery, it will first loop indefinitely until it connects to Wi-Fi (step 1 in **Figure 4**.). Because of issues with Wi-Fi redirecting to log-in pages (like with UT-Open and Eduroam), we used an iPhone hotspot for the development of the sensor module. Once connected to Wi-Fi, it sends an HTTP request to our backend server to register the sensor (step 2) and tell the server that this module is online. It sends

the chip ID to the server so the module can now be identified within our system and other databases with a unique identifier. After registering, it sends a message to the Arduino board that the set-up phase is complete, and it is ready to accept data from the sensors (step 3). Then, the module waits. Once the Arduino sends data from all sensors to the ESP-01 module, it accepts it (step 5) and structures it as a string in the format of a JSON object. Finally, it sends the data to the backend through the 'send_basic_report' API endpoint and the data is successfully stored in the database (step 6). The code for the ESP-01 module also has built-in checking to ensure that Wi-Fi is still connected and if it ever becomes disconnected, a message is sent to the Arduino to tell it to stop collecting data until it reconnects.

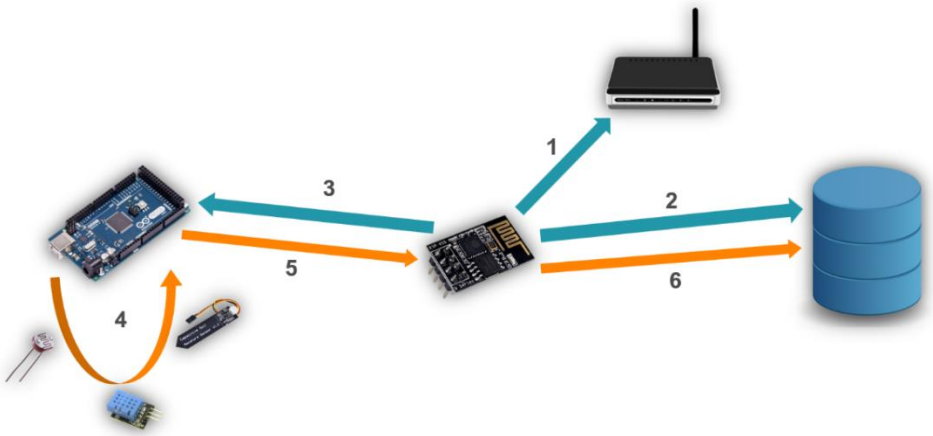

Figure 4 -Hardware Flow Diagram

Arduino Programming

Our loaned Arduino equipment from Dr. Gan had two microcontrollers: an Uno R3 and an ATmega 2560. We began the project developing for the R3 since it was smaller and provided basic capabilities such as sensor I/O, UART serial pins, and a comparable clock speed (16 MHz) to that of the 2560. Many of the original circuits and programming guides we used to establish original drafts of our final project also used an R3 module which streamlined development further.

As mentioned in the ***Connecting to Wi-Fi*** section, our first goal was to establish communications with the Arduino and the ESP-01. The quickest way to do this was using the ESP-01's default firmware and sending AT commands over the serial pins of the Arduino board. In the earliest stages of programming, Arduino was our only programmable device. This meant that all configuration for the ESP-01 chip needed to come through manual serial pin connection and manually typing the AT command strings in the Arduino IDE's Serial Monitor. We decided to configure the Arduino and ESP-01 to share the maximum allowable baud rate supported by the Serial Monitor, which was 115200 bps. Once manually configured, we programmed the Arduino to automate certain parts of the AT command interface by creating a generalized 'sendCommand' function that allowed hardcoded AT commands to be sent to the ESP-01 using the R3's dedicated TX/RX pins.

We then discovered how to overwrite the default AT command firmware on the ESP-01, so we pivoted our Arduino programming approach. Instead of sending AT commands to the ESP-01, we

established a simple client/server like architecture where the two devices could send predetermined structured messages to each other independently of one another. This model is characterized in **Figure 4**.

The Arduino now focused solely on collecting data and forwarding the values to ESP-01. This allows for the ESP-01 to separately handle the logic for posting data to the backend server. Due to the new separation of devices, we decided on a loosely structured message passing protocol between the Arduino and ESP-01. This protocol was simple and served as a basic demonstration of our system's capabilities but would not serve as a production-ready protocol due to the protocol's lack of integrity and security. From the Arduino to the ESP-01, we defined the following steps:

1. *Data collection*: The Arduino issues read calls to all its connected sensors and stores the raw values in corresponding global variables.
2. *Data transmission*: The Arduino sends the value of each sensor variable to the ESP-01 with the name of the read sensor and its value. The names of these values were agreed upon with the backend team for easier parsing. The name of the sensor reading is separated from its value with a single space. Each name/value pair is terminated with a single newline ('\n') character.

   After all the sensor names/values have been sent to the ESP-01, the Arduino sends a single newline terminated "DONE" message, signaling the end of Arduino transmission. An example transmission is shown below:

   ```
   temperature 70.6
   humidity 54.0
   soil_moisture 340
   light_level 700
   DONE
   ```

3. *Low power idling*: After the "DONE" message is sent, the ESP-01 takes care of the transmission to the backend. Since the Arduino will wait a specified number of seconds before reading from the sensor pins again, we use a simple Low Power library that takes advantage of AVR Arduino board's "idle" capabilities, which . We idle for the remainder of the cycle before the next read.

This entire process is represented by step 4 in **Figure 4.** If at any point the ESP-01 became disconnected from our Wi-Fi hotspot, the Arduino was programmed to listen for a single line containing a single ASCII character, zero ('0'). This prevented the Arduino from sending arbitrary sensor data to the ESP-01 without the latter being prepared for it. After the ESP-01 reconnected to Wi-Fi, it would send a single ASCII character, one ('1'). The Arduino saw this message and could begin sending data to the ESP-01 again.

After developing the R3, we realized debug messages from the Arduino were also sent along the same TX/RX pins that were connected to the ESP-01. Because of this, we would send junk strings to our backend. We decided to transition our development board to the ATmega 2560 in our possession due to its multiple serial pin pairs. Doing this separated debug and business logic channels.

Enclosure

We wanted to house as many of the hardware components inside of a single enclosure as we could. We found a decently sized hard-plastic container that was large enough to comfortably fit our prototype boards, Arduino board, power supply, and sensors. This container had two strong clips to seal the unit, and about five inches of vertical depth to fit our components. However, getting high-fidelity sensor readings and Wi-Fi signal meant pushing some of the hardware components outside of the container. For lack of resources and simplicity, we made modifications to the container that included several openings. We sketched these plans out on paper and were able to use tools in the Kao ICS to make holes and slots. **Figure 5** shows our dimensional plans for the enclosure. We drilled two holes on the left side of the unit: one for allowing the Arduino USB power cord into the unit to power and program the Arduino (dubbed the Developer Hole), and one for allowing the soil moisture sensor to leave the module. We also drilled several holes on the right side of the container for the light and DHT sensor to be outside. We also Dremelled out a 2x1 inch slot for the Wi-Fi chip (the slot is dubbed the Wi-Fi porch). We made this slot originally planning to attach the other sensors to it but realized later we could make smaller protrusions higher up on the right side of the unit. **Figure 6** shows the results of a single Connected Farm hardware enclosure.

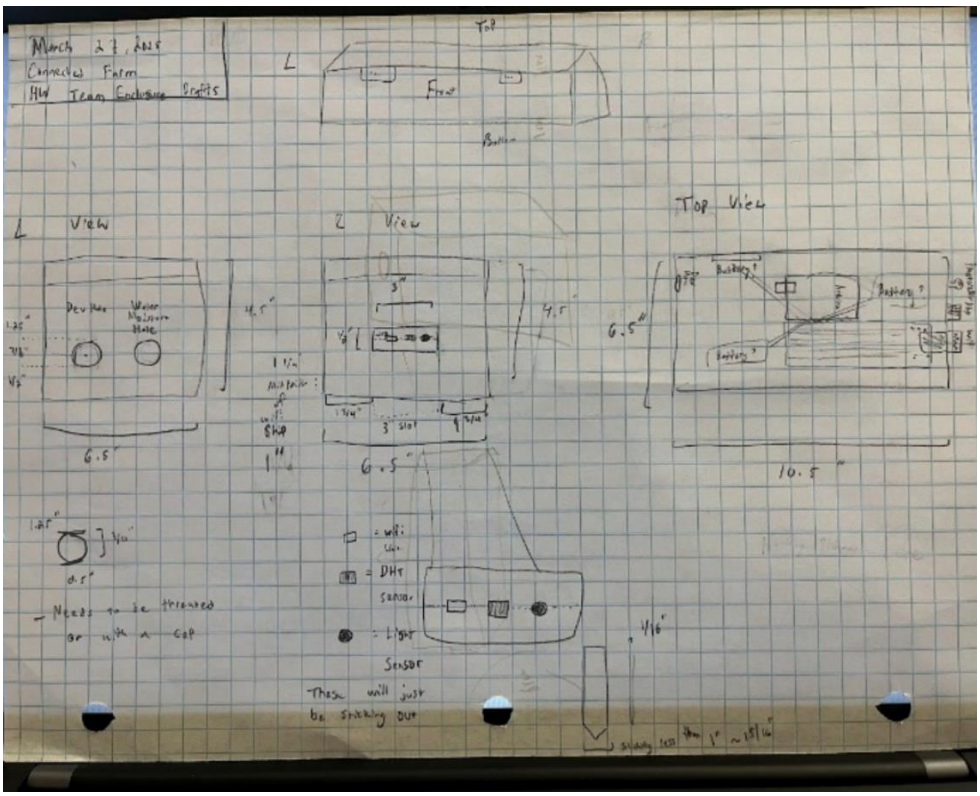

Figure 5 - Holes and Slots for Enclosure

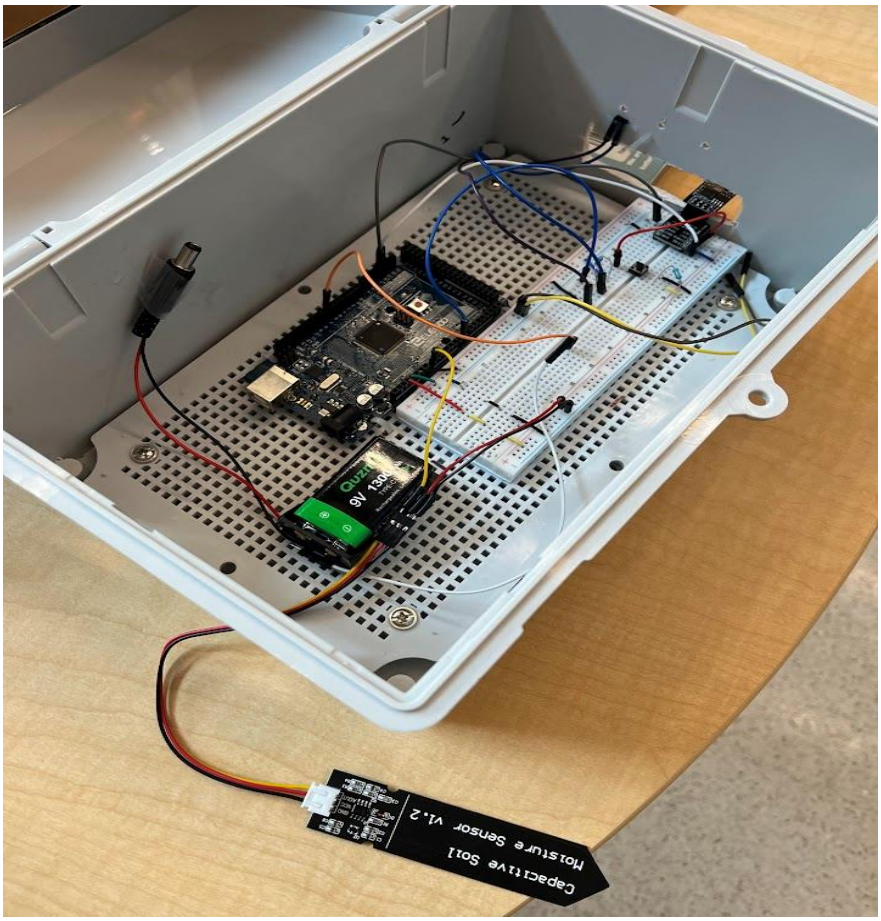

Figure 6 - Final Hardware Enclosure

Challenges and Limitations

Being solely computer science majors both benefitted and slowed the development of this sensor module. We ran into hardware and software issues at every phase of development that challenged the way we thought about our product and allowed us to grow into more complete programmers, having now dealt with implementing an, albeit rudimentary, hardware system.

As previously mentioned, we initially learned about the ESP-01 module and developed it through the AT commands. We first started by following a useful Instructables guide [16] that outlined the basic process of interfacing with the ESP8266 ESP-01s Wi-Fi module. After completing the tutorial, we had a simple and repeatable way to interface with the ESP module through modem AT commands, which were used to reconfigure the module's default baud rate, connect to a public Wi-Fi access point (*ut-open)*, and send ASCII data to and from a local server. We then learned how to automate the AT command initialization process through an Arduino program by printing AT commands directly to the Wi-Fi module over its serial pins. This enabled reusable blocks of code to get our module connected and sending data without the need for terminal interfacing.

However, while AT commands provide decent device control for initial prototyping, they are not robust functionally nor syntactically for the entire Connected Farm system. After spending a week or two on the AT commands, we pivoted into researching and utilizing dedicated ESP8266 Wi-Fi libraries. These libraries allowed us to directly program the ESP-01 module using C++. Previously, we programmed the Arduino itself in C++ to send AT commands to the ESP-01 module to perform actions. Decoupling this process lets the ESP-01 module receive dedicated programming that operates independently of the Arduino, which additionally allows more Arduino flash memory to hold more sensor related code. We planned to utilize a library that enables the Arduino to execute serial communication through two of its pulse-width modulation pins. That way, we can program the Arduino and ESP module using a client-server paradigm: The Arduino can act as a "client" that collects sensor data and send serialized messages to the ESP-01 "server." The ESP-01 can then use its dedicated code to listen to incoming sensor data and forward it to our AWS backend.

Once we were able to program the ESP-01 and the Arduino independently, we encountered another problem with how we would communicate between the two. The initial Instructables guide that we followed had us connect the Arduino's TX and RX pins to the respective TX and RX pins on the ESP-01. Though we didn't know at the time, it was because we were coding the ESP-01 using the Arduino Uno as a pipe to get the program uploaded from our computers to the ESP-01. Finding the SoftwareSerial library showed us that the wires were supposed to be inverted such that the TX pin of the Arduino was connected to the RX pin of the ESP-01 and vice versa. We had varied success with the SoftwareSerial library. We were able to send strings from ESP-01 to the Arduino at very small baud rates (slow communication times), but our strings would be output as garbage characters. After multiple debugging attempts, it was clear that this was a pressure point in our development as we wanted to have clear and consistent communication between the two modules. We switched to trying to communicate through the built-in UART pins. After finding a small GitHub repo doing exactly what we were trying to accomplish, we were able to set up communication between the two modules. The hardware UART allowed us to create a "Serial" object at high baud rates and send strings from either module and receive them clearly and reliably.

While solving the communication issue, we realized a disadvantage to using the Arduino Uno. It only had one pair of UART TX/RX pins. This meant we had to unplug the ESP-01 module every time we wanted to upload code to the Arduino. Upon further research, this is because the serial of the USB and the serial of the hardware TX/RX pins are physically connected. An additional issue was separating the debugging statements we were sending to the serial monitor and the serial buffers that the data was being transmitted with. To alleviate the challenges that further development would bring using the Arduino Uno, we switched the Arduino board we were using to the ATmega 2560 board which came with multiple hardware TX/RX pins. This new board was larger and allowed us to streamline the debugging process and isolate it from data communication from the board to the ESP-01.

Once we had a prototype sensor module created, we realized the lack of user interface with the module itself. The only way to adjust the module is through the software that we upload to the ESP-01 and the Arduino board. This means that aspects like the Wi-Fi and data collection period would be hardcoded in and wouldn't be able to be changed by the user. Ideally, the user would be able to choose what Wi-Fi network as well as how often data gets collected from within the app. This would require back-and-forth communication with the sensor module and the backend server but currently, communication only goes one way, from the sensor module to the backend. This sort of

user interaction has become out-of-scope but is an essential next step if this project were to continue.

As we finalized the project, trying to set up all the hardware in the enclosure, we encountered some unforeseen design flaws with the Connected Farm hardware system. Firstly, the enclosure was originally going to be fully enclosed from the elements to best align with the goals of where the product would be placed. Unfortunately, we did not purchase nor have access to better electrical cabling, so we were constrained to prototyping wires and jumper wires. This forced us to crudely expose electrical components to the elements and remove the waterproofing of the enclosure. Secondly, we realized there were not many ideal locations to secure the battery box, which has an on/off switch on its otherwise flat backside. The front side of the box holds the battery itself. Both considerations forced us to plan to either *a)* 3D print a separate enclosure for the 9V battery with an external switch, *b)* create a small slot in the front or back wall of the enclosure to make room for the power switch, or *c)* find a battery holder that more suited our needs. In the end, we purchased another battery holder that allowed easy access to changing the batteries out. Finally, the prototyping wires made some of the final design messy and harder to manage. In a more realized project, we could have achieved better design, appearances, and performance with higher quality sensors that had dedicated weather proofing, stronger cables, and dedicated connectors. Additionally, we would have chosen a much better power source than a single 9V battery. Solar power would be the ideal method of power for a sensor unit such as Connected Farm, but our limited personnel experience with power electronics and team financial resources precluded advanced power circuity.

Software Approach

On the software side of our project, we had three primary project spaces that we are working in, the frontend, the backend, and networking. The frontend is a mobile application that provides three main functions: device management, data visualization, and machine learning predictions. We decided to remove the machine learning predictions from our app functionality for now due to time and budget constraints. The backend has three main sections: a relational database, a time series database, and the hosting service. Finally, the networking consists of communication infrastructure between the module and database. **Figure 2** models the high-level interaction between the project spaces.

Frontend

As mentioned, the frontend is a mobile application that allows users to interact with their smart farm. An important aspect of a smart farm is being able to manage your IoT devices. We want our users to avoid having to physically manage the in-field devices unless they are doing the initial setup or replacing the power supply. To achieve this desired goal, we wanted to have our application be able to send signals to our Arduinos that represent various commands. The user needs to have the flexibility to determine which Arduinos and sensors are active, at what interval they send and record data, and find where they are physically located on their farm. However, unfortunately, we never were able to get this functionality working. The learning curve for the hardware portion of

our projected turned out to be very steep, so we never got a chance to determine how the Arduino would receive and process posts from the backend. Users can still add new modules into app, but it requires them to physically set up the module and have it send data into our database to be pulled.

The next important part of the frontend is data visualization. After the data has been collected and stored into the database, we need to provide the user with a meaningful way to observe the collected data. This takes form as a collection of time series graphs that display the information each sensor collects. Based on the current sensors available to us, we have time series graphs that display soil moisture data, temperature data, and humidity data.

Finally, we had planned to train and deploy an ML algorithm to provide users with predictions based on the data collected by the sensors as well as recommended actions to take based on the data collected and history. We wanted the user to observe predictions such as what the estimated soil moisture will be for the next seven days, with each day out being less accurate than the previous. These predictions would allow for speculative farm resource planning. But as we mentioned previously, we have decided to remove the machine learning aspect from our project's scope. A fully realized project would still have this feature as it is one of our selling points, so we have decided to keep our idea of its functionality in our report. **Figure 7** shows wireframes of two pages for the frontend.

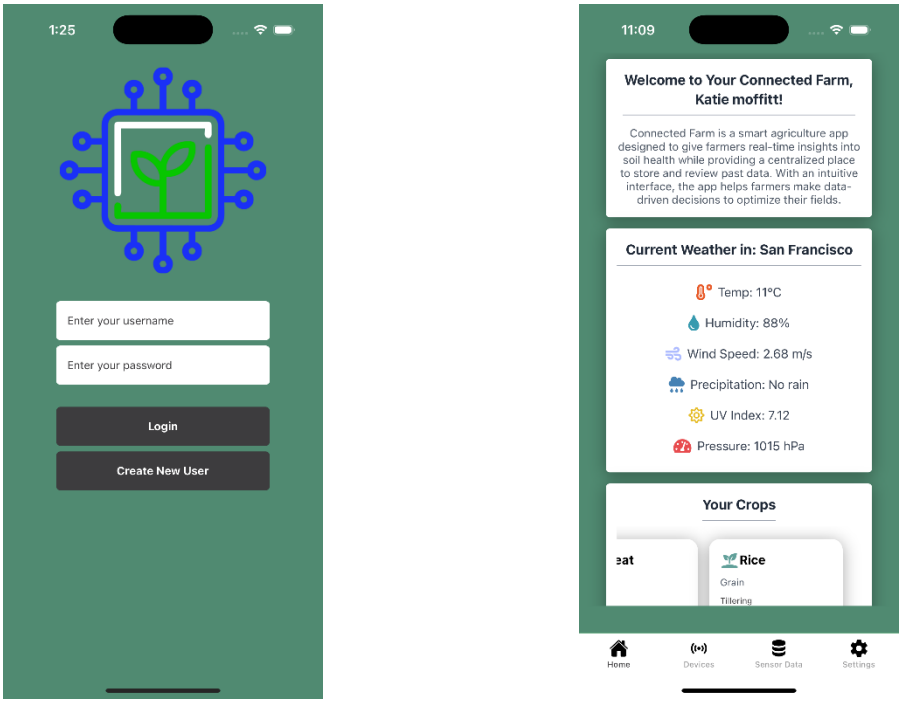

Figure 7 – Final Frontend Layout

One of the first major decisions that we had to make for the frontend was how we wanted to navigate between the different phone screens on our app. Our two options were using Expo Router and React Navigation. While these two options can be used together, as Expo Router is built off the core components of React Navigation, when you begin to nest different routing methods in a project you run the risk of over complicating the project's foundation which can lead to unexpected

compatibility issues. We noticed this issue when the nested navigation methods began to override the written styles for our different pages. Being able to properly route between screens is crucial for a functional mobile application, so we wanted to make sure that we handled this problem as early as possible. In the end, we decided to go with the Expo Router approach for two primary reasons. The first reason is that the project was created and is tested as an Expo project, so the initial setup already included basic Expo Router setup. The second reason is that Expo Router is supposed to be more efficient than React Navigation since it optimizes the navigation process with its file-based routing extension of React Navigation.

Developing easy-to-use and intuitive screens was a major focus for the frontend team. As we mocked up each screen, we steadily worked to improve them as we continued to meet with the team and receive feedback. We anticipated revisions and made it a priority to create components that can easily be reused or completely repurposed to help streamline development.

The devices page allows the user to add a sensor module by filling out a form. There the user can specify where the sensor module is located and provide a name for the module. To make this page more engaging, we added the ability to allow the user to add labels to the modules so that they may add any relevant information they might wish. This information could be things such as the crops that the module is collecting data on, the kind of cover crops in the area it is reporting from, or anything else that the user feels is relevant.

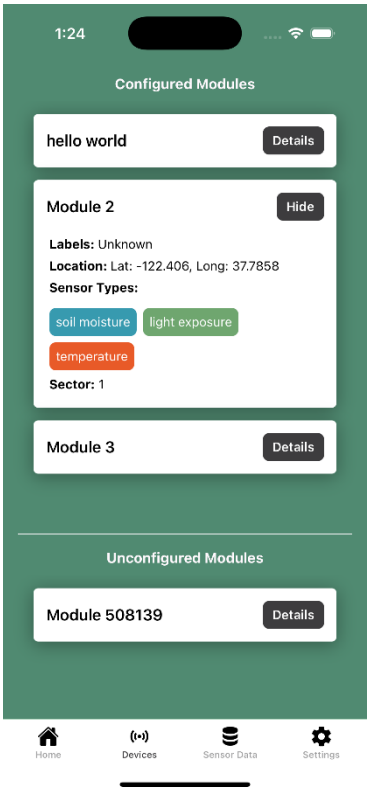

Figure 8 – Current Devices Screen Layout

The sensor data screen has undergone several revisions and will continue to receive updates based on user feedback. The initial screen was a list of sensor modules that the user can then select and

navigate to a separate page that shows the sensor data reported by that module. After receiving feedback from the other teams and much discussion, we decided to develop this page to instead list sections of the farm as defined by the user. These sections may have several sensor modules, but they must have at least one. The user will be able to select a section and view the data that has been collected by the module(s) within that defined area. As previously stated, we prioritized making reusable components and because of this we were able to make this new screen reusing components that can be seen on other screens.

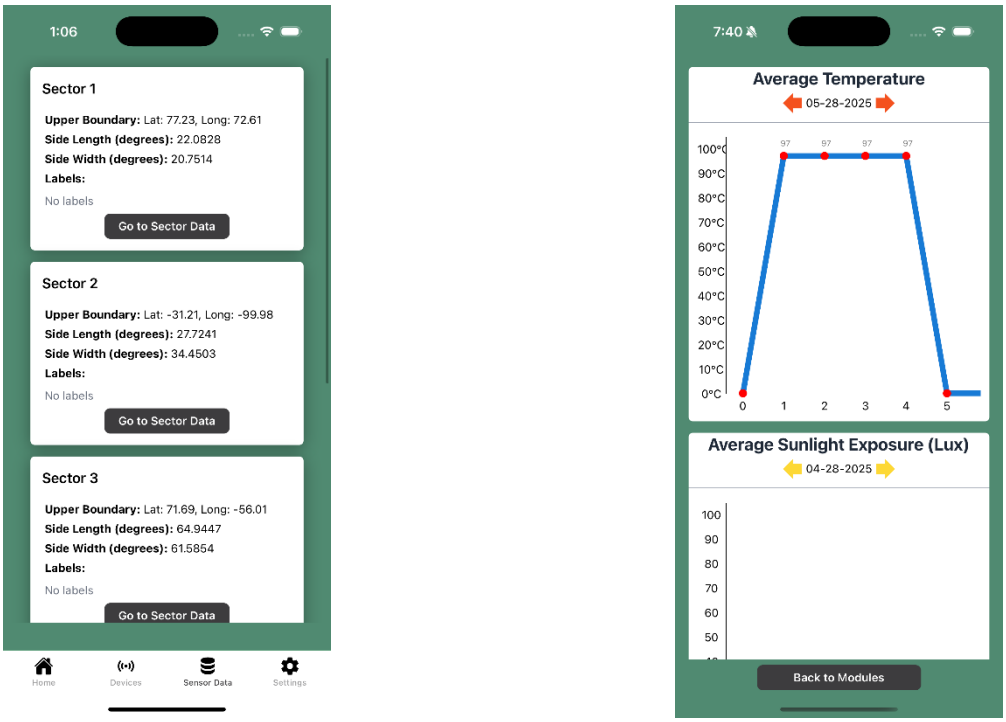

Figure 9 – Final Sensor Data Screen Layout

Our next major decision was to remove the Predictions screen as one of our core screens. When we made the difficult decision to axe the machine learning aspect of our project, the Predictions screen unfortunately became obsolete. Fortunately, we had not done much work with this screen yet, so we did not lose a lot of labor because of that choice.

The final stretch of our project involved incorporating the many API endpoints from the backend to help our frontend become more dynamic. Now the frontend supports creating and authorizing new users for their new smart farm, creating and updating sector information, displaying and configuring the modules stored in the backend, and displaying the data collected by each of the sensors on each of the modules.

Users can create an account on the login screen. They can define their name, username, password and email. The frontend then sends a request to the backend to store the user's information for later authentication. Once the user has created their account, they can login using their username and password. This will generate a token that is then stored securely by the frontend for the duration of

their login session. All endpoints require this token to be appended to the API call, and this is handled by a service created by the frontend to handle all authorization of API calls.

The sectors screen lists each defined area that the user has created in the settings. Sectors are simply four-sided areas within a farm. The location of the top right corner is specified by the user during the sector creation. The user can add labels that describe the area so that they can give more context as to what it is. A sector may contain any number of modules, and this is reflected on the modules page as well. The sectors screen routes the user to the sensor data screen so that they may view the data being reported by the modules within the sector. The user may also update the sector from the settings screen to expand or add new labels to the defined area. The modules page will simply recalculate the area and assign the appropriate modules to the sector should the area be redefined.

The final form of our module screen involves two separate sections: the unconfigured modules list and the configured modules list. The unconfigured modules are any modules that have sent data into our database but have not yet been assigned a sector and location. From our devices screen, the user can update the location, assign it a sector, and give the new module a custom name if desired. Once the unconfigured module has been updated, it will now appear on the configured modules list upon the next refresh of the screen. Images of the devices screen can be seen in Figure 10 below.

For our sensor data screen, we now have four graphs that display the temperature data, humidity data, soil moisture data, and light exposure data. Temperature and light exposure are line graphs to see the change in both of those values over time, while humidity and soil moisture are represented by bar charts since we are displaying the percentage of each. The graphs display data daily, so you can see the 24-hour view and the values at each of the hours. The graphing capabilities were achieved via a package called “react-native-gifted-charts" so we did not have to make any of these components ourselves fortunately. Figure 9 shows the sensor data screen.

Backend

The backend is built on ExpressJS and is hosted with AWS Elastic Beanstalk. Data is stored in an AWS RDS database which integrates well with Elastic Beanstalk. Figure 10 shows the schema of the database. Each module can have an arbitrary number of sensors associated with it which each have a type of data that they record. The database also keeps track of sectors containing modules to aid in spatial grouping of data.

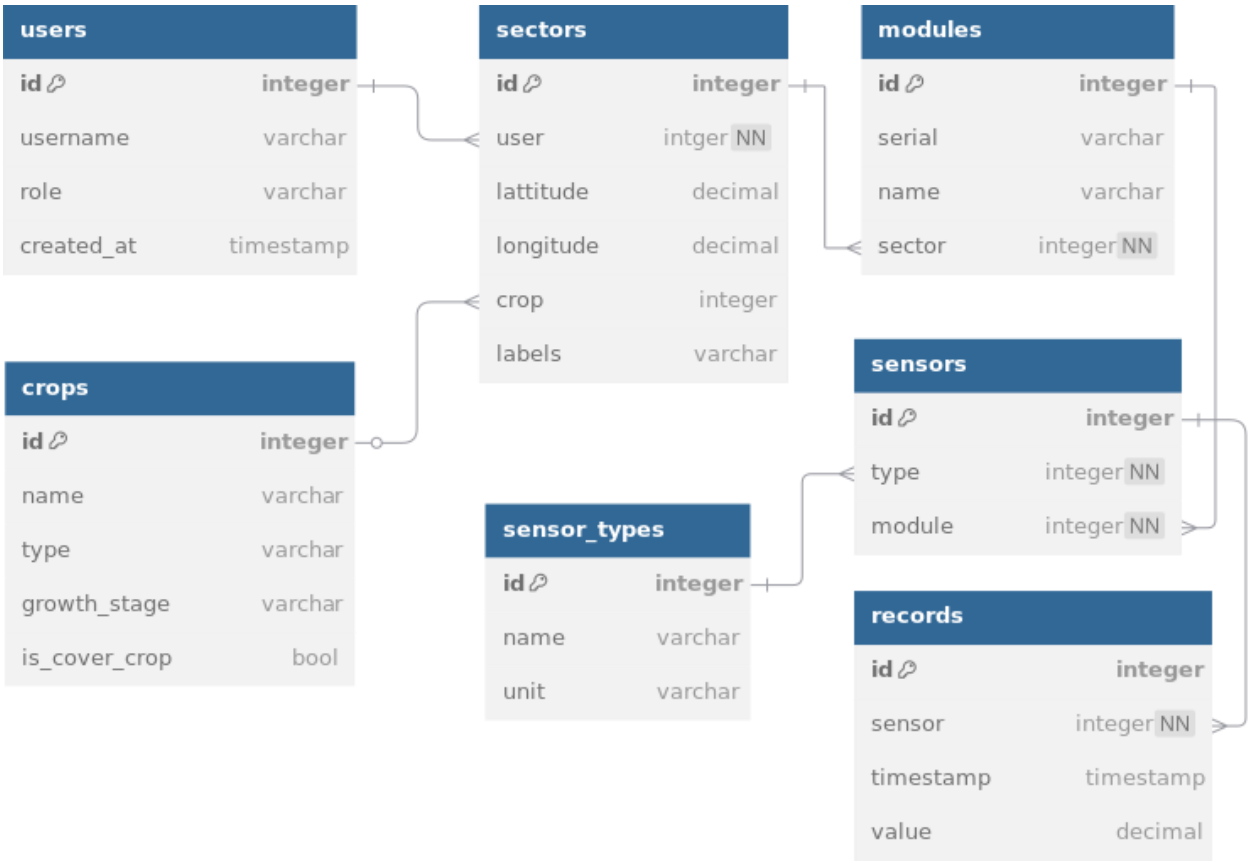

Figure 10 - Database Diagram

The back end has a variety of API calls to interact with the data **Table 2**:

Table 2 – Draft of Backend API Calls

API call	Request type	Description
register_module	POST	Register a module based on the module's serial number. An ID will be generated for the sensor and the sensor will be marked as inactive in the database so that it can be activated by the frontend.
configure_module	PUT	Activate a registered module by providing a position and a sector as well as other metadata. This information will be used to update the entry in the RDS database and ensure that the new sensor is authorized by the user. This can also be used to update a already configured module.
remove_module	DELETE	Removes a module along with its associated sensors and records.

get_non_configured_modules	GET	Returns a list of all of the modules which have been registered but not configured.
get_modules	GET	Get a list of fully configured modules.
get_modules_by_sector	GET	Get all of the modules within a specified sector.
get_sensors	GET	Get a list of sensors associated with a module.
get_records	GET	Get all the records for a module and sensor type ordered by timestamp.
get_records_by_date	GET	Get the records associated with a module and sensor type between two times sorted by timestamp.
send_basic_report	POST	Receives a report from a module with the default sensor types. If the sensors associated with those types do not exist for the module in the database new entries are created for them.
define_sector	POST	Create a new sector with a specified region and crop.
udpate_sector	PUT	Modify a sector with new values for crops and region.
remove_sector	DELETE	Remove a sector.
get_all_sectors	GET	Get all sectors.
add_crop	POST	Add a crop with a name, growth stage, and whether it is a cover crop.
get_all_crops	GET	Get a list of the crop types in the database.
get_crop	GET	Get a crop by its id.
update_crop	PUT	Modify a crop.
delete_crop	DELETE	Remove a crop from the database.
add_user	POST	Register a new user with a username and password.
get_user	GET	Get the data associated with the logged in user.
update_user	PUT	Modify a user's data.
remove_user	DELETE	Remove a user.
login	POST	Authenticate the provided username and password and return a token that the app includes with any API call for authentication.

The implementation of the endpoints facing towards the sensor modules needed to be changed in order to prevent the need of the hardware persisting additional ID information since storing state that is maintained through powering off is more difficult on the sensor side. One example of these changes is that we are now going to handle new sensors by setting them to send a request to register themselves when they are turned on, which will enter them into the database in an inactive state. The front end can view these inactive sensors and activate them while providing position and other sensor configuration information.

Authentication

Authentication is handled by storing usernames and password hashes for each user. When the user first logs in, the back end generates a JavaScript Web Token with a specified timeout. Any further API calls that the front-end uses will include the token in the request header which is checked at each endpoint (except those facing the hardware). Once the token times out, the backend will send an error for any API call letting the front end know that the user needs to be logged in again.

HTTPS

Originally, we wanted to use HTTPS for our requests for security, but we realized that in order to generate an SSL certificate for the back end we needed to own its domain name. We used Amazon Route 53 to register the domain: connectedfarmutk.com so that we could generate an SSL certificate using Amazon Certificate Manager.

Networking

Even though networking will be the smallest part of our software work, it will be vital to a functioning product. The wireless networking will allow our Arduino to communicate to the backend of our system so that we can collect the data that the sensors are gathering. The wireless network protocol that we will use is Wi-Fi since it is easy to access and will make initial testing easier. Wi-Fi will be used by the Arduino to communicate between two different points: the device manager and the database. We need the Arduino and the device manager to communicate so that we can send the signals that we defined earlier in this section, and we the Arduino to communicate with the backend so that we can receive the data that the sensors are collecting. If we were going to scale this project to an entire actual farm, then we would need to use a networking protocol like LoRaWAN since it has a much greater range outdoors, 15 kilometers compared to Wi-Fi's 450 meters.

Design Concepts, Evaluation & Selection

With our overall technical approach in mind, we had to make crucial design decisions when it came to the software part of our project. We had to choose the frontend framework that we would develop our mobile application with, what hosting platform to stand up our project with, and what networking protocol we'd use to communicate from the sensors to our hosting platform. To assist, for each decision, we created a weighted decision matrix which analyzed our options along a few key factors.

Frontend Framework

Choosing the frontend framework for our app is an integral part of the design process for this project. Our group knew that we'd want to create a mobile app for farmers since there are existing web applications that accomplish similar things. Looking into our options, the main frontend frameworks to choose from were React Native, Flutter, and Swift UI. The five factors we analyzed the frameworks by included performance, productivity (learning curve for developers, ease of use),

developer community (documentation, community forums), flexibility (existing libraries, integration), and whether it was cross-platform.

Swift UI was eliminated from our decision-making process early on, as it is not cross-platform. Additionally, it uses Swift, a programming language that is exclusively used to develop applications for iOS devices, which is why it received a grade of 5 in terms of productivity. Our group is unfamiliar with Swift so there would be a steep learning curve when we could be using that time to develop the application. Similarly, although Flutter allows cross-platform development, it uses the programming language Dart, which we have no experience with.

With both performance and flexibility being relatively even between React Native and Flutter, our decision came down to the developer community and productivity. Our group has had previous project experience with JavaScript, which is used with React Native. Coupling that with the years of community forum development that has occurred since 2015, React Native was chosen to be the framework for us to develop our mobile application in.

Table 3 – Frontend Decision Matrix

		Options					
Factor	**Weighting**	**React Native**		**Flutter**		**Swift UI**	
Performance	10	8	**80**	9	**90**	9	**90**
Productivity	8	10	**80**	5	**40**	5	**40**
Developer Community	6	10	**60**	8	**48**	6	**36**
Flexibility	7	8	**56**	9	**63**	6	**42**
Cross Platform	10	10	**100**	10	**100**	0	**0**
			376		**341**		**208**

Hosting Platform

When choosing between AWS Elastic Beanstalk and Azure App Service, we looked at the performance, security, cost, documentation, and scalability. Both choices are nearly equal across the board, especially for our specific use case. In terms of performance and security, for what our project would be using from each service, they offer the same resources. AWS Elastic Beanstalk offers AWS Identity and Access Management (IAM) to manage the users accessing and using our project's data and resources. Azure App Services had Azure Active Directory and app authentication. The main difference between the two in terms of security is the hands-on, manual set-up of AWS Elastic Beanstalk. We must manually set up the data encryption method with SSL [17]. At the same time, we are able utilize this manual nature to our advantage by fully customizing the VPC [18]. This allows us to have full control over our hosting platform rather than having features like data encryption built-it by default like Azure [19].

One of the main factors in this decision was the free tier of the two platforms. Azure App Service's free tier offers shared computing cores of up to 60 CPU minutes per day with 1 GB of RAM and storage [20]. With AWS Elastic Beanstalk, we get 750 hours per month of one of their micro instances and database usage as well as 5 GB of storage [21]. Because of its extensive free tier, it was a clear decision to use AWS Elastic Beanstalk as our hosting platform.

Table 4 – Hosting Platform Decision Matrix

		Options			
Factor	**Weighting**	**AWS Elastic Beanstalk**		**Azure App Service**	
Performance	10	9	**90**	8	**80**
Security	8	9	**72**	9	**72**
Cost/Free Tier	9	10	**90**	8	**72**
Documentation	6	8	**48**	8	**48**
Scalability	4	7	**28**	8	**32**
			328		**304**

Networking Protocol

When comparing Wi-Fi, Zigbee, and LoRaWAN as networking protocols for our smart farm system, we must look at their data transfer rate, power consumption, communication range, and cost. Wi-Fi can transfer data faster than Zigbee and LoRaWAN, but because of this faster bandwidth, it requires more power [22]. Zigbee is more energy efficient than Wi-Fi is. In terms of communication distance, while outdoors, both Zigbee and Wi-Fi can communicate up to 100 meters away. LoRaWAN can communicate up to 10 kilometers with a slower data rate, which makes it a good choice for real-life implementations [23]. Specific costs for each part can vary between $15-$30.

Taking all these factors into consideration, our group chose to use a Wi-Fi module to communicate data to our database. There was not a clear "best" option when evaluating based on our weighted decision matrix. Because our team has software-focused backgrounds, we are not trying to reinvent an optimized smart farm system. Wi-Fi provides us with a fast transfer rate over a medium distance that will allow us to retrieve real raw data from plants.

Table 5 - Networking Protocol Decision Matrix

		Options					
Factor	**Weighting**	**Wi-Fi**		**LoRaWAN**		**Zigbee**	
Data Transfer Rate	8	10	**80**	5	**40**	6	**48**
Power Consumption	7	6	**42**	9	**63**	8	**56**
Distance	9	8	**72**	10	**90**	9	**81**
Cost	9	8	**72**	7	**63**	9	**81**
			266		**256**		**266**

Deliverables

In the end, we developed our complete project that meets the criteria listed below. These goals have changed in scope and implementation as we further created our hardware sensor module and the mobile application.

1 - Mobile interface allows users to view sensor data and manipulate sensors

The mobile interface serves as the primary user interaction point for the Connected Farm system. It provides a user-friendly dashboard to visualize sensor data including soil moisture, temperature, and humidity.

2 - Backend API providing the required API calls listed in the software approach section

The backend API acts as the bridge between the mobile interface and the sensor network. It handles data transmission, storage, and retrieval. Key functionalities include adding and editing sensors, receiving sensor data, and providing data to the mobile interface.

3 - Sensor code and design information

The sensor code and design information detail the hardware and software components used in the sensor modules. This includes schematics for the sensor systems' hardware as well as source code allowing the sensors to integrate with the backend server. This information allows adoptees to easily set up their own sensors without needing to do additional research on that front.

4 - Documentation on cloning the repo and setting up the AWS services required to host the product

The deployment documentation guides users through the process of setting up the Connected Farm system, from cloning the repository to configuring AWS services. It covers steps such as deploying the backend API, configuring the database, and deploying the mobile application. This documentation is essential for both allowing adoptees to easily set up their Connected Farm system.

5 - Information about the sample smart farm developed over the semester including images of the interface and sensors

The sample smart farm showcases a real-world implementation of the Connected Farm system. It includes images of the sensor modules, the mobile interface, and visualizations of collected data. This provides a tangible example of the system's capabilities and potential applications.

Project Management

For our project to be completed in a timely manner, we established a general timeline and team roles to ensure that each aspect of our project would be equally developed.

Team Assignments

Although each team member did not have a designated title (e.g. Developer, Researcher), we divided our efforts into the five distinct aspects of our projects so that each has two to three people in charge of developing them. Our project was split into four parts: frontend, backend, data management, and sensors. Machine learning was a previous team that we removed from our scope. During one of our initial meetings, each team member mentioned which parts interested them the most and then we split into sub-teams for each section accordingly. Further collaboration shifted team members onto different teams as we worked on the project in the spring. The full breakdown can be found below in **Table 6**.

Table 6 - Team Roles for Project Development

Team Member	**Frontend**	**Backend**	**Data Management**	**Hardware (Sensors)**	**~~Machine Learning~~**
Andrew			**X**	**X**	**~~X~~**
Levi			**X**	**X**	**~~X~~**
Dominic	**X**	**X**	**X**		
Gabe	**X**	**X**	**X**		
Katie	**X**		**X**	**X**	

Gabe primarily worked with the backend server and managing our databases and other teammates worked to develop communications and data transmission with the backend. Andrew, Levi, and Katie built the prototype smart farm system with the Arduino and various sensors with Katie pivoting to the frontend team through the midpoint of the spring semester. Dominic and Katie led the frontend development.

Projected Timeline

There were five main deadlines for the spring semester as detailed in **Table 7**. It was vital to have substantial progress in between each deadline for the overall development of the application and hardware system and to ensure we had a functional and complete project at the end of the semester.

Table 7 - Timeline for Development in the Spring

Item	Deadline	Milestone Progress
Elastic Beanstalk instance created	1/31/25	On Time
Wired Arduino/sensor prototype	2/1/25	Partially Complete
Static frontend	2/12/25	Partially Complete
EARLY PROTOTYPE	**2/12/25**	***Slightly behind***
Wireless Arduino/sensor prototype	2/19/25	On Time
API gateways	2/26/25	On Time
Frontend developed	3/4/25	Mostly Complete
MINIMUM VIABLE PRODUCT (MVP)	**3/5/25**	***Relatively On-Time***
Machine learning model	*N/A*	*Removed from scope*
Finalize HW enclosure	4/2/25	Partially Complete
Refine application	4/2/25	On Time
PRE-RELEASE	**4/2/25**	***Relatively On-Time***
Testing, documenting, and further refinement	4/30/25	On Time
PRODUCT RELEASE	**4/30/25**	**Complete**

We met a majority of our first milestone (2/12/2025) goals, but we had some struggles on the hardware end. Our frontend also has a limited number of screens and features owing to the number of dependencies required for mobile app development. Our backend team has reached a strong foundation of the AWS system and has already begun working on the next milestone's API gateway goals. Overall, we are not far behind our plans for the fall but need to ensure we catch up to our expectations in time for the MVP's completion.

As of 3/6/2025, our minimum viable product has reached a solid point while being only slightly behind in the overall scheme. We can send data from our prototype sensor module to the backend database. The frontend is still working on connecting to making plots around these legitimate data points, but the endpoints exist and have shown to work. The hardware team has still to encapsulate our rudimentary prototype in the enclosure. Our independent component development has been going well, but as we move to the final sprints, our focus in tying the pieces together is paramount. Documentation will also begin to ensure our process is understandable.

After the pre-release, we continued refining our application for the beginning of April when the pre-release was due (4/2/25). As mentioned in the Requirement Specification section, we concluded there was neither enough time nor meaningful features to train machine learning models that

improved the quality of our entire Connected Farm. We used this time instead to work with our hardware enclosure and finalize the interior layout of electrical components.

After our pre-release, we put the finishing touches on our product. We ensured the data pipeline from hardware to the backend to the frontend worked as expected. The hardware team put the last sensors on the module and set up a new battery holder that installed more easily within the enclosure. The frontend team cooperated with the backend team to ensure stable communication between data retrieval endpoints. The frontend team also finalized the apps primary features, including the support for user account creation and authentication.

Budget

The final cost of our project is $126.24. The budget for our project was relatively small, with it consisting of some additional hardware pieces for powering, connecting to Wi-Fi, and cases to secure our Arduino and battery since we received several sensors and Arduinos from Dr. Gan in Biosystems Engineering and Soil Sciences. This is different from our initial estimate of $98.24 because we have accumulated some costs from our AWS hosting service.

The items that we ordered include a waterproof enclosure, an Arduino Wi-Fi module and USB, a breadboard adapter for the Wi-Fi module, a 9V rechargeable battery, and a battery case. Each of these items is essential for us to build a basic functioning product. The waterproof enclosure and battery case are to protect our electronics from the elements since they are supposed to be outside. The 9V battery will be our power source for our product. Finally, the Wi-Fi hardware is needed so that the Arduino will be able to communicate with our backend so that we can store the data we collect. The cost for each of these items is listed in **Table 8**.

Table 8 - Project Budget

Item	Cost	Notes
Waterproof Enclosure	$27.49	Outdoors electronic protection
Arduino Wi-Fi Module and USB	$22.97	Allows Arduino to communicate wirelessly
Breadboard Adapter for Wi-Fi Module	$12.90	Allows Wi-Fi module to be used with breadboard
Battery (9V)	$19.90	Power supply
Battery Case	$6.99	Protection for battery
Battery Case	$7.99	Protection for battery that worked better with our enclosure
AWS Service Fee	$28.00	The service where we host our backend

The AWS service fee reflects the amount that our backend team has had to pay out during the length of our project. This cost accumulated at a much higher rate than we were anticipating due to the

growing pains and unfamiliarity with the technology. There were several times where the environment was left running without us realizing it, which caused us to acquire some unnecessary costs.

Finally, we have the human hours that our team will be putting into the entire project. The recommended credit hour for outside-of-class work is one-to-three. Since COSC 402 is a three-credit class, each of us will ideally put in **nine hours of work each week** for the project. That will amount to a total of 630 hours of total work from the team over the fourteen weeks of the semester. The breakdown of hours per milestone is given in

Table 9. It is important to note that while our total expected hours of work have not changed for this project, our scope was too large for the time we had available to fully realize everything we wanted to do. As mentioned previously, we decided to remove the machine learning aspect from our project because we simply will not have the time for it.

Table 9 - Projected Hours per Milestone

Item	Deadline	Total Hours of Work
~~Elastic Beanstalk instance created~~	~~1/31/25~~	~~22.5~~
~~Wired Arduino/sensor prototype~~	~~2/1/25~~	~~22.5~~
~~Static frontend~~	~~2/12/25~~	~~90~~
EARLY PROTOTYPE (EP)	**2/12/25**	**Hours for EP: 135 Total Hours: 135**
~~Wireless Arduino/sensor prototype~~	~~2/19/25~~	~~45~~
~~Front/Backend API Gateways~~	~~2/26/25~~	~~45~~
~~Frontend app developed~~	~~3/4/25~~	~~45~~
MINIMUM VIABLE PRODUCT (MVP)	**3/5/25**	**Hours for MVP: 135 Total Hours: 270**
Machine learning models (*Removed from scope)*	N/A	N/A
~~Finalize HW enclosure~~	~~4/2/25~~	~~45~~
~~Refine application~~	~~4/2/25~~	~~135~~
PRE-RELEASE	**4/2/25**	**Hours for PRE-RELEASE: 180 Total Hours: 450**

~~Testing, documenting, and further refinement~~	~~4/30/25~~	~~180~~
PRODUCT RELEASE	**4/30/25**	**Hours for Release: 180** **Total Hours: 630**

References

[1] USDA, "Ag and Food Sectors and the Economy," USDA, November 1 2024. [Online]. Available: https://www.ers.usda.gov/data-products/ag-and-food-statistics-charting-the-essentials/ag-and-food-sectors-and-the-economy/. [Accessed 24 November 2024].

[2] USDA, "Farming and Farm Income," USDA, 6 September 2024. [Online]. Available: https://www.ers.usda.gov/data-products/ag-and-food-statistics-charting-the-essentials/farming-and-farm-income/. [Accessed 24 November 24].

[3] World Bank Group, "Health Nutrition and Population Statistics: Population estimates and projections," World Bank Group, 1 July 2024. [Online]. Available: https://databank.worldbank.org/Population/id/622a9444. [Accessed 25 November 2024].

[4] Y. Liu, X. Ma, L. Shu, G. P. Hancke and A. M. Abu-Mahfouz, "From Industry 4.0 to Agriculture 4.0: Current Status, Enabling Technologies, and Research Challenges," *IEEE Transactions on Industrial Informatics,* vol. 17, no. 6, pp. 4322-4334, 2021.

[5] K. H. Coble and e. al., "Big Data in Agriculture: A Challenge for the Future," *Applied Economic Perspectives and Policy,* vol. 40, no. 1, pp. 79-96, 2018.

[6] G. Users, "ThingsBoard GitHub Repo," ThingsBoard, 25 November 2024. [Online]. Available: https://github.com/thingsboard/thingsboard. [Accessed 26 November 2024].

[7] G. Users, "OpenRemote GitHub Repo," OpenRemote, 26 November 2024. [Online]. Available: https://github.com/openremote/openremote. [Accessed 26 November 2024].

[8] V. Kumar, K. V. Sharma, N. Kedam, A. Patel, T. R. Kate and U. Rathnayake, "A comprehensive review on smart and sustainable agriculture using IoT technologies," *Smart Agricultural Technology,* vol. 8, p. 100487, 2024.

[9] O. Friha, M. A. Ferrag, L. Shu, L. Maglaras and X. Wang, "Internet of Things for the Future of Smart Agriculture: A Comprehensive Survey of Emerging Technologies," *IEEE/CAA J. Autom. Sinica,* vol. 8, no. 4, pp. 718-752, 2021.

[10] J. Bauer and N. Aschenbruck, "Design and implementation of an agricultural monitoring system for smart farming," in *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, Tuscany, 2018.

[11] R. Akhter and S. A. Sofi, "Precision agriculture using IoT data analytics and machine learning," *Journal of King Saud University - Computer and Information Sciences,* vol. 34, no. 8, pp. 5602-5618, 2022.

[12] S. K. S. Durai and M. D. Shamili, "Smart farming using Machine Learning and Deep Learning techniques," *Decision Analytics Journal,* vol. 3, p. 100041, 2022.

[13] "Hao Gan," UTK, 10 August 2023. [Online]. Available: https://utia.tennessee.edu/spotlights/hao-gan/. [Accessed 1 December 2024].

[14] Y. Chen, W. Lee, H. Gan, N. Peres, C. Fraisse and Y. Zhang, "Strawberry Yield Prediction Based on a Deep Neural Network Using High-Resolution Aerial Orthoimages," *Remote Sensing,* vol. 11, no. 13, p. 1584, 2019.

[15] "Field Soil Samples," UT INSTITUTE OF AGRICULTURE, 2024. [Online]. Available: https://soillab.tennessee.edu/soil-testing/field-soil-samples/. [Accessed 1 December 2024].

[16] JayconSystems, "Getting Started With the ESP8266 ESP-01," Instructables, [Online]. Available: https://www.instructables.com/Getting-Started-With-the-ESP8266-ESP-01/. [Accessed 11 February 2025].

[17] Amazon, "Protecting data using encryption - AWS Elastic Beanstalk," docs.aws.amazon.com, 7 September 2024. [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/security-data-protection-encryption.html. [Accessed 2 December 2024].

[18] Amazon, "Configuring Amazon Virtual Private Cloud (Amazon VPC) with Elastic Beanstalk - AWS Elastic Beanstalk," Amazon.com, 7 September 2024. [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.vpc.html. [Accessed 2 December 2024].

[19] Microsoft, "Azure Storage encryption for data at rest," Microsoft.com, 12 February 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/storage/common/storage-service-encryption. [Accessed 2 December 2024].

[20] Microsoft, "Pricing – App Service for Linux | Microsoft Azure," Microsoft, 2024. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/app-service/linux/#pricing. [Accessed 2 December 2024].

[21] AWS, "AWS Free Tier," Amazon Web Services Inc, 2023. [Online]. Available: https://aws.amazon.com/free/. [Accessed 1 December 2024].

[22] Lilisa, "Breaking Down the Zigbee vs. WIFI Debate: A Professional Comparison," MOES, 28 April 2024. [Online]. Available: https://moeshouse.com/blogs/news/zigbee-vs-

wifi?srsltid=AfmBOoo96Ohea02mKHaQSNOV1R0a7HDmBIzLlM3jevnWhPCZOM8z1mUH. [Accessed 2 December 2024].

[23] T. Zyarych, "LoRaWAN vs Zigbee for Your IoT Project | TEKTELIC Blog," TEKTELIC, 23 October 2023. [Online]. Available: https://tektelic.com/expertise/lorawan-vs-zigbee/. [Accessed 1 December 2024].

[24] S. T. M. Y. S. H. G. W. S. L. Y. C. Z. W. Chapel Reid Rice, "Perception, path planning, and flight control for a drone-enabled autonomous pollination system," *Robotics,* vol. 11, no. 6, p. 144, 2022.