6-2021

IoT Garden Frost Alarm

Andrew James
Union College - Schenectady, NY

IoT Garden Frost Alarm

Andrew James

ECE 499

Engineering Capstone

Advised by Prof. Cherrice Traver

June 8, 2021

Home gardeners are faced with yearly challenges due to spring frosts harming young plants. This is frequently mitigated by covering crops with frost blankets, but only on nights when a frost is predicted. In areas with less predictable climate, an unexpected frost can kill vulnerable plants, reducing the amount of food produced. A system is proposed and designed here to use internet of things (IoT) technology to enable a small weather station in the home garden to report current climate data and predict frosts, then alert the gardener in time for them to cover their plants.

The system as designed consists of an IoT endpoint, powered by a microcontroller in a watertight housing and connected to a weatherproof temperature and humidity sensor, as well as cloud services configured to collect and analyze weather data, and finally an iOS app for gardeners to monitor the conditions in their gardens and receive push notifications about incoming frosts. To make the system accessible to home gardeners, the design was required to cost under $200 to construct and be as inexpensive as possible in operating costs, measure temperature to within half a degree centigrade, be capable of operating at temperatures down to -20ºC, support ethernet connections for gardens far from the house, and must allow the gardener to configure a temperature threshold below which they will always be notified, whether or not the dew point indicates an incoming frost.

An ESP32 microcontroller, coded in C++ in the Arduino environment, was used for the final endpoint design, along with an ethernet shield and the SHT31 temperature and humidity sensor. Google Cloud services were used for the cloud data pipeline, and Google Firebase was used for a database and for communication with the iOS app. The iOS app was developed in Swift using the SwiftUI framework, with significant support from Firebase libraries for communication with the cloud.

All design goals were met by the final design except for the low yearly operating cost. This was due to the unexpected requirement of having an Apple Developer account, which costs $99 per year, in order to send push notifications to iOS apps. Due to this unexpectedly high cost, several alternative notification solutions, including an Android app and a secondary physical endpoint with alarm functionality, are proposed at the end of this report, as well as suggestions for potential commercialization.

Table of Contents

List of Figures and Tables

Introduction

Home gardeners cultivating crops in cold climates, particularly those with short growing seasons and microclimates that make weather unpredictable, face yearly issues of frost damaged crops. Young crops in the spring are especially vulnerable to frost damage, potentially stunting their growth and limiting the produce a gardener can harvest at the end of the season. This reduces the economic efficiency of home gardening as an alternative to commercial agriculture, posing a problem not just for the gardener but also for society (since home gardens produce positive health and environmental externalities) [1],[2]. Gardeners in these environments have developed techniques, such as frost blankets that cover crops at night [3], to mitigate these losses, but these techniques are generally only employed when a frost is predicted by the National Weather Service (a division of NOAA) or other government agencies for non-US gardeners. Thus, frosts not predicted in advance have the most potential to cause crop loss.

The goal of this project is to create and implement an open-source system that predicts imminent frosts in the garden in a way that is inexpensive for gardeners to build and use at home. This is accomplished using a microcontroller-based Internet of Things (IoT) endpoint connected to a temperature and humidity sensor, all located in the home garden, that sends collected hyperlocal weather conditions to a cloud service. This cloud service processes and stores current weather conditions, analyzing them for the likelihood of imminent frost, and alerts the gardener via a mobile application on their phone when a frost is predicted. It also allows the gardener to view up-to-the-minute data, provided by the sensors, on the mobile app.

Throughout the rest of this report, the necessary technical background for development of this system is laid out, along with the design requirements of the system and the methodology by which components of it were chosen. The initial and final design details are then explained, along with the performance results obtained from tests of the system. The production schedule is outlined, then a short analysis of the cost of building and operating the system, a user's manual, and finally a discussion of conclusions and recommendations is presented to enable future work on this issue.

Background

Preventing frost damage to plants is an important undertaking for farmers and home gardeners in both the spring and fall seasons when frosts are difficult to predict [3]. This is particularly true in northern regions, where the growing season is generally shorter and must be extended by frost mitigation techniques for some crops to reach maturity. Mountainous regions and other microclimate areas further complicate these efforts, as temperatures predicted and measured at a weather station may differ significantly from the actual temperature in nearby fields. Farmers often deal with this issue by using automated watering, heating, and fan systems to keep the temperature around plants above freezing throughout the night and early morning. For home gardeners, however, simply covering their plants with fabric or plastic sheets is more economically feasible [3]. The issue for gardeners is that these coverings must be removed promptly in the morning when the temperature rises. This covering and uncovering of crops is an unproductive use of time when frost does not occur, making it difficult to judge when it is worth doing so and imperiling crops when a mistake is made. Increasing crop yield for home gardeners produces positive environmental and health benefits [1],[2] and makes the gardening experience more economical and rewarding, so systems that reduce frost damage are valuable to both the gardener and society at large.

Market Solutions

Systems solving the problem of frost warnings are available on the market, but are priced for agricultural use, making them unreasonably expensive for gardeners and family farmers. The cheapest and most widely available of these systems are made by a company called Onset and start at $315 for a USB-only data logger with a temperature sensor; their cheapest model with internet communication, the HOBO MicroRX Station, starts at $560 and requires an expensive yearly data plan [4]. These devices offer significant benefits to farmers due to their many sensor input options, but for gardeners these extra options drive up the cost and complexity to install. Moderately cheaper smart home temperature sensors also exist, but generally are designed for indoor use and are not suitable for gardens.

Figure 1: Plant damage caused by frost. Image: Phil Romans, 2009

Meteorology

The relationship between freezing, frost, and dew point is also relevant to this project. The dew point is the temperature at and below which relative humidity will be 100% for the current amount of moisture in the air. When the air temperature reaches the dew point, the air cannot hold any more moisture, causing dew to form on nearby surfaces. Thus, the dew point has a direct relationship with the total level of moisture in the air; the lower the dew point temperature is, the lower the absolute humidity [5]. If the dew point is below the freezing point of water, the air temperature is more likely to drop below freezing because the dry air has less water vapor to absorb and retain heat. Thus, higher dew points indicate the air temperature is less likely to fall below freezing [5].

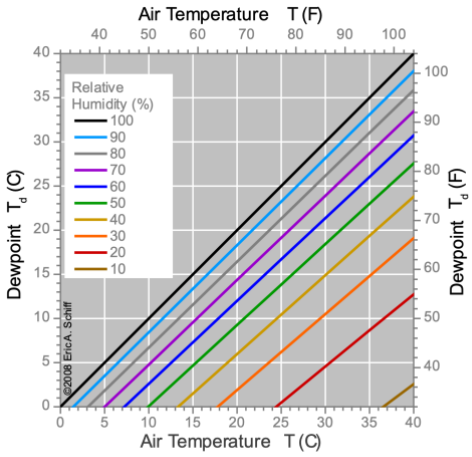

Figure 2: Relationship between temperature, dew point, and humidity. Image: Easchif, Wikimedia, 2008

If a dew point below freezing is reached by the air temperature, the dew will generally condense as frost, causing crop damage. Even without frost, however, temperatures more than a few degrees Fahrenheit below 32º can cause significant crop damage, sometimes even more than if frost had formed, since a mild frost can insulate plants from further freezing damage [6]. The primary type of freeze condition that causes problems for gardens and farms, radiation freezes, generally occur when the air is dry, causing the ground to rapidly lose heat into the air and creating a temperature inversion [5]. This means that the air right above the ground is several degrees cooler than the air only a few feet above, necessitating that temperature is measured as low to the ground as possible to determine when a freeze is likely to occur.

Security in Internet of Things Applications

The security of Internet of Things (IoT) applications is a recurring ethical and privacy issue for IoT devices and systems [7]. Since most IoT applications collect large amounts of data, which may be private in nature, and many allow for interfacing with the outside world via the internet, it is vital that data is secured throughout its entire pipeline, from endpoint devices to cloud services and on to the end user [8]. In transit, data can be secured via encryption, and data inputs and outputs at the cloud/storage level can be secured by confirming that devices and users attempting to send or receive data have been properly authenticated, either with a token generated by a sign-in service or through private/public key signatures to confirm the originator of a given packet [8],[9]. When security roles are configured at these inputs and outputs, the established best practice is the "principle of least privilege," wherein each (human or device) user of a system is assigned the minimum access and modification privileges needed to complete its task, and not given any other rights until they are absolutely necessary [8]. Cloud services built for IoT applications generally have these security features and practices included in their architecture, but it is necessary to configure them correctly to ensure that data security is upheld.

Standards Utilized

IP Code

The ingress protection code (IP code), developed by the International Electrotechnical Commission (IEC), specifies the ability of device housing to prevent the intrusion of both solid particles and water. The code's different protection standards are written as IPXX. The first X is a number from 0 to 6 indicating the size of solid particles prevented from ingress (0 indicating no protection and 6 indicating full resistance against dust). The second X is a number from 0 to 8 indicating the resistance of the device enclosure to various degrees of water ingress (0 indicating no protection and 8 indicating total protection to immersion in water at depths below 1 meter, along with an uncommon 9K rating to indicate resistance to high-temperature, high-pressure water) [10]. Most consumer electronics that are certified are rated for either IP67 or IP68, indicating complete resistance to dust and water resistance either to an immersion depth of 1 meter or greater than 1 meter, respectively [11].

MQTT

MQTT is a low-data IoT message transport protocol, built on top of the standard TCP/IP stack, that utilizes a client-server architecture with connect, publish, subscribe, unsubscribe, and disconnect packets to control information flow. All control packets have an optional acknowledgement packet associated with them that, depending on the quality-of-service level set during the initial connection, affirms when each packet is received, and optionally specifies the specific packet payload received (increasing overhead along with reliability). The server in this model, referred to as a broker, receives published packets on a particular topic and forwards these on to subscribers to that topic, with varying checks in place to ensure receipt depending on the connection settings of each subscriber [12].

MQTT is the preferred IoT standard for messaging between servers/cloud services and microcontroller-based endpoints, utilized as the primary communication method for the IoT components of Google Cloud [13], Amazon AWS [14], and Microsoft Azure [15] due to its security, low data overhead, and robust feature set.

I²C

I²C is a standardized bidirectional two-wire data bus for communication over short distances between integrated circuits [16]. It supports a wide feature set, including data collision detection, a master/slave relationship between various ICs, and unique addresses for each IC so that one I²C bus can connect many ICs without introducing errors where one IC inadvertently makes use of data intended for another. I²C is the communication standard of choice for most integrated circuit-based digital temperature and humidity sensors for interfacing with a microcontroller. Most microcontrollers support it natively and have first party libraries for the standard that simplify interfacing with I²C devices in code.

Design Requirements

Economic Limitations

As this project is intended to be used by home gardeners, it should be as inexpensive as possible to construct and operate. Since existing market solutions are on the order of $500+, *the maximum price for all components needed to assemble and run this system should be under $200.* The prices for each component should be minimized as much as possible, even below the overall $200 threshold. Additionally, the ongoing cost of operation should be as low as possible, ideally less than $5 a year.

Endpoint Requirements

Sensor Accuracy

To ensure accurate prediction of frost, *temperatures between -5 and 5ºC must be readable within an accuracy of ±0.5ºC* (equivalent to about 1ºF in this range), and *relative humidity must be readable within ±5%*, as this is sufficient to predict dew point within 5ºF for RH≥20% at 32ºF. When selecting a temperature and humidity sensor (almost all digital humidity sensor ICs also contain onboard temperature sensors, so using the same sensor for both is more economically and programmatically feasible), preference should be given to more accurate temperature readings than humidity readings, as temperature is the more significant component in determining when frost is likely to occur.

Communications & Reliability

For purposes of reliable communication and ease of accessing the internet in outlying gardens that may be beyond the range of home wireless routers, *the chosen microcontroller should support internet communication over Ethernet.* Optionally, it may also support WiFi communications to provide gardeners with a secondary communication option if this does not significantly increase cost.

Communication between the temperature sensor and microcontroller should be easy to parse and in a standardized format with library support; thus, *both the temperature sensor and microcontroller should support either I²C, SPI, or 1-Wire communications.*

For ease of development and reliable operation, *libraries should be available for the chosen microcontroller to communicate with cloud services using MQTT.* Preference should be given to microcontroller platforms actively supported by the major cloud service providers.

Component Temperature Ratings

All outdoor components must be capable of operating in temperatures from -20ºC to at least 30ºC to withstand the most extreme conditions they are likely to experience in use.

Power

The system *must be powered via line power for regular operation* to ensure reliability; the system *may optionally have a backup battery and/or solar power option if cost allows.*

Waterproofing

For the purposes of this project, it is not realistic to achieve a true IP rating due to the high cost of testing, as well as a lack of access to proper manufacturing technology to reach one of the standard consumer electronics ratings. Although housings are available with these ratings, the necessity of drilling holes in the case for this project, as explained later, effectively compromises them, limiting the realistic possible resistance to both dust and water. However, a lower-than-standard rating of IP43, indicating protection from solid objects as small as most insects and water sprayed up to 60º from the vertical axis, should be possible if an IP66/67 housing is utilized and cable-routing holes are only drilled in the bottom side of the case that faces downwards when mounted. Thus, *it is required that the project housing be IP66 or IP67 rated (or equivalent).*

Additionally, since the temperature and humidity sensor must be placed slightly outside the housing to ensure proper humidity readings, *the sensor must be waterproofed or capable of being waterproofed.*

Enclosure Size

The project housing (enclosure) *should be as small as is feasible, ideally under 6"x6"x6"*, to enable easier mounting in gardens without taking up too much space.

Cloud Service Requirements

IoT Communication

The selected cloud service *must be capable of communicating with the microcontroller endpoint by acting as both an MQTT broker and subscriber.*

Serverless Applications

The selected cloud service *must have the ability to run serverless functions to calculate frost likeliness based on sensor data and use this to determine whether to send a warning push notification to the user.*

Databases

The selected cloud service *should ideally have a hosted database application to allow for the storage of historical temperature and humidity data for later reference.*

App Communication & Push Notifications

The selected cloud service *must have a way to send push notifications to a mobile app on the user's phone,* and *must allow for communication of current sensor data when requested by the mobile app.*

Mobile App Requirements

Notifications

The mobile app *must be able to receive and display push notifications from the cloud platform about likely imminent frosts.*

Sensor Data

The mobile app *must be able to pull current sensor readings from the cloud service when open and must display these readings as they are updated in a human readable form.*

User Configurability

The mobile application *must allow the user to set a temperature threshold for sensor readings at which they will always receive a push notification warning of frost, and this threshold must be synced with the cloud service.* Additionally, the application *should allow the user to view temperatures and set temperature thresholds in both Fahrenheit and Celsius, depending on their choosing.*

Design Alternatives

The following design alternatives represent the choices made for each major component of the system, as represented by the block diagram in Figure 3 to the right, as well as discrete sub-components like the power supply and housing that are not included in the block diagram due to their lack of logical relevance. The temperature and humidity sensor were chosen as a single unit for reasons described previously, as was the cloud service provider after determining that it would allow for easier development and reproduction to make use of only one cloud ecosystem. Components of the system were chosen in the order they are listed in below; some component choices, such as the sensor, determined additional requirements for other components, like the microcontroller, as they imposed limitations on what communication standards, power usages, etc. must be supported.

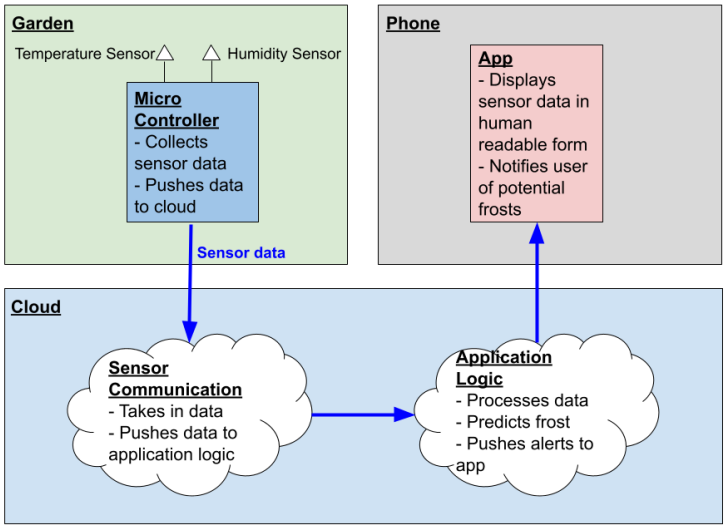

Figure 3: High-level Block Diagram

Temperature and Humidity Sensor

The specifications used to select a temperature and humidity sensor include the design requirements specified previously, as well as the following list, all of which are compiled in the decision matrix in Table 1 below:

- The sensor should have an operating voltage between 3.3 and 5V and an operating current ≤10mA so it can be powered easily by a microcontroller.
- The sensor should have a response time of 10 seconds or less.

Table 1: Temperature and Humidity Sensor Decision Matrix

Brand	Item	Waterproofing	Operating Temperature	Operating Voltage	Max Operating Current	Max. error in temp. from -5 to 5ºC	Max. error in RH	Data comm. standard	Response Time	Price for 1
DFRobot	SEN0385	Built-in	-40 to 125ºC	3.3 to 5V	<1.5mA	±0.25ºC	±2% RH	I^2C	8 sec	$19.90 [17]
Seeed Technology Co.	101990561	Addon	-40 to 80ºC	3.3 to 5.5V	≤0.22mA	±1.0ºC	±6% RH	One-wire bus	10 sec	$4.99 [18]
Adafruit Industries	393	Addon	-40 to 80ºC	3.3 to 6V	≤1.5mA	±0.5ºC	±5% RH	One-wire bus	2 sec	$15.00 [19]
DFRobot	SEN0227	Built-in	-40 to 125ºC	3.3 to 5V	≤0.2mA	±0.5ºC	±7.5 % RH	I^2C	8 sec	$22.50 [20]
Adafruit Industries	1293	Addon	-40 to 125ºC	3.3 to 5.5V	≤0.5mA	±0.15ºC	±3% RH	I^2C	5 sec	$29.95 [21]
Amphenol Advanced Sensors	T9602-3-A-1	Built-in (IP67)	-20ºC to 70ºC	3.3 to 5V	≤0.75mA	±1.0ºC	±3.5 % RH	I^2C	29 sec	$51.42 [22]

Based on the information in the table, the DFRobot SEN0385 [17] was chosen because its operating voltage, current, and temperature ranges all fit the desired criteria, and it had the second-best temperature error (behind the Adafruit 1293) and the best relative humidity error. It also uses I^2C for communication, meaning it will be simple to interface with a microcontroller, has an acceptable temperature change response time, a reasonable price, and built-in waterproofing, all also making it a good candidate. The Amphenol Advanced Sensors T9602-3-A-1 was the only sensor researched that had a proper IP67 rating, but it was also significantly more expensive at over $50 and had worse temperature error, relative humidity error, and response times than most of the other sensors. For a commercial product with a warranty, requiring an IPxx water/dustproof rating would make more sense, but given the constraints of this project, the SEN0385, which is designed for outdoor use but not officially certified, is an acceptable compromise.

Microcontroller

The microcontroller was selected based on the design requirements specified previously, as well as the following criteria, both of which are compiled in the decision matrix in Table 2 below:

- Must operate on either 3.3 or 5V
- Should be programmable with common languages (like C, C++, Arduino C++, Python)
- Should not have an operating system, or only have a barebones RTOS if necessary, to minimize development time and points of failure
- Should be well documented with an active development community

Table 2: Microcontroller Decision Matrix

Brand	Item	Operating Voltage	Operating Temperature	I/O Options	Network Standards	Cloud Services/ Protocols	Programming Language	OS?	Price
Raspberry Pi	Pi 4 Model B	5V	0 to 50ºC	I^2C, SPI, 1-wire, (with extra configuration), UART	Gigabit Ethernet	MQTT; SDKs for AWS, Azure, Google Cloud	Python, C/C++	Raspbian (Default), other Linux distros	$35.00 [23]
Adafruit	HUZZAH 32	3.3V	-40 to 125ºC	I^2C (2 channels); SPI (2 channels); UART (2 usable channels); ADC (12 inputs); DAC (2 outputs); I2S (1 channel)	802.11 b/g/n WiFi, Bluetooth, 10/100 Ethernet (w/ shield)	MQTT; SDKs for AWS, Azure, Google Cloud	C++	None (Arduino), FreeRTOS (ESP-IDF)	$19.95 [24] (extra $19.95 for Eth. [25])
Adafruit	HUZZAH with ESP8266	3.3V	-40 to 125ºC	9 GPIO pins usable as up to 4 I^2C channels or up to 2 SPI channels; ADC (1 input)	802.11 b/g/n WiFi, 10/100 Ethernet (w/ shield)	MQTT; SDKs for AWS, Azure, Google Cloud	C++, Lua	None	$16.95 [26] (extra $19.95 for Eth. [25])
Arduino	MKR WiFi 1010	3.3V	-40 to 125ºC	SPI (1 channel), I^2C (1 channel), UART (1 channel), ADC (7 inputs), DAC (13 outputs)	802.11b/g/n WiFi, Bluetooth, 10/100 Ethernet (w/ shield)	MQTT; SDKs for AWS, Azure, Arduino IoT Cloud, Google Cloud	C++	None	$32.10 [27] (extra $26.40 for Eth. [28])
DF-Robot	XBoard V2	3.3 or 5V	-40 to 125ºC	SPI (1 channel), I^2C (1 channel), UART (1 usable channel), ADC (8 inputs), DAC (4 outputs)	10/100 Ethernet	MQTT	C++	None	$19.90 [29] (extra $12.90 for programmer [30])

The Adafruit HUZZAH32 [24] was selected, along with the corresponding Adafruit Ethernet FeatherWing [25], from the microcontroller options above because of its comparatively low price and high feature set. The cheapest option, the DFRobot XBoard V2, was eliminated largely due to it lacking sufficient documentation, cloud service SDK support, and an onboard WiFi chip, along with the hassle involved in programming it with an outboard programmer. The Raspberry Pi was eliminated because it is designed to run a full operating system (usually

various Linux on ARM distros), making it overcomplex for the needs of this project and therefore more likely to be error-prone; it also does not have an onboard WiFi chip and has a much more limited suggested temperature range that does not cover the low end of the intended operating range of this project. The Arduino MKR board considered (the MKR WiFi 1010, the lowest cost MKR board available) was ruled out due to its high expense, especially when paired with the corresponding ethernet shield, and lack of substantially different features from the HUZZAH32 and HUZZAH ESP8266. Finally, when deciding between the HUZZAH32 and HUZZAH ESP8266, the HUZZAH32 was selected because it has substantially more I/O options, Bluetooth support, more advanced SDKs available, more processing power, and because the ESP32 chip it is based on is replacing the ESP8266 as a new industry standard for low-cost IoT devices, all of which contribute only $3 more to the overall cost of the project and increase the reusability and long-term support of the dev board.

Power Supply

The two power options suggested for the HUZZAH32 by the manufacturer Adafruit are: a) via the onboard Micro-USB port and b) connecting to a 3.7/4.2V LiPo battery (also sold by Adafruit) via the onboard JST port. They recommend against using an external power supply due to potential logic issues it could introduce, despite being technically possible. Given the nature of this project, a battery that requires recharging on a semi-frequent basis and is sensitive to cold (as LiPo batteries are) is not appropriate as a primary power source. The battery circuitry on the board is hot-swappable between charging and discharging when USB power is connected and disconnected, and the battery port is plug-and-play, so it could be easily added as a backup if desired. Due to the ease of accomplishing this, no design is necessary; a user that wants a backup battery can simply plug one in [31].

Power-over-ethernet is also a potential option, providing both power and internet access with one cable, but requires a special routing switch to inject power, as well as a PoE splitter and a voltage regulator to bring the voltage down to the 5V needed for the HUZZAH32's USB input, making the project significantly more expensive to implement. Given the difficulties, this option has been ruled out.

For the USB power, the simple and modular solution is to connect the HUZZAH32 to a USB wall wart whose prongs can protrude from the bottom of the project housing to plug into an outdoor extension cord. This wall wart:

- Must output 5V and be rated for at least 1A of current
- Should be as physically small as possible

Table 3: Power Supply Decision Matrix

Brand	Item	Rated output current at 5V	Rated operating temperature	Dimensions	Price
Phihong USA	PSAA05A-050QL6-R	1.0A	-10 to 40ºC	2.19x1.32x0.82in	$4.45 [32]
Qualtek	QFAW-05-05	1.0A	-20 to 40ºC	1.52x1.18x1.18in	$6.15 [33]

Mean Well USA	GS05U-USB	1.0A	-20 to 50ºC	1.65x1.29x0.94in	$11.58 [34]

The Qualtek model [33] listed above was selected because of its adherence to the output and operating temperature requirements, and because it was nearly half the cost of the Mean Well USA model.

Housing

The choice of housing for this project was determined by the waterproofing and size requirements laid out previously, along with the goals of minimizing price and using materials that are easy to drill holes into but provide sufficient protection from the sun by not exposing the electronics through a transparent cover. Two primary companies were identified, Bud Industries and Boxco, that manufacture suitable enclosures for this project, each company offering a wide variety of sizes for each of their product lines. The different product lines from each brand are specified in the decision matrix in Table 4 below, along with their physical properties, the most applicable size for this project, and its price. Only product lines made from plastic and utilizing a hinged closure mechanism were considered for ease of use reasons, and product lines that did not come in an appropriate size were also eliminated before consideration.

Table 4: Project Housing Decision Matrix

Brand	Product Line	Waterproofing Standard	Enclosure Material	Cover Transparent or Opaque?	Applicable Size	Price (from DigiKey)
Boxco	Q Series	IP66	ABS Plastic	Either	5.91×5.91×3.54	Not listed [35]
	P Series	IP66	ABS or Polycarbonate	Either	4.33×8.27×2.95	$18.30 [36]
	R Series	Not certified	ABS Plastic	Either	5.91×5.91×3.74	Not listed [37]
Bud Industries	NBB Series	IP66	ABS/Poly Blend	Either	5.91" x 5.91" x 3.57"	$29.00 [38]
	NBF Series	IP66	ABS/Poly Blend	Opaque	5.91" x 5.91" x 3.54"	$20.60 [39]

The Bud Industries NBF Series enclosure was chosen in the applicable size specified in Table 4, with product number NBF-32110 [39], due to its significantly lower price than the flame retardant NBB series version and its more convenient size and shape than the closest Boxco P series equivalent. Two of the Boxco options, the Q and R series, are not carried by DigiKey, requiring that they be custom ordered from the manufacturer at greater expense, so they were ruled out. The chosen enclosure is fully opaque, made of off-white plastic that should effectively reflect the sun, comes with mounting brackets for easy installation outdoors, and remains fairly inexpensive.

Cloud Service Provider

A cloud service, or combination of cloud services, is needed to receive temperature data from the IoT endpoint (HUZZAH32), process this data to detect when a frost is imminent, and send a push notification to the end user when an oncoming frost is detected. It should also be able to send the end user data about the current temperature and humidity in their garden on

request (when they open the app or refresh while in the app). Ideally, it would also be able to store temperature and humidity data over time as well so that trends could be identified.

Most cloud services relevant to this project are parts of larger cloud platforms offered by large vendors. The specific relevant components of these platforms are identified in the tables below and evaluated based on the previously specified design requirements, along with the use allotments in the service's free tier:

Table 5: Cloud MQTT Broker Decision Matrix

Cloud Platform	MQTT	
	Service	Free tier allotment
Microsoft Azure	IoT Hub	4MB per day for 12 months [40]
Amazon AWS	IoT Core	2.25 million connection minutes and 250MB of messages per month for 12 months [41]
Google Cloud	IoT Core	250MB of messages per month; no charge for connection but PINGREQs count as messages [42]

Table 6: Serverless Cloud Application Decision Matrix

Cloud Platform	Serverless Applications	
	Service	Free tier allotment
Microsoft Azure	Functions	1 million invocations and 400,000 GB-seconds compute per month [43]
Amazon AWS	Lambda	1 million invocations and 400,000 GB-seconds compute per month [44]
Google Cloud	Cloud Function	2 million invocations and 400,000 GB-seconds compute per month [45]

Table 7: Push Notification Service Decision Matrix

Cloud Platform	Push notification Platform	
	Service	Free tier allotment
Microsoft Azure	Notification Hubs	1 million publishes per month [46]

Amazon AWS	Simple Notification System	1 million publishes per month [47]
Google Cloud	Firebase Cloud Messaging	Unlimited [48]

Table 8: Cloud Database Decision Matrix

Cloud Platform	Database	
	Service	Free tier allotment
Microsoft Azure	Cosmos DB	5GB storage and 400 request units per second for 12 months [49]
Amazon AWS	DynamoDB/ DocumentDB	25GB data storage, 2.5 million read requests, unlimited data transfer in, and 1GB data transfer out per month (both) [50]
Google Cloud	Cloud Firestore	1GB data storage, 10GB network data per month, 20k writes per day, 50k reads per day [51]
	Realtime Database	1GB data storage, 10GB network data per month, 100 simultaneous connections [52]

All three cloud platforms considered provide all of the necessary components required for this system with free tiers generous enough to support at least one instance of the IoT endpoint for an individual user. However, Google Cloud was selected due to its perpetual free tier; Amazon AWS and Microsoft Azure both have a 12-month free limit for their MQTT service, which is essential for communicating with the IoT endpoint, whereas Google Cloud is not limited, and Azure has an additional 12-month limit for their Cosmos DB cloud database that the other two do not. Google Cloud also offers an unlimited number of push notifications (theoretically, although they reserve the right to cut off abusive behavior that would not be necessary to any reasonable product) through Firebase Cloud Messaging. Although the costs associated with the MQTT and database services from Azure and AWS would be very low after the end of the free trial period, it is preferable for this project as an open source and low-cost endeavor to not require any charge at all. Additionally, using Google Cloud allows for easier and lower-maintenance app development using the Google Firebase SDK [53] to configure push notifications and data downloads to client devices, as well as simplified and secure connections between the MQTT IoT core and the document databases used.

App Development Platform

For the mobile application, iOS was chosen as the target development platform because of the availability of devices to test the app with. Due to this decision, only two software development packages, Apple's XCode (with the Swift programming language) [54] and Facebook's React Native (a fork of the React package for JavaScript) [55], were considered, as

only these two had any kind of support for Google Firebase SDKs. Both development packages support all of the features required for this project and React Native has the added benefit of easier porting of the app to other platforms (i.e., Android or the web). However, React Native does not have an official, Google-supported SDK for Firebase, and instead relies on a community-developed package, with far less documentation and bug-checking than the official SDKs. For this reason, Apple XCode with Swift was selected as the development package and language for the mobile app.

Preliminary Proposed Design

At a high level, the design of this project involves three primary subsystems: the microcontroller based IoT endpoint located in the user's garden, the cloud services used to receive and work with sensor data, and the mobile application on the user's phone. These systems are depicted in block diagram form in Figure 4 below, with the cloud services component broken further down into sensor communications and application logic functions. As conceptualized in this diagram, temperature and humidity data is produced by sensors in the garden connected to a microcontroller, which collects the data from the sensors and pushes it on to the cloud via MQTT publishes over the internet. That data is taken in by the sensor communication portion of the cloud and pushed on to the application logic component, which processes and stores the data, using a serverless function to determine whether an imminent frost is likely. If a frost is predicted, it also sends a warning push notification to the mobile app. The mobile app receives these notifications and displays them to the user through the standard notification paradigm of the host operating system. When the app is open, it can also read current sensor data from this portion of the cloud services and display it to the user in a human-readable format.

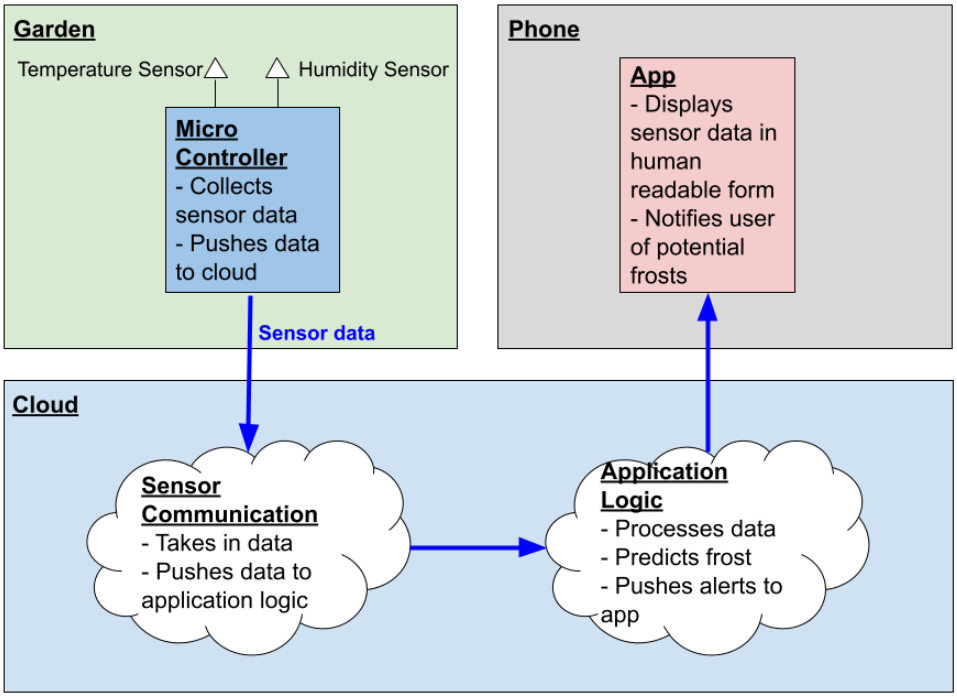

Figure 4: High-level Block Diagram

Endpoint

The microcontroller endpoint, whose logic is depicted in a state diagram in Figure 6 below, is controlled by an ESP32 chip embedded in the Adafruit HUZZAH32 development board.

Hardware

The board receives power via its Micro-USB port over a cable connected to the Qualtek USB power adapter. It connects to the internet via Ethernet using a WIZ5500 chip in the Adafruit Ethernet FeatherWing shield that sits atop the HUZZAH32 on stackable headers; the electrical connections between the ESP32 and the WIZ5500 are shown in the schematic in Figure 5 below. The DFRobot SEN0385 (with SHT31 chip) temperature and humidity sensor is connected to the microcontroller via four separate wires. The sensor's V_{CC} input is connected to the 3.3V output pin (labeled 3V) of the HUZZAH32 and the sensor's GND input is connected to the microcontroller GND output pin. Similarly, the sensor's SCL (clock) and SDA (data) inputs are connected to the corresponding SCL and SDA outputs of the microcontroller, but with 4.7kΩ pullup resistors connected between each to the 3.3V input as well, as is required by the I^2C specification.

All of the components described are housed inside the Bud Industries NBF-32110 case, mounted with the ethernet port oriented towards the bottom of the case. Three holes are drilled in the bottom of the case (which can be any of the narrow sides), through which an ethernet cable and an extension cord are routed for internet connection and power supply. The third hole is used to mount the external, waterproofed tip of the temperature and humidity sensor, held in place by the included washer on the sensor. The former holes are provided an extra degree of insulation from the elements by grommets through which the cables run, while the latter is already insulated by the tight washer seal created by the sensor.

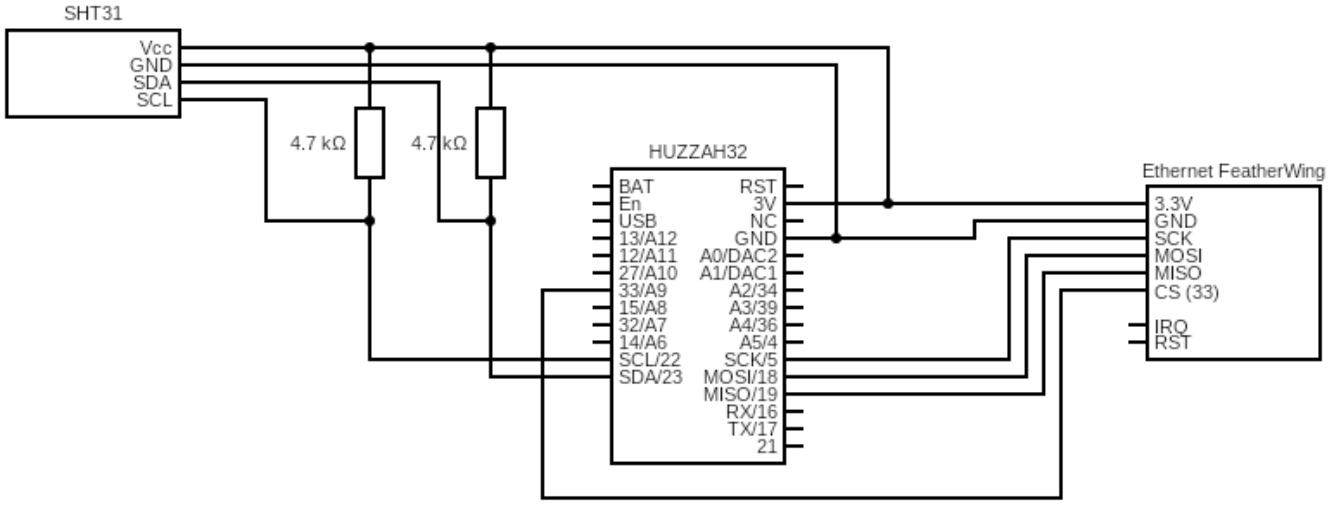

Figure 5: Endpoint Circuit Schematic

Software

The logic of the software running on the microcontroller, as depicted in Figure 6 below, involves a setup/power-on portion and a looping measurement and publication portion. When the device is powered on, it first attempts to connect to the SHT31 sensor over I^2C. If the connection is unsuccessful, it will retry making the connection every second until it is successful. Once the sensor is successfully connected, it will attempt to connect to the internet over the ethernet shield, retrying every 10 milliseconds until it succeeds. Finally, MQTT CONNECT packets are sent over the internet to the Google Cloud IoT Core broker at a regular interval (every second) until the connection is acknowledged with a CONNACK packet from IoT Core.

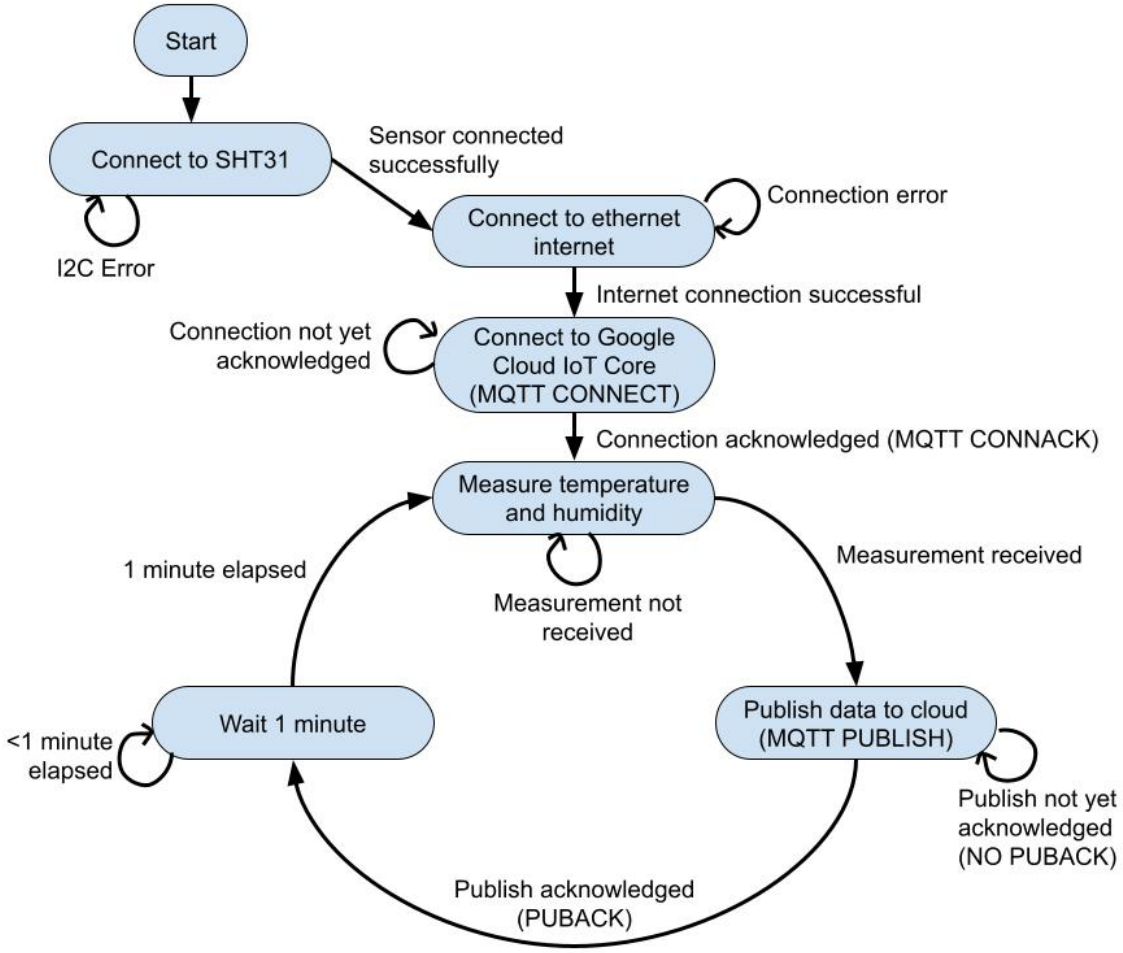

Figure 6: Endpoint Logic State Diagram

The looping portion of the code begins by reading temperature and humidity data from the sensor, waiting for the data to be received, and then publishing this data to IoT Core via a PUBLISH packet. The system then waits until the publication is acknowledged with a PUBACK or resends the packet if this is not received within 5 seconds. Once the publication is acknowledged, the system waits 1 minute, then returns to reading the sensor data.

The connection to and reading from the sensor is implemented using the DFRobot_SHT3X Arduino library [56]. A sensor object of the **sht3x** class is created in the variable setup that occurs prior to compilation, setting the correct pins to use for I^2C communication as well as the address of the sensor (0x44) and specifying a pin to use for

resetting the sensor chip (not usable with the given package but required by the library for compilation). Communication with the sensor is then initiated in the setup code using the sensor object's **begin** function inherited from the library. When reading the temperature and relative humidity from the sensor, an **SRHAndTemp_t** object (an alias for a C++ structure defined by the library) is created that holds the returned results (temperature in both Celsius and Fahrenheit, as well relative humidity and error codes) from the sensor object's **readTemperatureAndHumidity** function. This function is given as input an **eRepeatability_High** object, again defined in the library, that specifies that the sensor should measure as accurately (repeatably) as possible. This mode draws slightly more current while reading, but due to the use of line power, this is not a significant concern for this project.

The Ethernet internet connection is established utilizing functions from the standard Arduino Ethernet library [57]; the MAC address of the Ethernet shield is established in the pre-compilation section of the code as a byte array, along with a default IP address and DNS address to use for connection if DHCP fails. An **EthernetClient** object is then established, along with an **SSLClient** object on top of it (utilizing the SSLClient library created by the OPEnSLab at Oregon State University [58]), and the **Ethernet.init(pin)** command is called in the startup portion of the code with pin 33 specified (the SPI pin of the HUZZAH32O). The **Ethernet.begin(mac)** function is then called with the MAC address specified prior, and the program loops until Ethernet connection is successfully established with the router.

The Google Cloud IoT JWT library [59] is used for the generation of MQTT packets along with the arduino-mqtt library maintained by Joël Gähwiler [60] which the former references. An **MQTTClient** object from the latter class is created in the pre-compilation section of the code. A **CloudIoTCoreDevice** object from the former class is then created in the setup portion with references to the specific project, device ID, and other identifying data in the project's Cloud IoT Core instance, along with the private key used to sign messages sent to the cloud. The **MQTTClient** object is initialized with a 180ms keep-alive time and 1 second timeout, and a **CloudIoTCoreMqtt** object is then created that references the **SSLClient** along with the **MQTTClient** and **CloudIoTCoreDevice**. The **startMQTT** function inherited by this new object is then called, and once an MQTT connection is successfully established the temperature and humidity data read from the sensor are published using its **publishTelemetry** command. All encryption, packet formulation, signing of packets, and communication with Cloud IoT Core are handled by these libraries, obviating the need for further development in this area.

Cloud Functionality

At a high level, the Google Cloud functionality, as shown in Figure 7 below, operates as a pipeline for data starting with MQTT packet ingestion by Cloud IoT Core, which passes on published sensor data to a Cloud Pub/Sub topic. This topic triggers a Cloud Function instance when data is received that takes in the data and uses it to populate records in a Cloud Firestore database. The Cloud Function also calculates the likelihood of frost and sends a push notification via Firebase Cloud Messaging to the user's mobile app if the current temperature is below the user defined threshold, as captured in Figure 8 in the Firebase section below.

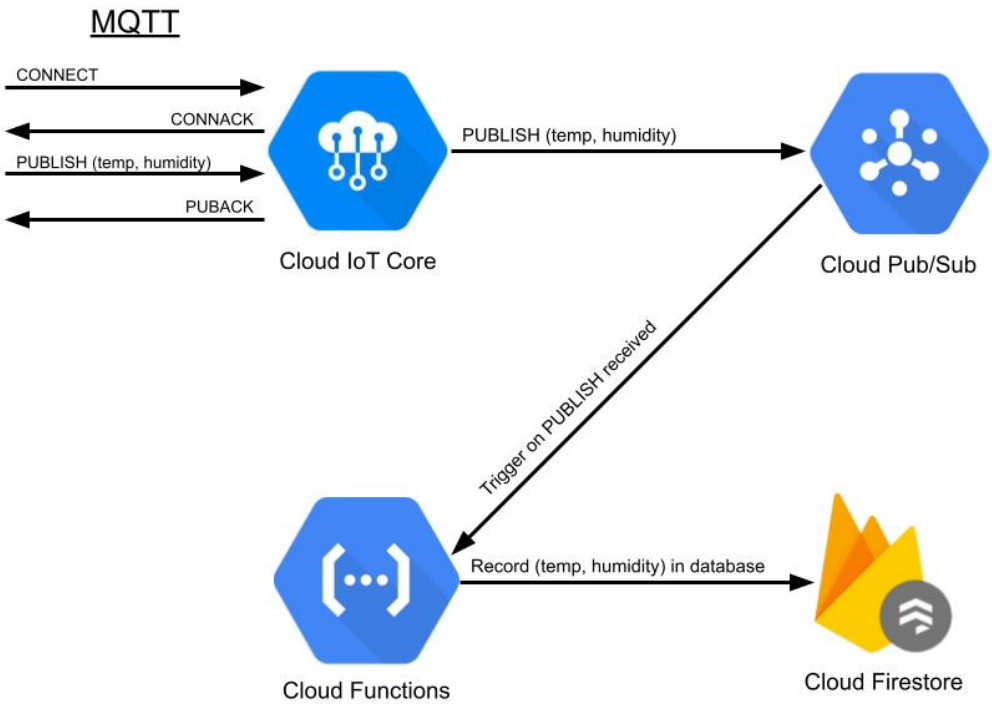

Figure 7: Cloud Logic Diagram

Cloud IoT Core

The Cloud IoT Core instance used for this project is configured with a device registry for endpoints (potentially allowing for many endpoint instances linked to one project) that is set up as an MQTT broker with a subscribed Cloud Pub/Sub topic for telemetry data. No Pub/Sub topic is configured for device state data, as this functionality is not used for this project. The endpoint is configured as a device in the IoT Core registry with communication allowed and a ES256 public key (paired with private key stored on the device itself), which is used to sign the JSON Web Token (JWT) password when it authenticates over MQTT to IoT Core. No registry-level CA certificate is used for authentication because this would create further overhead on the microcontroller publish functionality and is an unnecessary security step for a project that is very unlikely to be compromised due to its low-value functionality. At worst, if a hacker obtained a private key associated with an endpoint, they would be able to push bad temperature and humidity data; Cloud IoT Core has no other access that could compromise the system.

Cloud Pub/Sub

A Cloud Pub/Sub topic tied to the IoT Core registry is configured with an automatically-generated subscription that pulls incoming sensor data and stores it for up to 7 days or until it is acknowledged. The topic uses an AVRO (JSON) schema that ensures that input data is in the correct form (with key names "temp" and "hum" for temperature and humidity, and floating-point values for each) to ensure that data integrity is maintained when it is used in the Cloud Function and forwarded to the Firestore database. A Cloud Function is also set to trigger when new data is received, which also acknowledges the data in the process so that it is no longer stored by Pun/Sub.

Cloud Functions

The Cloud Function used in this project, triggered by Pub/Sub, takes as input the JSON-formatted sensor data and stores it as a timestamped entry in the Cloud Firestore database. After it has been stored, the likelihood of frost is calculated by determining the dew point. If the dew point is at or below 0ºC and the temperature is at or below 3ºC, a frost warning notification is triggered through the Firebase Cloud Messaging API and sent to the phone of the user. A warning notification is also sent regardless of the dew point if the temperature is below the user's threshold set in the user document in Cloud Firestore.

Cloud Firestore

The Cloud Firestore database, associated with the Firebase project, stores information about application users, endpoint sensor devices, and temperature data. As a document database, the information is stored in collections of documents, where each document can contain sub-collections of associated documents [61]. This structure allows for easy and low-bandwidth data retrieval and storage. At the root level, there are collections of users and collections of endpoint devices. The user documents contain each user's associated sign-in information, desired temperature notification threshold, and associated endpoint devices. The endpoint device documents contain the registry and device IDs of the endpoint from Cloud IoT Core along with the current temperature and humidity received from them. The device documents also contain a sub-collection of days, each of which is populated with the timestamped temperature and humidity values received each minute they are in operation. This data can be easily exported to a BigQuery database in the future to implement machine-learning frost prediction, time allowing.

Firebase

The Firebase project associated with this design contains the Firestore database, along with user and authentication data received from the mobile app. As displayed in Figure 8 below, the Firebase project communicates with the app in several different ways. When the app is launched and a login is initiated, Firebase's authentication SDK is used to either register the user or sign them back in, associating their account on the backend with their user document in Firestore. When the app is open, it requests temperature and humidity every minute using the Firestore API, which sends the current temperature and humidity data associated with the user's device to the app for display. The app also uses this API to set the alert notification temperature threshold in the database. Finally, the Cloud Function uses the Firebase API to trigger frost alert notifications, which are sent on to the mobile app.

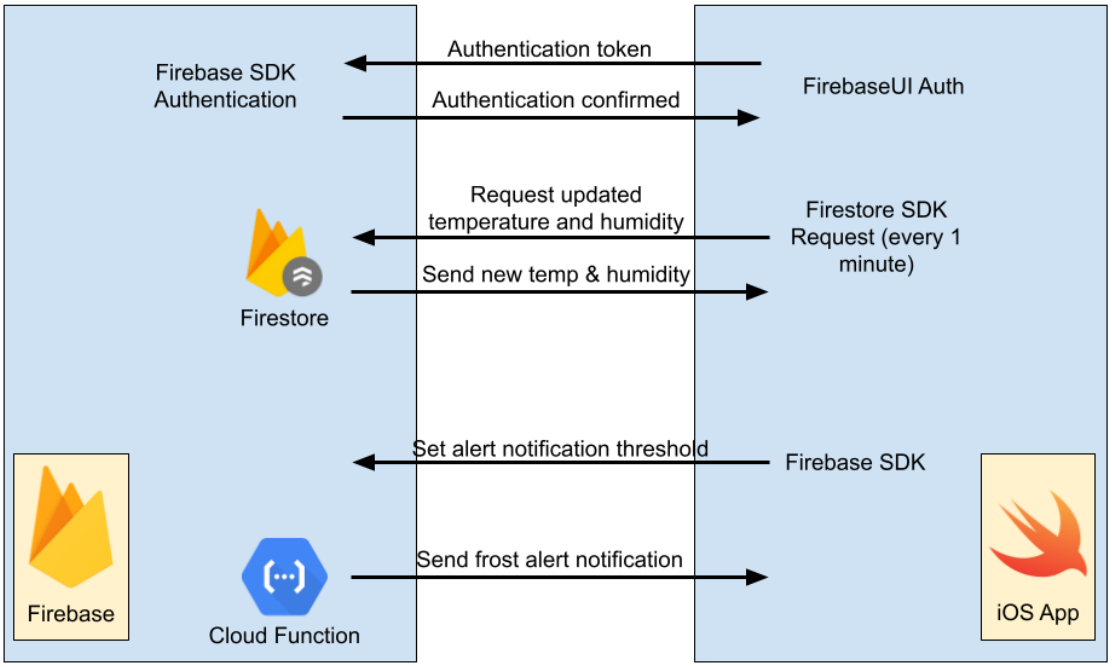

Figure 8: Firebase-App Communication Logic Diagram

iOS App

The iOS app, for which mockups are shown in Figure 9 below, utilizes the Firebase SDK with Apple's Swift language to control authentication and data flow. As pictured in the leftmost mockup, the initial page shown when the app is first launched is a sign-in sheet, allowing the user to authenticate with Firebase using either Google or Apple authentication tokens. The flow from here is not mocked up, but if the user chooses Google authentication, a Safari popup card is shown that allows the user to choose the Google account to sign in with and asks them to confirm whether to share their account information with the app. If the user chooses Apple authentication, a popover card (handled by an Apple API) is shown that confirms whether to share the email associated with their Apple ID or to use an anonymous email alias provided by Apple. Once the user is logged in, the second screen from the left in Figure 9 is shown, allowing the user to input the device ID of their sensor endpoint, along with its physical location. Once the user has entered a legitimate device ID that is registered in Firestore, the main screen (third from the left) is displayed. This screen displays the device's name and the most recent sensor data received from it, along with the weather data reported by NOAA for the address they have set for it, allowing easy comparison. This screen becomes the default app screen shown when the app is opened once the user has signed in. It also provides a button to open the app settings at the bottom. When this is pressed, the settings popover card (shown in Figure 9 on the far right) is opened, allowing the user to change their temperature display between Fahrenheit and Celsius and to set the notification temperature threshold, or to change the endpoint device (returning them to the second screen) or log out of their account (returning them to the first screen). Changes to the former two settings are saved to Firestore when the user presses "Done," or cancelled if they press "Cancel." Changes to the latter two are automatically saved.

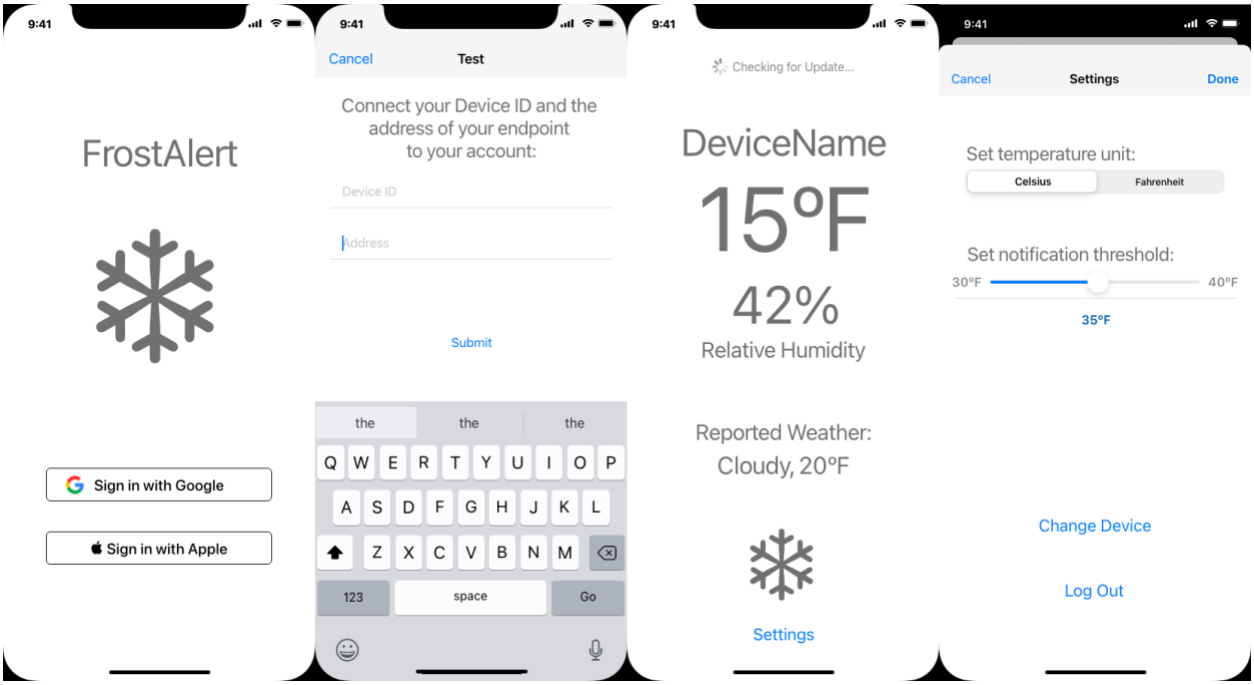

Figure 9: iOS App Design Mockups

Final Design and Implementation

The final design largely follows the specifications and processes as proposed in the preliminary design. The code for the endpoint was unmodified except for some minor performance improvements by reducing the number of libraries and extraneous variables used for testing but not necessary for the final design. Implementation details were finalized for the cloud components, code for the cloud functions was written and tested, and the iOS application was written and integrated with Firebase, all described in further detail here. At the high level, a Google Cloud project was initialized, and the region of operation chosen was us-central since it operates across multiple Google Cloud server farms and thus has higher uptime. This comes at a higher cost for certain operations, but this system largely operates within the free tier of Google Cloud, so this was not a concern.

Endpoint

The endpoint was built largely as described in the preliminary design. Pictures of the semi-constructed endpoint are shown in Figure 10 below. The final hole for an extension cable has not been drilled due to time constraints and a lack of access to an extension cable suitable for outdoor use, but the endpoint is otherwise finished.

Figure 10: Constructed Endpoint

Cloud Functionality

Cloud IoT Core

The Cloud IoT Core setup largely followed the preliminary design closely. A single device registry, *endpoints*, shown below in Figure 11, was configured with support for MQTT communications and telemetry data sent to the *weather2* Pub/Sub topic (detailed below). A single endpoint, *garden32*, was registered and authenticated as described in the preliminary design.

Registry ID ↑	Region	Protocol	Telemetry Pub/Sub topics
endpoints	us-central1	MQTT	projects/frost-alert-21/topics/weather2

Figure 11: IoT Core Registry Setup

Cloud Pub/Sub

The *weather2* Pub/Sub topic was generated with the service roles shown in Figure 12 below, giving it permissions to call and send data to cloud functions (the cloud build and cloud functions service agents) as well as to receive data from the IoT Core registry. A cloud function, *tempAlert*, was created to trigger on the receipt of new messages by the *weather2* topic, and the forwarding of message data to *tempAlert* was achieved by creating a push subscription to forward new messages and start running the cloud function automatically. The AVRO message schema planned in the preliminary design was scrapped because it would require bloating the program size at the endpoint, as it requires the use of serialized JSON objects for message bodies. If this system were to be commercialized, an enforced message schema would be advisable to prevent potential nuisances caused by hackers inserting bad data into the database through modified endpoints or driving up costs by overloading the message size and rate, but, given that a private key is already required to connect to IoT Core, an enforced schema is unnecessary overhead for a system configured and operated by an individual.

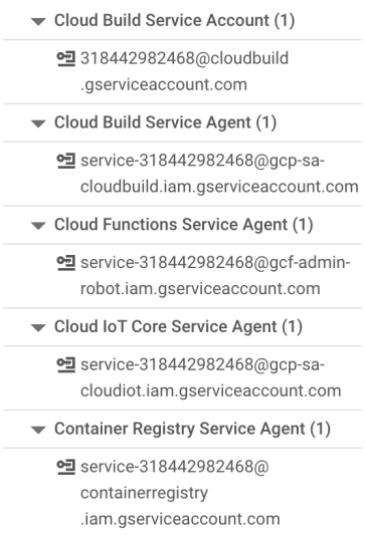

Figure 12: Cloud Pub/Sub Service Roles

Cloud Functions

The primary cloud function, *tempAlert* (simplified pseudocode for which is shown below in Figure 13, and full code for which is in the appendix), is written in the Python 3.7 runtime of Cloud Functions and triggered by new message events on the *weather2* topic. It is allocated to use the minimum amount of memory available in Cloud Functions, 128 Mebibytes (about 134 Megabytes), to minimize potential expenses related to running the function. Similarly, the timeout is set to the recommended minimum of 60 seconds to prevent potential errors causing the function to run overtime and add expenses. The number of maximum instances is set to 2 for the same reason. This configuration limits the system to a maximum of only two endpoint devices operating at a time, but this could be easily changed as needed for a system with more endpoints or a commercialized version of the system. The *tempAlert* function is executed using the *firebase-admin* service account tied to the Firebase portion of the project to allow it to send notifications and read to and write from the Cloud Firestore database.

```
def messageTriggerHandler(event):
      pubsub_message = decode(event)
      deviceID = event['messageID']

      endpoint_document = getFirestoreDocument('endpoints/%s', % deviceID)
      uid = endpoint_document.get('userID')

      user_document = getFirestoreDocument('users/%s', % uid)
      fcm_token = user_document.get('fcm_token')
      threshold_temp = user_document.get('theshold_temp')

      if pubsub_message[0] == '{':
            string_array = pubsub_message.strip('{').strip('}').split(';')
            tempf = float(string_array[0].strip("temp:"))
            humf = float(string_array[1].strip("hum:"))

            H = (math.log10(humf)-2)/0.4343 + (17.62*tempf)/(243.12+tempf)
            Dp = 243.12*H/(17.62-H)

            if ((Dp < 0) or (tempf < threshold_temp)):
                  message = messaging.Message(
                    notification=messaging.Notification(
                        title='Incoming Frost Detected!',
                        body='Prepare your garden for an incoming frost!',
                        ),
                    token=fcm_token,
                  )
                  response = messaging.send(message)

            values = {
              "current_hum": humf,
              "current_temp": tempf,
              "user": uid
            }
            endpoint_document.set(values)
```

Figure 13: tempAlert Cloud Function Pseudocode

The simplified code in Figure 13 captures the function's logic. The function is triggered by a new message event from Pub/Sub, and that trigger passes in event data. The body of the event is decoded to get the proxied message from the endpoint, which contains the data about temperature and humidity. The device ID of the endpoint is also stripped from the event's header. This is then used to get a reference to the endpoint's document in the Firestore database, from which the owner of the endpoint's user ID (*uid*) is extracted. A reference to the user's document is also created, and from that their Firebase Cloud Messaging token (*fcm_token*) is extracted along with their notification temperature threshold (*threshold_temp*). The function then checks if the message body starts with an open brace, indicating whether it contains data or is just a connection start message. If it does not start with a brace, it is the latter, and the function ends with no further actions taken. If it does start with a brace, then the message body is stripped of its starting and ending braces and split into the *string_array*. The temperature and humidity values are obtained by stripping their definitions from the array and converting them back to floating points. The current dew point (*Dp*) is then approximated using a formula from the humidity sensor's datasheet: [64]

$$H = \frac{\log_{10}(RH) - 2}{0.4343} + \frac{17.62 * T}{243.12 + T}$$

$$Dp = \frac{243.12 * H}{17.62 - H}$$

where *RH* is the percentage relative humidity percent and *T* is the temperature in degrees Celsius.

If the dew point is calculated to be below freezing (0º Celsius), or if the measured temperature is below the user's *threshold_temp*, a Firebase Cloud Message is prepared. A notification for the message is configured within the *Message* instantiation method that alerts the user of the likelihood of an imminent frost. No action is associated with the message, so the notification simply opens the app if tapped. The user's *fcm_token*, obtained earlier, is used in the *Message* instantiation to indicate the user that should receive a push notification. After the message is instantiated, it is sent using the Firebase Cloud Messaging (FCM) library. Finally, whether or not the message send was triggered, the new temperature and humidity values are wrapped back into a dictionary, along with the owner's user id (necessary because it was part of the original document and it needs to be preserved). This dictionary is set as the new endpoint document so that the user can get the latest temperature and humidity data on their mobile device.

One other cloud function, *newUser*, is configured for the project. It uses the Node.js 12 runtime and is triggered by the *user.create* trigger in Firebase. When a new user signs up for the app using their Google account, a new Firebase Auth user profile is created for them. The profile can include several fields, but as configured it only identifies them using a unique ID (*uid*). When a new profile is created, it sends the *uid* as part of the trigger, which is then handled by the cloud function. The code for *newUser*, which is quite short, is presented in Figure 14 below:

```
1  const functions = require("firebase-functions");
2  const {Firestore} = require("@google-cloud/firestore");
3  const admin = require("firebase-admin");
4  const firestore = new Firestore();
5  admin.initializeApp();
6
7  exports.newUser = functions.auth.user().onCreate((user) => {
8    // ...
9    const uid = user.uid;
10   const docRef = firestore.collection("users");
11   const document = docRef.doc(uid);
12   console.log(uid);
13   document.set({
14     uid: uid,
15     endpoint: "",
16     threshold_temp: 2.0,
17     fcm_token: "",
18   });
19 });
20
```

Figure 14: newUser Cloud Function

After importing some Firebase and Firestore libraries, an environment Firestore admin client is created on line 4, then the Firebase admin account is initialized on line 5. The *newUser* function is defined between lines 7 and 19, and is tied to the new user account created, whose ID is obtained for use on line 9. A reference to the Firestore user collection is created on line 10, and a new user document is added to that collection on line 11 with a document ID matching their user ID. On lines 13 to 18, this document is modified to contain the user's ID, a blank entry for their endpoint device that they can register later in the app, a default notification threshold of 2 degrees, and another blank entry for their FCM token, which again can be registered later in the app. At the end of the *newUser* function, the database has been updated to allow for communication and synchronization with the mobile app.

Cloud Firestore

The Cloud Firestore database is configured largely as described in the proposed design. At the root level, it contains two collections: one for users and one for endpoints. The user documents contain the unique user ID (uid), cloud messaging token, and device ID of the endpoint registered to that user, along with their notification threshold temperature. The endpoint documents contain the current humidity and temperature values reported by their sensors, as well as the *uid* of the user to whom the endpoint is registered. Historical temperature and humidity data storage was not implemented because it could start to drive up the cost of cloud hosting over time; if it were to be implemented, a cheaper database like Cloud Datastore could easily be configured to receive and this data for future reference.

Firestore databases use a set of security rules defined in a simple JavaScript-like schema to allow or deny incoming read and write requests [65]. The security rules for this design, shown below in Figure 15, are configured to allow users to only read and write to the user and endpoint documents registered to their accounts. If an endpoint has not yet been registered, any authorized user can claim it, but after it has been registered only the user who it is registered to can access it.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /endpoints/{endpoint} {
      allow read,write, update: if request.auth != null && (request.auth.uid == userId || userId == "");
    }
    match /users/{userId} {
      allow read, write, update, delete: if request.auth != null && request.auth.uid == userId;
    }
  }
}
```

Figure 15: Cloud Firestore Security Rules

Firebase

The other Firebase services used in this design are Firebase Auth (an end-user authentication system that supports most federated identity platforms) [66] and Firebase Cloud Messaging (FCM), a cross-platform notification service. Both of these services interact with the iOS app, whose bundle ID and Apple Developer credentials are registered with Firebase. Firebase Auth is set up to allow registration and sign-in only through the Sign in with Google API, and the only data it collects about the user is their email address.

Firebase Cloud Messaging is configured to send push notifications to the iOS app through the Apple Push Notification Service (APNS). This service, run by Apple, receives authenticated message bundles from Firebase and proxies them to the iOS app, where they appear as push notifications. To authenticate the Firebase project with APNS, an APNS developer certificate was created on the Apple Developer console, and an APNS authentication key linked to that certificate was set up. The authentication key was uploaded to the FCM backend, enabling Firebase to send notifications using cloud functions to any iOS client with the FrostAlert app installed.

iOS App

The final iOS app design was programmed in Swift, with a graphical user interface (GUI) written in Apple's new SwiftUI toolkit. SwiftUI utilizes a modern declarative UI paradigm, wherein all UI elements are declared within structures called views that also contain or reference all associated state data directly, allowing for simplified UI construction and interaction [67]. At the top level, an *app* structure contains the *ContentView*, an overarching view structure that contains all other views and the programmatic logic that controls switching between them. The *app* structure also contains an adaptor to interface with an *AppDelegate* class, which enables some extra, non-declarative behaviors that are still necessary to use since the new GUI paradigm is still under development. It also contains global variables and objects that need to be passed to all views underneath it in the structure. The breakdown of the iOS app design in this section will only cover a high-level explanation of the app's structure and internal functionality because many of the finer details require more conceptual knowledge of SwiftUI than can reasonably be explained here.

Several functional changes were made to the app compared to the proposed design due to time constraints and to better fit with the SwiftUI data model. The sign-in screen was simplified to only allow Sign-in with Google, removing the planned Sign-in with Apple option because enabling Sign-in with Apple requires additional developer agreements with Apple. The device registration screen was not implemented because it added unnecessary complexity; instead, the device ID was added as a setting on the settings pane, as shown below in the fourth screenshot in Figure 16. The device location and current NOAA-reported weather information were also scrapped due to the difficulty of integrating the public NOAA API and the added expense required to implement an easier to use commercial weather API. The last major functional change was the addition of a "Get Notifications" button to the settings, which synchronizes the device's FCM token with their user document in Firestore. This is necessary for the *tempAlert* cloud function to send frost warning notifications to the user.

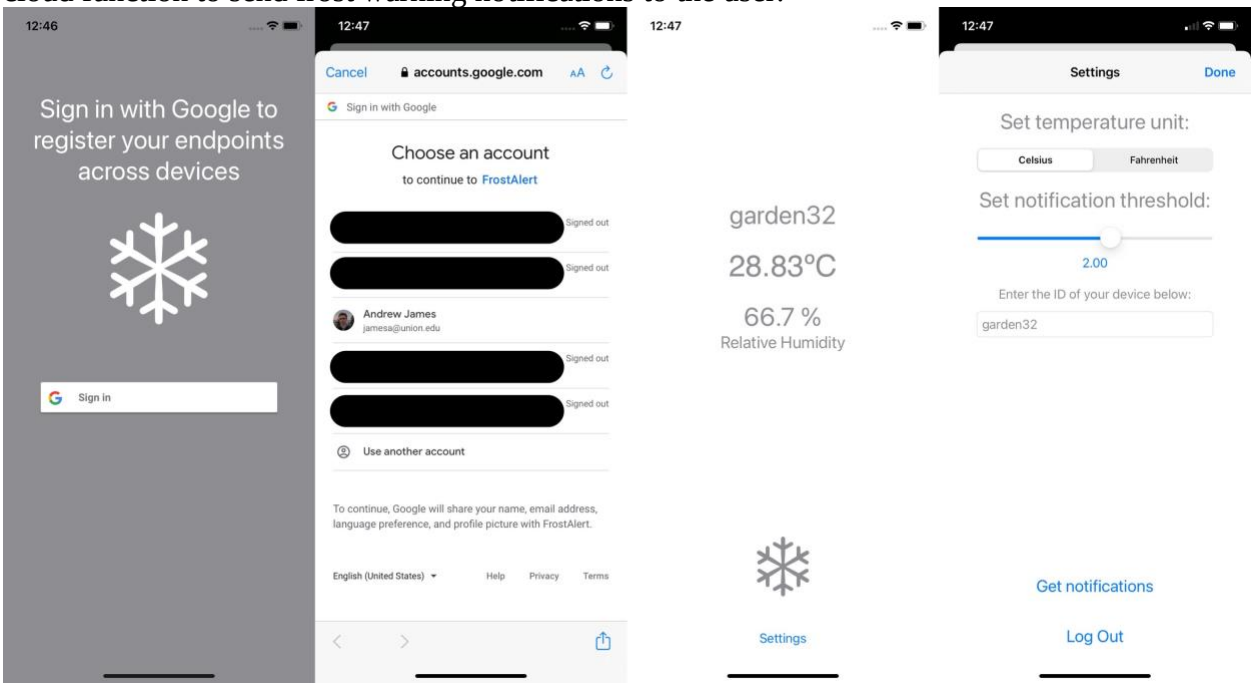

Figure 16: Screenshots from final iOS app design

Firebase Integration

Integration with Firebase in the FrostAlert app is achieved primarily through the use of Google-maintained libraries. The installation and updating of these libraries use a Swift dependency manager called CocoaPods [68]. Information about the specific Firebase project that the app is tied to comes from a file called GoogleService-Info.plist, downloaded from the Firebase console, which is then initialized using the *configure* function from the primary Firebase library in the *AppDelegate* class. The *AppDelegate* class also contains the code to initialize the Sign-in with Google library, handle the callback URLs from the Google sign-in page, and handle incoming notifications from FCM. The callback URLs for the app, an iOS feature that enables the passing of authentication tokens and other data between different apps and services, are also registered in the app's build settings so that devices with the app installed automatically forward all calls to those URLs to the app for handling. Thus, when a WebKit (Safari instance) pane is called for the user to sign in with Google, they are automatically redirected back to the app after authentication is confirmed.

The notification handlers in the *AppDelegate* class are mostly configured with the default settings suggested by Google, but modifications were made to support updates to SwiftUI notifications in iOS 14. The notifications as configured display a banner on screen that must be manually dismissed, play a sound (or vibrate the phone if it is in do not disturb mode), and show a red badge on the app icon on the homescreen. This is the most intrusive notification style currently possible on iOS aside from a true alarm, which must be pre-programmed for a specific time and cannot be delivered on the fly, making it a non-starter for this app.

Internal Data Model

Beyond the SwiftUI views, which are discussed in the following section, the internal data model of the FrostAlert app is based on several other classes and structures (which behave like simplified classes in Swift [69]) that are used to contain state data and communicate with the Firestore database. At the top level, information about user authentication and the user's FCM token is stored in a class called *SessionStore*, which contains a static instance of its own type that is used to allow access to the class from the *AppDelegate* portion of the code. The FCM token associated with the static *SessionStore* instance is assigned to it by the *AppDelegate* class when the user first starts the app and is automatically updated if the token changes. *SessionStore* also contains methods to quickly check if the user is logged in and to listen for changes to the login state and automatically execute code based on those changes. When the user logs in, the latter function sets its published *User* document (a public variable that automatically updates all functions that use it when it changes) to contain the user's sign-in details. When they log out, the *User* document is destroyed, and its pointer is reset to a nil value.

Data from communication with the Firestore database is stored in a top-level class called *DBDocuments*, which conforms to the ObservableObject protocol in SwiftUI. This protocol makes the class a publisher, meaning that every time one of its instance's published properties (like the *User* property from *SessionStore* described previously) change, those changes are published to all other functions and views that make use of the *DBDocuments* instance itself. The class contains three properties: the device ID, stored as a String, and codable objects (of custom types *DBUser* and *Endpoint*) for the user and endpoint documents associated with the app's user when signed in. Since the *DBuser* and *Endpoint* objects are codable, they are easily mapped to

exactly replicate the documents they are associated with in Firestore when both reading from and writing to the database.

To this end, the *DBDocuments* class also contains all methods for communicating with the database: *loadDBUser*, which takes the user's *uid* string from Firebase Auth and uses it to load their preferences from Firestore with a listener that automatically keeps them in sync; *loadEndpoint*, which uses the endpoint's deviceID to do the same; *changeTempThreshold* and *setDevice*, which update the user's temperature notification threshold and registered endpoint in the database, respectively; and *setFCMToken*, which takes the FCM token from *SessionStore* and adds it to their database entry. All of these functions and their inputs are called from views, following the standard SwiftUI model, and the instance of *DBDocuments* is stored in the top SwiftUI view (*ContentView*) so that state and UI are kept automatically synchronized.

SwiftUI Views

The top-level view, *ContentView*, is declared in the *app* structure, and supplied on creation with a binding link to the static *SessionStore* instance. It initializes with state variables for the user's notification threshold, their preference for displaying degrees in Celsius or Fahrenheit, their endpoint's device ID, the instance of *DBDocuments* used for cloud synchronization, and a toggle to show the settings screen, which starts out off. When the view first loads, it triggers the *SessionStore.listen* function, which detects whether the user is logged in. If the user is not logged in, another view called *SignInView* is initialized and displayed; once they are logged in, the app's main screen (the third screenshot in Figure 16 above) is displayed. The main screen sub-view has a trigger when it loads to call the *DBDocuments.loadDBUser* function, since the *uid* is now available. This function also calls the *loadEndpoint* function when it finishes, so that by the time the main view is displayed the *DBUser* and *Endpoint* documents tied to the *DBDocuments* instance (if the endpoint is registered) are fully populated.

Once the documents tied to Firestore are loaded, the data associated with them is displayed on the main screen sub-view using standard iOS text elements. Since these documents are published, every time the Firestore server pushes updates to them the text displays are updated immediately by SwiftUI. The "Settings" button at the bottom of the main screen is tied to the state *settingsIsShowing* Boolean variable, and when tapped sets it to true. This variable is tied to a *.sheet* popover display controller, which passes state variables to and shows the *SettingsView* (shown in the last screenshot in Figure 16 above) when it is set to true. All of the elements in the *SettingsView* are standard SwiftUI elements whose inputs are bound to the state variables associated with them. Thus, when the user sets the unit to display temperature in, all temperature displays throughout the app, including the display of the notification threshold, are set to that unit. Similarly, when they update the notification threshold temperature or the device ID to register, these changes are automatically propagated into the associated data models and synchronized with Cloud Firestore. The "Get notifications" button calls the *DBDocuments.setFCMToken* method so that the FCM token for their device is registered, and the "Log Out" button calls the Firebase Auth API to log out of their account. Since the sign in status is also a state variable, this automatically brings the user back to the top level of *ContentView*, where they are forwarded to *SignInView* since *SessionStore* no longer has a reference to an authenticated user account

The *SignInView*, shown in the first screenshot in Figure 16, is a very simple structure, whose only elements are the title text, a snowflake symbol, and the Sign in with Google button. This button is a standardized element from the GoogleSignIn library for Swift, but was written

for the old UIKit GUI model, so it is wrapped in a *UIViewRepresentable* structure that converts these old UI elements to work with SwiftUI. When the button is tapped, it opens a WebKit popover (shown in the second screenshot in Figure 16) prompting the user to choose one of their Google accounts to sign in with. When they tap an account, the Google sign in page calls the internal URL discussed before with details about their authentication information, triggering the authentication code in the *AppDelegate* to run and thus updating *SessionStore*. This causes *ContentView* to update, switching the user to the main screen, completing the login cycle.

As discussed previously, this is only a high-level explanation of the FrostAlert app's functionality. Understanding the final implementation details requires a significant degree of knowledge about Swift programming that is beyond the scope of this paper. However, interested readers are encouraged to review the full Swift code, linked to at the top of the appendix, to gain a fuller understanding of the app's structure and design.

Performance Estimates and Testing Results

SHT31 Temperature and Humidity Sensor

I^2C communication with the SHT31 and reading of temperature and humidity values was tested using the default examples that come with the DFRobot_SHT3x Arduino library [56]. I^2C communication worked after changing the address of the sensor to 0x44 from the default 0x45 for the master/slave configuration. Readings came through as expected, but the library does not convert correctly from Fahrenheit to Celsius due to a C++ syntax issue. I forked the repo on GitHub [62] to correct the issue, which resulted from an improper mixing of floating point and integer values in C++, and filed a pull request for the company to correct the issue. After fixing this issue, the temperature read accurately in both Fahrenheit and Celsius, with minimal difference in value ($\leq$0.5C) compared to a reference HTC-1 temperature sensor. The humidity sensor in the SHT31 also read accurately when compared to the HTC-1's onboard humidity sensor, with $\leq$5% difference, and generally even closer. Given that the HTC-1 has higher error tolerances for both temperature and humidity than the SHT31, it is likely that the latter's values are actually closer to the real ambient temperature and humidity, but without access to a high-grade thermometer and hygrometer to compare with it is difficult to prove this.

Adafruit FeatherWing Ethernet Shield

Ethernet connectivity through the Adafruit Ethernet FeatherWing was tested using the Arduino WebClient example [63], which is automatically configured for the WIZ5500 ethernet chip onboard the FeatherWing. Several changes had to be made before connection could be established; first, the MAC address variable had to be changed to match the MAC listed on the chip itself; second, the Ethernet object had to be initiated on the correct pin (33 for the HUZZAH32); finally, the web address requested by an HTTP GET was changed from the default www.google.com/search?q=arduino to wifitest.adafruit.com/testwifi/index.html to simplify the data returned, as the Adafruit server simply informs that the connection was successful without returning a full HTML page (the full HTML page was returned by Google, but was not parsed further by the HUZZAH32, making it difficult to read). No further issues were experienced with the ethernet setup.

MQTT Connection with Google Cloud IoT Core

After researching different potential MQTT libraries for the ESP32 on the Arduino platform, the arduino-mqtt library published by Joël Gähwiler [60] was chosen for interfacing with the cloud due to its high level of documentation and interoperability with the Google Cloud IoT JWT library [59] for generating JSON web tokens to authenticate with Cloud IoT Core. Before MQTT testing could begin, the Google Cloud components (IoT Core and Pub/Sub) had to be configured to allow for proper communication. First, a weather topic was created in Pub/Sub with a corresponding AVRO schema specifying temperature and humidity as floating-point values to ensure received messages were of the correct types. A private/public RS256 keypair was generated and the public key registered to a device in IoT Core representing the ESP32 endpoint. For MQTT testing, a QoS level of 1 (requiring acknowledgement of published messages via PUBACK) was set to avoid any ambiguity as to whether the Google Cloud server was receiving messages. Testing was initially conducted using the HUZZAH32's onboard WiFi with an example test program provided in the Google Cloud IoT JWT library, but this exposed an issue with the current release of the ESP32 WiFiClientSecure library not verifying SSL certificates correctly. After some research, an earlier version of this library that was identified to be functional was found, provided by GitHub user debsahu in a demo project for the ESP32 using MQTT. This earlier library revision was substituted into the IoT Core test code, and an MQTT connection was established successfully with Google Cloud IoT Core.

Weeklong endpoint test

The last major test of the endpoint software and hardware was to leave it running for a little over a week and use the Google Cloud console to look for any errors. As expected, the JSON web token refreshed every 20 minutes, causing a reconnection to the MQTT server each time. It still succeeded in delivering a temperature and humidity update in the same minute despite needing to reconnect. However, there was one minute every day, immediately after local midnight, that messages were not delivered. Upon investigation, this was being caused by the internet router used refreshing its DHCP leases at that time, and the endpoint was successful in getting reconnected after these brief outages.

iOS App

The iOS app was continuously tested throughout its development using the iOS simulator built into XCode and the LLDB debugging tool. By running the project in the descriptive debug mode, all potential issues caused by bad state storage were identified early on. Since SwiftUI is declarative, all possible state outcomes are easily checked by modifying variables in the debug mode, and this was used to ensure that all data calls with uncertain outcome (especially to Firestore) were guarded with error catching code. Final testing of the iOS app was performed on an actual iPhone, which was required in order to test push notifications (as the simulator cannot receive push notifications from an external server). The app performed essentially as expected, with the unpredicted but useful bonus that it automatically switched color scheme, as shown in Figure 17 below, from white background and dark gray text to a black background and light gray text when the phone switched into dark mode.

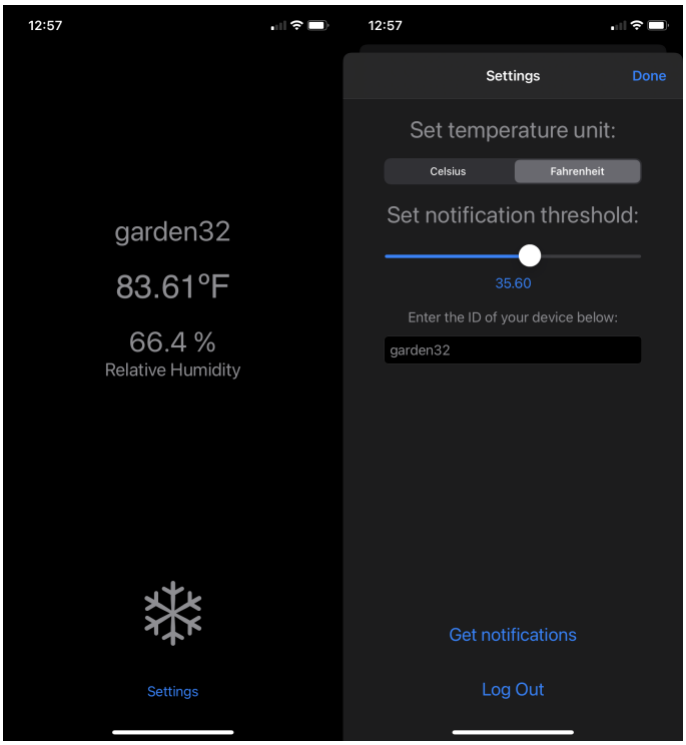

Figure 17: App in Dark Mode

Production Schedule

The production schedule for this project can largely be broken down into three phases: research and initial design, development and implementation, and refinement. These phases overlapped because different parts of the project were completed at different times. The final schedule that was actually followed is shown below in Tables 9 and 10. The first four weeks of winter term were focused entirely on researching the design alternatives presented previously and deciding on components and services to use. Starting in week 5, development and initial testing of the endpoint microcontroller and sensors began, and this transitioned into connecting the microcontroller to the cloud in week 7. Co-development and refinement of both the endpoint code and cloud service configuration continued through the first two weeks of spring term, after which all focus shifted to development of the iOS app and integration of it with the cloud. The refinement phase for the app began in week 7 and continued throughout the rest of the term, with small tweaks improving performance made in week 10.

The order the project was developed in largely tracks with the flow of data through the system, from the endpoint through the cloud and finally to the iOS app. This proved to be a reasonable way to develop the system, as each service and step in the data pipeline could be tested using real data from the last step rather than having to develop separate tests. If this project had been undertaken by a group, it would likely have made more sense to develop both the endpoint code and iOS app at the same time and using test inputs in the cloud until the endpoint was connected.

The main time sinks that caused delays in the planned production schedule had to do with library incompatibilities that had to be worked around and poor documentation. For the endpoint in particular, implementing ethernet connectivity through the Ethernet FeatherWing with the

ESP32 proved challenging, as the default SSL library for the ESP32 was incompatible with the WIZ5500 chip in the FeatherWing. This was resolved by using a third-party library called OpenSSL, but the documentation for it was inscrutable and the developer had to be contacted for clarification on its use. Similar issues occurred when trying to use some of the older Firebase Swift library features with SwiftUI; these were resolved by using the *AppDelegate* class, which was an inelegant, poorly documented, and non-obvious solution that caused additional production delays. Despite these delays, however the project was completed (aside from the stretch goal of adding ML frost prediction) by the end of week 10 of spring term.

Table 9: Winter Term Implementation Schedule

Task	Week									
	1	2	3	4	5	6	7	8	9	10
Research and decide whether to use an app or a text alert										
Research cloud hosting vs direct communication										
Write student research grant proposal										
Research and decide on a communication standard										
Research and select a microcontroller										
Research and select sensors and other parts										
Research and select outdoor case										
Decide on power supply										
Test microcontroller ethernet functionality										
Research and select a cloud service										
Test sensor functionality										
Design IoT endpoint software flow										
Test MQTT Functionality										
Connect sensor data to cloud										
Design app UI and functionality										
Create and present final presentation										
Work on design report										

Table 10: Spring Term Implementation Schedule

Task	Week									
	1	2	3	4	5	6	7	8	9	10
Connect endpoint ethernet to MQTT/Cloud IoT Core										
Set up Firebase project										
Connect Pub/Sub to Firestore										
Configure security and login parameters in Firebase										
Set up iOS app with Firebase SDK										
Develop iOS app logic and cloud messaging										
Refine iOS app & finish cloud messaging infrastructure										
Create and present final presentation										
Write design report										
Stretch goal: setup BigQuery for ML frost prediction										

Cost Analysis

The overall cost of building and operating this design is broken down in Table 11 below. The total cost to construct the IoT endpoint is $86.55, well under the $200 goal, but the yearly operating cost turned out to be $99.60 because of the expensive Apple Developer account that is necessary in order to send notifications to iOS apps. If the app were reimplemented on Android or Firebase was reconfigured to send emails rather than push notifications to iOS devices this cost would be eliminated, driving the yearly cost down to just $0.60, but the existing model is the only way to send reliable notifications to iOS users. This cost would be negligible if the app were commercialized, however.

Table 11: Final Cost Breakdown

Item	Supplier	Price (USD)
Endpoint (Fixed Costs)		
DFRobot SEN0385 - SHT31 Temp/Humidity Sensor	DigiKey	19.90 [17]
Adafruit HUZZAH32 w/ Loose Headers	DigiKey	19.95 [24]
Adafruit Ethernet FeatherWing	DigiKey	19.95 [25]
Qualtek QFAW-05-05	DigiKey	6.15 [33]
Bud Industries NBF-32110	DigiKey	20.60 [39]
	Subtotal	86.55
Subscriptions (Recurring Costs)		
Google Cloud - Estimated usage for one user		0.05/month
Apple Developer Account - Billed yearly		99.00/year [70]
	Subtotal (monthly)	8.30/month
	Subtotal (yearly)	99.60/year
	Total cost of endpoint and 1 year of operation	186.15

User Manual

Starting with a constructed endpoint that has been registered with Firebase, attach the endpoint to a wooden post at the lowest point in the garden using the mounting screws and brackets that come with the housing, with the temperature sensor and openings for ethernet and power facing down to prevent rain damage.

Open the endpoint cover and connect an ethernet cable of appropriate length between your router or access point and the endpoint, being sure to route the ethernet cable through the appropriate hole on the bottom. Next, connect the internal power brick to an extension cable through the appropriate hole in the bottom, then close the endpoint cover, latch it, and connect the other end of the extension cable to power. The device will connect to the network automatically when used with ethernet, so no further steps are necessary to get it working.

Once the endpoint is plugged in, download the FrostAlert mobile app to your iPhone and open it. On your first run, the app will ask to send notifications; tap "Allow" to be alerted when a frost is incoming. Next, sign in with your Google account to get started. Once logged in, tap the blue "Settings" button at the bottom of the screen. On the settings page, tap in the text box labeled "Device ID" and type in the ID your endpoint is registered under, then press "Return" to register the device to your account. Choose whether you would like to view the temperature in Celsius or Fahrenheit with the appropriately labeled buttons, then set the slider beneath to the temperature that you want to receive frost alert notifications at. Finally, tap "Get Notifications" to register your device for frost warning notifications, then tap "Done."

At this point your setup is complete. When a frost is detected as likely to occur, your phone will be notified. You can also view the temperature and humidity in your garden at any time by opening the FrostAlert app, which will update automatically. If at the end of the season you wish to decommission your endpoint, simply unplug it from power and ethernet, unscrew it from the mounting brackets, and store it indoors until you are ready to use it again. It will remain registered with the cloud, so you can simply put it back up and plug it in when it is needed again.

Discussion, Conclusions, and Recommendations

The initial goal of this project was to alert home gardeners about unexpected frosts to give them time to cover their crops and reduce potential frost damage. To this end, an internet-of-things system was devised, with an endpoint located in the garden to measure and report the hyperlocal temperature and humidity conditions. Based on meteorological science, these two variables are used to predict the dew point, which indicates the likelihood of frost as well as further dips in temperature if it is below freezing. This data is reported to an MQTT broker, Google Cloud IoT Core, which passes the data to a universal cloud messaging system, Google Cloud Pub/Sub. Pub/Sub triggers a Cloud Function on the receipt of new messages, which calculates the dew point and updates a Cloud Firestore database, sending a frost warning notification to an iOS app. Using information about the owner of the endpoint stored in Cloud Firestore, it targets these notifications to the specific user that has requested to receive them.

The iOS app, FrostAlert, allows the user to sign into the system with their Google account, and then to register an endpoint to their account to receive notifications about. They can also view the current temperature and humidity in their garden, as reported by the endpoint, any time they open the app, and the data is synchronized with Cloud Firestore in real time. The app also allows them to set a custom frost alert temperature notification threshold, which is also synched with the server, and to display temperature units in either Fahrenheit or Celsius.

Based on the goal of creating a system for home gardeners to be alerted when a frost is approaching in their garden, this project has been a reasonable success. It has high uptime, little required maintenance, and a reasonably low cost to build. The main concern is the high cost of the yearly Apple Developer account, which is required to enable push notifications on iOS. This cost would be much more reasonable if the project were commercialized; with as few as 100 users paying a dollar or two per year in subscription fees, the cost of the Apple Developer account, as well as the Firebase and Google Cloud fees for usage, could easily be covered. The endpoint design would need to be significantly overhauled to make it manufacturable as a consumer product, however, and manufacturing costs would need many more sales to offset.

As an open-source project, the system provides an easy platform for an enterprising gardener with a software development background to customize the system for their own use. Future developments that would increase the flexibility of the system could include an Android app, which would open up the system to many more potential users (and be much cheaper for hobbyists to implement for themselves), or the construction and development of a secondary endpoint that could be placed in a convenient location in the home and set up with an alarm to notify users who may not always have their phones to hand. A Raspberry Pi would be a good contender in that context, as it could run a simple server that could receive Firebase Cloud Messages and play an alarm sound when received. Both of these options would cut down significantly on yearly operating costs, but the latter will likely be the next stage of official development of the project.

References

[1] M. Herrmann, "The Modern Day Victory Garden," *Procedia Engineering*, vol. 118, 2015. [Online Serial]. Available: https://doi.org/10.1016/j.proeng.2015.08.498.

[2] D. Galhena, R. Freed, and K. Maredia, "Home gardens: a promising approach to enhance household food security and wellbeing," *Agriculture & Food Security*, vol. 2, no. 1, 2013. [Online Serial]. Available: https://doi.org/10.1186/2048-7010-2-8.

[3] F. Brown, "How to protect plants from frost," *Marin Master Gardeners*. http://marinmg.ucanr.edu/Our_Projects/Leaflet/How_to_protect_plants_from_frost.

[4] "HOBO Family of Weather Monitoring Solutions," *Onset*. https://www.onsetcomp.com/products/data-loggers/weather-stations/.

[5] M. Longstroth, "What is the difference between a frost and a freeze?" *Michigan State University Extension.* https://www.canr.msu.edu/news/what_is_the_difference_between_a_frost_and_a_freeze

[6] L. Slattery, "Frost vs Freeze," *Iowa State University Extension and Outreach*. https://www.extension.iastate.edu/linn/news/frost-vs-freeze.

[7] F. Allhoff and A. Henschke, "The Internet of Things: Foundational ethical issues," *Internet of Things*, vol. 1-2, August 18, 2018. [Online Serial]. Available: https://doi.org/10.1016/j.iot.2018.08.005.

[8] "The ultimate IoT security best practices guide," *Amazon AWS*. [Online]. Available: https://pages.awscloud.com/rs/112-TZM-766/images/IoT_Security_Best_Practices_Guide_design_v3.1.pdf.

[9] "Device Security," *Google Cloud IoT Core*. [Online]. Available: https://cloud.google.com/iot/docs/concepts/device-security.

[10] A. Awalt, "IP Code," *Digi-Key*, Sep. 10, 2018. [Online]. Available: https://www.digikey.com/en/blog/ip-code.

[11] J. Jacobi, "IP ratings explained: What those codes tell you about how well a device is protected from water and dust," *TechHive*, Feb. 11, 2020. [Online]. Available: https://www.techhive.com/article/3518536/ip-ratings-explained.html.

[12] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0," *Oasis*, March 7, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[13] "Publishing over the MQTT bridge," *Google Cloud IoT Core*. [Online]. Available: https://cloud.google.com/iot/docs/how-tos/mqtt-bridge.

[14] "Device communication protocols," *AWS IoT Core Developer Guide*. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html.

[15] "Communicate with your IoT hub using the MQTT protocol," *Azure IoT Hub Documentation*, Oct. 12, 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support.

[16] "I^2C-bus specification and user manual," *NXP*, rev. 6, Apr. 4, 2014. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[17] "DFRobot SEN0385," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/dfrobot/SEN0385/13590873.

[18] "Seeed 101990561," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/seeed-technology-co-ltd/101990561/10451874.

[19] "Adafruit 393," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/adafruit-industries-llc/393/5356714.

[20] "DFRobot SEN0227," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/dfrobot/SEN0227/7897986.

[21] "Adafruit 1293," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/adafruit-industries-llc/1293/5356796.

[22] "Amphenol T9602-3-A-1," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/amphenol-advanced-sensors/T9602-3-A-1/5027897

[23] "Raspberry Pi 4," *Raspberry Pi*. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

[24] "Adafruit 3405," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/adafruit-industries-llc/3405/7244967

[25] "Adafruit 3201," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/adafruit-industries-llc/3201/6165788

[26] "Adafruit 2821," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/adafruit-industries-llc/2821/5775536

[27] "Arduino MKR WIFI 1010," *Arduino Store*. [Online]. Available: https://store.arduino.cc/usa/mkr-wifi-1010

[28] "Arduino MKR ETH Shield," *Arduino Store*. [Online]. Available: https://store.arduino.cc/usa/mkr-eth-shield

[29] "DFRobot DFR0162," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/dfrobot/DFR0162/6588568

[30] "DFRobot DFR0164," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/dfrobot/DFR0164/7597132

[31] "Overview," *Adafruit HUZZAH32 Guide*. [Online]. Available: https://learn.adafruit.com/adafruit-huzzah32-esp32-feather

[32] "Phihong PSAA05A-050QL6-R," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/phihong-usa/PSAA05A-050QL6-R/6560437

[33] "Qualtek QFAW-05-05," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/qualtek/QFAW-05-05/6412289

[34] "Mean Well USA GS05U-USB," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/mean-well-usa-inc/GS05U-USB/7703346

[35] "Q Series," *Boxco*. [Online]. Available: http://www.boxco.eu/productView.do?page=en/products&CATE_CD=300001&P_CATE_CD=100001&MENU_CD=200124&PROD_NM=Q%20Series

[36] "Boxco BC-AGP-112110," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/boxco/BC-AGP-112110/13419847

[37] "R Series," *Boxco*. [Online]. Available: http://www.boxco.eu/productView.do?page=en/products&CATE_CD=300002&P_CATE_CD=100001&MENU_CD=200124&PROD_NM=R%20Series

[38] "Bud Industries NBB-15240," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/bud-industries/NBB-15240/428995

[39] "Bud Industries NBF-32110," *Digi-Key*. [Online]. Available: https://www.digikey.com/en/products/detail/bud-industries/NBF-32110/2328548

[40] "Azure IoT Hub pricing," *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/iot-hub/

[41] "AWS IoT Core Pricing," *Amazon AWS.* [Online]. Available: https://aws.amazon.com/iot-core/pricing/

[42] "Pricing," *Google Cloud IoT Core*. [Online]. Available: https://cloud.google.com/iot/pricing

[43] "Azure Functions pricing," *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/

[44] "AWS Lambda Pricing," *Amazon AWS.* [Online]. Available: https://aws.amazon.com/lambda/pricing/

[45] "Pricing," *Google Cloud Functions*. [Online]. Available: https://cloud.google.com/functions/pricing

[46] "Notification Hubs pricing," *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/notification-hubs/

[47] "Amazon SNS Pricing, *Amazon AWS.* [Online]. Available: https://aws.amazon.com/sns/pricing/

[48] "Firebase Cloud Messaging," *Google Firebase*. [Online]. Available: https://firebase.google.com/products/cloud-messaging

[49] "Azure CosmosDB Pricing," *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/cosmos-db/

[50] "Amazon DynamoDB Pricing," *Amazon AWS.* [Online]. Available: https://aws.amazon.com/dynamodb/pricing/

[51] "Understanding Cloud Firestore Billing," *Google Firebase Documentation*. [Online]. Available: https://firebase.google.com/docs/firestore/pricing

[52] "Understanding Realtime Database Billing," *Google Firebase Documentation*. [Online]. Available: https://firebase.google.com/docs/database/usage/billing

[53] "Firebase API Reference," *Google Firebase Documentation*. [Online]. Available: https://firebase.google.com/docs/reference

[54] "Swift," *Apple Developer*. [Online]. Available: https://developer.apple.com/swift/

[55] "React Native," *Facebook Open Source*. [Online]. Available: https://reactnative.dev/

[56] DFRobot, "DFRobot_SHT3x," *GitHub*. [Online]. Available: https://github.com/DFRobot/DFRobot_SHT3x

[57] "Ethernet library," *Arduino Reference*. [Online]. Available: https://www.arduino.cc/en/Reference/Ethernet

[58] OSU OPEnS, "SSLClient," *GitHub*. [Online]. Available: https://github.com/OPEnSLab-OSU/SSLClient

[59] G. Class, "Google Cloud IoT JWT," *GitHub*. [Online]. Available: https://github.com/GoogleCloudPlatform/google-cloud-iot-arduino

[60] J. Gähwiler, "arduino-mqtt," *GitHub*. [Online]. Available: https://github.com/256dpi/arduino-mqtt

[61] "Cloud Firestore Data model," *Google Firebase Documentation*. [Online]. Available: https://firebase.google.com/docs/firestore/data-model

[62] A. James, "Closes #1 Pull Request," *GitHub*. [Online]. Available: https://github.com/DFRobot/DFRobot_SHT3x/pull/2

[63] "Web Client," *Arduino Tutorials*, Feb. 5, 2018. [Online]. Available: https://www.arduino.cc/en/Tutorial/LibraryExamples/WebClient

[64] "Application Note: Dew Point Calculation," *Sensirion*, Oct. 3, 2006. [Online]. Available: http://irtfweb.ifa.hawaii.edu/~tcs3/tcs3/Misc/Dewpoint_Calculation_Humidity_Sensor_E.pdf

[65] "Writing Conditions for Cloud Firestore Security Rules," *Google Firebase Documentation*, Jun. 10, 2021. [Online]. Available: https://firebase.google.com/docs/firestore/security/rules-conditions

[66] "Firebase Authentication," *Google Firebase Documentation*, Jun. 10, 2021. [Online]. Available: https://firebase.google.com/docs/auth

[67] P. Hudson, "What is SwiftUI?" *Hacking With Swift*, Feb. 9, 2021. [Online]. Available: https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui

[68] "Add Firebase to your iOS project," *Google Firebase Documentation*, Jun. 10, 2021. [Online]. Available: https://firebase.google.com/docs/ios/setup

[69] "Structures and Classes," *The Swift Programming Language*, 2021. [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html

[70] "How the Program Works," *Apple Developer Program*, 2021. [Online]. Available: https://developer.apple.com/programs/how-it-works/

Appendices

Full code for the ESP32 endpoint configured for ethernet is presented here, along with code for both cloud functions. The Swift code for the iOS app is too long to present here, but is available at the GitHub link for this project, along with a version of the ESP32 code written to support use with WiFi: https://github.com/andrewmartinjames/FrostAlert

ESP32 Endpoint - Ethernet

FrostAlertEndpointEthernet.ino

```
/*  Endpoint code for FrostAlert system using WIZ5500 ethernet chip
 *  Written by Andrew James
 *  Functions for libraries are derived from example programs for those
libraries
 */

#include <Wire.h> // used for SHT3x library
#include <DFRobot_SHT3x.h> // communicates with SHT31 (modified library used,
but default works fine)
#include <SPI.h> // used for ethernet
#include <EthernetLarge.h> // modified ethernet library to allow for larger
SSLClient buffer size
#include <Client.h> // generic client class used for mqtt connection
#include <MQTT.h> // lwmqtt library
#include <jwt.h> // google cloud iot core library, for generating JWTs
#include <SSLClient.h> // used for SSL over ethernet
#include <CloudIoTCore.h> // main google cloud iot core library
#include <CloudIoTCoreMqtt.h> // google cloud iot core library, for managing
mqtt connection
#include "secrets.h"; // MUST BE UPDATED with individual project details to
connect to IoT core
#include "certificates.h" // contains SSL certificates for
mqtt.2030.ltsapis.goog:8883 for SSLClient
#include <EthernetUdp.h> // used for time server sync

// SHT31 config
DFRobot_SHT3x sht3x(&Wire,/*address=*/0x44,/*RST=*/4); //address may be 44 or
45

// MAC address of ethernet shield; MUST BE UPDATED for your WIZ5500 chip
byte mac[] = { 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX };

// IP if DHCP connection fails
IPAddress ip(192, 168, 0, 177);
IPAddress myDns(192, 168, 0, 1);

// declare clients and iot core objects
Client *sslClient; // instantiated as client class for compatibility with iot
core library
CloudIoTCoreDevice *device; // iot core device object
CloudIoTCoreMqtt *mqtt; // iot core mqtt object
MQTTClient *mqttClient; // lwmqtt client used with iot core
unsigned long iat = 0; // stores epoch time
```

```
String jwt; // stores JSON web token for authentication over MQTT with iot
core
EthernetClient ethClient; // declare Ethernet client object
EthernetUDP Udp; // UDP Instance for communication with NTP server

// initialize variables & constants for NTP
unsigned int localPort = 8888;       // local port to listen for UDP packets
const char timeServer[] = "time.nist.gov"; // government NTP server
const int NTP_PACKET_SIZE = 48; // NTP time stamp is in the first 48 bytes of
the message
byte packetBuffer[NTP_PACKET_SIZE]; //buffer to hold incoming and outgoing
UDP packets


/* Function to get current JWT for authentication with Cloud IoT Core
 * JWT returned will not authenticate if time is not returned by NTP server
 * NTP functionality based on
https://www.arduino.cc/en/Reference/EthernetUDPBegin
 * JWT functionality based on CloudIoTCore library examples
 */
String getJwt(){
  sendNTPpacket(timeServer); // send an NTP packet to a time server

  // wait 5 seconds to ensure reply is received
  delay(5000);
  if (Udp.parsePacket()) {
    // We've received a packet, read the data from it
    Udp.read(packetBuffer, NTP_PACKET_SIZE); // read the packet into the
buffer

    // the timestamp starts at byte 40 of the received packet and is four
bytes,
    // or two words, long. First, extract the two words:

    unsigned long highWord = word(packetBuffer[40], packetBuffer[41]);
    unsigned long lowWord = word(packetBuffer[42], packetBuffer[43]);
    // combine the four bytes (two words) into a long integer
    // this is NTP time (seconds since Jan 1 1900):
    unsigned long secsSince1900 = highWord << 16 | lowWord;
    Serial.print("Seconds since Jan 1 1900 = ");
    Serial.println(secsSince1900);

    // now convert NTP time into everyday time:
    Serial.print("Unix time = ");
    // Unix time starts on Jan 1 1970. In seconds, that's 2208988800:
    const unsigned long seventyYears = 2208988800UL;
    // subtract seventy years:
    iat = secsSince1900 - seventyYears;
    // print Unix time:
    Serial.println(iat);
  }
  Serial.println("Refreshing JWT");
  Serial.println(iat);
  jwt = device->createJWT(iat, jwt_exp_secs);
  return jwt;
}
```

```
// send an NTP request to the time server at the given address
// based on https://www.arduino.cc/en/Reference/EthernetUDPBegin
void sendNTPpacket(const char * address) {
  // set all bytes in the buffer to 0
  memset(packetBuffer, 0, NTP_PACKET_SIZE);
  // Initialize values needed to form NTP request
  // (see URL above for details on the packets)
  packetBuffer[0] = 0b11100011;   // LI, Version, Mode
  packetBuffer[1] = 0;     // Stratum, or type of clock
  packetBuffer[2] = 6;     // Polling Interval
  packetBuffer[3] = 0xEC;  // Peer Clock Precision
  // 8 bytes of zero for Root Delay & Root Dispersion
  packetBuffer[12]  = 49;
  packetBuffer[13]  = 0x4E;
  packetBuffer[14]  = 49;
  packetBuffer[15]  = 52;

  // all NTP fields have been given values, now
  // you can send a packet requesting a timestamp:
  Udp.beginPacket(address, 123); // NTP requests are to port 123
  Udp.write(packetBuffer, NTP_PACKET_SIZE);
  Udp.endPacket();
}

// Starts ethernet and runs checks for connection, the opens UDP port
void setupEth(){
  Serial.println("Starting ethernet");
  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    // Check for Ethernet hardware present
    if (Ethernet.hardwareStatus() == EthernetNoHardware) {
      Serial.println("Ethernet shield was not found.  Sorry, can't run
without hardware. :(");
      while (true) {
        delay(1); // do nothing, no point running without Ethernet hardware
      }
    }
    if (Ethernet.linkStatus() == LinkOFF) {
      Serial.println("Ethernet cable is not connected.");
    }
    // try to congifure using IP address instead of DHCP:
    Ethernet.begin(mac, ip, myDns);
  } else {
    Serial.print("  DHCP assigned IP ");
    Serial.println(Ethernet.localIP());
  }
  // give the Ethernet shield a second to initialize:
  delay(2000);
  Udp.begin(localPort);
}

// connect to Google IoT Core over mqtt
void connect(){
  mqtt->mqttConnect();
}
```

```
//// set up objects and clients for Cloud IoT core, then start the mqtt
connection
void setupCloudIoT(){
  device = new CloudIoTCoreDevice(
      project_id, location, registry_id, device_id,
      private_key_str);
  setupEth();
  sslClient = new SSLClient(ethClient, TAs, (size_t)TAs_NUM, A0);
  mqttClient = new MQTTClient(512);
  mqttClient->setOptions(180, true, 1000); // keepAlive, cleanSession,
timeout
  mqtt = new CloudIoTCoreMqtt(mqttClient, sslClient, device);
  mqtt->setUseLts(true);
  mqtt->startMQTT();
}

// MQTT publish function for strings
bool publishTelemetry(String data){
  return mqtt->publishTelemetry(data);
}

// MQTT publish function for character arrays
bool publishTelemetry(const char *data, int length){
  return mqtt->publishTelemetry(data, length);
}

// MQTT callback function, currently unused
void messageReceived(String &topic, String &payload){
  Serial.println("incoming: " + topic + " - " + payload);
}

// get temperature and humidity data from SHT31
String getTRH() {
  DFRobot_SHT3x::eRepeatability_t repeatability =
DFRobot_SHT3x::eRepeatability_High;
  DFRobot_SHT3x::sRHAndTemp_t curHT =
sht3x.readTemperatureAndHumidity(repeatability);
  String temp = String(curHT.TemperatureC);
  String hum = String(curHT.Humidity);
  Serial.println("{\"temperatureC\": " + temp + ", \"humidity\":" + hum +
"}");
  return "{temp:" + temp + "; hum:" + hum + "}";
}


void setup() {
  Ethernet.init(33);  // set ethernet pin for ESP32 with Adafruit Featherwing
Ethernet
  Serial.begin(115200);
  pinMode(13, OUTPUT);
  while (sht3x.begin() != 0) {
    Serial.println("Failed to Initialize the chip, please confirm the wire
connection");
    Serial.println(sht3x.begin());
    delay(1000);
  }
  if(!sht3x.softReset()){
```

```
    Serial.println("Failed to Initialize the chip....");
  }
  setupCloudIoT();
  Serial.println("mqtt established");
}


unsigned long lastMillis = 0;
void loop() {
  mqtt->loop();
  delay(10);  // <- fixes some issues with connection stability
  if (!mqttClient->connected()) { // maintain active mqtt connection
    connect();
    Serial.println("connection attempted");
  }
  if (millis() - lastMillis > 60000) { // publish every 1 minute
    lastMillis = millis();
    publishTelemetry(getTRH());
  }

}
```

certificates.h

```
#ifndef _CERTIFICATES_H_
#define _CERTIFICATES_H_

#ifdef __cplusplus
extern "C"
{
#endif

/* This file is auto-generated by the pycert_bearssl tool.  Do not change it
manually.
 * Certificates are BearSSL br_x509_trust_anchor format.  Included certs:
 *
 * Index:    0
 * Label:    GlobalSign
 * Subject:  CN=GlobalSign,O=GlobalSign,OU=GlobalSign ECC Root CA - R4
 *
 * Index:    1
 * Label:    GTS LTSR
 * Subject:  CN=GTS LTSR,O=Google Trust Services LLC,C=US
 */

#define TAs_NUM 2

static const unsigned char TA_DN0[] = {
    0x30, 0x50, 0x31, 0x24, 0x30, 0x22, 0x06, 0x03, 0x55, 0x04, 0x0b, 0x13,
    0x1b, 0x47, 0x6c, 0x6f, 0x62, 0x61, 0x6c, 0x53, 0x69, 0x67, 0x6e, 0x20,
    0x45, 0x43, 0x43, 0x20, 0x52, 0x6f, 0x6f, 0x74, 0x20, 0x43, 0x41, 0x20,
    0x2d, 0x20, 0x52, 0x34, 0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55, 0x04,
    0x0a, 0x13, 0x0a, 0x47, 0x6c, 0x6f, 0x62, 0x61, 0x6c, 0x53, 0x69, 0x67,
    0x6e, 0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55, 0x04, 0x03, 0x13, 0x0a,
    0x47, 0x6c, 0x6f, 0x62, 0x61, 0x6c, 0x53, 0x69, 0x67, 0x6e,
```

```
};

static const unsigned char TA_EC_CURVE0[] = {
    0x04, 0xb8, 0xc6, 0x79, 0xd3, 0x8f, 0x6c, 0x25, 0x0e, 0x9f, 0x2e, 0x39,
    0x19, 0x1c, 0x03, 0xa4, 0xae, 0x9a, 0xe5, 0x39, 0x07, 0x09, 0x16, 0xca,
    0x63, 0xb1, 0xb9, 0x86, 0xf8, 0x8a, 0x57, 0xc1, 0x57, 0xce, 0x42, 0xfa,
    0x73, 0xa1, 0xf7, 0x65, 0x42, 0xff, 0x1e, 0xc1, 0x00, 0xb2, 0x6e, 0x73,
    0x0e, 0xff, 0xc7, 0x21, 0xe5, 0x18, 0xa4, 0xaa, 0xd9, 0x71, 0x3f, 0xa8,
    0xd4, 0xb9, 0xce, 0x8c, 0x1d,
};

static const unsigned char TA_DN1[] = {
    0x30, 0x44, 0x31, 0x0b, 0x30, 0x09, 0x06, 0x03, 0x55, 0x04, 0x06, 0x13,
    0x02, 0x55, 0x53, 0x31, 0x22, 0x30, 0x20, 0x06, 0x03, 0x55, 0x04, 0x0a,
    0x13, 0x19, 0x47, 0x6f, 0x6f, 0x67, 0x6c, 0x65, 0x20, 0x54, 0x72, 0x75,
    0x73, 0x74, 0x20, 0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x73, 0x20,
    0x4c, 0x4c, 0x43, 0x31, 0x11, 0x30, 0x0f, 0x06, 0x03, 0x55, 0x04, 0x03,
    0x13, 0x08, 0x47, 0x54, 0x53, 0x20, 0x4c, 0x54, 0x53, 0x52,
};

static const unsigned char TA_EC_CURVE1[] = {
    0x04, 0xcd, 0xf1, 0x8c, 0x8e, 0xda, 0xef, 0xb2, 0x09, 0x0a, 0x19, 0x77,
    0x00, 0x24, 0x50, 0xdb, 0xf9, 0x73, 0x77, 0x68, 0x91, 0xf5, 0x0b, 0x7e,
    0xb0, 0x3a, 0x40, 0x98, 0x05, 0x57, 0x65, 0xcc, 0xb8, 0x43, 0x6d, 0x41,
    0x92, 0x06, 0xe4, 0x75, 0x0e, 0x4b, 0xa8, 0xc5, 0x9f, 0xc7, 0xf4, 0xc9,
    0x29, 0x55, 0x78, 0xe4, 0x42, 0xc6, 0xa1, 0x72, 0x8c, 0x32, 0x72, 0x46,
    0x7f, 0x3a, 0x77, 0xe2, 0x24,
};

static const br_x509_trust_anchor TAs[] = {
    {
        { (unsigned char *)TA_DN0, sizeof TA_DN0 },
        BR_X509_TA_CA,
        {
            BR_KEYTYPE_EC,
            { .ec = {BR_EC_secp256r1, (unsigned char *)TA_EC_CURVE0, sizeof
TA_EC_CURVE0}
        }
        }
    },
    {
        { (unsigned char *)TA_DN1, sizeof TA_DN1 },
        BR_X509_TA_CA,
        {
            BR_KEYTYPE_EC,
            { .ec = {BR_EC_secp256r1, (unsigned char *)TA_EC_CURVE1, sizeof
TA_EC_CURVE1}
        }
        }
    },
};

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* ifndef _CERTIFICATES_H_ */
```

secrets.h

```
 // Configuration secrets for Cloud IoT Core,
 // based on Google Cloud IoT Core JWT Library ciotc_config.h

// Wifi network details.
const char *ssid = "xxxxxxxx";
const char *password = "xxxxxxxx";

 // Cloud iot details, may need to be updated for your project.
const char *project_id = "frost-alert-21";
const char *location = "us-central1";
const char *registry_id = "endpoints";
const char *device_id = "garden32";


// Configuration for NTP
const char* ntp_primary = "pool.ntp.org";
const char* ntp_secondary = "time.nist.gov";

// private key from keypair used for IoT registry device ID
const char *private_key_str =
    "xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:"
    "xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:"
    "xx:xx";

// jwt token expiration time (max 24 hours, 3600*24)
const int jwt_exp_secs = 60*20;

// ssl CA certificate for mqtt.2030.ltsapis.goog:8883
// to refresh, run openssl s_client -showcerts -connect
mqtt.2030.ltsapis.goog:8883
// and copy all lines here with appropriate formatting
const char *root_cert = \
    "-----BEGIN CERTIFICATE-----\n" \
    "MIIESjCCAzKgAwIBAgINAeO0mqGNiqmBJWlQuDANBgkqhkiG9w0BAQsFADBMMSAw\n" \
    "HgYDVQQLExdHbG9iYWxTaWduIFJvb3QgQ0EgLSBSMjETMBEGA1UEChMKR2xvYmFs\n" \
    "U2lnbjETMBEGA1UEAxMKR2xvYmFsU2lnbjAeFw0xNzA2MTUwMDAwNDJaFw0yMTEy\n" \
    "MTUwMDAwNDJaMEIxCzAJBgNVBAYTAlVTMR4wHAYDVQQKExVHb29nbGUgVHJ1c3Qg\n" \
    "U2VydmljZXMxEzARBgNVBAMTCkdUUyBDQSAxTzEwggEiMA0GCSqGSIb3DQEBAQUA\n" \
    "A4IBDwAwggEKAoIBAQDQGM9F1IvN05zkQO9+tN1pIRvJzzyOTHW5DzEZhD2ePCnv\n" \
    "UA0Qk28FgICfKqC9EksC4T2fWBYk/jCfC3R3VZMdS/dN4ZKCEPZRrAzDsiKUDzRr\n" \
    "mBBJ5wudgzndIMYcLe/RGGFl5yODIKgjEv/SJH/UL+dEaltN11BmsK+eQmMF++Ac\n" \
    "xGNhr59qM/9il71I2dN8FGfcddwuaej4bXhp0LcQBbjxMcI7JP0aM3T4I+DsaxmK\n" \
    "FsbjzaTNC9uzpFlgOIg7rR25xoynUxv8vNmkq7zdPGHXkxWY7oG9j+JkRyBABk7X\n" \
    "rJfoucBZEqFJJSPk7XA0LKW0Y3z5oz2D0c1tJKwHAgMBAAGjggEzMIIBLzAOBgNV\n" \
    "HQ8BAf8EBAMCAYYwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMBIGA1Ud\n" \
    "EwEB/wQIMAYBAf8CAQAwHQYDVR0OBBYEFJjR+G4Q68+b7GCfGJAboOt9Cf0rMB8G\n" \
    "A1UdIwQYMBaAFJviB1dnHB7AagbeWbSaLd/cGYYuMDUGCCsGAQUFBwEBBCkwJzAl\n" \
    "BggrBgEFBQcwAYYZaHR0cDovL29jc3AucGtpLmdvb2cvZ3NyMjAyBgNVHR8EKzAp\n" \
    "MCegJaAjhiFodHRwOi8vY3JsLnBraS5nb29nL2dzcjIvZ3NyMi5jcmwwPwYDVR0g\n" \
    "BDgwNjA0BgZngQwBAgIwKjAoBggrBgEFBQcCARYcaHR0cHM6Ly9wa2kuZ29vZy9y\n" \
    "ZXBvc2l0b3J5LzANBgkqhkiG9w0BAQsFAAOCAQEAGoA+Nnn78y6pRjd9XlQWNa7H\n" \
    "TgiZ/r3RNGkmUmYHPQq6Scti9PEajvwRT2iWTHQr02fesqOqBY2ETUwgZQ+lltoN\n" \
    "FvhsO9tvBCOIazpswWC9aJ9xju4tWDQH8NVU6YZZ/XteDSGU9YzJqPjY8q3MDxrz\n" \
```

```
    "mqepBCf5o8mw/wJ4a2G6xzUr6Fb6T8McDO22PLRL6u3M4Tzs3A2M1j6bykJYi8wW\n" \
    "IRdAvKLWZu/axBVbzYmqmwkm5zLSDW5nIAJbELCQCZwMH56t2Dvqofxs6BBcCFIZ\n" \
    "USpxu6x6td0V7SvJCCosirSmIatj/9dSSVDQibet8q/7UK4v4ZUN80atnZz1yg==\n" \
    "-----END CERTIFICATE-----\n";
```

Google Cloud Functions

tempAlert

```
import base64
import math
import firebase_admin
from firebase_admin import db, firestore
from firebase_admin import messaging

fb = firebase_admin.initialize_app()
fst = firestore.client()

def hello_pubsub(event, context):
    """Triggered from a message on a Cloud Pub/Sub topic.
    Args:
         event (dict): Event payload.
         context (google.cloud.functions.Context): Metadata for the event.
    """
    pubsub_message = base64.b64decode(event['data']).decode('utf-8')
    attributes = event['attributes']
    deviceID = attributes['deviceId']

    endpointDocument = fst.document('endpoints/%s' % deviceID)
    endpoint = endpointDocument.get().to_dict()
    uid = endpoint["user"]
    userDocument = fst.document('users/%s' % uid)
    user = userDocument.get().to_dict()
    fcmToken = user["fcm_token"]
    threshTemp = user["threshold_temp"]

    if pubsub_message[0] == "{":
         strArr = pubsub_message.strip("{").strip("}").split("; ")
         temp = strArr[0].strip("temp:")
         tempf = float(temp)
         hum = strArr[1].strip("hum:")
         humf = float(hum)
         H = (math.log10(humf)-2)/0.4343 + (17.62*tempf)/(243.12+tempf)
         Dp = 243.12*H/(17.62-H)
         if ((Dp < 0) or (tempf < threshTemp)):
               message = messaging.Message(
                    notification=messaging.Notification(
                        title='Incoming Frost Detected!',
                        body='Prepare your garden for an incoming frost!',
                        ),
                    token=fcmToken,
               )
               response = messaging.send(message)

         values = {
              "current_hum": humf,
              "current_temp": tempf,
              "user" : uid
         }
         endpointDocument.set(values)
```

```
            print("Firestore updated")
        else:
            print("Not a temperature message")
```

newUser

```
const functions = require("firebase-functions");
const {Firestore} = require("@google-cloud/firestore");
const firestore = new Firestore();

const admin = require("firebase-admin");
admin.initializeApp();

exports.newUser = functions.auth.user().onCreate((user) => {
  // ...
  const uid = user.uid;
  const docRef = firestore.collection("users");
  const document = docRef.doc(uid);
  console.log(uid);
  document.set({
    uid: uid,
    endpoint: "/endpoints/",
    threshold_temp: 2.0,
    fcm_token: "",
  });
});
```