

# This Butter Be Worth It: IoT Sensor System for Wildfire Detection

---

*Nathan Do, Shreejan Gupta, Tahmid Kazi, Alec Ross, and Bill Yang*

10 December 2020

**Capstone Design ECE 4440 / ECE4991**

## Signatures

Nathan Do - 

---

Shreejan Gupta 


---

Tahmid Kazi - 

---

Alec Ross - 

---

Bill Yang - 

---

## **Statement of Work:**

### *Nathan*

Nathan took on the majority of the work that involved UI and AWS. As a result, he built the AWS infrastructure necessary to receive, store, and send data in a database and alert users when there was a potential fire. He also individually coded the UI with the Angular framework. Achieving this required learning about various AWS services; Nathan utilized AWS lambda to programmatically process the data every time a POST request is made through AWS API Gateway. This POST request is also connected to the AWS SNS system, which would alert users if sensors read incoming data and interpret potential danger for a fire. Also, the POST request that causes all of this also triggers the receiving and saving of data in the non-relational DynamoDB service. The webpage uses a Cognito identity pool to query the database for the information it needs. As the individual behind the AWS infrastructure, he also provided the microcontrollers' means to update the database. He played a role in shaping the process for the firmware to make an update to the database.

### *Shreejan*

As program manager, Shreejan was in charge of constructing and maintaining the Gantt charts and the timeline throughout the semester. He was in charge of getting weekly updates and updating the current tasks' status promptly. Besides holding the secondary role of a software architect, Shreejan also contributed to developing the visual dashboard in many ways. To begin, he aided in the blueprint for the visual dashboard for the proposal. He researched the tools (AWS, JavaScript, DynamoDB, etc.) to finalize the dashboard selection. He also implemented an automated method to pipeline the data from the ESP8266 to the DynamoDB using AWS. Finally, he also contributed to integrating the Wi-Fi modules to the visual dashboard alongside Nathan and Tahmid. Shreejan's contribution helped with the implementation and development of the graphical dashboard. It established the necessary pipeline to effectively transfer the data from the sensors to the database, an essential requirement for the sensor system's functionality.

### *Tahmid*

As the primary lead on firmware for the microcontrollers, especially the wireless communication component, Tahmid was responsible for implementing the firmware that enabled Wi-Fi communication. He was also responsible for implementing the necessary firmware that allowed SSL encryption to securely connect and pipeline sensor data in JSON format to the AWS API that Nathan created via HTTPS POST requests. To accomplish this required learning from datasheets, SDKs, and software libraries, including the ESP8266 core for the Arduino IDE, and devising a way for SSL encryption to be implemented on the microcontroller to enable the POST request to be authenticated, encrypted, and sent via HTTPS. Furthermore, Tahmid also contributed to the development, testing, and debugging of the I<sup>2</sup>C firmware developed to extract data from the sensors, perform the necessary calculations, and gather sensor data on the microcontroller before being packetized and sent to the cloud API.

### *Alec*

Alec was primarily in charge of hardware design, component selection, and circuitry development. His responsibilities included constructing the schematics for the power and sensor subsystems using KiCad along with the composition of the PCB layout. He debugged and resolved many technical challenges throughout the development process, notably with the LDO issue. Acting as firmware support, Alec also contributed to the development of the sensor code to fetch and store the data from the sensors through the I<sup>2</sup>C protocol working closely with Bill throughout the project.

### *Bill*

Serving as the embedded system support, Bill was responsible for integrating the embedded system, guaranteeing both the hardware and firmware. Towards the beginning, Bill was mainly involved in the PCB design, including schematics and layout with Alec using KiCad, double-checking the logic and the math. He was also responsible for budget supervision, parts ordering, and communication with WWW Electronics for proper PCB assembly. Throughout the hardware fabrication, Bill worked closely with Alec, especially during testing, notably solving both the “heatsink” and the “ohmmeter” issue of the LDO. Towards the end, Bill also pair-programmed and peer-reviewed the sensor and the Wi-Fi communication firmware code.

# Table of Contents

<b>Capstone Design ECE 4440 / ECE4991</b>	<b>1</b>
<b>Signatures</b>	<b>1</b>
<b>Statement of Work:</b>	<b>2</b>
<b>Table of Contents</b>	<b>4</b>
<b>Table of Figures</b>	<b>5</b>
<b>Abstract</b>	<b>6</b>
<b>Background</b>	<b>6</b>
<b>Constraints</b>	<b>7</b>
Design Constraints	7
Economic and Cost Constraints	8
<b>External Standards</b>	<b>9</b>
Tools Employed	9
<b>Ethical, Social, and Economic Concerns</b>	<b>11</b>
Environmental Impact	11
Sustainability	11
Health and Safety	11
Manufacturability	11
Ethical Issues	11
Intellectual Property Issues	12
<b>Detailed Technical Description of Project</b>	<b>13</b>
<b>Project Timeline</b>	<b>17</b>
<b>Test Plan</b>	<b>17</b>
<b>Final Results</b>	<b>20</b>
<b>Costs</b>	<b>20</b>
<b>Future Work</b>	<b>21</b>
<b>References</b>	<b>23</b>
<b>Appendix</b>	<b>26</b>

# Table of Figures

Figure #1: Battery Power System Schematic	14
Figure #2: The Battery Level Indicator Schematic	14
Figure #3: Sensor Subsystem Schematic	15
Figure #4: Sensor Configuration Schematics	15
Figure #5: Lambda Visualization and Code	16
Figure #6: Visual Dashboard of Sensor Data	16
Figure #7: Initial Project Timeline	17
Figure #8: Final Project Timeline	17
Figure #9: PCB Design	26
Figure #10: 200 HTTP Response in UART Terminal	26
Figure #11: Project Design Overview	27
Figure #12: Power Subsystem Test Plan	28
Figure #13: Sensor Subsystem Test Plan	29
Figure #14: Firmware Test Plan	30
Figure #15: Visual Dashboard Test Plan	31
Figure #16: Total Budget Breakdown	32
Figure #17: DigiKey Bill of Materials	32
Figure #18: Mouser Bill of Materials	32
Figure #19: Proposal Expectations (Rubric)	33
Figure #20: Lambda Function and Dashboard Code	34
Figure #21: Firmware Code for Wi-Fi/HTTPS Communication & I2C Sensor Integration	36

## **Abstract**

The recent phenomenon known as the Internet of Things (IoT) shows us a glimpse into a future where people rely on smart connected devices to solve various problems. However, people have used IoT devices primarily in dense, urban areas, and homes because they rely on existing electronic network technologies such as Ethernet, Wi-Fi, and Bluetooth. These technologies depend on devices' proximities to each other to transmit data reliably. However, with the emergence and ubiquity of novel wireless technologies such as Zigbee and LoRa (with the proper antennas) and ultra-low-power hardware, it is now possible to use IoT devices remotely in rural areas to gather data from a broader geographical surface area. Doing so enables more excellent coverage and data acquisition from locations that were previously difficult to monitor. The emergence of these new technologies enables engineers to solve an even greater host of problems based on wireless technologies' increased capabilities and reliability. The sensor system that we aim to design is a distributed IoT network that can detect and monitor hazardous conditions, such as wildfires, remotely to help humans respond to these threats and prevent large scale fires and other infrastructure damages.

## **Background**

The 2019-2020 bushfire season in Australia destroyed an estimated 46 million acres and close to 6000 buildings in a few months. Similarly, countless wildfires ravaged California areas, causing millions in damages while displacing many families and calling for massive evacuations. A fast and reliable fire detection network can save countless lives and prevent millions of dollars in damages. There are currently many implementations of satellite systems to detect forest fires; however, all satellite-based observation has limitations, often leading to failure in the detection speed, the quality, or the running cost to produce effective control for forest areas [1]. It is also worth noting satellites orbiting thousands of miles above the Earth have deadends; they cannot provide "omniscient" coverage, hindering their ability to detect forest fires in their early stages. Another way that has come up is to utilize automatic smoke detection, but all current implementations will cost too much money to build and maintain [1].

The project will explore a vast list of emerging technologies to construct an Internet of Things wireless network that utilizes sensors to collect and transmit data existing web technologies. These web technologies, which consist of databases and data visualization dashboards, create a robust and smart monitoring system for the early detection of wildfires. Aeris and LADSensors have implemented a similar implementation of our solution in the past, where metrics such as temperature, ambient CO2 levels, humidity, wind direction, and speed were measured to detect wildfires in early stages and predict their potential spread [2]. The sensors used in LADSensors send the data to a LoRa gateway connectivity, which are then processed with AI to provide a detailed and comprehensive view of conditions. LABSensors uses LoRa similar to our design for data collection and remote long-distance connectivity. This project's design varies from prior systems such that the operational and manufacturing costs will be substantially minimized while keeping performance as the top priority. One method of achieving this would be by integrating a recharging subsystem for the battery power supply.

The project calls for prior knowledge from embedded systems courses (ECE 3430, ECE 3501, ECE 3502), Digital Logic Design (CS/ECE 2330), Fundamentals of Electrical Engineering I, II, and III (FUN I-ECE 2630, FUN II-ECE 2660, FUN III-ECE 3750), and Advanced Software Development (CS 3240). Embedded C programming is required to connect all the sensors and the wireless microcontroller ESP8266 [3], which was introduced to the team in the embedded courses. To design the motherboard that integrates all the components needed to build the sensing nodes, the team used prior knowledge and skills, such as circuit design, testing, and fabrication skills, from the countless labs and projects from DLD and the FUN series. This design involves building an electrical interface between the ESP8266 microcontroller, the air quality sensor, the temperature and humidity sensor, and the wireless module, all of which will utilize SPI, I<sup>2</sup>C, and GPIO pins in the microcontroller. To build the data acquisition and visualization platform, prior knowledge in web application development skills were used. The team learned from a range of classes, including Advanced Software Development (CS3240), Program and Data Representation (CS2150), Cloud Computing (CS 4740), and the open-source software tools they worked with during their collective internships this summer. Overall, each member used their collective knowledge from numerous computer science courses over the last three years with specifics from the three classes listed above to complete the UI interface. These tasks include using the open-source database software (DynamoDB [4]) and understanding how to use modern Javascript libraries such as Angular [5] and Chart.js [6].

## **Constraints**

### **Design Constraints**

Most of the product components, whether batteries, casing, electrical components, PCBs, or wires, etc., required for the device's construction, are relatively available and inexpensive. The design cost will not be a considerable burden, ensuring that the device can be replicated easily for scalable production. The power supply and sensor subsystem are relatively easy to assemble, and materials are common. The availability of the components will not be an issue; however, the use of plastic and batteries at this stage will present the most concern for this design's environmental impact. To minimize these risks, damaged or nonfunctional components will be donated to companies to recycle and dispose of [7] correctly.

To elaborate, the components necessary for the development of the power system, sensor system, and the Wi-Fi module are readily available through DigiKey and Mouser, even considering supplies for mass production as indicated in Figures x and y in the Appendix. For the PCB boards' assembly, the vendor of choice was WWW Electronics Inc, based in Charlottesville, VA (3W). 3W imposed the following manufacturing restrictions: silkscreen standard requires that all writing is in the same orientation, and the font must be at least 0.06" or 1.5mm tall, with a preferred size of 0.08" or 2.0mm. The PCB board itself must be at most 30 square inches in size and with a maximum complexity of 2 layers. The PCB also needs to pass the FreeDFM inspection, where the entire layout must be within 100 mils of the board outline, and the units must be in metric upon submission.

In regards to the tools and resources for the development of the visual dashboard, concerns in the availability of the services required was not a significant concern as Amazon Web Services (AWS) provided the services (API Gateway [8], Lambda [9], DynamoDB [4], SNS [10]) with excellent reliability and low costs.

### **Economic and Cost Constraints**

The ESP8266 [3], which is the microcontroller that was utilized to enable collection and wireless transmission of sensor data, was designed to be a mass-market, low-cost option for IoT applications with a price tag of as low as \$3/unit when purchased in bulk (the larger the quantity, the lower the price) from international suppliers. If this device is implemented as a mass-market commercial product, its per-unit cost would steadily decrease as the manufacturing capacity increases per the economies of scale for constant cost products. ESP8266 would be the more viable option when targeting a larger quantity of sales over the MSP432 [11] (which was the prior microcontroller of choice) with a per-unit cost of \$20-50 (the price includes the price of a separate network co-processor CC3120 [12]. In contrast, the ESP8266 integrates the microcontroller CPU and the network processor into one package).

Furthermore, free and open-source software avoids the reliance on potentially expensive multi-year software licensing contracts (the business model for a company like Texas Instruments). It ensures that the code will never become deprecated from changes in the manufacturer's software development kit.

### *Accounting Costs*

This project's initial budget is \$500 to ensure its timely completion and professional production. Regarding the breakdown of all the individual components and parts, their running total accounts are listed below:

1. PCB Board(s) Production (\$36 x2) ~ \$72 (Two prototypes were made in total)
2. MSP432 (\$23.99 x3) ~ \$71.97 (Initial choice of microcontroller)
3. Wi-Fi Module TI CC3120 Booster Pack (\$35.99 x3) ~ \$107.97
4. Hardware Components ~ \$69.97 (Two orders in total)
  - a. Batteries, Connectors, Electrical components, Sensors, and USB Charger
  - b. Casing ~ \$10 (Out of pocket)
5. PCB Assembly (By "3W Electronics") ~ \$21.60 (Two boards in total)
6. Software ~ \$50 (Out of pocket)
  - a. Database, cloud storage, external applications, etc.
7. Remaining Fund ~ \$106.79
8. ESP8266 (\$10 x3) ~ \$30 (Final choice for Wi-Fi microcontroller)
  - a. Paid out of pocket; not included for the final accounting cost

As seen above, the accounting cost of designing and implementing the systems for this project fell within the specified budget constraints of \$500. Major expenses include the PCB fabrication and components (namely the sensors), the wireless communication modules, and the TI MSP432 launchpads. After making the switch to the ESP8266 as the Wi-Fi microcontroller of choice, team member Tahmid had three microcontrollers in possession, so no further expenses



were incurred from the allocated budget since these were paid for out of pocket. Due to the project timeline and other circumstance constraints, only two prototypes were produced at the end; thus, a sizable “emergency fund” is left for future endeavors.

## External Standards

When defining the project’s requirements, it was determined that to build a system that is both low power, able to connect to the internet, and can do so wirelessly, Wi-Fi would be the ideal protocol of choice (outweighing ZigBee and LoRa at the end). The choice for the embedded system (eventually the ESP8266 microcontroller [3]) was made on the premise that it utilizes 2.4Ghz Wi-Fi to communicate with the internet, a feature core to our product since the sensor data was collected and pipelined to AWS cloud services through this communication channel. Wi-Fi, which was standardized as IEEE 802.11 b/g/n, can perform WLAN (wireless local area network) services [13]. Through the use of SDKs [14][15] available for the embedded system, a custom firmware was developed to use the HTTP and TCP/IP network stack to packetize the sensor data collected and subsequently connect to AWS cloud computing services to store collected data onto the cloud. The IEEE 802.11 standard [13] enabled the implementation of the functionality mentioned above while also keeping the devices wireless and battery-powered (reasonably low power consumption), leaving the design to meet all initial specifications.

## Tools Employed

### *Power and Sensor Subsystem*

- **Code Composer Studio [17]** - Code Composer Studio (CCS) IDE was initially utilized to develop and test the Wi-Fi modules and the sensor subsystems. CCS includes a large array of tools, such as C/C++ compiler, source code editor, debugger, and much more, for which we had to improve upon skills from prior classes to take full advantage of during the development of the project.
- **KiCad [18]** - KiCad was the schematic and PCB layout editor used in the power and sensor subsystems’ design. This tool was also utilized in the testing and debugging phases for the remainder of the project to resolve any issues that arose during the development and integration of the systems, namely, challenges with the proper LDO function and I<sup>2</sup>C bus interference. By and large, prior knowledge and experience were used; new software features such as assembly printout and automatic Digikey footprint library integrations were explored during the project.
- **Multisim [19]** - Multisim’s circuit simulator feature was utilized during the design and verification phases of the power and sensor subsystems to ensure all of the components function as intended and no unexpected reading during the systems’ manufacturing. Prior knowledge from courses was primarily employed in the use of Multisim for this project.

### *Wi-Fi Communication Subsystem*

- **ESP8266 Libraries for Arduino IDE [14][15][16]** - The open-source Github repository and subsequent documentation containing the SDK allowed the Arduino IDE to write all the functionality to enable Wi-Fi on the ESP8266. This includes setting the board as a

client, connecting it to a WPA2 encrypted access point with credentials, assembling an HTTP packet, and then authenticating using SSL certificates (in the form of a verifiable SHA-1 thumbprint) that is sent as part of the HTTPS request. Throughout the project, prior skills were improved throughout the development process and implementation of the Wi-Fi modules, particularly HTTPS communication and the AWS cloud integration aspect when writing the firmware for wireless data transmission.

- **Arduino IDE [20]** - Arduino IDE was the development environment used in conjunction with the ESP8266 core to develop the firmware for the Wi-Fi subsystem. This includes a UART terminal, firmware flashing functionality, and a code editor where all the firmware was written. Most of the skills were improved upon throughout the project, testing and debugging new errors.

### *Database and Notification System*

- **AWS DynamoDB [4]** - The NoSQL database option on AWS was used to store the different sensing nodes' data. Like Lambda, there was minimal exposure to it, and the research in accessing and inserting items into it in programs was something learned for this project. Skills revolving around DynamoDB were discovered while implementing the data transfer pipeline for the visualization feature.
- **AWS API Gateway [8]** - This service was used to create a POST endpoint that would trigger the aforementioned Lambda. Unlike the other Amazon Web Services, the API Gateway was something researched and learned all from scratch. Skills surrounding the API Gateway were obtained during the development of the visual dashboard as well.
- **AWS Lambda [9]** - Lambda is a service in AWS that works based on events and triggers, and it was used to call other services to update the database and send notifications. Knowledge of AWS Lambda was minimal and needed to be learned about integrating it with other services with the python library boto3. However, there was minor exposure to it in previous classes. Most of the features utilized were from prior knowledge gained from university courses; however, new skills were obtained working with AWS throughout the project.
- **Amazon SNS [10]** - Amazon's Simple Notification Service was used to send alerts about potential wildfires. As a part of AWS, it was reasonably straightforward to implement with the other Amazon Web Services and required additional research to utilize. With no prior knowledge of Amazon SNS, skills revolving around implementing the warning system were learned during the project.

### *Visual Dashboard*

- **Visual Studio Code [21]** - VSCode was the code-editor used to program the Lambda functions' dashboard and iterations. Prior knowledge obtained throughout college courses were used in this project.

- **Angular [5]** - The web framework is used to create the dashboard. While Angular was not new, using the Javascript library for AWS and integrating it into the dashboard's dependencies was troubleshoot and researched to make it functional. For the large part, previous knowledge around the tool was utilized for the project; however, small improvements were made.
- **GitHub Pages [22]** - The platform that currently hosts the dashboard. Prior knowledge about GitHub Pages were used for the use of this tool.

## **Ethical, Social, and Economic Concerns**

### **Environmental Impact**

The device will be exposed to various environmental forces during its operational phases. Given the chances of high heat and fire exposure in the field, the device's burning due to a fire can release toxins from the plastic and the batteries. Another concern would be that we may need to build the required infrastructure to be connected and fully functional for the network to operate. To combat the risks and challenges with the disposal of the damaged or unusable devices, recycling centers and technology companies can be called upon to dispose of these components [7] correctly.

### **Sustainability**

With the mission of saving the environment, the device itself will not be harmful to the environment. The components used are durable and meant to operate for long periods without maintenance. The biggest concern will be the usage of the plastic casing and batteries for the current implementation, which, as mentioned previously, will be recycled safely.

### **Health and Safety**

If a microcontroller and its respective components are successful in detecting a fire, it may be imminent that it catches on fire and burns; the destruction of the case and electronic components may release harmful fumes that could be a concert to human health. The electrical components onboard do not pose any harm to humans.

### **Manufacturability**

The larger components and the unique casing design for the device can limit the device's manufacturability. Though in the short run, a fixed budget and better can further implicate the case. Proper long term production analysis will lead to a significantly reduced cost and more feasible manufacturability.

### **Ethical Issues**

With a highly integrated and connected network, the biggest concern will be data and privacy in general. This device will be capable of gathering an immense amount of information, and the question of making this data public or not for the welfare of other countries will be explored down the road.

## Intellectual Property Issues

Overall, given the use of open-source software (from GitHub and the Arduino SDK) within the firmware architecture for sensors and the Wi-Fi modules, the current implementation of this project is not patentable [14][15]. Furthermore, as referenced in the patent CN104658156A, Claim 1 states the following: “a forest fire monitoring system, is characterized in that: comprising: multiple sensors (3) be installed in the forest, multiple centers with several sensors (3) are base station (2), the Surveillance center (1) of fixed point; Described sensor (3) comprises wireless communication module (4), smoke transducer (5), the pyroelectric sensor (6) of detection of fires, processor, battery; Wireless communication module (4) and the base station (4) wireless connections of the described sensor (3); Smoke transducer (5), pyroelectric sensor (6), a wireless communication module (4) is connected with the processor, described processor carries out logical operation and Logic judgment to the data of smoke transducer (5), the data of pyroelectric sensor (6), and judged result is transferred to the base station (2) by wireless communication module (4); Described base station (2) and Surveillance center (1) wireless connections” [25]. After dissecting this dependent claim, it can be concluded the characteristics of using multiple sensors and wireless communication modules and to a processor to carry out logical operation and judgments is a similar implementation to the data collection and transfer pipelines used in this project.

In addition, a forest fire monitoring and early warning system based on IoT has already been documented. Specifically, Claim 2, which is dependent on Claim 1 of the patent, states the following: “The forest fire monitoring early warning system based on Internet of Things according to claim 1; It is characterized in that described sensing terminal node comprises a plurality of sensors and sensing terminal ZigBee module; Each sensor is as the signal input of forest fire monitoring early warning system; Be used to gather the environmental information of woodland to be monitored; The signal output part of each sensor links to each other with the respective signal input end of sensing terminal ZigBee module, and sensing terminal ZigBee module links to each other with corresponding routing node through wireless network” [26]. This particular system uses terminal nodes composed of ZigBee modules, which is analogous to the proposed sensor module in the initial technical description of the project and the construction of the sensor node implemented with Wi-Fi modules is similar enough to make the project not patentable.

Furthermore, the monitoring system detailed in US6624750B1 further decreases the patentability of the project as the patent’s documentation provides in-depth analysis for the use of IoT technologies. These include but are not limited to Bluetooth, ZigBee, LoRa, Wi-Fi, radio frequency identification (RFID), and much more within “a management system and method for automatically monitoring and dynamically reacting to events and reconstructing application systems” [27]. As stated in claim 7, which is a dependent claim for the prior claims in the patent, states the following: “The system of claim 7 in which the multiple sensor devices are fire, smoke, or intrusion sensor devices that further comprise associated speakers and in which one of the multiple sensor devices transmits an alarm condition message signal to which the base station responds by transmitting a speaker activating message instructing the multiple sensor devices to vocally announce a location of the sensor transmitting the alarm condition message and whether

the alarm condition is a fire, smoke, or intrusion alarm condition.” The idea for an alarm condition message was seen in the project’s implementation for the visual dashboard. Additional features to this system are outlined in the patent, which further decreases the project's patentability.

## **Detailed Technical Description of Project**

The project is a distributed IoT network consisting of many sensor nodes that collect and pipeline the data to a cloud service terminal and display the data on a graphical user interface. In large, the system is made up of a microcontroller with wireless communication capability, a PCB of sensors, a PCB of a battery power supply, an independent recharging module, and a software cloud-based service. Each subsystem is discussed in technical detail in the following sections.

On the microcontroller, the firmware implemented to enable wireless communication via 2.4Ghz Wi-Fi utilizes the open-source Arduino SDK built for the ESP8266 microcontroller [14][15]. This includes the use of the base ESP8266Wi-Fi library and the ESP8266HTTPClient, and the Wi-FiClientSecureBearSSL libraries. The way that the firmware was designed is listed in the following steps. In the setup stages, the microcontroller utilizes the SSID and password credentials to connect to the access point. Then the HTTP client constructs a packet containing the sensor information in the form of a JSON payload in addition to other relevant HTTP headers. To enable this packet to be transmitted and securely received by AWS, the BearSSL library [16] allows using an SHA-1 fingerprint to enable sending HTTPS POST requests by taking care of the authentication. The firmware is designed to perform this process on the microcontroller every 15 minutes and send the data gathered to the AWS pipeline, picked up by the AWS API Gateway.

When a POST request is made, it is made to an AWS API Gateway endpoint [8] set up to trigger an AWS Lambda function [9] created to parse and process the data sent. When activated, this function parses the POST request’s payload, containing the sensor information and readings, and puts it into the database. The lambda function also checks to see if the sensor readings indicate a potential wildfire. If so, it will send a notification to all users who have subscribed to notifications. The integration of the database and SNS capabilities is possible by leveraging various services of AWS. The database used is DynamoDB [4], Amazon’s NoSQL database, a strong choice for unfiltered, constant data. The notification service is Amazon’s Simple Notification Service [10], which can be programmatically accessed to send notifications. Programmatically modifying the database and notification service is done through the Lambda function using the boto3 library in Python. Furthermore, email notifications are enabled to alert the user to abnormal temperature readings. The notification occurs when the Lambda function is executed. As the data is processed and put into the database, they are also checked for irregularities, spikes, and dips in temperature and humidity.

The design initially contained both the power and the sensors on one board in terms of the hardware. During phase one testing, the LDO’s heat dissipation quickly became a big issue. Later, it was found out that this LDO required a specific heatsink layout design to function in the “normal range.” The power supply eventually moved onto its separate board partly for debugging purposes. On the second prototype PCB, larger copper pads were added surrounding

the LDO and managed to dissipate excess energy. It is also worth noting that multiple LEDs and connectors were added to help identify and isolate electrical issues. A p-channel and an n-channel MOSFET along with multiple resistors were also added as an additional output to measure the power supply's voltage. Figure #1 is the top-level schematic of the power subsystem, whereas Figure #2 delves deeper into the voltage measurement subcircuit.

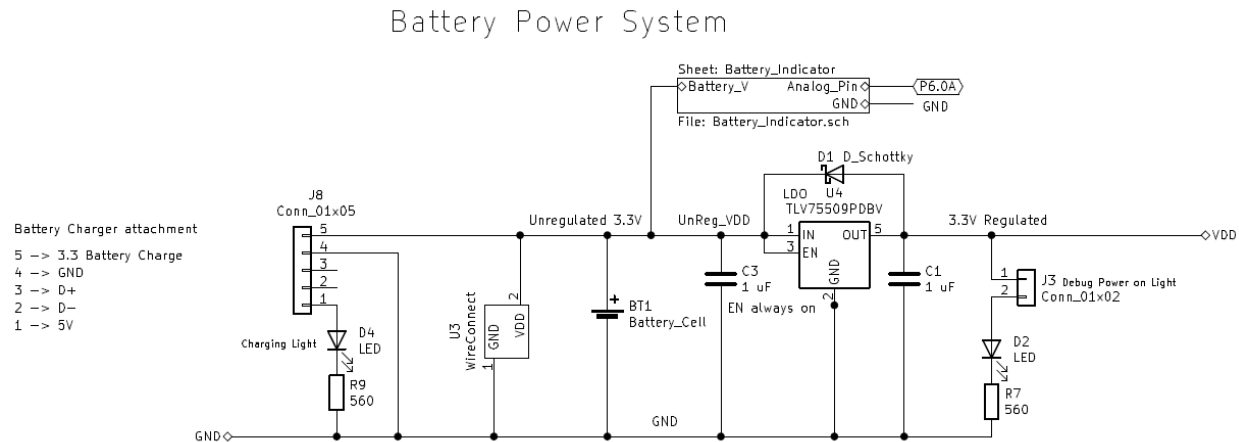


Figure #1: Battery Power System Schematic

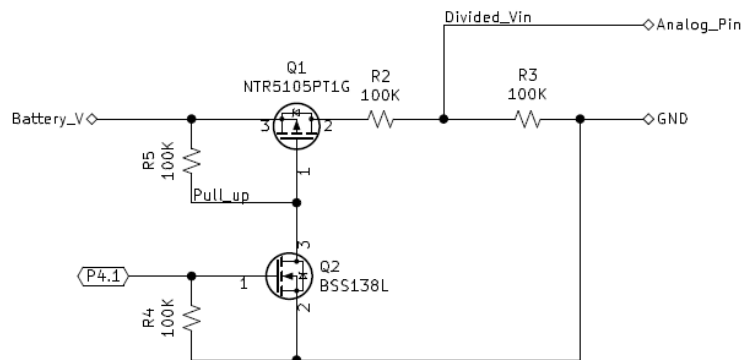


Figure #2: The Battery Level Indicator Schematic

Much remained the same as the original design on the sensor board, except for the additional debugging LEDs, load resistors, and connectors. Figure #3 shows the top-level schematic of both the humidity and the gas sensor. At the same time, Figure #4 takes a closer look inside the humidity sensor's hierarchical block and explores the gas sensor's inner circuit. Both sensors communicate through the I<sup>2</sup>C bus and therefore require the same pull-up resistors and bypass capacitors. It was realized not until phase two testing, unfortunately, the two sensors could not be connected to the I<sup>2</sup>C bus simultaneously; physical isolation was needed to avoid interference. The result became a small change to the original sensor node design; the gas sensor was cut out for the humidity sensor to function correctly. Figure #9 in the appendix shows a printout of the PCB layout.

## Sensor Subsystem

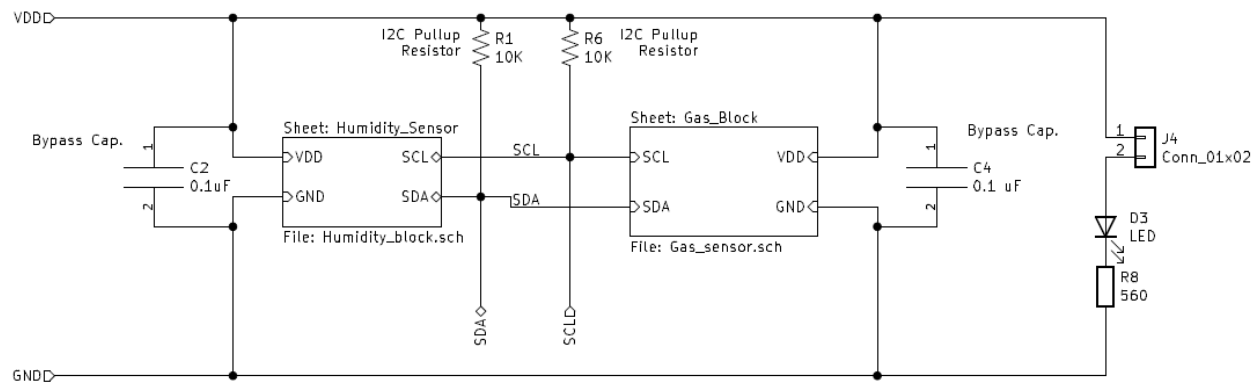
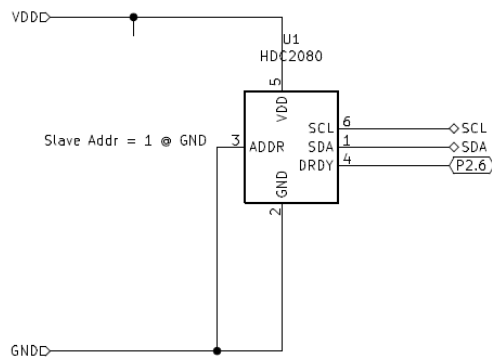


Figure #3: Sensor Subsystem Schematic

## Humidity Sensor Config



## The Gas Sensor Config

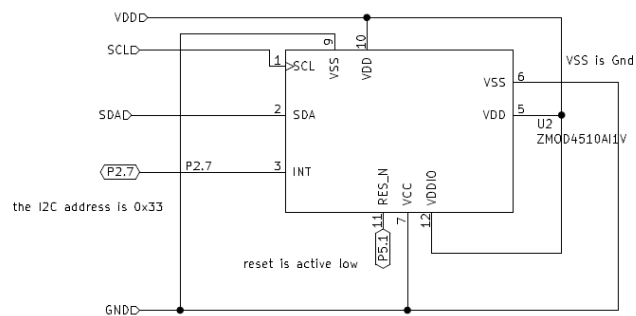


Figure #4: Sensor Configuration Schematics

The webpage (found at <https://ndd7xv.github.io/This-ButterBeWorth-It/>) is hosted on GitHub pages [22] and was created with the Angular [5] framework and Chart.js library [6]. For security, it accesses the database of sensing data through a federated Cognito identity for AWS DynamoDB [4] read permissions. It queries the database mentioned above for each node's past ten readings, which comes in 15-minute increments. Note that the dashboard will only display data when a sensor is running.

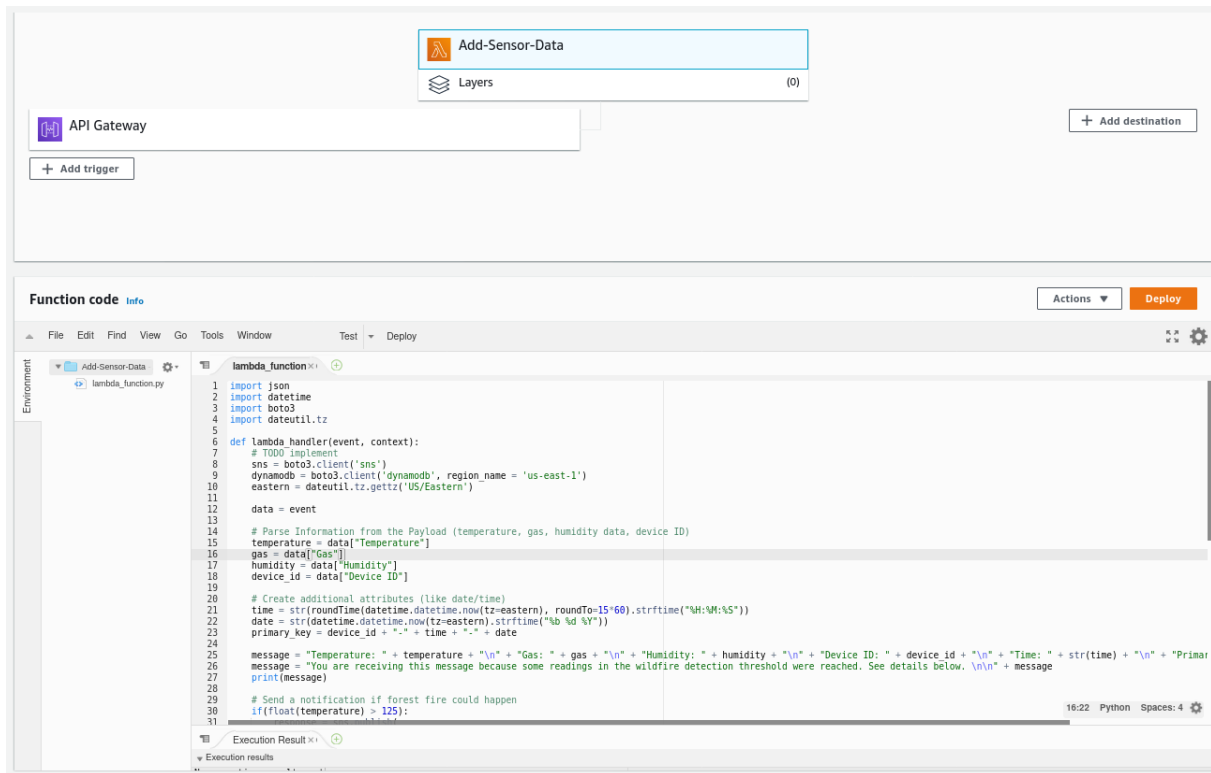


Figure #5: Lambda Visualization and Code

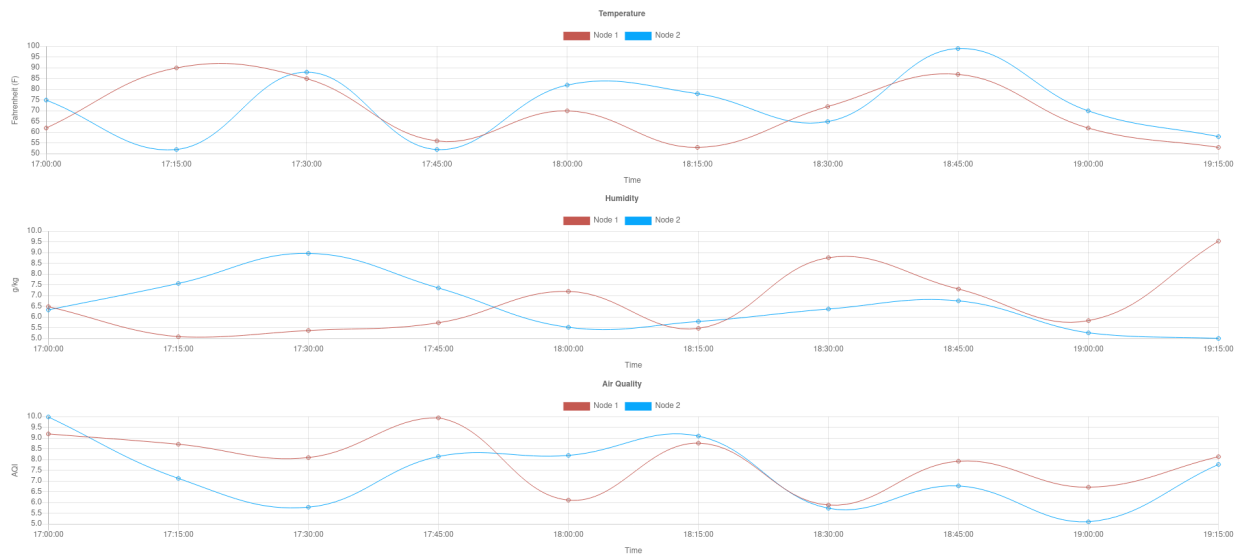


Figure #6: Visual Dashboard of Sensor Data

Using a POST request to transmit data was a choice between ease of security and implementation/scalability. In the future, both factors could be improved. For the sake of time constraints and troubles integrating embedded firmware into AWS, the POST request was the most feasible approach to take. Using GitHub Pages to host the dashboard was chosen over an



AWS Elastic Compute Cloud (EC2) instance [23] because of cost. Still, AWS's extensive use was a decision made while keeping software/processing capabilities in mind. The ease of implementing variously related tasks like receiving, processing, and storing data, sending subscribers alerts, and getting the data onto a web page made Amazon Web Services the best suite of tools for the software challenges.

## Project Timeline

The Gantt chart from the initial proposal and the final chart can be seen in Figures #7 and 8 below. The final chart was more detailed than the one seen in the proposal as more definite tasks were added and outlined as the semester progressed. Furthermore, the outline for tasks, completed in parallel and a serial manner, was also determined in the final chart, which cannot be seen in the initial diagram. Additionally, throughout the semester, tasks that were not completed as expected on their end date were highlighted in red and were completed on the dates marked by the “extended” cell.



Figure #7: Initial Project Timeline

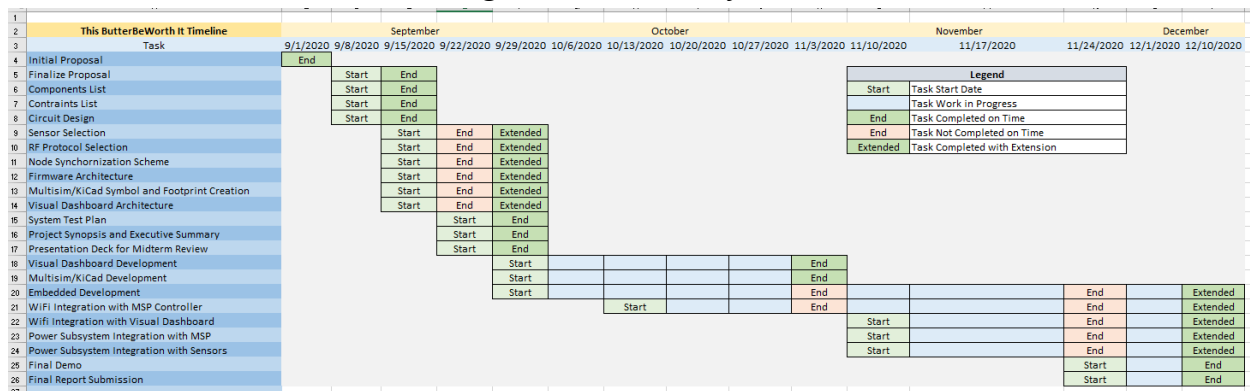


Figure #8: Final Project Timeline

Parallel tasks were executed alongside each other for the same timeframe; these tasks can be distinguished by reading the final chart vertically and finding tasks that overlap for the

specific period. For example, it can be seen that the following tasks were completed in a parallel manner as their time frame (9-15-2020 to 9-29-2020) overlap with each other: Sensor Selection, PF Protocol Selection, Node Synchronization Scheme, Firmware Architecture, Multisim/KiCad Symbol, Footprint Create, and Visual Dashboard Architecture. Serial tasks were initiated upon completing a prerequisite task; these can be distinguished in the final chart by looking at the end date (extended included) for a given task followed immediately by a new task's start date. For example, completing the Circuit Design task on 9-15-2020 initiated the new task for Sensor Selection (9-15-2020 to 9-29-2020).

The tasks were allocated among the team members based on their primary roles and secondary roles. As Nathan held the primary role of software architect and firmware support's secondary role, he contributed to the following tasks: Visual Dashboard Architecture, Firmware Architecture, Visual Dashboard Development, Wi-Fi Integration with ESP8266 microcontroller, and Wi-Fi Integration with Visual Dashboard. As Shreejan acted as the primary program manager, he was in charge of constructing and maintaining the Gantt charts and the timeline throughout the semester. Besides holding the secondary role of a software architect, Shreejan also contributed to the following task: Visual Dashboard Architecture, Visual Dashboard Development, and Wi-Fi Integration with Visual Dashboard. Tahmid's primary role was firmware architect; thus, he contributed to the following task: Node Synchronization Scheme, Firmware Architecture, and Wi-Fi Integration with ESP8266 microcontroller. Alec's primary role and Bill's second role were hardware architect; as such, they contributed to the following tasks: Circuit Design, PF Protocol Selection, Multisim/KiCad Symbol, and Footprint Creation, Multisim Development, Power Subsystem Integration with ESP8266 microcontroller, and Power Subsystem Integration with Sensors. Bill's primary responsibility dealt with embedded systems and design; thus, he contributed to Embedded Development.

Finally, all team members contributed to the remainder of the tasks as they met on call for substantial hours, all contributing to the final result. This includes the initial proposal, the final proposal, generating the components and constraints list, making system test plans, and the submission documents (the presentation deck also includes the midterm review, the final demonstration, and the final report).

## **Test Plan**

The test plans for the power subsystem, sensor subsystem, Wi-Fi firmware, and the visual dashboard can be referenced in Figures #12, #13, #14, and #15, respectively, in the Appendix. These test plans were developed as flow charts with two outcomes for each action, yes or no. Based on the test result in the current step, the next step's corresponding path was followed. If any issues or errors were found, incident responses were handled accordingly at that time to ensure appropriate steps were followed afterward.

### *Power Subsystem:*

The initial test plan seen in Figure #12 was followed through without much alteration; voltage (and current, if the voltage is inaccurate) was measured at both the input and the LDO output. However, during the prototype's testing, a significant heat dissipation design flaw was

found on the LDO and nearly led to a device change. Eventually, consultations led to close re-examinations of the datasheet, specifically at the recommended layout. Large copper pads were added to the PCB for the subsequent design in response to the findings. In addition, multiple diodes were added to prevent current backflow from endangering the battery, and LEDs were attached at critical input and output joints to facilitate the testing. The final prototype successfully solved the heat dispersion problem but produced the side-effect of “unlimited heat sink,” where solder became a lot more difficult to melt. This negative externality generated by the copper pads unsurprisingly led to connectivity issues with the LDO’s ground pin. Ultimately, this problem was solved with the extensive usage of an ohmmeter, and the power system finally began to deliver the desired 3.3V output. The circumstance caused considerable delays in the sensor configuration progress, but no deadline was changed to the timeline to stay on track.

#### *Sensor Subsystem:*

Though the power system’s complication caused a delay in the sensor development, the test plan was unchanged and followed through. As seen in Figure 13 in the proposed test plan, addressing and power are the two critical components for running the sensors. After the input voltage to the power system sensors was confirmed, the sensors were turned on from the microcontroller. The sensors’ data outputs were first measured using the NI VirtualBench digital analyzer to verify the reading. The incorrect addressing and significant noise interference led to cutting one of the sensors because this design did not have any isolation techniques between the sensors sharing the same I<sup>2</sup>C bus. Once the gas sensor was clipped and the humidity sensor was correctly verified functioning, the rest of the issues were addressed based on the microcontroller’s UART terminal’s output.

#### *Wi-Fi-Firmware:*

As seen in Figure #14, the original test plan included the use of MSP as the microcontroller for the Wi-Fi module. However, after initial success testing Wi-Fi capabilities with the MSP, errors were found integrating to the AWS IoT Core and subsequently AWS API Gateway to complete the visualization data pipeline. After countless attempts to resolve the errors, the decision was made to make the switch from the MSP432 to ESP8266. This decision was made as Tahmid, the firmware architect, had more experience working with the ESP microcontroller and developed more features with data transfer, including email notifications and SSL encryption from the Wi-Fi modules, AWS API Gateway, and IFTTT. Changes to the technical project, test plans, budget were made accordingly to account for this change. No significant changes to the timeline were made because the main functionalities left to be implemented the same as with the MSP microcontroller.

#### *Visual Dashboard*

As seen in Figure #15, preliminary testing was conducted with dummy test values to ensure functional connection from the microcontroller to AWS's database. After establishing a successful connection, storing the data into the database was tested. At this step, few errors were encountered, such as loss in data and undesirable format. These errors were fixed after changing the structure for the sensor data's primary keys, and little changes were made to the overall

process and test plan. In addition, testing the data transfer pipeline from the database to the visual database also threw minor syntax and logical errors and were resolved quickly. As stated above, a switch in microcontroller choice was made (from MSP432 to ESP8266), which subsequently resulted in the switch from using AWS IoT Core to integrate the data transfer pipeline to using AWS API Gateway using HTTPS POST requests.

## **Final Results**

The success criteria defined in the proposal can be seen in Figure #19 in the appendix. All expectations were met in each category's highest level in the implementation of the sensor monitoring system. The overall device was able to function solely on the power subsystem with a rechargeable battery. The power supply consistently delivers a 3.3 V output and draws no more than 500 mA of current, satisfying all requirements for successfully powering the entire unit described in the proposal.

Regarding the Data Acquisition and Transmission, information from the sensors was indeed gathered from the sensors, processed by the microcontroller, packetized into JSON, sent via Wi-Fi (using an HTTPS POST request) to AWS Cloud API, and then correctly channeled to the database using the implemented transfer pipeline. This firmware category lies in this project's heart, and it is the cornerstone of the entire system. Though one of the two sensors (the gas sensor) was ultimately not included in the finished prototype, this design met the primary communication goal over the I<sup>2</sup>C bus. A battery level indicator was added as an essential feature indicating any device's state. A bonus feature was added in battery indication, that the indicator would output a voltage of “0” when the device is charging.

The data collected was successfully visualized in the dashboard in real-time and in a clear and easily readable manner. The visual dashboard also included a warning system (involving the email notification system as seen in our video demonstration) for any measurements that crossed the predefined thresholds, thus satisfying all the requirements outlined in the rubric for this category.

## **Costs**

The cost and bill of materials for the hardware subsystems can be found in Figures 16, 17, and 18 in the Appendix. As indicated in the budget breakdown (Figure #16), the total cost for the project came to approximately \$394, while the total cost of the build for two units (microcontrollers with Wi-Fi modules, sensors and power system, assembly labor cost, and cloud service) was approximately \$334. This resulted in an average of \$167 in total per unit, including the cloud service fee. The average cost may seem jaw-dropping at first, but this conservative calculation represented the total cost of the entire project, and the average per unit took into account the cloud service's fixed cost. In reality, the average cloud service fee would only decrease as the quantity of units increases, rendering it insignificant in the long run. The total production cost of the two sensor nodes in this project, excluding the cloud fee, comes down to \$283, averaging around \$142 each. However, this average still seems exceptionally high for a supposedly low-power and low-cost device, and that is because half of this cost is from the MSP432 and its CC3120 Wi-Fi Booster Pack. Different microcontroller choices will

significantly impact the average cost per unit. Looking at the PCB boards only, the two prototypes cost around \$164, averaging \$82 each. With further breakdown, the PCB fabrication was \$36 each, the assembly fee totaled approximately \$63, averaging to less than \$32 each, while the BoM for each unit came just above \$41 buying in cut-tape.

For the production of large quantities, such as 10,000 units of sensor nodes (including the power supply and the sensor module), however, could tremendously decrease the cost. Not only does DigiKey offer a lower unit price for reel-tape per BoM, but finding an appropriate long term production point would also adjust PCB fabrication and assembly labor accordingly. The BoM's estimations for mass production can be seen in Figure #17 below based on Digikey's pricing. A conservative estimate of the PCB fabrication and assembly labor should cut their current cost in half, averaging a total just shy of \$50 per sensor unit.

The switch from MSP32 and CC3120 to ESP8266 is an even more cost-saving measure as the per-unit cost of an ESP8266 is only around \$3~\$6. In contrast, the MSP432 and CC3120 (requiring the integration of 2 processors into the custom PCBs) came up to a total of around \$14~\$16 in reel-tape. Furthermore, all the firmware running on the embedded system utilized free and open-source libraries, averting potentially expensive multi-year SDK contracts with companies like Texas Instruments. Doing so also ensured the possibility to maintain the firmware even if the companies were to abandon their SDK. Besides, improvements in manufacturing and the increase in automation would also decrease each sensor node's unit cost. To come up with an exact estimate for these changes would require much more research about the manufacturer and the market beyond this project's scope.

Regarding the visual dashboard, the highest cost would be for the use of AWS cloud services (storage, read and write to DynamoDB, etc.) and external applications required for the implementation of the data transfer pipeline. For this project, \$50 was allocated towards the complete development of the visual dashboard. But realistically, improvements and maintenance for a sizable sensor system will most likely see a substantial increase in the cost and, therefore, will need to be accounted for in the long-term planning to find the optimal production point on the cost curve.

## **Future Work**

In retrospect, there were a variety of pitfalls and difficulties completing the project. While the established goals were achieved, a lot of time and effort was spent to address a variety of problems that, while seemingly inevitable, could have been anticipated and mitigated. This project's underlying concept and the system can be expanded upon, howbeit its standing professionalism, in terms of costs, security, and performance/power efficiency to be less of a proof-of-concept and more of a commercialized product.

To elaborate on improvement and expansions, it is essential to establish that this project involved creating a system for wildfire detection using sensing nodes composed of microcontrollers. In achieving this, hardware was designed to develop a sensing node, embedded firmware was programmed to obtain temperature/gas data, and cloud infrastructure was created to receive information and display it on a webpage. However, to make a commercially viable

product, improvements to reducing each sensing node's cost would be necessary. In the same vein, securing transmitted data and the cloud infrastructure that supports the processing and storage of said data would also need to be deliberately designed. While having a cloud-native environment makes the system optimal for scalability, further changes to the firmware and the cloud database and API would be required to support a more extensive network of sensors and data.

In addition, the physical capabilities of the system were also minimal; while it can be incredibly insightful to know the gas and temperature information around a node, the use of machine learning to process such data can be used to make informed decisions about *where* and *when* a fire might occur. Besides, if physically independent, additional sensors (such as the second initial gas sensor) can also be added to the system, providing even more informative data. The system also uses Wi-Fi, which has a more limited range and greater power consumption than some other communication protocols, like Zigbee or LoRa. The high-power usage was made more problematic because each sensing node is powered by a battery, which would need to be replaced once drained. Therefore, in future iterations of the system, it could be beneficial to look into renewable energy sources, such as solar power.

Ultimately, all of these concerns could be addressed if a future group of students wanted to create a similar project based on this one; the challenges encountered with this project came more from designing and implementing things from scratch than power, security, and capability issues. Regardless of specifications, any similar project should expect to run into all sorts of problems. The trouble with advising future groups of students on the tribulations encountered during this project is that there are many variables at play simultaneously, and it is impossible to say what specific parts of a project will throw what error when. A perfect example is the I<sup>2</sup>C bus configuration; multiple sensors cannot talk to the microcontroller simultaneously. Though this is well-known, it is often hard to pinpoint the issue in an integration environment. While all groups were warned that hardware and software problems would be inevitable and time-consuming, such problems that ended up using more time troubleshooting than anticipated still occurred. For example, connecting the webpage to the database took less time than perceived, and connecting the device to the internet took a lot longer than planned. Being conscious of the constraints and the reality of a situation is a must. There were various ways to transmit sensor information to the database, and the final choice to implement was ultimately due to the time and design constraints.

Finally, it is also helpful to have a fleshed-out game plan initially, but it is also necessary to be flexible and assume anything can go wrong because it will. With so many potential problems and so many different solutions with various limitations, one should never take for granted that the final product will be the way it was envisioned in the beginning. It is crucial to think, design, and solve like an engineer, after all.

## References

- [1] A. Alkhatib, "Forest Fire Monitoring," Forest Fire, pp. 53-72, DOI: 10.5772/intechopen.72059
- [2] Aeris. 2020. Fight Forest Fires With Tech: How Forest Fire-Prone Regions Leverage IoT To Limit Wildfires And Mitigate Destruction. [online] Available: <https://www.aeris.com/news/post/fight-forest-fires-with-tech-how-forest-fire-prone-regions-leverage-iot-to-limit-wildfires-and-mitigate-destruction/>
- [3] Espressif Systems, "ESP8266EX Datasheet," ESP8266 Datasheet, Dec-2015. [Online]. Available: [https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf). [Accessed: 10-Dec-2020].
- [4] "What Is Amazon DynamoDB? - Amazon DynamoDB", *Docs.aws.amazon.com*, 2020. [Online]. Available: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>. [Accessed: 10- Dec- 2020].
- [5] "Angular Documentation." Angular.io. <https://angular.io/docs> (accessed 13 September 2020).
- [6] "Introduction · Chart.js documentation", *Chartjs.org*, 2020. [Online]. Available: <https://www.chartjs.org/docs/latest/>. [Accessed: 10- Dec- 2020].
- [7] "Electronics Donation and Recycling." The United States Environmental Protection Agency | US EPA. <https://www.epa.gov/recycle/electronics-donation-and-recycling> (accessed Sep. 13, 2020).
- [8] "What is Amazon API Gateway? - Amazon API Gateway", *Docs.aws.amazon.com*, 2020. [Online]. Available: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>. [Accessed: 10- Dec- 2020].
- [9] "What is AWS Lambda? - AWS Lambda", *Docs.aws.amazon.com*, 2020. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. [Accessed: 10- Dec- 2020].
- [10] "What is Amazon SNS? - Amazon Simple Notification Service", *Docs.aws.amazon.com*, 2020. [Online]. Available: <https://docs.aws.amazon.com/sns/latest/dg/welcome.html>. [Accessed: 10- Dec- 2020].
- [11] Texas Instruments, CC3120 SimpleLink™ Wi-Fi® Wireless Network Processor, Internet-of-Things Solution for MCU Applications, Feb-2017. [Online]. Available:

[https://www.ti.com/lit/ds/symlink/cc3120.pdf?ts=1607635592646&ref\\_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FCC3120](https://www.ti.com/lit/ds/symlink/cc3120.pdf?ts=1607635592646&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FCC3120). [Accessed: 10-Dec-2020].

- [12] Texas Instruments, MSP432P401R, MSP432P401M SimpleLink™ Mixed-Signal Microcontrollers, Mar-2015. [Online]. Available: [https://www.ti.com/lit/ds/symlink/msp432p401r.pdf?ts=1607568848716&ref\\_url=https%253A%252F%252Fwww.google.com%252F](https://www.ti.com/lit/ds/symlink/msp432p401r.pdf?ts=1607568848716&ref_url=https%253A%252F%252Fwww.google.com%252F). [Accessed: 10-Dec-2020].
- [13] "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," in IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999), vol., no., pp.1-1076, 12 June 2007, doi: 10.1109/IEEESTD.2007.373646.
- [14] Espressif Systems, "esp8266/Arduino," GitHub. [Online]. Available: <https://github.com/esp8266/Arduino>. [Accessed: 10-Dec-2020].
- [15] Espressif Systems, "ESP8266 Arduino Core's documentation," *Welcome to ESP8266 Arduino Core's documentation! - ESP8266 Arduino Core documentation*. [Online]. Available: <https://arduino-esp8266.readthedocs.io/en/latest/>. [Accessed: 10-Dec-2020].
- [16] T. Pornin, "earlephilhower/bearssl-esp8266," GitHub. [Online]. Available: <https://github.com/earlephilhower/bearssl-esp8266>. [Accessed: 10-Dec-2020].
- [17] Texas Instruments. Code Composer Studio User Guide (2019). Accessed: Sep. 13 2020. [Online]. Available: [https://software-dl.ti.com/ccs/esd/documents/users\\_guide/index.html](https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html)
- [18] "KiCad EDA", *Kicad.org*, 2020. [Online]. Available: <https://www.kicad.org/>. [Accessed: 10-Dec- 2020].
- [19] National Instruments. NI Multisim User Manual (2009). Accessed: Sep. 13 2020 [Online]. Available: <https://www.ni.com/pdf/manuals/374483d.pdf>
- [20] The Arduino Team, *Arduino IDE*. [Online]. Available: <https://www.arduino.cc/en/software>. [Accessed: 10-Dec-2020].
- [21] "Documentation for Visual Studio Code." Visual Studio Code - Code Editing. Refined. [https://software-dl.ti.com/ccs/esd/documents/users\\_guide/index.html](https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html) (accessed Sep. 13, 2020).
- [22] "GitHub Pages", *GitHub Pages*, 2020. [Online]. Available: <https://pages.github.com/>. [Accessed: 10- Dec- 2020].
- [23] "What is Amazon EC2? - Amazon Elastic Compute Cloud", *docs.aws.amazon.com*, 2020. [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>. [Accessed: 10- Dec- 2020].



- [24] R. Bohn. "Nema Rating Buying Guide." NEMA Enclosures.  
<https://www.integraenclosures.com/resources/about-nema-enclosure-types/> (accessed Sep. 13, 2020).
- [25] Forest Fire Monitoring System. <https://patents.google.com/patent/CN104658156A/en>.  
Accessed 10 Dec. 2020.
- [26] Forest Fire Monitoring and Early Warning System Based on IOT.  
<https://patents.google.com/patent/CN202472841U/en>. Accessed 10 Dec. 2020.
- [27] Wireless Home Fire and Security Alarm System.  
<https://patents.google.com/patent/US6624750B1/en>. Accessed 10 Dec. 2020.

## Appendix

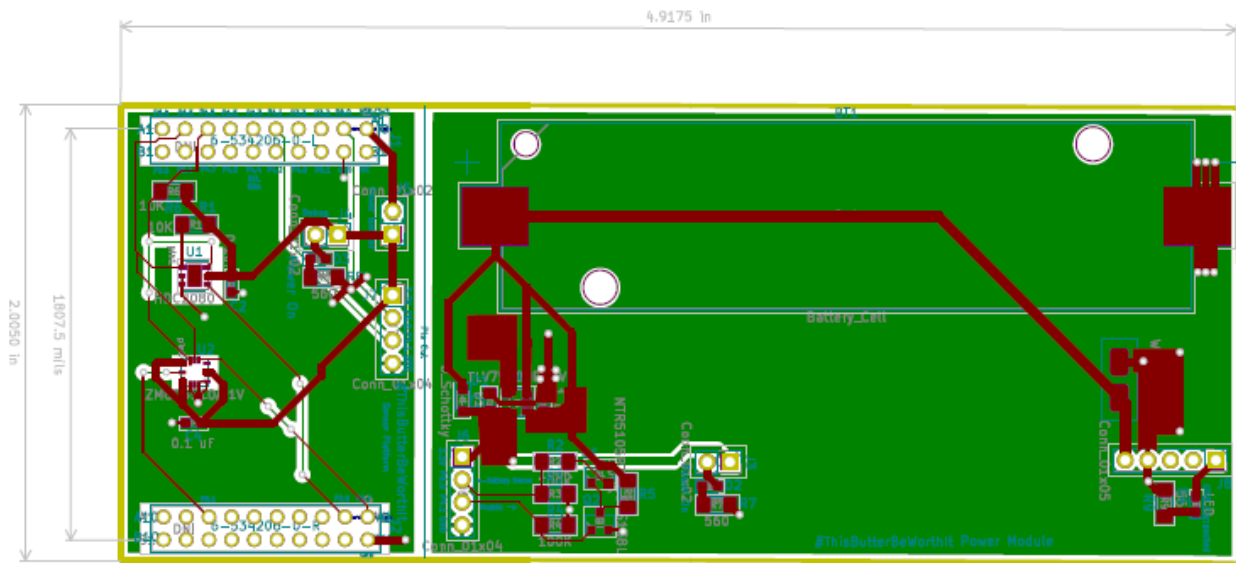


Figure #9: PCB Design

```
14:59:22.411 -> [SETUP] WAIT 3...
14:59:23.414 -> [SETUP] WAIT 2...
14:59:24.409 -> [SETUP] WAIT 1...
14:59:25.475 -> Battery ADC reading: 835 Battery value: 3.47 V
14:59:26.471 -> Temperature (C): 23.67 Relative Humidity (%): 48.72
14:59:26.471 -> Begin wifi connection..
14:59:26.471 -> [HTTPS] begin...
14:59:26.471 -> [HTTPS] POST...
14:59:27.836 -> [HTTPS] POST... code: 200
14:59:27.836 -> {"statusCode": 200, "body": "\"Hello from Lambda!\""}
14:59:32.874 -> Battery ADC reading: 841 Battery value: 3.50 V
14:59:33.905 -> Temperature (C): 23.68 Relative Humidity (%): 48.13
14:59:33.905 -> Begin wifi connection..
14:59:33.905 -> [HTTPS] begin...
14:59:33.905 -> [HTTPS] POST...
14:59:35.277 -> [HTTPS] POST... code: 200
14:59:35.277 -> {"statusCode": 200, "body": "\"Hello from Lambda!\""}
14:59:40.349 -> Battery ADC reading: 842 Battery value: 3.50 V
14:59:41.364 -> Temperature (C): 23.68 Relative Humidity (%): 47.96
14:59:41.364 -> Begin wifi connection..
14:59:41.364 -> [HTTPS] begin...
14:59:41.364 -> [HTTPS] POST...
```

☒ Autoscroll ☒ Show timestamp

Newline 115200 baud Clear output

Figure #10: 200 HTTP Response in UART Terminal

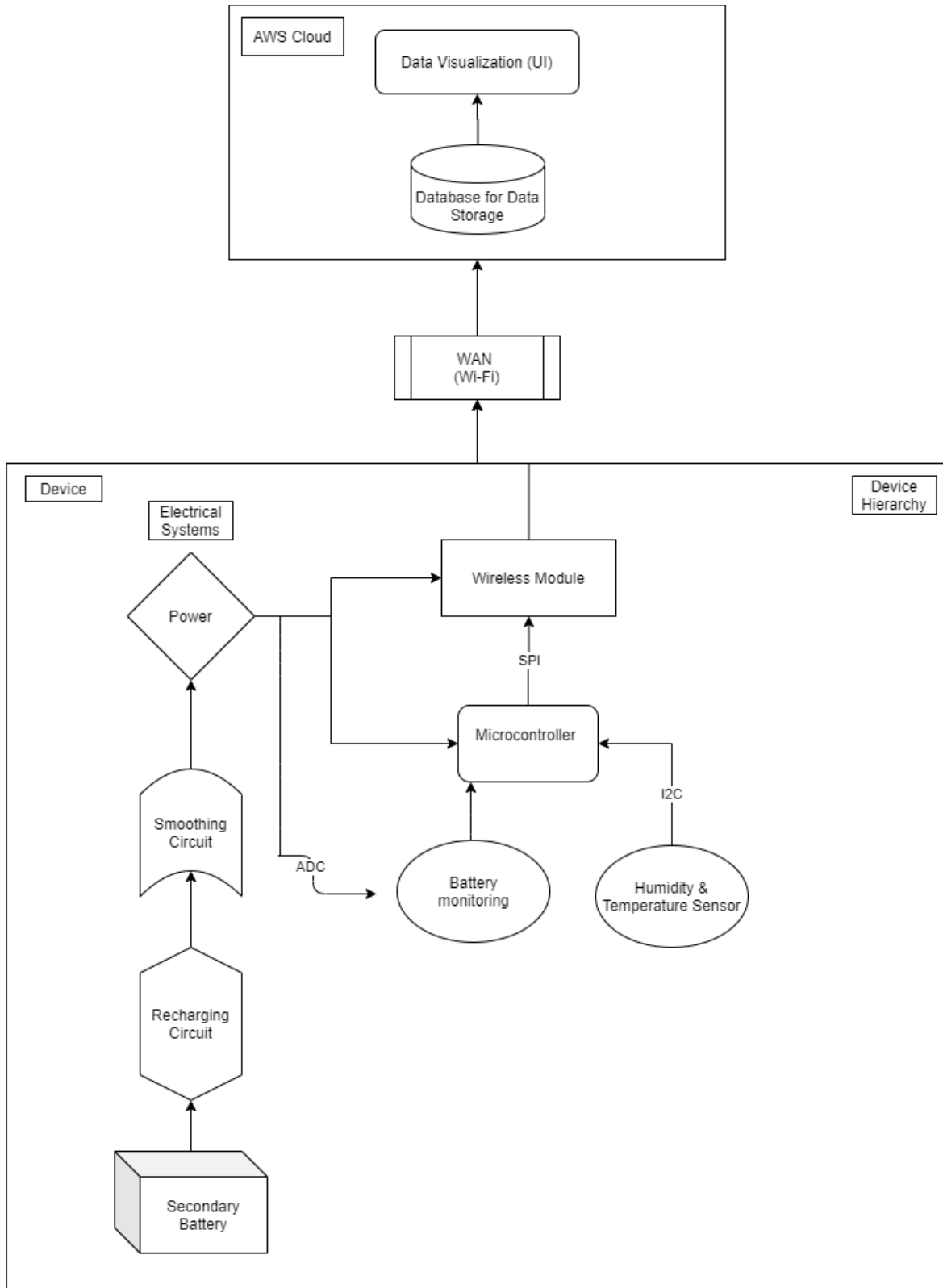


Figure #11: Project Design Overview

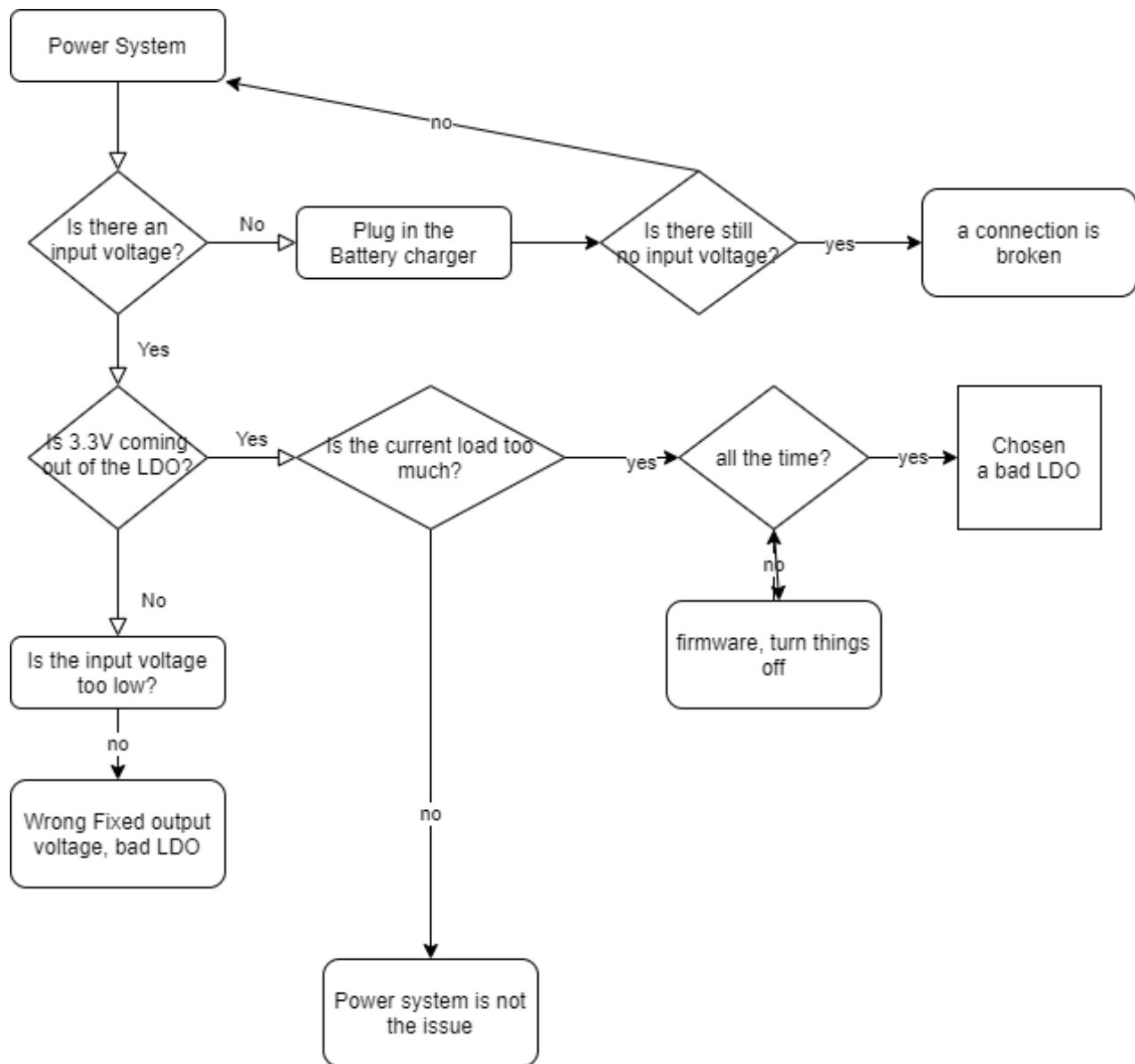


Figure #12: Power Subsystem Test Plan

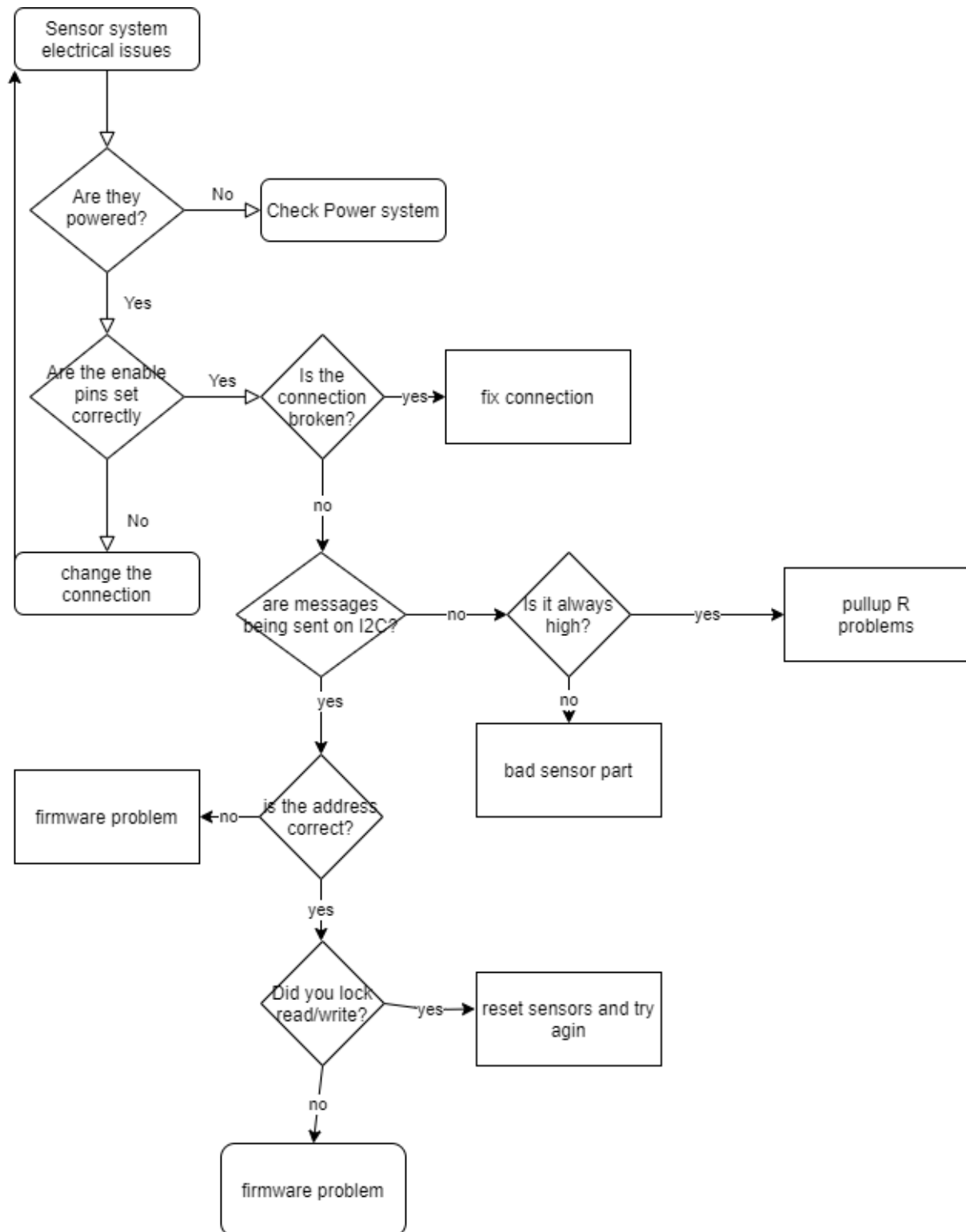


Figure #13: Sensor Subsystem Test Plan

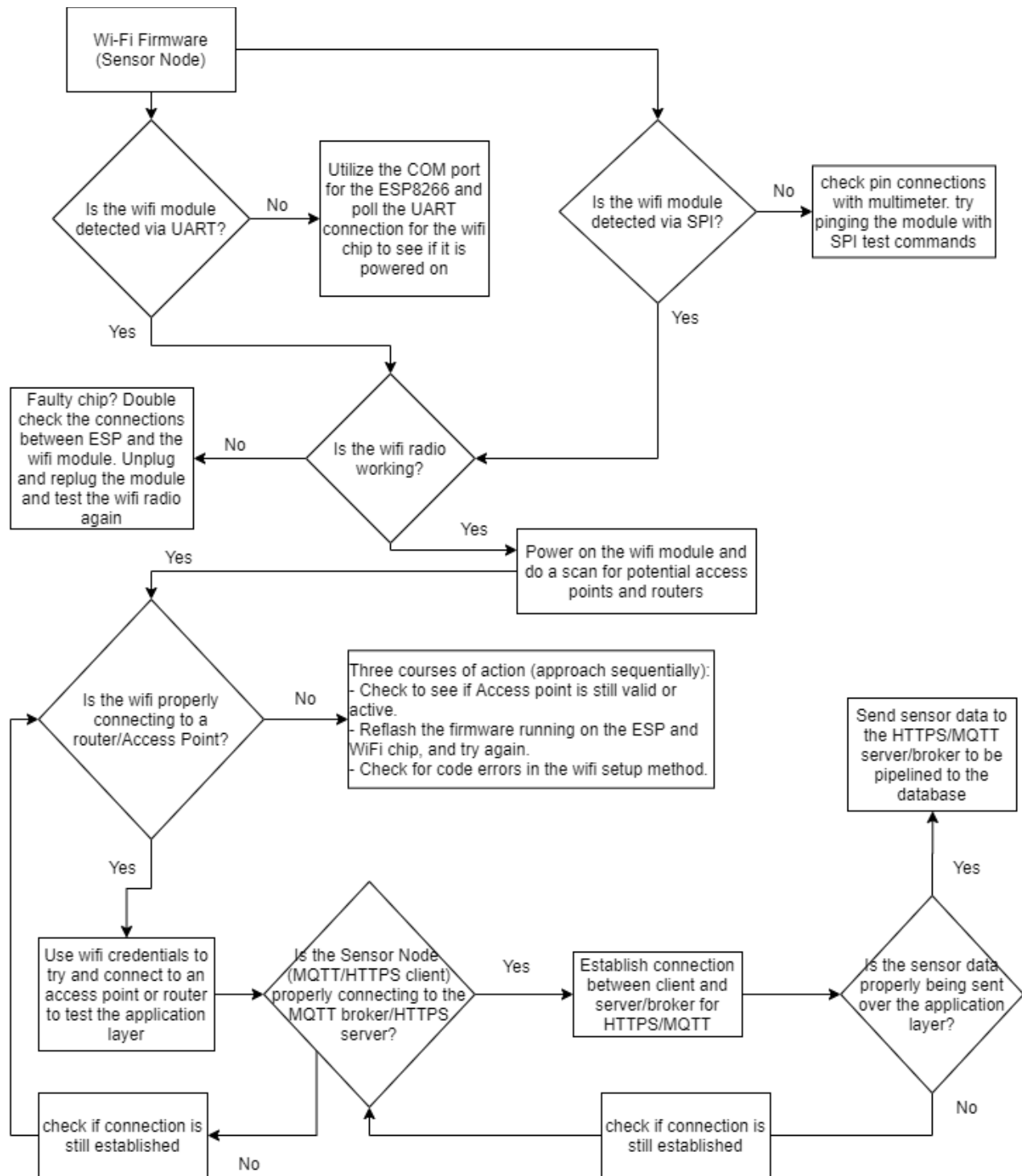


Figure #14: Firmware Test Plan

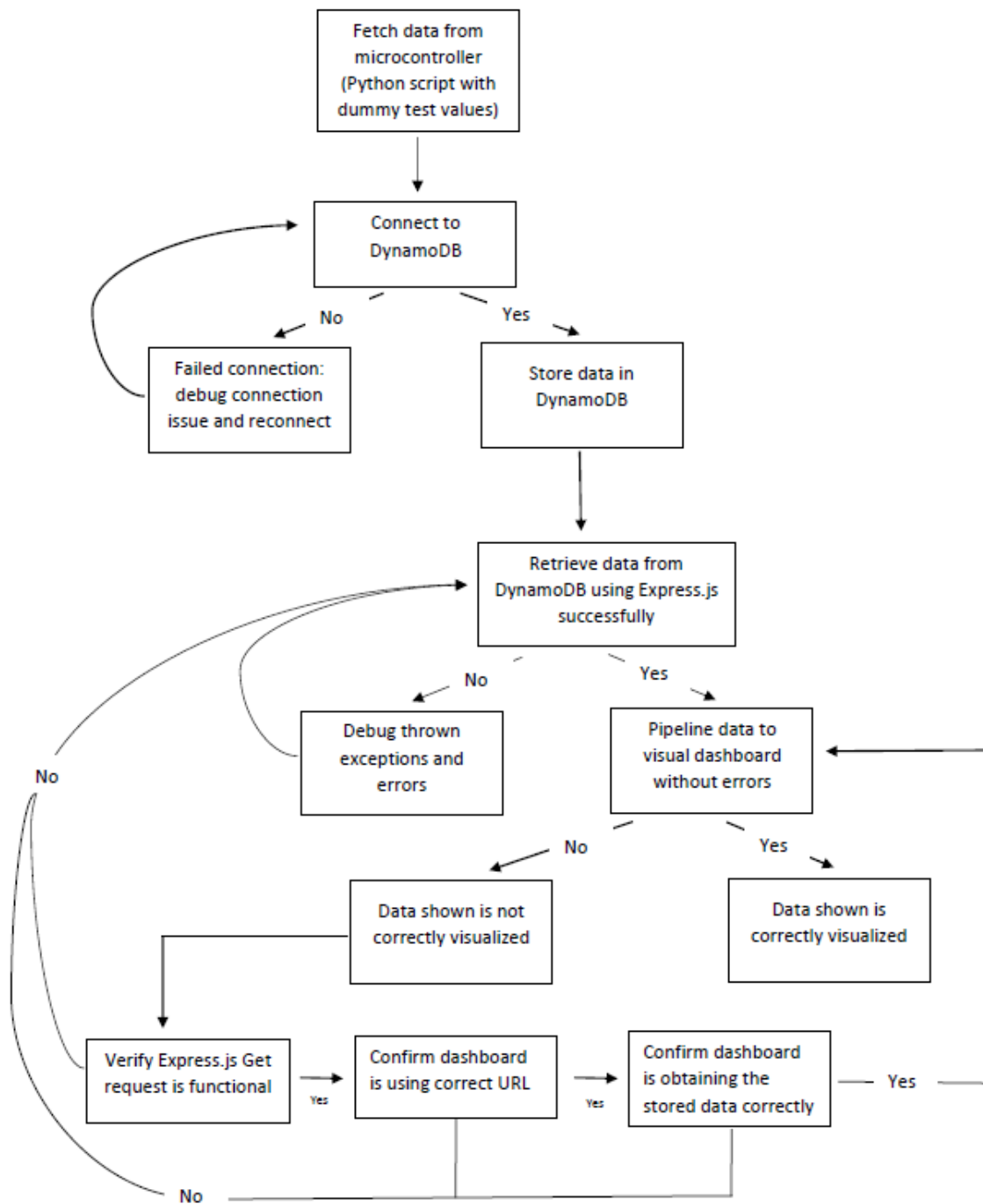


Figure #15: Visual Dashboard Test Plan

### Figure #16: Total Budget Breakdown

Figure #17: DigiKey Bill of Materials

### Figure #18: Mouser Bill of Materials



## Expectations

Points	Power	Data Acquisition/ Transmission	Data Visualization
3	The device functions off of a rechargeable battery	Information gathered properly in the database from the many wirelessly connected sensor nodes	Data is presented in real-time and in an aesthetic and understandable manner
2	The device functions off of a battery, but it cannot recharge	Data is transmitted from the sensor but is not properly integrated into the database	Data is presented in real-time, but with some limitations. Some information is difficult to read
1	The device is not able to use a battery and requires wired power	Data is not transmitted at all, or the sensor fails to acquire data and communicate with the microcontroller	Data is not presented in real-time, and information is presented with minimal GUI
0	The device is not able to receive power properly	Data is not transmitted at all, and sensors fail to communicate with the microcontroller	Data cannot be seen from any form of GUI

Points	Grade
7-9	A
5-6	B
3-4	C
0-2	D

Figure #19: Proposal Expectations (Rubric)

```

import json
import datetime
import boto3
import dateutil.tz

def lambda_handler(event, context):
    sns = boto3.client('sns')
    dynamodb = boto3.client('dynamodb', region_name = 'us-east-1')
    eastern = dateutil.tz.gettz('US/Eastern')

    data = event

    # Parse Information from the Payload (temperature, gas, humidity data, device ID)
    temperature = data["Temperature"]
    gas = data["Gas"]
    humidity = data["Humidity"]
    device_id = data["Device ID"]

    # Create additional attributes (like date/time)
    time = str(roundTime(datetime.datetime.now(tz=eastern), roundTo=15*60).strftime("%H:%M:%S"))
    date = str(datetime.datetime.now(tz=eastern).strftime("%b %d %Y"))
    primary_key = device_id + "-" + time + "-" + date

    message = "Temperature: " + temperature + "\n" + "Gas: " + gas + "\n" + "Humidity: " + humidity + "\n" + "Device ID: " + device_id + "\n"
    + "Time: " + str(time) + "\n" + "Primary Key: " + primary_key
    message = "You are receiving this message because some readings in the wildfire detection threshold were reached. See details below. \n\n"
    + message

```

```

# Send a notification if forest fire could happen
if(float(temperature) > 125):
    response = sns.publish(
        TopicArn='arn:aws:sns:us-east-1:064764567903:This_ButterBeWorth_It',
        Message= message,
        Subject='Lambda Triggered with the API Gateway',
    )

# Update the database
add_to_db = dynamodb.put_item(
    TableName = 'test',
    Item = {
        'primaryKey' : {'S': primary_key},
        'time' : {'S': time},
        'deviceId' : {'S': device_id},
        'date' : {'S': date},
        'temp' : {'S': temperature},
        'humidity' : {'S': humidity},
        'gas' : {'S': gas},
    })

return {
    'statusCode': 200,
    'body': json.dumps('Hello from Lambda!')
}

def roundTime(dt=None, roundTo=60):
    if dt == None : dt = datetime.datetime.now()
    seconds = (dt.replace(tzinfo=None) - dt.min).seconds
    rounding = (seconds+roundTo/2) // roundTo * roundTo
    return dt + datetime.timedelta(0,rounding-seconds,-dt.microsecond)

```

Figure #20: Lambda Function and Dashboard Code\*

\*For the sake of readability, the code for the dashboard is not all listed out. It can be found at <https://github.com/ndd7xv/This-ButterBeWorth-It> - the most crucial files are *src/app* and are *linegraph/linegraph-component.ts*, *app.component.ts*, *app.component.html*, and *data.service.ts*.

```

1
2  /*****
3
4  Copyright 2020 "This ButterBeworth It"
5  Author: Nathan Do, Shreejan Gupta, Tahmid Kazi, Alec Ross, and Bill Yang
6
7  *****/
8
9  #include <Arduino.h>
10 #include <ESP8266WiFi.h>
11 #include <ESP8266WiFiMulti.h>
12 #include <ESP8266HTTPClient.h>
13 #include <WiFiClientSecureBearSSL.h>
14 #include <HDC2080.h>
15
16 /* Macros */
17 #define ADDR 0x40
18 const uint8_t fingerprint[20] = {0x0B, 0x15, 0xD6, 0x33, 0x82, 0xF8, 0xBA, 0x15, 0xF7, 0xD5, 0x7A, 0x94, 0xE0, 0xE2, 0xAD, 0xBB, 0x00, 0x95, 0xA4, 0xA5};
19 //Should be this ----> 0b 15 d6 33 82 f8 ba 15 f7 d5 7a 94 e0 e2 ad bb 00 95 a4 a5
20 ESP8266WiFiMulti WiFiMulti;
21
22 /* Constructor and the setup() function, triggers only once */
23 HDC2080 sensor(ADDR);
24
25 void setup() {
26     // Start the serial communication
27     Serial.begin(115200);
28     sensor.begin();
29     // Begin with a device reset
30     sensor.reset();
31     pinMode(D3, OUTPUT);
32
33     Serial.println();
34     Serial.println();
35     Serial.println();
36
37     // Setting up the Wi-Fi
38     for (uint8_t t = 4; t > 0; t--) {
39         Serial.printf("[SETUP] WAIT %d...\n", t);
40         Serial.flush();
41         delay(1000);
42     }
43
44     WiFi.mode(WIFI_STA);
45     WiFiMulti.addAP("SSID", "PASSWORD"); // <----- ENTER WIFI CREDENTIALS HERE
46 }
47
48 /* The loop() function, repeated after startup */
49 void loop() {
50     // Wait to establish Wi-Fi connection
51     digitalWrite(D3, HIGH);
52     delay(50);
53     int read_adc = analogRead(A0);
54     digitalWrite(D3, LOW);
55
56     // Starting the battery life measurement
57     double battery = ((double) read_adc * 0.0009765625) * 4.259;
58     Serial.print("Battery ADC reading: ");
59     Serial.print(read_adc);
60     Serial.print("Battery value: ");
61     Serial.println(battery + "V");
62
63     // Starting the sensor measurement
64     sensor.triggerMeasurement();
65     delay(1000);
66     float temp = sensor.readTemp();
67     float humidity = sensor.readHumidity();
68     Serial.print("Temperature (C): "); Serial.print(temp);
69     Serial.print("Relative Humidity (%): "); Serial.println(humidity);

```

```

70 Serial.println("Begin wifi connection...");
71
72 // Starting the HTTPS POST request
73 if ((WiFiMulti.run() == WL_CONNECTED)) {
74
75     std::unique_ptr<BearSSL::WiFiClientSecure>client(new BearSSL::WiFiClientSecure);
76     client->setFingerprint(fingerprint);
77     HTTPClient https;
78     Serial.print("[HTTPS] begin...\n");
79
80     if (https.begin(*client, "https://po98mu7785.execute-api.us-east-1.amazonaws.com/default/Add-Sensor-Data")) { // <----- HTTPS link
81         Serial.print("[HTTPS] POST...\n");
82
83         //Start connection and send HTTP header
84         https.addHeader("Content-Type", "application/json");
85         //Working: int httpCode = https.POST("{\"value1\":\"" + String(14) + "\",\"value2\":\"" + String(72) + "\",\"value3\":\"" + String(91) + "\"}");
86
87         //===== WHERE THE JSON IS ADDED TO THE PAYLOAD AND SENT TO AWS =====
88         int httpCode = https.POST("{\"Device ID\":\"" + String(2) + "\",\"Temperature\":\"" + String(temp) +
89         "\",\"Humidity\":\"" + String(humidity) + "\",\"Gas\":\"" + String(battery) + "\"}");
90         //===== Just change the numbers in String(x) to the int/float variables you are using for the I2C and you should be good to go =====
91
92         //httpCode will be negative on error
93         if (httpCode > 0) {
94             //HTTP header has been send and Server response header has been handled
95             Serial.printf("[HTTPS] POST... code: %d\n", httpCode);
96
97             //File found at server
98             if (httpCode == HTTP_CODE_OK || httpCode == HTTP_CODE_MOVED_PERMANENTLY) {
99                 String payload = https.getString();
100                 Serial.println(payload);
101             }
102         }
103         else {
104             Serial.printf("[HTTPS] POST... failed, error: %s\n", https.errorToString(httpCode).c_str());
105         }
106         https.end();
107     }
108     else {
109         Serial.printf("[HTTPS] Unable to connect\n");
110     }
111 }
112
113 /* Hard-RESET is the only viable option with deepSleep on ESP8266 */
114 Serial.println("Going to sleep for 5 seconds...");
115 delay(1000 * 60 * 15); //change for demo
116 //ESP.deepSleep(5000);
117
118 }
119

```

Figure #21: Firmware Code for Wi-Fi/HTTPS Communication & I<sup>2</sup>C Sensor Integration