EDIT FINAL DOC not this one

Authors: Jake Cerwin, Ryan Huang, Angela Zhang
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Currently there is no efficient way to find an open seat in within Carnegie Mellon University's shared spaces, especially during peak hours. Our project is a system designed around solving the current lack of an efficient way to find open seating within Carnegie Mellon University's open spaces by tracking table occupancy in real time and display current occupancy status on a web application, specifically focusing on the UC second floor. Our project is split into hardware design, hardware and software integration design, and software design. Each section has their own use-case requirements, summing up to five requirements relating to the battery life time, target number of users and devices supported at once and communication time. Our hardware solution consists of an OMRON MEMS thermal sensor and a given PIR sensor to sense occupation, as well as an ESP8266, a voltage regulating module, and a 6 Amp-Hour Lithium Ion battery. Each component was chosen to realistically satisfy the battery life requirement. The hardware-software integration mainly consists of the AWS IoT Core, which is allows for safe communication without risking credential leak as well as easy management and scalability of the IoT fleet of embedded systems. Using AWS IoT would hence satisfy the use-case requirements associated to the hardware-software integration aspect of our project as well as the software use-case requirement. Finally, our software implementation consists of using AWS RDS to hold our MySQL database, decreasing the volume of calls to our AWS server and freeing up more computing power to support 25 concurrent users. We are also using AWS EC2 to host the Django application, allowing us to program the frontend and backend in Python. We also completed a design trade study on ideas we've come upon and abandoned. In order to ensure the use-case requirements are satisfied, we also added tests and validation for each aspect of our project. Finally, we discuss our schedule for the rest of the semester and the Bill of Materials of our project.**

***Index Terms*—AWS EC2, AWS IoT, Battery,Design, Django, MySQL, Occupancy Tracking, PIR, Python, Seating, Sensor, Testing**

1 INTRODUCTION

Students at Carnegie Mellon University often struggle with finding a table to sit at on campus, mindlessly wandering from room to room looking for an open table often wasting time just to confirm there is nothing available. The scope of this project focuses on solving this issue within the second floor of the UC where the problem is particularly exacerbated.

The designed solution to the problem is to track table occupancy in real time and display this information within a web application available to anybody. This implementation will be such that when students arrive to the UC, or even before they proceed to head over, they can see if and where space is available thus reducing both congestion and wasted time.

The extent of our design at a high level is to have a fleet of occupancy sensors deployed on the underside of each table within the UC that detect whether that table is occupied or not. Each sensor's data is uploaded in real time to the cloud and organized in an easily understandable format that is made accessible through a web application to any user in real time. This design is visualized in Figure 1.

2 USE CASE REQUIREMENTS

Our project has 5 use-case requirements. These are broken down into three main subsystems that each govern a major aspect of the project's design: hardware, hardware-software communication, and software.

2.1 Hardware Requirement

The hardware subsystem only has one use case requirement. This requirement is that each component needs to have a minimum battery life of 55 active hours with a 52+ hour deep sleep battery life requirement. This is derived from the desire for ultimate freedom and flexibility in terms of where the sensors are placed. The specific numbers of hours are derived from the existing Google Maps data on visitation rates of the Cohen University Center as displayed in Figure 2. This data indicates peak hours within the UC are from 9am to 8pm on weekdays with relatively minimal crowds at all other times. As battery changing is often annoying to the system's administrators, we extrapolated from this data that we would at minimum need to support 55+ hours of active use during peak occupancy times and 52+ hours of deep sleep battery life during non peak weekday hours to confine all battery changes to once a week.

2.2 Hardware-Software Communication Requirements

The hardware-software communication subsystem has two use case requirements, which stem from balancing hardware power saving with the speed and reliability of an occupancy status update within the user's web application experience. The first of these two requirements is that

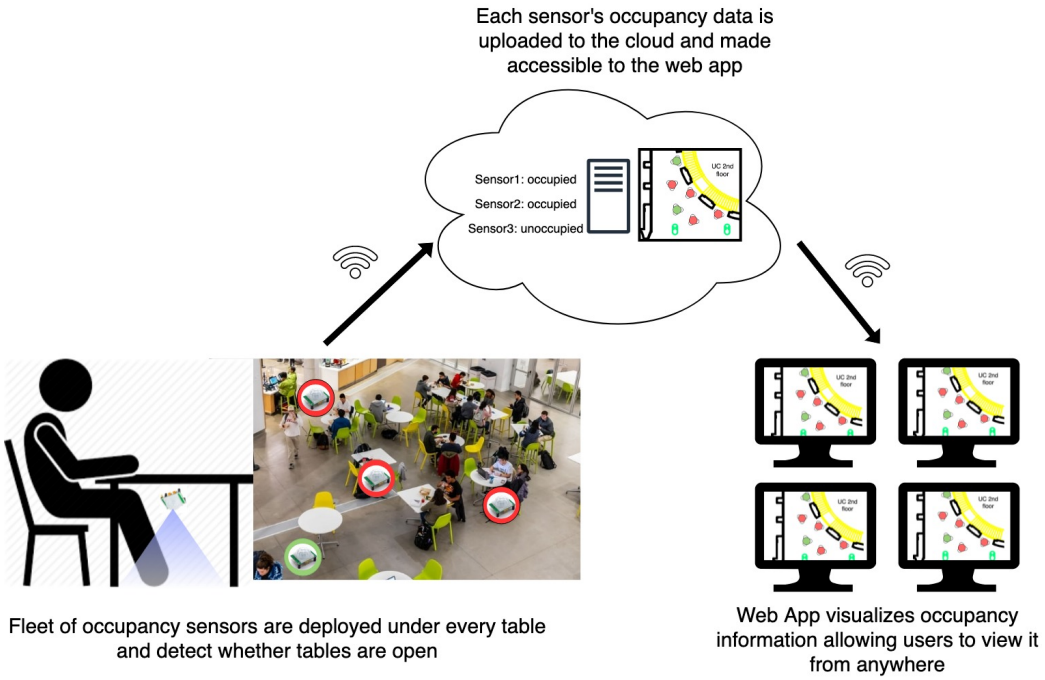

Figure 1: High level system overview

occupancy status as recognized by the hardware detecting device is accurately updated on the web in no more than 1 minute. This metric was derived from not only the necessity of ensuring a faster alternative to student's mindlessly wandering and looking for a seat which takes 3-5 minutes within the UC but also as any less times are not realistically compatible with the act of student's packing up as they leave which takes around a minute and is a difficult edge case for detection.

The second use case requirement within this area is that the hardware portion of the design will never go more than 30 minutes without sending some information. For power saving it makes sense to only send updates from the hardware components when there is a change in status. However, this means that a loss of signal could lead to incorrect information being displayed indefinitely. To mitigate tarnishing the user experience it is important to be able to detect when this has occurred. This specific metric is based on CMU's courses release schedule has classes ending every 30 minutes. A 30 minute window of maximum silence means indefinite loss of signal would never tarnish the user experience of more than one wave of students without being detected.

The third requirement is that the cloud deployment can support 80 hardware devices at once. This is because the UC has 80 tables, and for our use case, each table will have a hardware node detecting occupancy. The cloud deployment will have to support every device for our project to work.

2.3 Software Requirements

The software use case requirement is about the scale of support for this design. The software use-case requirement is that the web app has to be able to support 25 concurrent users. As an application designed around crowd mitigation, a lot of concurrent users is the expectation. We envision that our web application will take no more than 5 minutes to be used. From first-hand research, during peak hours, at most 25 people enter the UC within a 5 minute window; thus, we need to ensure 25 concurrent users can be supported.

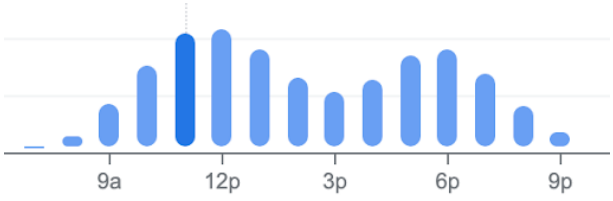

Figure 2: Google Maps' reported occupancy within the UC on a typical weekday.

3 ARCHITECTURE OVERVIEW

In this section, we will provide a high-level overview of our project. This can be seen in Figure 9. The system is broken down into three subsystems: the Hardware IOT Device, the AWS IOT Core, and the AWS EC2 Instance, which correspond with our project's hardware, hardware-software communication, and software subsystems respec-

tively. The Hardware IOT Device will communicate occupancy data to the AWS IOT Core through MQTT, which will then store the data in an Amazon RDS for the Amazon EC2 Instance to access. Finally, the Amazon EC2 Instance will service user HTTP requests and display all information in a user browser. A more specific overview for each subsystem will be described in further detail in the following subsections.

3.1 Hardware IOT Device

Figure 3 focuses specifically on the Hardware IOT Device System. The entire system is battery-powered, and this battery's voltage is broken up in a voltage converter module, consisting of a boost and a buck switching regulator, to properly supply the rest of the hardware system. Specifically, the PIR sensor and the thermal sensor both receive a 5V input, and the ESP8266 processing unit receives a 3.3V input. Both sensors are used to detect occupancy by monitoring the ground beneath a table for traces of movement or heat, and report data back to the processing unit as GPIOs. The processing unit will be written with code to wake it or put it to sleep, along with code to use sensor inputs to detect occupancy. The code will be optimized to save power, in accordance with our 55+ hour battery life use case requirement. Finally, the processing unit's Wi-Fi block will send any data it wants to communicate to the server to the AWS IOT Core.

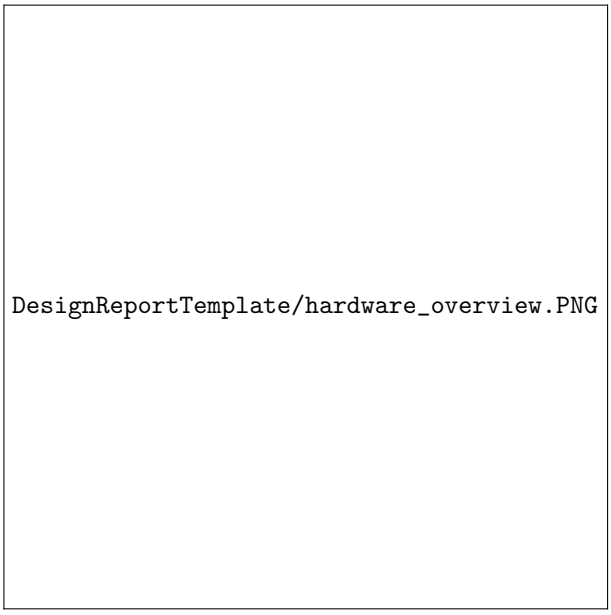

Figure 3: High level overview of the Hardware IOT Device

3.2 AWS IOT Core

Our hardware software interface consists of a cloud pipeline, broken into 3 components, all within the AWS product line. The first level of communication starting from the hardware device is through AWS IoT core which acts as our security point authenticating each device, verifying information and then based on whether data needs to be updated in the database will send an update. This data arrives at an AWS hosted relational database where it is stored for access by our Django server.

3.3 AWS EC2 Instance

Specifically, we will be deploying our Django app on AWS EC2 instance using Ubuntu as our AMI. The Django server queries the MySQL database for data on the occupancy status of tables and will display information and statistics on the frontend as needed by HTTP requests.

4 DESIGN REQUIREMENTS

In this section, we will discuss the qualitative specifications of each subsystem in the implementation of the final product. Specifically, we will discuss how each subsystem's quantitative values align with our use case requirements.

4.1 Hardware Design Requirements

The only use case requirement related to the hardware subsystem is that of the minimum battery life of 55+ active hours and 52+ deep sleep hours. Therefore, most of the quantitative specification work was done towards determining battery capacity and voltage. Specifically, we performed first-order power analysis using voltage and current data from the PIR sensor, thermal sensor, and processing unit datasheets [1] [5] [6], where energy consumed can be calculated as

$$E = I \cdot V \cdot t \tag{1}$$

where E is energy in Joules, I is current in Amperes, V is voltage in volts, and t is time in hours.

The results of this analysis can be seen in Figure 4. The processor's power consumption is broken down into three categories: transmitting, light sleep, and deep sleep. We expect the processor to spend around 0.5 seconds transmitting data throughout the week, assuming that the occupancy status of a table changes once in approximately every 30 minutes, since the processor takes less than 2 milliseconds to send data [6]. The processor is expected to spend around 52 hours in deep sleep outside of active hours, and around 55 hours in light sleep during active hours, when it does not actively need to transmit data. Both the thermal and PIR sensor are assumed to be active for 55 hours.

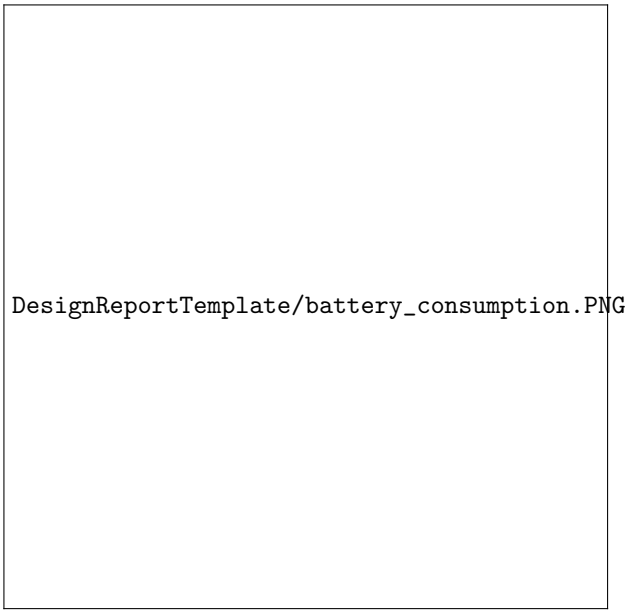

Figure 4: Table outlining the power consumption, in Joules, of each hardware component. The processor's power consumption is broken down into transmitting, light sleep, and deep sleep.

From this analysis, we determined that we'd need a battery to supply at least 19.42 Joules. After researching batteries that fit this requirement, we found that a 3.7V, 6 Amp-hour battery, which would supply over 22 Joules of energy, would make the most sense to use to meet our use case requirement.

We chose our PIR and thermal sensors based off of performance and capabilities, and both sensors require a 5V input. Additionally, our processor requires a standard 3.3V input. As a result, the other quantitative values we decided for the hardware subsystem were that our boost switching regulator would need to input 3.7V and output 5V, and our buck switching regulator would need to input 3.7V and output 3.3V.

4.2 Hardware-Software Integration Design Requirements

The hardware software interface consists of our cloud infrastructure into 3 components, all within the AWS product line. The first level of communication starting from the hardware device is through AWS IoT core. This provides a secure connection layer that allows easy connection, management, and scalability of the IoT fleet of embedded systems allowing 80 devices to be well within means of support meeting the use case requirements. The MQTT protocol is used to send messages from the devices over wifi that then have their keys validated by AWS IoT's message broker, all ensuring security within our application. Traffic through AWS IoT is also securely sent over TLS (Transport Layer Security).[3]

4.3 Software Design Requirements

The AWS IoT rules engine allows updates to be sent to the MySQL database, which we are hosting through Amazon RDS. By using Amazon RDS, we can host the database separately from the EC2 instance to free up computing power and ensure support of 25 concurrent users on the website. Amazon RDS also automates a lot of time-consuming and out-of-scope administrative tasks, such as "hardware provisioning, database setup, patching, and backups".[2] We chose to use MySQL because within our application, the necessity for scalability is minimized. Hence, we are able to take advantage of this excellent, free option that is quickly capable of being integrated with our Django server.

Based on prior experience and ease of use, a Django server allows the design to meet necessary use case requirements. Particularly the elastic nature of an EC2 deployment in combination with the "all under one roof" integration with the rest of our AWS system means secure and compact organization and accessibility to our data. Django also provides an ideal framework for both the front ends and back ends of our web application code, due to the non-compromising nature of Python.

As for frontend design, we made the decision to make our map a zoomable interface. This means that the user will be able to scroll and zoom to navigate the map. This allows the illusion of constantly accessible information, while in reality, the need for backend queries would be decreased. For example, when a user scrolls past a table such that it's no longer displayed on the screen, that data would not be needed and an extra backend call would not be necessary. On top of that, we also decided that when the user zooms out far enough on the map, individual tables would not be displayed anymore and that area would turn into a colored block paired with the overall occupancy statistic of the area, as shown in Fig. 8. By using this Zoomable UI, a zoomed out view of the UC would not seem too cluttered and would not require each table's occupancy status to be displayed, ultimately limiting the volume of backend calls.

5 SYSTEM IMPLEMENTATION

In this section, we will discuss the system implementation for each subsystem. This will include specific details on what hardware and software each subsystem will be using.

5.1 Hardware Implementation

Based on the specifications determined for our battery in Section 4.1 Hardware Design Requirements, we decided to use a SparkFun 3.7V, 6 Amp-Hour Lithium Ion battery. The main reason we chose this battery is because it is rechargeable. Since the hardware system will ideally be a permanent installment, we thought that it would cost much less and save a lot of battery waste to use a rechargeable battery, as opposed to replacing batteries each week as they

run out of charge. These batteries would have all weekend to charge back to full capacity.

We have tentatively chosen Texas Instruments boost and buck switching regulators to step up to 5V for the sensors, and step down to 3.3V for the processor. However, the both regulators are surface mount mounting types, as opposed to the dual inline package mounting type that would fit on a breadboard, so we acknowledge that they might not be permanent solutions. We have already planned to meet with our advisors to find a better approach.

Our PIR sensor was already in stock in the lab when we began the project, and we were recommended it by our advisors, so we decided to use it for factors of availability and convenience. Upon further research, it seems to be a DIYmall sensor. However, we have not extensively tested the sensor yet, and as noted in Figure 4, it consumes nearly all of the power of the hardware system, so we may replace this sensor as we see fit.

We chose an Omron MEMS thermal sensor to take thermal readings. The sensor has a 4x4 array of data points, providing 16 points or 4 bits of resolution for detecting occupancy. Although this resolution is certainly on the small side, the next thermal sensor in the series with a 32x32 array of data points is much more expensive at around $170 per sensor, and would not fit within our budget. A more in-depth discussion of our project's budget and Bill of Materials can be found in Section 8.2 Bill of Materials. We decided to incorporate a thermal sensor into our design as a secondary means of occupancy detection to help with noise from the PIR sensor, and we are hopeful that this sensor's resolution will be sufficient for that purpose.

Finally, our processor is an ESP8266 ESP-12E. An in-depth explanation of why we chose this processor can be found in Section 6 Design Trade Studies, as we initially began with using a different processor before switching to this one. The processor is the brain of our hardware system, and will execute code we will write to detect occupancy and send it to the web app. At a high level, the processor will sit in a deep sleep outside of the UC's active hours, with a timer to awaken itself the next day. During active hours, the processor will sit in a light sleep until it gets input from either sensor, where it will awaken to gather data and determine if the table is changing state (i.e. being occupied or being unoccupied). If the processor determines that the input is noise, it will go back into a light sleep. However, if it determines a state change, it will send an update to the web application on the table's new occupancy status before going back into a light sleep.

5.2 Hardware-Software Integration Subsystem

5.3 Software Subsystem

Our MySQL database will be periodically updated with each device's occupation status. To keep track of each device, the devices would have to have table location and table shape associated with it. While the table shape is not as important as the location, displaying the table accurately on the web application would require this data to be stored.

Our web application UI will be represented by a map interface, overlaid with a search bar and a bottom statistics tab. Upon opening the web application, the user will see a similar UI to that in Fig. 5.

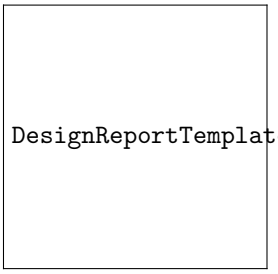

Figure 5: UI on initial load of the web application

The initial map displayed will be a zoomed in area of the UC 2nd floor in which each table will be colored magenta if it's occupied or green if it's open. The color palette is specifically designed such that people with red-green colorblindness can distinguish the table colors. The statistic displayed on the bottom would be that of the entire UC 2nd floor. This statistics tab can extend upward with a swipe to reveal a more detailed analysis of the selected area. The resulting UI would be that of Fig. 6.

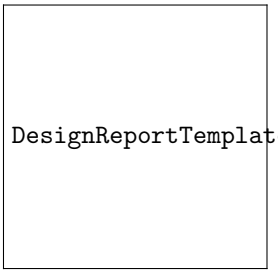

Figure 6: UI after upward swipe on the bottom tab

The map area and the displayed statistic can be changed if the user clicks on the search bar, which would reveal the UI in Fig. 7.

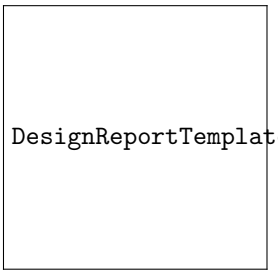

Figure 7: UI after user clicks on search bar

If the user taps a specific area, the screen will switch back to the UI in Fig. 5 but with the new area that the user

selected. The map interface is designed to be a Zoomable UI, meaning that the user will be able to swipe continuously around the map as well as zoom in and out. Once the user zooms out far enough, the UI will change to look like Fig. 8, where each area would just be a block of color based on the overarching occupation percentage of the area.

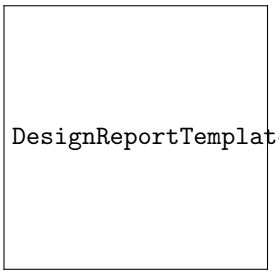

Figure 8: UI after map is zoomed out

The color of each block would be a range from the magenta (100% occupation) to the bright green (0% occupation). This UI would only happen after the user has zoomed out to the point where each table covers 1% or less of the map interface.

6 DESIGN TRADE STUDIES

While iterating through the design process, we considered three implementation ideas which we ultimately chose not to pursue. The first implementation idea involved using an Arduino Uno as our processor with an ESP8266 ESP-01 added on for Wi-Fi communication. We primarily wanted to use an Arduino because we are all familiar with the Arduino IDE. However, we realized that we would likely need another module to support the Wi-Fi communication which we would have to integrate with the Arduino. We ended up settling on only using an ESP8266 ESP-12E, as it not only supports the required Wi-Fi communication but also has a built-in processor with GPIO pins which would be effective in replacing the Arduino. As a bonus, the ESP8266 ESP-12E supports the Arduino IDE, so this was an obvious choice, as it would save us extra effort to integrate two devices otherwise.

The second implementation idea involved alternate sources of powering our hardware. Specifically, we received an early suggestion to use solar panels to charge our battery. This would eliminate the necessity of a high-capacity battery, as the battery would be able to charge throughout the day as opposed to needing to last on one charge all week. Although this was an intriguing option, we chose not to pursue it as we found that it did not fit our use case. Specifically, we wanted the hardware on each table to be minimally intrusive on the user experience, and we were already planning on all other hardware being mounted on the underside of the table. Since a solar panel would only be able to rest on top of the table, we decided not to include it in our design.

One final implementation idea involved including a button in our hardware to account for an edge case within our use case. Specifically, if a user sits down at a table but gets up to use the bathroom or do something else while leaving their possessions at a table, our current solution would indicate that table as open, while in reality it would be occupied. We quickly ruled out the possibility of detecting a user's possessions, as that would require a camera which would have to be deployed above the table and would raise privacy issues. We also considered allowing the user to request to hold the table on the web application, but quickly realized that could get abused by users holding tables while leaving the UC to do other things. Finally, we considered incorporating a button into the hardware which would hold the table for a short duration when pressed. However, similar to the solar panel, this does not fit within our use case, as the button would have to sit on top of the table, and wiring it from the top of the table to the rest of the hardware on the table's underside would be haphazard. After much deliberation, we decided that our use case would not consider this edge case at all. An average uses the bathroom 6-7 times in a 24 hour period [4]. Assuming that 3 of those trips happen during the 9am to 8pm active time period of the application and each bathroom trip lasts an average of 10 minutes, then the maximum of 30 minutes of bathroom time per table out of the 720 minutes of activity per day would result in a 4% chance of a user encountering a free table that actually is occupied. We decided this percentage was insignificant enough for us to ignore the case entirely.

7 TEST & VALIDATION

To test to make sure that our project is functional according to our 5 use case requirements, we have come up with 6 tests. These tests are broken into subsystems as the use case requirements are in Section 2 Use Case Requirements, with the hardware subsystem having two tests for one use case requirement. Additionally, we have come up with risk mitigation plans for each use case requirement, should our system fail testing.

7.1 Hardware Tests

There are two ways we plan on testing our use case of a 55+ active hour and a 52+ inactive hour battery life. The first test simply involves running the hardware system in a real-life environment from Monday to Friday. We are currently planning on setting up a system on a Hamerschlag Hall 1300 wing table, as we believe that it will be easier to implement the project within an ECE-controlled space as opposed to the UC, and we believe that the environment should be similar enough to emulate that of the UC. We will turn on the system at 9am on Monday, and at 8pm on Friday, we will measure the battery. If there is still any charge left in the battery, the test will pass, with alignment to our use case.

The second test for this use case involves verifying that components within the hardware system are drawing the expected amount of power, as compared to their datasheets. Specifically, we will set one hardware system up in a Hamerschlag Hall 1300 Lab with a multimeter and read the voltage differences and current draws from the PIR sensor, the thermal sensor, and the processor. For the processor specifically, we will take three sets of readings: one when the processor is in deep sleep, one when the processor is in light sleep, and one when the processor is detecting occupancy and sending the data to the web server. We will take five readings for each set, and use the average current and voltage of the set of readings to measure power, which is calculated as

$$P = I \cdot V \tag{2}$$

where P is power in watts, I is current in Amperes, and V is voltage in volts. We will compare these measurements to each device's theoretical maximum power consumption, using maximum current and voltage listed in each device's datasheet. If the calculated power is less than the theoretical power for all three devices, the test will pass, which supports that the energy consumption calculations in Section 4.1 Hardware Design Requirements used to support our battery choice are accurate.

If these tests fail, we have a risk mitigation plan to ensure we still pass our battery life use case. As noted in Section 4.1 Hardware Design Requirements, the PIR sensor consumes almost 18 Joules of energy, which is nearly all of the energy consumed by the hardware system, due to its high current draw. We plan to reduce the amount of energy this sensor consumes by decreasing the time it is powered on. Since we expect there to be long stretches of time where occupancy status doesn't change, we will only have the thermal sensor on detecting any possible changes in occupancy during that time. If the thermal sensor detects a possible change, only then will we turn on the PIR sensor to confirm a change in occupancy status. Doing this should greatly reduce the operating time and energy drawn by the PIR sensor without sacrificing occupancy detection accuracy.

7.2 Hardware-Software Communication Tests

7.3 Software Tests

For testing concurrent users and simulating web traffic for stress testing, we will be using Loadview. Although the price is $199/month, there is a free trial we can use for the scope of this project. Loadview allows the user to define the amount of concurrent users, test duration, real user test scenarios, ramp up and ramp down periods, etc.

While testing with Loadview, we should monitor EC2 CPU credit balance loss. If the credit balance loss exceeds the assigned credit per hour, then we will know that this website cannot host this many concurrent users. We will be able to test for 25 concurrent users and make changes accordingly for each test.

8 PROJECT MANAGEMENT

8.1 Schedule and Team Member Responsibilities

A full breakdown of the project schedule and the work distribution can be found in Fig. 10. Notably, Ryan is largely in charge of the hardware bring-up and sensor detection, Jake is largely in charge of setting up AWS and facilitating from the hardware to the web app, and Angela is largely in charge of the front-end and back-end of the web application. Specifically, as it relates to this report, Ryan has been in charge of everything in the hardware subsystem, Jake has been in charge of everything in the hardware-software communication subsystem, and Angela has been in charge of everything in the software subsystem.

8.2 Bill of Materials

Table 1 lists the Bill of Materials for our project. As can be seen, most of the line items within the Bill of Materials are for hardware to support the hardware subsystem, with one line item for AWS credits to facilitate our web application. The total cost for the materials in our project is $379.32. Importantly, this Bill of Materials only supports the 4 hardware nodes which we will use for testing purposes, as opposed to the approximately 80 hardware nodes that would be necessary to track every table in the UC. As our budget for this project is only $600, it is outside of our scope to purchase and instantiate all 80 hardware nodes for testing. We decided to only purchase 4 hardware nodes to leave some room in our budget in case we need to add to our design or buy additional support equipment for testing.

Glossary of Acronyms

Include an alphabetized list of acronyms if you have lots of these included in your document. Otherwise define the acronyms inline.

- AMI - Amazon Machine Image
- AWS - Amazon Web Services
- EC2 - Amazon Elastic Compute Cloud
- ECE - Electrical and Computer Engineering
- GPIO - General Purpose Input/Output
- IOT - Internet of Things
- MEMS - Micro Electronic Mechanical System
- MQTT – Message Queuing Telemetry Transport
- OBD – On-Board Diagnostics
- PIR - Pyroelectric Infrared Sensor
- RDS - Relational Database Service

Table 1: Bill of Materials

Description	Model #	Manufacturer	Source	Quantity	Cost @	Total
Processor with Built-In Wi-Fi	ESP8266 ESP-12E	KeeYees	Amazon	4	$15.98	$63.92
3.7V 6Ah Lithium Ion Battery	PRT-13856	SparkFun	SparkFun	4	$32.50	$130.00
5V Boost Switching Regulator	TPS613222ADBVR	Texas Instruments	Digikey	4	$0.61	$2.44
3.3V Buck Switching Regulator	TPS62046DRCR	Texas Instruments	Digikey	4	$2.39	$9.56
PIR Sensor	HC-SR501	DIYmall	ECE Lab	4	$0.00	$0.00
Thermal Sensor	D6T44L06	OMRON	Digikey	4	$43.35	$173.40
AWS Credits	N/A	AWS	AWS	1	$50.00	$50.00
Breadboard	N/A	N/A	ECE Lab	4	$0.00	$0.00
Jumper Cables	N/A	N/A	ECE Lab	4	$0.00	$0.00
						$379.32

- RPi – Raspberry Pi
- UC - (Cohon) University Center (Second Floor)
- UI - User Interface
- V - Voltage

References

[1] lady ada. *PIR Motion Sensor*. 2021. URL: https://learn.adafruit.com/pir-passive-infrared-proximity-motion-sensor.

[2] *Amazon Relational Database Service (RDS)*. Amazon Web Services. URL: https://aws.amazon.com/rds/#:~:text=It%20provides%20cost%2Defficient%20and,security%2C%20and%20compatibility%20they%20need..

[3] *AWS IoT Security*. Amazon Web Services. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-security.html.

[4] Chaunie Brusie. *Does How Often You Pee Say Something About Your Health?* Healthline. 2018. URL: https://www.healthline.com/health/how-often-should-you-pee.

[5] *D6T MEMS Thermal Sensors*. OMRON.

[6] Espressif Systems IOT Team, ed. *ESP8266EX Datasheet*. Version 4.3. Espressif Systems, 2015.

DesignReportTemplate/architecture_overview.PNG

Figure 9: A high-level overview of the project's system architecture.

DesignReportTemplate/ganttchart.png

Figure 10: Gantt Chart