Green Grid Advisor

Calvin Li, Nicholas Prakoso, Anuj Zore, Nestor Tiglao
Department of Electrical and Computer Engineering, University of Maryland, College Park, Maryland 20742, USA
cli48@terpmail.umd.edu, nprakoso@terpmail.umd.edu, zoreanuj@umd.edu, ntiglao@umd.edu

I. Table of Contents

II. Signature (approval) of each team member

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

In summary, I became the front-end developer for this capstone project with my main responsibility consisting of creating the app for Android devices through Android Studio using the Kotlin programming language. In addition to the app development, I conducted market research alongside my fellow team member and explored possible business opportunities of our product through the Equity Incubator at USG.

Approved by
Nicholas Prakoso

[illegible]

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

I am responsible for the backend of the capstone project, and my duties include developing a REST server using 'PythonAnywhere' with the Flask library to route HTTP requests from the client frontend. The REST server will handle tasks such as login, registration, database tool editing, and savings algorithms. Additionally, I will be responsible for the development of the IRMS sensor, a non-invasive current sensing device that will transmit data back to the REST server for recording in a CSV format. Later on, I will also perform data analysis on the IRMS data, as it is crucial to proving the validity of the capstone project for future development. I will also support my fellow team members in the equity incubator at USG.

Approved by
Calvin Li

Calvin Li

III. Executive Summary

Our project addresses the escalating energy consumption crisis that is affecting the United States by providing users with a user-friendly Android app that is designed to optimize energy usage through participation in Time-of-Use (ToU) programs. The primary objective of this project is to empower users to efficiently manage their energy consumption and reduce their electricity bills by strategically utilizing ToU rates better.

We achieve our goals through the development of an Android application that features scheduling. This scheduling functionality enables users to plan their day to day tasks to occur at off-peak hours to more efficiently utilize electricity and avoid the higher costs of variable electricity rates.

In further support of achieving our goals, we established a centralized database for ToU rates to enhance the app's accuracy and reliability. The database not only acts as a backend for our application, but also acts as a foundation for potential future expansion of the database should we decide to reach out to more companies and consumers.

Python, specifically Flask and PythonAnywhere, emerged as our preferred programming languages due to their sim-

plicity and versatility. They facilitated the development of a robust backend, ensuring efficient data manipulation and graph creation. The integration of wattage data collection further showcased the flexibility of our backend, allowing for future software and hardware enhancements. However, realistic constraints, such as variable appliance wattage and the absence of real-time TOU rates, posed challenges. To address inaccuracies, we proposed a wattage sensor to calculate real averages, and we acknowledged the need for real-time TOU data to prevent discrepancies.

Our simulation results indicate the potential for significant cost savings through efficient energy consumption. The prototype fabrication involves the Android app and the centralized ToU rate database, both of which take significant time.

To validate our goals, we conducted thorough testing using appliances at home and compared them in conjunction with ToU rates against fixed rates. These values would then be used to create a visual graph which is then sent to the app to display to the user.

IV. Main Body

A. Introduction

With the continued evolution of technology, energy usage is recorded to be the highest it has ever been in history. This results in higher amounts of supply and demand and alongside that, huge amounts of wastage of energy doing damage to the environment and costing many families more on their electricity bills. As a solution, we propose an app that would predict/forecast/read the most efficient times to use energy. The app will work reactively to the user actions with notifications. The final outcome of the project will include a server that stores all data on the backend that we would be able to manipulate and an Android app that we develop on the frontend that displays all analytics to the user.

This project helps users better utilize time of use rates to effectively achieve our goal of reducing electricity bills and energy wastage. In short, time of use rates outline a plan where a homeowner is charged a variable electricity rate based on demand throughout the day. It is important to note that throughout the day, there are times that are considered peak and off-peak. Peak meaning the time of day where the variable electricity rate is considered the highest and off-peak being the opposite.

The project may be used in applications where users wish to test the effectiveness of time of use rates in comparison to fixed rate electricity plans. In addition, this project may be used in government programs where time of use rate plans are more often utilized.

End-users primarily include residential homeowners, but with enough research and expansion, we can extend that reach to business owners as well.

Design inspiration did not come from any specific project or app, however we did do a lot of market research on smart sockets and smart plugs in particular which served as inspiration in developing our project.

B. Goals and Design Overview

The primary goal of our project was to create an app that would make it easier for users to participate in Time-of-Use (TOU) programs. To achieve this goal, we decided to implement a scheduling feature within the app.

The implementation of this scheduling feature was crucial because it would allow users to better manage their energy consumption during different periods of the day. However, to make this feature work effectively, we needed to establish a prototype centralized database containing up-to-date TOU rates and intervals. This aspect of our project serves a secondary purpose to the community by creating a centralized database other apps or products can expand off.

In summary, our project's main aim was to simplify user participation in TOU programs through the app, with the scheduling feature being a key component. Simultaneously, we aimed to provide users with access to accurate and timely TOU rate information by maintaining a custom centralized database.

Once we had a broad understanding of the core functionalities our app needed to encompass, the design process hinged largely on the choice of programming languages that could effectively break down our objectives into manageable components. We decided to utilize Python due to its simplicity and versatility in performing a range of actions with minimal code.

Python Flask and PythonAnywhere proved to be the ideal choices because of their ability to handle various tasks efficiently. For example, they allowed us to easily create graphs and manipulate files with concise code. Flask's versatility not only streamlined our development process but also ensured that we could implement the necessary features and functionalities in a straightforward and efficient manner.

When integrating wattage data collection, our group realized that our backend has become highly flexible thanks to Flask, and our app can now function more effectively as a platform for additional software features and hardware implementations.

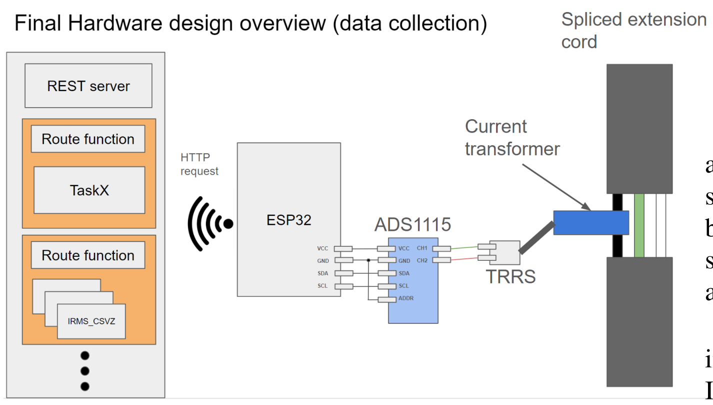


Fig. 1: The figure above is a depiction of the final hardware design overview

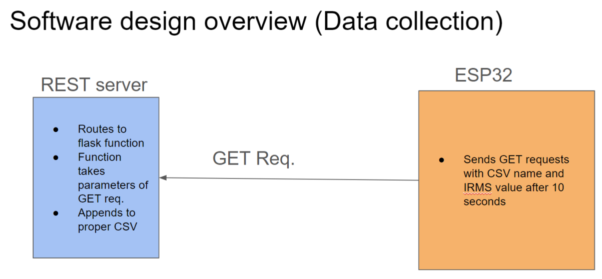

Software design overview (Data collection)
REST server
Routes to flask function
Function takes parameters of GET req.
Appends to proper CSV
GET Req.
ESP32
Sends GET requests with CSV name and IRMS value after 10 seconds

Fig. 2: The figure above is a short overview of our software design for data collection

C. Realistic Constraints

It is known savings calculations are to be skewed since home appliance wattage usage varies from appliance to appliance. Our group realized that to alleviate inaccurate cost savings calculations, average wattage per hour needed to be found by the wattage sensor that would tell real wattage use averages per appliance. Rather than use some average wattage for some appliance.

Not having real time ToU rates can result in bad cost savings calculations, companies could potentially change out the peak and low peak rates without knowledge to us the developers which can result in inaccurate savings calculations.

There is no centralized database for the ToU data. The realistic constraint behind this is that there are several power authorities in a given state in addition to a multitude of ToU programs, so realistically there is far too much realtime data for us to hand track at the moment, but given time that could change. In addition, it is apparent that not every single power authority or company provides real time ToU data possibly to keep that information out of competitors' hands which makes it a bit harder for us to keep the server constantly up to date.

D. Engineering Standards

The major backbone that connects all the features behind the app is the REST API standard for web based applications. This standard was used in tandem with pythonanywhere + flask to be primarily used for data transfer between app, hardware and server. This standard allows for heavy versatility between front and backend design capabilities.

I2C is a data transfer protocol intended for addressing,routing and distributing data amongst several components. I2C is used in our project between the ESP32 and ADS1115 component to interface values read in by analogue to digital converter. This setup is fairly standard for most sensors to be interfaced with by a microcontroller of some type.

HTTP requests are a staple to the functionality of a REST API set up. HTTP requests send payload/parameter prompts to a webserver and the server returns with some response. This is an essential standard for the setup of our backend to front end data transfer system.

Adafruit isn't necessarily providing a standard but the community provides a significant amount of open source code that serves as template code that many developers use as their base or "standard" code for sensor usage. This is important for getting projects to work in general and code legibility.

TRRS stands for tip, ring, ring, and sleeve. This is typically an audio standard for transferring multiple digital signals (music), but the current transformer sensor had a TRRS standard 3.5 mm jack output. Using this standard, we realized we had to perform a signal differentiation between the tip and sleeve to determine data output with an ADC of programmable gain.

E. Alternative Designs and Design Choices

Designed specifically for the app, it will send HTTP requests and parse incoming return strings from the website. This is how the app communicates with the REST server, enabling it to invoke specific functions on the backend. User profile information is also stored locally on the device. Simultaneously, all time-of-use information is stored in JSON format on the server to minimize data storage requirements on mobile device as the database expands.

The REST server stores all user information in CSV format and appends new entries when new registrations occur. Functions like 'login' are implemented to validate user login, and a security token generator is also integrated. Additionally, a function to modify user profile data in the CSV is established, which is crucial for enhancing the user experience with the app.

Furthermore, the REST server can handle requests from the ESP32 device. It includes a function to write to a CSV, with parameters from the ESP32 to the REST server specifying the CSV name and the IRMS value at the time. If an existing CSV does not match the CSV name passed from the ESP32, a new CSV with that name is created. This ensures tracking of device statistics when programming the ESP32 device.

Initially, alternate designs depended on live Time-of-Use (TOU) updates, but it was soon realized that not all companies have this feature or maintain a central database with state and power authority information. Due to the lack of data infrastructure, we abandoned the dependency on live data and opted for a design reliant on static values. These values are researched and input into the JSON file by our team for later extraction and use in the app.

Given the versatility of the REST server, hardware access is equally straightforward. We initially considered incorporating a device shutdown feature in our app, allowing users to non-invasively cut off amperage consumption by devices via app control. However, due to time constraints, we had to abandon this idea as it would have required additional research into the hardware required for its implementation.

Another idea we had was to use the IRMS data collection device to provide real-time displays of wattage usage, enhancing the functionality of our existing hardware. Unfortunately, we were unable to implement this idea within the project's time constraints. Nevertheless, it remains a crucial aspect of the app, as it would enable users to gain a better understanding of their power consumption, further motivating users to consider switching to a time-of-use program and utilizing the Green Grid Advisor app.

F. Technical Analysis for System and Subsystems

Ideally, PythonAnywhere should already be scaled to handle thousands of requests in a given day, as HTTP requests do not require any specific space or processing allocation. Flask, acting as overhead, should efficiently route these HTTP requests to streamline function calls. The responses from the REST server are generic and depend on parameter passing from the user.

The backend of our system will utilize Python, as it is the most versatile and straightforward language capable of performing a wide range of functions. Specifically, we will employ a library called 'Flask' to set up a REST server. Flask enables parameter passing and routing of functions based on HTTP requests. Fortunately, during our research, we discovered a website called 'PythonAnywhere' that offers the capability to host online Flask environments. This platform provides various libraries to support diverse project types, making 'PythonAnywhere' the most suitable choice for our backend development moving forward.

The Android app will manage the sending and parsing of requests, and it will return strings to the web service gateway REST server. This facilitates basic parameter passing and reading between the app and the server when a request is processed. We recognized the necessity of creating basic profiles for individual users. The server's responses must be generic, as there will be no allocated storage or performance resources for individual users; the primary function will be handling requests in a queue.

In data handling, Kotlin sends GET and POST HTTP requests to the server to obtain information already stored for a particular registered user. Once the request is successfully processed, parsing occurs to be able to separate the data into small bits to be used independently to update UI elements to display to the user. Should the user not be registered already, they can register through the register screen which allows for successful execution of the POST request to the server to store the data for a longer time. It is especially important for the application to receive data from the backend because Kotlin can handle data retrieval and processing, but to prevent further resources being consumed by the application continuously and for more secure data storage, all user data and information is stored on the backend server.

The inclusion of user profiles necessitates the implementation of features such as user login, registration, and the ability to modify information on the server from within the app. This is crucial when users need to make changes to their power authority or state settings. Additionally, the concept of individual savings played a significant role in the decision to develop user profiles. These profiles enable us to keep track of how much money each user has saved. Furthermore, we generate and issue security tokens to eliminate the need for constantly transmitting user credentials (username and password) for enhanced security.

As previously mentioned, since multiple profiles will access the server, we will primarily handle generic input and output responses with parameters being passed. The server will maintain time-of-use data in a JSON file and respond based on the power authority and state parameters provided. Additionally, the server will store and modify user data in a CSV file, responding to function requests and associated parameters. Given the project's current scope, using a CSV was a straightforward method for data storage and organization.

To read the IRMS from a 15A rated wire, it was necessary to use a current transformer for non-invasively measuring the current flowing from the outlet to the device. The current transformer needed an ADC (Analog-to-Digital Converter) with programmable gain and the capability to differentiate a signal to determine amperage readings. After considering the options, we opted for the ADS1115, as it met the required parameters, had readily available base code, and was compatible with I2C communication for the ESP32.

Data collection was achieved by having our backend receive and append data to a large CSV file every 10 seconds. The ESP32 device utilized Arduino libraries to make HTTP requests, and we determined that sending HTTP requests every 10 seconds was more efficient than setting up an MQTT server to perform the same task. This approach allowed us to consolidate our data and functions in one central location.

G. Design Validation for System and Subsystems

The REST server has demonstrated its reliability, having successfully handled 350k requests, primarily from data collection sensor devices. 'PythonAnywhere' hosts the server 24/7, and we haven't encountered any connectivity issues as long as the HTTP link is correct.

There was a minor issue observed in the data collection process between the server and the ESP32 device, which sends HTTP requests. The data collection device becomes unresponsive after 6-8 days of continuous data transmission every 10 seconds, requiring a manual restart of the device. This results in a data blackout during the period when the device was unresponsive. To address this, we need to integrate a developer warning system that signals the need for a reset or indicates a data blackout. Other than this intermittent issue, data is received and appended as expected.

Backend functions on the REST server related to profiling tasks are functioning as intended, including successful registration and checks for redundant information to ensure only unique users exist. It correctly stores values into the user profile CSV, establishing a database for users and their settings.

However, there are challenges with the accuracy of savings calculations, primarily stemming from the lack of wattage data for individual devices. We now recognize that for our app to provide accurate results to users, we need to calculate an average wattage per hour based on data collected from a device. Therefore, in future development, there is a significant reliance on and necessity for users to also use the IRMS device to assist in calculating precise savings.

As development of the Android application came along, the end product did not come without challenges. In obtaining more information on handling data through HTTP GET and POST requests, there were problems in utilizing the initial API to help complete the task which was Ktor. Ktor initially seemed like the best choice in handling HTTP requests, but being much newer to Kotlin led to becoming overwhelmed by the documentation which in turn resulted in failure to use the API. To solve this problem, further research was conducted and the solution found was OkHttp. A much simpler and user friendly API that handled HTTP requests.

Another problem that occurred came through during development of the profile page. The profile page should display all user information they originally input upon registration and include the amount of savings they have made through use of the app. Upon first try, the profile page would not work correctly as the fields would show null when showing data that was obtained through the login page. Upon login, the app sends a GET request to the server to obtain all data related to the user password combination which occurred, but the data would not transfer between the pages correctly. To fix this issue, an additional GET request was made on the actual profile page to get the information again. Once the request was processed, parsing occurred and the UI elements were updated successfully.

Due to time constraints, there was only so far we could get in app development, but there were quite a few features we thought of at the start that we thought about implementing should we have the time. One of those features was local data storage of profiles or login information. This was a feature that we found to be useful, but not something to be prioritized since there was already quite a bit to do. This feature would have been very useful should the user close the app and want to log back in quickly. In implementing this feature, we thought about utilizing SQLite, but unfortunately we were unable to get to that point.

H. Test Plan

The test plan for the REST server is to ensure the ability to run all functions on the Flask list and ensure responsiveness. It also involves running all combinations of power authorities and states to ensure proper data extraction for all scenarios. User profile redundancies are built into the registration function; a simple request to register with an existing username is a proper test.

Security token checking is crucial for functions that change user information, ensuring that changes to a category occur only if a security token is passed.

A simple testing environment would be the Kotlin app itself since it relies heavily on backend processing. The deliverables would include the Kotlin app and REST server, alongside a JSON file containing several time-of-use rates based on state and power authority.

For testing the IRMS sensors, ensure a CSV is generated when a name isn't present inside the CSV folder. Check to see if values are being recorded every 10 seconds for 30 minutes, which should suffice as evidence for sensor functionality. A deliverable is an example CSV for a device used for 20+ days.

I. Procedure

The CSV data was collected over the course of 25+ days for four power intensive appliances found in a typical residential home. The energy intensive appliances in question are a desktop,washing machine, dryer and microwave (specs for devices listed in appendix). Using the non-invasive nature of the IRMS data collection device created by Green Grid advisor, simply using the male to outlet and using the female to appliances, a readable point is now established. All that is needed now (assuming that the hardware device is already assembled properly) is to assign a CSV name to the ESP32 and begin collecting data. System will return IRMS value every 10 seconds, system will run as long as desired.
Precaution: The ESP32 becomes unresponsive after 6-8 days of continual usage requiring manual reset.

J. Data Analysis

The CSVs contain 1-2 data blanks resulting from unexpected hardware malfunctions. In total, over 360,000 data points were recorded over a 25-day period (excluding the data blanks). Time, date, and IRMS values were recorded every 10 seconds. To obtain hourly averages, the recorded points were averaged for each hour, calculated as the sum of 360 points divided by 360 for the recorded hour. This calculation provides the average wattage usage for that specific hour.

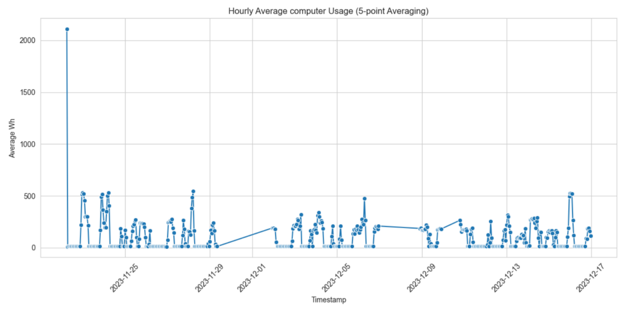


Fig. 3: Above is a visual for 25 days of wattage usage for a computer. Isolating parts of the graph with python is just as easy as well, the data accurately reflect real time behavior and can tell when appliance usage is most prevalent

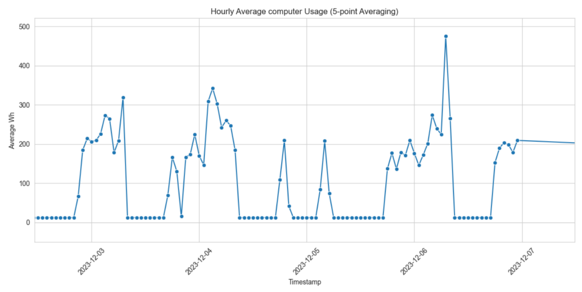
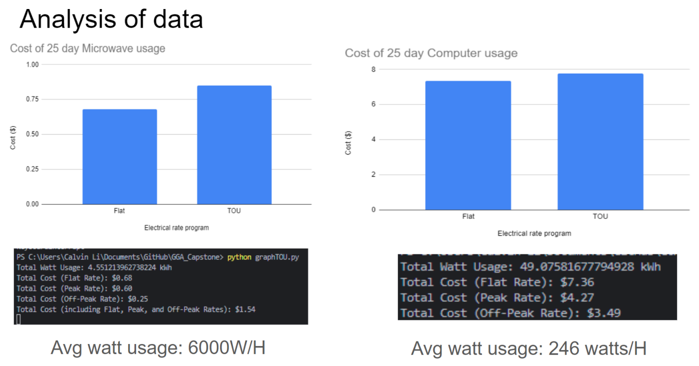


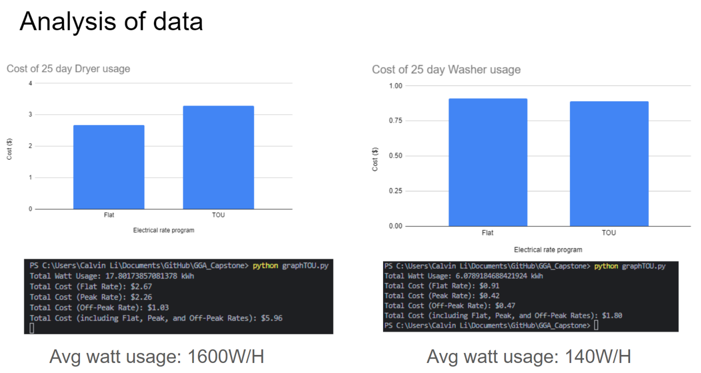
Fig. 4: Above is a zoom in on the dates Dec 3 to Dec7, this is to demonstrate capability of plotting data and realizing hours of computer usage.



```
PS C:\Users\Calvin Li\Documents\GitHub\GGA_Capstone> python graphTOU.py
Total Watt Usage: 4.551213962738224 kWh
Total Cost (Flat Rate): $0.68
Total Cost (Peak Rate): $0.60
Total Cost (Off-Peak Rate): $0.25
Total Cost (including Flat, Peak, and Off-Peak Rates): $1.54
```

```
Total Watt Usage: 49.07581677794928 kWh
Total Cost (Flat Rate): $7.36
Total Cost (Peak Rate): $4.27
Total Cost (Off-Peak Rate): $3.49
```

Fig. 5: Above is a price comparison between flat and TOU for a computer and a microwave.



```
PS C:\Users\Calvin Li\Documents\GitHub\GGA_Capstone> python graphTOU.py
Total Watt Usage: 17.80173857081378 kWh
Total Cost (Flat Rate): $2.67
Total Cost (Peak Rate): $2.26
Total Cost (Off-Peak Rate): $1.03
Total Cost (including Flat, Peak, and Off-Peak Rates): $5.96
```

```
PS C:\Users\Calvin Li\Documents\GitHub\GGA_Capstone> python graphTOU.py
Total Watt Usage: 6.078918468842l924 kWh
Total Cost (Flat Rate): $0.91
Total Cost (Peak Rate): $0.42
Total Cost (Off-Peak Rate): $0.47
Total Cost (including Flat, Peak, and Off-Peak Rates): $1.80
PS C:\Users\Calvin Li\Documents\GitHub\GGA_Capstone>
```

Fig. 6: Above is a price comparison between flat and TOU for a computer and a microwave.

Due to the small size of the data cluster, it's difficult to accurately narrow intervals of usage. Visual inspection reveals a dryer and washer correlation with a constant offset of 1-3 hours, as expected from nominal usage. Computer usage appears to be more random, but typical usage totals 8 hours on average, occurring between 11 am and 10 pm. Microwave usage tends to be more frequent later into the night, typically at 1-2 am.

Performing a mass calculation with python,finding TOU was somewhat challenging as an algorithm had to be developed to find when intervals were not peak rates and when they are not to determine which peak rate to multiply by.

It appears that the Time-of-Use (TOU) rates were more expensive, based on unbiased participation. The average wattage per hour was calculated for the purpose of determining cost savings. Surprisingly, the only appliance that demonstrated cost savings under theoretical TOU rates is the washer. This is not surprising, as the data indicates increased usage during non-peak hours, typically towards the end of the night. However, the savings appear to be minimal and may discourage users if the savings are only marginally small.

The computer incurred the highest cost compared to other devices, mainly due to the user spending an average of 8-10 hours on it. Surprisingly, despite its off-peak usage, the microwave, being a high-power device for short intervals, had a relatively high cost. The dryer, on the other hand, was frequently used during peak hours for 1-2 hours at a time, resulting in slightly higher energy expenditure.

The usefulness of this app relies heavily on the next set of tests, where Green Grid Advisors will need to recruit volunteers participating in TOU programs to record and calculate savings over a longer period than 25 days. This will help determine how much more users can save compared to flat rates when actively participating in TOU programs.

K. Project Planning and Management

After establishing a basic design, a project backlog was created following Scrum guidelines for success. Subsequently, a Gantt chart was developed to track progress week by week. Nick assumed the role of the product owner and was responsible for defining the product vision and managing the project backlog. Calvin took on the role of the Scrum Master, ensuring the progression of the project backlog and the completion of tasks assigned by the product owner.

Nicholas was tasked with researching Kotlin topics, integrating backend HTTP requests, managing string parsing, display graphics using variables, and login/page visuals. He was responsible for troubleshooting and implementing smaller features, such as drop-down menus of available options and controlling user experience through control flow. Nicholas played a significant role in communicating how variables are passed from the backend to the frontend to streamline client usage. These tasks are crucial for the usability of backend algorithms and the overall user experience.

Calvin was responsible for researching, developing, testing, and conducting data analysis on CSV data produced by IRMS sensors. He created visual plots for data analysis and calculated the validity of using time-of-use (TOU) and fixed rates at recorded times. Calvin also had the responsibility of developing REST servers, which included creating backend algorithms for login, registration, user profile data storage, IRMS data storage, and savings calculations. All of these components are vital for the functionality of the frontend, with the goal of reducing client processing and maintaining the security of sensitive server-side information, such as user details.

V. Conclusions

Based on the analysis of IRMS data, it appears that the savings from non-participation are exceedingly marginal. However, it's essential to note that these savings can vary significantly from person to person due to unique schedules. To determine the validity of developing this app, further studies must be conducted on both active and inactive participants in TOU programs.

The results from the IRMS also suggest that switching to TOU rates, as opposed to a fixed rate, will result in only slight differences in the total bill based on the sampled schedule. This means it serves as an incentive for first-time TOU users to consider this option, as it wouldn't result in any more of a financial burden than a fixed rate.

Backend development with a REST API set up is incredibly straightforward, as PythonAnywhere allows for fast implementations of parameter passing to functions. Individual profiling is complete, along with data collection sensor apparatus for future studies. The next step in the development of our app is to include a methodology that allows users to employ the IRMS sensors to track the energy usage of appliances.

Front-end development in Kotlin was a big learning experience for the team as it was a language that was learned as we went along with development. However, this resulted in great practice in app development alongside a product that may not necessarily have been the best one we looked for. Regardless, Android app development is something that can be picked up relatively quickly with prior coding knowledge, but with the time constraints set, intermediate level resources could not be integrated into app development such as SQLite.

VI. References

References

[1] A. Choudhari, P. Dhongde, S. Ranglani, A. Lekurwale, and J. Gawai, *A mobile app for Smart Electricity Usage Monitoring.* In *IEEE conference.* Available: https://ieeexplore.ieee.org/document/9742787

[2] J. Engländer, V. Stich, V. Zeller, T. Schmid, Y. Kledzinski, and L. Wenger, *A methodology to realize energy-related use cases for a...* In *IEEE Xplore.* Available: https://ieeexplore.ieee.org/document/9091408/

[3] C. Li, W. L. Woo, and T. Logenthiran, *Development of mobile application for Smart Home Energy...* In *IEEE Xplore.* Available: https://ieeexplore.ieee.org/document/7584199/

[4] P. M. van de Ven, N. Hegde, L. Massoulié, and T. Salonidis, *Optimal Control of End-User Energy Storage.* In *IEEE Transactions on Smart Grid*, vol. 4, no. 2, pp. 789-797, June 2013. doi: 10.1109/TSG.2012.2232943

[5] S. Shao, T. Zhang, M. Pipattanasomporn, and S. Rahman, *Impact of TOU rates on distribution load shapes in a smart grid with PHEV penetration.* In *IEEE PES T&D 2010*, New Orleans, LA, USA, 2010, pp. 1-6. doi: 10.1109/TDC.2010.5484336

[6] J. Zhang, X. Han, J. Qiu, and Y. Tao, *Design of a smart socket for Smart Home Energy Management...* In *IEEE Xplore.* Available: https://ieeexplore.ieee.org/document/8832486/

VII. Appendix

Bill of Materials

The detailed Bill of Materials (BOM) is provided in a separate file named `BOM.xlsx`. Please refer to this document for a comprehensive list of components and their specifications.

Block Diagram/Technical Drawings/Modeling Details

Please refer to the following Figma link: https://www.figma.com/file/iYDPTm5yb6NcgsVlSZEknn/GGA?type=whiteboard&node-id=0-1&t=1mnpWke5eGIKawXY-0

Gantt Chart

Please refer to the separate file named `Gantt.xlsx` for the Gantt Chart

Software Codes

Please refer to the separate folder named `Software Code` for a complete list of code created

Code Documentation

Please refer to a separate file named `GGA_markdownfile.pdf` for complete documentation of all code created

3-minute Demo/Teaser Video

Please refer to a separate file named `Teaser.mp4` for a teaser video