CS595 (Saturday, 9:00-12:00)

COMPUTER SCIENCE CAPSTONE COURSE

San Francisco Bay University, Fremont, CA
FINAL REPORT
Spring 2025

Prof. Ahmed Banafa

Smart Home Energy Management System (SHEMS) with AI Optimization

10 May, 2025

Team Members

Zhiyu (Zoey) Zhang	146344
Nang Thiri Wutyi	154980
Yin Yin Phyo	141168

Contents

Abstract

This project presents a Smart Home Energy Management System (SHEMS) that integrates IoT automation, cloud-based forecasting, and AI-driven recommendations to optimize residential energy usage. The system connects real-time sensor data from a Raspberry Pi with Firebase Cloud Firestore and Google Cloud services to support intelligent appliance control and personalized energy insights.

A fine-tuned Prophet model forecasts daily and hourly energy consumption, while an AI agent generates personalized energy-saving suggestions using historical data, forecasts, and weather inputs via OpenAI and Open-Meteo APIs. Both services are deployed as containerized microservices on Google Cloud Run and fully automated using Cloud Scheduler. Model performance is continuously optimized through a monthly hyperparameter grid search, and all backend services securely manage credentials through Google Secret Manager.

The Swift-based iOS application enables users to remotely control appliances, define custom automation rules, view forecasts and AI suggestions, and receive real-time notifications. Users can also personalize system behavior through settings such as electricity rates and preferred control schedules.

SHEMS demonstrates a cohesive and scalable approach to smart home energy management, combining AI, automation, and user personalization to support both sustainability and cost efficiency.

Why SHEMS?

Energy consumption in homes is often inefficient, with appliances such as lights, fans, and heaters left running unnecessarily, leading to higher electricity costs and wasted energy. Many homeowners forget to turn off devices when they are not in use, or leave appliances running longer than needed due to lack of monitoring or convenience. As energy demands continue to grow, there is an increasing need for smart automation solutions that help manage energy consumption efficiently.

A Smart Home Energy Management System (SHEMS) addresses this issue by integrating real-time sensor data, automation, and remote accessibility to ensure that energy is used only when necessary. By leveraging sensors and cloud connectivity, the system provides a seamless and efficient way to manage household appliances without requiring constant manual intervention. The key benefits from this system include:

1. Energy Efficiency – SHEMS automates appliance control based on real-time conditions. Lights turn on only when motion is detected, and heating/cooling devices respond to temperature changes. AI can further optimize usage patterns to reduce waste and improve efficiency.

2. Cost Savings – By ensuring devices only use energy when needed, SHEMS helps lower electricity bills over time. AI-driven predictions can fine-tune schedules for maximum savings without sacrificing comfort.

3. Automation & Smart Control – The system runs autonomously using data from sensors, with the Raspberry Pi making intelligent decisions. AI analyzes usage trends to adapt to user habits and environmental changes for a seamless experience.

4. Remote Monitoring & AI Optimization – With Cloud Firestore, users can monitor and control appliances remotely via a mobile app. AI enhances this by learning from data and recommending adjustments, like heater timing based on weather or lighting based on occupancy.

In conclusion, SHEMS provides an automated, AI-driven, and data-backed solution to common household energy challenges. It offers a combination of efficiency, cost reduction, and remote accessibility, making it a practical and scalable approach for modern smart homes. With its ability to reduce energy waste, optimize device operation using AI, and improve user convenience, SHEMS represents an important step toward sustainable and intelligent energy management.

System Overview

The Smart Home Energy Management System (SHEMS) is designed to optimize energy consumption by integrating real-time data collection, cloud-based storage, and AI-driven analysis. The system allows users to monitor and control appliances remotely through an iOS mobile app, while automation ensures efficient energy usage.

System Components

1. **Users & Mobile App**

   Users interact with SHEMS through an iOS mobile app, which allows them to:

   - Register and authenticate using Firebase Authentication
   - View real-time energy usage and appliance status
   - Remotely control appliances such as lights and heaters

2. **Sensors & Raspberry Pi**

   Sensors (motion and temperature) collect real-time data about the environment. The Raspberry Pi processes this data and:

   - Determines when to turn appliances ON or OFF
   - Sends real-time sensor data to Firestore for cloud storage
   - Ensures automation based on energy-saving logic

3. **Cloud Firestore & AI Analytics**

   Cloud Firestore acts as the database, storing:

   - Sensor readings
   - Appliance status updates
   - User preferences and settings

AI analytics process this data to:

- Identify patterns in energy usage and make predictions
- Optimize appliance schedules for efficiency
- Provide smart recommendations for further energy savings

4. **Appliance Control & Energy Optimization**

   The Raspberry Pi sends control signals to smart appliances, such as:

   - Turning the heater or fan ON/OFF based on temperature changes
   - Activating lights based on motion detection
   - Allowing remote ON/OFF commands via the mobile app

How does the System work?

The following diagram shows the overview of how our SHEMS works.

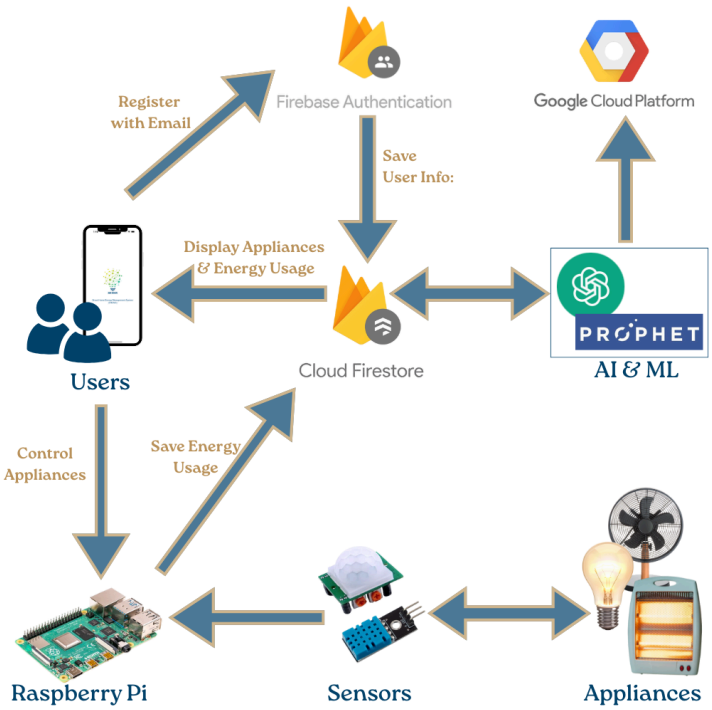


Figure 1. System Overview

1. Users register on the mobile app using Firebase Authentication with Email.
2. Sensors collect data (motion, temperature) and send it to the Raspberry Pi.

3. The Raspberry Pi processes the data and decides whether to turn appliances ON or OFF.
4. Sensor readings and appliance status updates are stored in Cloud Firestore.
5. AI processes data to optimize energy efficiency and generate insights.
6. Users can remotely control appliances from the mobile app based on live data.

Hardware and Cloud Integration

Nang Thiri Wutyi

1. Role and Contributions

As an Electrical Engineer and an IoT developer, my role in our Smart Home Energy Management System (SHEMS) is hardware integration, device automation, and cloud-based data logging. My work focused on ensuring reliable data collection from the environment, real-time control of smart appliances, and automatic operation of the system without manual intervention. Collaborating with my teammates, who focused on mobile and web development, I provided the sensor data backbone and device control layer required for remote access and intelligent energy decisions.

2. Hardware and Sensor Integration

To gather environmental data and control smart devices, I used the following components, chosen for their compatibility and ease of integration with the Raspberry Pi:

Component	**Purpose**	**Why was it chosen?**
Raspberry Pi 4 Model B	Main controller handling sensor data, processing, and cloud communication.	Supports Python and works seamlessly with Firebase and smart plugs.
PIR Motion Sensor	Detects human motion for automated device control.	Reliable, low-power, and connects easily to GPIO pins.

DHT11 Temperature Sensor	Monitors room temperature to trigger heating or cooling.	Accurate for basic use, cost-effective, and simple to set up.
Kasa Smart Plugs(HS103P2)	Controls appliances (light, heater, fan) based on sensor input.	WiFi-controlled, integrates with Python, and requires no GPIOs.

Table 1. Hardware Components, their purposes and why they were chosen

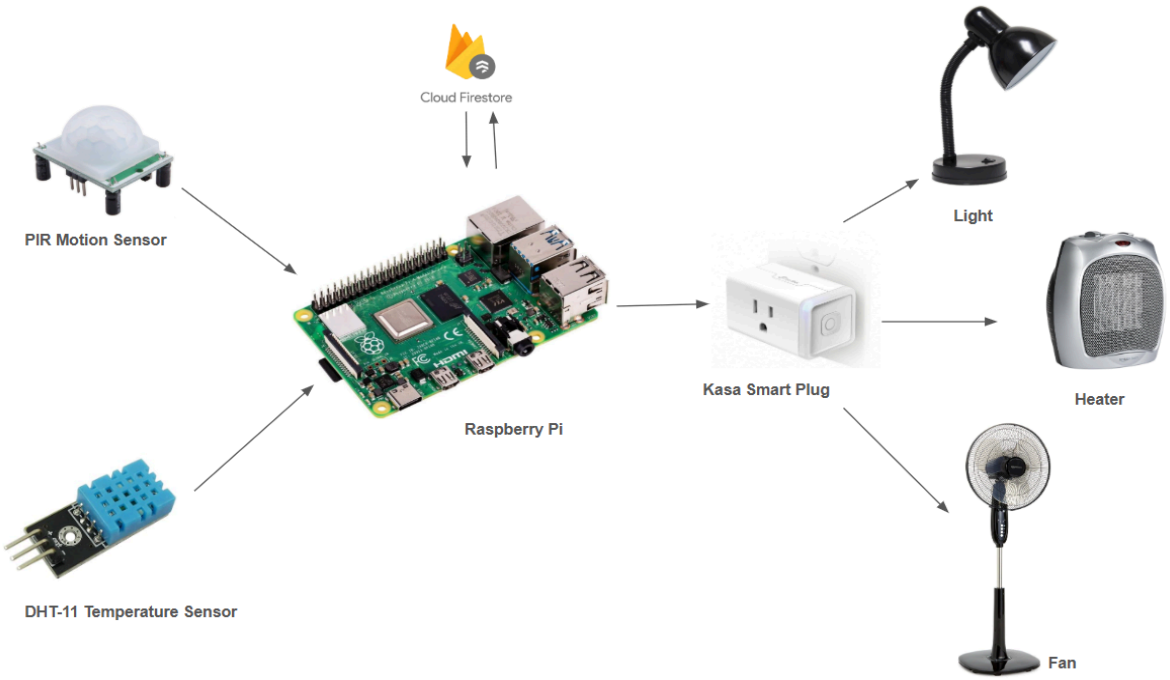


Figure 2. IoT Hardware and Cloud Overview

The above diagram illustrates the Smart Home Energy Management System (SHEMS) hardware and cloud integration. The PIR motion sensor detects movement to control devices, such as turning the light on when motion is detected and turning devices off if no motion is detected for

1 hour. The DHT11 temperature sensor monitors room temperature to automate the heater and fan based on predefined thresholds. The Raspberry Pi serves as the central controller, processing sensor data and sending commands to the Kasa smart plugs over WiFi. It also uploads real-time sensor readings and device statuses to Cloud Firestore, enabling remote monitoring and future mobile app integration. This setup ensures automated device control based on real-time conditions, optimizing energy efficiency and convenience.

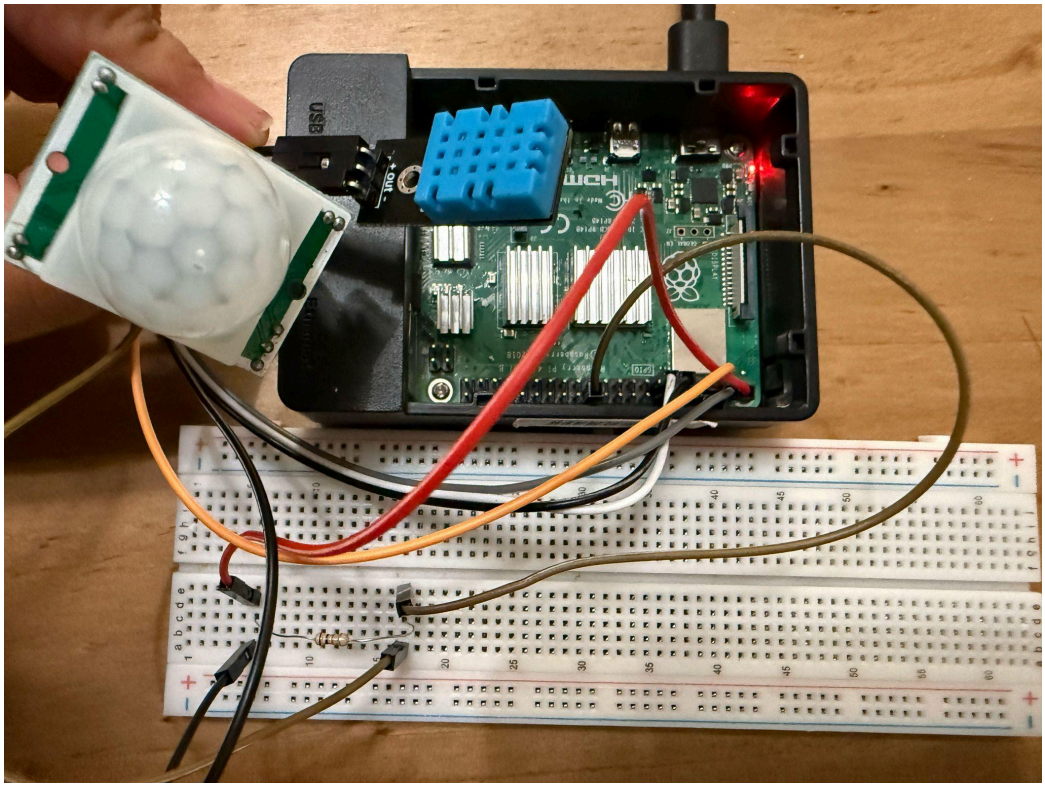


Figure 3. Hardware Setup (Raspberry Pi, PIR Motion Sensor, DHT 11 Temperature Sensor with Pull-up Resistor)

3. Cloud Integration with Firestore

To allow real-time remote monitoring, I integrated Firebase Firestore to store and retrieve sensor data and device states. The database is updated continuously with values such as temperature, motion detection status, and smart plug activity.

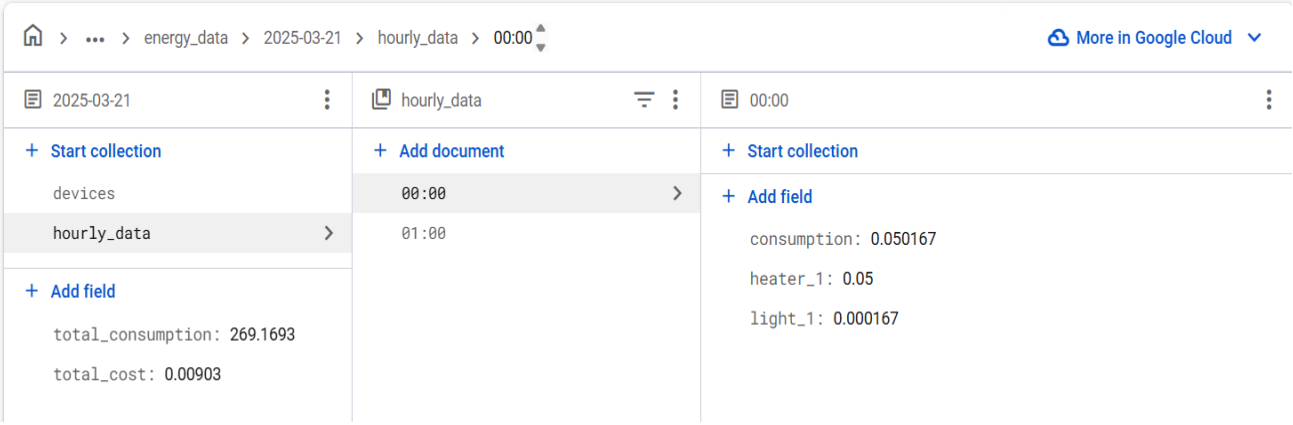


Figure 4. Hourly Energy Data Uploaded in Firestore

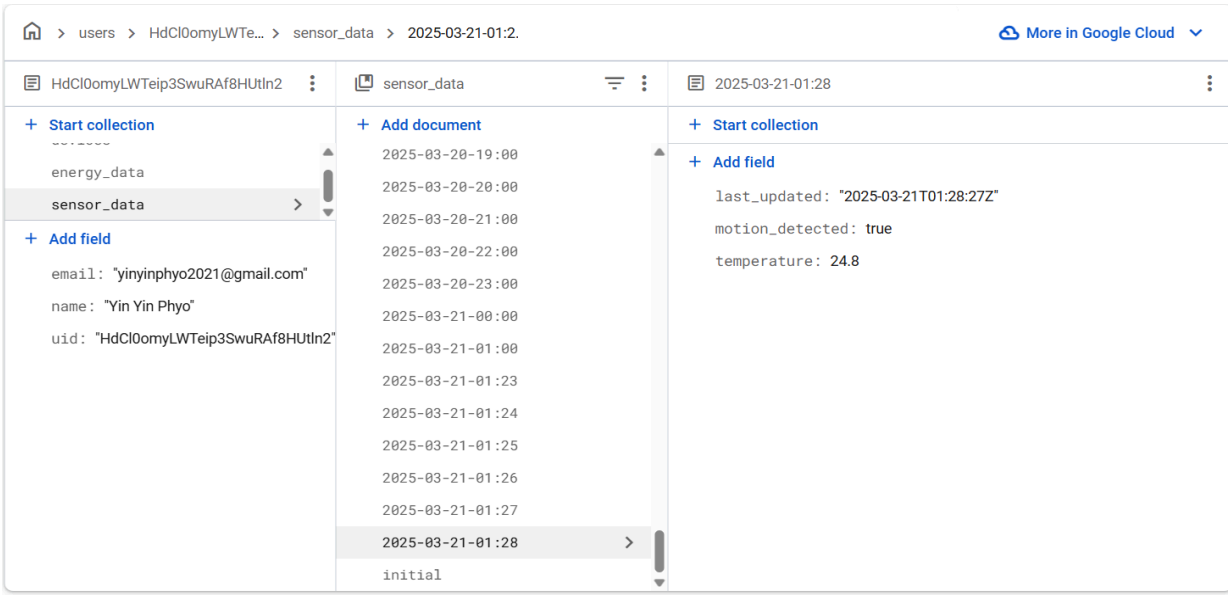


Figure 5. Hourly Updated Sensor Status

4. Ensuring System Reliability

To ensure uninterrupted data transmission and automation, various reliability measures were implemented:

WiFi Auto-Reconnect - Automatically reconnects if WiFi disconnects.

Systemd Service for Auto-Restart - Ensures the script runs on boot and restarts if it crashes.

Persistent Execution Using `tmux` - Keeps the script running even after SSH disconnection.

5. Hardware and Cloud Integration (Final Phase)

This section outlines the implementation and refinement of remote control logic, manual override handling, and automated appliance management based on environmental sensor data. It focuses on completing the roadmap set during the midterm report, emphasizing system robustness, efficiency, and energy-saving behavior.

5.1 Manual ON/OFF Command and Energy-Saving Automation

To provide users with real-time control, I implemented a system where the iOS mobile app can send ON or OFF commands to smart plugs (for both light and heater) by updating their respective Firestore documents. Each device has a unique document in Firestore containing fields such as **isOn** and **lastUpdated**. When a user toggles a device from the app, these fields are updated.

A Python listener script running on the Raspberry Pi constantly monitors these Firestore fields. When a change is detected, the Raspberry Pi sends an appropriate command to the smart plug over WiFi, instantly turning the device ON or OFF. This setup allows seamless remote control of devices from anywhere with internet access.

However, to prevent users from accidentally leaving appliances running for long periods—which would defeat the purpose of energy management—I implemented a 1-hour manual override window. When a device is turned on manually via the app, automation is temporarily suspended for that device. After 1 hour, automation resumes automatically based on the latest sensor readings.

For example, if the heater is manually turned ON from the app, it will stay on for up to one hour unless turned OFF earlier by the user. Once that hour has passed, the automation logic evaluates

the room temperature again. If the temperature is still above the threshold (25°C), the system turns the heater OFF automatically. Similarly, the light turns OFF if no motion has been detected during that hour.

This structure ensures user flexibility while preventing unnecessary energy consumption. It strikes a balance between convenience and sustainability.

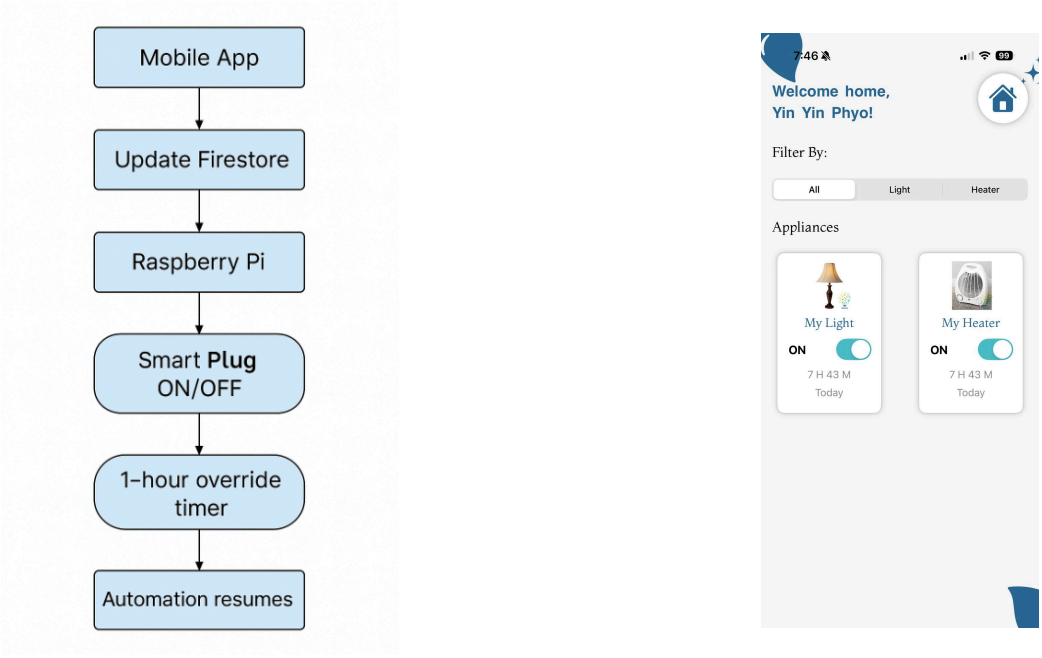


Figure 6. Mobile Command Flow and Mobile App Interface for Appliance Control

5.2 Energy Efficiency Through Automation

The automation system plays a crucial role in reducing energy waste. Unlike manual control—which can leave devices running longer than needed—automation ensures that appliances only operate when truly necessary, based on environmental conditions.

Here's how automation reduces energy consumption:

- Lights only turn ON when motion is detected and turn OFF if no motion is detected for 1 hour.

- Heaters are triggered only when temperature drops below 25°C and turned OFF automatically when it rises back to or above that point.
- No device runs continuously or unnecessarily—every decision is driven by live sensor input and optimized logic.

This approach not only lowers the energy bill but also contributes to environmental sustainability by minimizing power usage. It enables the system to self-regulate without requiring constant user attention, making SHEMS a reliable and eco-friendly home automation solution.

Figure 7. Daily Total Consumption and Cost in Firestore After Applying Automation Logic

5.3 Final Tests and Optimization

During the final phase of the project, I conducted a comprehensive series of tests and optimizations to ensure the system was robust, responsive, and energy-efficient.

- **System Reliability:** The Raspberry Pi was configured with a **systemd** service to automatically restart the listener script upon reboot. Testing confirmed that the system successfully resumed operations after power or network interruptions.
- **Manual Override Validation:** The 1-hour manual override mechanism was thoroughly tested. When a device was manually activated via the mobile application, automation was temporarily suspended and resumed precisely after the one-hour window based on current sensor data.

- **Automation Logic Verification:** Automation rules were validated in a real-time environment. The system consistently turned the light ON upon motion detection and OFF after one hour of inactivity. The heater responded accurately to temperature thresholds, activating below 25°C and deactivating at or above 25°C.
- **Dynamic Billing Rate Testing:** The flexible billing mechanism was verified by updating the **rate** field in Firestore. Cost calculations reflected changes immediately without requiring a system restart, enabling user-specific and location-based energy billing.
- **Performance Optimization:** To ensure system efficiency and reduce database overhead, sensor data and energy metrics were written to Firestore at one-minute intervals. Redundant operations were minimized to improve performance within Firebase usage limits.
- **Mobile Application Integration:** Close collaboration with the mobile development team ensured seamless synchronization between hardware actions and the user interface. Real-time ON/OFF commands, device status updates, and energy usage data were successfully tested and confirmed to display correctly within the application.

Model Development, Deployment, and Database Design

Zhiyu (Zoey) Zhang

1. Role and Contributions

As the AI and backend systems developer, I was responsible for designing the energy consumption forecasting model, implementing the AI suggestion engine, managing backend deployment and automation, and architecting the Firestore database for scalable and real-time access. My work spanned the full lifecycle of backend development, from modeling and optimization to secure, production-ready deployment.

2. Forecasting Model Development

2.1 Model Development Overview

The forecasting component of SHEMS was built using Facebook's Prophet library to predict household energy consumption. Initial development used the UCI Individual Household Electric Power Consumption dataset, which was preprocessed into hourly and daily formats to accommodate different use cases (see Figure 8 and 9).

```
df_hourly.head()
```

datetime	Global_active_power	Sub_metering_1	Sub_metering_2	Sub_metering_3
2006-12-16 17:00:00	2.533733	0.0	19.0	607.0
2006-12-16 18:00:00	3.632200	0.0	403.0	1012.0
2006-12-16 19:00:00	3.400233	0.0	86.0	1001.0
2006-12-16 20:00:00	3.268567	0.0	0.0	1007.0
2006-12-16 21:00:00	3.056467	0.0	25.0	1033.0

Figure 8. Preprocessed hourly data from UCI public dataset

```
df_daily.head()
```

datetime	Global_active_power	Sub_metering_1	Sub_metering_2	Sub_metering_3
2006-12-16	20.152933	0.0	546.0	4926.0
2006-12-17	56.507667	2033.0	4187.0	13341.0
2006-12-18	36.730433	1063.0	2621.0	14018.0
2006-12-19	27.769900	839.0	7602.0	6197.0
2006-12-20	37.095800	0.0	2648.0	14063.0

Figure 9. Preprocessed daily data from UCI public dataset

I evaluated three configurations: a daily model, an hourly model, and a daily aggregate built from hourly predictions. The model was configured using:

- seasonality_mode="multiplicative"
- changepoint_prior_scale=0.2
- holidays_prior_scale=10
- Custom seasonalities:
    - Weekly (period=7, fourier_order=15)
    - Monthly (period=30.5, fourier_order=10)

Performance evaluation showed that the **hourly model provided the best accuracy**, supporting high-resolution forecasting with improved responsiveness to usage changes (see Figure 10).

	Model	MAE	MSE	RMSE
0	Hourly	0.520450	0.498458	0.706015
1	Daily	5.411585	54.160145	7.359358
2	Aggregated Hourly	5.381160	50.564797	7.110893

Figure 10. Evaluation results for 3 models

2.2 Fine-Tuning with User Data

After establishing a robust baseline, I shifted focus to personalized forecasting by incorporating energy data from a real household as a sample user. The data was cleaned, normalized, and split into training (first 11 months) and testing (last month). Figure 11 and 12 show the daily and hourly energy consumption from this user.

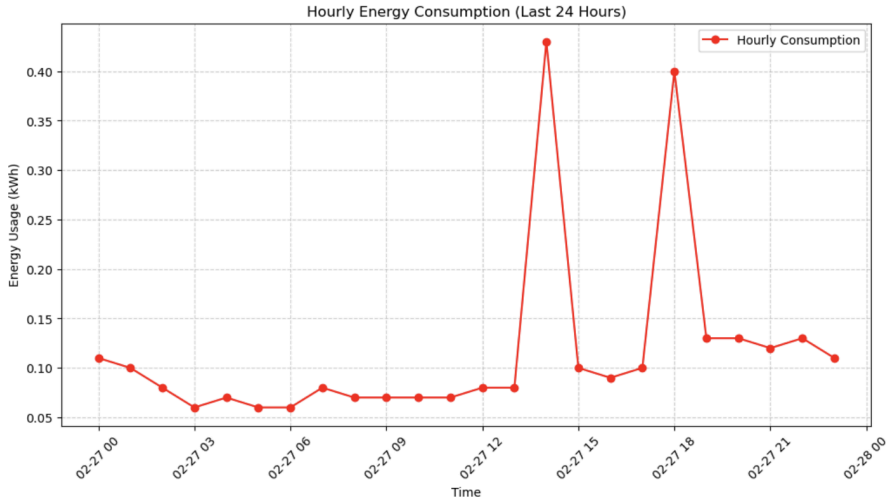


Figure 11. Hourly energy consumption of sample user

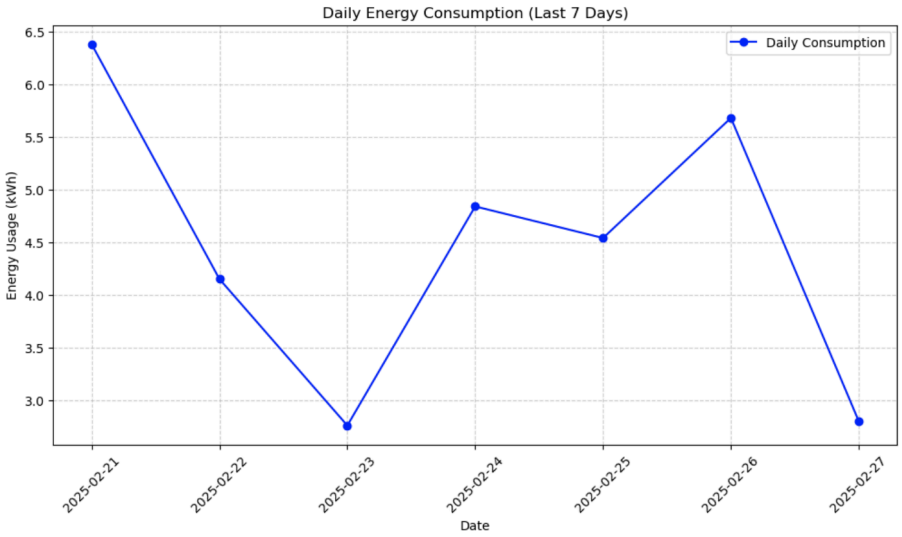


Figure 12. Daily energy consumption of sample user

I then retrained the Prophet model on this data using:

- changepoint_prior_scale=0.05
- holidays_prior_scale=20

- seasonality_mode="multiplicative"
- Custom seasonality for weekly and monthly patterns using fourier_order=15

and compared the prediction with actual energy consumption data, as shown in Figure 13 and 14.

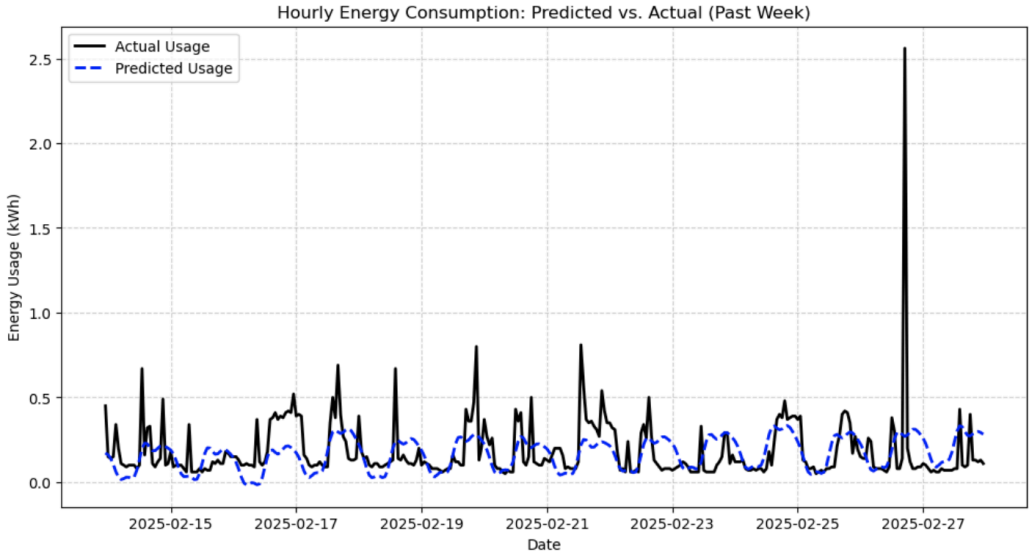


Figure 13. Predicted vs. actual hourly energy consumption of sample user

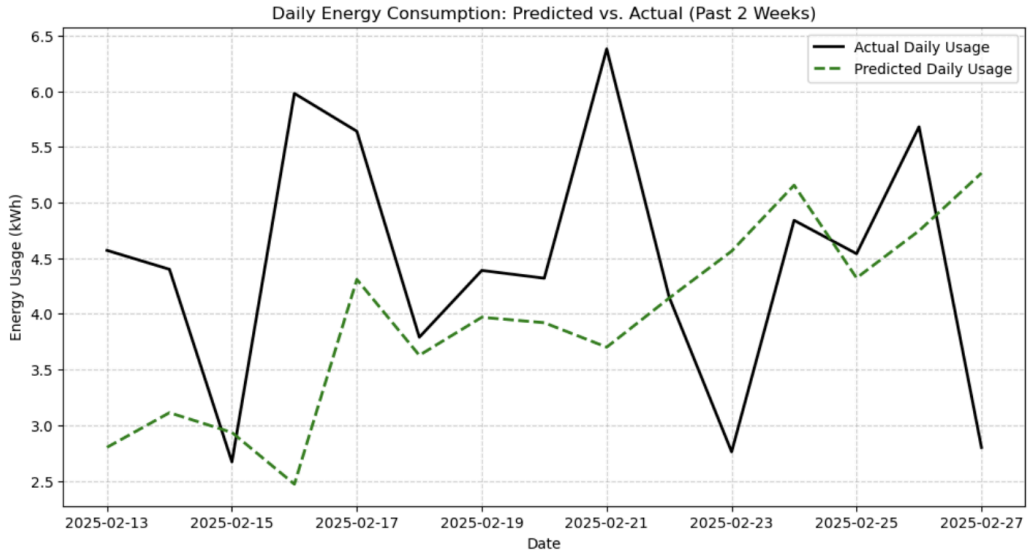


Figure 14. Predicted vs. actual daily energy consumption of sample user

As shown in Figure 15, this fine-tuned model achieved strong predictive accuracy with an RMSE of 0.179 kWh for hourly prediction and 1.56 kWh for daily (aggregated from hourly). The personalized model adapts better to the unique energy habits of individual users.

	Model	MAE	MSE	RMSE
0	Hourly (Fine-Tuned)	0.107321	0.032279	0.179663
1	Daily (Aggregated Hourly)	1.171207	2.455835	1.567110

Figure 15. Evaluation of the fine-tuned model for the sample user

2.3 Model Optimization

To ensure reliable and interpretable forecasts across diverse users, I implemented several model optimization techniques beyond hyperparameter tuning:

- **Data Filtering**: Each user's training set is dynamically filtered by time range using timestamp constraints to ensure temporal consistency and avoid data leakage.
- **Log Transformation**: While not used in the final pipeline, log-scaling (`log1p`) was tested as a stabilizing transformation for highly variable energy usage. This option remains available for future tuning scenarios.
- **Value Clipping**: To prevent negative or unstable model outputs, I implemented lower-bound clipping during both training and inference.
- **Missing Data Handling**: The input forecast window is forward- and backward-filled to mitigate the impact of missing hourly data.
- **Inverse Transformations**: For interpretability, post-forecast values undergo exponential transformation to revert from log-space (if applied), followed by additional clipping to maintain physical realism.
- **Debugging and Logging**: Intermediate input and forecast dataframes are saved to CSV logs for every user during batch runs to facilitate reproducibility and error tracing.
- **Electricity Rate Customization**: The model now incorporates a user-defined rate stored in Firestore. Forecasted consumption values are multiplied by this rate to generate

personalized daily and monthly cost projections, which are surfaced in both the app interface and AI agent outputs.

Forecasting performance is further maintained through an automated monthly **grid search pipeline** that re-optimizes Prophet hyperparameters per user. The best-performing parameters are stored in Firestore to support version tracking and future re-tuning.

Additionally, I optimized the backend prediction pipeline to **minimize Firestore reads and writes** by batching database interactions and caching reusable queries, improving both performance and scalability across users. The predictions are written intoFirestore as shown in Figure 16.

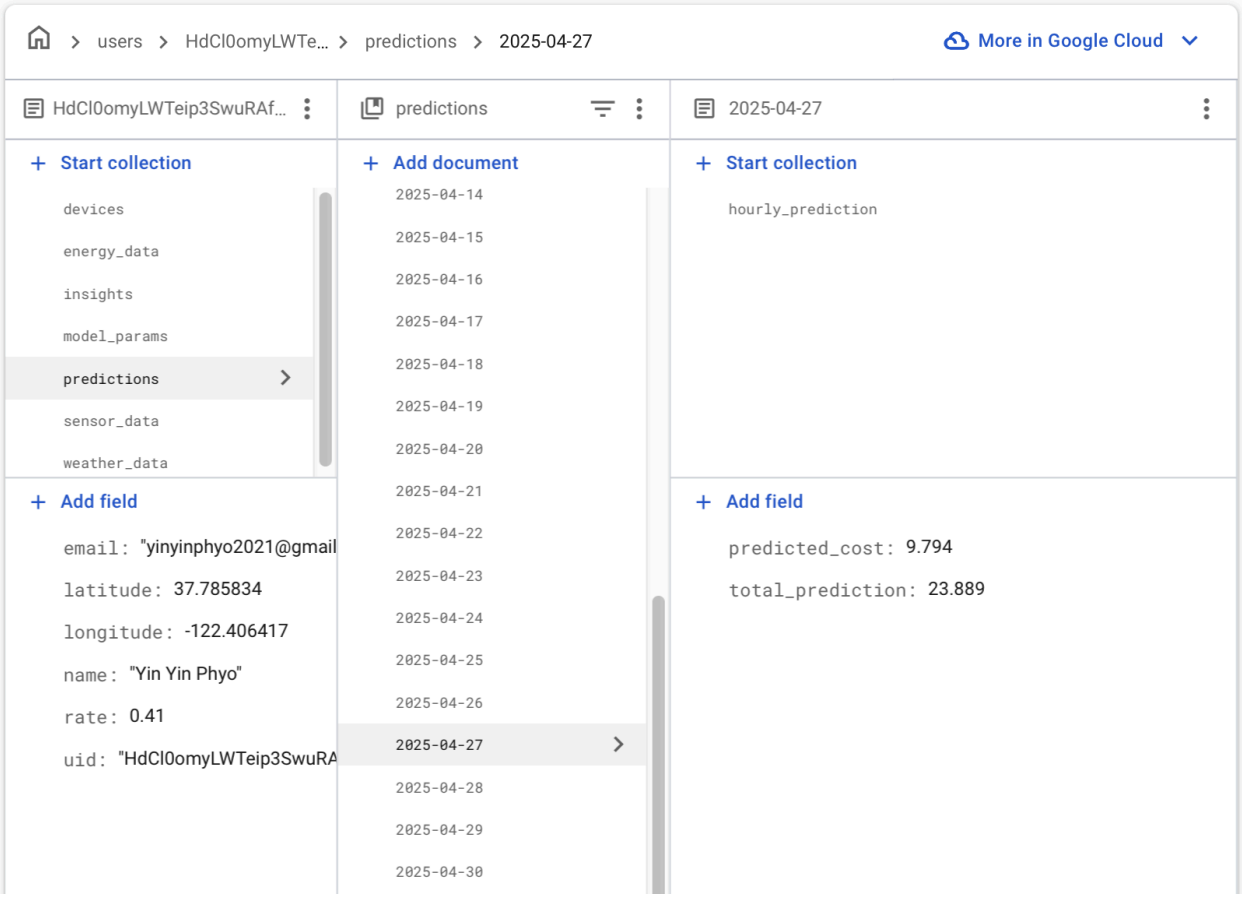


Figure 16. Predictions stored in Firestore

3. AI Energy Optimization API

I developed a cloud-based AI agent called the **AI Energy Optimization API**, which generates personalized energy-saving suggestions daily for each user. The agent combines:

- Forecast outputs from the Prophet model
- Historical energy usage (device-level and aggregate)
- Weather forecast data from the **Open-Meteo API**

Based on these inputs, the system formulates targeted recommendations—such as reducing usage of a specific appliance at predicted high-load periods or adjusting heating/cooling plans ahead of hot or cold days. The agent uses the **OpenAI API (gpt-4o-mini)** to generate human-readable suggestions contextualized to each household.

All suggestions are automatically generated each morning, stored in Firestore under an insights collection (see Figure 17), and displayed in the user's mobile app interface.

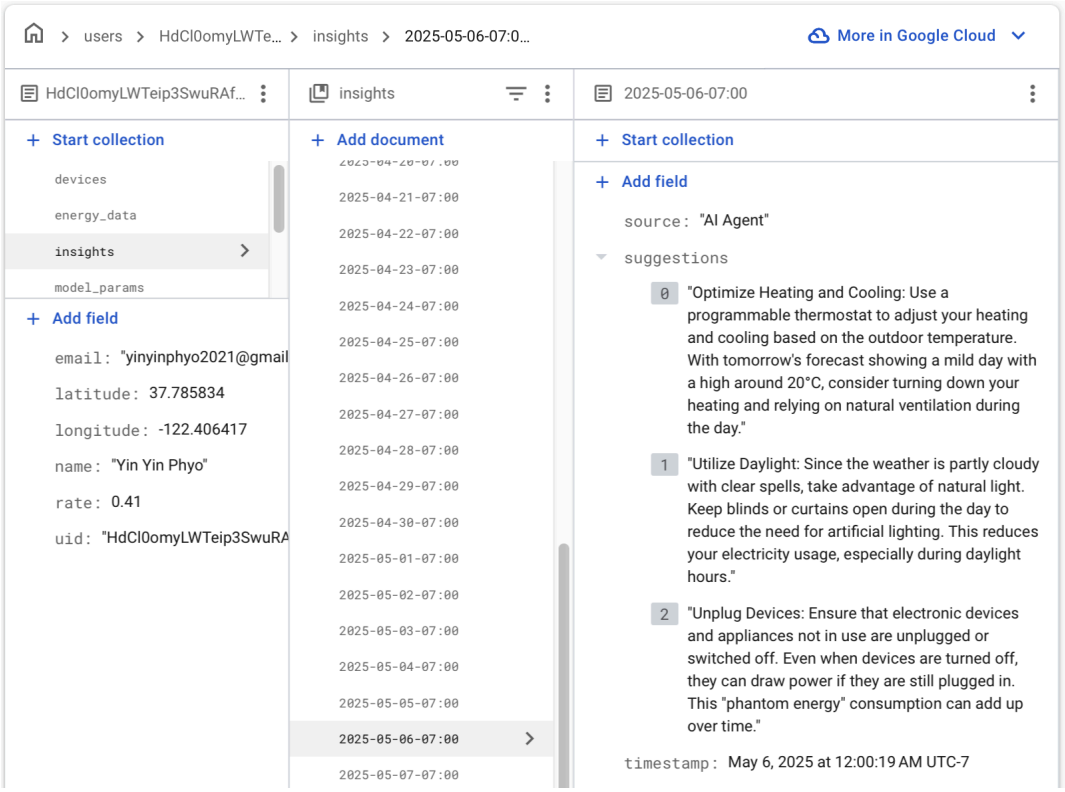


Figure 17. AI suggestions stored in Firestore

4. Cloud Deployment and Automation

To ensure flexibility, scalability, and long-term maintainability, I deployed the forecasting model and AI agent as independent microservices on **Google Cloud Run**, fully decoupled from the mobile frontend and hardware systems (see Figure 18). This modular deployment architecture allows each service to be updated, optimized, or replaced independently without impacting other components of the system.

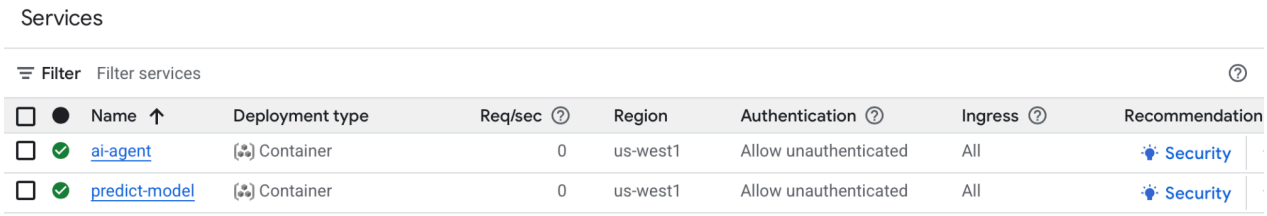


Figure 18. Microservices on Google Cloud Run

Both the forecasting pipeline and the AI Energy Optimization API were **containerized using Docker**, enabling reproducible builds, fast deployment cycles, and environment isolation. To transition from development to production, I replaced the default Flask development server with **Gunicorn**, a WSGI-compliant server that offers better concurrency handling and process management under production workloads (see Figure 19).

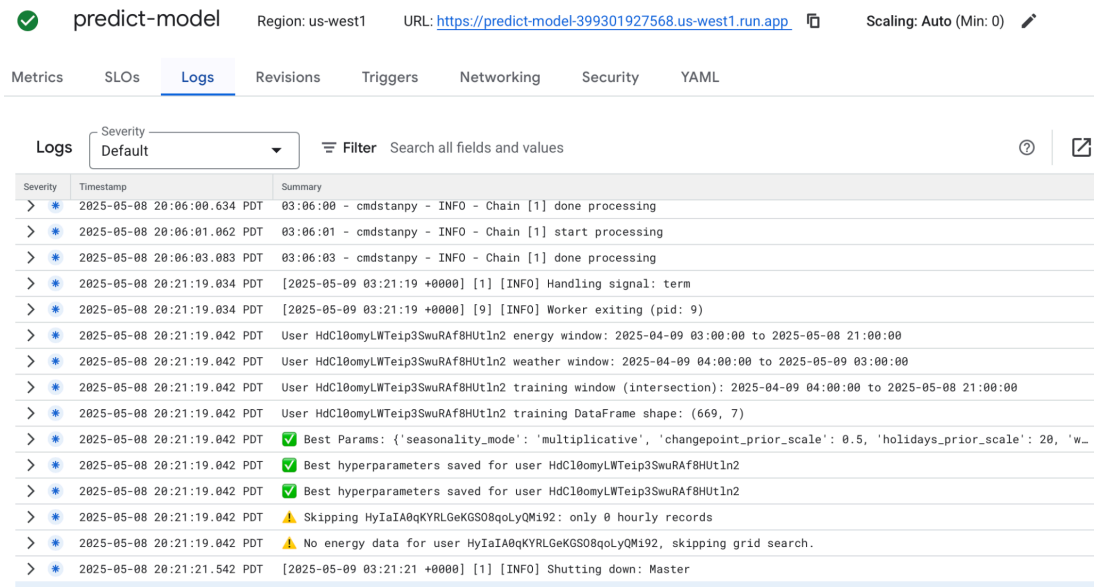


Figure 19. Logs of the forecasting model service with Gunicorn

Three RESTful API endpoints were implemented for automation and internal orchestration:

- GET /get_predictions: Executes the daily forecast pipeline, retrieving recent energy data from Firestore, generating household-specific predictions using the Prophet model, and storing the results.
- POST /run_grid_search: Triggers the monthly grid search to evaluate and update the optimal Prophet hyperparameters for each user, improving long-term accuracy.
- POST /run_ai_suggestions: Calls the AI Energy Optimization API, which combines forecast results, user history, and weather data to generate and store personalized energy-saving suggestions using the OpenAI API.

To automate backend execution, I configured **Google Cloud Scheduler** with three cron jobs:

- A **daily trigger** for /get_predictions, scheduled at midnight PST, to ensure each user receives an updated forecast every morning.
- A **daily trigger** for /run_ai_suggestions, shortly after prediction completion, to generate energy-saving tips tailored to the updated forecast and environmental context.
- A **monthly trigger** for /run_grid_search, to adapt the model configuration to seasonal changes and evolving usage behavior.

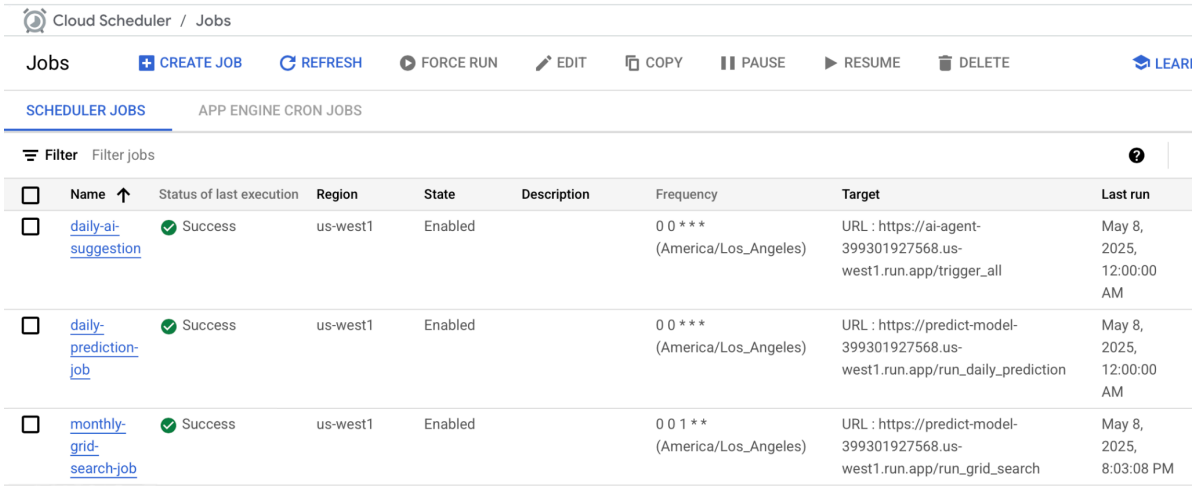


Figure 20. Cron jobs on Google Cloud Scheduler

All external API keys—including OpenAI (used for the AI agent) and Open-Meteo (used for weather integration)—are securely managed using **Google Cloud Secret Manager**. This ensures that sensitive credentials are encrypted, auditable, and never hard-coded into deployment artifacts.

In addition to job scheduling, I implemented a **failure monitoring and alert system** that sends email notifications if a service or scheduler fails, helping ensure system uptime and reliability (see Figure 21).

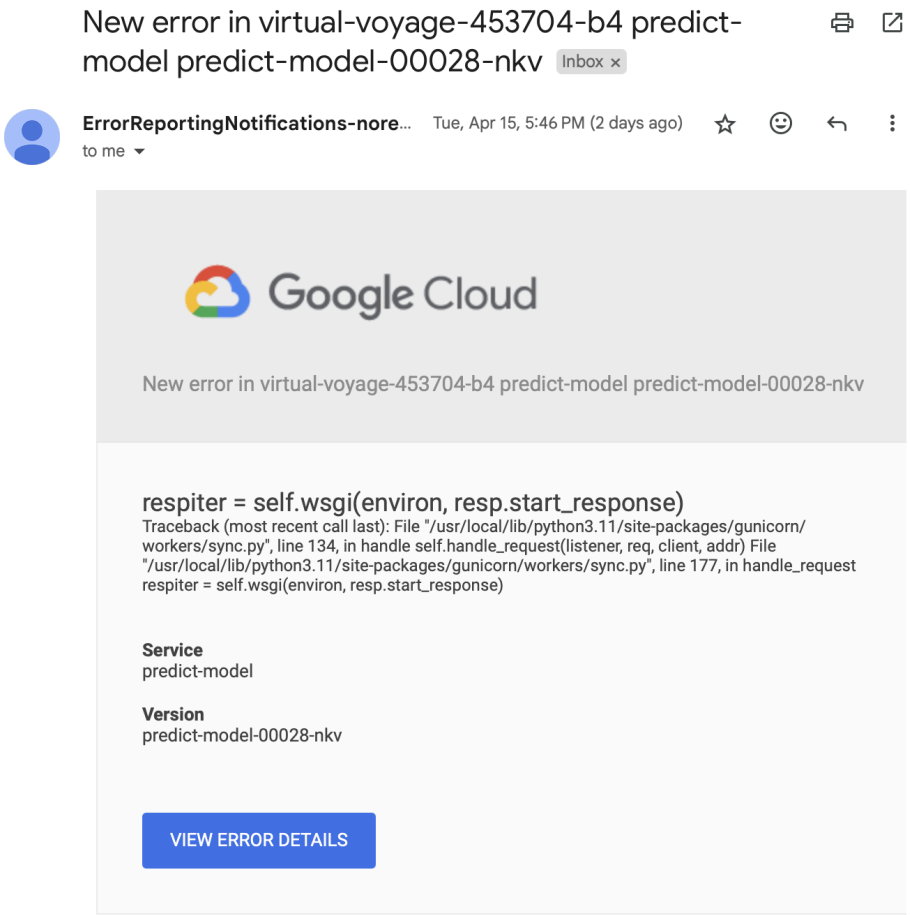


Figure 21. Email notification for service failure

All prediction results and AI suggestions are written directly to Firestore and consumed asynchronously by the iOS application, which avoids the need for the mobile frontend to communicate with model services directly.

This infrastructure supports continuous learning, dynamic scaling, and modular extensibility—enabling real-time, cloud-native AI forecasting and energy optimization at scale.

5. Firestore Database Design

To support long-term scalability, real-time updates, and integration with the Swift app and AI agent, we designed a hierarchical, modular Firestore schema. It accommodates both aggregate and device-level energy data, real-time sensor readings, and predictions.

```
/users/{user_id}/
    ├── {name, email, latitude, longitude, rate, uid}
    ├── energy_data/
    │   ├── {YYYY-MM-DD}/
    │   │   ├── total_consumption, total_cost
    │   │   ├── hourly_data/{HH:00} → {consumption}
    │   │   └── devices/{device_id} → {isOn, consumption, cost,
usageTime, last_updated}
    ├── devices/
    │   └── {device_id} → {category, id, name, image}
    ├── predictions/{YYYY-MM-DD} → {predicted_cost,
total_prediction, hourly_prediction}
    ├── insights/{YYYY-MM-DD-HH:00} → {source, suggestions}
    ├── weather_data/{YYYY-MM-DD-HH:00} → {temperature, humidity,
cloud_cover, wind_speed, solar_radiation, precipitation}
    ├── sensor_data/{YYYY-MM-DD-HH:00} → {temperature,
motion_detected, last_updated}
    └── model_params/best_params → {seasonality_mode,
changepoint_prior_scale, holidays_prior_scale, weekly_fourier_order,
monthly_fourier_order, score, timestamp}
```

This schema ensures fast lookups by user and date, supports historical logging, and provides flexible document-level access for both frontend (app display) and backend (training, inference).

Mobile Application Development

Yin Yin Phyo

1. Role and Contributions

As a Full-Stack developer, my role in the Smart Home Energy Management System (SHEMS) involved preparing requirements for iOS application development, designing the wireframe prototype, and developing the iOS application. My primary responsibilities focused on creating UI pages, integrating data from the Firestore database, and enabling appliance control through the application by interfacing with the hardware device.

Collaborating with teammates responsible for hardware implementation and AI model development, I ensured efficient integration of IoT data and AI-driven energy consumption predictions. This ensured a seamless user experience, efficient energy consumption tracking, and automation for smart home appliances.

2. Requirement Specification for Mobile Application

The mobile application for the Smart Home Energy Management System is designed to provide an intuitive interface for users to manage their smart home devices, monitor energy consumption, and automate energy-saving actions.

2.1 Login & Authentication Screen

- The application supports secure user authentication, allowing users to register and log in using their email and password.
- A password recovery option is also available to reset credentials when necessary.

2.2 Landing Page Screen

- Upon login, users are presented with an overview of their connected smart devices. This screen displays device statuses, current power consumption statistics, and quick control toggles for appliances.
- Navigation links direct users to the main home control page, where they can access features such as energy monitoring, AI-based energy-saving suggestions, and application settings.

2.3 Device Control Screen

- Users can remotely manage their smart home appliances through intuitive control mechanisms. The app provides options to switch devices on and off manually.
- It also provides automatically based on motion sensor detection. This ensures efficient energy utilization by turning off appliances when no movement is detected in a room.

2.4 Energy Monitoring Screen

- The application includes a comprehensive energy monitoring system that visualizes real-time power usage through interactive graphs.
- Users can analyze their weekly and monthly energy consumption trends, estimate upcoming energy bills, and receive AI-driven recommendations for reducing energy costs.

2.5 Automation Screen

- Automation features allow users to create customized rules for their smart home appliances. Users can schedule device operations based on specific conditions, such as time-based automation or environmental triggers. This feature enhances convenience and energy efficiency by ensuring appliances operate only when needed.

2.6 Settings Screen

- The Settings section allows users to customize key preferences for the application, including setting the energy rate (default or custom), selecting the app theme (System Default, Light, or Dark), and managing notification preferences such as Appliance ON/OFF Alerts, AI Suggestions, and Energy Usage Prediction Updates.

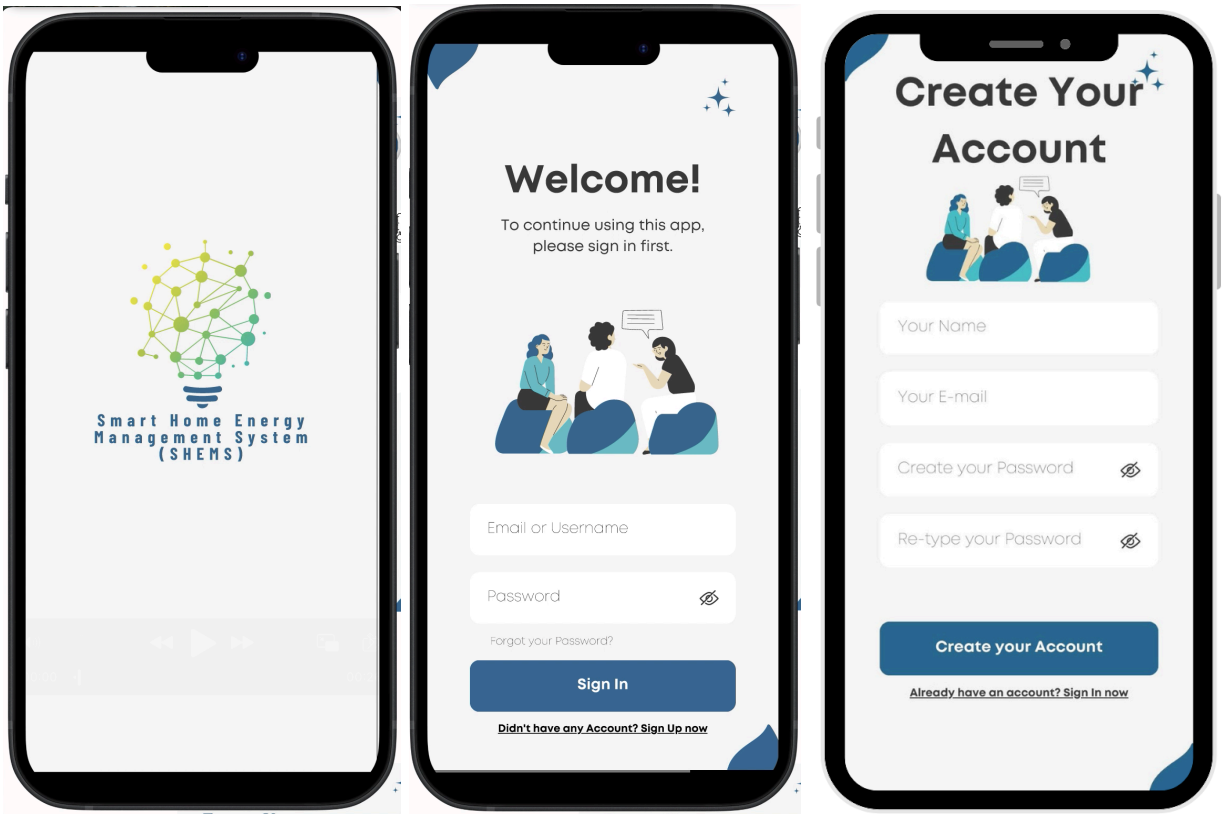
3. Tech Stack for Mobile Application

- **Language & Framework:** Swift, SwiftUI
- **Authentication:** Firebase Authentication (Email/Password)
- **Real-time Communication:** WebSockets or MQTT (for instant device updates)
- **Database:** Firebase Firestore (for user data and device states)
- **Hosting:** Firebase Hosting (for app resources)
- **Automation:** Firebase Firestore rules

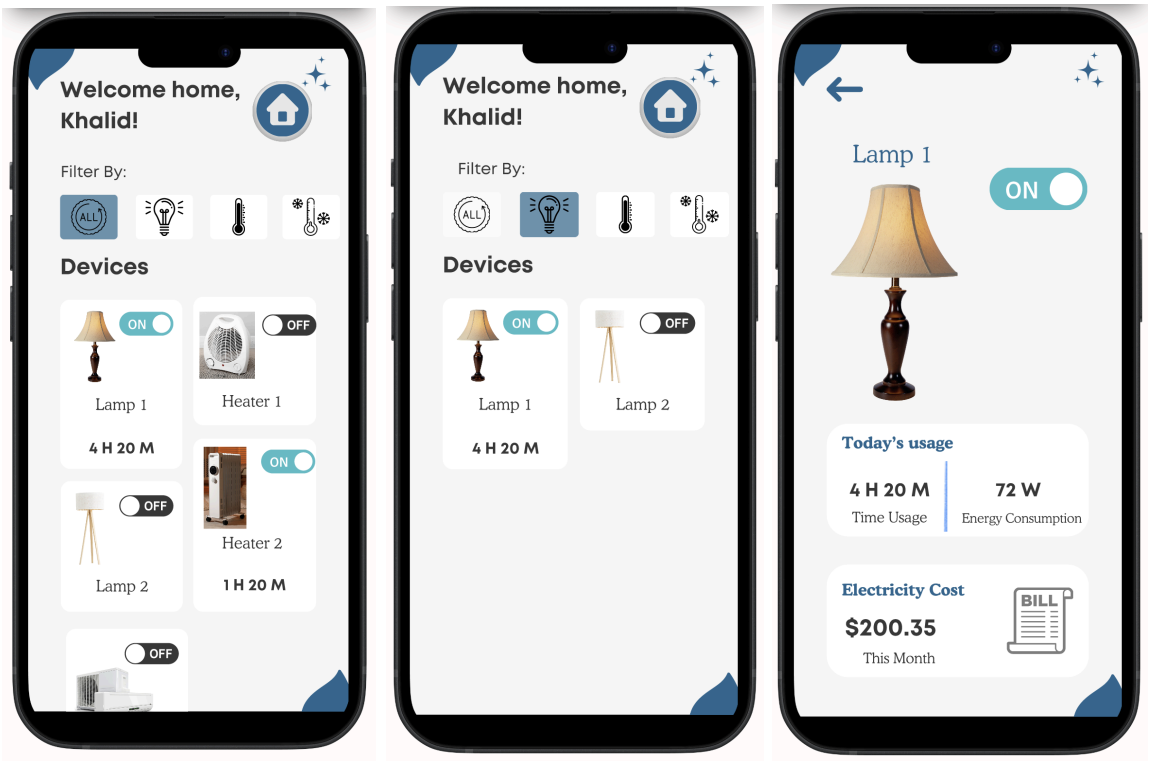
4. Prototype Design

The prototype design phase involved creating interactive wireframes and UI mockups using **Canva** and **Figma**. These tools allowed for rapid design iterations and user interface refinements before implementation. The interactive prototype provides a visual representation of the application's layout, user flows, and key functionalities, ensuring a seamless user experience.

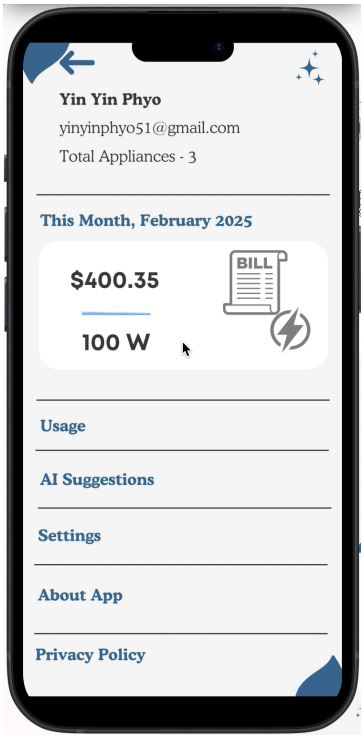
4.1 Splash Screen, User Sign-In and Registration Pages



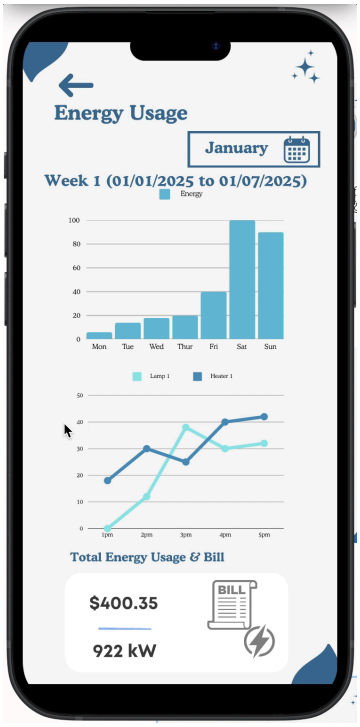
4.2 Landing Page and Detail Appliance Page

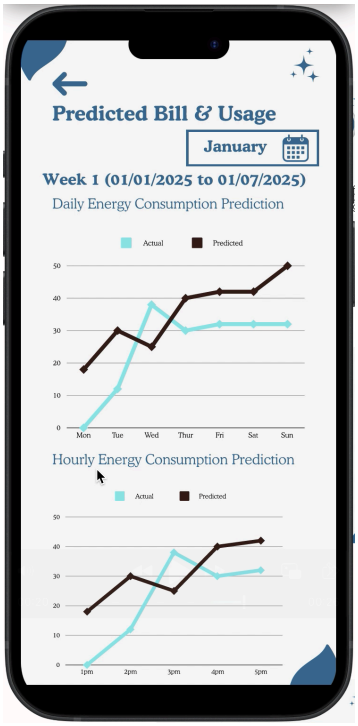


4.3 User Home Page



4.4 Energy Usage and AI Suggestions Pages





5. UI Development & Implementation

The UI development phase involved translating the designed prototypes into functional SwiftUI code within Xcode. The user interface was implemented using SwiftUI components for a modern and responsive design. Key features, such as authentication screens and device management, were developed with a focus on user experience and performance.

The MVVM (Model-View-ViewModel) architecture was adopted to structure the project efficiently. This architecture helps separate concerns, ensuring the UI is decoupled from the business logic. Each screen in the app has its corresponding ViewModel, which is responsible for handling data and logic, while Views focus on displaying the UI.

5.1 Project Folder Structure

The app's folder structure is shown in Figure 22 to follow the MVVM pattern and maintain an organized project.

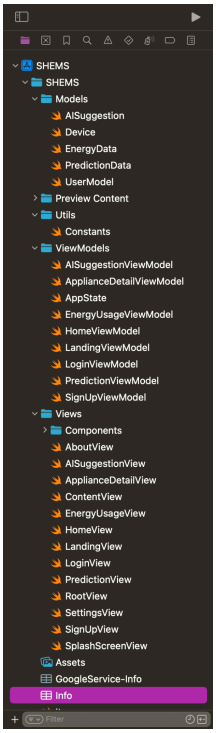


Figure 22. Swift Project Structure

5.2 User Sign-In and Registration Pages

The sign-in and registration pages were built using SwiftUI, integrating Firebase Authentication for secure login and account creation. The UIs show as Figure 23:

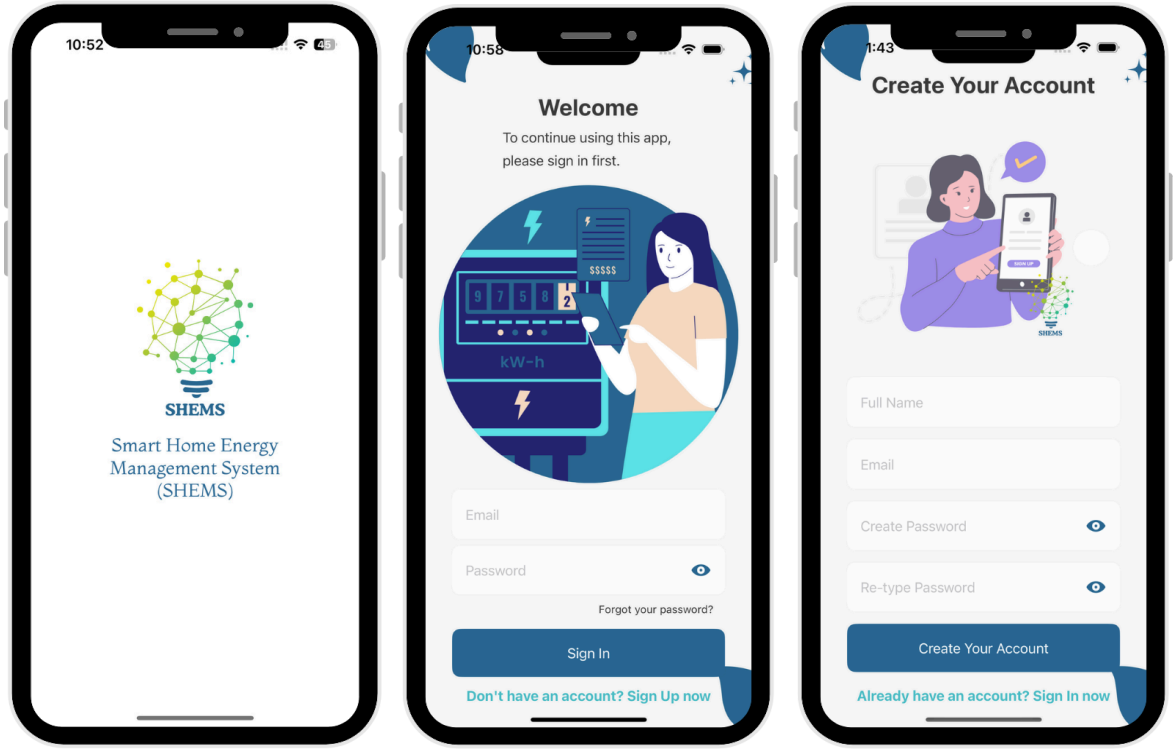


Figure 23. User Sign-In and Registration Screens

The authentication flow includes an email verification step. Upon signing up, users receive a verification email containing a confirmation link as shown in Figure 25. They must verify their email before they can log in. Once verified, the user information is stored in the Firestore as shown in Figure 26.

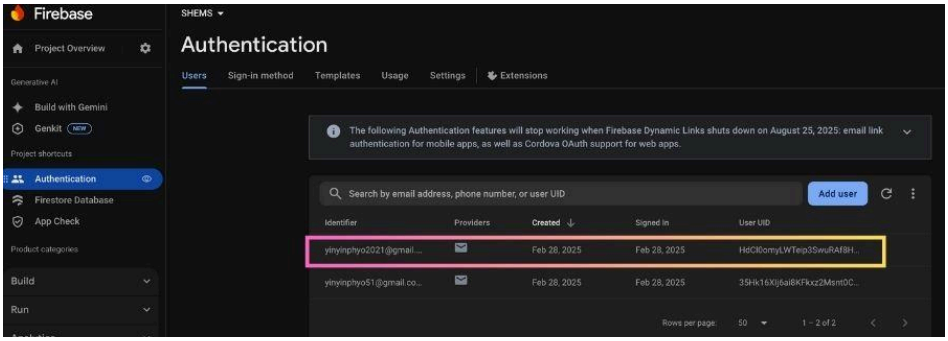


Figure 24. User Authentication data in Firebase

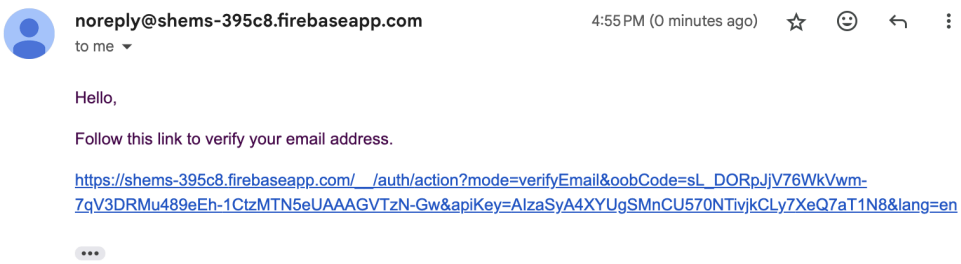


Figure 25. Email Verification Link

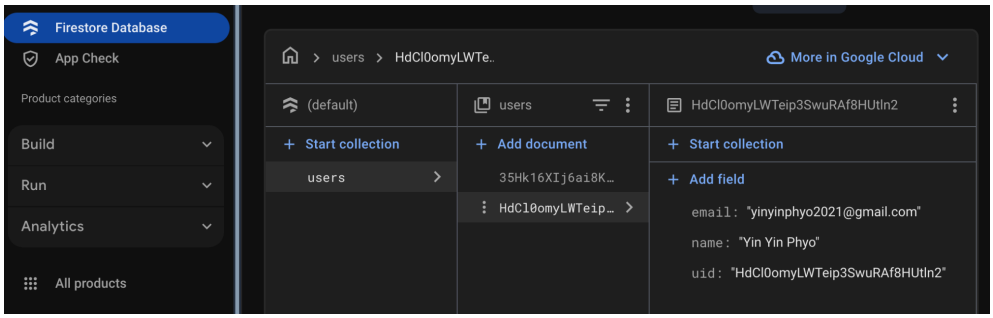


Figure 26. User Information stored in Firestore

```
79    func saveUserData(user: User) {
80
81
82        let userModel = UserModel(uid: user.uid, name: self.name, email: self.email)
83
84        do {
85            try self.db.collection("users").document(user.uid).setData(from: userModel)
86            // After saving data, you may want to close the SignUpView or let the user know
87        } catch {
88            self.errorMessage = "Failed to save user data: \(error.localizedDescription)"
89        }
90    }
```

Figure 27. User Information stored in Firestore (Code Snippet)

5.3 Landing Page

The Landing Page serves as the main interface where users can view all connected appliances and apply filters to refine their selection, including appliance categories including Light and Heater. This page enhances user experience by providing an intuitive and efficient way to manage household appliances, as shown in Figure 28.

UI Elements:

- **Appliance List**: Displays all connected appliances in a structured format, each with an icon, name, status (ON/OFF), and total usage of the current day.

- **Filter Option**: Users can filter appliances based on category, which is the appliance type.
- **Navigation Bar**: Provides quick access to other sections of the application, the User Home Page.

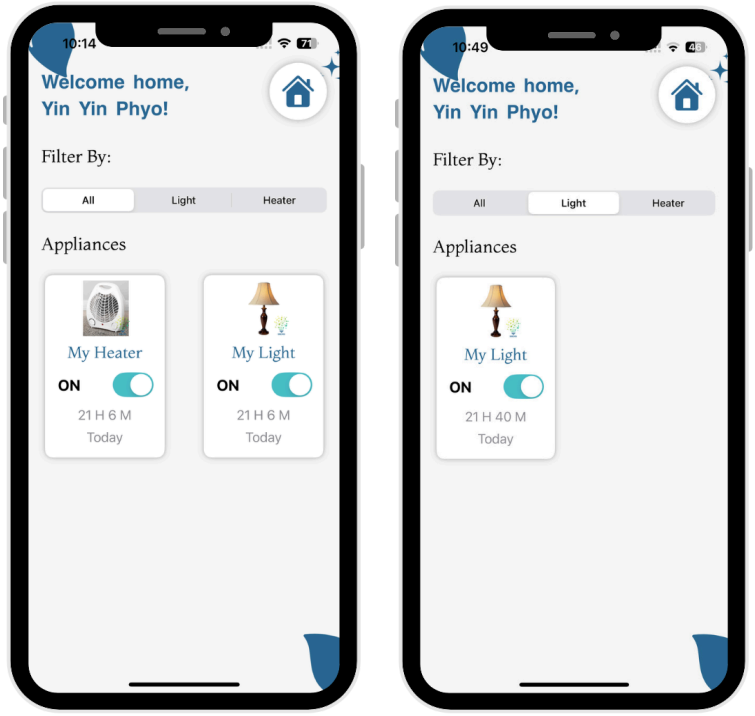


Figure 28. All of the Connected Appliances & Filter Option Result Screen

```
func fetchDevicesAndEnergyData(userId: String) {
    let devicesRef = db.collection("users").document(userId).collection("devices")
    let energyDataRef = db.collection("users").document(userId).collection("energy_data")
        .document(currentDateString()).collection("devices")

    // Fetch devices with energy data initially
    devicesRef.getDocuments { snapshot, error in
        if let error = error {
            self.errorMessage = "Error fetching devices: \(error.localizedDescription)"
            return
        }

        guard let snapshot = snapshot else { return }

        let devices = snapshot.documents.compactMap { doc -> Device? in
            let data = doc.data()
            return Device(
                id: doc.documentID,
                name: data["name"] as? String ?? "Unknown Device",
                image: data["image"] as? String ?? "AppLogo.png",
                category: data["category"] as? String ?? "Other"
            )
        }
```

Figure 29. Getting Appliances Information from Firestore (Code Snippet)

Filter Functionality:

Users can apply filters to narrow down appliances based on specific criteria. The filtering is implemented as shown in Figure 30.

```
Text("Filter By:")
    .font(.body)
    .padding(.horizontal)

Picker("Filter By", selection: $selectedFilter) {
    Text("All").tag("all")
    Text("Light").tag("light")
    Text("Heater").tag("heater")

}
.pickerStyle(SegmentedPickerStyle())
.padding()

Text("Appliances")
    .font(.headline)
    .padding(.horizontal)

ScrollView {
    LazyVGrid(columns: [GridItem(.flexible()), GridItem(.flexible())], spacing: 16) {
        ForEach(viewModel.devices.filter { selectedFilter == "all" || $0.device.category == selectedFilter }, id: \.id) {
            deviceWithEnergy in
            DeviceCardView(deviceWithEnergy: Binding(
                get: { deviceWithEnergy },
                set: { newValue in

                }
            ))
        }
    }
    .padding()
}
```

Figure 30. Filter Picker View (Code Snippet)

Custom Font Integration:

To enhance the visual aesthetics and align with the app's branding, custom fonts were integrated into the application. Specifically, **Montaga-Regular.ttf** and **LilitaOne-Regular.ttf** were added to the project and properly registered in the Info.plist file under the Fonts as shown in Figure 31. These fonts are now utilized throughout the user interface to provide a distinct and consistent typographic style.

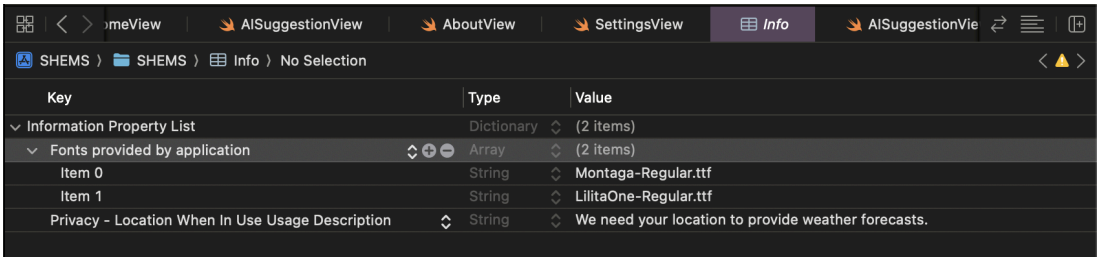


Figure 31. Custom Font Integration

5.4 Detail Appliance Page

The Detail Appliance Page, as shown in Figure 32, provides comprehensive information about a selected appliance, including its current status, energy consumption, and usage history.

UI Elements:

- **Appliance Image & Name:** Displays the appliance's image and name at the top.
- **Status Toggle:** Users can turn the appliance ON or OFF.
- **Energy Consumption Graph:** A visual representation of the appliance's energy consumption over time and historical data on when the appliance was used.
- **Usage Cost:** Displays the estimated monthly cost based on energy usage.

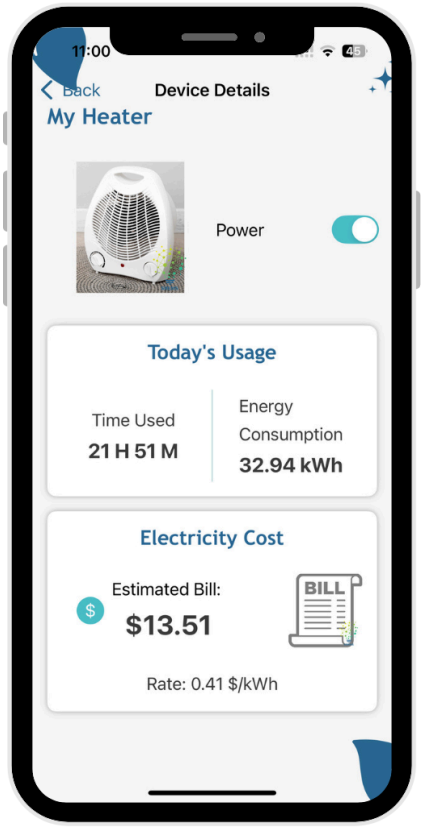


Figure 32. Appliance Detail Screen

```swift
import SwiftUI

struct ApplianceDetailView: View {
    @StateObject private var viewModel = ApplianceDetailViewModel()
    let deviceId: String

    var body: some View {
        ZStack {
            Image("BG")
                .resizable()
                .scaledToFill()
                .edgesIgnoringSafeArea(.all)

            VStack(alignment: .leading, spacing: 16) {
                if viewModel.isLoading {
                    // Show loading indicator while data is being fetched
                    ProgressView("Loading...")
                        .progressViewStyle(CircularProgressViewStyle())
                        .padding()
                } else if let device = viewModel.device {
                    // Title
                    Text(device.device.name)
                        .font(.title)
                        .fontWeight(.bold)
                        .foregroundStyle(AppColors.primaryColor)
                        .padding(.top)

                    HStack {
                        Image(device.device.image)
                            .resizable()
                            .scaledToFit()
                            .frame(height: 150)
                            .padding()

                        if let energyData = device.energyData {
                            Toggle("Power", isOn: Binding(
                                get: { energyData.isOn },
                                set: { newValue in
                                    // Unwrap device.id and update Firestore
                                    if let deviceId = device.device.id {
```

Figure 33. Appliance Detail View (Code Snippet)

5.5 User Home Page

The User Home Page, as shown in Figure 34, acts as a central hub where users can get detailed information about their energy consumption with visualized charts, AI suggestions, and the mobile application.

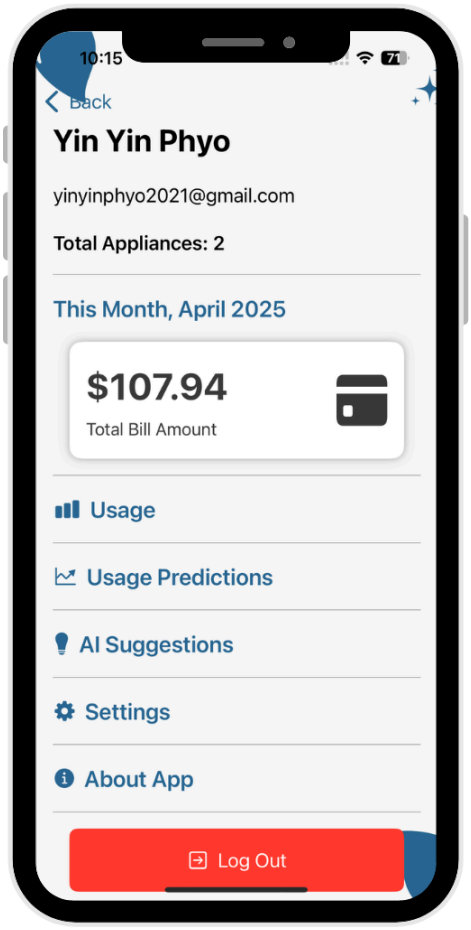


Figure 34. User Home Screen

```swift
import SwiftUI
import FirebaseAuth

struct HomeView: View {
    @EnvironmentObject var appState: AppState  // Access the AppState environment object
    @StateObject private var viewModel = HomeViewModel()

        @State private var newName: String = ""
        @State private var isEditing: Bool = false
    @State private var navigateToUsage = false

    var body: some View {
        ZStack {
            Image("BG")
                .resizable()
                .scaledToFill()
                .edgesIgnoringSafeArea(.all)
            VStack(alignment: .leading, spacing: 16) {

                        // Header Section
                        Text(viewModel.userName)
                .font(.title)
                            .fontWeight(.bold)
                            .foregroundColor(AppColors.textColor)
                            .padding(.top)


                        Text(viewModel.userEmail)
                            .font(.body)
                            .foregroundColor(AppColors.textColor)

                        // Total Appliances
                        Text("Total Appliances: \(viewModel.totalAppliances)")
                            .font(.headline)


                Divider().background(AppColors.textColor)

                            // Current Month Title
                            Text("This Month, \(currentMonthYear())")
                                .font(.title3)
                                .fontWeight(.semibold)
                                .foregroundStyle(AppColors.primaryColor)

                            // Bill Amount Card
                            VStack {
                                HStack {
                                    VStack(alignment: .leading) {
                                        Text("$\(String(format: "%.2f", viewModel.totalBillAmount))")
                                            .font(.largeTitle)
                                            .fontWeight(.bold)
                                            .foregroundStyle(AppColors.textColor)
```

Figure 35. User Home View (Code Snippet)

5.6 Energy Usage Page

The Energy Usage page provides users with a detailed visualization of their energy consumption, allowing them to switch between daily and hourly views. Users can select a specific date or week to view corresponding energy usage data in bar chart format, helping them monitor patterns and identify peak usage periods. It also displays the total energy consumed and the estimated cost

based on the current default rate ($0.41/kWh), giving users a clear summary of their energy expenses for the selected period.

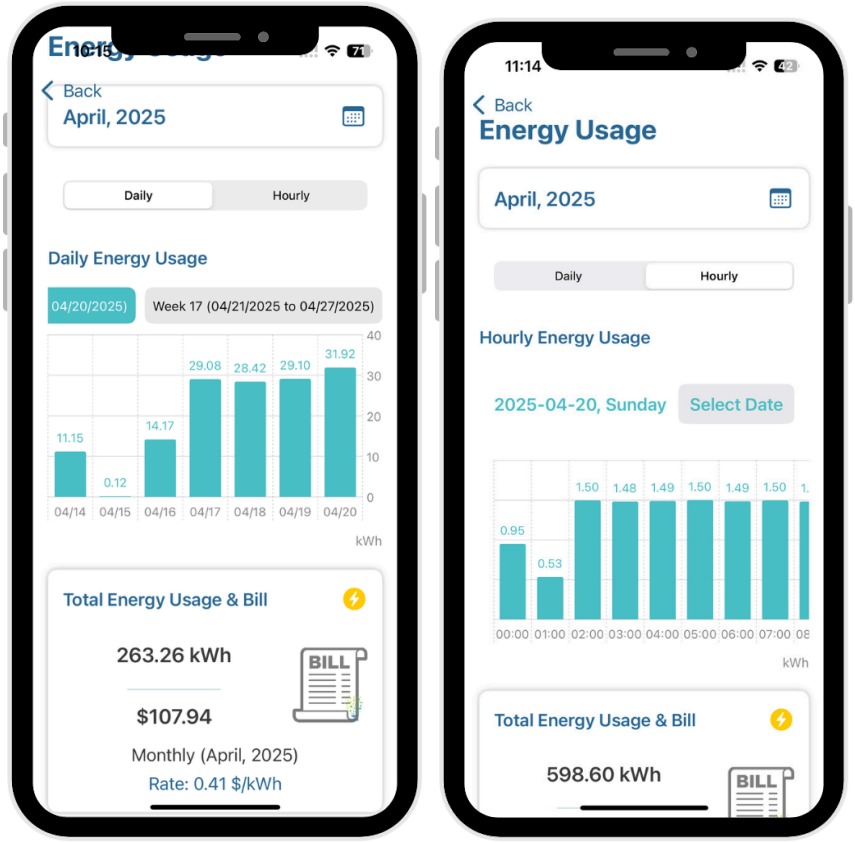


Figure 36. Energy Usage Screen (Daily and Hourly)

5.7 Energy Usage Prediction Page

The Energy Usage Prediction page enables users to compare actual and predicted energy consumption in both daily and hourly formats. Visualized through bar and line charts, the page displays side-by-side comparisons of real-time and forecasted usage (in kWh) along with associated energy costs. This helps users better understand consumption trends and anticipate future usage. By analyzing prediction accuracy, users can make informed decisions on energy-saving strategies and budgeting. It also supports date selection for detailed inspection of past and projected data.

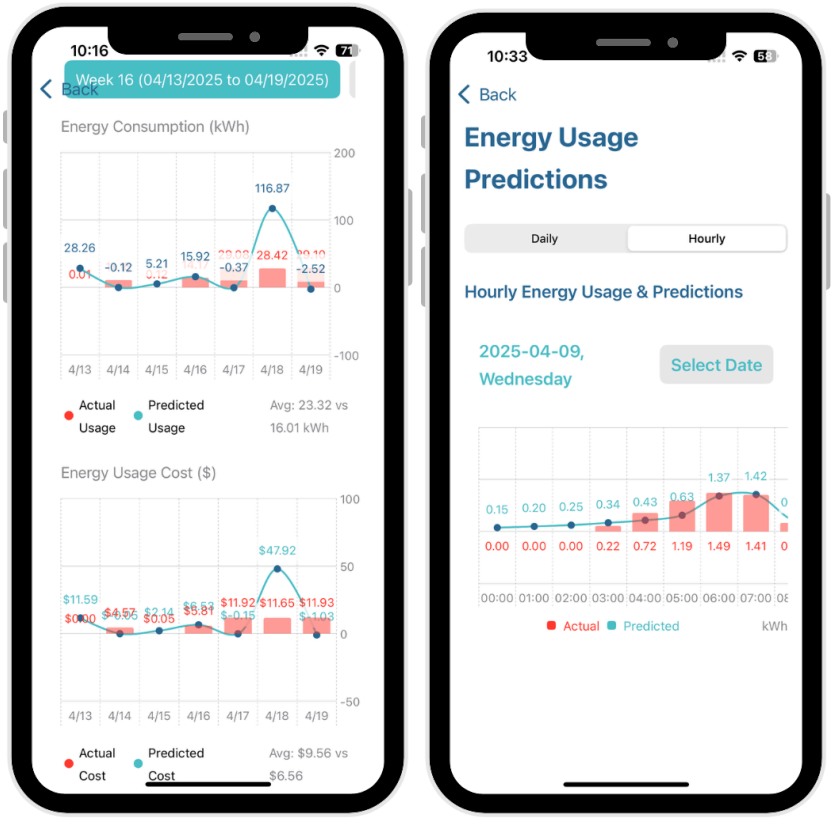


Figure 37. Energy Usage Prediction Screen (Daily and Hourly)

5.8 AI Suggestions Page

The AI Suggestions page provides users with personalized, date-specific energy-saving tips generated by an AI agent. Based on environmental data such as weather forecasts and temperature trends, the AI offers practical advice—such as utilizing natural daylight or adjusting thermostat settings—to help reduce energy consumption without compromising comfort. Users can select a date to view relevant suggestions and make informed decisions to improve energy efficiency in their homes.

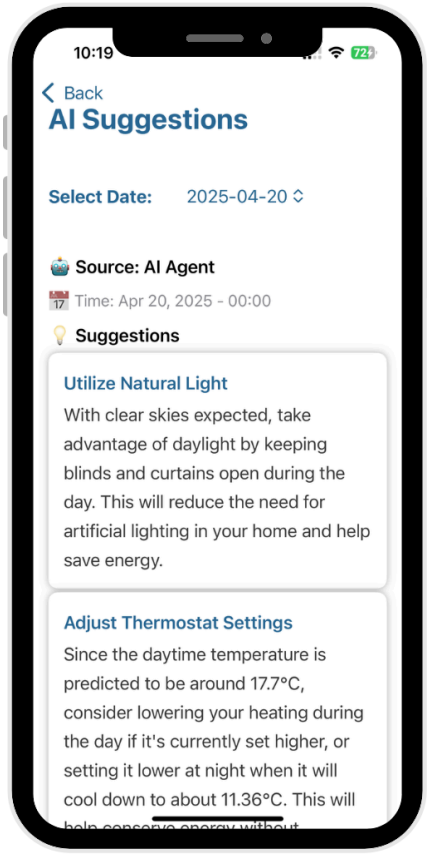


Figure 38. AI Suggestions

5.9 Settings Page

The Settings page allows users to personalize their app experience by configuring preferences such as energy rate usage, theme appearance, and notification controls. Users have the option to enable or disable the default energy rate ($0.41/kWh) and input a custom rate if desired. This flexibility ensures accurate billing estimates based on each user's electricity cost.

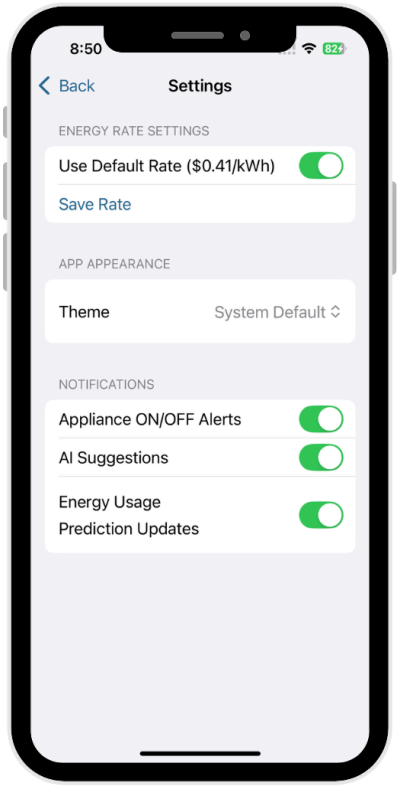


Figure 39. Settings Page

Notification function:

The application provides notification settings that users can toggle based on their preferences. These include Appliance ON/OFF Alerts, AI Suggestions, and Energy Usage Prediction Updates. When users first launch the app, they are prompted to allow notifications. Once permitted, the system delivers real-time alerts—such as when a device is turned off automatically—and proactive suggestions to help users optimize energy consumption.

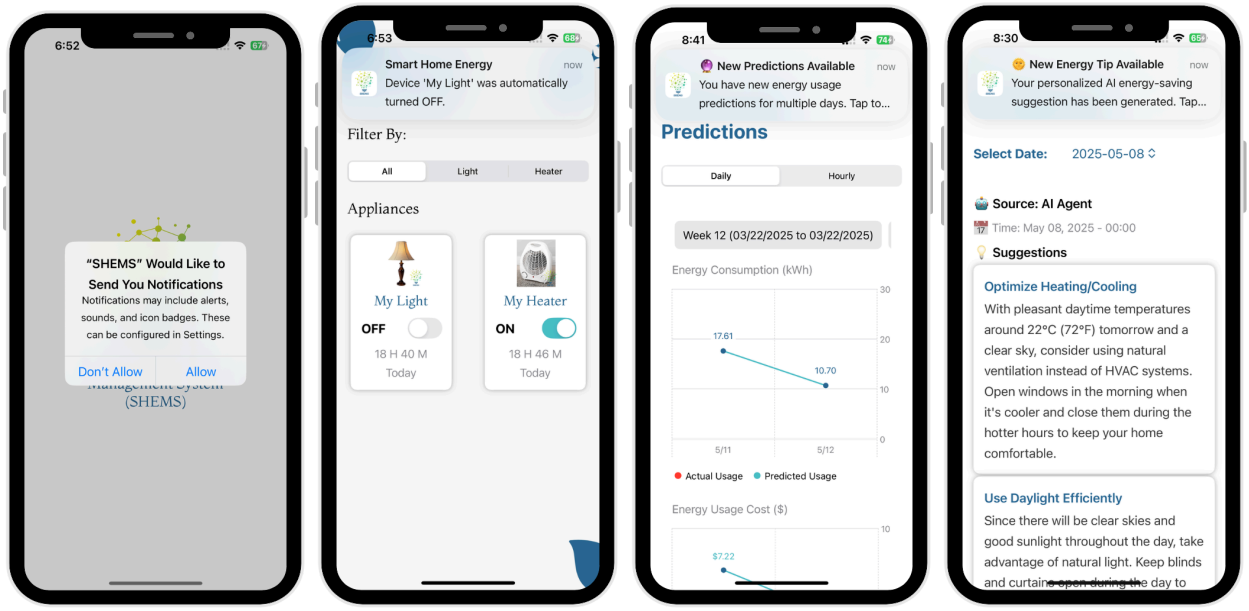


Figure 40. Notifications

Theme Customization Options:

To improve usability and accommodate user preferences, the app offers theme customization under the "App Appearance" section. Users can switch between System Default, Light, or Dark modes.

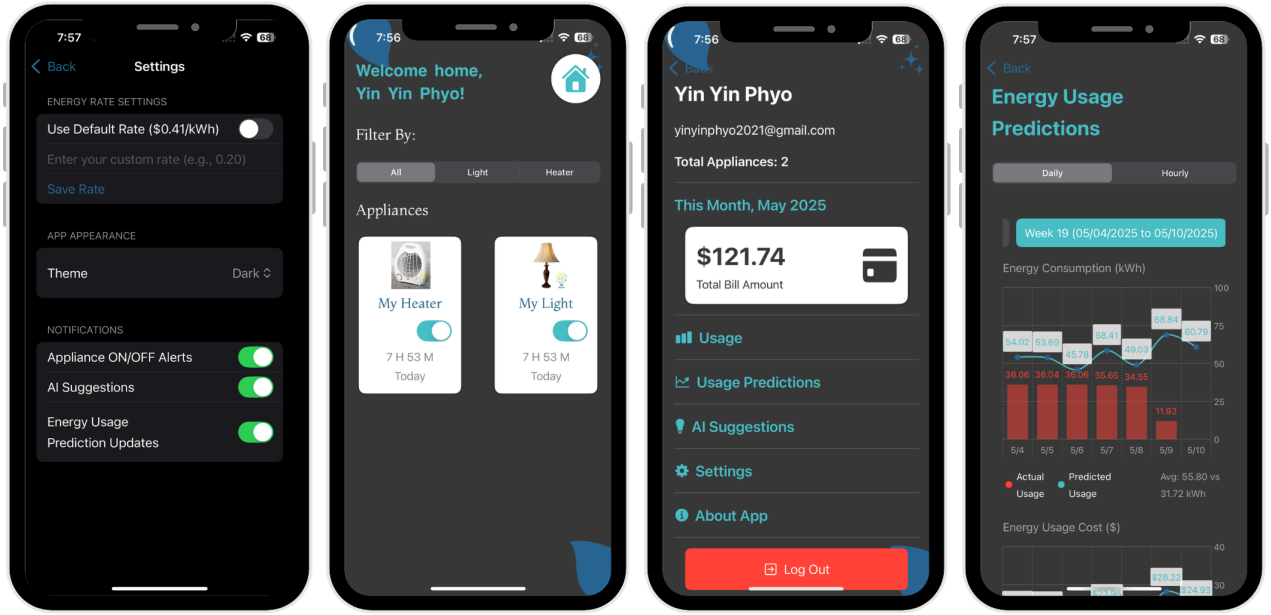


Figure 41. Some pages with Dark Theme

6.0 About App Page

The About App page summarizes the app's mission to make smart energy management simple and eco-friendly. It highlights that user location data is used only during app usage for energy optimization and is never shared. The terms of use encourage responsible behavior and prohibit unauthorized access to smart devices.

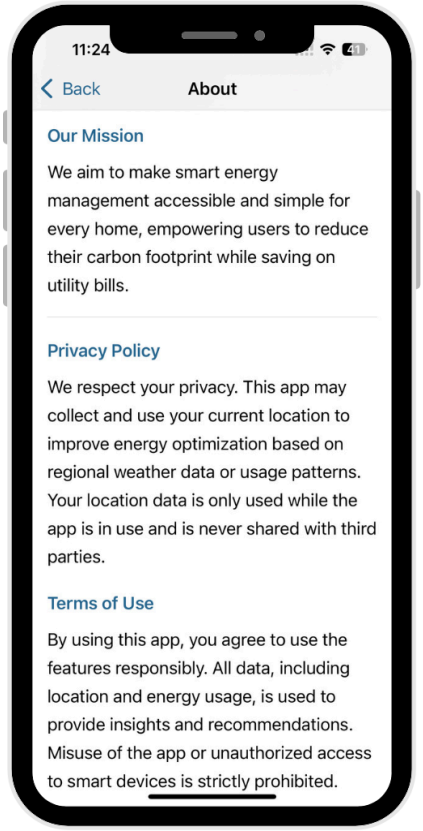


Figure 42. About App Screen

Conclusion

The final implementation of the Smart Home Energy Management System (SHEMS) presents a fully functional and scalable platform for intelligent residential energy management. By combining real-time sensor-based automation, cloud-hosted forecasting, and mobile user interaction, SHEMS offers a seamless integration of hardware, software, and AI technologies.

The system's hardware layer is anchored by a Raspberry Pi connected to motion and temperature sensors, enabling automated control of smart plugs based on environmental data. This setup allows appliances such as lights, heaters, and fans to respond dynamically to occupancy and temperature, reducing unnecessary energy use without requiring manual intervention.

On the backend, a fine-tuned Prophet model generates daily energy forecasts for each household, while an AI agent provides personalized energy-saving suggestions based on forecast data, user history, and weather conditions. These services are deployed as containerized microservices on Google Cloud Run and automated via Cloud Scheduler, with secure API management and monitoring tools ensuring stability and maintainability.

The Swift-based iOS app completes the system by offering users real-time device control, custom automation rules, personalized cost estimates, and intelligent alerts. Features like user-defined electricity rates and push notifications enhance engagement and allow for fully customized energy oversight.

SHEMS demonstrates the power of full-stack integration—from edge devices to AI-driven cloud intelligence—to support sustainable, cost-efficient smart living. The project showcases a robust and extensible architecture that is ready for real-world application and future feature expansion.

References

Adafruit Industries. (n.d.). *DHT11 Temperature and Humidity Sensor*. Retrieved from https://www.adafruit.com/product/386

Banafa, A. (2019). *Secure and Smart Internet of Things (IoT): Using Blockchain and AI*. River Publishers.

Cron Job Scheduling. (n.d.). *Crontab Manual Page*. Retrieved from https://man7.org/linux/man-pages/man5/crontab.5.html

Firebase. (n.d.). *Cloud Firestore Documentation*. Google Developers. Retrieved from https://firebase.google.com/docs/firestore

GPIO Zero Library. (n.d.). *Controlling GPIO in Python*. Raspberry Pi Foundation. Retrieved from https://gpiozero.readthedocs.io/en/stable/

Hebrail, G. & Berard, A. (2006). Individual Household Electric Power Consumption [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C58K54

Raspberry Pi Foundation. (n.d.). *Getting Started with Raspberry Pi*. Retrieved from https://www.raspberrypi.org/documentation

TMUX. (n.d.). *Tmux Terminal Multiplexer*. Retrieved from https://github.com/tmux/tmux/wiki