SmartWatt

Anya Bindra, Erika Ramirez, Maya Doshi

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—This project develops a household energy optimization system that monitors energy appliance usage and reduces power consumption costs through efficient dynamic scheduling of devices. Using linear programming and machine learning, the system schedules deferrable loads and balances power drawn from solar and grid sources in a cost optimal manner. The system automates appliance control switching based on the generated schedule. A web-based UI dashboard provides real-time data and user controls. The frontend displays appliance usage insights, including consumption trends, solar power contribution, and recommendations on optimal on/off times. It also displays scheduled appliance operations outputted by optimization model**

***Index Terms*—Backend, Frontend, Database, Energy, ESP32, Forecasting, HA (Home Assistant), Loads, Model Predictive Control, Optimization, Power, Raspberry Pi, Solar Panel, MQTT**

I. Introduction

The number of energy-consuming devices has surged in recent years. With the increasing complexity of smart appliances, electric vehicles, temperature control systems, fluctuating grid prices, and renewable energy sources like solar and wind, managing power consumption efficiently has become challenging for individuals to handle manually [10]. Moreover, the rise in power consumption has not only led to higher utility costs for individuals but has also placed greater stress on aging power grids, making them more vulnerable to failures caused by sudden surges in demand [11]. To address these challenges, optimizing home energy usage has become essential. By leveraging machine learning and optimization algorithms, we forecast power generation, grid prices, and consumption patterns, enabling intelligent appliance scheduling that maximizes renewable energy usage while minimizing costs. SmartWatt provides personalized recommendations, empowering users to make informed energy decisions. By synchronizing energy consumption with renewable power generation, we significantly reduce dependence on non-renewable energy sources, contributing to a more sustainable future.

This represents a significant advancement over existing home automation systems such as Ecobee [15] and Enphase Energy [16], which primarily rely on manually programmed, rule-based automations. These traditional systems lack adaptability, as they do not dynamically respond to real-time conditions such as fluctuating grid prices, solar generation, and changing load consumption patterns. Manually configured optimizations are neither as thorough nor effective as an automated system because users are unlikely to consider every possible optimization scenario. The optimization framework integrates with Home Assistant (HA), an open-source home automation platform that enables users to control and automate smart home devices. Home Assistant is a centralized hub for managing IoT-enabled appliances, energy monitoring, and automation rules. By leveraging Home Assistant's flexible architecture, SmartWatt serves as an energy management add-on, providing forecasting and optimization capabilities to users who already utilize Home Assistant for smart home automation.

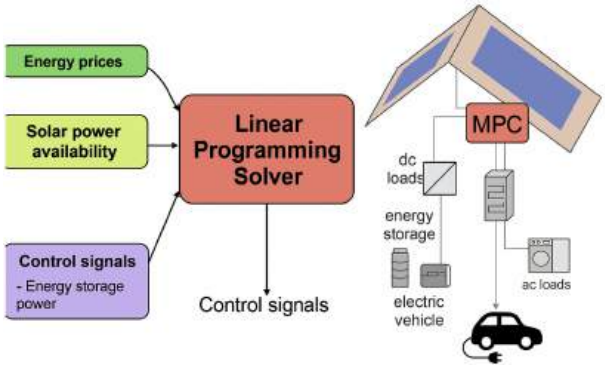


Fig.1:Graphical representation of the optimization system

II. Use-Case Requirements

A. Reduction in Electricity Costs

The goal is to achieve a minimum **10%** reduction in electricity costs within a household (compared to the baseline costs)

By predicting the solar power generation, we schedule high-consumption activities during peak solar production times, thereby reducing reliance on grid electricity. We use weather forecasts from Solcast and OpenMeteo. Studies on photovoltaic energy forecasting report MAPE values ranging from 15% to 25% for short-term forecasts (0-24 hours) using advanced machine learning and numerical weather prediction models [17]. The target we set is **80%** accuracy for predicting the solar power output for the next 24h period. We aim for at least **75%** of the solar power to be consumed onsite [we refer to this as maximizing self-consumption later]. One mechanism of enforcing this is to have about **30%** of high power deferrable loads being scheduled during peak solar production times [as a constraint in our optimization algorithm]

Understanding daily fluctuations in electricity prices enables the scheduling of deferrable loads during off-peak, lower-cost periods. We aim to have < **25% RMSE** for grid pricing forecasts. A sub-25% MAPE ensures that our optimization algorithms can reliably shift energy consumption to lower-cost periods, minimizing electricity expenses. Further, we can enforce a reduction of electricity prices by having about **20%** of loads shifted to a low cost time-of-use period as a constraint.

Initially, we planned to incorporate household load forecasting into SmartWatt. Accurate demand prediction would have

allowed us to better align energy consumption with available power resources. However, as the project progressed, we decided not to pursue load forecasting for the capstone. The main reason was that forecasting these loads requires large amounts of historical data and introduces substantial uncertainty and variability in the output. Additionally, load usage patterns can differ drastically from household to household. Instead, we chose to focus on optimizing controllable loads and solar usage using user-defined schedules, which aligned better with user preferences.

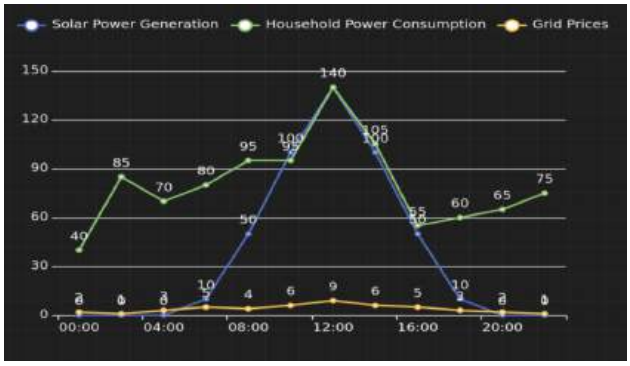


Figure 2: Expected Output of SmartWatt, aligning consumption to periods of high solar power output and low time of use pricing

B. *User Interaction and Recommendations*

The dashboard should feature two key sections: a live log of automated actions and a table of suggested actions for user intervention. The live log displays a real-time feed of all the actions the system has executed automatically, including details such as the timestamp, action taken [appliance switched on/off], estimated energy savings (kWh), and completion status. This ensures transparency, allowing users to track how the system optimizes energy usage without manual input. The dashboard contains a suggested actions table for tasks that the system is unable to complete on its own. This section should list recommendations such as rescheduling some appliances to low time-of-use periods , along with an estimate of the potential energy savings. By combining these features, the dashboard provides users with both insight into automated energy optimizations and actionable guidance to further reduce electricity consumption

C. *System Responsiveness*

SmartWatt ensures efficient real-time power monitoring of appliances. Device sensors connected to the ESP32 sample power consumption data at 1 Hz. These readings are then averaged over a 5-minute window to filter out noise and provide a stable, reliable measure of energy consumption. The dashboard should be designed to update every 5 minutes, reflecting the most recent power consumption trends while ensuring minimal computational load on the system.

To provide a good user experience, the response latency should be ≤1.5 seconds, ensuring that whenever a user interacts with the dashboard—such as refreshing data, viewing

D. *Compute Requirements*

Since we are optimizing home power consumption, we do not want the device running the optimization algorithms to be a significant contributor to the power bill. The most common device to run Home Assistant on is a Raspberry Pi. Home Assistant is pretty resource light so we should be able to run SmartWatt and all the optimization and forecasting models on the same device as Home Assistant. The Raspberry Pi currently consumes 2.7 to 3.5W on idle (which it will be running at most of the time) and up to 10-15 watts when fully loaded, so it is the ideal computer for a task like this.

E. *Model House Requirements*

We construct a model house that replicates real-world residential power consumption patterns while ensuring safety and efficiency. To achieve this, we use DC appliances instead of AC appliances due to safety considerations [also aligns with modern renewable energy solutions such as solar power and battery storage]. The system simulates realistic load and power consumption patterns by scaling all appliance power consumption values by a specific factor (which ended up being ~1/1000) determined on a per-appliance basis, to ensure proportional energy usage across different household devices.

To effectively demonstrate the real-time actions taken by SmartWatt, we scale simulated days down to 5-minute intervals. This accelerated timeline allows us to observe and analyze the system's decision-making, automated optimizations, and user recommendations in a more compact and interactive manner. By implementing this approach, SmartWatt provides a realistic yet controlled environment for testing home energy management strategies, optimizing energy usage, and validating the effectiveness of automated energy-saving interventions.

III. Architecture and/or Principle of Operation

SmartWatt has a machine learning and optimization subsystem, software and database backend, as well as an embedded control layer which are interconnected and work to predict, monitor and control energy usage within the household. We have data sources which collects real-time data

- Power Sensors: Measure the current output of photovoltaic panels, power consumption of loads
- Weather Data API: Retrieves local weather forecasts affecting solar production.
- Grid Price API: Accesses real-time electricity pricing from utility providers

1. A forecasting module, utilizing collected data to predict future energy distribution 2. An optimization module: Processes forecasts to develop schedules for running appliances. 3. A user interface dashboard which provides homeowners with insights and control over their energy usage. 4. Microcontroller unit which facilitates appliance switching and control, polling sensors and actuation of resistive loads

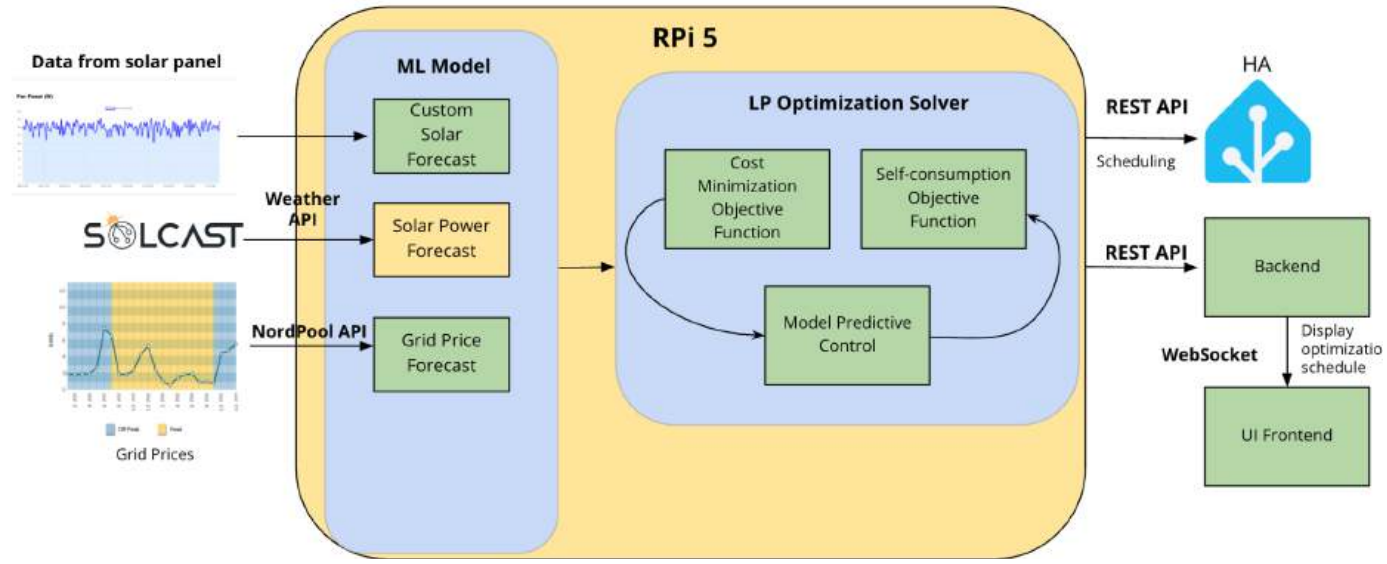


Figure 3: Machine Learning and LP Solver Subsystem

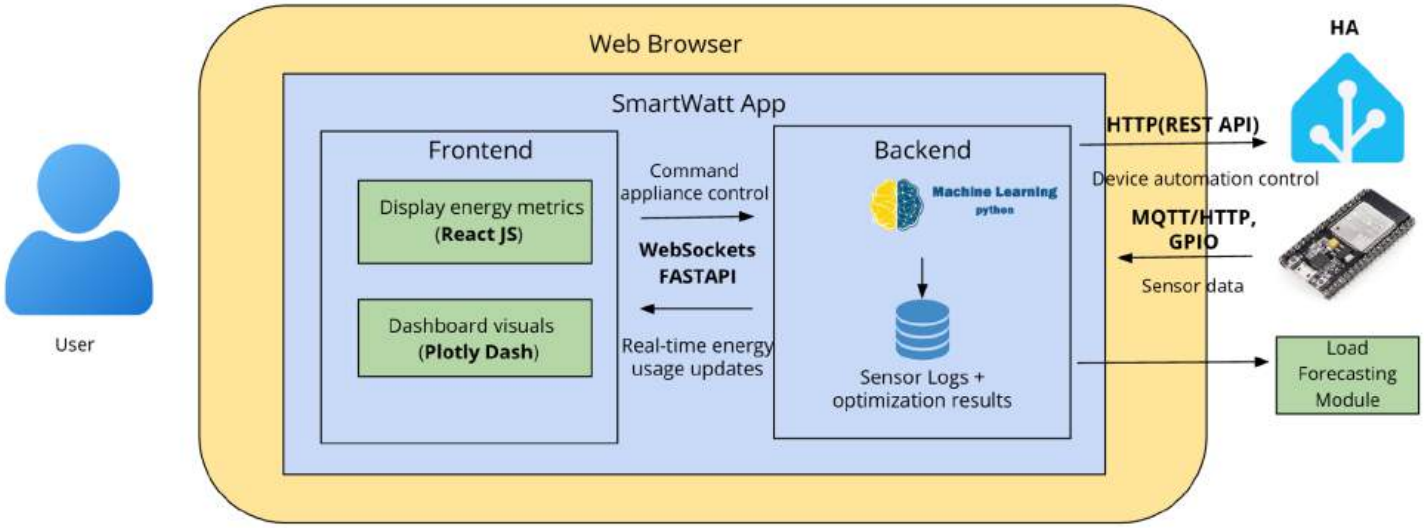
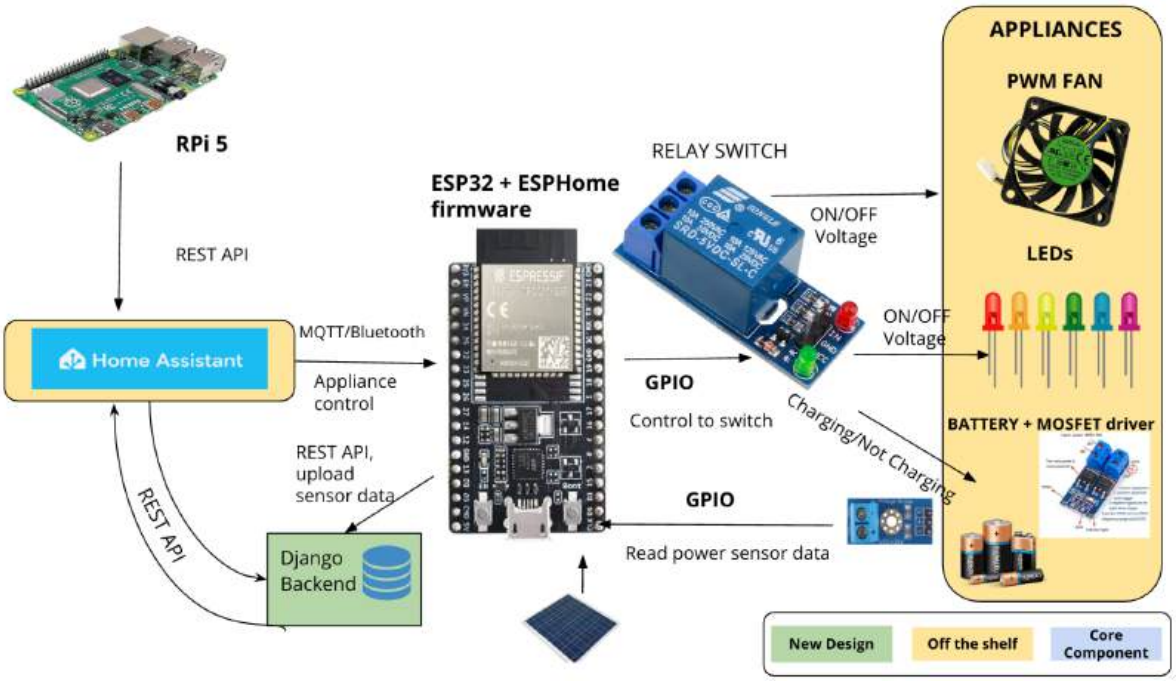


Figure 4: Software, Frontend and Backend Subsystem, Embedded Subsystem for Appliance Control



The main difference between the block diagrams above and in the design phase was that we removed the load forecasting module and added a custom Solar Forecast that is trained on power data we obtained from our own solar panels. Solcast provides generalized forecasts based on satellite data and broad weather models, which we found did not capture the shading patterns specific to our rooftop setup. So we obtained historical power data from our solar panels and chose to train our XGBoost models using that.

We use the ESPHome Firmware which simplifies the configuration and integration of the ESP32 in the embedded module with the sensors and actuator (motor, LED). A detailed control flow diagram listing communication protocols between the different subsystems (end-to-end) is included in the implementation section.

Here is the operational workflow:

User Input: Homeowners interact with the system via the React Typescript application, with a frontend dashboard displaying energy metrics and the backend processing results from the optimization algorithm.

Data Collection: The ESP32 microcontroller gathers real-time data from sensors and transmits it to the Raspberry Pi which is then transferred to the backend.

Data Analysis and Forecasting: The sensor data, grid price data and weather data is fed to the forecaster. The forecaster outputs what the next 24h solar power output and grid prices are, and these are used by the optimization algorithm.

Optimization Execution**:** Based on the forecasts, the system uses linear programming and MPC to generate a schedule for running appliances. Recommendations are relayed back to the user via the display monitor.

Appliance Switching : The backend communicates the output of optimization in the form of commands to the ESP32 via MQTT/HTTP and then the ESP controls the appliances (LED, Motor etc) via GPIO by sending switch ON/OFF commands.

A. Engineering Principles

We applied modular engineering design principles by breaking SmartWatt into independently developed components: solar forecasting, load scheduling, optimization, and device control. This streamlined integration and testing. We used embedded systems principles in working with the ESP32 microcontroller, managing GPIO and power constraints. Real-time control and system optimization required integrating both hardware and software with user-defined preferences.

B. Scientific Principles

We used the **conservation of energy** principle. In our linear programming formulation, we enforce a constraint that ensures energy consumed by devices at any time does not exceed the sum of solar generation and grid availability. This reflects the fundamental physical law that energy cannot be created or destroyed—only transferred. By modeling net energy flow in each time slot, the optimizer schedules loads in a way that respects real-world energy balance, preventing unrealistic behavior like consuming more energy than is available from the combined sources. Additionally, our solar forecasting relied on atmospheric science and the photoelectric effect, particularly how cloud cover, humidity, and irradiance influence photovoltaic output. These features were mapped from real-world weather data to improve forecast accuracy, informing our scheduling decisions.

C. Mathematical Principles

For solar prediction, we used XGBoost regression, which relies on minimizing a regularized loss function through gradient boosting. This involves principles from statistics and numerical optimization (gradient descent). Our scheduling system is formulated as a binary linear program. The objective function we formulate for our scheduler minimizes total energy cost while satisfying constraints on device duration and energy availability.

IV. Design Requirements

A. Energy Optimization Requirements

One of the primary goals of SmartWatt is to ensure that at least **75%** of the generated solar energy is consumed on-site. To achieve this, the system will actively schedule > **30%** of deferrable loads during peak solar generation hours. The solar energy utilization efficiency is constrained by the ratio of **onsite consumption to total generated power**:

$$\eta_{solar} = \frac{P_{onsite}}{P_{total}} \geq 0.75$$

To achieve at least a **10**% reduction in electricity cost compared to a baseline consumption pattern, the algorithm should output a schedule such that > **20%** of deferrable loads shift their consumption to lower-cost Time-of-Use periods. Grid electricity price forecasting will be used to optimize the load shifting, with a RMSE of <25%.

$$\eta_{shift} = \frac{E_{shifted}}{E_{total}} \geq 0.2$$

The convergence time of the linear optimization algorithm must be within **20s**, with a sub-optimality gap of <10%

(difference between true optima and value returned by solver). To enforce this, an iteration limit will specify the maximum number of iterations the solver will perform. Each iteration involves processing constraints and variables to move closer to an optimal solution. Setting this limit helps prevent excessive computation time on particularly challenging optimization problems.

B. Data Accuracy

SmartWatt uses forecasts from Solcast API and Open-Meteo to provide real-time solar irradiance data, which will be used to estimate solar power generation with a MAPE accuracy of > **80**%. Open-Meteo API will supply weather forecasts, including cloud cover, temperature, and solar radiation levels, to refine solar power generation predictions.

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{P_{pred,i} - P_{actual,i}}{P_{actual,i}}\right| \times 100 \leq 20\%$$

Accurate measurement of power consumption across loads is critical to the optimization and control of energy usage. The **INA226** power sensor must measure current and power within 98% accuracy. According to the datasheet [8], measurement error is within **±0.1% to ±0.5%**.

C. System Monitoring and Responsiveness

To provide users accurate and timely insights about which appliances are consuming the most power, SmartWatt features a dashboard with real-time power consumption updates. The dashboard must update power consumption data **every 5 minutes.** To achieve this, the sensors will sample data to ESP32 at a frequency of **1 Hz**, with an averaging window of **5 minutes** before being sent to the backend.

A responsive and low-latency dashboard enhances the user experience. The system must achieve a dashboard response latency of **≤3s** when loading real-time data. Here is the end-to-end latency breakdown:

- ESP32 ↔ Django Backend: Network latency **≤100 ms**, request processing **≤200 ms**.
- Backend ↔ Database: Query execution time **≤500** ms
- Backend ↔ Frontend: API response **≤200 ms**, JSON parsing **≤100 ms**

D. Load Simulation and Hardware Performance

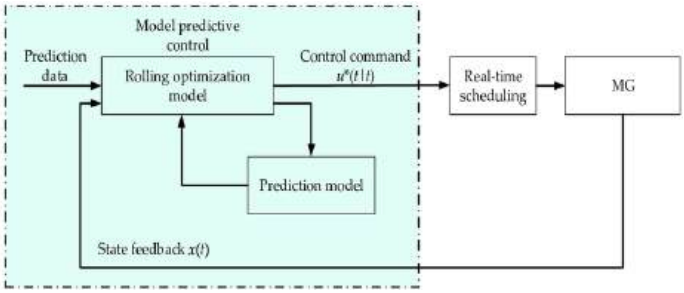
To accurately replicate household consumption behavior in a scaled prototype, the system must operate within realistic power ranges. The ESP32 microcontroller is capable of operating within a **3.3V-12V range**, drawing up to **600mA** current as needed. This falls in the range of 0.1W to 5W. In the real world, most appliances operate between **0.1 kW to 5 kW**, so we will use a scaling factor of 1/1000 to simulate loads. 5, 1W solar panels will be used for power generation (which can be connected in series to provide up to 5W). Internal resistive elements (**100mΩ - 10Ω**) such as LEDs and PWM fans will be used to simulate power draw.

During testing, we experimented with 1.5–3V 15,000 RPM DC motors to simulate variable appliance loads. However, we found that these motors were power-hungry and did not adhere to the expected wattage constraints for our scaled model. Despite their low voltage ratings, they drew high instantaneous currents - exceeding the safe operating range of our ESP32 microcontroller and bypassing the intended 0.1W–5W power envelope that was designed to mimic household appliances at a 1:1000 scale.

V. Design Trade Studies

A. Optimization Algorithms for Scheduling Loads

The optimization module schedules deferrable loads throughout the day and balances solar and grid power. Our approach uses linear programming (LP) and model predictive control (MPC). Decision variables record each time step and load (load_on[t, load] = 1 if on, 0 if off), continuous variables for grid power (positive for consumption, negative for injection), and battery charge/discharge powers. MPC uses a model to predict future behavior over a finite horizon. The algorithm has a prediction horizon which looks ahead to account for time-varying factors like energy prices, solar generation, and load demand, rolling optimization that adapts to real-time changes in load consumption patterns, ensuring the schedule remains optimal as conditions evolve. We chose this approach as it is crucial for energy systems where electricity prices may follow time-of-use tariffs, and solar power availability depends on weather and time.



Alternative 1 : Stochastic Dynamic Programming is an optimization method that models the energy management problem as a Markov Decision Process, where the state variables are current time, battery state of charge, and deferrable load status. Decision variables are actions such as turning loads on/off or charging/discharging the battery, with transition probabilities accounting for randomness in solar generation, load demand, and energy prices. This approach is not used by us because SDP requires solving a Bellman equation iteratively, which is computationally intensive, especially with continuous state and action

spaces. In our design requirement, we have that the algorithm should converge within **20s**, which is not feasible with SDP. Additionally, SDP relies on well-defined probabilistic models of solar generation, energy prices, and load demand. In real-world scenarios, these distributions are hard to estimate accurately, leading to suboptimal decisions

Alternative 2: Rule Based Programming which consists of predefined logic and thresholds for decision-making. Instead of solving an optimization problem, a rule-based system follows if-then conditions based on real-time sensor inputs. For example, if solar power is greater than a set threshold, then turn on deferrable loads. Or if grid prices exceed a predefined cost, then discharge the battery to supply loads. We did not choose this method because the system only reacts to current conditions, not future forecasts (e.g., tomorrow's energy prices or solar generation). Also the rules are hardcoded, so if a rule turns on appliances whenever solar power is high, it might still do so even if tomorrow's electricity price is cheaper, leading to higher costs, suboptimal.

B. Solar and Grid Price Forecasting Models

We initially considered using LSTM networks for forecasting grid electricity prices due to their ability to capture long-term dependencies and memorize temporal trends in sequential data. LSTMs are well-suited for learning patterns like daily or weekly price cycles and responding to lagged effects. However, we found that LSTMs required extensive training data (more than 3 weeks), were sensitive to tuning, and had long inference times—making them less practical for frequent, lightweight retraining in our real-time pipeline. We pivoted to **XGBoost**, which handled time-based features (lagged prices, hour of day, day of week) effectively. It trained faster, generalized well with less data, and provided more interpretable feature importance, making it a better fit for SmartWatt's retraining and deployment requirements. Here is a trade-offs table describing the pros and cons of different forecasting models

Model	Overfitting Risk	Accuracy	Limitation
XGBoost	Low	Medium-high, Balances feature selection and regularization	High memory usage
ElasticNet (Skforest Autoregressive)	Low	Medium-high, Balances feature selection and regularization	Requires tuning L1 (Lasso), L2 (Ridge) ratios, struggle with nonlinear relationships
LSTM	High	High for capturing long-term	Needs large training datasets, memorizing noise instead of
		dependencies	trends
ARIMA	Moderate	Medium, good for short-term time-series	Cannot capture sudden price/demand fluctuations

Since grid prices and solar power do not exhibit extreme fluctuations, we do not need to train nonlinear models such as neural networks which are computationally intensive.

C. MCU Platform for Embedded Control

The control subsystem automates appliance switching based on the generated optimization schedule, requiring a device with I/O capabilities, communication, and processing for executing commands. ESPHome simplifies device setup and control by allowing us to define device configurations using YAML files which are compiled to firmware and flashed onto the ESP MCU. ESP32 has great peripheral support and works with GPIO, I2C, UART, ADC, and PWM for flexible sensor and actuator control. However, we will be using RPi5 to actually host and run our optimization models. ESP32 has only a dual core , 520 KB of SRAM and limited flash storage which are too small to process compressed ML models efficiently, slowing down Wi-Fi communication, sensor polling. Bi-directional communication of the switching commands between RPi5 and ESP32 done through MQTT/Bluetooth BLE (wireless)

Attributes	ESP32	ESP8266	Arduino MKR Wifi 1010	TI CC3220
Connectivity	WiFi, Bluetooth, native MQTT, API, UART, I2C	No Bluetooth, WiFi, native MQTT, API, UART ,I2C	BLE, WiFi,MQTT, UART, I2C	BLE, WiFi,MQT, UART, I2C
API Support	ESPHome API	ESPHome API	REST/WebSocket	REST/Web Sockets
Processing Power	Dual-core 240 MHz	Single-core 80/160 MHz	32-bit ARM Cortex-M0+	32-bit ARM Cortex-M4
GPIO	34	17	22	27
ESPHome Compatibility	Full Support	Full Support	Custom Firmware (no native support)	Custom Firmware (no native support)
RAM	320 KB SRAM, 4MB Flash	80 KB RAM, 512 KB to 4MB Flash	32 KB SRAM, 256 KB Flash	256 KB SRAM, 1 MB Flash

D. Software Stack (Frontend + Backend)

We chose a FastAPI+ React + Python stack. FastAPI provides low-latency API responses and asynchronous request handling, making it ideal for real-time interactions like scheduling updates, sensor polling and solar forecast queries. Initially, we considered using Django due to its built-in database and ORM, but found it to be heavier and

less suited to building a fast, lightweight API. It also supports real-time data streaming using WebSockets, making it suitable for handling real-time power monitoring data from devices. On the frontend, React was chosen for its component-based architecture, supporting WebSockets and API polling, making it ideal for building dynamic dashboards with real-time data visualization. React enables a highly interactive user experience.

E. Chatbot Interface System

We tested several LLMs for SmartWatt's chatbot, including Meta's LLaMA2 and Google's Gemma, but found their responses to be inconsistent and too verbose, with higher latency and lower relevance in a home energy context. To improve user experience, we added a natural language interface to guide and clarify queries. Ultimately, we chose OpenAI's GPT-4 via API, which offered more reliable, concise, and accurate responses—with roughly 20–40% better accuracy and latency around 750 ms. This ensured smooth, interpretable answers to user questions about energy savings, device schedules, and solar forecasts.

VI. System Implementation

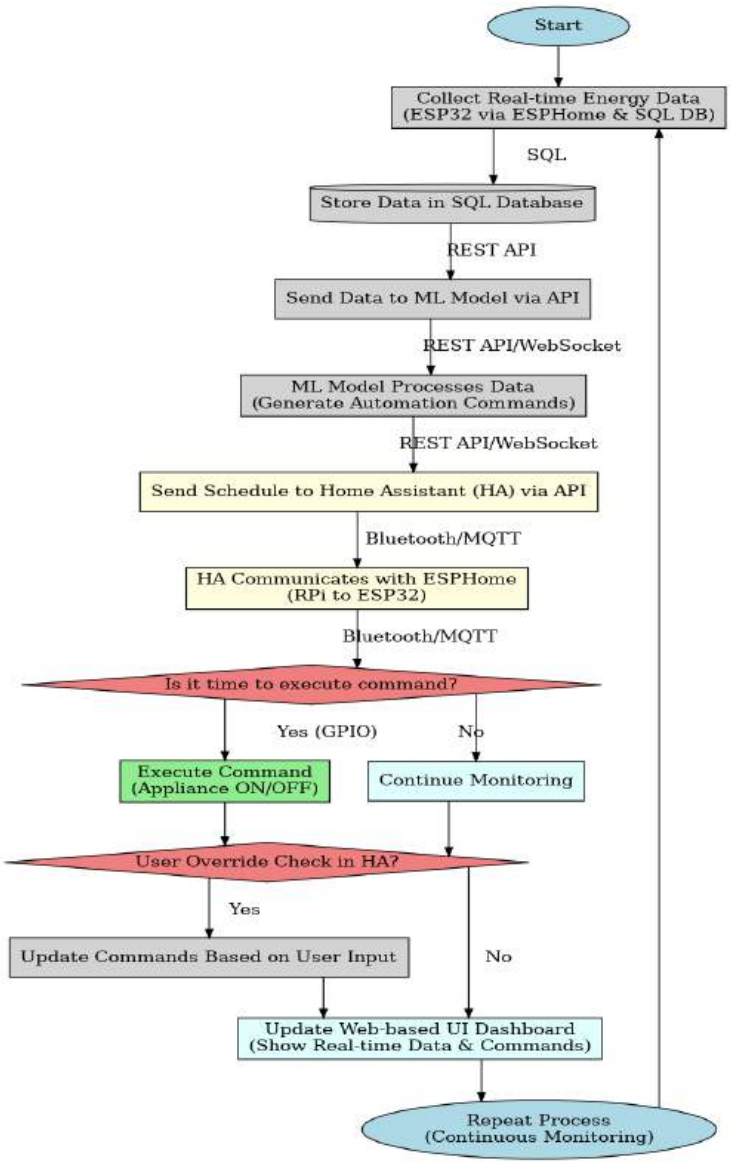


Figure 5: End to End Control Flow Diagram

The flowchart depicts the end-to-end control flow of the entire system. The ESP32 receives appliance schedules and toggles relay switches connected to appliances such as PWM fans, LEDs, battery packs and other DC loads (3- 12 V, 0.1W to 5W, 0.1Ω to 100Ω) via GPIO. We chose lower wattage resistive elements, instead of directly controlling full-scale home appliances that require high-voltage AC power and a wall socket connection. This decision was made based on public safety considerations. High-power inductive loads (compressors, large motors) **c**ause voltage spikes and electromagnetic interference that can damage ESP32 circuits.

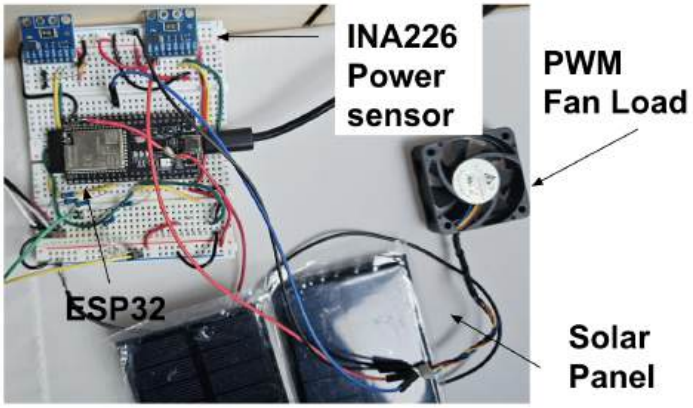


Figure 5 : Device Control setup with ESP32, INA226 power sensors, solar input, and a PWM fan load.

We used relays in the circuit to control the appliances using the low-voltage digital outputs of the ESP32. Since the ESP32 operates at just 3.3 volts, it cannot directly switch devices which require higher voltages and significant current. Relays act as electrically isolated switches, allowing the microcontroller to safely toggle power to these appliances without any direct electrical connection. This provides protection against surges or short circuits, ensuring the ESP32 isn't damaged by power fluctuations.

The core scheduling and optimization logic remains the same whether controlling small DC loads or AC appliances. Once the optimization model is validated, the system can be scaled to real appliances using relay switches, allowing integration with smart plugs and energy meters. We have programmable relay switches which are electronic on/off controllers, enabling the ESP to efficiently manage devices based on load forecasts and energy availability. Additionally, the ESP32 continuously polls sensors such as the INA 226 power sensors over I2C, GPIO, transmitting real-time data back to RPi5 for optimization feedback and updates to the backend. This setup ensures stable appliance switching and low-latency (<=1.5s) communication.

A. *Forecasting Module Implementation*

1. Solar Power Forecasting

- Open-Meteo API [12]: Uses latitude/longitude-based weather forecasts (solar irradiance which is then converted to PV power using following formula)
- Solcast API: Provides PV generation predictions.
- Persistence Model: Updates forecasts with real-time PV production data.

$$\text{Energy output (kWh/hr)} = \text{Solar array area (m}^2\text{)} \times \text{Conversion efficiency} \times \text{Solar radiation for the month (kWh/m}^2\text{/day)} \div 24$$

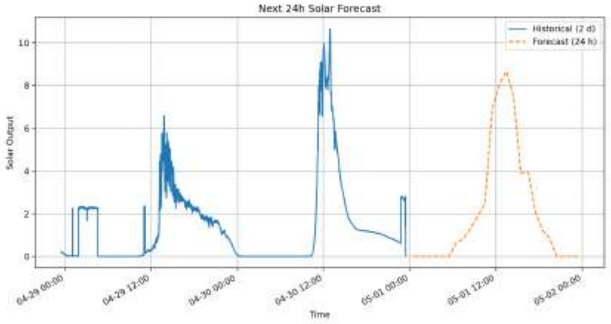


Figure 6 : Solar Forecast predicted by XGBoost, mapping OpenMeteo weather features

2. Grid Price Forecasting

We used the Nordpool Integration [2] which provides spot market electricity prices for regions in Europe. We could not find a good API providing real-time grid prices for the U.S. market, so we opted for Nordpool which is open-source, and integrates with Home Assistant.

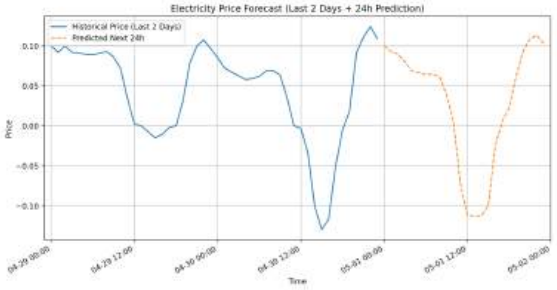


Figure 7 : Grid Price Forecast predicted by XGBoost, mapping OpenMeteo weather features to time of day

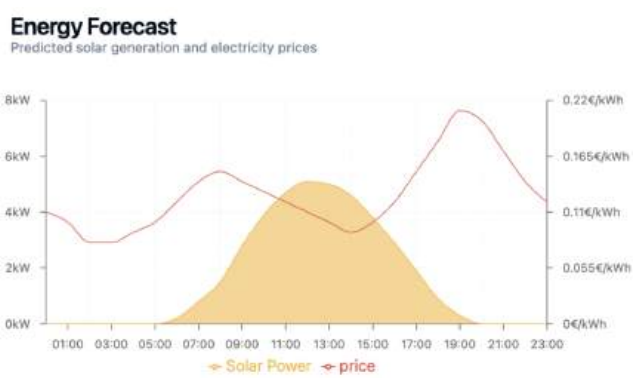


Figure 8: Forecasts rendered on the dashboard

B. *Optimization Framework*

We formulated the energy management problem as a linear optimization problem, with linear objective function and affine constraints. The main advantage of LP is that if the problem is well posed and the region of feasible possible solutions is convex, then a solution is guaranteed and solving times are usually fast when compared to other techniques such as dynamic programming, MDPs or simplex method. However, if a problem is too big (too many objective functions/ constraints) or it is not converging fast enough to a solution, memory limits can be exceeded.

Goal: Minimize cost of grid electricity across T time slots, accounting for solar offset.

Decision Variables:

- $x_{i,t} \in \{0, 1\}$ — whether device i runs at time t

Objective Function

$$\min_{x_{i,t}} \sum_{t=1}^{T} c_t \cdot \left(\sum_{i=1}^{N} P_i x_{i,t} - s_t \right)_+$$

Parameters:

- P_i — Power draw of device i
- c_t — Predicted grid price at time t
- s_t — Forecasted solar power available at time t
- d_i — Required run duration for device i

The constraints ensure that self-consumed power at any given time step cannot exceed the available solar power, and the self-consumed power cannot exceed the total power demand (If solar power generation is high, self-consumption is limited by demand, if load is high, self-consumption is limited by solar power available)

2. The second objective function aims to model how much power is drawn from the grid and quantify costs

$$\sum_{i=1}^{\frac{\Delta_{opt}}{\Delta_t}} -0.001 \cdot \Delta_t \cdot (\text{unit}_{\text{LoadCost}}[i] \cdot P_{\text{grid, consumed}}[i])$$

where Δ_{opt} is the total period of optimization (hrs), Δ_t is timestep (hr), $P_{grid,\,consumed}$ (W) is the power drawn from grid, $unit_{LoadCost}$ is the cost of drawing power from the grid ($/KWh)

Constraints

$$\sum_{t=1}^{T} x_{i,t} = d_i \quad \text{(Each device must run for required duration)}$$

$$x_{i,t} \in \{0, 1\} \quad \text{(Binary on/off decision)}$$

$$\left(\sum_{i=1}^{N} P_i x_{i,t} - s_t \right) \geq 0 \quad \text{(No negative grid draw)}$$

Power Balance Equation

$$P_{\text{solar},i} - P_{\text{deferrable load},i} - P_{\text{fixed load},i} - P_{\text{heat},i} + P_{\text{grid, exported},i} + P_{\text{grid, consumed},i} + P_{\text{battery, discharge},i} + P_{\text{battery, charge},i} = 0$$

This follows from the conservation of energy. It ensures that the sum of all power inflows and outflows is equal to zero, so that all energy produced, consumed, stored, or exchanged with the grid is accounted for.

To actually solve the LP problem, we used solvers in PuLP, a Python library for linear programming. After testing several solvers, we chose PULP_CBC_CMD, which is PuLP's default CBC (Coin-or branch and cut) solver. It provided a reliable balance between performance and compatibility. CBC was sufficient for our problem size and scheduling horizon, offering sub-second solve times with consistent results. Also sometimes GLPK and COIN_CMD which are other solvers occasionally failed to find a feasible solution, even when one clearly existed—under tight constraint conditions

We also incorporated user defined priorities, so in the case where the user chose a high/low priority, the new objective function became

$$\min \sum_t x_t \cdot (\alpha \cdot \text{price}_t - (1-\alpha) \cdot \text{solar}_t)$$

subject to the same constraints as before.

Priority	Objective Focus	α
High	Minimize Grid Cost	0.8
Medium	Balanced Cost and Solar	0.5
Low	Maximize Solar Self-Use	0.4

This flexibility allows users to tailor scheduling recommendations to their energy goals—whether they're aiming to save money, increase reliance on solar power instead of the grid, or find a balance between both.

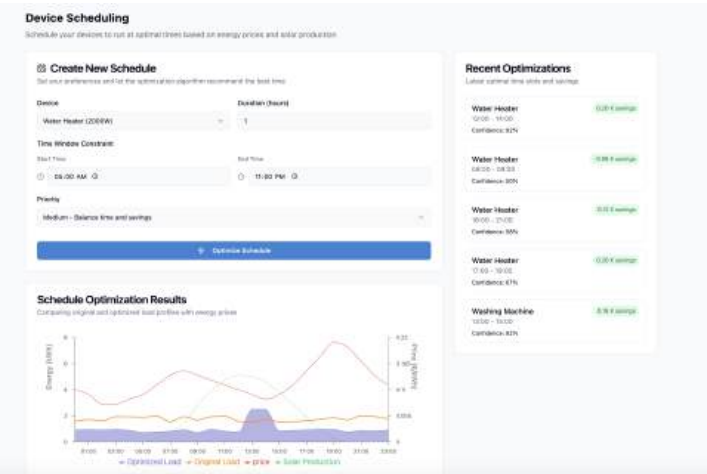


Figure 9 : SmartWatt's scheduling interface

C . Software (Frontend, Backend)

SmartWatt is built on a FastAPI + Python + React stack. The FastAPI backend is responsible for handling API requests between Home Assistant, executing optimization algorithms, managing real-time data streaming (storing power monitoring data), and running machine learning models. The backend also exposed RESTful endpoints for the frontend, sensor data, control commands and optimization results, and connected with Home Assistant via HTTP and MQTT protocols. SmartWatt frontend subscribes to WebSocket updates from the backend and Home Assistant. This allows the dashboard to reflect live changes in power consumption, solar generation, and device status without requiring manual refreshes. For example, when the ESP32 sends updated energy readings or a device changes state, those changes are pushed instantly to the UI via WebSocket events. This ensures the user interface remains synchronized with the underlying system state.. WebSocket integration was essential for enabling a smooth user experience in a system that operates on continuous real-time data.

For the frontend, we used React.js to create a responsive and interactive web interface. Users view real-time data from sensors, monitor device schedules, and manually override system-generated commands. The dashboard also visualized solar generation forecasts and price trends, allowing users to make informed decisions about appliance usage and tracking power consumption.

Initially, we implemented *on-demand inference* by loading the ML model during each POST request. This caused severe latency spikes and server crashes under concurrent load, as multiple model loads overwhelmed memory and CPU resources. To address this, we shifted to a *preloaded background model* approach, initializing the model once at server startup and sharing it across requests. This improved performance — reducing inference latency from **10s** to **<600 ms** and enabling reliable concurrent request handling.

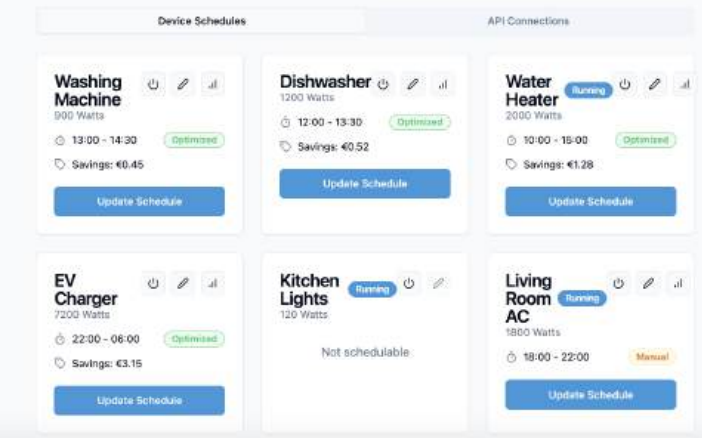


Figure 10 : Dashboard providing manual control over devices

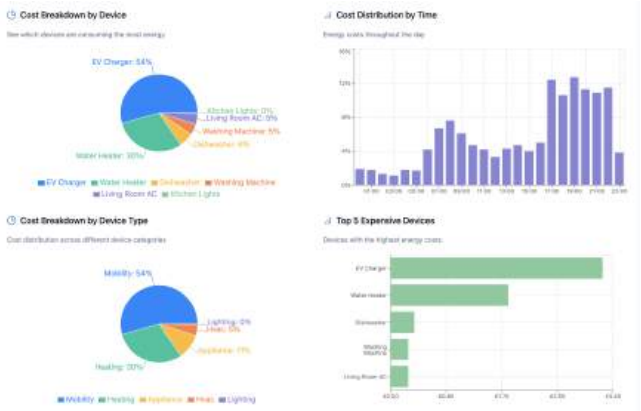


Figure 11 : SmartWatt's analytics dashboard visualizes energy costs by device, category, and time of day

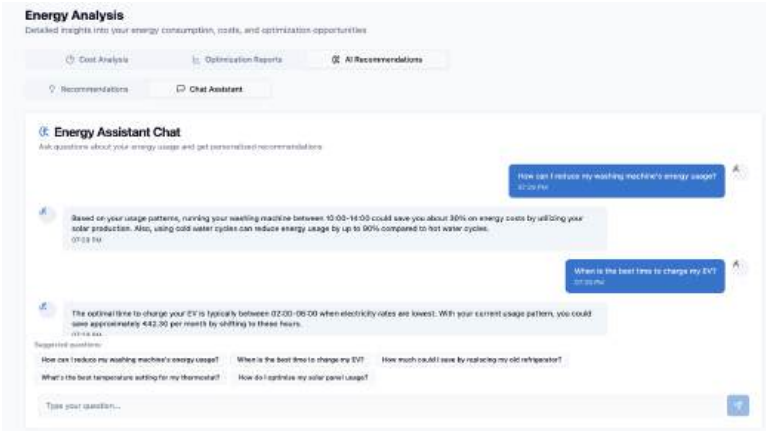


Figure 12 : SmartWatt's Chat Interface powered by OpenAI

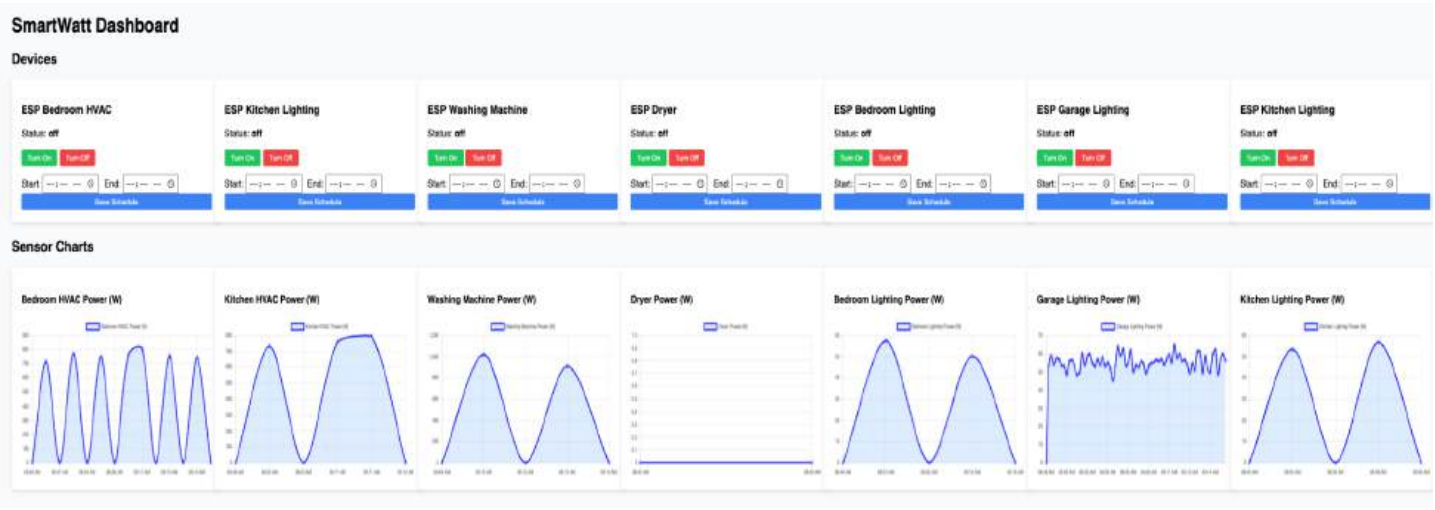


Figure 13 : Device Actuation Control Panel where users can switch on/off devices, schedule when to start and end devices, see device status (on/off) and see graphs of power consumption across devices

Figure 14: Model House displaying controllable electronic components

VII. Test, Verification and Validation

A. *Tests for Forecasting Models*

To validate the solar power forecasting model, we performed testing across 7 days of data and compared predicted solar output against real sensor readings from the solar panel. We achieved an average MAPE of 8.5% which is within our design requirement (MAPE <20%), RMSE of ~0.15W. To evaluate the performance of our solar forecasting model fairly, we split our dataset into 80% training and 20% testing based on chronological order to preserve time-series integrity. This ensured that the model was trained only on past data and evaluated on future, unseen data.

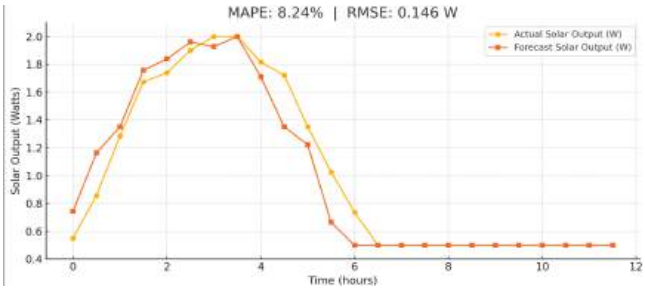


Figure 15 : Testing Accuracy for Solar Forecast

To test the grid price forecasting model, we trained on historical hourly price data and evaluated predictions against actual future prices from data obtained through NordPool API. After testing we found that the model had a MAPE of ~5% on average and RMSE of 0.0175 which is within our design requirements

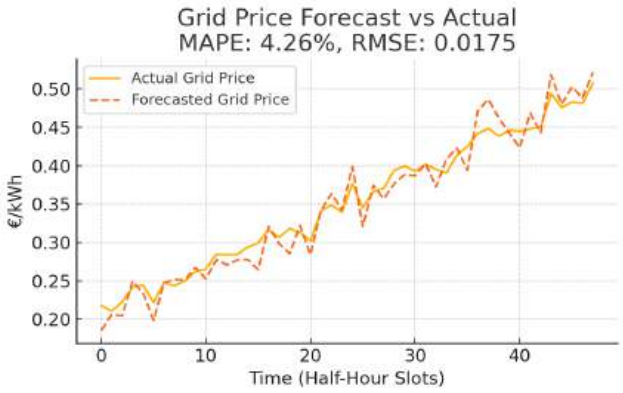


Figure 16 : Testing Accuracy for Grid Price Forecast

B. *Tests for Optimization Algorithm*

We created a series of controlled test cases where grid prices, solar output, and device constraints were manually specified. Each test scenario involved devices with known durations, priorities, and energy demands. We then verified whether the linear programming solver produced schedules that satisfy all constraints—such as exact duration allocation and binary on/off values at each time slot. We also compared the pre-optimization cost (if a device ran naively during peak hours) to the post-optimization cost computed using our cost function. On average, we found that the daily energy cost was **17%** lower when using SmartWatt's optimization compared to a random schedule. This was because we found that about **24**% of consumption of deferrable loads were moved to times of low grid prices and high solar availability.

The algorithm converged across all test scenarios, even for large 48 slot intervals. Additionally, feasibility testing was done to ensure that constraints (device duration, grid/solar capacity limits) were never violated. Additionally, we stress tested the model against inputs under varying conditions, such as empty solar forecasts and flat and spiky grid prices, to make sure that the optimizer returns logical output.

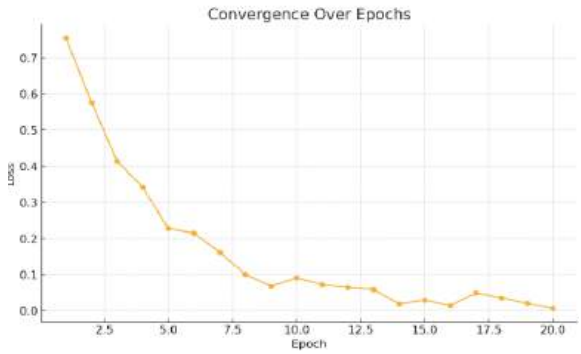


Figure 17: Training loss curve showing convergence of the solar forecasting model over 20 epochs, with loss decreasing steadily and stabilizing near zero

We tested for the duality gap—the difference between the primal and dual objective values at convergence. In theory, a properly solved LP should yield a duality gap of zero, indicating strong duality and an optimal solution. We ran the optimizer across multiple device scheduling scenarios and logged both the primal and dual values using the solver's diagnostic output. In all cases, the duality gap was within a negligible numerical tolerance (10^{-6}) confirming that the solver reached optimality. This validated that our LP formulation was well-posed and that the solver was functioning correctly.

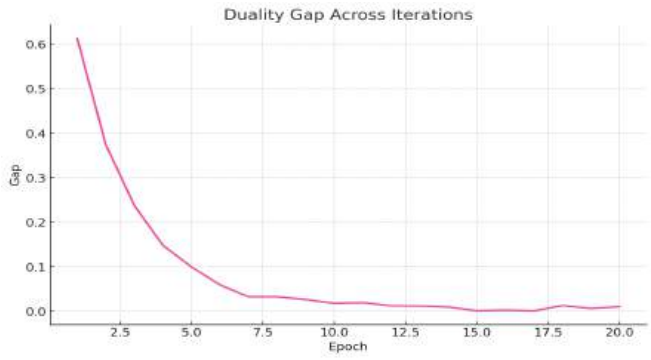


Figure 18 : Duality Gap Graph of the Optimizer

We measured solver runtime to ensure that the optimization algorithm could operate in real time, even on resource-constrained devices. We recorded the time taken to compute optimal schedules across varying numbers of devices and time slots. For typical use cases involving 3–5 devices over a 24-hour horizon (48 half-hour slots), the solver consistently produced results in **<1s.** This meets our design case requirements.

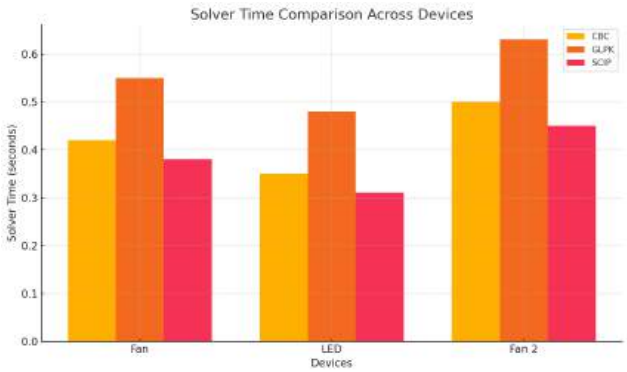


Figure 19: Solver Time Convergence

C. Tests for Device Monitoring & Control

To verify the accuracy and responsiveness of the ESP32-based power monitoring system, we measured how accurately the power sensors were logging power. The INA226 sensors were configured to sample power data continuously. Based on our calibration tests (measured V and I separately, then multiplied to get power and see how off that was from the data logged onto the dashboard), the measurement error remained within **±1%**, satisfying the design specification for accuracy.

To validate system reliability, we developed automated scripts to query backend API endpoints and confirm consistent data flow from the ESP32 to the dashboard. We also measured packet loss and transmission delays, both of which remained negligible under normal operating conditions.

To validate the reliability of the device control interface, we developed a script that programmatically sent control commands to the backend API—such as turning devices on or off and updating schedules. The script logged each HTTP request, along with the corresponding timestamp, response code, and device state returned by the server.

```
INFO:     Started reloader process [79820] using WatchFiles
INFO:     Started server process [79822]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:59503 - "GET / HTTP/1.1" 200 OK
INFO:smartwatt: Schedule POST received → entity=switch.maya_big_le
d, start=19:17, end=19:18
INFO:smartwatt:Schedule saved for switch.maya_big_led: {'start': '
19:17', 'end': '19:18'}
INFO:     127.0.0.1:59519 - "POST /schedule HTTP/1.1" 302 Found
INFO:     127.0.0.1:59519 - "GET / HTTP/1.1" 200 OK
```

These logs allowed us to verify that all commands were executed within an average response time of under 200 milliseconds, and that the server correctly propagated the new device states to the frontend via WebSocket updates. Additionally, this testing helped us confirm that the backend logic maintained state consistency and handled edge cases gracefully. The device control script thus served both as a stress test and a logging tool for ensuring robust API interactions.

To evaluate the responsiveness of device control, we measured the actuation latency—the time between issuing a control command via the dashboard and the physical activation of the device. Using a stopwatch, we found that the average actuation latency was approximately **2s** (the LEDs switched on/off in under 1s, the motors which had higher wattage took about 2s) under normal network conditions. This meets our design requirements. This latency includes API processing time, ESP32 command reception, and hardware switching delay. The system was also tested under Wi-Fi load to ensure performance stability, and no command failures or missed activations were observed. These results confirm that the system delivers near real-time actuation.

D. User Satisfaction and User Friendliness

We conducted a round of user testing with 6 participants who are energy aware. Testers were given common tasks such as checking their device power usage, adjusting optimization priorities, running the optimizer, and manually toggling devices through the dashboard. We observed their interactions and collected feedback on navigation clarity, responsiveness, and visual layout. Users gave a 7/10 on the scheduling understanding, 8/10 on feature usability and 7/10 on UI Navigation. Common suggestions included improving tooltip explanations for optimization parameters and adding clearer indicators for real-time data refresh. Most users found the system intuitive and said they would use SmartWatt to monitor their energy usage.

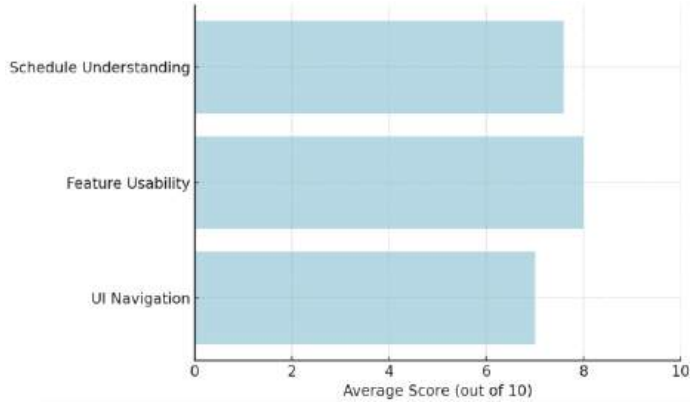


Figure 20 : User Satisfaction Survey Results

VIII. Project Management

A. Schedule

The original design schedule outlined a four-phase milestone plan: hardware prototyping, ML model development, backend integration, and dashboard interface. While the core structure remained intact, model training and hardware integration took longer than anticipated due to sensor calibration issues and API latency debugging, shifting optimization module testing by approximately one week. However, parallel progress on the dashboard and user testing allowed us to catch up by the final review stage. A detailed Gantt chart with milestone adjustments is provided on the final page of this report.

B. Team Member Responsibilities

Anya worked on the machine learning and optimization models, including forecasting models, REST API design, backend logic, and frontend dashboard development. She also handled device actuation and control via the ESP32 and integration with Home Assistant. Maya focused on the hardware layer, setting up the Raspberry Pi, sourcing components, building the prototype power system, and wiring the components into the house. They also supported backend hardware integration and system testing. Erika took charge of physical demo construction, visual layout design, chatbot integration, and full-stack frontend/backend coordination. The division of tasks allowed us to meet overlapping milestones without bottlenecks, even when some components—such as hardware integration—took longer than originally planned.

C. Bill of Materials and Budget

We spent about $350 of our budget. Please refer to the Bill of Materials at the end of our report.

D. TechSpark Usage

We used the laser cutters in TechSpark for our model house, as well as the laser cutters in the IDeATe space in Hunt Library.

E. Risk Management

We encountered several risks across all subsystem developments. In the ML Inference subsystem, a performance issue arose from loading trained models inside the FastAPI route handlers. This caused server crashes and latency spikes above 3 seconds under concurrent requests. To address this, we preloaded the models at server startup using FastAPI's lifecycle hooks, reducing average inference time to <1s to meet the design requirement. We also implemented fallback logic to handle cases where the model or cache failed. Additionally, we found that errors in the solar and price forecasts could propagate into the optimization module, leading to unreliable device schedules. This was mitigated by introducing forecast smoothing and uncertainty buffers. During model training, our LSTM and CNN forecasters sometimes plateaued due to vanishing gradients, especially when predictions already closely matched real values. Initially, we added regularization terms and white noise perturbations to maintain learning dynamics. However, after the predictions were suboptimal, we pivoted to using XGBoost for our forecasting model. For the solar forecast, by feeding in recent cloud cover, irradiance, and precipitation forecasts from OpenMeteo, XGBoost produced more stable and interpretable solar output predictions.

Additionally, closer to the final week, the router on Maya's laptop stopped working. Relying on a laptop-generated hotspot led to intermittent dropouts between the Raspberry Pi and backend services, resulting in unreliable device control and data synchronization. To resolve this, we transitioned to a dedicated router to establish a more stable local network, while also conducting stress tests and keeping the old setup as a fallback in case of connectivity issues. On the Hardware side, integrating the ESP32, INA226 sensors, relays, and Raspberry Pi into the limited physical space of the demo house presented both spatial and wiring challenges. To reduce risk, we adopted a modular prototyping approach, building and validating each subsystem independently before integration. This helped isolate and fix electrical issues early. Also several relays had to be swapped out since they failed under sustained inductive loads from the motors.

Within the Backend and API layer, performance degraded when users made concurrent optimization requests or when the Matplotlib charts were generated on demand. Chart rendering added ~400 ms of latency per user. We resolved this by precomputing figures in the background and caching results.

Finally, to manage Team and Schedule risk, we distributed responsibilities across subsystems. When hardware delays occurred, other team members progressed on independent parallel tasks. The division allowed us to recover from unexpected setbacks and maintain the timeline.

IX. ETHICAL ISSUES

While SmartWatt aims to empower users with real-time energy optimization and sustainability insights, several ethical considerations arise in privacy and public safety.

A core feature of our system is a personalized chatbot assistant that helps users understand their energy usage, recommend schedule changes, and explain optimization outcomes in natural language. However, this level of personalization introduces privacy risks. The chatbot interacts with historical device usage patterns, forecasted behaviors, and potentially sensitive data (e.g., routines inferred from load consumption schedules). If improperly secured or logged, this data could be misused or accessed

by unauthorized actors. To mitigate this, we ensured that all user interactions are processed locally or through authenticated APIs, and no personal identifiers are stored with logs. Long-term deployments should consider incorporating encryption, role-based access control, and transparent data policies to ensure user trust.

Another ethical concern involves algorithmic decision-making, particularly in how our optimization system balances cost-saving against comfort or fairness. Edge cases—such as unexpected solar drop-offs or inaccurate forecasts—can lead the system to schedule essential devices at inconvenient times or delay operation entirely. This could disproportionately affect users with strict routines, accessibility needs, or less flexibility in appliance use. For example, someone relying on a medical device or needing consistent hot water might be negatively impacted if their device is deprioritized due to cost considerations. We addressed this by allowing users to set strict scheduling constraints and override optimization preferences. Still, further refinement of the optimization engine to prioritize equity, not just efficiency, would be essential in broader deployments.

SmartWatt also raises broader environmental ethical considerations. While its core mission aligns with reducing carbon emissions by shifting load to cleaner energy windows, there's a risk of unintentionally increasing grid strain if poorly scheduled or scaled across many users without grid coordination. To mitigate this, we designed the system to prioritize solar self-consumption and support demand flattening, not just cost minimization. In the future, integrating carbon intensity signals into the objective function could align optimization with environmental goals.

From a public safety and welfare perspective, automation of high-power devices (like HVACs, water heaters, or EV chargers) poses potential risks if optimization overrides essential usage. Edge cases—such as a misforecasted solar drop could delay device activation or cluster schedules, inadvertently disrupting daily routines or, in extreme cases, endangering individuals who rely on consistent access (e.g., for medical or mobility devices). To address this, we allow users to set non-negotiable constraints (e.g., minimum run windows) and manually override optimization when needed.

Lastly, over-automation risk must be acknowledged. As users come to rely on SmartWatt's recommendations, there's potential for overdependence or reduced visibility into how decisions are made. If a model error occurs or a system fails silently, users may miss critical issues. To mitigate this, we built transparency into the dashboard—allowing users to see forecasted prices, solar output, and optimized decisions with confidence scores—so they can override or inspect results as needed.

X. Related Work

Google Nest has played a significant role in shaping smart home energy solutions with products like Nest Renew and the Nest Thermostat [13]. Nest Renew, launched in 2021, optimizes household energy usage by shifting consumption to cleaner energy periods through its Energy Shift feature. Additionally, Savings Finder continuously analyzes user habits to recommend energy-efficient settings. The Nest Thermostat, redesigned for affordability and ease of use, incorporates smart scheduling and HVAC monitoring, making energy efficiency more accessible to a broader audience. These developments highlight Google's commitment to integrating AI-driven automation into household energy management.

Academic projects (OpenEnergyMonitor [18], Home Assistant's Energy Dashboard) offer strong foundations for energy data collection and visualization. However, they often lack machine learning–driven forecasting and optimization or require manual configuration of automations. Our project extends these ideas by adding real-time LP-based scheduling, personalized chatbot interaction, and integration of solar and price forecasting.

Tesla's Powerwall enables homes to store excess solar energy for later use, reducing reliance on the grid during peak hours. Startups like Span and Sense are innovating with smart electrical panels and real-time energy monitoring, allowing users to track and control their energy consumption at the circuit level. Additionally, academic research in edge computing and IoT-based home automation has explored decentralized control systems that optimize energy use with minimal latency. These advancements collectively demonstrate a shift toward intelligent, self-regulating home energy ecosystems, paving the way for more sustainable and cost-effective living environments.

XI. Summary

Our system successfully met the majority of the design specifications laid out at the beginning of the project. We achieved reliable real-time power monitoring with <2% sensor error, functional device control with sub-200 ms actuation latency, and optimization algorithms that consistently reduced energy cost by 20%. Additionally, the ML forecasting components achieved acceptable accuracy for both solar output and grid price prediction, enabling effective schedule generation.

Together, these components formed a tightly integrated system that responded in real time to user inputs, sensor data, and forecasted conditions—delivering to the user transparency over their energy consumption, improved control over their home energy use and cost savings.

Future Work

To improve system robustness and scalability, several extensions can be made. Refining the cost function to include user comfort and usage continuity can produce more intuitive, user-friendly schedules. Adaptive learning can be implemented to retrain forecast models on rolling data, improving long-term accuracy. Parallelizing solver execution and improving backend caching would reduce latency and improve performance under load. The system can also be extended to support battery, wind, and geothermal power, enabling broader renewable energy integration. Finally, developing a mobile app would be useful since most users use phones to remotely control appliances and view their power consumption data.

Lessons Learned

Integration of all the subsystems was challenging, but taught us how to effectively synchronize hardware with software. Stress testing was good for diagnosing failures and improving stability. Through iterative testing and user feedback, we gained valuable experience in user-centered design—ensuring transparency, manual override, and real-time feedback were central to the interface. We learned that design modularization is essential for achieving parallel development, maintaining steady progress, and ensuring effective collaboration across subsystems. Identifying risks early and designing for safe failure modes and redundancy is important.

Glossary of Acronyms

MQTT – Message Queuing Telemetry Transport
OBD – On-Board Diagnostics
RPi – Raspberry Pi
GPIO – General Purpose Input-Output
TOU : Time-of-Use Pricing
RMSE: Root Mean Square Error
MAPE: Mean Average Percentage Error
API: Application Programming Interface
REST: Representational State Transfer

References

[1] "SDKs and Developer Resources: SolcastTM." *Solcast*, solcast.com/sdk. Accessed 26 Feb. 2025.

[2] Home Assistant. "Nord Pool." *Home Assistant*, www.home-assistant.io/integrations/nordpool/. Accessed 26 Feb. 2025.

[3] Home Assistant. "Raspberry Pi." *Home Assistant*, www.home-assistant.io/installation/raspberrypi/. Accessed 26 Feb. 2025

[4] "Energy Management for Home Assistant." *EMHASS*, emhass.readthedocs.io/en/latest/. Accessed 26 Feb. 2025.

[5] Lauinger, D., et al. "A linear programming approach to the optimization of Residential Energy Systems." *Journal of Energy Storage*, vol. 7, Aug. 2016, pp. 24–37, https://doi.org/10.1016/j.est.2016.04.009.

[6] Davidusb-Geek. "DAVIDUSB-Geek/Solarhome-Control-Bench: Open Testbench for Control and Optimization Methods for the Energy Management of a Simple Solar Home." *GitHub*, github.com/davidusb-geek/solarhome-control-bench. Accessed 26 Feb. 2025.

[7] Custom-Components. "Custom-Components/Nordpool: This Component Allows You to Pull in the Energy Prices into Home-Assistant." *GitHub*, github.com/custom-components/nordpool. Accessed 26 Feb. 2025.

[8] "INA226." *INA226 Data Sheet, Product Information and Support | TI.Com*.

[9] Simmini, Francesco, et al. "Model Predictive Control for Efficient Management of Energy Resources in Smart Buildings." *MDPI*, Multidisciplinary Digital Publishing Institute, 7 Sept. 2021, www.mdpi.com/1996-1073/14/18/5592.

[10] Caldera, Matteo, et al. "Energy-Consumption Pattern-Detecting Technique for Household Appliances for Smart Home Platform." *MDPI*, Multidisciplinary Digital Publishing Institute, 11 Jan. 2023, www.mdpi.com/1996-1073/16/2/824.

[11] Almughram, Ohoud, et al. "Home Energy Management Machine Learning Prediction Algorithms: A Review." *Atlantis Press*, Atlantis Press, 2 Feb. 2022, www.atlantis-press.com/proceedings/iciai-21/125969975.

[12] "Free Weather API." *Open*, open-meteo.com/. Accessed 28 Feb. 2025.

[13] Google Store. *Google*, https://store.google.com/?hl=cs&pli=1. Accessed 28 Feb. 2025.

[14] Tesla. *Powerwall*, https://www.tesla.com/powerwall. Accessed 28 Feb. 2025.

[15] "Ecobee." *Smart Thermostats & Smart Home Devices*, www.ecobee.com/en-us/. Accessed 28 Feb. 2025.

[16] "Harness the Sun to Make, Use, Save, and Sell Your Own Power." *Enphase*, enphase.com/?srsltid=AfmBOoryH-AhxqQKFMHED5-FIIkp6yu4Jt3k1B9LR_pXMIxEmPiatxu4. Accessed 28 Feb. 2025.

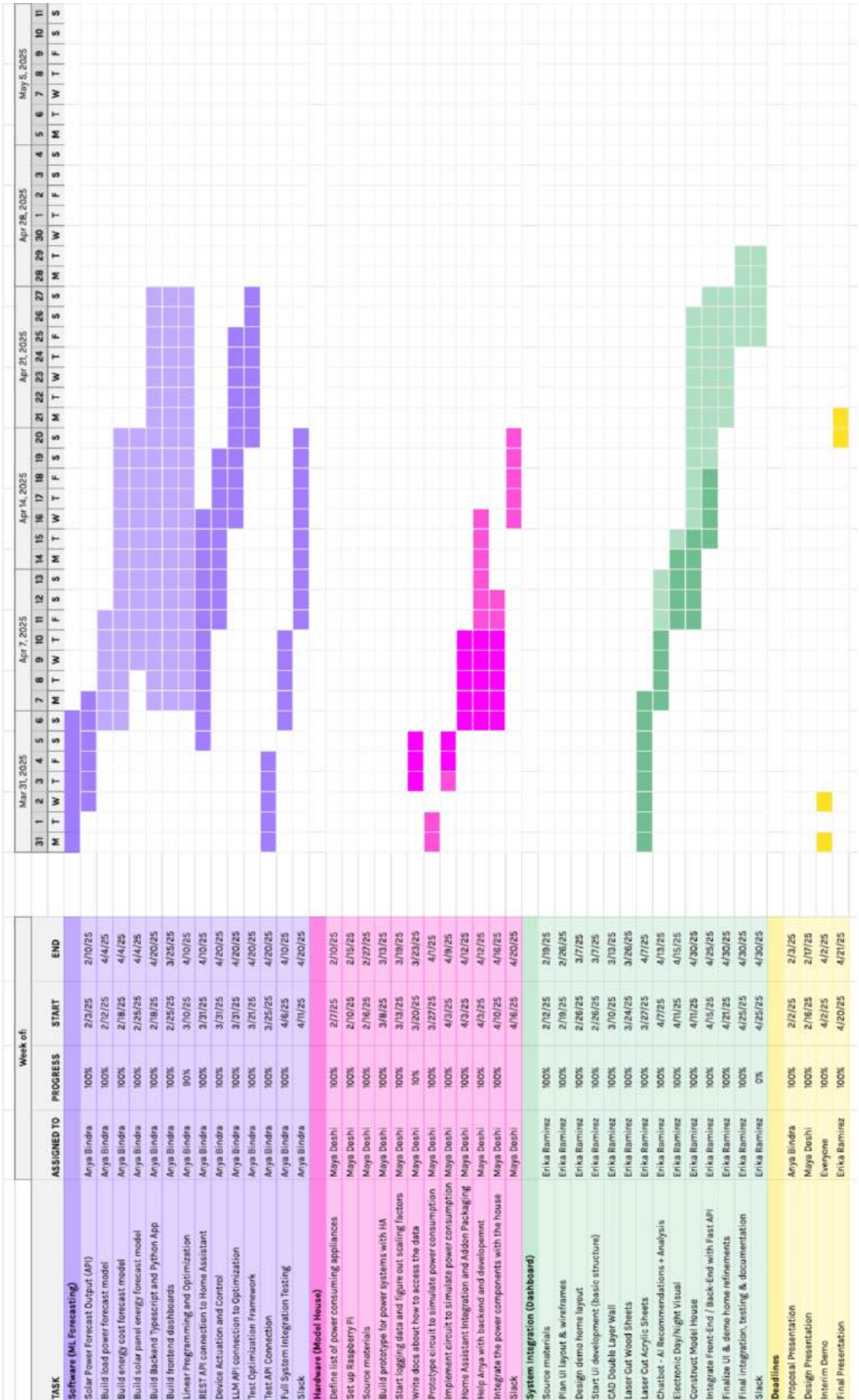
[17] Dávid Markovics, et al. "Comparison of Machine Learning Methods for Photovoltaic Power Forecasting Based on Numerical Weather Prediction." *Renewable and Sustainable Energy Reviews*, Pergamon, 23 Mar. 2022, www.sciencedirect.com/science/article/pii/S136403212200274X.

[18] "Open Energy Monitor" *OpenEnergyMonitor*, openenergymonitor.org/. Accessed 2 May 2025.

BILL OF MATERIALS

Subsystem	Used?	Introduced after Design Report?	Item	Quantity	Unit Cost	Total Cost	Provider
Circuitry + Electronic Components	Yes	No	ESP32	1	$0.00	$0.00	Roboclub
	Yes	No	RPi5	1	$0.00	$0.00	Inventory
	Yes	No	ESP32 Relay Board	1	$22.00	$22.00	Amazon
	Yes	No	Solar Panels	1	$15.00	$15.00	Amazon
	Yes	No	Power Bank	2	$0.00	$0.00	Personal Device
	Yes	No	Breadboard	1	$0.00	$0.00	Personal
	Yes	No	INA226 Power Sensors	1	$16.99	$16.99	Amazon
	Yes	Yes	Relay Switches	4	$0.00	$0.00	Roboclub
	Yes	Yes	LEDs	1	$0.00	$0.00	Roboclub
	Yes	No	Fans	1	$12.90	$12.90	Amazon
	No	Yes	Speaker (Bluetooth only)	1	$9.99	$9.99	Amazon
	No	Yes	6 Pack DC 1.5-3V 15000RPM mot	1	$6.99	$6.19	Amazon
	Yes	No	Spool of Wire	1	$0.00	$0.00	TechSpark
Model House for Demo	Yes	No	Basswood Sheets 12"x20" (12pk)	1	$36.99	$36.99	Amazon - AWIZOM
	Yes	No	Titebond Wood Glue (16oz)	1	$5.48	$5.48	Amazon - Titebond
	Yes	No	8" Exacto Knife	1	$5.97	$5.97	Amazon - Fiskars
	Yes	No	Acryllic Sheets	10	$21.98	$219.80	Amazon - Lesnlock
						$351.31	

GANTT CHART TABLE



TASK	ASSIGNED TO	PROGRESS	Week of: START	END
Software (ML Forecasting)				
Solar Power Forecast Output (API)	Anya Bindra	100%	2/3/25	2/10/25
Build load power forecast model	Anya Bindra	100%	2/12/25	4/4/25
Build energy cost forecast model	Anya Bindra	100%	2/18/25	4/4/25
Build solar panel energy forecast model	Anya Bindra	100%	2/25/25	4/4/25
Build Backend Typescript and Python App	Anya Bindra	100%	2/18/25	4/20/25
Build frontend dashboards	Anya Bindra	100%	2/25/25	3/25/25
Linear Programming and Optimization	Anya Bindra	90%	3/10/25	4/10/25
REST API connection to Home Assistant	Anya Bindra	100%	3/31/25	4/10/25
Device Actuation and Control	Anya Bindra	100%	3/31/25	4/20/25
LLM API connection to Optimization	Anya Bindra	100%	3/31/25	4/20/25
Test Optimization Framework	Anya Bindra	100%	3/21/25	4/20/25
Test API Connection	Anya Bindra	100%	3/25/25	4/20/25
Full System Integration Testing	Anya Bindra	100%	4/6/25	4/10/25
Slack	Anya Bindra		4/11/25	4/20/25
Hardware (Model House)				
Define list of power consuming appliances	Maya Doshi	100%	2/7/25	2/10/25
Set up Raspberry Pi	Maya Doshi	100%	2/10/25	2/15/25
Source materials	Maya Doshi	100%	2/16/25	2/27/25
Build prototype for power systems with HA	Maya Doshi	100%	3/8/25	3/13/25
Start logging data and figure out scaling factors	Maya Doshi	100%	3/13/25	3/19/25
Write docs about how to access the data	Maya Doshi	10%	3/20/25	3/23/25
Prototype circuit to simulate power consumption	Maya Doshi	100%	3/27/25	4/1/25
Implement circuit to simulate power consumption	Maya Doshi	100%	4/3/25	4/9/25
Home Assistant Integration and Addon Packaging	Maya Doshi	100%	4/3/25	4/12/25
Help Anya with backend and development	Maya Doshi	100%	4/3/25	4/12/25
Integrate the power components with the house	Maya Doshi	100%	4/10/25	4/16/25
Slack	Maya Doshi		4/16/25	4/20/25
System Integration (Dashboard)				
Source materials	Erika Ramirez	100%	2/12/25	2/19/25
Plan UI layout & wireframes	Erika Ramirez	100%	2/19/25	2/26/25
Design demo home layout	Erika Ramirez	100%	2/26/25	3/7/25
Start UI development (basic structure)	Erika Ramirez	100%	2/26/25	3/7/25
CAD Double Layer Wall	Erika Ramirez	100%	3/10/25	3/13/25
Laser Cut Wood Sheets	Erika Ramirez	100%	3/24/25	3/26/25
Laser Cut Acrylic Sheets	Erika Ramirez	100%	3/27/25	4/7/25
Chatbot - AI Recommendations + Analysis	Erika Ramirez	100%	4/7/25	4/13/25
Electronic Day/Night Visual	Erika Ramirez	100%	4/11/25	4/15/25
Construct Model House	Erika Ramirez	100%	4/11/25	4/30/25
Integrate Front-End / Back-End with Fast API	Erika Ramirez	100%	4/15/25	4/25/25
Finalize UI & demo home refinements	Erika Ramirez	100%	4/21/25	4/30/25
Final integration, testing & documentation	Erika Ramirez	100%	4/25/25	4/30/25
Slack	Erika Ramirez	0%	4/25/25	4/30/25
Deadlines				
Proposal Presentation	Anya Bindra	100%	2/2/25	2/3/25
Design Presentation	Maya Doshi	100%	2/16/25	2/17/25
Interim Demo	Everyone	100%	4/2/25	4/2/25
Final Presentation	Erika Ramirez	100%	4/20/25	4/21/25