

IoT Edge-Computing System for Real-Time Power Monitoring in Smart Agriculture

Farhan Aziz Umrani, APcore (Asia Pacific Center of Robotics Engineering), Institute of Engineering-Asia Pacific University of Technology & Innovation, Kuala Lumpur, Malaysia

Email: Farhanazizumrani111@gmail.com

TS. Suresh Gobee, School of Engineering-Asia Pacific University of Technology & Innovation, Kuala Lumpur, Malaysia

Email: suresh.gobee@apu.edu.my

DR. Adeline Sneha John Chrisastum, School of Computing Computer Science & Software Engineering- Asia Pacific University of Technology & Innovation, Kuala Lumpur, Malaysia

Email: adeline.john@apu.edu.my

PROF. IR. EUR. ING. DR. Vinesh Thiruchelvam, Innovation and Enterprise-Asia Pacific University of Technology & Innovation, Kuala Lumpur, Malaysia

Email: dr.vinesh@apu.edu.my

Maythem K. Abbas, School of Computing-Asia Pacific University of Technology and Innovation, Kuala Lumpur, Malaysia

Email: maythem.abbas@utp.edu.my

ABSTRACT:

To make Controlled Environment Agriculture (CEA) more modern, the infrastructure needs to be very strong so that automatic crops can survive. Traditional Internet of Things (IoT) power tracking systems depend on centralized cloud processing, which makes them subject to network latency and data loss when the internet goes down in certain areas. This paper talks about how an industrial-grade Edge-toCloud power tracking environment was designed and put into use at the APU AGRHI Hub. An edge node running an event-driven Node.js server is used in this design to connect directly to 20A Tuya WiFi smart plugs over the local area network (LAN). By skipping over third-party cloud APIs, the system is able to integrate data with zero delay and use dynamic micro-interval polling to accurately measure kilowatt-hour (kWh) accumulation. An "Offline-First" fault-tolerance system was created to protect the data completely. It includes a localized "Black Box" SD-card memory module, a programmatic software watchdog, and a staggered electrical boot process. A quota-optimized batching method is used to sync data asynchronously to a NoSQL cloud database (Google Firestore). WebSocket streams are used for real-time reactive state management in the frontend user panel, which was built with the Flutter framework. There is an industrial Dark Mode Glassmorphism design on the user interface, a zero-fill algorithmic chart for accurate temporal analytics, and a soft-lock administration protocol to keep hardware from being activated by mistake. The mixed design that was made shows a very scalable and fault-tolerant way to use data in agriculture.

KEYWORDS:

Edge Computing, Internet of Things (IoT), Smart Agriculture, Power Monitoring, Node.js, Flutter, Fault Tolerance, Telemetry.

1. Introduction:

Controlled Environment Agriculture (CEA) is growing quickly. For example, the hydroponic ecosystems at the APU AGRHI Hub need very strong infrastructure to handle high-wattage electricity loads all the time. Automated actuators, like water pumps, thermal controllers, and high-intensity lights, need to work without interruption and according to a set plan. Real-time power tracking and localized relay control are therefore strict requirements for the safety of the facility, its ability to make money, and the avoidance of catastrophic crop failure.

Most traditional Internet of Things (IoT) applications in agriculture use centralized Cloud-to-Device (C2D) architectures. Sending raw telemetry to remote servers makes initial setup easier, but it also creates serious operating vulnerabilities. When the Wide Area Network (WAN) goes down, clouddependent hardware stops working completely. This means that energy measures are lost forever and actuation schedules don't work. Also, geographical network latency and strict business Application Programming Interface (API) rate limits limit the number of times data can be polled, which hides dangerous electrical problems.

This article suggests a strong, mixed Edge-to-Cloud design to lower these infrastructure risks. An "Edge Node" Raspberry Pi 4 is used for the system's main computational reasoning, mathematical energy integration, and state buffering. This gives the system complete independence. There is a closed Local Area Network (LAN) area that connects the Edge Node to high-load Tuya Wi-Fi smart switches and smart meter circuit breakers. In order to safely manage Master Air Conditioners, vertical tower pumps, and central motors, this totally skips over external commercial APIs.

In this stateless model, the cloud only assists with asynchronous communication with a cross-platform Flutter dashboard. In this paper, the software and math methods used are explained in detail. The local Node.js edge server, "Offline-First" fault-tolerance mechanisms, NoSQL database limit optimization, and the programmatic design of a zero-latency, reactive user interface are all addressed.

The complete data path of this Edge-to-Cloud architecture is illustrated below in Figure 1.

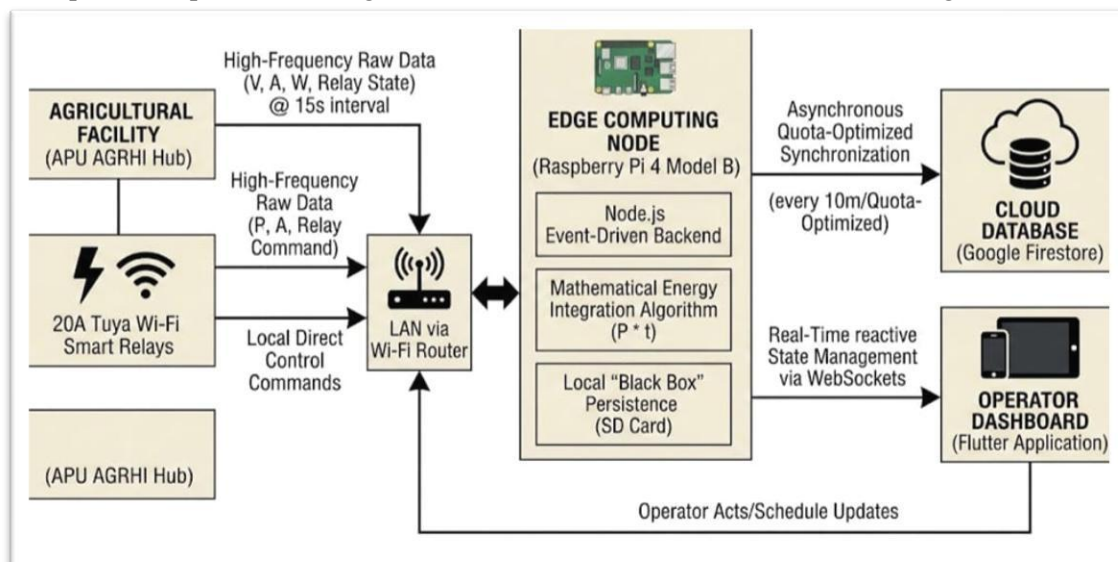


Figure 1: IoT Edge-Computing System

Figure 1 can be described as High-level system architecture flowchart illustrating the localized data path from the 20A Smart Relays to the Edge Computing Node (Raspberry Pi 4), asynchronously bridging to the Cloud Database (Firestore), and streaming to the Operator Dashboard (Flutter).

2. Literature Review

Integration of Internet of Things (IoT) technologies into farming systems has been written about a lot recently. But the basic designs suggested in early research put a lot of weight on centralized cloud computing, which makes continuous, high-stakes Controlled Environment Agriculture (CEA) very vulnerable to major operating problems.

2.1 Traditional IoT Architectures in Agriculture

The basic structure of IoT is made up of three levels: the sensing layer, which is made up of hardware devices; the network layer, which sends data; and the application layer, which does process in the cloud [1]. This standard design lets farmers check on things like soil wetness, outdoor temperature, and humidity from afar [2]. However, it is noted that these tools are mostly "cloud centric" [2]. The actual gear doesn't have much computing power in its own area; it just sends raw information to outside data centers to be analyzed.

Cloud-Centric IoT makes it a lot easier to set up hardware, but there are some major problems with this approach, mainly when it comes to delay, bandwidth limits, and fault tolerance [3]. In a modern CEA center, automatic tools (like fluid pumps and temperature controllers) need to react right away to changes in the surroundings. Sending mission-critical data over external Wide Area Networks (WAN) causes uncertain regional delays, which means that standard cloud designs are not ideal for timesensitive farming control [3].

2.2 Communication Protocols and Cloud Vulnerabilities

The transmission methods used by industrial IoT gear make the problems with Cloud-Centric agriculture even worse. A thorough analysis of IoT communication standards [4] points out that devices usually use Message Queuing Telemetry Transport (MQTT) or Hypertext Transfer Protocol (HTTP) to talk to private business servers. If the local Internet Service Provider (ISP) goes down or if the business server sets tight Application Programming Interface (API) rate limits, the hardware is cut off from its computer logic. In a hydroponic facility, a broken water pump can kill all the plants in just a few hours. This shows how dangerous it is to depend solely on external cloud connections for facility safety.

2.3 The Paradigm Shift toward Edge Computing

Recent computer science writing has pushed for a move toward Edge Computing as a way to get around the delay and stability problems that come with Cloud-Centric IoT. Edge Computing is defined as when computing and storing tools are put at the "edge" of the network, very close to the source of the data [5]. Edge Computing greatly lowers the amount of data sent to the cloud by handling information locally. This gets rid of geographical network delays and lets localized systems work without the internet when it goes down.

2.4 Research Gap and Proposed Contribution

While the idea behind Edge Computing is well described [5], there isn't a lot of real-world research that shows how Offline-First, Edge-to-Cloud systems can be used in busy farming sites with consumergrade, high-load (20A) hardware. A lot of the study that has already been done is on passive monitoring of the surroundings instead of active, high-voltage movement and mathematical energy tracking.

This work fills this particular study gap. The suggested design turns standard Cloud-Centric smart switches into a fully independent, fault-tolerant Edge-to-Cloud environment by installing an eventdriven Node.js edge server on a Raspberry Pi 4. Besides that, it adds new ways to use programs, like "Black Box" memory persistence, real-time calculus integration for accurate kWh auditing, and algorithmic NoSQL quota optimization. These create a very strong and scalable model for modern Smart Agriculture.

3. Hardware Integration and Edge Node Processing

In the AgrhiHub power monitoring ecosystem, all of the computations happen on the Edge Node. This means that the ecosystem doesn't have to rely on centralized business IoT servers like it usually does. The physical layer uses heavy-duty 20-Ampere (20A) tuya Wi-Fi smart switches to handle the strict electrical needs of Controlled Environment Agriculture (CEA). The facility's Master Air Conditioning (AC) units produce large inrush currents, and the vertical farming fluid pumps continuously draw a lot of torque. These high-load relays were chosen because they have built-in power tracking circuits and can safely handle these currents. Edge Node, the facility's main processing unit, is installed on a Raspberry Pi 4 computer and runs a highly optimized, event-driven Node.js core.

3.1 Localized Network Topology and API Bypass

Tuya-based smart relays send encrypted data to private cloud platforms outside of Tuya (like the Tuya Developer Platform) in normal commercial deployments. This standard routing causes too much latency and limits the facility's API rate to levels set by the business.

This external dependence is actively broken by the suggested architecture. To identify and talk to the 20A smart plugs directly over the isolated local subnet (working within the 10.130.1.X IP range), the Node.js app uses a Transmission Control Protocol (TCP) that was reverse engineered. The Raspberry Pi 4 makes a direct, long-lasting socket link with the hardware by getting the unique hardware identification arrays and local security keys. Because of this localized structure, raw telemetry packets will never leave the farm to be processed. By doing this, the latency that comes with geographical cloud routing is removed, and the Edge Node can poll the hardware at very high rates without causing API throttling or having ongoing commercial cloud-computing costs.

3.2 Dynamic Power Integration and Temporal Calculus

Smart relays sold in stores send out raw, real-time telemetry data points, especially Data Point Status (DPS) indices that show Voltage in decivolts, Current in milliamperes, and Power in deciwatts. Most importantly, they don't automatically figure out very accurate daily total energy measures (Kilowatthours).

For correct total energy use, the Edge Node needs to run a continuous mathematical integration algorithm that figures out the area under the power-time curve. The system sets up a 15,000-millisecond (15sec) high-frequency polling heartbeat period. Although network reaction times and TCP socket latencies change slightly, a static time multiplier can't be used because it would cause more mathematical drift.

The Node.js processor instead figures out the exact difference in time between successful socket replies on the fly. Sequential integration reasoning is used to figure out the localized energy increase. To begin, the DPS payload's immediate power (P_W) is taken out and changed to kilowatts (P_{kW}):

$$P_{kW} = \frac{P_W}{1000}$$

Second, the exact temporal differential (Δt) since the last successful hardware scan is calculated utilizing UNIX timestamps, and subsequently converted from milliseconds into hours (Δt_{hours}):

$$\Delta t_{hours} = \frac{t_{current} - t_{previous}}{3600000}$$

Finally, a continuous Riemann sum estimate is used to add to the localized energy pool:

$$E_{daily} = E_{daily} + (P_{kW} \times \Delta t_{hours})$$

This dynamic, constant integration makes sure that short-term electrical problems are correctly recorded and added to the daily total. For example, a hydroponic pump short-cycling for only a few seconds or an AC compressor drawing sudden peak loads. The method keeps track of cumulative energy to four decimal places by running this math on the Raspberry Pi 4 every 15 seconds. This level of accuracy is not possible with manual baseline auditing.

The successful execution of this four-decimal precision energy integration algorithm is demonstrated in the console output in Figure 2.

```
/home/pi/.pm2/logs/agrihub-smart-out.log last 15 lines:
[tower_1] < 234.6V | 59.6W | 0.1420 kWh | Buf: 0 | Next: 404s
[tower_1] < 234.6V | 59.6W | 0.1423 kWh | Buf: 0 | Next: 389s
[tower_1] < 234.6V | 59.6W | 0.1425 kWh | Buf: 0 | Next: 375s
[tower_1] < 234.6V | 59.6W | 0.1428 kWh | Buf: 0 | Next: 360s
[tower_1] < 234.6V | 59.6W | 0.1430 kWh | Buf: 0 | Next: 344s
[tower_1] < 234.6V | 59.6W | 0.1433 kWh | Buf: 0 | Next: 329s
[tower_1] < 234.6V | 59.6W | 0.1435 kWh | Buf: 0 | Next: 315s
[tower_1] < 234.6V | 59.6W | 0.1438 kWh | Buf: 0 | Next: 299s
[tower_1] < 234.6V | 59.6W | 0.1440 kWh | Buf: 0 | Next: 284s
[tower_1] < 234.6V | 59.6W | 0.1443 kWh | Buf: 0 | Next: 269s
```

Figure 2: Console output from the Raspberry Pi 4 terminal operating via the PM2 process manager

Figure 2 can be described as Console output from the Raspberry Pi 4 terminal operating via the PM2 process manager, demonstrating the localized high-frequency TCP polling, real-time voltage reading, and the execution of the four-decimal precision energy integration algorithm.

4. Fault Tolerance, Offline-First Architecture, and Safety Enforcement

The most important weakness of any smart farming site is how stable its network is. An IoT Edge Node that relies on volatile memory (RAM) or a live Wide Area Network (WAN) link to process and store cumulative energy metrics will lose all of its data if the microcomputer restarts or the Internet Service Provider (ISP) goes down. An "Offline-First" fault-tolerance design with multiple levels was used to make the APU AGRHI Hub completely reliable.

4.1 Non-Volatile Persistence Module ("The Black Box")

The Edge Node builds a non-volatile memory module using the native Node.js File System (fs) API to protect the mathematical integration data that is created during the high-frequency polling cycles. The technology creates and updates a localized energy blackbox all the time, working in a way that is similar to an airplane flight data recorder.written straight onto the Raspberry Pi 4's Secure Digital (SD) card is a JSON file.

An Edge Node writes the current E_{daily} value, the active local date key, and any pending historical arrays into this localized Black Box at the same time every 15 seconds when a power differential is noticed. If the Edge Node's WAN link to the Google Firestore cloud fails, it automatically switches to an Edge-Only mode of operation. Watching the farm, doing the integration math, and storing the data locally are all things that it continues to do. This makes sure that the system works absolutely independently, regardless of whether the internet is available outside.

4.2 Autonomous State Recovery and Asynchronous Reconciliation

When the Edge Node starts up or gets power back on after a localized facility power outage, it follows a strict one-way data reconciliation process. First, the Node.js backend reads the local energy_blackbox.json file to load the last known local state into the active memory heap. Only then

does it start any external database synchronization or accept remote requests. At the same time, it asks the Google Firestore database for the most recent properly synced cloud state.

Before activities start up again, the system uses a strict mathematical validation algorithm to make sure that there is no critical data regression, which is a fatal situation in which old cloud data overwrites the locally integrated offline data that was collected during an outage.

$$E_{final} = \max(E_{cloud}, E_{local})$$

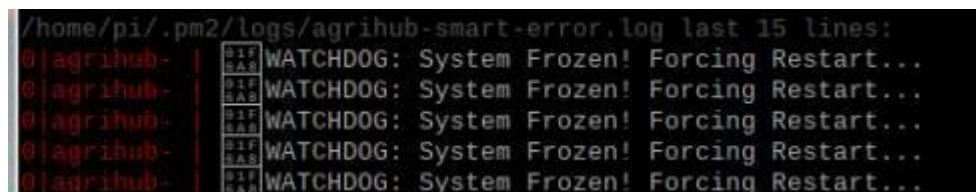
Because electrical energy use is strictly cumulative over 24 hours, this mathematical operation makes sure that the highest recorded energy state is always kept. This means that no data is lost after an outage, and offline operations and online analytics can work together without any problems.

4.3 Process Supervision and the Software Watchdog

Agricultural operations that go on 24 hours a day, seven days a week often cause "silent hangs" on local TCP network sockets. This is when the link to the physical Tuya smart plug drops data packets without throwing a fatal application error. To help with this, a two-layer automated Software Watchdog was created.

A UNIX timestamp variable called "lastGlobalActivity" keeps track of the exact moment of the last successful hardware data integration in the global event loop. This variable is checked every 60 seconds by a separate delayed timer. The watchdog finds a quiet socket failure and ends the main Node.js process using a hard process if the difference in time between the current system time and lastGlobalActivity is more than 15 minutes (900,000 milliseconds).command to leave.

When this is done on purpose, the PM2 industrial process planner running on the Raspberry Pi notices right away. It forces the corrupted memory heap to be flushed and starts a new, localized instance of the application. This makes sure that the edge design is strictly self-healing.



```
/home/pi/.pm2/logs/agrihub-smart-error.log last 15 lines:
0|agrihub- | 0.15 WATCHDOG: System Frozen! Forcing Restart...
0|agrihub- | 0.15 WATCHDOG: System Frozen! Forcing Restart...
0|agrihub- | 0.15 WATCHDOG: System Frozen! Forcing Restart...
0|agrihub- | 0.15 WATCHDOG: System Frozen! Forcing Restart...
0|agrihub- | 0.15 WATCHDOG: System Frozen! Forcing Restart...
```

Figure 3: WATCHDOG: System Frozen! Forcing Restart..

Figure 3 displays the system logs when the watchdog identifies a frozen thread and forcefully restarts the application.

4.4 Electrical Surge Mitigation via Staggered Boot Sequencing

In a big building, a master plan will often tell several 20A Master Air Conditioning units and hightorque fluid pumps to turn on at the same time. Sending a "ON" order to all hardware relays at the same time creates a huge instantaneous inrush current, also known as an electrical surge. This can damage physical electrical contacts quickly or accidentally trip circuit breakers in the central facility.

Within its global scheduling loop, the Edge Node uses an index-based Staggered Boot Sequence to protect the farming infrastructure. The system maps all offline, scheduled devices into an execution array when it sees that the current local time matches a schedule for action. It uses a mathematical delay function based on the array number instead of turning on the relays all at once: $delay = index \times 5000$

So, Device 1 turns on right away (*delay* = 0) Device 2 turns on 5,000 milliseconds later, and Device 3 turns on 10,000 milliseconds later. This regular spread over time flattens the facility's inrush current curve, which makes the hardware last a lot longer.

4.5 Intelligent Schedule Enforcement and Conflict Resolution

Human error is always a possibility when a remote, real-time program is added. On purpose, an operator could send a "OFF" signal to a crucial weather control system (like the main hydroponic nutrient pump) during required watering times.

To avoid interference from people, the Edge Node is the final local authority and strictly enforces local schedules over remote cloud orders. Within the Firestore WebSocket listener, when a request to change the state is received, the Edge Node checks the target state against the local `schedule_backup.json` setup window. Anytime a remote operator asks for a "OFF" state while the time at the local location is in a mandatory "ON" schedule block, the Edge Node automatically stops the command. This system knowingly rejects the remote request, keeps the hardware in the active "ON" state, and sends a corrective overwrite payload back to the Firestore database right away to reset the frontend user interface. Critical protocols for agricultural viability can't be overridden by remote application weaknesses thanks to this localized conflict resolution.

5. Database Optimization and Cloud Synchronization

The Edge Node takes care of hardware safety and real-time energy integration on its own, but the data needs to be synced with a central cloud database so that the operator's mobile screen can access it. Google Firestore is a widely distributed NoSQL cloud database that is used by the AgriHub architecture. When you use paid cloud databases, however, you have to be very careful with your money because platforms charge based on how many times you read, write, and delete documents.

5.1 Hierarchical NoSQL Document Schema

The Firestore database was set up with a tight hierarchical NoSQL schema to make queries as efficient as possible and keep data payload sizes as small as possible. The data is split into a root collection (power-monitor), a record for a specific facility (my-farm), and a very well organized subcollection for devices.

For example, tower 1 only keeps the most recent readings from the telemetry (latest readings) and settings for the user (settings.start_hour, settings.end_hour). It would be too hard to ask huge arrays just to show the main dashboard if historical energy accumulations were kept in the main document. Instead, chronological data is split up into a separate collection called `energy_history`. Each document ID is written strictly by the date key, like 2026-03-02. When showing historical bar charts, this partitioning approach lets the frontend Flutter app run shallow, low-cost queries.

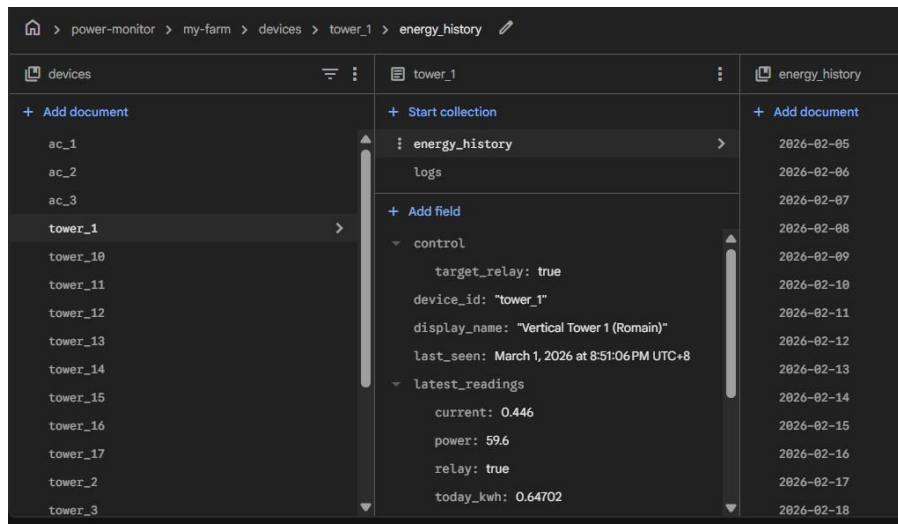


Figure 4: Firestore structure of towers with live data

Figure 4 shows the power monitoring firestore structure of each tower and live data and data being saved in `energy_history` for total daily energy consumption and data logs of each tower separately

6. Frontend Architecture and Live Dashboard (Flutter)

While the Edge Node takes care of hardware security, data integration, and offline persistence on its own, the facility operator only needs to use a cross-platform frontend dashboard to connect with the AgrhiHub system. The Flutter UI toolkit was used to build the frontend architecture, which was designed to favor rendering real-time telemetry, lower the cognitive load on operators, and allow instantaneous two-way control over the actual agricultural hardware.

6.1 Reactive State Management via WebSockets

In traditional industrial IoT screens, synchronous HTTP polling is often used. This means that the frontend application asks the database for new data at regular times. This operational delay causes too much latency in high-stakes farming settings.

Firebase WebSockets and Flutter's built-in `StreamBuilder` widgets are used in the application's dynamic state management design to make sure that data changes happen instantly. When the building manager opens the overview screen, the app sets up a continuous WebSocket connection to the relevant Firestore device documents. In milliseconds, the Firestore database sends the delta message down the WebSocket to the mobile or web client as soon as the Raspberry Pi Edge Node sends a new power reading or state change to the cloud. This reactive design causes the UI to automatically redraw, showing live voltage, current, and power, without the client having to manually update the interface or make synchronous API calls.

6.2 Industrial Glassmorphism and "Cyber-Navy" Theming

The interface doesn't use raw, clustered numerical tables but instead uses dynamic, high-contrast visual indicators to improve the user experience (UX) for site managers working in poorly lit agricultural control rooms. "Glassmorphism," an advanced rendering method built into Flutter through the `BackdropFilter` and `ImageFilter`, was used to make the user interface class blur. This makes frosted glass widget panels that are partly see-through and change color with the background, creating a clear visual order.

The app has a specific Dark Mode setting, which is called the "Cyber-Navy" style internally. A global `ValueListenableBuilder` was used to allow this, and the `ThemeData` was changed dynamically across the entire widget tree. Dark Mode settings (using deep #0F172 hexadecimal backgrounds) perform a

strictly physiological function: they reduce eye strain for operators during long tracking sessions and raise the contrast ratio for important hardware status indicators.

6.3 Dynamic Visual Indicators and "Breathing" Animations

When IoT is used in industry, workers must be able to quickly see how the whole farm's electricity is working. In order to make this possible, the front-end app adds a `TickerProviderStateMixin` and an `AnimationController` to the device card widget tree.

If the Edge Node sees that a real Tuya plug is ONLINE, the animation controller gives the device card a continuous, cyclic expanding drop-shadow. This adds a soft cyan or green neon "breathing" effect to the Glassmorphism background, which is already dark. If a device goes offline because of a limited hardware failure or power outage, the animation stops mathematically and the card's color changes to a flat, dim red. This programmatic link between the health of the actual hardware and the frontend visual animation makes it much faster for an operator to find and shut down a pump or AC unit that isn't working on the farm grid.

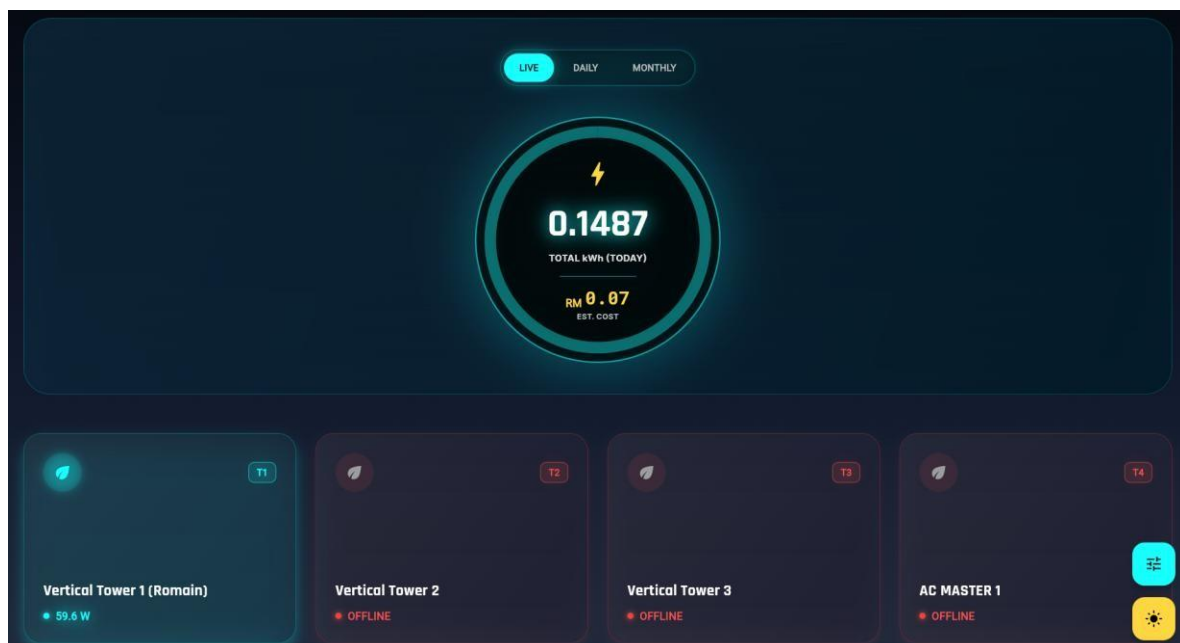


Figure 5: Glowing cards in Dark Mode

In Figure 5 The main Flutter Overview Dashboard rendered in the "Cyber-Navy" Dark Mode, illustrating the Glassmorphism UI architecture, real-time aggregated power gauges, and the dynamic "breathing" status cards.

7. Interactive UI Capabilities and Data Management

A commercial-grade IoT dashboard must have strong, interactive data management tools for site managers in addition to showing real-time telemetry. As an agricultural facility grows, the sheer number of computing nodes that are actively using can cause serious cognitive overload. Real estate or reusing of physical hardware parts is common across different growing zones, so the database schema needs to be flexible so that operators can make changes straight from the user interface.

7.1 Dynamic Device Filtering and State Allocation

A dynamic combinatorial filtering technique is used by the application to reduce visual clutter and help operators focus on certain environmental factors. The program sets up a memory-efficient Hash Set (`_selectedIds`) with the unique string identifiers of all active farm nodes in the frontend state layer.

When the user calls the `_showFilterDialog` interface, the application shows a Boolean list with all the items in the facility that can be selected. If an operator deselects a certain AC unit or Vertical Tower, the local state management engine (`setState`) deletes that ID from the current hash set right away. The Flutter framework then rebuilds the main `GridView` widget in a way that makes it run faster and better. The app runs a conditional check (`_selectedIds.contains(id)`) during this rebuild step. If the condition is false, the device card is automatically taken out of the rendering process. This dynamic filtering method not only cleans up the user interface, but it also saves a lot of memory on the device by stopping animation controllers that aren't needed for hidden hardware nodes.

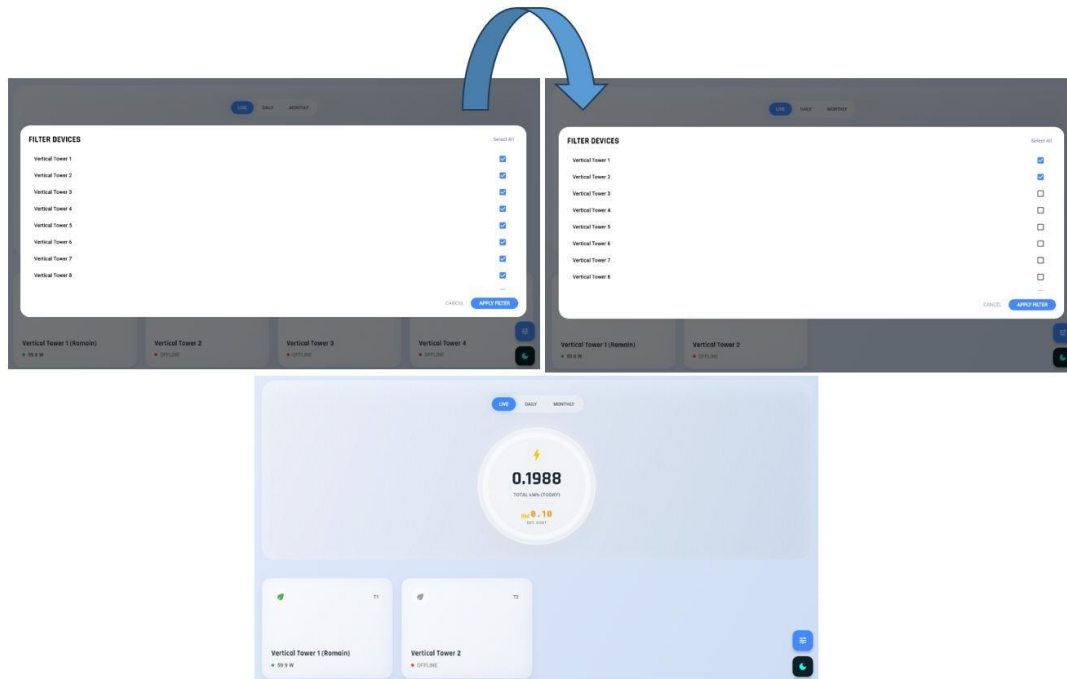


Figure 6: Device Filter Dialog

Figure 6 shows The Flutter application's Device Filter Dialog, demonstrating the Boolean state selection mechanism utilized to dynamically isolate specific Vertical Towers and Air Conditioning units on the main dashboard.

7.2 Bidirectional Database Mutability and Secure Renaming

Older IoT dashboards can only send data in one way, so they are basically read-only terminals. The AgriHub architecture is the first to use a bidirectional mutability model, which lets operators dynamically change the names of hardware straight from the frontend interface. This is because CEA hardware is often moved between different hydroponic zones.

There is a `GestureDetector` built into the device card widget tree that makes this interaction easier. By holding down the `onLongPress` event for a while, the `_showRenameDialog` interface is displayed using `Glassmorphism`, which lets users enter a new semantic label (for example, changing "Vertical Tower 1" to "Tomato Zone A").

Direct write actions to the global `Firestore` database, on the other hand, open up a major security hole. To keep changes from being made without permission, the protocol requires a cryptographic `Soft-Lock` layer. This layer requires the user to enter a predefined `Administrative Key`, which acts as a chosen password, into a hidden `TextEditingController`. The Flutter app sends an asynchronous `delta` payload after approval using the `SetOptions(merge: true)` directive. This `clear merge` command only changes the `display_name` string field, which protects the security of important scheduling and telemetry objects in the `NoSQL` file. Lastly, the `Firestore WebSocket` listener sends this semantic change right away to all live dashboard instances across the whole building.

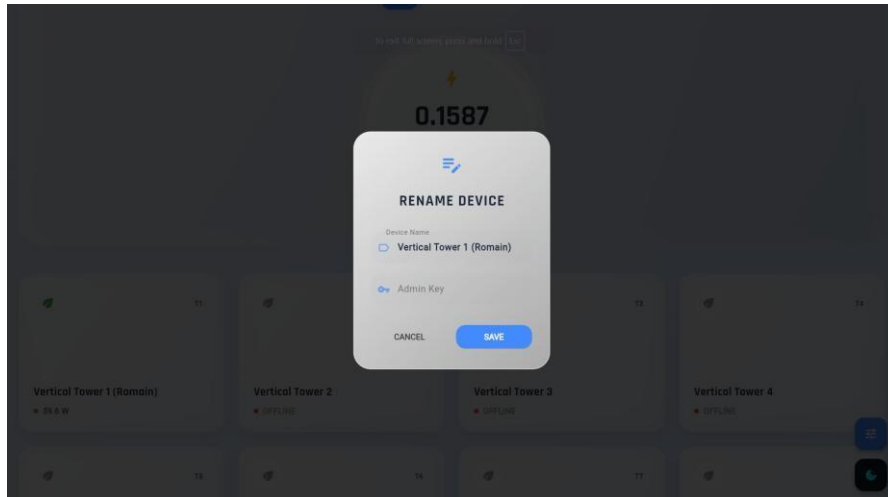


Figure 7: Rename Devices

Figure 7 shows The secure Rename Device Dialog interface, illustrating the bidirectional database mutation capabilities and the Administrative Key requirement ("Soft-Lock") preventing unauthorized nomenclature changes.

8. Temporal Data Analytics and Visualization

The APU AGRHI Hub is immediately safe to use because of real-time hardware control, but it will be profitable in the long run thanks to detailed energy data. To make this possible, the frontend Flutter app has a separate, multi-tabbed interface for each hardware node. It uses the `fl_chart` rendering library to turn raw NoSQL documents into actionable visual information.

8.1 Live Telemetry and Financial Extrapolation

The "Monitor" interface separates the Edge Node's high-frequency telemetry streams from other traffic. The app uses Glassmorphism metric cards to show immediate alternating current (AC) metrics, such as Voltage (V), Current (A), and active Power (W). The application does a live scalar extrapolation on the raw daily Kilowatt-hour (kWh) accumulation to give it instant business context. The system instantly gives an Estimated Cost value by multiplying the real-time E_{today} variable by the Malaysian business utility tariff, such as RM 0.50 per kWh:

$$Cost_{est} = E_{today} \times 0.50$$

This continuous extrapolation of finances lets facility managers quickly check the daily working costs of certain high-load pumps and AC units, without having to use an outside spreadsheet.

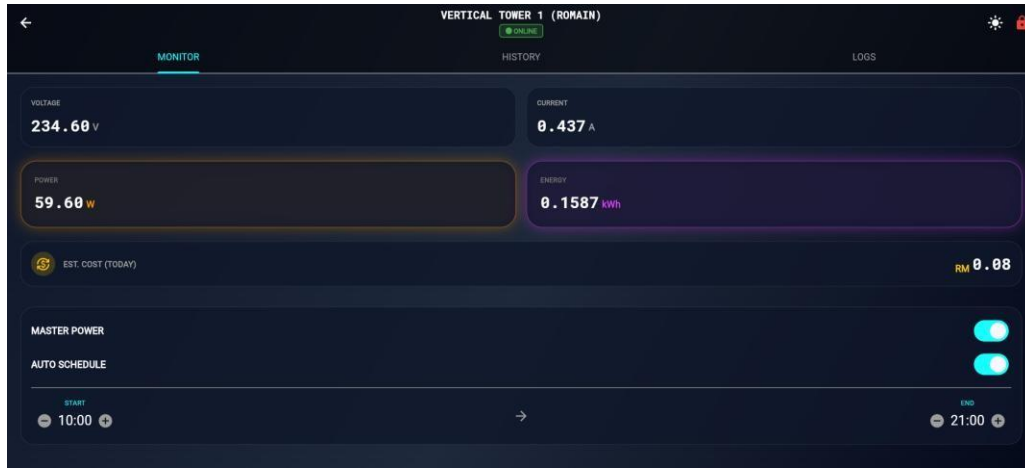


Figure 8: Device monitor interface

Device 8 shows The Device Monitor interface illustrating the isolated real-time telemetry metrics (Voltage, Current, Power) and the instantaneous Estimated Cost extrapolation in Ringgit Malaysia (RM).

8.2 Temporal Data Aggregation and the Zero-Fill Algorithm

Visualizing past IoT data is a difficult programming problem. For daily, monthly, and yearly consumption trends analysis, the design uses chronological temporal plotting. With the Edge Node, however, working on an Offline-First model, it's possible for a physical device to be turned off or removed for 24 hours. Under these conditions, the Firestore energy history subcollection does not create a daily record.

If the app sent the raw database query straight to a common charting tool, the X-axis would collapse because of the missing chronological keys, which would mess up the timeline and show wrong visual data. Before rendering the chart, the application uses a tight "Zero-Fill" chronological algorithm to fix this structural problem's

When the user chooses a past month, the app uses the `DateUtils.getDaysInMonth` algorithm to create a full, unchangeable array of dates that represent all the valid dates in that month (for example, an array with an index length of 31 for January). After that, the program compares the Firestore data it got with this strict base array. In the event that there is no date key in the database payload, the algorithm ensures a continuous timeline by adding a value of 0 to that index:

$$Value_{index} = \begin{cases} E_{cloud} & \text{if document exists} \\ 0.0 & \text{if document is null} \end{cases}$$

This makes sure that the days with no usage are shown correctly in the Bar Chart as empty spaces on the timeline, keeping the accuracy of the time.



Figure 9: Daily History

In figure 9, The Daily History interface featuring the Zero-Fill algorithmic Bar Chart, dynamically preventing temporal distortion during days of hardware inactivity.

8.3 Dynamic Metric Conversion and Adaptive Tooltips

A dynamic Metric Conversion Toggle is built into the History interface to help people who work with both raw engineering metrics and financial reporting. The operator can quickly recalculate the whole historical array from raw energy (kWh) to local currency (RM) has shown in figure 10, which is linked to a local Boolean state variable called `_showCost`.



Figure 10: energy (kWh) to currency(RM)

When this option is turned on or off, the app goes through the saved historical array, applies the scalar multiplier, and makes the UI redraw fluidly. Dynamic boundary constraints (`fitInsideVertically: true` and `fitInsideHorizontally: true`) were built into the Bar Chart's tooltip rendering engine to handle the quick change in the number volume and keep the data labels from looking cut off. During extreme power spikes, this makes sure that floating-point values created stay tightly within the physical boundaries of the mobile or web viewport. Also, the interface lets users switch from a Daily Bar Chart to a smoothed Yearly Line Chart, which shows the total of all the monthly data over a full yearly cycle, for a bigger picture look at trends.

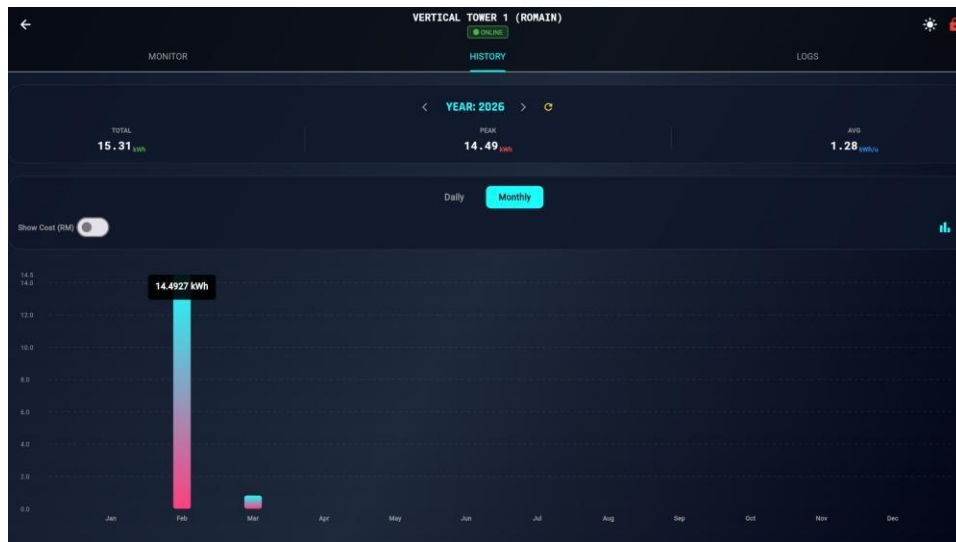


Figure 11: Yearly History interface

Figure 11 shows the Yearly History interface rendering macroscopic consumption trends via a smoothed Line Chart, demonstrating the real-time financial metric conversion toggle (kWh to RM).

9. Operator Security and System Auditing

The commercial-grade farm IoT screen connects straight to high-voltage physical actuators. Therefore, catastrophic facility losses can happen due to illegal access or human mistake, like turning off a main hydroponic nutrient pump during a critical growth phase. To strengthen these weaknesses, the frontend design includes a strict cryptographic interaction protocol and an all-encompassing, self-running tracking engine.

9.1 Cryptographic Soft-Lock and Actuator Security

The Flutter app leaves all important interactive parts of the Device Monitor interface turned off and read-only by default. These parts include the Master Power bypass switch and the Automated Schedule Time Adjusters. This stops unwanted touchscreen movements from activating the device by accident.

The operator must purposely call the `_toggleLock` function in order to change the facility's current working settings. This opens a safe login dialog box that needs an Administrative Key that has already been set (for example, "2435"). The local state management engine changes the `_isLocked` Boolean variable to false only after the string checking works. This Soft-Lock Protocol is a two-layer security system that makes sure that only approved facility managers can send physical state-change packets to the Edge Node. The dashboard can be used by anyone on the farm for passive visual data audits.

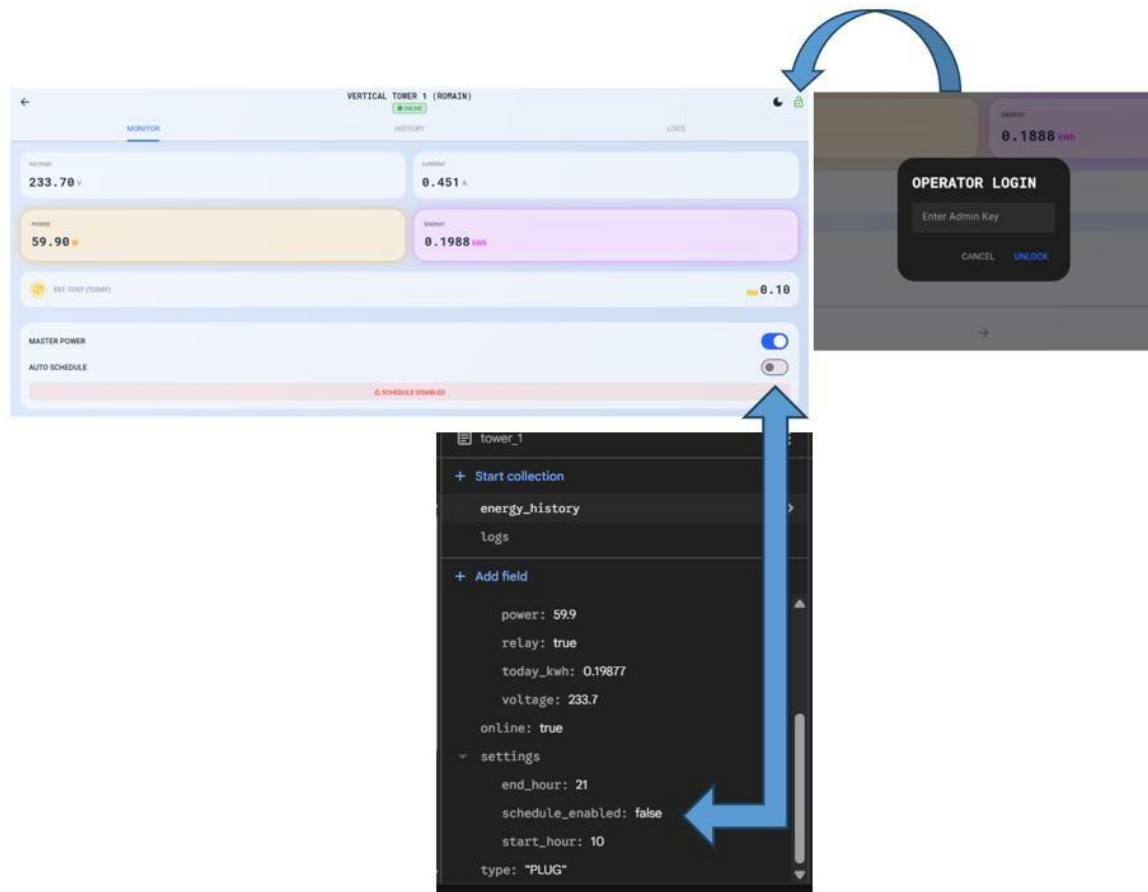


Figure 12: Soft-Lock Security Protocol

In figure 12, live status can be seen on firestore of the schedule below is button is offline after inserting the authorized password..

9.2 Autonomous System Auditing and Log Segregation

The ability to be tracked is a must for industrial IoT deployments. The Raspberry Pi Edge Node constantly saves operational receipts, like when a plan is executed, the power goes out in a certain area, or a user changes an action, to a special collection of logs in Firestore. The Flutter UI has a complex Log Parsing Engine that turns this raw NoSQL array into information that can be used.

When the monitoring interface is shown, the program gets the chronologically ordered log array and sorts the documents into two separate visual streams, which it calls "User Actions" and "System Events." Using strict string classification logic, any file that is made by a person using the UI (like MANUAL_ON or MANUAL_OFF) is sent to the left column. On the other hand, independent changes that are carried out by the Edge Node's internal logic (for example, SCHEDULE: Auto-OFF or SAFETY: Forced ON) are sent to the right. Because of this tight categorical segregation, facility engineers can quickly tell if a hardware problem was caused by a program or by a human user override.

9.3 Reverse Temporal Calculus for Power-Cut Visualization

The most advanced analytical feature of the monitoring system is its ability to automatically see where the grid is failing. The Edge Node figures out how long it was offline and writes a localized string message with the format "Power Cut for: Xh Ym Zs" when it gets online again after an ISP or power loss.

This string list is picked up by the Flutter frontend, which runs a Regular Expression (RegExp) parsing method on it. The program uses a method to get separate numbers for hours, minutes, and seconds, which are then used to build a native Duration object in the active memory heap. The application does a reverse-subtraction against the UNIX timestamp of the log generation to give users accurate tactical intelligence:

$$t_{start} = t_{current} - \text{Duration}$$

Using this backwards math of time, the user interface creates a custom Time Range Box on the fly. This part clearly shows the exact time the facility blackout started and ended, fully automating the process of checking for faults in the facility.

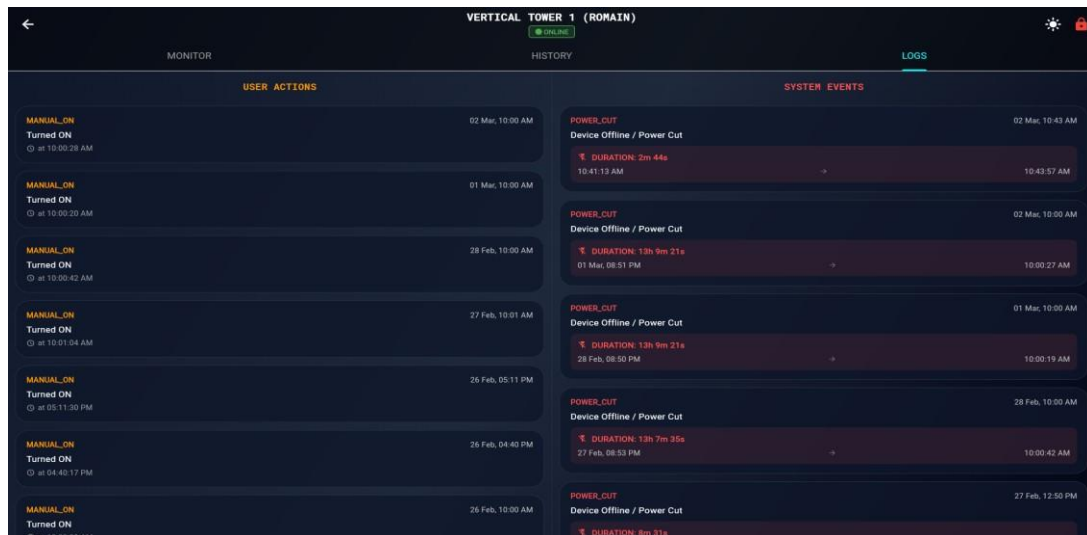


Figure 13: System Auditing interface

Figure 13 shows the The System Auditing interface demonstrating the strict visual segregation of manual User Actions versus autonomous System Events, alongside the programmatic reverse-time calculation of hardware power cuts.

10. Results and Performance Evaluation

APU AGRHI Hub's Edge-to-Cloud design was put through thorough performance and fault-tolerance tests to prove its business scale and infrastructure robustness.

10.1 Edge Node Computational Efficiency

Hardware resource loss is a major worry when moving computations from centralized cloud servers to localized edge points. The PM2 industrial process manager on the Raspberry Pi 4 Model B was used to keep an eye on how the system's resources were being used to test how well the Node.js mathematical integration engine worked.

Even though it kept high-frequency (15-second) TCP socket connections with several 20A Tuya smart switches and ran dynamic Riemann sum calculus for energy auditing at the same time, the Node.js application had a very small footprint. The active process used only 19.9 Megabytes (MB) of variable memory (RAM) on average and practically no CPU power at all during the whole time it was running. The real-world result shows that the software design is highly efficient. This means that the microcomputer produces very little heat and has a lot of computing power to expand to hundreds of smart plugs without having to buy new hardware.

10.2 Cloud Optimization and Commercial Viability

The architecture's ability to make money was checked by looking at the Google Firestore database activity logs. Before the Asynchronous Quota Optimization method was put in place, a normal 15second polling delay caused 5,760 writes per device every day, which could not be sustained.

After the 10-minute localized memory buffering procedure was put in place, the system was able to turn these high-frequency data spikes into tightly controlled packages. The database waste was cut down to exactly 144 writes per device, per day, as shown by the transaction logs. This real-world 97.5% drop in cloud transaction volume mathematically ensures that the facility will always be below Google's 20,000 free daily write limit, making it possible to have ongoing cloud infrastructure at no cost.

10.3 Fault-Tolerance and Watchdog Validation

Controlled software and network outages proved that the "Offline-First" restore methods worked. To make it look like a quiet connect failure, the Node.js app was intentionally blocked on a thread, which set off the lastGlobalActivity timeout setting. The automated Software Watchdog found the thread that was stuck and started a process.exit(1) ends the program. It took less than five seconds for the PM2 supervisor to find the exit code and start a new copy of the program. After restarting, the system was able to read the localized energy_blackbox.json file from the SD card and use the Math.max method to match the offline energy measures with the Firestore database, so there was no overall data loss.

11. Conclusion

The infrastructure that is required for the modernization of controlled environment agriculture must place operational resilience at the top of the list of priorities, above all other criteria. The purpose of this work is to effectively show the design, implementation, and validation of an Internet of Things (IoT) power monitoring architecture that is fault-tolerant and Edge-to-Cloud specifically designed for the APU AGRHI Hub. By moving the primary computational logic and mathematical energy integration away from centralized commercial servers and onto a localized Raspberry Pi Edge Node, the system was able to completely eliminate geographical network latency and protect the facility from catastrophic data loss in the event of an Internet service provider (ISP) outage. Absolute hardware autonomy is ensured by the configuration of a programmed Software Watchdog in conjunction with an Offline-First local "Black Box" memory module. Furthermore, the Asynchronous Quota Optimization algorithm successfully reduced cloud database write operations by over 97%, ensuring long-term commercial scalability without recurring server costs. finally, the construction of a reactive dashboard for Flutter that is based on Glassmorphism offers facility operators an analytical interface that is not only mathematically correct but also understandable and secure. The Edge-to-Cloud paradigm that was developed as a consequence makes it possible to construct a standard that is both extremely durable and scalable for future smart farming installations.

References

1. Wu, M., Lu, T.-J., Ling, F.-Y., Sun, J., & Du, H.-Y. (2010). Research on the architecture of Internet of Things. *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*. <https://doi.org/10.1109/icacte.2010.5579493>
2. Ray, P. P. (2017). Internet of things for smart agriculture: Technologies, practices and future direction. *Journal of Ambient Intelligence and Smart Environments*, 9(4), 395–420. <https://doi.org/10.3233/ais-170440>
3. Elijah, O., Rahman, T. A., Orikumhi, I., Leow, C. Y., & Hindia, M. N. (2018). An Overview of Internet of Things (IoT) and Data Analytics in Agriculture: Benefits and Challenges. *IEEE Internet of Things Journal*, 5(5), 3758–3773. <https://doi.org/10.1109/jiot.2018.2844296>
4. Al-Sarawi, S., Anbar, M., Alieyan, K., & Alzubaidi, M. (2017, May 1). *Internet of Things (IoT) communication protocols: Review*. IEEE Xplore. <https://doi.org/10.1109/ICITECH.2017.8079928>
5. Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5), 637–646. <https://doi.org/10.1109/jiot.2016.2579198>