Reverse Engineering a Bluetooth Low Energy Thermal Printer for Cloud-Connected Production IoT

Md Raisul Amin Hasib[1]

[1]Independent Researcher , raisulaminhasib@tiangong.edu.cn

July 13, 2026

Abstract

Low-cost consumer thermal printers are usually locked to a vendor mobile application that speaks an undocumented Bluetooth Low Energy (BLE) protocol. That arrangement leaves the hardware far less programmable than it could be, and rules out any integration into a larger system. This paper reports an end-to-end engineering case study in which a commodity 384-pixel BLE thermal printer is reverse engineered at the protocol level and then grown, in stages, into a globally reachable serverless Internet-of-Things (IoT) service. The work has three phases. In the first, the proprietary BLE command protocol is recovered through packet inspection and comparison against a prior community effort, producing a documented packet layout, a command set, a table-based CRC-8 integrity check, and a bitmap encoding. In the second, an always-on ESP32 microcontroller is added as a BLE-to-cloud bridge. An early design based on HTTP polling collapses under the microcontroller's memory limits, so the system is rebuilt around the lightweight MQTT publish/subscribe protocol, which brings end-to-end print latency below 100 ms. A serverless cloud stack, built from a Next.js dashboard, stateless API routes, a serverless PostgreSQL database, and an external cron trigger, supplies scheduling and remote control. The third phase examines failures that appeared only after the system had run continuously for some time. The most serious was database quota exhaustion, caused by aggressive polling combined with a poor choice of storage tier. Fixing it meant replacing polling with server-pushed events, moving ephemeral device state into a key-value store, letting the common online-print path skip the relational database, and closing a silent MQTT buffer-truncation bug. Together these changes cut monthly database compute from roughly 200 to 1–2 compute-unit hours, and idle API traffic from about 1,200 to under 50 calls per hour. The study draws out lessons on protocol reverse engineering, resource-aware protocol choice for constrained devices, and storage-tier matching in low-traffic serverless IoT systems.

Keywords: Internet of Things, Bluetooth Low Energy, reverse engineering, MQTT, embedded systems, ESP32, serverless architecture, thermal printing, edge-to-cloud.

1 Introduction

Consumer thermal printers are now cheap and everywhere, usually sold as pocket companions for a smartphone. Devices in this class rarely ship with any documented programming interface. The only intended way to drive them is a vendor mobile application that talks to the printer over an undocumented Bluetooth Low Energy (BLE) link. Such a closed arrangement pins the device to whatever features the vendor chose to expose, blocks any use from desktop or server software, and leaves no room for automation.

This paper documents a system that removes those restrictions for one representative device, an X6h-class 384-pixel BLE thermal printer, and sets out a repeatable path for turning a closed BLE peripheral into a programmable, cloud-connected IoT service. The contribution is an engineering case study rather than a new algorithm. What it offers is the specific run of design decisions, the failures that turned up along the way, and the fixes each failure forced once the system met real use. The deployed system remains publicly reachable, and the firmware and web application are the basis for the concrete details reported throughout.

The work moved through three phases, and each one motivated the next.

1. **Protocol reverse engineering.** The proprietary BLE command protocol was recovered by inspecting captured packets, spotting the structural constants (a fixed header and footer, a command byte, a length field, and a checksum), confirming a table-based CRC-8 integrity scheme, and locating the correct GATT service and characteristic UUIDs. A desktop control application then validated the recovered protocol.

2. **Cloud and IoT architecture.** An ESP32 microcontroller filled the bridge role: always online, sitting between the BLE peripheral and the public internet. A first attempt based on periodic HTTP polling proved unworkable on the constrained device, so the system was rebuilt around MQTT and backed by a serverless cloud stack offering a web dashboard, remote printing, and time-zone-aware scheduling.

3. **Production hardening and failure analysis.** After the system looked finished, sustained operation surfaced a cluster of resource and correctness problems: database quota exhaustion, wasteful polling, mismatched storage tiers, a silent message-truncation bug, and a missing authentication check. Each is analyzed and fixed, with before-and-after measurements.

The contributions of this work are:

- A documented, reproducible reverse-engineering of a commodity BLE thermal-printer protocol, including its packet layout, command set, table-based CRC-8 integrity scheme, GATT identifiers, and 1-bit bitmap encoding.
- An edge-to-cloud architecture in which the choice of edge transport is driven by microcontroller memory rather than throughput, together with a concrete account of why an HTTP-polling design fails and an MQTT publish/subscribe design succeeds on the same hardware.
- A failure analysis of the deployed system under sustained real-world use, with quantified remediations (server push in place of polling, storage-tier matching, hot-path avoidance of the relational database, and elimination of a silent message-truncation defect) that reduce monthly database compute and idle API traffic by one to two orders of magnitude.
- A set of transferable lessons on resource-aware protocol selection for constrained devices and storage-tier matching in low-traffic serverless IoT systems.

The remainder of the paper proceeds as follows. Section 2 reviews related work. Section 3 gives a system overview. Section 4 covers the hardware and protocol reverse engineering. Section 5 presents the cloud and IoT architecture, and Section 6 the implementation. Section 7 analyzes the production failures and their fixes. Section 8 discusses lessons that generalize beyond this project, and Section 9 concludes.

2 Related Work

Reverse engineering of BLE peripherals. Recovering an undocumented BLE protocol is a familiar exercise in the hobbyist and security communities. One widely cited community write-up documents the reverse engineering of a "cat"-themed thermal printer that shares much of its protocol structure with the device studied here, down to the `0x5178` packet header and the command-oriented framing [1]. The present work adopts the same style of methodology and then carries it further, toward a full production IoT deployment rather than stopping at local control. Broader literature on BLE security and protocol analysis supplies the general techniques applied here, including GATT enumeration, characteristic probing, and packet differencing [2, 3].

Messaging protocols for constrained devices. MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe protocol built for low-bandwidth, resource-limited, or high-latency networks; OASIS maintains the standard [4]. Studies comparing MQTT with request/response protocols such as HTTP tend to report lower overhead and latency for telemetry and command workloads on embedded hardware [5]. The architectural pivot in Section 5 is a concrete case of that trade-off, though here it was decided by device memory rather than by throughput.

Serverless and edge-to-cloud IoT. Serverless (Function-as-a-Service) platforms are popular for IoT back-ends because they remove idle server cost and scale down to zero, at the price of statelessness and a per-invocation billing model [6]. The failure analysis in Section 7 illustrates a consequence of that model which is easy to overlook: because the functions are stateless and billed by activity, naive polling and a mismatched persistence choice turn straight into runaway cost. Matching the storage tier to the data, that is, keeping durable relational records apart from ephemeral device telemetry in an in-memory key-value store such as Redis [7], is a recurring recommendation in the IoT systems literature, and this study confirms it with measurements.

3 System Overview

The finished system spans four tiers, from a browser client down to the physical printer, joined by a serverless cloud and an edge microcontroller bridge. A print request begins at the web dashboard, submitted by a user. A stateless API picks it up and, whenever the device is online and no backlog exists, publishes the job straight to an MQTT broker. The broker pushes the command to a subscribed ESP32 bridge, which renders the payload into a bitmap and sends it to the printer over the reverse-engineered BLE protocol. Ephemeral device state lives in an in-memory key-value store, durable data such as schedules and queued jobs lives in a serverless relational database, and the dashboard learns of changes through a server-push channel instead of by polling. Figure 1 shows the arrangement.

Table 1 breaks down the deployed system's technology choices by layer.

4 Hardware and Protocol Reverse Engineering

4.1 Objective and approach

The first goal was modest: drive the printer directly from a personal computer and retire the vendor mobile application. Since the device talks only over BLE with an undocumented protocol, the protocol itself had to be recovered before anything else. Python was chosen for

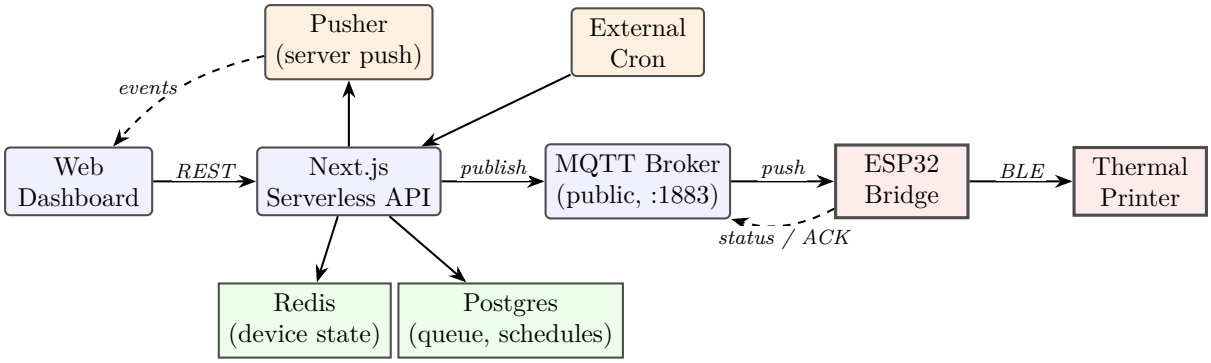

Figure 1: End-to-end architecture of the cloud-connected thermal printing system. Solid arrows are the command path; dashed arrows are asynchronous status and server-push traffic. Print commands reach the edge bridge over an MQTT publish/subscribe channel, while status updates return to the dashboard through a separate push channel. Durable and ephemeral state are held in separate, purpose-matched stores.

Table 1: Technology stack of the deployed system, by layer.

Layer	**Technology**
Frontend	Next.js 15, React 19, TypeScript, Tailwind CSS, shadcn/ui
Backend	Vercel serverless functions; NextAuth.js; bcrypt
Database	Serverless PostgreSQL (Neon); Prisma / `@vercel/postgres`
Cache	Serverless Redis (Upstash), ephemeral device state
Messaging	MQTT over TCP (public broker, plaintext, port 1883)
Realtime	Server-push channel (Pusher Channels)
External data	Weather API (OpenWeather), cached
Time zones	Luxon date/time library
Scheduling	External cron trigger (cron-job.org)
Edge firmware	Arduino/C++ on ESP32; `PubSubClient`, `ArduinoJson`, custom BLE
Hardware	ESP32 DevKit; 384-pixel BLE thermal printer

prototyping, mainly because the `bleak` cross-platform BLE library [2] allows fast experimentation with no compilation step. Because its advertised BLE name had already shown up in the vendor application, the device was easy to spot during scanning.

4.2 Packet structure recovery

Analysis started from a handful of captured command packets. Two of them, shown in hexadecimal, are representative:

```
51 78 A2 00 30 00 [48 bytes of data] C7 FF
51 78 A1 00 02 00 03 00 8A FF
```

Listing 1: Representative captured BLE command packets.

Comparing packets exposed a stable set of structural constants:

- Every packet opens with the two-byte header `0x51 0x78`.
- Every packet ends with the footer byte `0xFF`.
- The third byte is a *command* selector; `0xA2` prints a bitmap and `0xA1` feeds paper, for example.

- A length field records the size of the data payload.
- The byte just before the footer is a checksum over the payload.

From these observations the general packet layout in Figure 2 follows.

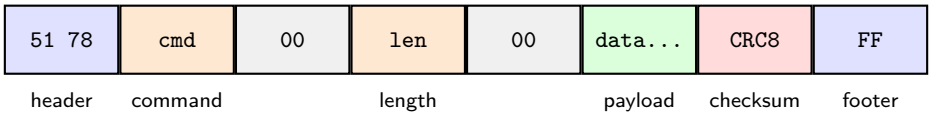

Figure 2: Recovered packet structure of the proprietary BLE thermal-printer protocol. Header and footer are fixed, the command byte selects the operation, the length field bounds the payload, and a single CRC-8 byte protects payload integrity. The deployed firmware confirms this layout: every command it sends is framed as `0x51 0x78`, command, `0x00`, length, `0x00`, payload, CRC-8, `0xFF`.

4.3 Integrity scheme and command set

The byte ahead of the footer turned out to be a CRC-8 value computed over the data payload, a common integrity mechanism in hardware protocols. The firmware implements it as a 256-entry table lookup with polynomial `0x07` (the CRC-8/SMBus variant), accumulating over the payload bytes. Getting that computation right was the most delicate part of the whole protocol. The printer *silently drops* any command whose checksum is wrong, giving no error at all, which makes a faulty implementation hard to diagnose because the device simply does nothing.

Table 2 summarizes the command bytes recovered and in active use.

Table 2: Recovered command byte set for the BLE thermal printer.

Command	Function	Notes
`0xA1`	Paper feed	Advances paper without printing
`0xA2`	Print bitmap	Sends one 48-byte line of image data
`0xA4`	Set quality	Print resolution level; firmware uses 3
`0xAF`	Set energy	Print darkness; firmware uses `0x10 0x00`
`0xBE`	Drawing mode	Selects bitmap drawing mode (`0x00`)

On connection the firmware issues a fixed initialization sequence, setting energy (`0xAF` with `0x10 0x00`), quality (`0xA4` with `3`), and drawing mode (`0xBE` with `0x00`), each followed by a short delay.

4.4 GATT service and characteristic discovery

An early mistake highlights a common BLE trap. With the packet format and checksum believed correct, the first commands still drew no response. The problem was that they were being written to the wrong GATT characteristic. Enumerating the device's GATT table located the correct write characteristic. In the deployed firmware the printer is reached at a fixed BLE address, its service is resolved under UUID `0000AE30-0000-1000-8000-00805F9B34FB`, and commands are written to characteristic `0000AE01-0000-1000-8000-00805F9B34FB`. Once commands went to the right characteristic, a paper-feed command produced the first visible physical response and confirmed the recovered protocol end to end.

4.5 Bitmap encoding

Printers of this class take 1-bit monochrome raster data, not text and not grayscale. On the target device each printed line is 384 pixels wide, so a single line is packed into exactly 48 bytes (384/8) and sent with command `0xA2`. Pixels are packed eight to a byte, and correct output depends on matching the bit ordering exactly; any mistake in the packing produces visibly corrupted prints. The firmware transmits a page in small batches of ten scan lines, yielding to the runtime between batches so that the network and BLE stacks are not starved during a long print. To make photographs look acceptable on a strictly two-level device, Floyd–Steinberg error-diffusion dithering was applied in the desktop application, which approximates grayscale using spatial patterns of black and white dots [8].

4.6 Thermal pacing

Testing revealed a physical constraint of the print head. Send successive lines too quickly and the head overheats, and the output fades. A configurable inter-line delay fixed the problem and protects the hardware, and it is kept as a tunable parameter in every later implementation.

5 Cloud and IoT Architecture

5.1 Motivation

The validated desktop application ended the dependence on the vendor mobile app, but two limitations remained. It needed the controlling computer to be present and powered on, and it offered no unattended automation, such as printing a daily weather summary on its own. Fixing both problems meant putting something permanently online between the public internet and the BLE-only printer.

5.2 Edge bridge selection

The ESP32 microcontroller [9] was chosen for the bridge role because it carries both Wi-Fi and BLE radios, costs little, and can stay powered continuously. Its job is to terminate a cloud connection over Wi-Fi and relay commands to the printer over BLE.

5.3 First design: HTTP polling and its failure

The first cloud design had the ESP32 poll a REST endpoint every few seconds, asking the server whether any print jobs were waiting and fetching the content when one was. It worked in testing, then failed under sustained use, and for a resource reason rather than a logical one. The ESP32 has little RAM, and running a secure HTTPS client at the same time as the BLE stack exhausted the heap. The TLS handshake alone consumed most of the free memory, and once the BLE stack was added on top the device dropped into repeated out-of-memory reboots. In short, the request/response model did not fit the device's memory budget, regardless of how fast or slow it was. A vestige of this early design survives in the firmware as a disabled HTTP command-poll routine, retained only as a fallback and superseded entirely by MQTT.

5.4 Second design: MQTT publish/subscribe

The system was rebuilt around MQTT [4], a publish/subscribe protocol meant for constrained devices. Instead of repeatedly opening costly secure request/response connections, the ESP32 now holds a single persistent connection to a public MQTT broker over plain TCP and subscribes to a per-device command topic (`esp32stp/<device-id>/commands`), while publishing status and acknowledgments on sibling topics. The broker pushes messages down to the device as

they arrive. That removes both the per-request TLS handshake and the polling loop, and it fits inside the device's memory budget. Figure 3 shows the resulting command flow.

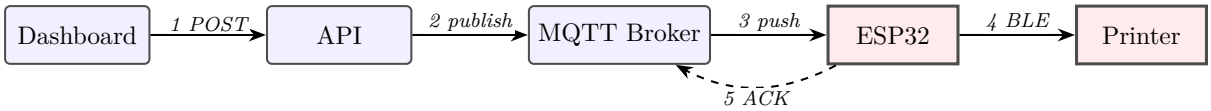

Figure 3: Manual print command flow under the MQTT architecture. A persistent broker connection replaces per-request HTTP polling, so repeated TLS handshakes disappear and end-to-end latency falls below 100 ms. Step 5 returns an acknowledgment on a separate topic.

After the migration, print latency fell below 100 ms, memory use stabilized, and the reboots stopped. This pivot is the central decision of the second phase. Protocol choice for a constrained edge device was governed by memory footprint, not by throughput or convenience.

5.5 Serverless cloud stack

A serverless back-end was chosen to keep operating cost low at small traffic volumes. The frontend is a Next.js application whose dashboard lets a user compose text, pick a font size, and print, on desktop or mobile alike. The back-end itself is just a few stateless API routes, covering print commands, schedules, and device status. On a manual print the API reaches out to an external weather service, then turns the result into a message and pushes it to the broker. Access is gated by session-based authentication, and every print-related route is protected by IP-based rate limiting (for example, on the order of 60 print requests, 20 weather lookups, and 120 queue operations per minute).

5.6 Persistence and scheduling

Serverless functions are stateless, so anything that needs to survive between invocations has to live in external storage. A serverless PostgreSQL database (Neon) was adopted at first to persist schedules, each with a target time, a time zone, active days of the week, a weather-inclusion flag, and font-size preferences, alongside a priority-ordered print queue.

Automated scheduling raised a further problem. The hosting platform's free tier has no cron jobs. An external scheduling service (cron-job.org) was therefore set to call protected API endpoints periodically, with a shared bearer secret for authentication. On each call the schedule endpoint reads every active schedule and, for each one, checks whether it is enabled, whether the current time in the schedule's own time zone matches its target, and whether it has already run that day. Once all three are met, the endpoint grabs weather data, puts together a message suited to the time of day, then sends it on to MQTT. Correct time-zone conversion, so that 07:00 local fires at the right instant in every region, was delegated to the Luxon date/time library. Figure 4 summarizes the scheduled path.

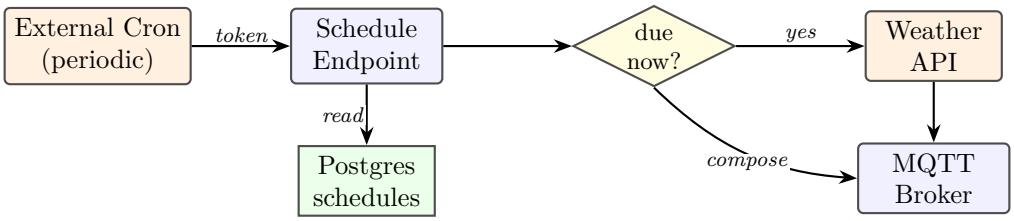

Figure 4: Scheduled print flow. An external cron service triggers a protected API endpoint; the endpoint reads the schedule store, checks time-zone-aware and once-per-day conditions, fetches weather when required, and publishes any matching job to the MQTT broker.

6 Implementation Details

6.1 Desktop control application

The desktop application was written in Python and split into three loosely coupled layers so that each could be built and tested on its own:

- a *GUI layer* (Tkinter) for user interaction and display;
- an *image-processing layer* (PIL and Matplotlib) that turns text, images, and mathematical expressions into printer-ready 1-bit rasters; and
- a *protocol layer* that manages BLE communication and packet framing.

Text prints with an adjustable font size, images print with contrast and dithering controls, and math formulas print through a LaTeX renderer — three modes built into the application.

One implementation snag stood out. BLE communication runs asynchronously; Tkinter's event loop does not, being single-threaded and synchronous. BLE calls cannot be allowed to block the interface, and Tkinter was never built to share its main thread with an `asyncio` event loop. The solution runs a dedicated `asyncio` loop on a separate daemon thread and marshals coroutines onto it from the GUI callbacks:

```python
import asyncio
import threading
import tkinter as tk

class App:
    def __init__(self, root):
        # Dedicated event loop for async BLE operations
        self.loop = asyncio.new_event_loop()
        # Run the loop on a separate daemon thread
        self.thread = threading.Thread(target=self.run_async_loop, daemon=
            True)
        self.thread.start()
        # GUI holds a handle to the async loop
        self.app = PrinterApp(root, self.loop)

    def run_async_loop(self):
        asyncio.set_event_loop(self.loop)
        self.loop.run_forever()

    def start_scan(self):
        asyncio.run_coroutine_threadsafe(
            self.app.scan_for_devices(), self.loop)

if __name__ == "__main__":
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```

Listing 2: Bridging the Tkinter main thread with a dedicated asyncio event loop.

6.2 ESP32 firmware and the print state machine

The edge firmware is written in Arduino/C++ and juggles three concurrent duties: keeping Wi-Fi up, keeping the MQTT connection up, and controlling the printer over BLE. On boot it connects to Wi-Fi, then to the broker, then to the printer, and it re-establishes any of these connections automatically if it drops.

Rendering is not trivial, because the printer wants bitmap data rather than text. Rather than block the main loop for the full duration of a print, the firmware drives a non-blocking state machine over successive iterations of `loop()`, shown in Figure 5. Incoming text is word-wrapped to the active font width (24 or 26 characters per line), split into as many as 240 lines, and paginated. Each page goes into a graphics buffer, gets sent ten scan lines at a time, and only the last page is followed by a paper feed. Successive pages are printed seamlessly, without an intervening feed, and the line array is cleared afterward to avoid heap fragmentation.

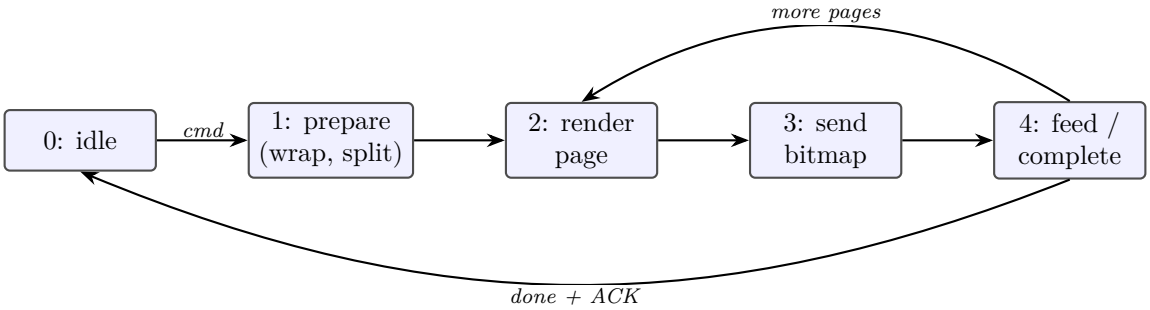

Figure 5: Non-blocking print state machine in the ESP32 firmware. A print command advances the device from idle through preparation, per-page rendering, and bitmap transmission. Multi-page jobs loop back to render the next page without a paper feed; the final page triggers a feed and a completion acknowledgment.

Two liveness mechanisms guard the BLE link. First, the printer slips into a low-power state after a few minutes without a command, so the firmware issues a wake sequence roughly once a minute and, if the printer stops responding, tears down and rebuilds the BLE connection. Second, a connection watchdog reboots the ESP32 if it has been unable to hold a printer connection for five minutes, clearing any accumulated heap or BLE-stack corruption. Device status (online flag, free heap, uptime, and Wi-Fi RSSI) is published over MQTT every two minutes.

7 Production Challenges and Failure Analysis

The system described so far seemed complete. Printing worked, the dashboard worked, scheduling worked, and the daily weather print ran unattended. Running it continuously, however, brought out problems that short tests had never shown. This section works through each one and its fix.

7.1 Trigger: database quota exhaustion

Within 13 days the serverless database burned through its monthly free-tier budget of 100 compute-unit hours and went offline, on course to consume roughly 200. What makes this striking is that there was only one user; the cause was architectural, not load. Three patterns dominated database activity:

1. **Status polling.** The dashboard polled the API every few seconds to ask whether the printer was online, and every poll reached the database.

2. **Unconditional state writes.** The ESP32 reported its status every 2 minutes through an upsert that always wrote, even when nothing had changed.

3. **Per-print write amplification.** Every print job cost three database writes on top of the constant polling reads: an insert at submission, an update at start, and a delete at completion.

The fix reworked each layer, rather than simply buying a larger database quota.

7.2 Layer 1: replacing polling with server push

Client polling gave way to a server-push model built on a managed channel service (Pusher Channels). Instead of the dashboard asking for changes over and over, the server now emits an event whenever state changes. Four events were defined: `status-update` for device and printer connectivity and Wi-Fi signal, `queue-added` when a job enters the queue, `queue-status` when a job's status changes, and `print-completed` when a job finishes. Idle-dashboard API traffic fell from about 1,200 to under 50 calls per hour. The interface also became instant, instead of lagging behind reality by up to a poll interval.

7.3 Layer 2: matching storage tier to data lifetime

Ephemeral device telemetry, that is, online/offline status, Wi-Fi signal strength, heap, and uptime, written every 2 minutes, was moved out of the relational database and into a serverless in-memory key-value store (Upstash Redis [7]). This data is transient by nature, so a persistent relational store was the wrong tool for it. A change-detection filter was added as well, so that insignificant fluctuations are not written at all: a status update is persisted only when the Wi-Fi signal moves by more than 5 dBm or the free heap by more than 10 KB. That filter suppressed about 93% of status writes and recorded only meaningful changes, while cutting the status-read latency from 15–20 ms on Postgres to 1–3 ms on Redis.

7.4 Layer 3: bypassing the relational database on the hot path

The most consequential change took the relational database off the common print path altogether, using the smart-routing logic in Figure 6. Because the great majority of prints happen while the device is online and no backlog exists, the system was reshaped so that such a print performs one key-value read (to confirm the device is online) and one queue-emptiness check, then publishes directly over MQTT with no relational write. The relational database now enters the picture only when the device is offline or a backlog already exists, cases where durable queuing really is needed, and where a background cron drains the queue. About 95% of prints now bypass the database. A resilience measure was added on top: if a database insert fails, say because the quota is exhausted, while the device is online, the system falls back to publishing directly over MQTT, so the print still succeeds even with the database fully unavailable.

7.5 Layer 4: a silent MQTT buffer-truncation defect

The audit uncovered a latent correctness bug. The ESP32's MQTT client buffer holds 2,048 bytes, and the JSON wrapping around each payload eats into that budget with field names, brackets, and quotation marks. The server, however, accepted messages up to 3,000 characters. Any payload beyond roughly 1,900 characters was therefore *silently truncated* by the client: the printer received malformed JSON, failed to parse it, and did nothing, with no error and no log entry. The fix enforces conservative limits in the user interface before a message is ever sent, 1,500 characters for a quick message and 1,200 for a letter body, the latter because it carries more JSON overhead. A live character counter warns at 90% of the limit and the send control is disabled once the limit is hit, so silent truncation can no longer happen.

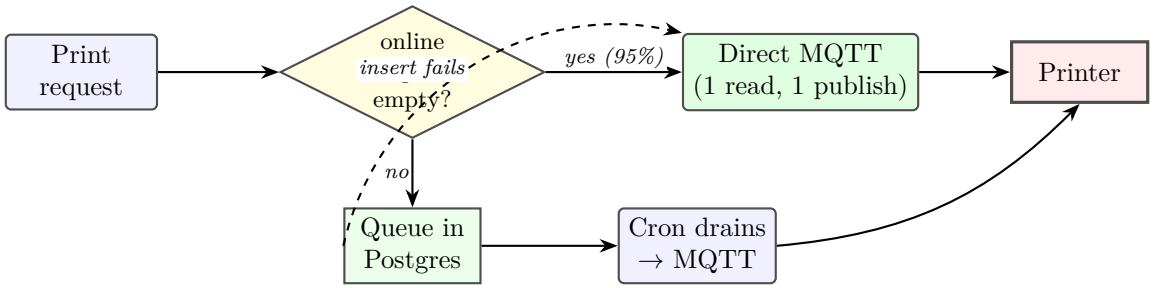

Figure 6: Smart-routing logic for print requests. When the device is online and the queue is empty, the request takes the direct MQTT path with a single Redis read and no relational write. Otherwise it is queued in Postgres and drained by a background cron. A fallback (dashed) reverts to direct MQTT if a database insert fails while the device is online.

7.6 Layer 5: endpoint authentication and configuration hygiene

Two security and configuration defects were corrected. The first was that the queue-processing endpoint, the one that drains the print queue and issues MQTT commands, had no authentication, so anyone who found the URL could set off cascading database queries and MQTT publishes. A bearer-token secret, known only to the external cron service, was added, and every other request now gets an HTTP 401. The second was a stale cron declaration left in the deployment configuration from an earlier architecture; had it deployed, it would have overwritten the externally managed schedule. It was removed, and explicit maximum-duration settings were added for the routes that genuinely need them.

7.7 Layer 6: incremental refinements

A set of smaller changes helped as well. On the server side, the MQTT client's keep-alive interval was shortened so that each short-lived serverless connection closes promptly after its status check rather than lingering. Prints gained an optional sender-name field, so sending anonymously or under a name now takes a deliberate choice. The weather widget moved from a sidebar card to a compact strip in the navigation bar, which keeps the information while giving back page space. Server-side logging was routed through a utility that disables itself in production, removing dozens of hot-path log statements, and scattered magic numbers were collected into a single constants module.

7.8 Quantitative results

Table 3 gathers the measured effect of the production hardening, and Figure 7 visualizes the reduction in the four resource metrics. The changes cut monthly database compute by about two orders of magnitude and idle API traffic by more than one, while also improving latency and closing a silent data-loss condition.

8 Discussion

A few lessons carry beyond this particular project.

Reverse engineering leans on prior art and structural reasoning. Recovering the BLE protocol went far faster thanks to prior community documentation of a structurally similar device [1] and to reasoning about the packet constants: the fixed header and footer, the command byte, the length field, and the trailing checksum. The hardest parts were not conceptual but

Table 3: Measured impact of the production-hardening changes.

Metric	Before	After
Monthly DB compute (CU-hrs)	∼200 (exceeded)	∼1–2
API calls, idle dashboard (/hr)	∼1,200	<50
Print latency, device online	up to minutes (cron)	<100 ms
Status read time	15–20 ms (Postgres)	1–3 ms (Redis)
Status writes skipped	0%	∼93%
Prints bypassing the database	0%	∼95%
Silent truncation risk	present (3,000-char)	eliminated (1,500/1,200)

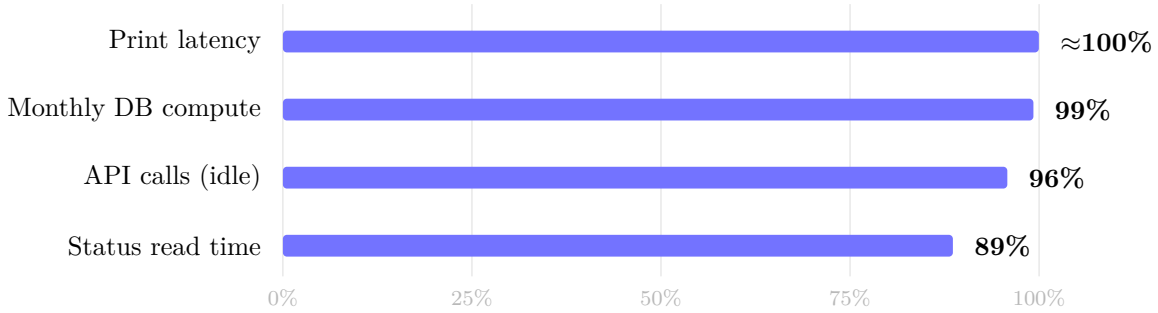

Figure 7: Reduction in key resource metrics after the production-hardening changes, expressed as the percentage decrease from the "before" to the "after" value in Table 3 (a longer bar is a larger improvement). All four metrics fall by roughly one to two orders of magnitude; print latency and database compute are cut by almost 100%.

mechanical, namely the exact CRC-8 computation and the correct GATT characteristic. A device that fails silently, ignoring malformed input without any signal, makes such mistakes costly to diagnose, which is a strong argument for building a known-good reference implementation early.

On constrained devices, protocol choice is really a memory decision. The HTTP-polling design failed (Section 5.3) because concurrent TLS and BLE stacks exhausted the heap, not because of latency or throughput. Moving to a single persistent MQTT connection over plain TCP solved the memory pressure and the latency at once. The lesson is that on constrained hardware the binding constraint is often memory footprint, and that a persistent publish/subscribe connection can be cheaper than repeated request/response exchanges.

Serverless cost tracks activity, not users. The database quota was exhausted by a single user because cost in a serverless, activity-billed model depends on how often requests and writes happen, not on how many users there are. Patterns that are essentially free on a dedicated server, polling every few seconds, writing state unconditionally, expanding each action into several writes, become the dominant cost driver. Push-based updates, change detection before writes, and keeping the hot path off the most expensive store are the effective answers.

Match the storage tier to the data's lifetime. Putting ephemeral, high-frequency telemetry in a durable relational database was a tier mismatch that both raised cost and slowed reads. Moving it to an in-memory key-value store cut status-read latency from 15–20 ms to 1–3 ms and, together with change detection, removed most of the writes. The broader rule, durable data in a relational store, transient state in a key-value store, commands over a message bus, applies to IoT back-ends in general.

Test under sustained, realistic conditions. Every production defect, quota exhaustion, silent truncation, the open endpoint, the stale configuration, was invisible in short functional tests and showed up only after the system ran continuously for days. Behavior that depends on time or volume calls for soak testing and explicit resource-budget monitoring before a system is called complete.

Security considerations. The deployed system keeps its trust boundaries explicit, and its weaknesses teach as much as its strengths do. Dashboard access is tiered: a shared credential permits printing only, while schedule management sits behind a separate password, so that the most destructive operations are not exposed by the demonstration account. On the transport side, however, the command channel relies on a public MQTT broker over plain TCP, with per-device topic namespacing as its only access control. This is adequate for a low-stakes personal deployment, where the worst outcome is an unwanted print, but it is not defensible for any sensitive workload. A production version would move to an authenticated, TLS-encrypted broker with per-device credentials and an unguessable device identifier, and would place every state-changing API route behind authentication rather than relying on topic obscurity. Documenting this gap, rather than hiding it, reflects the same lesson as the rest of the study: the easy first design and the durable design are rarely the same, and security is one more dimension along which that holds.

Limitations. This is a single-device, single-user case study. The absolute measurements are tied to the chosen free-tier providers and the observed workload, and should be read as illustrative rather than as a controlled benchmark. The transport-security limits are discussed above under security considerations. The reverse-engineered protocol belongs to one printer family. Related devices share the same header, command framing, and CRC-8 scheme, but no claim is made about other vendors. The security review dealt with the specific defects found and is not a comprehensive audit.

9 Conclusion

This paper followed one project end to end: reverse engineering an undocumented BLE thermal-printer protocol, building a serverless cloud-connected IoT system around an ESP32 bridge, and then analyzing and fixing the failures that surfaced only in sustained production. Recovering the protocol meant identifying the packet constants, confirming a table-based CRC-8 integrity scheme, and finding the correct GATT characteristic. The cloud architecture moved from HTTP polling to MQTT once the device's memory limits made polling infeasible, which brought print latency below 100 ms. Running in production then revealed that activity-driven serverless cost, a storage-tier mismatch, and a silent buffer-truncation bug governed how the system actually behaved. Addressing them cut monthly database compute from roughly 200 to 1–2 compute-unit hours and idle API traffic from about 1,200 to under 50 calls per hour, leaving a system that runs comfortably inside free-tier limits while staying reachable worldwide. The overarching lesson is plain: the architecture that is easiest to build first is rarely the one that survives real use. Correct edge-protocol selection, push-based communication, and storage-tier matching were each needed to make even a small IoT system durable.

Reproducibility and Availability

A live instance of the deployed system is publicly accessible at `https://esp32stp.vercel.app` so that reviewers may inspect and exercise it directly. A print-only demonstration account is available on request from the author for evaluation; a message submitted through this interface is printed on the physical device in real time when it is online, or queued otherwise.

Schedule management and other state-changing operations are deliberately gated behind a separate password and are not exposed by the demonstration account. The firmware framing, GATT identifiers, buffer sizes, timing intervals, and routing thresholds reported here are drawn directly from the deployed ESP32 firmware and Next.js application. Provider-specific configuration, such as free-tier limits and public broker availability, changes over time and should be checked against current documentation.

References

[1] WerWolv, *Thermal Printer BLE Protocol Reverse Engineering*, werwolv.net technical blog. https://werwolv.net/blog/cat_printer/.

[2] H. Blidh et al., *Bleak: A Cross-Platform Bluetooth Low Energy Client for Python*, Bleak project documentation. https://bleak.readthedocs.io/.

[3] Bluetooth SIG, *Bluetooth Core Specification* (Generic Attribute Profile, GATT), Bluetooth Special Interest Group. https://www.bluetooth.com/specifications/specs/.

[4] OASIS, *MQTT Version 5.0 – OASIS Standard*, 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[5] S. Nicholas, *Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android*, technical blog. https://stephendnicholas.com/posts/power-profiling-mqtt-vs-https.

[6] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, *The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures*, Proc. 2nd Int. Workshop on Serverless Computing (WoSC), ACM, 2017. DOI: 10.1145/3154847.3154848.

[7] Redis Ltd., *Redis: An In-Memory Data Structure Store*, Redis documentation. https://redis.io/docs/.

[8] R. W. Floyd and L. Steinberg, *An Adaptive Algorithm for Spatial Grayscale*, Proc. Society for Information Display, vol. 17, no. 2, pp. 75–77, 1976.

[9] Espressif Systems, *ESP32 Technical Reference Manual*, Espressif Systems. https://www.espressif.com/.