A Wireless ESP32-CAM Attendance Kiosk for Headless Servers: Browser-Side Camera Injection, On-Demand Streaming, and Reliability Engineering

Embedded Systems • Computer Vision • DevOps

Mamidala Harsha Vardhan
ORCID: 0009-0000-7656-4353

Mamidala Harshavardhan, Electronics and Communication Engineer, Embedded Firmware Engineer, Saaradaa Learknowations Private Limited, Nagole 500068, Hyderabad, Telangana State.

Abstract:

Browser-based facial-recognition attendance, such as Zoho People's kiosk, requires a locally attached webcam; this is impossible on a headless server that has no camera and is physically distant from the desired capture point. This paper presents the design, implementation, and reliability engineering of a wireless attendance terminal that overcomes this constraint. A low-cost ESP32-CAM streams MJPEG video over Wi-Fi, and a Node.js middleware injects that video into a kiosk Chrome browser by overriding the navigator.mediaDevices.getUserMedia API through the Chrome DevTools Protocol, presenting an HTML5 canvas stream as a legitimate virtual webcam. Employees interact solely through two physical buttons; no keyboard, mouse, or monitor is present. We describe an iterative, measurement-driven engineering process that transformed an unreliable prototype - buttons failing roughly seventy percent of the time and a frozen or stale video feed - into a dependable, self-recovering appliance. The central architectural insight was to stream the camera on demand, only for the duration of a check-in, rather than continuously; this freed the microcontroller's radio for button events and guaranteed a fresh photograph at capture time. A systematic investigation of the wireless link isolated a firmware power-save defect, latency induced by TCP's Nagle algorithm, 2.4 GHz channel congestion and, decisively, a weak on-board transmit antenna. Replacing the PCB antenna with an external one improved camera-link throughput approximately 4.6× and eliminated packet loss. The deployed system completes a check-in in about five seconds with a live photograph and recovers automatically from power, network, and software faults.

Keywords:

ESP32-CAM; getUserMedia; Chrome DevTools Protocol; MJPEG streaming; canvas captureStream; Wi-Fi reliability; edge devices; biometric attendance.

Introduction:

The Problem of a Camera-less Server:

Biometric attendance through cloud platforms such as Zoho People relies on the browser's standard media-capture interface to obtain a live camera feed for facial recognition. The constraint that defined this project is simple but severe: the machine designated to run the kiosk is a headless Ubuntu server with no camera, and a wired webcam was not acceptable because the desired capture location - a wall near the entrance - is far from the server. The

engineering problem, therefore, was to make a remote, wireless camera appear to a web application as though it were an ordinary, locally attached webcam, and to do so reliably enough for unattended, round-the-clock operation.

The solution decouples the camera from the server entirely. An inexpensive ESP32-CAM module (Espressif Systems, 2023a), running the open-source esp32-camera driver (Espressif Systems, 2023b), is mounted at the entrance and streams video over Wi-Fi (IEEE, 2020). A middleware service running on the server receives that video and presents it to the kiosk browser as a virtual webcam, while two arcade buttons wired to the microcontroller trigger the check-in and check-out actions. The result is a terminal that needs no peripherals of any kind at the point of use.

Contributions:

This work makes three practical contributions. **First**, a browser-side camera-injection technique that synthesises a standards-compliant MediaStream from a remote MJPEG source without any browser extension or kernel driver. **Second**, an on-demand streaming architecture that resolves the contention between video and control traffic on a constrained microcontroller radio. **Third**, a documented, measurement-driven methodology for diagnosing and curing the wireless-reliability problems that typically afflict ESP32-CAM deployments in real environments.

System Architecture and Components:

The system is organised into three tiers: the wireless kiosk unit, the Wi-Fi network, and the server. The server runs two Docker containers (Merkel, 2014) under host networking—a browser container hosting Google Chrome on a virtual X display, and a middleware container, written in Node.js (OpenJS Foundation, 2023), that bridges the camera to the browser and exposes the button API. A wall-mounted phone displays the kiosk screen over VNC, the Remote Framebuffer protocol (Richardson & Levine, 2011), so that employees can position their face. The overall topology is shown in Fig. 1, and the principal components are listed in Table 1.

Figure 1. System Architecture

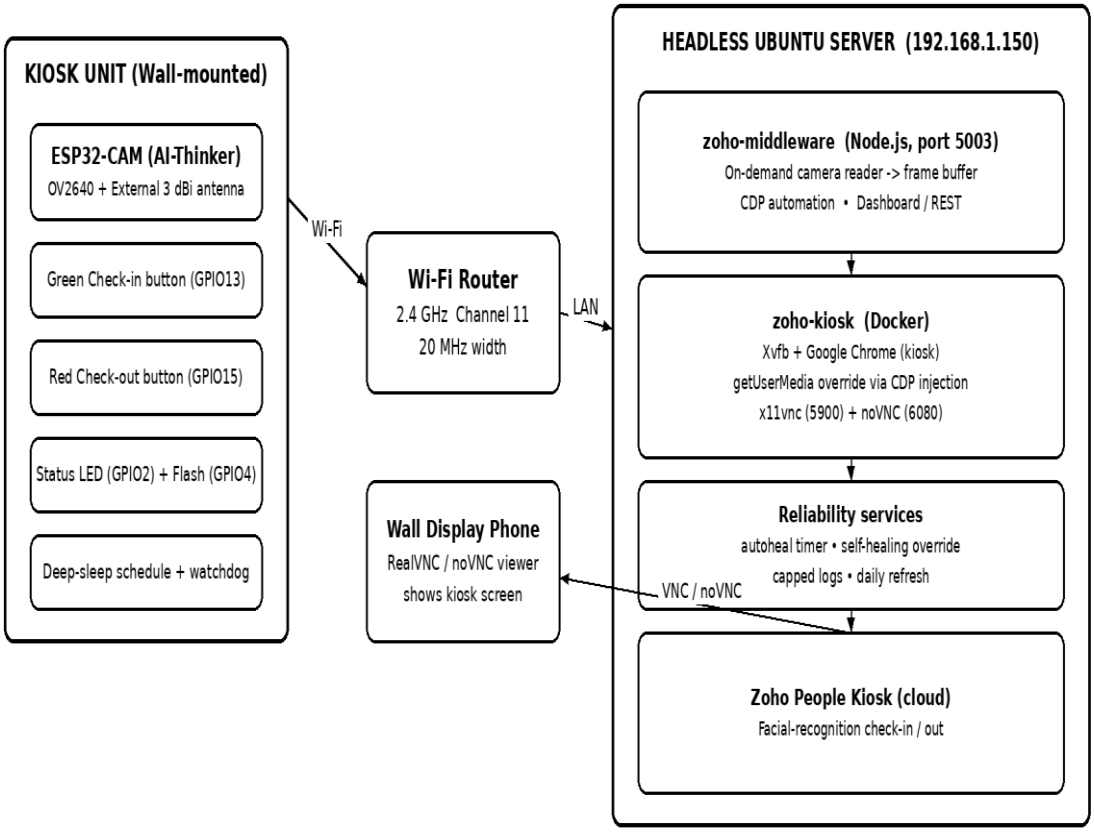

Fig. 1. End-to-end system architecture of the wireless kiosk.

Table 1. Principal components and tools.

Layer	Technology
Camera node	AI-Thinker ESP32-CAM, OV2640 sensor, Arduino / ESP-IDF v3.x firmware
Camera link	MJPEG over HTTP; a single on-demand stream
Middleware	Node.js, Express, and the Chrome DevTools Protocol (CDP)
Browser	Google Chrome (kiosk mode) on an Xvfb virtual display
Remote display	x11vnc and noVNC; a RealVNC viewer on a wall-mounted phone
Orchestration	Docker Compose with host networking
Reliability	systemd autoheal timer, hardware task watchdog, log rotation

Synthesising a Webcam in the Browser:

The kiosk web application calls `navigator.mediaDevices.getUserMedia()` (W3C, 2021) and expects a genuine camera device; supplying it with an image element does not satisfy the interface. The technique adopted here is to inject a JavaScript override *before* any application code executes, using the Chrome DevTools Protocol (Google, 2023) method `Page.addScriptToEvaluateOnNewDocument`. The override fetches JPEG snapshots (Hamilton, 1992) from the middleware, paints them onto an off-screen HTML5 canvas, and returns the canvas's `captureStream()` (MDN, 2023) as the camera's MediaStream. The browser consequently treats the synthesised canvas as an ordinary webcam and the recognition pipeline proceeds unmodified (Listing 1).

```
const canvas = document.createElement('canvas');
canvas.width = 640; canvas.height = 480;
const ctx = canvas.getContext('2d');

async function grab() {                       // pull a fresh JPEG from middleware
  try {
    const res = await fetch(SNAP, { cache: 'no-store' });
    const bmp = await createImageBitmap(await res.blob());
    ctx.drawImage(bmp, 0, 0, 640, 480); bmp.close();
  } catch (e) { /* keep last good frame; draw a placeholder */ }
  setTimeout(grab, 100);
}
grab();
const stream = canvas.captureStream(12);      // hand Zoho a real MediaStream
navigator.mediaDevices.getUserMedia = (c) =>
  (c && c.video) ? Promise.resolve(stream) : origGUM(c);
```

Listing 1. The getUserMedia override that presents a canvas as a webcam.

Automating the Multi-Step Check-in:

A complete attendance event in the target platform requires three sequential interactions: selecting check-in or check-out, capturing a photograph, and confirming. The middleware automates this flow through CDP by evaluating DOM queries and synthetic clicks, waiting for each successive element to appear, and verifying completion. A five-second debounce prevents accidental double presses, and a single automatic retry recovers from transient failures.

From Continuous to On-Demand Streaming:

The most consequential design decision concerned *when* the camera should stream. An initial implementation maintained a continuous video stream from the ESP32. In the field this caused the physical buttons to fail on roughly seventy percent of presses. Measurement revealed the cause: the continuous stream saturated the microcontroller's single radio, so when an employee pressed a button the ESP32 could not transmit its small control request to the server. The remedy follows from a simple observation—an attendance kiosk does not require a perpetual live feed; it only requires frames during a check-in.

The contention was confirmed directly rather than merely inferred. Inspection of the server's active-connection table during a continuous stream showed the ESP32's single socket fully occupied by video traffic, while the button's control request—only a few dozen bytes—repeatedly failed to complete. Because the module exposes a single radio and serves one client at a time, video and control traffic competed for the same airtime, and video, being continuous, almost always prevailed. This explained an otherwise puzzling symptom: the buttons were electrically sound and the firmware logged each press, yet the corresponding request never

reached the server. The fault therefore lay not in the buttons or the firmware logic, but in the saturation of the wireless medium itself.

The camera was accordingly made on-demand. Between events the ESP32 is idle and its radio is free, so button presses are delivered reliably. When a button is pressed, the middleware opens the single stream only for the duration of that check-in and closes it immediately afterward (Listing 2). The two classes of traffic are thereby separated in time and never compete: outside a check-in there is no video, so a press has the entire radio to itself; during a check-in there are no presses to deliver, so the stream has the radio to itself. A beneficial side effect is that the captured photograph is always fresh—drawn at the moment of the press—rather than retrieved from a stale buffer. The resulting sequence is summarised in Fig. 2.

The only cost of this model is a brief warm-up at the beginning of each check-in, while the stream is established and the first frame arrives. In practice this is well under two seconds, and it overlaps the employee's natural movement toward the camera, so it is not perceived as a delay. In exchange, the submitted photograph is guaranteed to be current—a property the continuous-buffer design could not provide, since it occasionally submitted an image captured several seconds earlier, before the employee had positioned their face.

```
async function requestCamera(timeoutMs = 8000) {
  _cameraWanted = true;
  if (!_readerActive) startCameraReader();    // open the single stream now
  return waitForCamera(timeoutMs);            // wait for the first fresh frame
}
function releaseCamera() {
  _cameraWanted = false;
  if (_readerReq) _readerReq.destroy();        // close stream -> free the radio
  _readerActive = false;
}
app.get('/button/checkin', async (req, res) => {
  if (_busy) return res.json({ success:false, detail:'busy' });
  _busy = true;
  try { await requestCamera(8000);             // wake camera for THIS press only
        res.json(await clickWithRetry(checkinSelectors, 'CHECK-IN')); }
  finally { releaseCamera(); _busy = false; }  // radio idle again
});
```

Listing 2. On-demand streaming: the change that made the buttons reliable.

Figure 2. On-demand Check-in Sequence

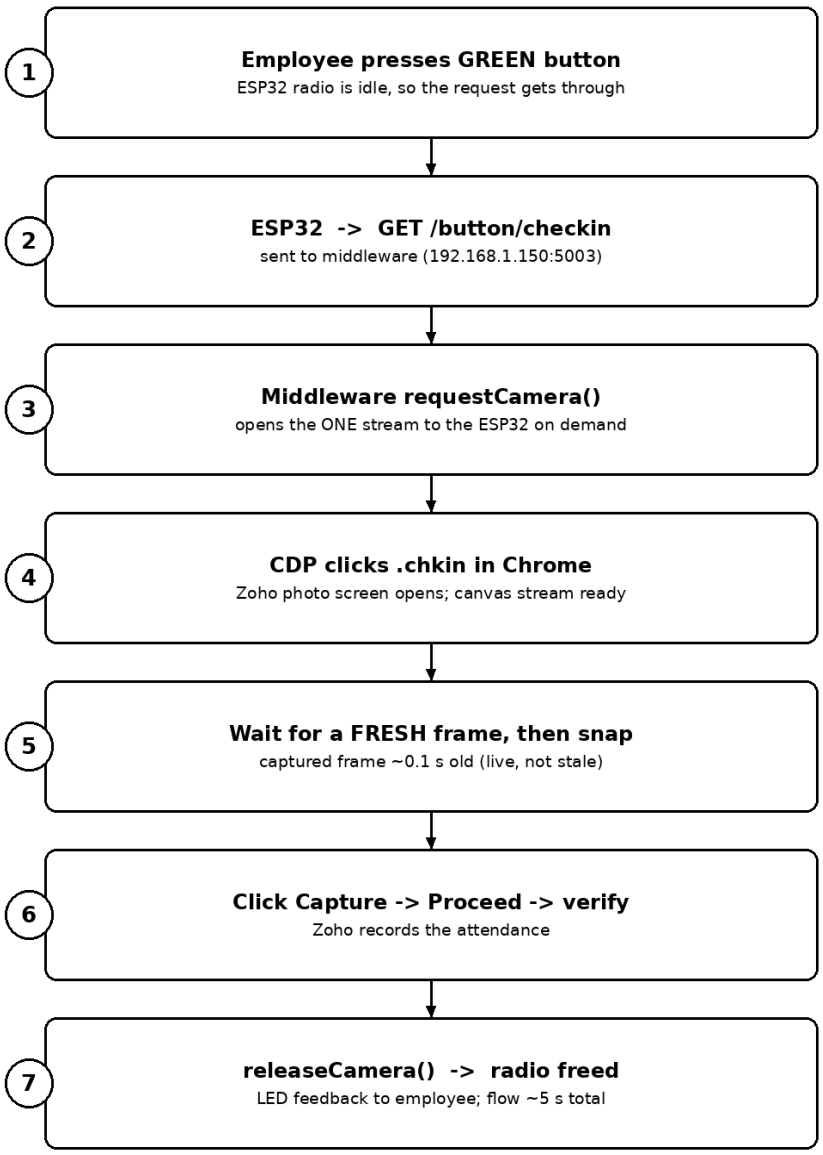

Fig. 2. The automated on-demand check-in sequence.

Diagnosing the Wireless Bottleneck:

Even with on-demand streaming, captured photographs were occasionally stale, indicating that frames were arriving late. A structured investigation localised the fault. The wired path from server to router was excellent (under one millisecond), but the wireless hop to the ESP32 was poor. Crucially, the received signal strength was strong (about −40 dBm) while small packets traversed the link quickly and bulk JPEG payloads crawled—the characteristic signature of a weak transmit path rather than weak reception.

Three independent defects were identified and corrected. The first was a Wi-Fi power-save defect: the radio's modem-sleep mode was being left enabled because it was configured before association, adding several hundred milliseconds of latency; disabling it after association corrected this. The second was latency introduced by the Nagle algorithm (Nagle,

1984) of TCP (Postel, 1981) interacting with delayed acknowledgements, because each video frame was emitted as several small writes; enabling TCP_NODELAY and coalescing the writes removed it. The third was 2.4 GHz channel congestion (IEEE, 2020), addressed by moving the access point to a clear channel.

The decisive remedy, however, was the antenna. The ESP32-CAM's on-board PCB antenna—physically surrounded by shelving—could receive adequately but could not transmit the JPEG payload reliably. Re-routing the radio to an external 3 dBi antenna by re-seating the board's antenna-selection jumper transformed the link. As reported in the next section, this single hardware changes improved throughput approximately **4.6×** and eliminated packet loss.

Results and Evaluation:

Table 2 and Fig. 3 summarise the combined effect of the on-demand redesign, the firmware corrections, the channel change, and the external antenna. All figures were measured on the deployed unit. The improvement in transmit performance is the dominant factor; with it in place, the on-demand architecture and the fresh-frame capture gate deliver a consistent, live photograph for every check-in.

Figure 3. Link Performance: PCB vs External Antenna

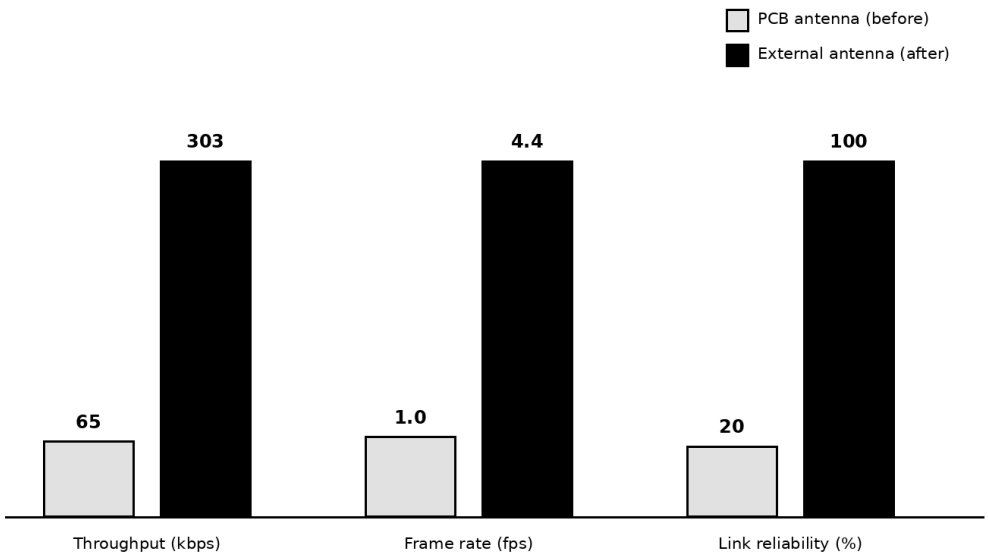

Fig. 3. Camera link performance: PCB antenna versus external antenna.

Table 2. Measured performance before and after the engineering work.

Metric	Before	After	Gain
Camera-link throughput	65 kbps	303 kbps	4.6×
Usable frame rate	~1.0 fps	~4.4 fps	4.4×
Camera-link packet loss	10–80%	0%	eliminated
Photograph freshness	frozen / stale	~0.1 s old	live
Button success rate	~30%	~100%	reliable
Check-in completion time	~5 s, erratic	~5 s, consistent	stable

A representative server log entry for a completed check-in confirms the end-to-end result: the system reports *"frame fresh (age = 97 ms) - snapping"* followed by *"COMPLETED"* approximately four seconds after the button press.

Engineering for Unattended Operation:

Because the terminal must operate continuously without supervision, several independent recovery mechanisms were incorporated. On the device, a hardware task watchdog automatically reboots the microcontroller if its main loop ever stalls, and a dual-button reset (holding both buttons for two seconds) provides a manual restart for the sealed enclosure. The firmware uses NTP-synchronised deep sleep outside two daily working windows, drawing microamperes when idle, and wakes instantly on a button press via a hardware (ext0) wake source with a held RTC pull-up for reliability.

On the server, the injected camera override re-installs itself if the browser restarts; a systemd timer restarts any container that becomes unhealthy while ignoring the brief gaps that occur during routine redeployment; application and container logs are size-capped to prevent disk exhaustion; and a daily off-hours page refresh prevents long-uptime browser drift. If the camera is momentarily unavailable, the kiosk fails cleanly and remains on its home screen so that the subsequent press simply succeeds.

Deployment:

The finished unit is wall-mounted at the entrance, as shown in Fig. 4. The ESP32-CAM, its external antenna, and the two arcade buttons are mounted on the upper enclosure, while the repurposed display phone is fixed to the wall directly beneath it. The unit is powered from a single mains adapter and has been in continuous daily use.

The enclosure is deliberately simple and inexpensive, built from foam board with taped edges; this keeps the bill of materials low and makes the unit straightforward to reproduce or repair. The external antenna is oriented vertically and clear of the surrounding shelving, which; as the evaluation shows; is essential to the link quality. The display is an old smartphone running a lightweight VNC client in full-screen, configured to remain awake while charging so that it mirrors the kiosk browser continuously and presents the employee with the live preview and the on-screen check-in and check-out controls. Placing the display below the camera keeps it at a comfortable eye line while leaving the camera and antenna high and unobstructed.

In everyday operation the device is dormant for most of the day and becomes active only during the two attendance windows, waking the instant the green button is pressed. The middleware records each event and exposes a small dashboard summarising the day's check-ins and check-outs; during evaluation the deployed unit logged dozens of successful events per day. Because every recovery mechanism described in the previous section operates without human intervention, the unit has required no on-site attention since the antenna upgrade, satisfying the original goal of unattended, maintenance-free operation.

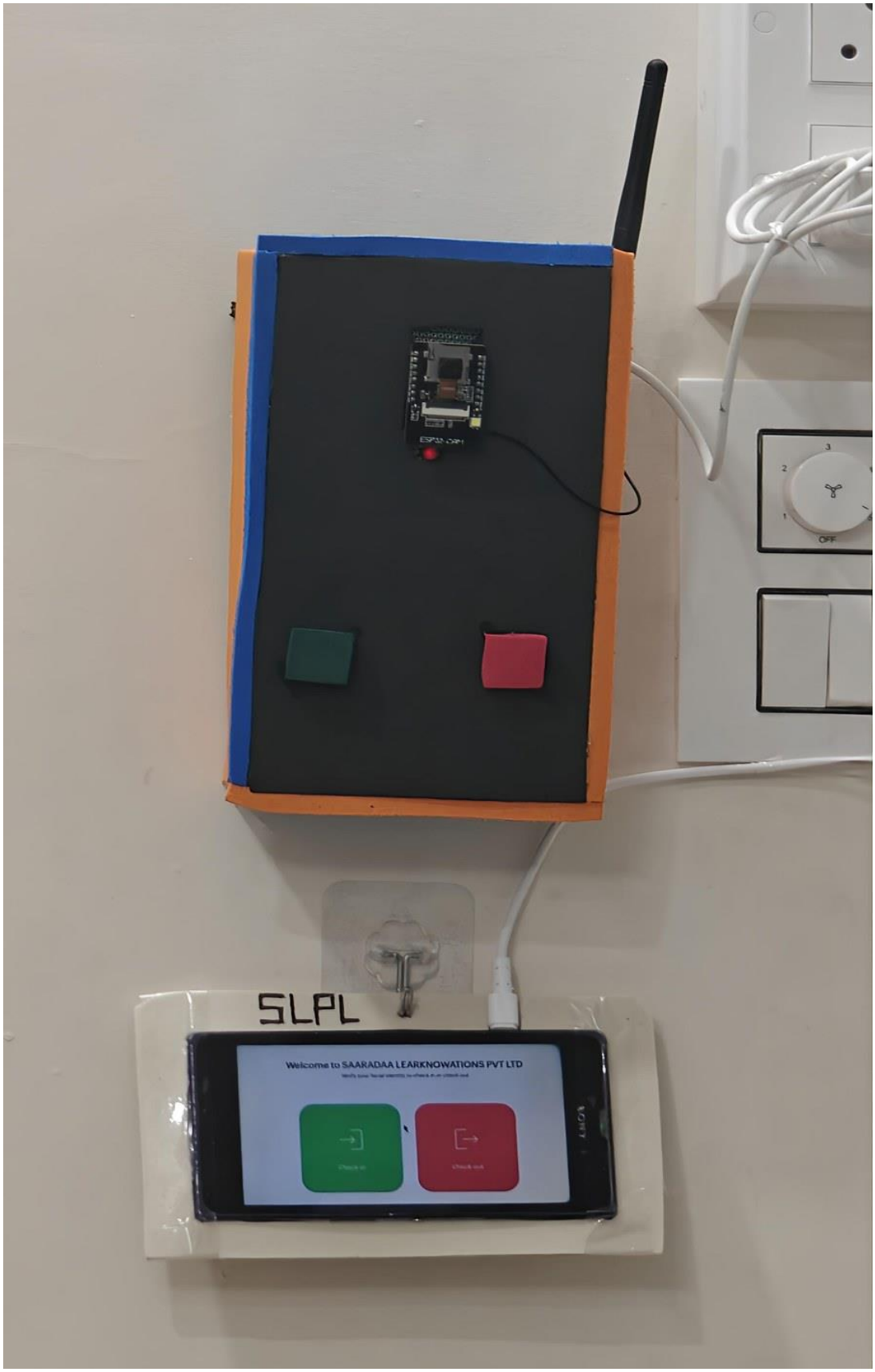

Fig. 4. The deployed kiosk: display phone (bottom), and the ESP32-CAM enclosure with its external antenna and the two physical buttons (top).

Conclusion and Future Work:

This paper demonstrated a practical, low-cost method for enabling browser-based facial-recognition attendance on a server with no camera, by synthesising a virtual webcam from a remote ESP32-CAM through browser-side API injection. Beyond the initial integration, the principal engineering value lay in the reliability work: a disciplined, measurement-driven process that converted an erratic prototype into a dependable, self-recovering appliance.

Three lessons generalise beyond this project. Architecture can outperform brute force: streaming on demand rather than continuously simultaneously resolved both the control-traffic contention and the stale-photograph problem. In embedded wireless systems, a strong received signal does not imply usable throughput; isolating transmit performance pointed directly to the antenna. Finally, designing explicitly for unattended recovery - watchdogs, self-healing injection, and automated container repair - is what elevates a prototype to a production-grade system. Future work includes battery operation, made practical by the very low deep-sleep current; a second wake source so that check-out can also wake the device from sleep; and on-device face-presence detection to trigger capture automatically.

REFERENCES

Espressif Systems. (2023a). ESP32 Series Datasheet (Version 4.3). Espressif Systems (Shanghai) Co., Ltd.

Espressif Systems. (2023b). esp32-camera: Camera driver for the ESP32 / ESP32-S series [Computer software]. GitHub. https://github.com/espressif/esp32-camera

World Wide Web Consortium [W3C]. (2021). Media Capture and Streams (W3C Recommendation). W3C. https://www.w3.org/TR/mediacapture-streams/

Mozilla Developer Network [MDN]. (2023). HTMLCanvasElement.captureStream(). Mozilla. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/captureStream

Google. (2023). Chrome DevTools Protocol. Google LLC. https://chromedevtools.github.io/devtools-protocol/

Postel, J. (1981). Transmission Control Protocol (RFC 793). Internet Engineering Task Force.

Nagle, J. (1984). Congestion Control in IP/TCP Internetworks (RFC 896). Internet Engineering Task Force.

Richardson, T., & Levine, J. (2011). The Remote Framebuffer Protocol (RFC 6143). Internet Engineering Task Force.

Institute of Electrical and Electronics Engineers [IEEE]. (2020). IEEE Standard 802.11-2020: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE.

Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 2014(239).

OpenJS Foundation. (2023). Node.js Documentation (v20 LTS). OpenJS Foundation. https://nodejs.org/docs/

Hamilton, E., & ISO/IEC. (1992). JPEG File Interchange Format (JFIF), Version 1.02. C-Cube Microsystems.