EchoGuard: Smart Sound-Activated Camera Surveillance System using Raspberry Pi and IoT Integration

Jewel C.[1], Dylan W.[2], Hongzhao T.[3], Yangfei W.[4], S. M. Alavi[5]

[1] W Booth School of Engineering Practice and Technology, McMaster University, Hamilton, Ontario, Canada

October 30, 2025

Abstract

This paper presents the design and implementation of a sound-activated Internet of Things (IoT) surveillance system built on Raspberry Pi 4 platform. The prototype integrates an LM393 sound sensor for acoustic event detection, an active-high buzzer and LED for real-time alerts, an SG90 servo for horizontal camera panning, and an OV5647 camera for image capture. A Flask-based backend coordinates sensing, actuation, and communication through RESTful APIs, while a web interface provides manual control and live status monitoring for user control and real-time notifications. When a sound is detected, the system enters a latched alarm state, activates both the buzzer and LED, and commands the servo which the camera is mounted to sweep multiple angles to capture environmental context. An email notification module further alerts the user once the sound is detected, in other words, this event is triggered simultaneously with other events that mentioned above. The design emphasizes low latency, modular hardware, software integration, and safe power isoltaion between 3.3V and 5 V domains. Experimental testing confirmed stable operation, quick response time, and reliable web-based control under typical indoor acoustic conditions.

Keywords

Raspberry Pi, Internet of Things (IoT), Flask Web Framework, Sound Detection, Email Notification, PWM Signal Generation, Servo Control, RESTful API, Web Dashboard, Embedded Control System, Edge Computing Device, Power Isolation

[1]Jewel Chen: 400375858, Email: chenj508@mcmaster.ca
[2]Dylan Wang: 400234428 Email: wangl189@mcmaster.ca
[3]Hongzhao Tan: 400136957, Email: tanh10@mcmaster.ca
[4]Yangfei Wang: 400313125, Email: wangy977@mcmaster.ca
[5]Seyedeh Marjan Alavi: -, Email: alavis2@mcmaster.ca

Nomenclature

Symbol / Term	Definition
A	Ampere, standard unit of electric current measuring charge flow rate
`dict`	Python dictionary data type (key–value pairs)
$\mathbb{Z}$	Integer data type, representing whole numbers
$\mathbb{R}$	Real data type, representing continuous numeric values
$\mathbb{B}$	Boolean data type, with values `True` or `False`
URL	Uniform Resource Locator mapped to Flask API endpoints
GPIO	General Purpose Input/Output for backend communication
API	Application Programming Interface for backend communication
PWM	Pulse Width Modulation signal for servo angle control
SMTP	Simple Mail Transfer Protocol for sending notification emails
VCC	Supply voltage
GND	Common ground reference shared by Pi and external power supply
D_0	Digital output pin of LM393 sound sensor module
JSON	JavaScript Object Notation used for data exchange between Flask and web client
$\wedge$	Logical AND
$\vee$	Logical OR
$\neg$	Logical NOT
$:=$	Assignment operator in programming, meaning 'assign'

Contents

1. Introduction

Smart Sound-Activated Camera Surveillance System. Cloud free, lightweight, edge-first platform that offers quick, privacy-preserving awareness for small spaces such as dorm rooms, rented apartments, and benchtop labs. Platform is built around a Raspberry Pi and commodity peripherals which is a digital sound sensor, an active buzzer and LED, a hobby servo, and the Pi camera which responds to acoustic events within a second, latching into an alarm state until manually acknowledged, and sweeping through predefined angles while capturing a series of images. By processing sensing, decision making, and actuation on the device, the system avoids dependence on the cloud and the fragility and data exposure that come with it.

It is designed to be very robust and minimalist. We offer a layer of hardware for the sound sensor's active-low output: GPIO edge detection and software debouncing. If the hardware does not support interrupts we will fall back to polling, but in both cases we will be responsive(Kumar et al., 2018). Once a sound event has been detected, the alarm is latched in software and mirrored in hardware by driving the buzzer and LED continuously. In parallel, a background task commands the servo to pan between common angles like 20°, 60°, 100°, and 140°, capturing photos that provide multi-view context of the scene. Images are written to a local directory and served by Flask's static file handler, keeping the media pipeline simple and easy to test.

A very thin REST interface provides access to status, latest image, and a reset point. A simple web dashboard, running as a single page and polling the API every two seconds, visualizes the state and logs its output in live graphs on phones and laptops running on the same local network. To maintain a nagging presence even when the dashboard is closed, an email is sent async when the first alarm is transitioned. This architecture places the attack surface in a small surface area, places a minimal configuration burden on operations, and delivers predictable performance on commodity hardware.

Though intentionally simple, the system is extensible. One could add additional endpoints to the REST surface to handle manual snapshots or angle control, one could add the detection front end to the device to handle audio classification or sensor fusion on device, etc., without changing the semantics of the alarm. This demonstrates a practical, reproducible solution to the problem of handling sound-triggered, latched alarms with an emphasis on low latency, operational clarity, and local data ownership(Nunes et al., 2014).

2. System Design and Methodology

2.1. System Architecture

The proposed system is a sound-activated surveillance device built on Raspberry Pi 4. Figure 1 presents the end-to-end pipeline: (i) an LM393 digital sound module monitors acoustic events and generates a binary trigger ($D0$); (ii) the Raspberry Pi (Flask HTTP server) debounces and interprets the trigger, then activates audible/visual alerts (active-high buzzer, active-high LED), and commands a pan servo (SG90) to sweep multiple angles; (iii) at each angle the Pi camera (OV5647) captures a still image via `rpicam-still`; (iv) images and device states are exposed through a web dashboard for remote viewing and manual override; (v) an email

notification can be sent to the user to prompt and review.

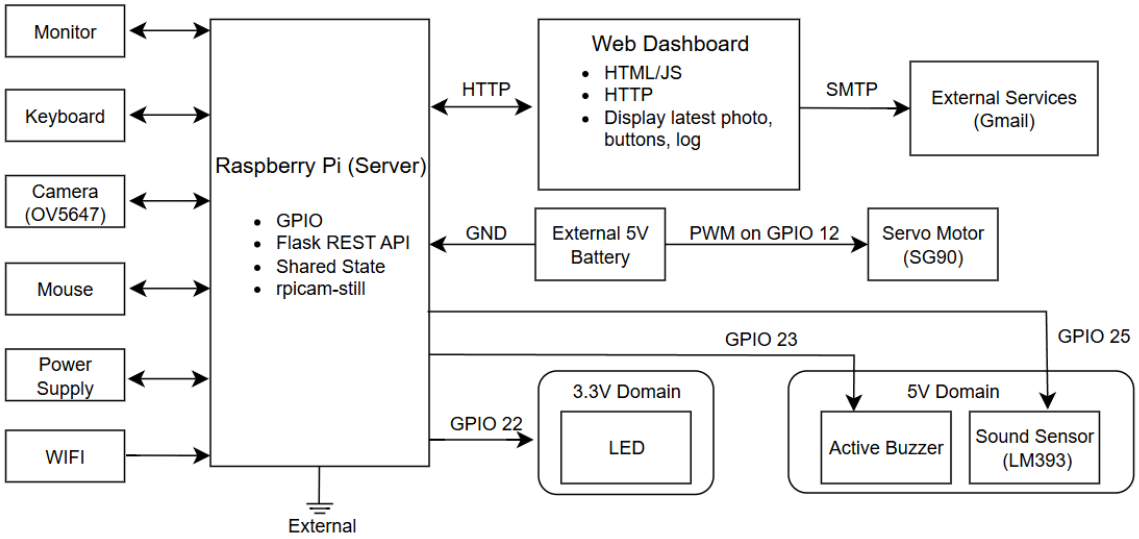

Figure 1: System architecture Diagram.

2.1.1. Design objectives.

The architecture was guided by four objectives: (i) responsiveness to short acoustic edges by GPIO BOTH-edge detection and software cooldown; (ii) safety in power delivery by isolating high-current 5 V loads (servo) from the Pi and enforcing a common ground; (iii) operator control via a lightweight REST API and web UI for manual alarm override; and (iv) reproducibility using off-the-shelf modules and stock `rpicam-still` without heavy runtime dependencies.

2.1.2. Operational Modes.

The controller exhibites two modes. *Idle*: the server polls status and awaits a GPIO edge. *Alarm*: upon a valid sound edge, a latched `alarm-running` flag starts a blinking thread (active-high buzzer and LED) and a pan-capture task; the alarm persists *until* the user resets it via the web UI, enabling post-event reviews.

2.1.3. Concurrency model.

Three cooperative activities run concurrently: (i) the Flask HTTP server (REST); (ii) a GPIO edge handler (debounce + cooldown + idempotent launch); and (iii) a background "blink-hold" thread toggling the buzzer/LED while `alarm-running` is true. Shared state (alarm flag, last-photo URL, device states) is protected by a mutex to prevent races between the edge handler and API requests.

2.1.4. Power domains and safety.

High-current loads (SG90 Servo) are powered from an *external* regulated 5 V rail (USB power bank or 5 V/2 A adapter). The Pi sources only logic signals (3.3 V domain, 5 V domain). A **single common ground** between the external 5 V rail and the Pi is mandatory; the external $+5$ V is never back-fed into the Pi 5 V pin.

2.1.5. Assumptions, constraints, and limitations.

The system assumes impulsive sounds that cross the LM393 threshold; continuous low-level noise may not trigger. Pan coverage is limited to the mechanical range of a single SG90; Email notification requires network connectivity and a configured SMTP account; a web-only workflow remains functional offline within the LAN. Servo motion is rate-limited to avoid stall currents; images are stored locally and referenced by the web UI via static file serving.

2.2. Hardware Components

Table 1 lists the main hardware. The LM393 comparator outputs a digital level ($D0$) that flips when the sound envelope crosses a trimmable threshold. An active buzzer module with *high-level trigger* is adopted (logic HIGH $\Rightarrow$ sound), which simplifies timing and avoids PWM tone synthesis. A single SG90 micro servo implements horizontal panning; the upper tilt servo serves as a fixed mechanical support. The external 5 V source (USB power bank or 5 V/2 A adapter) supplies the servo; the Raspberry Pi provides 3.3 V and 5 V only for sensors/logic. The OV5647 CSI camera captures images at up to 2592×1944.

Table 1: Bill of Materials and Roles.

Module	**Qty**	**Role / Notes**
Raspberry Pi 4B	1	Host; Flask server; GPIO control; runs `rpicam-still`.
OV5647 camera	1	Still imaging; CSI ribbon to Pi; used at 2592×1944.
LM393 sound sensor	1	Digital trigger via $D0$; on-board trimmer sets threshold; VCC=5 V.
Active buzzer (high-level trigger)	1	Audible alarm; logic HIGH $\Rightarrow$ ON; powered from Rasbperry Pi 5 V.
LED + resistor (220 Ω)	1	Visual alert; active-high from GPIO.
SG90 micro servo (pan)	1	Horizontal sweep (50 Hz PWM).
Pan-tilt bracket (upper SG90 as fixed tilt)	1	Mechanical mount; upper servo left unpowered or fixed at neutral.
External 5 V supply	1	USB power bank or 5 V/2 A adapter; feeds servo only.
Breadboard & jumpers	–	Prototyping; separate 3.3 V and 5 V rails; common ground.

2.2.1. Design Considerations.

Component selection followed a trade-off between availability, current draw, and software support. The LM393 was chosen over analog microphones because its digital threshold output simplifies GPIO edge detection and eliminates the need for ADC sampling. An active-high buzzer avoids PWM tone generation, reducing CPU load. A single SG90 servo satisfies the horizontal sweep requirement while keeping total current within the 5 V / 2 A budget. The OV5647 camera was selected for its native Raspberry Pi driver support and adequate still-image resolution for event verification.

2.2.2. Power budget and safety.

Table 2 summarizes nominal operating currents. The pan SG90 exhibits transient peaks approaching $\sim$0.5 A during start and stall, which previously caused overheated jumpers when powered from the Pi. Therefore, the **servo is powered from an external regulated 5 V rail**, while its PWM signal is still driven by the Pi. The active buzzer draws significantly less current (typically < 0.15 A) and is supplied from the Pi's 5 V rail without observable brownouts in our tests. All 5 V and 3.3 V domains share a **common ground**; the external $+5$ V is never back-fed into the Pi 5 V pin. If a higher-power buzzer is later adopted (> 0.15 A), it should be migrated to the external 5 V rail as well.

Table 2: Approximate current consumption and supply source.

Component	**Operating Voltage**	**Typical Current (A)**
Raspberry Pi 4 B (board)	5 V (main adapter)	0.6–1.0
LM393 sound sensor	3.3 V (from Pi)	0.02
LED (+resistor)	3.3 V logic (Pi)	0.01
Active buzzer (HIGH=ON)	5 V (from **Pi** 5 V)	0.06–0.15
SG90 servo (pan)	5 V (from **external**)	up to $\sim$0.5 (peaks)

Note. The Pi's 5 V header is a pass-through of the main adapter; practical limits are governed by the PSU rating and wiring. We use short, thicker jumpers for 5 V loads and verify that the buzzer's draw remains below $\approx$150 mA. Only grounds are bridged between the external 5 V rail and the Pi.

2.2.3. Interface summary.

Each module communicates through a dedicated GPIO line or interface: the sound sensor via digital D0, the LED and buzzer via single-bit GPIO outputs, and the servo via PWM (GPIO 12 @ 50 Hz). The camera connects through the CSI ribbon cable and is controlled in software using the `rpicam-still` utility.

2.3. Electrical Connections

The prototype uses two breadboards to separate the Pi 3.3 V and 5 V logic rail from the external 5 V high–current rail. One board hosts logic components (LM393, LED, GPIO interface) powered directly from the Raspberry Pi's volt pin, while the other board powers the servo motor from an independent 5 V battery pack or USB power bank. The two boards share a **common ground** connection, which is mandatory to provide a valid logic reference between Pi signals and 5 V peripherals.

2.3.1. Power domains.

- **3.3 V logic domain:** supplied by the Pi's 3.3 V pin (Pin 1); used for LED circuits. The LED connects from GPIO 22 through a 220 Ω resistor to its anode, with the cathode grounded.
- **5 V logic domain:** supplied by the Pi's 5 V pin (Pin 4); used for LM393 and active buzzer. The LM393's `VCC` pin is set to 5 V to ensure its digital output (`D0`) remains within the Pi's safe input range. The buzzer's VCC also goes to Pin 4 5 V, and its signal pin to GPIO 23.

- **5 V external domain:** powered by an external 5 V/2 A source for the SG90 servo. Its ground connect to the same GND rail that links back to the Pi's ground pin (Pin 6). The servo's red wire goes to external +5 V, the brown wire to GND, and the orange signal wire to GPIO 12 (PWM at 50 Hz).
- **Ground connection:** one wire bridges the two boards' GND rails to the Pi GND, maintaining a shared reference voltage. No other positive rail (3.3 V or 5 V) is cross–connected between the boards.

2.3.2. GPIO mapping and logic polarity.

Table 3 lists all pins in BCM numbering used in the final circuit.

Table 3: GPIO functions and logic polarity.

Function	**BCM Pin**	**Logic / Polarity**
Sound sensor `DO`	GPIO 25	Input, pull-up; trigger on LOW edge.
LED (visual alert)	GPIO 22	Active-high (1=ON, 0=OFF).
Active buzzer	GPIO 23	Active-high (1=ON, 0=OFF).
Servo (PWM signal)	GPIO 12	PWM, 50 Hz; 2–12% duty $\Rightarrow$ 0–180°.
Camera (OV5647)	CSI ribbon port	Controlled via `rpicam-still`.

2.3.3. Safety and wiring practices.

- High–current 5 V devices are *never* powered from the Pi's onboard 5 V pin. They use an external supply rated at least 2 A.
- All jumpers are inspected for discoloration or heat after extended testing; any overheated wire is replaced immediately.
- Before powering the system, confirm that only the GND rails are tied between 3.3 V and 5 V domains and that no positive rail is accidentally bridged.

2.3.4. Rationale for dual-breadboard layout.

Two breadboards are employed to ensure electrical isolation between the Raspberry Pi power domain and the external 5 V supply. The long board is connected directly to the Pi's 3.3 V and 5 V rails and hosts all low-power logic components such as the LED indicator. The short board is powered exclusively by an external 5 V source, supplying the servo motor. A single common ground wire bridges both boards to the Pi, providing a shared reference without mixing positive rails. This arrangement prevents back-feeding from the high-current devices into the Pi's power pins, minimizes voltage drop and noise coupling from servo switching, and allows each board to be tested independently (logic-only or actuator-only) before full integration.

2.4. Software Design

In this section, the design of each software module will be provided. The use hierarchy between the human user and the software modules will also be illustrated in Figure 2 as a directed acyclic

graph (DAG). According to Parnas (1978), two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification.

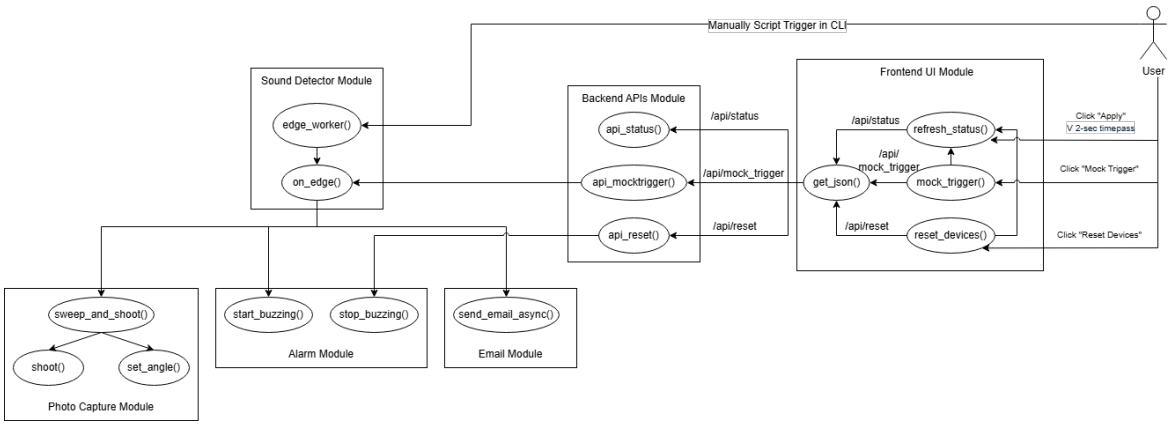

Figure 2: Software Use Hierarchy Diagram

2.4.1. Email Module

Exported Constants:

- EMAIL_FROM (string): Email address that will send the alarm message.
- EMAIL_PASS (string): Password of the EMAIL_FROM email address.
- EMAIL_TO (string): Email address to which the alarm message will be sent.
- SMTP_HOST (string): The address of the SMTP server that is responsible for sending and relaying the alarm email message.
- SMTP_PORT ($\mathbb{Z}$): The port of the SMTP_HOST address, which is responsible for sending and relaying the alarm email message.

Exported Access Programs:

Name	In	Out	Exceptions
send_email_async	subject: string, html_body: string	-	

Access Programs Semantics:

send_email_async($subject$, $html\_body$):

- transition: Start a new thread which logs into EMAIL_FROM with EMAIL_PASS and sends an email with $subject$ as title and $html\_body$ as its content, through SMTP_HOST and SMTP_PORT.

2.4.2. Alarm Module

Exported Constants:

- LED_PIN ($\mathbb{Z}$): GPIO pin number for LED light.
- BUZZ_PIN ($\mathbb{Z}$): GPIO pin number for buzzer.

Exported Access Programs:

Name	In	Out	Exceptions
start_buzzing	-	-	
stop_buzzing	-	-	

State Variables:
$state$: dict of ($buzzer$: $\mathbb{Z}$ from {0,1}, led: $\mathbb{Z}$ from {0,1}, $last\_event$: string, $last\_photo\_url$: string)

Access Programs Semantics:

start_buzzing():

- transition:
  - Set both GPIO outputs to LED_PIN and BUZZ_PIN to 1.
  - Set values of both $buzzer$ and led keys in dict $state$ to 1.

stop_buzzing():

- transition:
  - Set both GPIO outputs to LED_PIN and BUZZ_PIN to 0.
  - Set values of both $buzzer$ and led keys in dict $state$ to 0.

2.4.3. Photo Capture Module

Exported Constants:

- SERVO ($\mathbb{Z}$): GPIO pin number for the servo motor.
- PHOTO_DIR (string): File path to directory/folder that is used to contain the photos captured.
- RES_W ($\mathbb{Z}$): resolution width of the photos captured.
- RES_H ($\mathbb{Z}$): resolution height of the photos captured.
- ROTATE (string): the angle at which the photos captured will rotate, in units of degree.
- MOVE_SETTLE ($\mathbb{R}$): the length of time that the program will wait after each time the servo motor sweeping and before shooting, in units of second.
- ANGLES (list of $\mathbb{R}$): a list of angles which the servo motor will sweep to.

Exported Access Programs:

Name	In	Out	Exceptions
sweep_and_shoot	-	-	

State Variables:
$state$: dict of ($buzzer$: $\mathbb{Z}$ from {0,1}, led: $\mathbb{Z}$ from {0,1}, $last\_event$: string, $last\_photo\_url$: string)

Access Programs Semantics:

sweep_and_shoot():

- transition:
  - for each value in ANGLES, use set_angle($angle$) to control the servo motor to sweep to the corresponding angle, wait for MOVE_SETTLE-second length of time, then use shoot($angle$) to control the camera to take a picture and store it within PHOTO_DIR folder.
  - at the end, use set_angle($angle$) to control the servo motor to sweep back to the position of 90-degree.

Local Functions:
set_angle: $angle$: $\mathbb{R} \rightarrow$ None
set_angle $\equiv$ control the servo motor to the position of $angle$. $angle$ will be bounded within the range of [0,180]

shoot: $angle$: $\mathbb{R} \rightarrow$ None
shoot $\equiv$

- Runs `rpicam-still` to control the camera to take a picture with RES_W $\times$ RES_H resolution.
- store the image with naming convention ts_str_angle.jpg where ts_str is the timestamp, which the photo has been taken at, converted into a string.
- Modify $state$ to set value of $last\_photo\_url$ key to the file path where the picture has been stored at, and value of $last\_event$ key to ts_str.

2.4.4. Sound Detector Module

Uses:
Email Module (Section 2.4.1), Alarm Module (Section 2.4.2), Photo Capture Module (Section 2.4.3)

Exported Constants:

- SND_PIN ($\mathbb{Z}$): GPIO pin number the sound sensor.
- DEBOUNCE_MS ($\mathbb{Z}$): The time window (in milliseconds) during which further edge detections are ignored after the first one, will be used to debounce mechanical switches.

Exported Access Programs:

Name	In	Out	Exceptions
edge_worker	-	-	
on_edge	-	-	

State Variables:
$state$: dict of ($buzzer$: $\mathbb{Z}$ from {0,1}, led: $\mathbb{Z}$ from {0,1}, $last\_event$: string, $last\_photo\_url$: string)
$alarm\_active$: $\mathbb{B}$

Access Programs Semantics:

edge_worker():

- transition:
  - Utilize $RPi.GPIO.add\_event\_detect$ to add an event detection for pin SND_PIN with a bounce time window of length DEBOUNCE_MS.
  - When an event is detected on SND_PIN, call $on\_edge()$.

on_edge():

- transition:
  - When GPIO input signal from SND_PIN is equal to 0 and $alarm\_active$ is $False$, set $alarm\_active$ to $True$ and the value of $last\_event$ key in $state$ to timestamp of current time.
  - Utilize $start\_buzzing$ exported from Alarm Module (Section 2.4.2) to make the buzzer start buzzing and turn the LED light on.
  - Utilize $send\_email\_async$ exported from Email Module (Section 2.4.1) to send an alarm-notification email.
  - Create a new thread and utilize $sweep\_and\_shoot$ exported from Photo Capture Module (Section 2.4.3) to sweep servo motor and take pictures.

2.4.5. Backend APIs Module

Uses:
Alarm Module(Section 2.4.2), Sound Detector Module(Section 2.4.4)

Exported Access Programs:

Name	In	Out	Exceptions
api_status	-	status_data_json: JSON string	
api_reset	-	reset_result_json: JSON string	
api_mock_trigger	-	trigger_result_json: JSON string	

State Variables:
$state$: dict of ($buzzer$: $\mathbb{Z}$ from {0,1}, led: $\mathbb{Z}$ from {0,1}, $last\_event$: string, $last\_photo\_url$: string)
$alarm\_active$: $\mathbb{B}$

Access Programs Semantics:

api_status():

- output:
  Upon GET request to URL $/api/status$:
  out := JSON string created using a dict with keys:
  - $buzzer$:= value of key $buzzer$ in $state$
  - led := value of key led in $state$
  - $last\_event$:= value of key $last\_event$ in $state$
  - $last\_photo\_url$:= value of key $last\_photo\_url$ in $state$
  - $alarm\_active$:= value of $alarm\_active$

api_reset():

- transition:
  Upon POST request to URL $/api/reset$:
  - Utilize $stop\_buzzing$ exported from Alarm Module (Section 2.4.2) to make the buzzer stop buzzing and turn the LED light off.
  - Set $alarm\_active$ to $False$ and the value of $last\_event$ key in $state$ to timestamp of current time.
- output:
  out := Upon successful transition, outputs JSON string created using a dict with keys:
  - ok := $True$
  - $time$:= timestamp of current time
  - $buzzer$:= value of key $buzzer$ in $state$
  - led := value of key led in $state$
  - $last\_event$:= value of key $last\_event$ in $state$
  - $alarm\_active$:= value of $alarm\_active$

api_mock_trigger():

- transition: Upon POST request to URL $/api/mock\_trigger$, Utilize on_edge exported from Sound Detector Module (Section 2.4.4) to perform a mock noise trigger.
- output: out := Upon successful transition, outputs JSON string created using a dict (ok := $True$)

2.4.6. *Frontend UI Module*

Uses:
Backend APIs Module (Section 2.4.5)

Exported Constants:

- BASE (string): API-base string that the frontend will use to make API calls

Exported Access Programs:

Name	In	Out	Exceptions
refresh_status	-	-	FetchError
mock_trigger	-	-	FetchError
reset_devices	-	-	FetchError

State Variables:
$apiBase$: dict of ($value$: string)
$status$: dict of ($innerText$: string, $className$: string)
$buzzerText$: dict of ($innerText$: string, $className$: string)
$buzzerTime$: dict of ($innerText$: string)
$alarmBadge$: dict of ($innerText$: string, $className$: string)
$photo$: dict of (src: string, $innerText$: string)
$photorTime$: dict of ($innerText$: string)

Access Programs Semantics:

refresh_status():

- transition:
  - $backend\_data :=$ get_json("BASE/api/status", {method: "GET"})
  - $isAlarm := (backend\_data.alarm\_active) \vee (backend\_data.buzzer = 1)$
  - $buzzerText.innerText :=$
    $(isAlarm) \Rightarrow 'Buzzing(ALARM)'$
    $\neg(isAlarm) \Rightarrow 'Idle'$
  - $buzzerText.className :=$
    $(isAlarm) \Rightarrow 'value\,bad'$
    $\neg(isAlarm) \Rightarrow 'value\,ok'$
  - $alarmBadge.innerText :=$
    $(isAlarm) \Rightarrow 'Alarm : ACTIVE'$
    $\neg(isAlarm) \Rightarrow 'Alarm : idle'$
  - $alarmBadge.className :=$
    $(isAlarm) \Rightarrow 'pill\,bad'$
    $\neg(isAlarm) \Rightarrow 'pill\,ok'$
  - $buzzerTime.innerText :=$
    $(backend\_data.last\_event = None) \Rightarrow '--:--:--'$
    $\neg(backend\_data.last\_event = None) \Rightarrow backend\_data.last\_event$

- $photo.src :=$
  $\neg(backend\_data.last\_photo\_url = None) \Rightarrow$ value of BASE + $backend\_data.last\_photo\_url$ + '?t=now_ts' where now_ts is the timestamp of current time computed using $Date.now$
- $photoTime.innerText :=$
  $\neg(backend\_data.last\_event = None) \wedge \neg(backend\_data.last\_photo\_url = None) \Rightarrow backend\_data.last\_event$
  $\neg(backend\_data.last\_event = None) \wedge \neg(backend\_data.last\_photo\_url = None) \Rightarrow$ $'--:--:--'$
- $status.innerText := 'OK'$
- $status.className := 'pill\ ok'$

- exception: When an error is thrown from get_json function call, throw FetchError

mock_trigger():

- transition:
  - Utilize get_json("BASE/api/mock_trigger", {method: "POST"}) to make POST request to URL $/api/mock\_trigger$
  - Utilize $refresh\_status()$ to refresh status of the frontend UI
- exception: When an error is thrown from get_json function call, throw FetchError

reset_devices():

- transition:
  - Utilize get_json("BASE/api/reset", {method: "POST"}) to make POST request to URL $/api/reset$
  - Utilize $refresh\_status()$ to refresh status of the frontend UI
- exception: When an error is thrown from get_json function call, throw FetchError

Local Functions:
get_json: url: string, $opts$: dict of ($method$: string from {"GET", "POST", "PUT", "DELETE"})
$\rightarrow$ Javascript Object data type (i.e. key-value pairs)
get_json $\equiv$

- Utilize $fetch$ to make API call to url with $method$ specified in $opts$
- $response := fetch(url, opts)$
- $\neg(response.ok) \Rightarrow$ throw FetchError
- $\neg(response.ok) \Rightarrow status.innerText := 'Error'\ status.className := 'pill\ bad'$
- $output := response.json()$

3. Implementation and Testing

3.1. Safety and Power Test

3.1.1. Objective

To verify that the system wiring and power delivery are safe and that the 5 V rail remains within tolerance under idle and peak operating conditions, preventing undervoltage events and thermal hazards.

3.1.2. Equipment

Raspberry Pi (with one microphone sensor, one LED module, one buzzer, and one camera/servo attached); External 5V supply; Breadboard & jumper wires.

3.1.3. Method

1. **Wiring Inspection** Confirm correct polarity; ensure a common ground between the Raspberry Pi, servo, and sensors; verify GPIO assignments for the microphone, LED, and buzzer; inspect for exposed conductors and adequate strain relief.
2. **Idle Power Verification** Power the system, start the Flask application, and measure the 5 V rail and current draw at the GPIO header. Check for absence of Raspberry Pi undervoltage indicators.
3. **Peak Load Assessment** Induce an alarm event (acoustic trigger or `/api/mock_trigger`) to activate the servo sweep and buzzer. Record peak current and minimum rail voltage observed during motion.
4. **Thermal Check** After mixed operation for ≥ 5 min, check the components' temperatures (servo housing, power connectors).
5. **Emergency Reset** Issue `POST /api/reset` during an active alarm and observe immediate silencing of audible/visual alerts and continued application responsiveness.

3.1.4. Results and Acceptance Criteria

- The 5 V rail remained within 4.8–5.2 V at idle and under peak load.
- No undervoltage warnings or unintended reboots were observed.
- Component temperatures remained within touch-safe limits.
- The reset command reliably terminated alerts without destabilizing the service.

3.2. Functional Tests

3.2.1. Objective

To ensure the whole system works well, including noise detection, alerting, LED lighting, image capture, data storaging, and email notification.

3.2.2. Test Cases and Procedures

1. **Noise Detection** Generate a brief, high-amplitude sound near the microphone (or invoke `/api/mock_trigger`). *Expected:* `alarm_active = true`; buzzer and LED engaged; corresponding log entry appended; `GET /api/status` reflects the active state.

2. **Servo Sweep and Imaging** *Expected:* Run the code and the servo motor rotates to configured angles (e.g., 20°, 60°, 100°, 140°). A photograph is stored at each position under `captures/`, and `last_photo_url` references the most recent image.

3. **Logging Integrity** Trigger and reset the system, then query `/api/status`. *Expected:* The `logs` array contains recent events (limited to the most recent entries), including trigger, photo capture, and reset messages.

4. **Email Notification** Set `EMAIL_FROM`, `EMAIL_PASS`, and SMTP settings, then trigger an alarm. *Expected:* A single notification email is received; server response is unaffected due to asynchronous sending.

5. **Debounce/Re-trigger Behavior** Produce several rapid noises. *Expected:* A single alarm cycle persists until a reset is issued; overlapping sequences are prevented by design.

6. **Manual Reset** Invoke `POST /api/reset` during an active alarm. *Expected:* Immediate silencing of alerts; `alarm_active = false`; `last_event` updated; stable JSON response returned.

3.2.3. Results and Acceptance Criteria

All test cases executed as expected: one alarm cycle per trigger, completion of the angle sequence with valid images saved, consistent and bounded logs, and reliable reset behavior, one email was delivered per alarm.

3.3. Web Control Test

3.3.1. Objective

Validate that the Flask REST API and the custom web dashboard operate together to provide reliable remote monitoring (status and latest photo) and control (mock trigger and reset).

3.3.2. System Setup

- **Backend:** Flask on the Raspberry Pi (`http://<Pi-IP>:5000`) exposing `/api/status`, `/api/reset`, `/api/mock_trigger`, and serving images from `captures/`.

- **Frontend:** Single-page dashboard with an API Base input (Apply), a status badge, *Reset Devices* and *Mock Trigger* buttons, a *Latest Photo* panel bound to `last_photo_url`, and periodic polling of `GET /api/status`.

3.3.3. Procedure

1. **Connectivity & Base URL:** Enter the Flask base URL and click Apply. *Expected:* badge shows "OK" and current state fields populate.

2. **Status Polling:** Let the page run for $\geq$30 s. *Expected:* repeated `GET /api/status` $\rightarrow$ HTTP 200; UI text (Idle/Alarm) updates without reload.

3. **Mock Trigger:** Click *Mock Trigger*. *Expected:* `alarm_active = true`; buzzer/LED indicators on; after the sweep, `last_photo_url` updates and the *Latest Photo* panel refreshes.

4. **Reset:** Click *Reset Devices* during an active alarm. *Expected:* `POST /api/reset` $\rightarrow$ HTTP 200; alerts silence immediately; `alarm_active=false`; `last_event` updated; UI reflects the new state on the next poll.

5. **Image Serving:** Open the URL in Latest Photo. *Expected:* image loads from `/captures/` and timestamp matches the logs/status.

6. **Resilience & Latency:** Temporarily break connectivity. *Expected:* dashboard shows "disconnected" without crashing and recovers when the backend returns.

3.3.4. Acceptance Criteria

The dashboard connects and updates within one polling cycle; *Mock Trigger* starts exactly one alarm sequence; *Reset* terminates alerts immediately and returns consistent JSON; the Latest Photo is always valid and accessible.

4. Discussion

The Smart Sound-Activated Camera Surveillance System shows that a small stack running on the edge can provide relevant visibility without the cloud or heavyweight middleware.

We found several design decisions helpful in practice. First, latching the alarm until someone resets it matches human workflows. A transient alert might go unnoticed, but a persistent buzzer and blinking LED force interaction. Second, taking multiple angles gives better context than a single snapshot. The 130ms servo sweep yields a series of burst views with negligible additional latency. Third, a simple REST interface and single page dashboard reduce configuration overhead and failure modes compared with MQTT and WebSockets. Polling every two seconds is enough for a human interaction system, and keeps the browser and device loosely coupled. Fourth, the former in favor of robustness using edge detection with debouncing and a fallback to polling when interrupts are unavailable in user code. Background threads handle email delivery and photo capture outside the real-time alarm path.

These decisions come with tradeoffs and limitations. The digital sound sensor is sensitive to ambient noise and may trigger false positives in reverberant rooms or near appliances; we rely on simple debouncing rather than adaptive thresholds or classification. REST polling means the visualization is up to two seconds behind the event, acceptable for this use case but not high tempo automation. This is a classroom and lab prototype, serving images and APIs over plain HTTP on the local network, setup is simplified without authentication, but this is not safe for untrusted networks(Gourley & Totty, 2002). Basic storage management just gives images until you clear them. The servo is mechanical pain to deal with and must be mounted properly to minimize jitter (Low light = worse pictures). Finally the single node is a single point of failure. If power is lost, there is no redundancy or buffer for alarms from the device.

A few improvements could be made. On the sensing side, removing the edge trigger and replacing it with lightweight on-device audio classification, or using it in conjunction would eliminate false positives. Further tuning of the debounce and quiet hours scheduling could make it more responsive. On the networking side, Server Sent Events or WebSocket's could provide lower latency on data updates while still being configured mostly via HTTP(Pimentel & Nickerson, 2012). Security hardening should enable HTTPS (self signed / LAN issued certs), API keys or other token based access control, and disabling optional LAN discovery.

Persistent logs and image retention policies could provide better operational durability. A watchdog service and power loss graceful handling could be added, and containerization could enable reproducible deployment. Finally, small usability improvements like manual snapshot, angle control, and a mobile friendly progressive web app would make this easier to run day by day.

5. Conclusion

This work provides an end-to-end, reproducible, reference design for a sound triggered, latched alarm camera platform running on a Raspberry Pi. A simple acoustic trigger combined with persistent alerting, multi-angle imaging, a minimal REST API, and a simple browser dashboard enables this system to be both low-latency, operationally transparent, and have strong data locality with few dependencies. Our prototype demonstrates that reasonable household and lab monitoring can be implemented without cloud services or messaging infrastructure.

The design is limited today in terms of achievable robustness of sensing, security, and long-time operational reliability, but these can be addressed within the same architecture via sensor fusion, encrypted and authenticated transport, and some modest systems tooling to produce a small base platform that can be configured for a range of different small space scenarios where fast local awareness and a human acknowledgment is more valuable than remote analysis.

6. Appendix

System Assembly

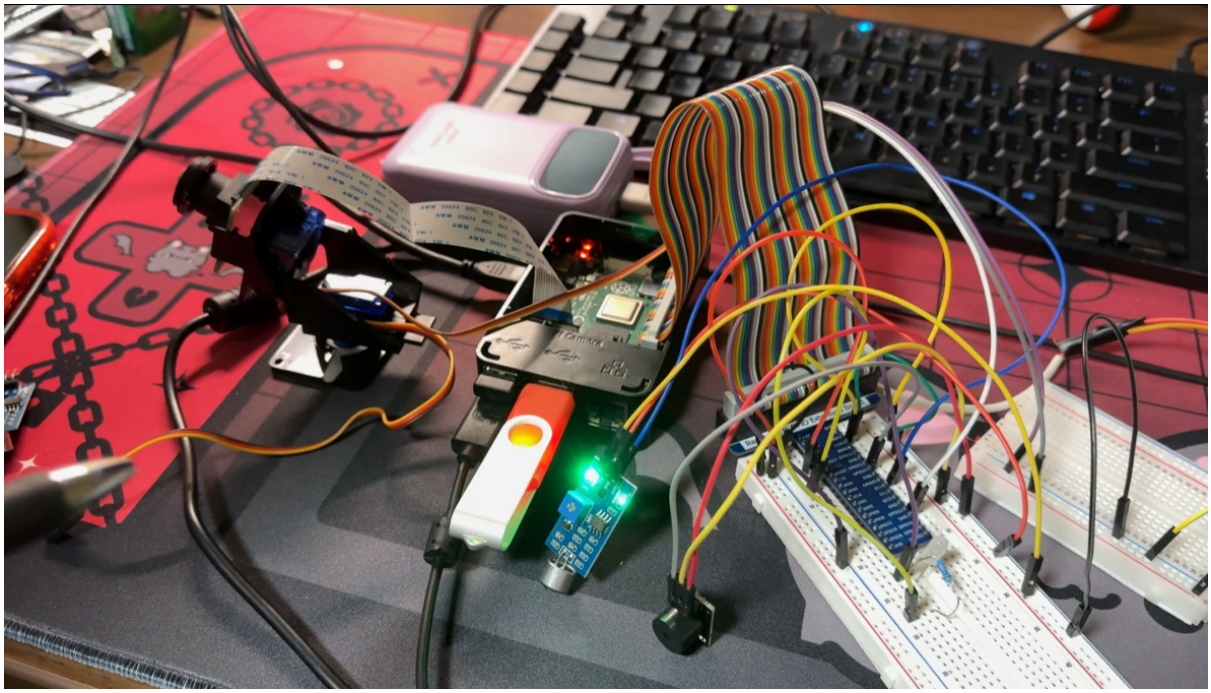

Figure 3: System Assembly

Web page

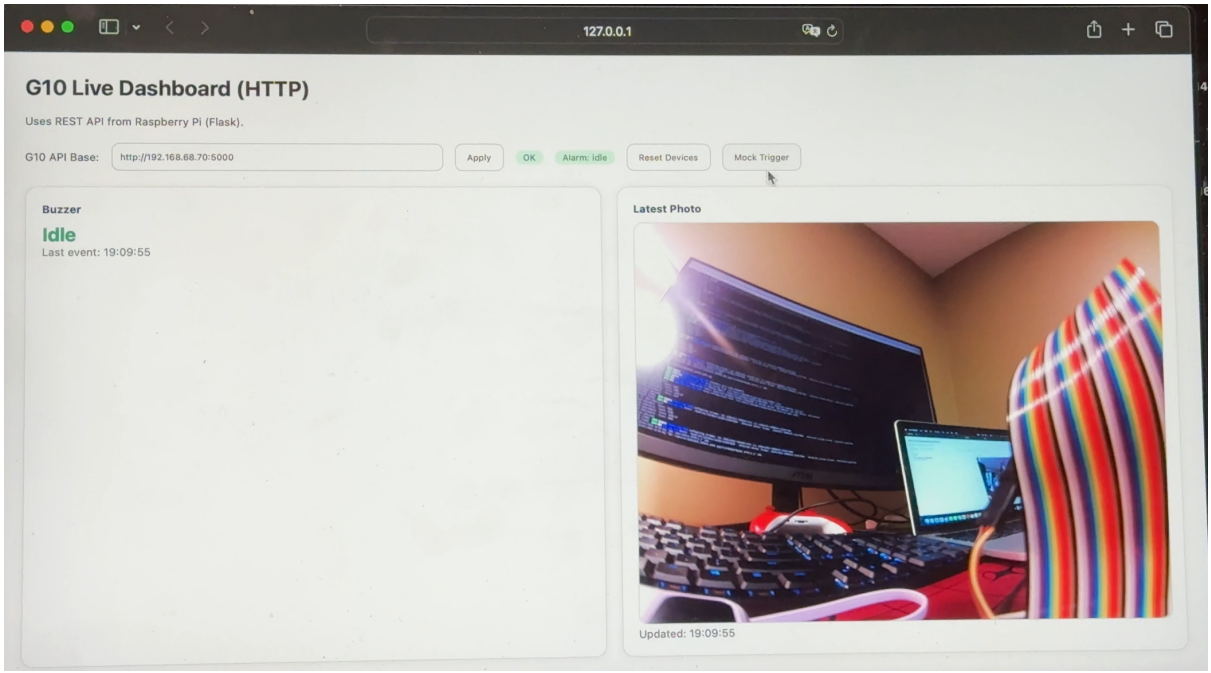

Figure 4: Web page

REST API Reference

GET `/api/status` Returns:

```
{
  "buzzer": 0|1,
  "led": 0|1,
  "last_event": "HH:MM:SS",
  "last_photo_url": "/captures/...jpg",
  "alarm_active": true|false,
  "logs": [ ... ]
}
```

POST `/api/reset` Returns:

```
{
  "ok": true,
  "time": "...",
  "alarm_active": false,
  "buzzer": 0,
  "led": 0
}
```

POST `/api/mock_trigger` Returns:

```
{ "ok": true }
```

Images are served statically at `/captures/`*`filename`*`.jpg`.

Software setup (Raspberry Pi OS, 64-bit)

1. Update packages and install dependencies:

   ```
   sudo apt update && sudo apt install -y \
     python3 python3-pip python3-flask libcamera-apps
   ```

2. Install Python packages:

   ```
   pip3 install flask-cors RPi.GPIO
   ```

3. Enable the camera (libcamera is default on recent Pi OS). Verify capture:

   ```
   rpicam-still -n -o test.jpg
   ```

4. Clone or copy the backend script and the HTML dashboard.

5. Configure email alert environment variables:

   ```
   export EMAIL_FROM="your@gmail.com"
   export EMAIL_PASS="your_app_password"
   export SMTP_HOST="smtp.gmail.com"
   export SMTP_PORT=465
   ```

6. Run the server with GPIO privileges:

   ```
   sudo -E python3 smart_sound_backend.py
   ```

7. Open the HTML file in a browser, set the API base to `http://<pi_ip>:5000`, and click *Apply*.

7. Acknowledgments

We appreciate the guidance of our instructor and teaching assistants about practical aspects of embedded systems implementation and safety, and the early feedback about usability and testing scenarios by our classmates. Our group recognizes the maintainers of Flask, Flask-CORS, and RPi.GPIO for the open-source libraries used in this work.

References

Gourley, D. and Totty, B. (2002). *HTTP: The Definitive Guide*. O'Reilly Media, Inc.

Kumar, S. S., Sushmitha, M., Sirisha, P., Shilpa, J., and Roopashree, D. (2018). Sound activated wildlife capturing. In *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 2250–2253. IEEE.

Nunes, L. H., Nakamura, L. H., Vieira, H. D. F., Libardi, R. M. D. O., de Oliveira, E. M., Estrella, J. C., and Reiff-Marganiec, S. (2014). Performance and energy evaluation of restful web services in raspberry pi. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pages 1–9. IEEE.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA. IEEE Press.

Pimentel, V. and Nickerson, B. G. (2012). Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53.