SISURAKSHA - INTELLIGENT CHILD SAFETY PLATFORM FOR SCHOOL BUSES

ML-Based Driver Monitoring and Footboard Safety System

R.I.S.R. Pinto

IT22610102

Group Number: 25-26J-282

B.Sc. (Hons) Degree in Information Technology Specialized in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology

Sri Lanka

April 2026

DECLARATION

I declare that this is my own work and this report does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or institute of higher learning. To the best of my knowledge and belief, it does not contain any material previously published or written by another person except where acknowledgement is made in the text.

Signature of the Supervisor Date

.....

Candidate Declaration Details

Name	Student ID	Signature
R.I.S.R. Pinto	IT22610102	.....

ABSTRACT

School bus safety is an important area in child transportation because students can be exposed to risk during boarding, travelling, and exiting. One of the most sensitive areas of a school bus is the entrance and footboard. A student standing near the footboard while the bus is stopped requires driver attention before the vehicle moves. However, a student standing near the footboard while the bus is moving becomes a critical safety condition that can cause serious accidents. This individual report presents the footboard safety and experimental driver monitoring work carried out under the SISURAKSHA intelligent child safety platform for school buses.

The main validated contribution of this individual work is the footboard safety monitoring module with movement-based risk classification. The module uses E18-D80NK infrared proximity sensors to identify footboard occupancy and an AI-supported camera input to provide visual detection support. The first movement detection idea used GPS-based speed readings. However, GPS was found to be unsuitable for short-distance and low-speed movement decisions required in footboard safety. Therefore, the movement detection method was changed to a Hall sensor pulse-based method. A Hall Sensor Module LM393 Linear Hall Effect sensor is mounted near the bus propeller shaft using a metal arm fixed to the chassis, while 10 mm x 5 mm magnets are attached to the shaft. When the shaft rotates, magnetic pulses are detected through GPIO 33 and the system identifies that the bus is moving.

The final risk logic is simple and practical. If no object or student is detected near the footboard, the state is SAFE. If the footboard is occupied while the bus is stopped or at a halt, the driver receives a WARNING. If the footboard is occupied while the bus is moving, the system changes the state to CRITICAL. The hardware implementation also includes an LM2596 buck converter to step down the 24V bus battery supply to 5V for the ESP32 and connected devices, a DFPlayer module with 4 ohm 3W speaker for audio alerts, and a DS-212 push button for manual input. The report also includes a driver monitoring prototype. Since this module produced several false detections due to camera angle, lighting, calibration sensitivity, and vibration, it is discussed as an experimental component rather than the main successful contribution.

Keywords - School bus safety, footboard detection, Hall effect sensor, ESP32, E18-D80NK, YOLOv8, driver monitoring, IoT.

ACKNOWLEDGEMENT

First and foremost, I would like to express my sincere gratitude to my supervisor for the guidance, feedback, and support provided during this research project. The discussions and suggestions received throughout the project helped me to improve the technical direction of my individual component and to identify practical issues in the implementation.

I also wish to thank the Sri Lanka Institute of Information Technology, the Department of Information Technology, and the academic staff who supported the research process. I am grateful to my project group members for their collaboration during the development of the SISURAKSHA platform. Their individual modules, ideas, and discussions helped the overall system to become more complete.

Finally, I thank my family and friends for their support, encouragement, and patience throughout the research period. I also appreciate the support received during prototype testing, real bus observation, hardware setup, image collection, and technical documentation.

TABLE OF CONTENTS

Draft Table of Contents

Note: These page numbers are draft values. After inserting the final figures and screenshots, update the table of contents in Microsoft Word.

LIST OF FIGURES

List of Figures

LIST OF TABLES

List of Tables

LIST OF ABBREVIATIONS

List of Abbreviations

Abbreviation	Full Term
AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolutional Neural Network
DFPlayer	Mini MP3 Player Module
ESP32	Espressif 32-bit Microcontroller
GPS	Global Positioning System
GPIO	General Purpose Input Output
IoT	Internet of Things
IR	Infrared
LM393	Comparator-based Hall Sensor Module
ML	Machine Learning
OpenCV	Open Source Computer Vision Library
PERCLOS	Percentage of Eye Closure
YOLO	You Only Look Once Object Detection Model

1 INTRODUCTION

1.1 General Introduction

Student transportation is an important responsibility for schools, parents, and transport providers. A school bus carries children who may not always understand the danger around a moving vehicle. Therefore, even a small unsafe situation near the entrance of the bus can become serious if it is not noticed by the driver at the correct time. This research was carried out under the SISURAKSHA intelligent child safety platform, where the main focus of this individual component is the footboard safety system and the supporting driver monitoring prototype.

In many school buses, the footboard is the area between the road surface and the bus interior. Students use this area when entering and leaving the bus. In normal conditions, a student may stand on or near the footboard for a short time while the bus is stopped. However, if the bus starts moving while a student is still standing near the footboard, the situation becomes unsafe. The driver may not always see the footboard clearly from the driving position, especially when the bus is crowded, when the door area is blocked, or when the driver is concentrating on traffic.

The proposed component tries to reduce this issue by combining hardware sensors, AI-based visual support, and movement detection. The idea is not only to detect whether something is near the footboard, but also to understand whether the bus is stopped or moving. This difference is important because the same footboard occupancy condition has different risk levels depending on vehicle movement. If the bus is stopped at a halt and someone is on the footboard, the system should warn the driver before movement starts. If the bus is already moving and someone is detected at the footboard, the system should identify this as a critical condition.

The original implementation considered GPS speed readings to identify movement. However, during analysis it became clear that GPS is not accurate enough for the type of short-distance and low-speed decisions required in footboard safety. School buses can move slowly during pickup or drop-off, and even a small forward movement can be dangerous if a student is standing near the footboard. Therefore, the improved implementation uses a Hall Sensor Module LM393 Linear Hall Effect sensor and magnets fixed to the propeller shaft of the bus. The shaft rotates when the bus moves, and each magnet passing the sensor generates a pulse. This makes the movement detection more directly connected to the mechanical motion of the vehicle.

The system is supported by ESP32-based hardware. The E18-D80NK infrared proximity sensors are used to identify object or student presence near the footboard steps. The Hall sensor pulse is read through GPIO 33. The audio alert path uses a DFPlayer module connected through GPIO 32 and a 4 ohm 3W speaker. A DS-212 red push button is connected through GPIO 35 for manual input. In the real bus environment, power is taken from the 24V bus battery and converted to 5V using an

LM2596 buck converter. This power design is important because the ESP32 and connected sensing devices cannot be powered directly from the 24V bus supply.

A driver monitoring module was also developed as part of the overall title. It uses camera-based face analysis and machine learning support to detect driver states such as looking away, looking down, phone use, yawning, and drowsiness. However, the driver monitoring module was not fully successful in final testing. It generated false detections and unstable results under changes in lighting, camera position, face angle, calibration, and expected bus vibration. For that reason, this report treats driver monitoring as an experimental prototype and does not overstate it as the main research contribution. The main validated part of the individual work remains the footboard safety module and the improved movement-based risk classification.

1.2 Background Literature

1.2.1 Child Safety in School Bus Transportation

School bus safety depends on several factors including vehicle condition, driver attention, passenger behaviour, route environment, and the ability to detect unsafe conditions early. Children are more vulnerable than adults because they can react unpredictably near vehicles. They may stand close to the entrance, hold the door frame, or remain on the footboard when the bus is preparing to move. These issues are practical and visible in real transportation environments, especially during school pickup and drop-off times.

Technology-based safety systems can support the driver by continuously monitoring areas that may not always be visible. A sensor-based system does not replace the driver, but it can act as an additional warning layer. For a school bus, this is useful because the driver has to monitor the road, mirrors, passengers, and the stopping area at the same time. A footboard monitoring system can reduce the chance that the driver accidentally starts moving while a child is near the entrance.

1.2.2 Footboard Safety Problem in School Buses

The footboard area is a transition zone between the inside of the bus and the outside environment. It is not designed for travelling, but students may sometimes stand there when the bus is crowded or during boarding. The danger increases when the vehicle begins to move because the student may lose balance or be pushed toward the outside of the bus. Therefore, the footboard problem is not only an object detection problem; it is also a movement-aware risk classification problem.

A simple sensor that only detects a person near the step can produce unnecessary alarms if the bus is stopped. However, if the same detection happens while the bus is moving, the alert must be urgent. This is why the proposed system separates the footboard condition into SAFE, WARNING, and CRITICAL states. This decision structure makes the system more practical because it can warn the driver during a halt and escalate the alert only when the bus moves while the footboard is occupied.

1.2.3 Sensor-Based Occupancy Detection

Infrared proximity sensors are commonly used in small embedded systems to detect whether an object is present within a certain range. In this project, E18-D80NK infrared proximity sensors are used to monitor the footboard step area. The advantage of this sensor type is that it gives a simple digital output that can be read by a microcontroller such as the ESP32. This makes the system easier to implement in a prototype and easier to test through a dashboard or serial monitor.

The footboard area is divided into step positions so that the system can identify whether the top, middle, or bottom area is occupied. In the proposed implementation, S1 is used for the top step, S2 for the middle step, and S3 for the bottom step. In the final risk logic, any of these detections means that the footboard is occupied. The movement condition then decides whether the final state is WARNING or CRITICAL.

1.2.4 AI-Based Visual Support for Footboard Detection

Computer vision can support sensor detection by identifying visual patterns around the footboard. This is useful because hardware sensors can sometimes be affected by placement, object angle, wiring issues, or reflection. The system uses YOLOv8-based detection with OpenCV runtime support to process camera input and display the current monitoring state. Dataset collection and annotation were carried out using Roboflow, and model training was supported using Google Colab.

The AI component should not be treated as the only source of safety decision in this project. Instead, it acts as a supporting input together with IR sensor detection. This combined approach is more suitable for a safety context because the system can continue to detect footboard occupancy even when one input is uncertain. In the Python fusion runtime, the AI and sensor inputs are used together before the final risk classification is made.

1.2.5 Limitations of GPS for Footboard Movement Detection

GPS-based speed estimation was considered at the beginning of the project because GPS speed is easy to obtain from mobile devices and can represent vehicle movement in many general applications. However, footboard safety needs a more sensitive movement decision. A school bus may move only a short distance or move at a very low speed while leaving a halt. In such cases, GPS readings can be delayed, noisy, or not accurate enough to identify movement at the exact time needed for safety.

Because of this limitation, GPS was not suitable as the final movement detection method for the footboard module. The report therefore describes GPS only as the initial approach and explains why it was replaced. This is an important research decision because it shows that the final design was improved based on practical limitations found during development.

1.2.6 Hall Effect Sensor-Based Movement Detection

A Hall effect sensor detects changes in magnetic field. In this project, the Hall Sensor Module LM393 Linear Hall Effect sensor is used with magnets attached to the bus propeller shaft. When the bus moves, the propeller shaft rotates. Each time a magnet passes close to the sensor, the magnetic field changes and the sensor produces a pulse. The ESP32 reads this pulse through GPIO 33 and uses it to decide whether the bus is moving.

This method is more suitable for footboard safety because it is directly connected to mechanical movement. It does not depend on GPS signal, satellite visibility, or location change over distance. The system currently uses the Hall sensor mainly to identify moving or stopped state. In future improvements, the same pulses can be counted over time to estimate shaft rotation speed and then calculate approximate vehicle speed after calibration.

1.2.7 Embedded IoT and Backend Communication

The ESP32 microcontroller is suitable for this type of safety prototype because it supports GPIO input, Wi-Fi communication, and integration with web dashboards or external servers. The system reads IR sensor states, Hall sensor pulses, and manual input. It then communicates with the Python runtime and backend API. The backend was implemented using Node.js, while Supabase and MongoDB were used in the broader platform for data storage and management.

During final documentation, backend database screenshots were not captured. However, backend server execution and Python terminal communication evidence were collected. Therefore, the backend part is presented as an implemented integration supported by terminal and runtime evidence, while full database-level verification is included as a limitation and future improvement.

1.2.8 Driver Monitoring as an Experimental Extension

Driver monitoring is a useful concept in intelligent transport systems because driver distraction, fatigue, and phone use can affect passenger safety. The prototype driver monitoring module uses camera input, MediaPipe FaceMesh, EAR, MAR, head pose estimation, and YOLOv8 phone detection. It also includes calibration logic to adjust to the driver face and camera angle.

Although the implementation was technically completed, the module was not fully reliable in practical testing. It was sensitive to lighting, calibration, head angle, and camera placement. In a real bus, vibration and movement may make these issues worse. Therefore, the driver monitoring module is included in this report as an experimental prototype, while the main successful contribution remains the footboard safety and movement detection module.

1.3 Research Gap

Although smart school bus systems have improved in recent years, most existing systems mainly focus on GPS-based bus tracking, RFID-based student attendance, route monitoring, cloud databases, and parent notification [5], [6]. These systems are useful for improving communication between the school, driver, and parents. However, they do not directly address the immediate danger that occurs at the entrance or footboard area when a child remains close to the bus steps while the vehicle begins to move.

The footboard safety problem is different from general bus tracking because it requires a fast and localized decision. A student standing near the footboard while the bus is stopped may be a normal boarding or alighting condition. In this case, the driver should receive a warning before moving. However, if the bus starts moving while the footboard is still occupied, the same condition becomes critical. Therefore, the safety decision must consider both footboard occupancy and actual vehicle movement.

Existing GPS-based school bus systems can identify the general location of the bus, but GPS is not suitable for detecting very short-distance and low-speed movement at the exact moment the bus starts moving from a halt [5], [6]. This creates a gap for a non-GPS movement detection method that can detect actual mechanical movement of the bus. Hall effect sensing is suitable for this purpose because magnetic pulses can be used to identify rotation without physical contact [9]. In this research, this principle is applied by placing magnets on the propeller shaft and reading pulse signals using a Hall Sensor Module LM393.

Another gap is that camera-based safety systems and YOLO-based models can support detection around bus boarding and alighting areas, but vision-only systems can be affected by occlusion, lighting, crowding, and camera position [10], [11]. Multi-sensor fusion literature shows that combining different sensing methods can improve reliability because each sensor type has different strengths and weaknesses [12]. Therefore, this research addresses the gap by combining IR proximity sensors, Hall sensor-based movement detection, AI-supported visual detection, ESP32-based processing, and real-time alerting.

Driver monitoring is also included in the project title, but the literature shows that driver monitoring systems based on eye aspect ratio, PERCLOS, head pose, and phone detection still face practical limitations in real-world environments [13]–[17]. Therefore, this report treats driver monitoring as an experimental prototype, while the main validated contribution is the movement-aware footboard safety module.

Table 1.1: Summary of Research Gap

Area	Existing limitation	How this work addresses it
Footboard monitoring	Often not treated as a separate safety risk area	Focuses directly on bus entrance and footboard occupancy

Movement decision	GPS is not accurate enough for low-speed short-distance movement	Uses Hall sensor pulses from propeller shaft rotation
Risk classification	Occupancy alone can cause unnecessary alerts	Separates SAFE, WARNING, and CRITICAL using movement state
Driver monitoring	Prototype systems can fail in uncontrolled environments	Reports driver monitoring as an experimental module with limitations

1.4 Research Problem

The main research problem addressed in this study is how to design and implement a low-cost school bus safety system that can detect footboard occupancy, identify whether the bus is moving, and classify the safety condition in real time.

Existing smart school bus systems commonly focus on GPS tracking, RFID attendance, route monitoring, and parent notification [5], [6]. Although these functions are useful, they do not directly detect the immediate hazard that occurs when a child remains at the footboard or entrance area while the bus starts moving. This creates a practical safety problem because the driver may not always have a clear view of the footboard area during pickup and drop-off.

A system that only detects footboard occupancy is not sufficient because students may naturally stand near the entrance while the bus is stopped. In this situation, the system should warn the driver before movement begins. However, when the bus is moving and the footboard is occupied, the risk becomes critical and the driver should be alerted immediately. Therefore, the problem requires both occupancy detection and reliable movement detection.

The initial idea of using GPS speed was not suitable for this requirement because GPS may not detect very slow or short-distance movement accurately at the exact moment the bus begins to move. To address this issue, this research uses a Hall Sensor Module LM393 Linear Hall Effect sensor with magnets attached to the bus propeller shaft. When the shaft rotates, the Hall sensor detects magnetic pulses and the system identifies the bus as moving [9].

Therefore, the research problem can be stated as follows:

How can a low-cost embedded school bus safety system combine footboard occupancy detection, non-GPS movement detection, AI-supported visual monitoring, and driver alerting to classify footboard risk as SAFE, WARNING, or CRITICAL in real time?

1.5 Research Objectives

1.5.1 Main Objective

The main objective of this individual research is to design and implement a movement-aware footboard safety monitoring module for school buses, supported by an experimental driver monitoring prototype, as part of the SISURAKSHA intelligent child safety platform.

1.5.2 Specific Objectives

- To detect footboard occupancy using E18-D80NK infrared proximity sensors placed around the bus entrance steps.
- To support footboard detection using AI-based camera input and YOLOv8 model output.
- To replace the inaccurate GPS speed method with Hall sensor-based bus movement detection.
- To design a SAFE, WARNING, and CRITICAL risk classification method based on footboard occupancy and bus movement state.
- To integrate ESP32, Hall sensor, IR sensors, LM2596 buck converter, DFPlayer, speaker, and push button hardware into the prototype.
- To test the prototype using controlled SAFE, WARNING, and CRITICAL scenarios.
- To document driver monitoring as an experimental prototype and identify its practical limitations.

2 METHODOLOGY

2.1 Requirements Gathering and Analysis

The requirements for the individual component were identified through team discussions, supervisor guidance, prototype testing, and practical observation of the school bus entrance area. The main functional requirement was to detect whether the footboard was occupied and to classify the risk using the movement state of the bus. Non-functional requirements included low cost, simple installation, understandable alerts, and safe testing.

The footboard module needed to work in both prototype and real bus environments. Therefore, the design had to support hardware testing on a bus prototype and practical evidence collection on a real bus. Since unsafe moving-bus testing with a person on the footboard is not ethical or safe, the system uses controlled testing and prototype simulation for the critical condition.

2.2 Proposed System Design

The proposed system has three main input categories: footboard occupancy input, movement input, and camera-based support input. Footboard occupancy is detected using E18-D80NK IR sensors. Movement is detected using the Hall Sensor Module LM393. Camera input is used by the Python/OpenCV runtime and AI model to support visual detection around the footboard.

The system output is a safety state. The SAFE state means that the footboard area is not occupied. The WARNING state means that the footboard is occupied while the bus is stopped. The CRITICAL state means that the footboard is occupied while the bus is moving. This logic is simple enough for practical use and understandable to the driver.

2.3 Hardware and Software Selection

The hardware selection was based on availability, cost, ease of integration, and suitability for school bus prototype testing. The ESP32 OV2640 Camera Bluetooth Wi-Fi Board was selected because it supports camera input and Wi-Fi communication. E18-D80NK sensors were selected for step occupancy detection. The Hall Sensor Module LM393 was selected because it can detect magnetic pulses created by magnets attached to a rotating shaft.

The software stack includes Arduino IDE for hardware programming, Python and OpenCV for the safety runtime, YOLOv8 for ML detection, Roboflow for dataset annotation, Google Colab for model training, Node.js for backend services, and React Native for frontend/mobile application development. Supabase and MongoDB are used within the broader SISURAKSHA platform for data management.

2.4 Movement Detection Method

The improved movement detection method uses the propeller shaft as the mechanical movement source. Four 10 mm x 5 mm magnets are fixed to the shaft. The Hall sensor is held close to the shaft using a metal mounting arm fixed to the chassis.

When the shaft rotates, the magnets pass near the sensing face and generate pulse signals.

The pulse signal is read by the ESP32 through GPIO 33. When pulses are detected within the defined time window, the bus is considered moving. When no pulse is detected, the bus is considered stopped. This is more suitable than GPS for the footboard module because it detects movement directly from shaft rotation.

2.5 Prototype Demonstration Method

Before the final bus shaft concept was documented, the movement detection method was demonstrated using a gear motor and magnets. The HW-618 resistance module was used only in this prototype demo to control the gear motor speed. This allowed different rotation speeds to be created so that the Hall sensor pulse detection could be observed.

This prototype setup is not the same as the real bus drivetrain. It was used only to demonstrate the principle of pulse-based movement detection. In the final real bus concept, the gear motor and HW-618 module are not required. The Hall sensor reads pulses from magnets attached to the actual bus propeller shaft.

2.6 Driver Monitoring Prototype Method

The driver monitoring prototype follows a camera-based pipeline. The system detects the driver face, extracts facial landmarks, calculates eye aspect ratio and mouth aspect ratio, estimates head pose, and uses YOLOv8 phone detection. The outputs are passed to a state machine to identify driver states.

However, the driver monitoring method is treated as experimental. The module produced false alerts and unstable detections under different lighting and camera positions. The methodology therefore includes driver monitoring as a prototype extension rather than a fully validated safety feature.

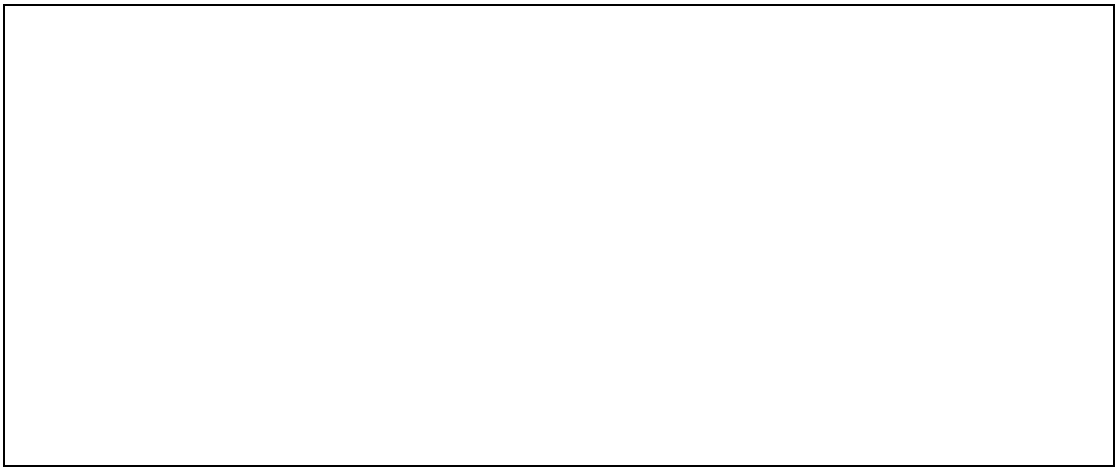


Figure 2.1: Overall architecture of the SISURAKSHA school bus safety system

Figure 2.2: Data flow of the footboard safety monitoring module

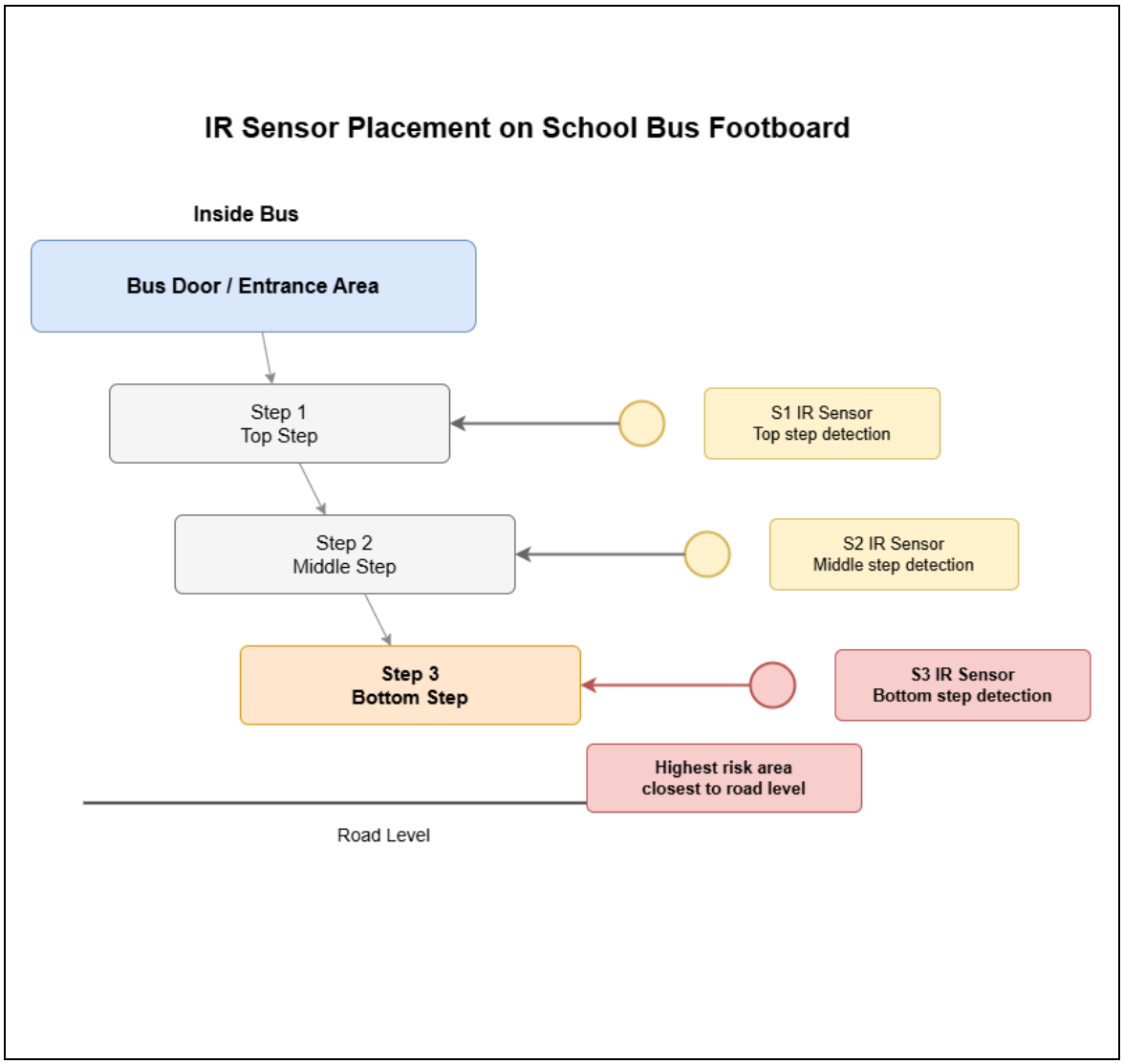


Figure 2.3: Proposed IR sensor placement on the bus footboard

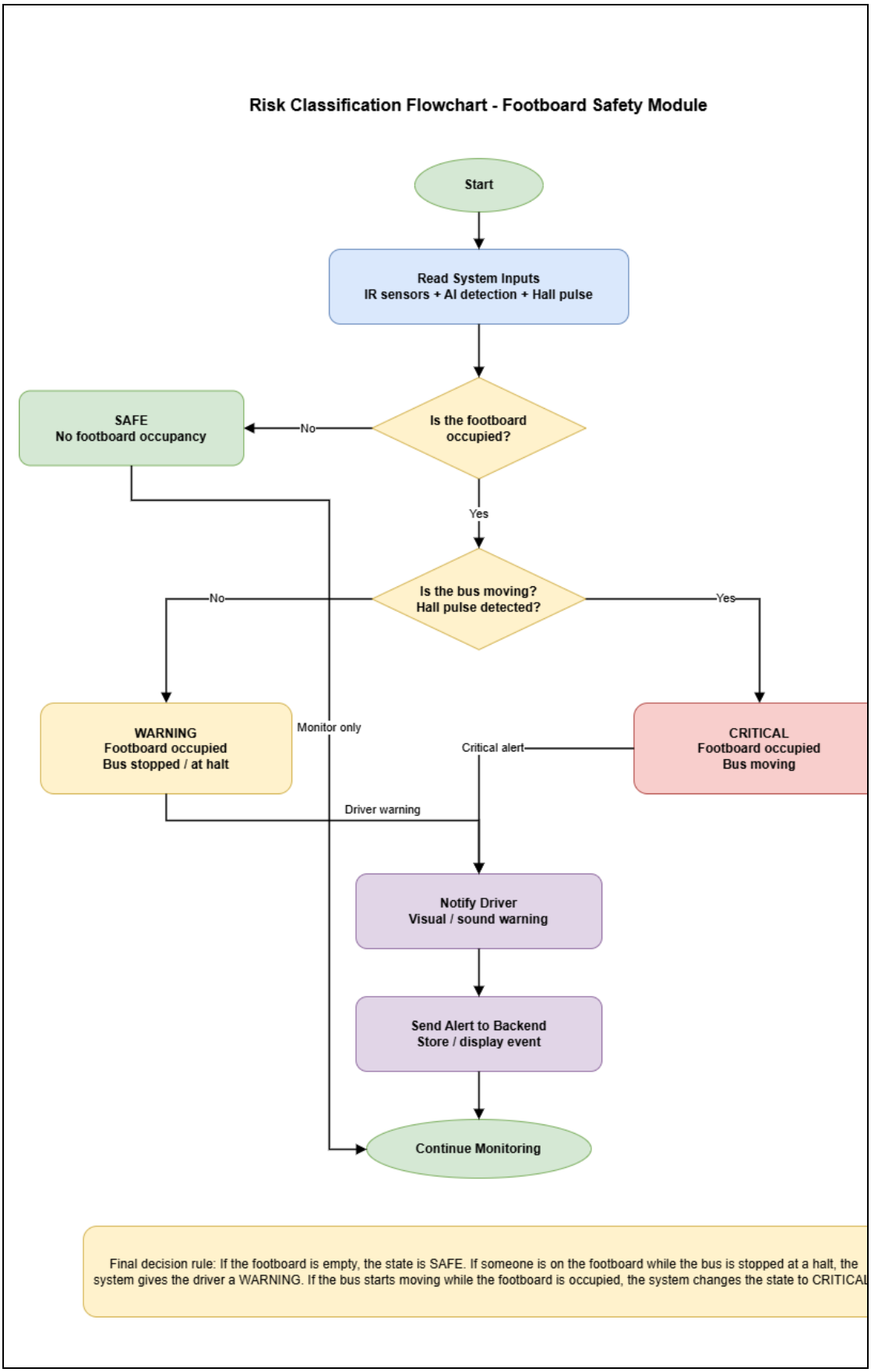


Figure 2.4: Risk classification flowchart of the footboard safety module

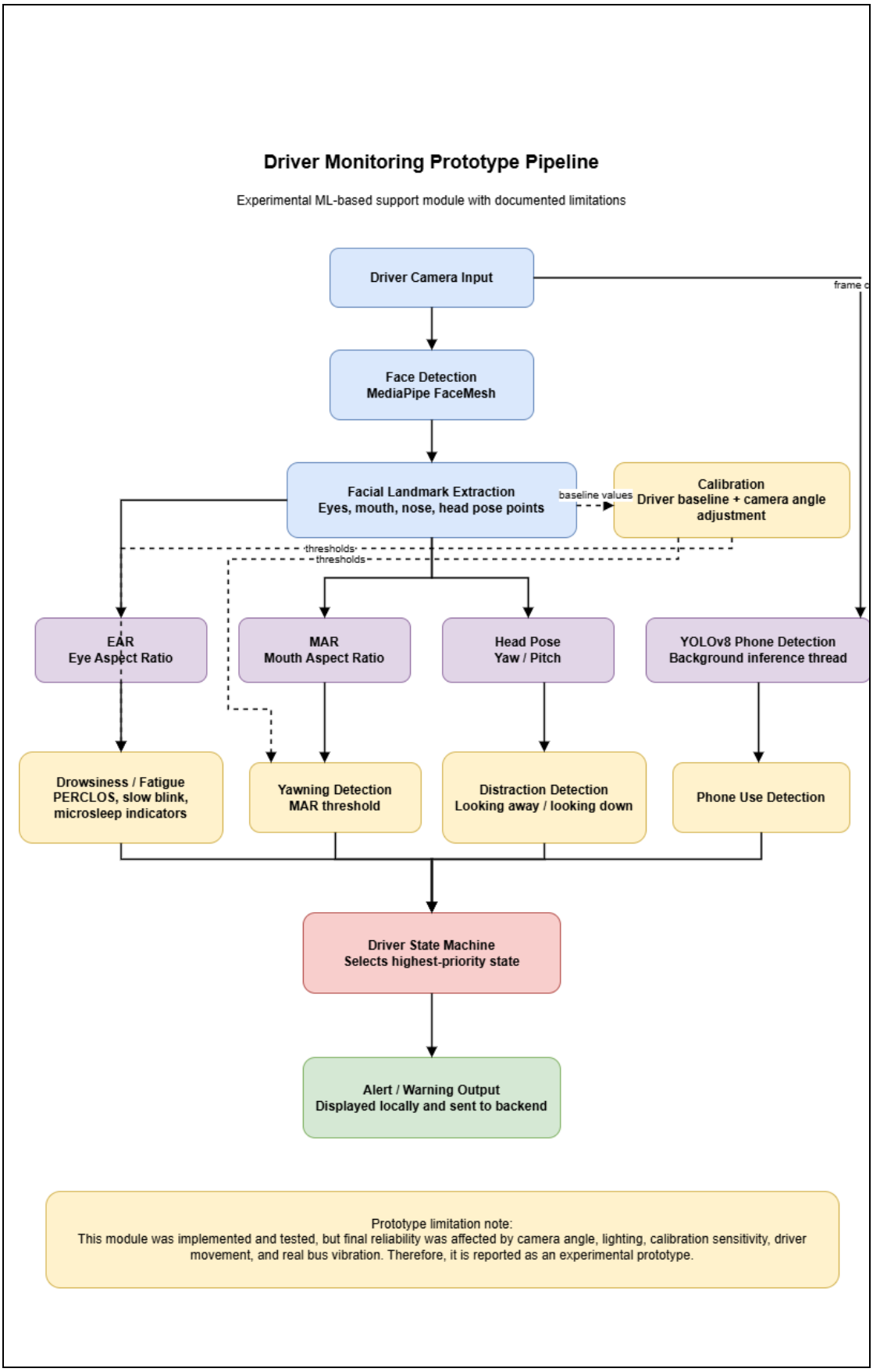


Figure 2.5: Experimental driver monitoring pipeline

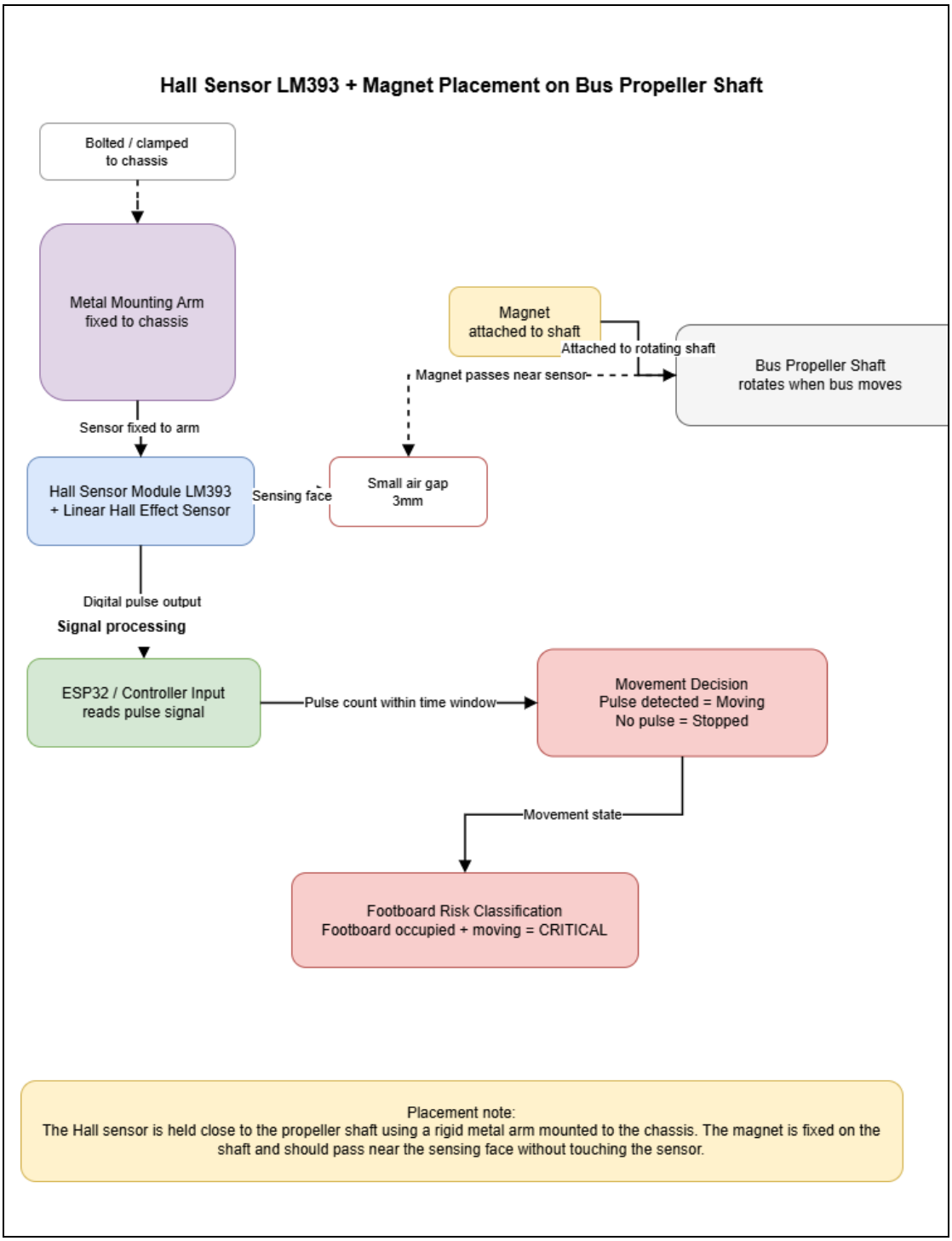


Figure 2.6: Hall sensor and magnet placement on the bus propeller shaft

Table 2.1: Hardware Components Used

Model / Type	Quantity	Purpose
ESP32 OV2640 Camera Bluetooth Wi-Fi Board	1	Main controller/camera board and Wi-Fi communication
ESP32-CAM Adapter Board MD0699	1	Programming and connection support
E18-D80NK Infrared Proximity Sensor	3	Footboard step occupancy detection
Hall Sensor Module LM393 Linear Hall Effect	1	Propeller shaft pulse-based movement detection
10 mm x 5 mm magnet	4	Magnetic pulse generation on rotating shaft
LM2596 Buck Converter	1	24V bus battery to 5V conversion
4 ohm 3W speaker	1	Audio warning output
DFPlayer module + 32GB memory card	1 each	Stored alert audio playback
DS-212 Red 16 mm Momentary Push Button BU0042	1	Manual input/control
HW-618 resistance module + gear motor	1 each	Prototype demo motor speed control
Metal arm fixed to chassis	1	Holds Hall sensor close to propeller shaft

Table 2.2: Software and Technologies Used

Technology	Use in this research
Arduino IDE	ESP32 firmware programming and sensor input testing
Python	Footboard safety runtime, sensor fusion, and AI integration
OpenCV	Real-time image/video processing and display overlay
YOLOv8	ML-based footboard and phone detection support
Roboflow	Dataset annotation/classification for ML model training
Google Colab	Training environment for ML models
Node.js	Backend API and safety alert communication

React Native	Frontend/mobile application interface
Supabase and MongoDB	Database support in the SISURAKSHA platform

Table 2.3: Footboard Safety Module Inputs

Input	Captured data	Processing method
IR Sensor S1	Top step occupancy	Digital input through GPIO 27
IR Sensor S2	Middle step occupancy	Digital input through GPIO 26
IR Sensor S3	Bottom step occupancy	Digital input through GPIO 25
Camera input	Footboard area image/video	YOLOv8/OpenCV support detection
Hall sensor	Propeller shaft pulse	GPIO 33 pulse reading
Push button	Manual input	GPIO 35 input

Table 2.4: Initial GPS Method vs Improved Hall Sensor Method

Area	Initial GPS method	Improved Hall sensor method
Movement principle	Uses GPS speed or location changes	Uses physical shaft rotation pulses
Low-speed response	Can be delayed or inaccurate	Detects movement from propeller shaft rotation
Signal dependency	Depends on GPS signal quality	Does not depend on GPS coverage
Final report use	Explained as initial approach only	Used as improved movement detection method
Main limitation	Not reliable for short movement	Needs correct magnet and sensor alignment

3 TESTING AND IMPLEMENTATION

3.1 Hardware Implementation

The hardware implementation was completed in two stages: prototype testing and real bus concept testing. The prototype was used to verify sensor connections, code behaviour, dashboard output, and movement simulation. The real bus environment was used to collect practical evidence of the entrance and footboard area and to plan the Hall sensor mounting location near the propeller shaft.

The 24V bus battery is not directly suitable for ESP32 and sensor modules. Therefore, the LM2596 buck converter is used to step down the 24V DC supply to 5V DC. This 5V supply rail powers the ESP32 and connected sensing devices. A common ground is maintained between the controller and sensors to keep the digital signals stable.

3.2 ESP32 Pin Mapping and Firmware

The ESP32 receives digital inputs from the footboard sensors and the Hall sensor. The top, middle, and bottom step IR sensors are connected to GPIO 27, GPIO 26, and GPIO 25 respectively. The Hall Sensor Module LM393 pulse output is connected to GPIO 33. The DFPlayer module is controlled through GPIO 32, while the DS-212 push button is connected to GPIO 35.

The firmware reads the sensors, builds the sensor state, exposes the state to a dashboard or serial monitor, and sends data to the Python safety runtime through Wi-Fi communication. During testing, the ESP32 dashboard or serial monitor was used to confirm that sensor states changed when the footboard area was occupied.

3.3 Hall Sensor and Movement Detection Implementation

The Hall sensor movement implementation was introduced after identifying the limitation of GPS for low-speed footboard safety. The Hall sensor is mounted close to the bus propeller shaft using a metal arm fixed to the chassis. Magnets attached to the shaft pass near the sensor when the shaft rotates. The resulting pulses are read by the ESP32.

In the prototype, the same idea was demonstrated using a gear motor. The HW-618 resistance module was used only for this demonstration to control the motor speed. When the motor rotated, the magnet passed the Hall sensor and generated pulses. This confirmed the basic working principle before applying it to the real bus shaft concept.

3.4 Footboard Safety Runtime

The Python runtime combines the hardware sensor states and AI-based camera support. It receives IR sensor data, reads movement state, displays the OpenCV monitoring window, and classifies the risk as SAFE, WARNING, or CRITICAL. The prototype screenshots captured during testing show these three states clearly.

The runtime also produces terminal logs that show alert communication or connection attempts. These logs were useful because database screenshots were not captured during the final documentation period. The backend terminal running screenshot and Python terminal alert log are therefore used as backend communication evidence.

3.5 Backend and Communication Implementation

The backend part of the system is implemented using Node.js. It receives safety-related messages and supports the broader SISURAKSHA platform. The frontend/mobile application is implemented using React Native, while Supabase and MongoDB support the data management needs of the platform. The final report includes the backend terminal screenshot as evidence that the backend server was active during testing.

Full database table screenshots and stored alert row screenshots were not captured due to time limitations. Therefore, the backend validation is presented using backend terminal output, Python runtime communication logs, ESP32 dashboard evidence, and Arduino webhook or endpoint code evidence.

3.6 Driver Monitoring Prototype Implementation

The driver monitoring module was implemented as a camera-based prototype. It includes face detection, facial landmark extraction, EAR, MAR, head pose, phone detection, calibration, and a state machine. It was intended to identify unsafe driver behaviours such as drowsiness, looking away, looking down, and phone use.

The module had several failures in practical testing. It was affected by lighting, camera position, calibration, and face movement. Therefore, the implementation is documented as a prototype and not as the main validated output of the research. The limitations are discussed in Chapter 4 and Chapter 6.

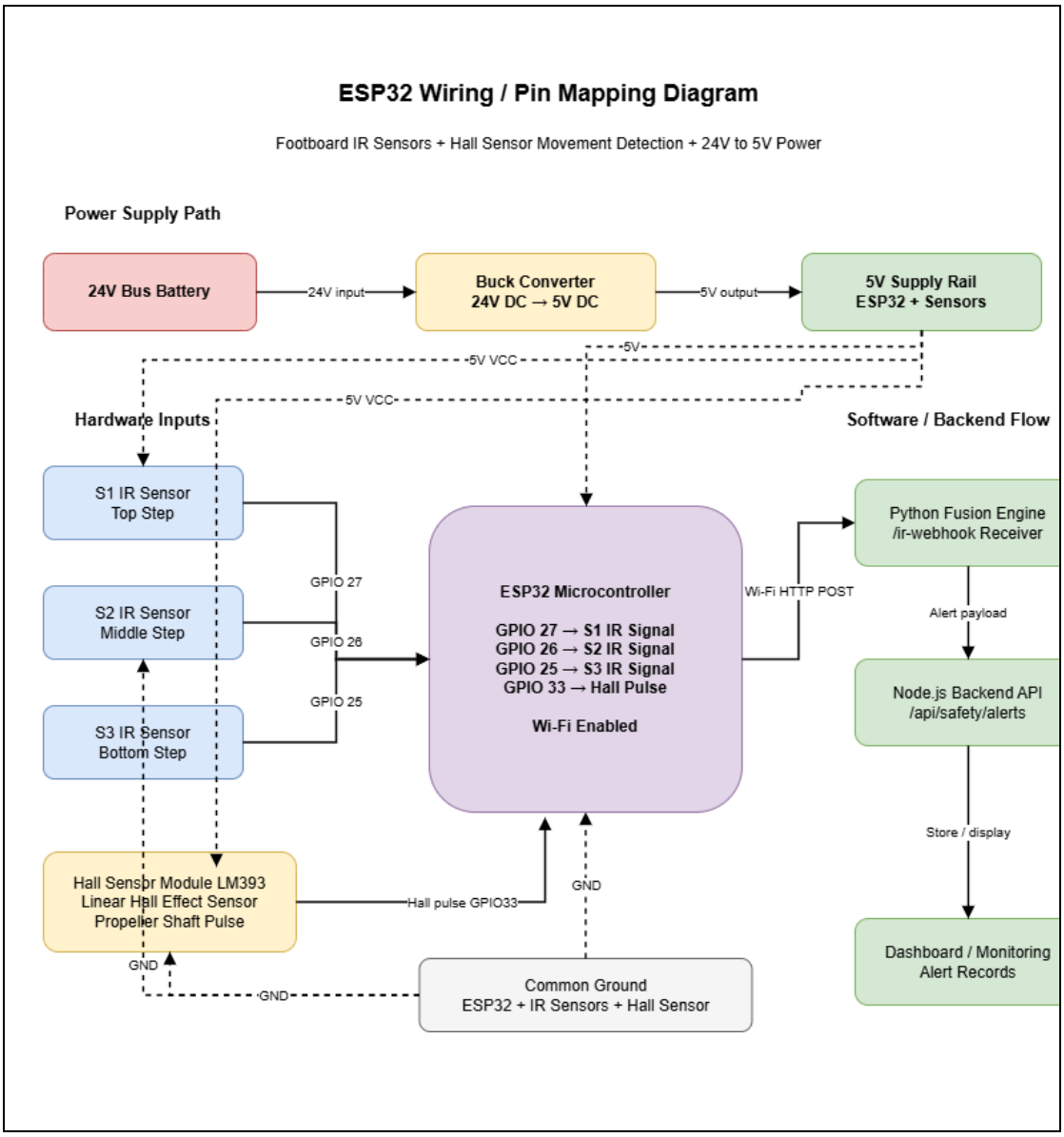


Figure 3.1: ESP32 wiring and pin mapping with 24V to 5V power conversion

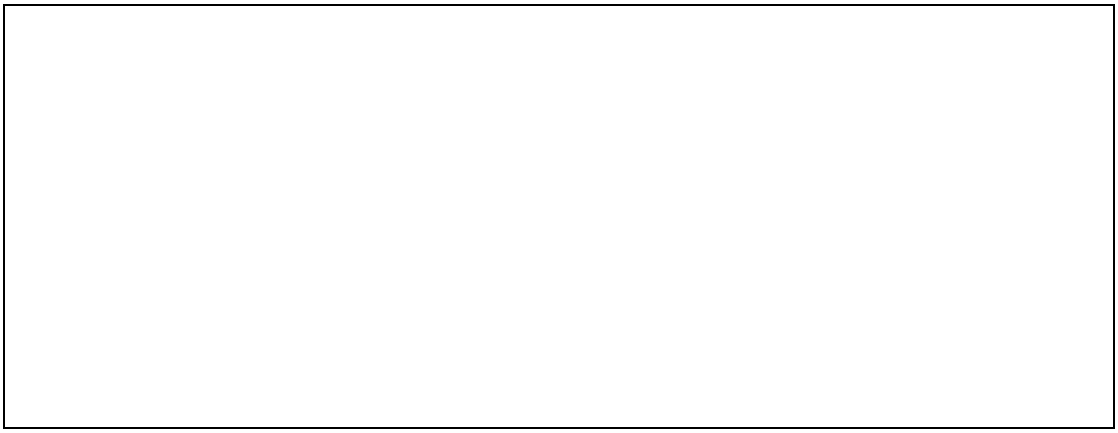


Figure 3.2: Hardware components used in the footboard safety module

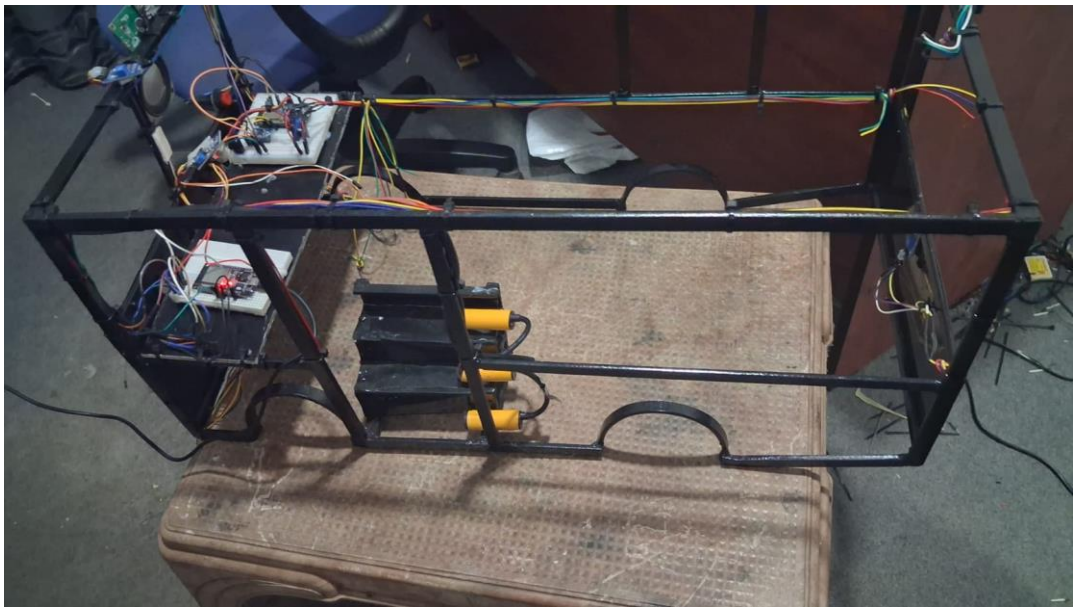


Figure 3.3: Complete prototype hardware connection

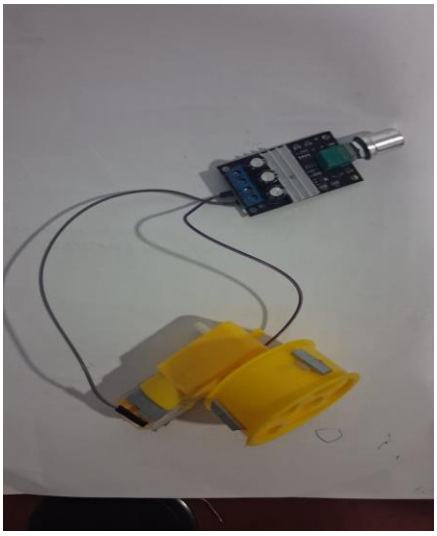


Figure 3.4: Hall sensor and gear motor prototype demonstration setup

Figure 3.5: Arduino IDE firmware configuration

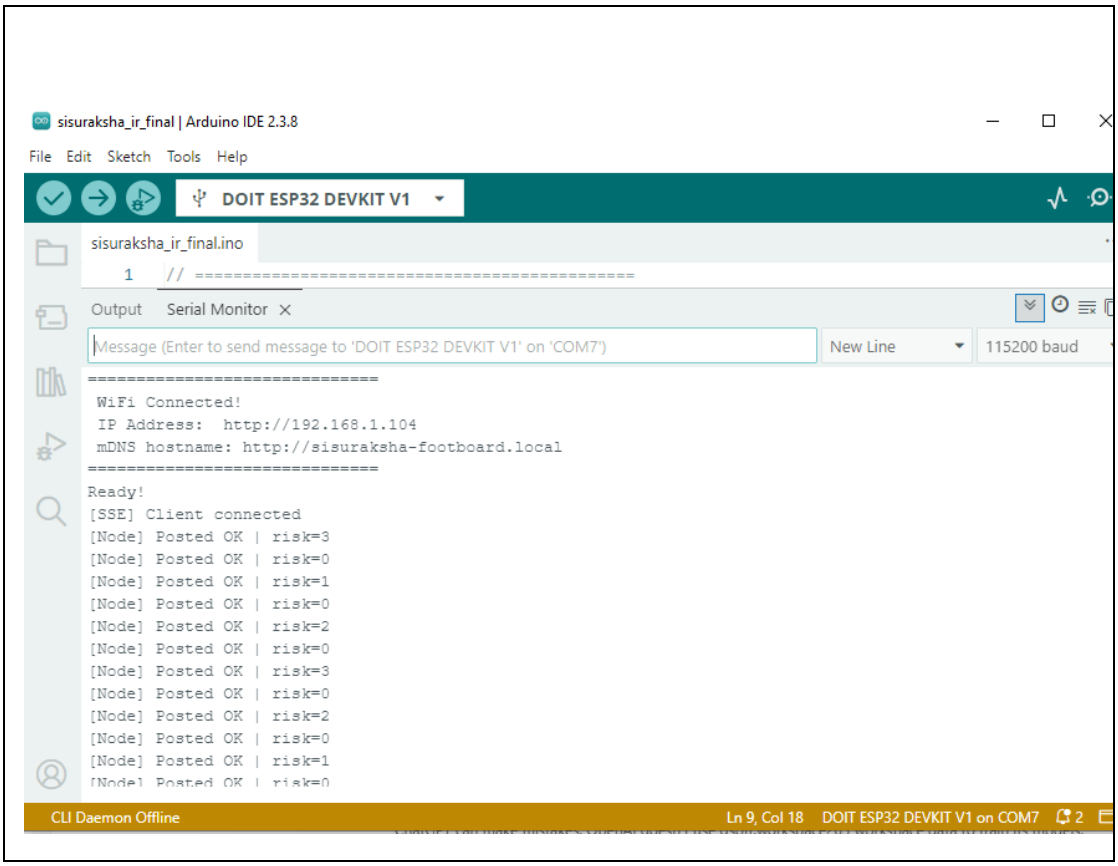


Figure 3.6: ESP32 dashboard or serial monitor output

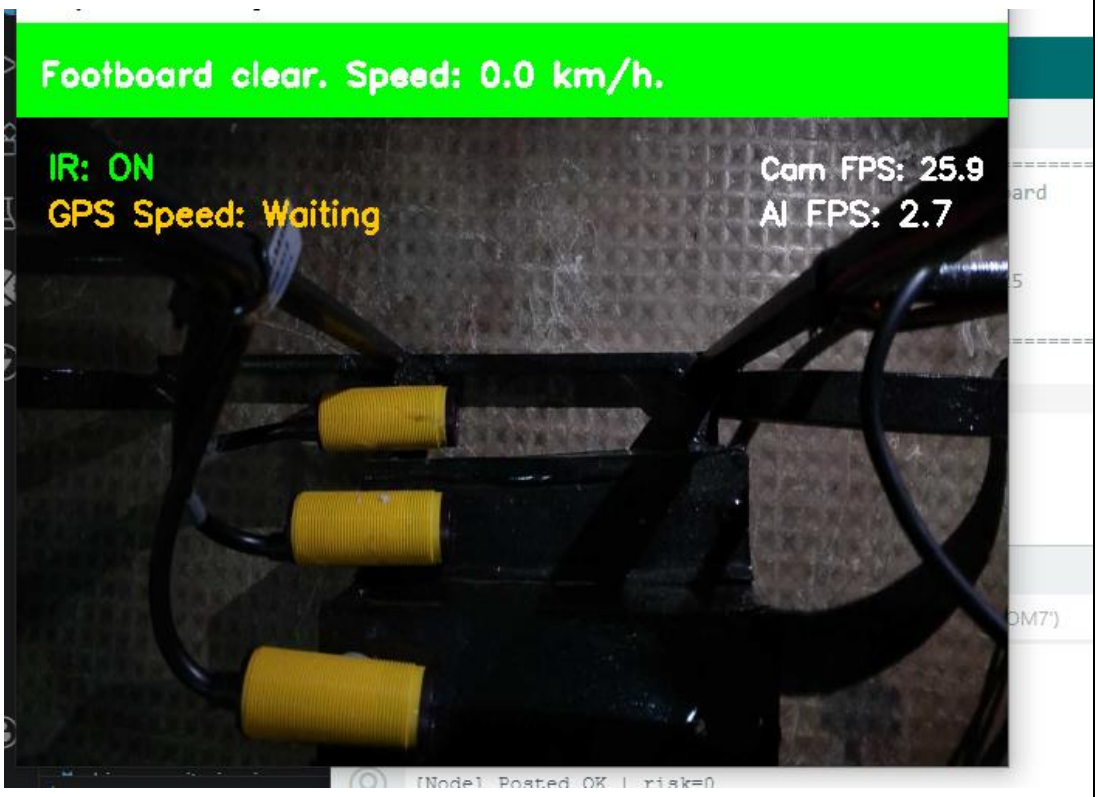


Figure 3.7: Python runtime window during footboard monitoring

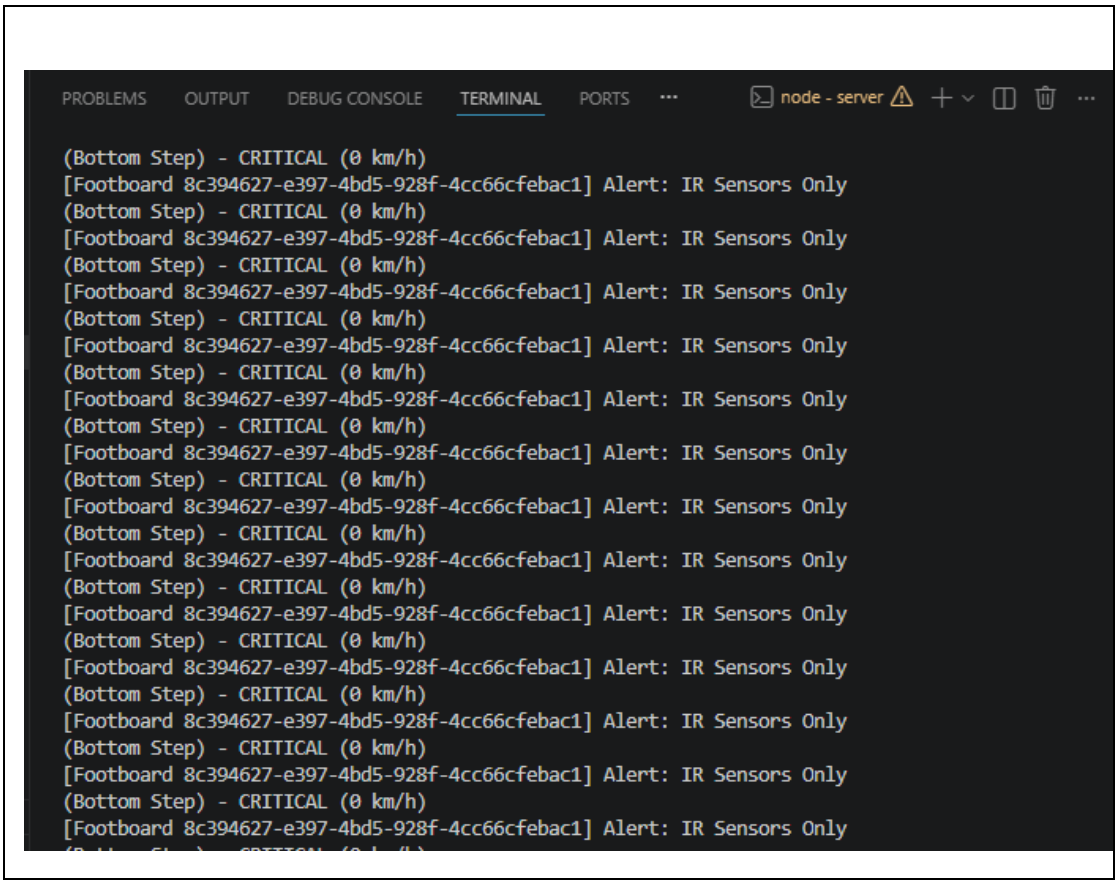


Figure 3.8: Backend server running during system testing

Table 3.1: ESP32 Pin Mapping and Connections

Device / module	Connection	Purpose
S1 IR Sensor	GPIO 27	Top-step footboard occupancy detection
S2 IR Sensor	GPIO 26	Middle-step footboard occupancy detection
S3 IR Sensor	GPIO 25	Bottom-step footboard occupancy detection
Hall Sensor Module LM393	GPIO 33	Propeller shaft pulse input
DFPlayer module	GPIO 32	Audio alert control
DS-212 push button	GPIO 35	Manual input/control
LM2596 buck converter output	5V rail	Power for ESP32 and connected devices
Common ground	GND rail	Shared signal reference

Table 3.2: Power Supply Design

Stage	Component	Input / output
Main power	24V bus battery	Provides 24V DC from the bus electrical system
Voltage conversion	LM2596 buck converter	Converts 24V DC to 5V DC
Controller power	ESP32 / ESP32-CAM	Uses 5V DC supply rail
Sensor power	IR sensors and Hall sensor	Uses 5V DC supply and common ground
Audio alert power	DFPlayer and speaker	Uses module output to drive warning sound

Table 3.3: IR Sensor Step Mapping

Sensor	Physical position	GPIO
S1	Top step near bus entrance	GPIO 27
S2	Middle step of footboard	GPIO 26
S3	Bottom step near road level	GPIO 25
Combined result	Any step occupied	S1 OR S2 OR S3

Table 3.4: Risk Classification Logic

Footboard condition	Movement state	System state
No occupancy	Stopped or moving	SAFE
Occupied	Stopped / at halt	WARNING to driver
Occupied	Moving / Hall pulse detected	CRITICAL alert
Uncertain sensor or AI reading	Any	WARNING or manual check

Table 3.5: Hall Sensor Movement Detection Logic

Condition	Sensor output	Interpretation
Propeller shaft not rotating	No pulse detected	Bus stopped
Shaft rotating slowly	Pulses detected	Bus moving

Shaft rotating continuously	Repeated pulses	Moving state confirmed
Magnet too far from sensor	Weak or missing pulse	Requires alignment adjustment
Prototype gear motor running	Pulse detected	Movement demo working

Table 3.6: Alert Output Design

State	Trigger	Driver output
SAFE	No footboard occupancy	No danger alert
WARNING	Footboard occupied while stopped	Warn driver before moving
CRITICAL	Footboard occupied while moving	Immediate critical alert and sound
Driver monitoring warning	Experimental driver issue detected	Prototype warning only

4 RESULTS AND DISCUSSION

4.1 Prototype Testing Results

The prototype testing confirmed the basic operation of the footboard safety logic. Screenshots were captured for SAFE, WARNING, and CRITICAL output states. These screenshots are important because they show that the classification logic worked in a controlled environment before real bus evidence was added.

The SAFE state was produced when no object or person was detected near the footboard. The WARNING state was produced when the footboard area was occupied while the movement state was stopped. The CRITICAL state was produced when the footboard was occupied while movement was triggered. This matches the final risk rule used in the system.

4.2 Hall Sensor Testing Discussion

The Hall sensor movement detection method was tested using the prototype motor setup and is designed for real bus shaft placement. The prototype used the gear motor and HW-618 resistance module to demonstrate different rotation conditions. The final real bus concept uses magnets on the propeller shaft instead of a demo motor.

This result shows that pulse-based movement detection is more appropriate for footboard safety than GPS. The system does not need to calculate exact road speed at this stage. It mainly needs to know whether the bus is moving or stopped. Actual speed calculation can be added later by measuring pulse frequency and calibrating it with shaft rotation and wheel/vehicle speed.

4.3 Real Bus Testing Evidence

Real bus testing evidence is used to show that the problem area exists in a practical bus environment. Photos of the bus entrance, footboard close-up, controlled occupancy condition, and system running near the bus should be inserted in this section. For safety reasons, risky moving-bus testing with a person standing on the footboard should not be performed.

The real bus evidence supports feasibility rather than unsafe scenario recreation. The critical condition is validated through controlled prototype testing and movement simulation. This is a safer and more ethical approach for undergraduate research.

4.4 Backend and Communication Evidence

The backend terminal screenshot shows that the Node.js backend server was active during testing. The Python terminal screenshot shows alert or connection attempts. The ESP32 dashboard or serial monitor screenshot shows hardware sensor state changes. These pieces of evidence together support the communication path even though database table screenshots were not captured.

The lack of database row screenshots is a limitation of final documentation. Future testing should capture database-level evidence showing stored alert rows with timestamp, alert type, movement state, confidence, and message.

4.5 Driver Monitoring Discussion

The driver monitoring prototype produced some working detections, but it was not reliable enough to be treated as a successful final module. Camera angle, face orientation, lighting changes, calibration quality, and vibration can all affect the output. Therefore, it is more suitable to report this module as an experimental prototype.

Including the driver monitoring limitation is important because the project title still contains driver monitoring. Instead of removing it, the report explains that it was implemented and tested, but that the main validated research output is the footboard safety module with movement-aware risk classification.

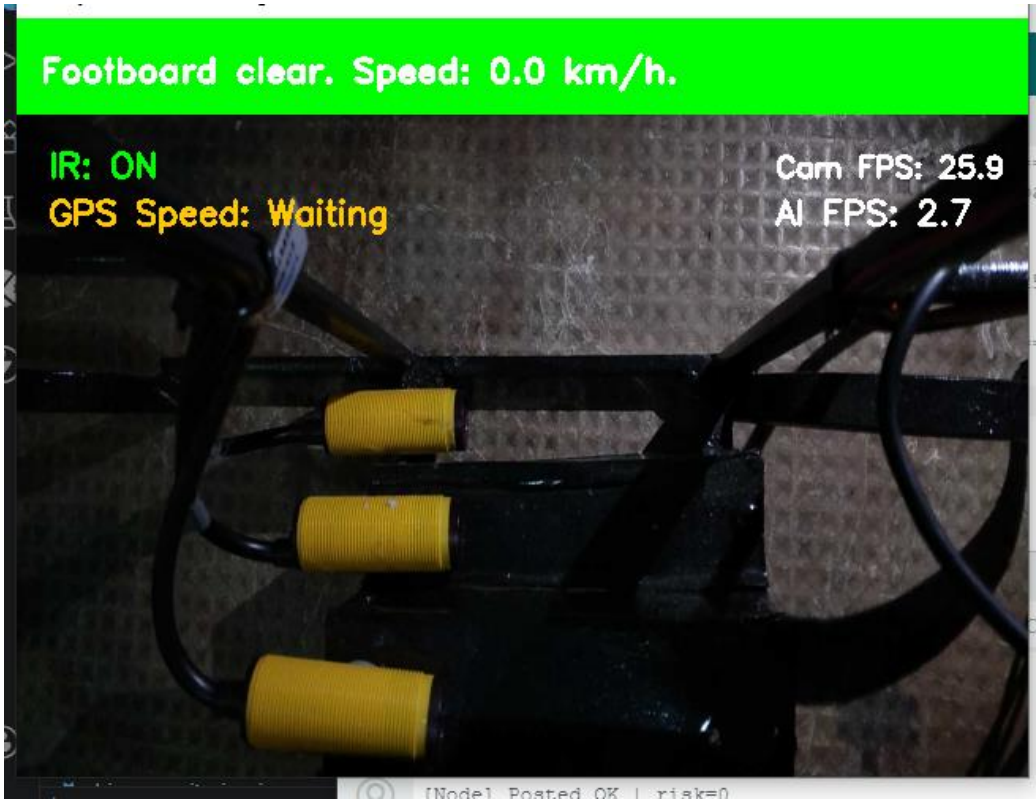


Figure 4.1: Prototype SAFE state output

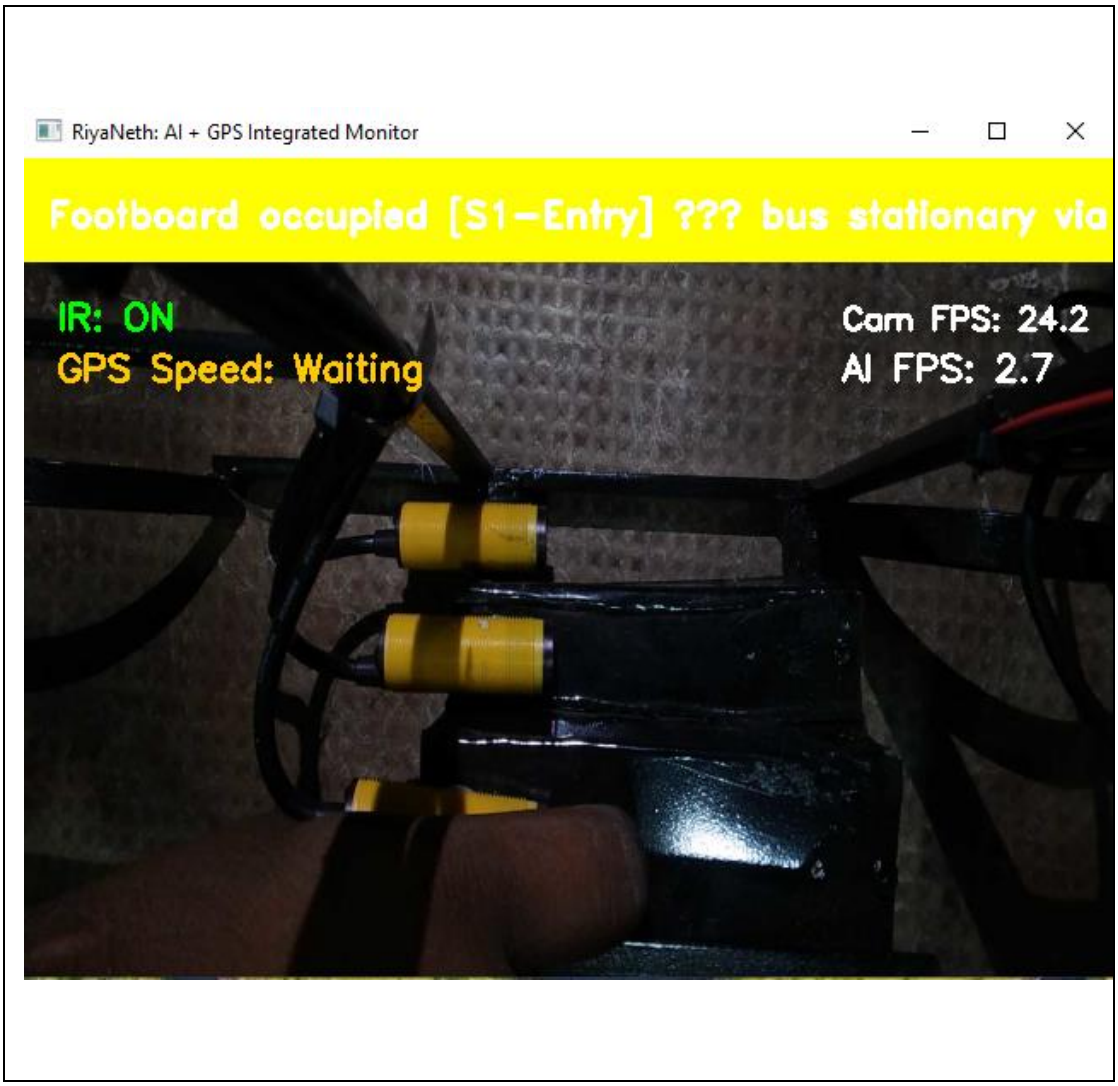


Figure 4.2: Prototype WARNING state output

Figure 4.3: Prototype CRITICAL state output

Figure 4.4: ML footboard dataset collection process

Figure 4.5: Roboflow annotation or dataset labelling screenshot

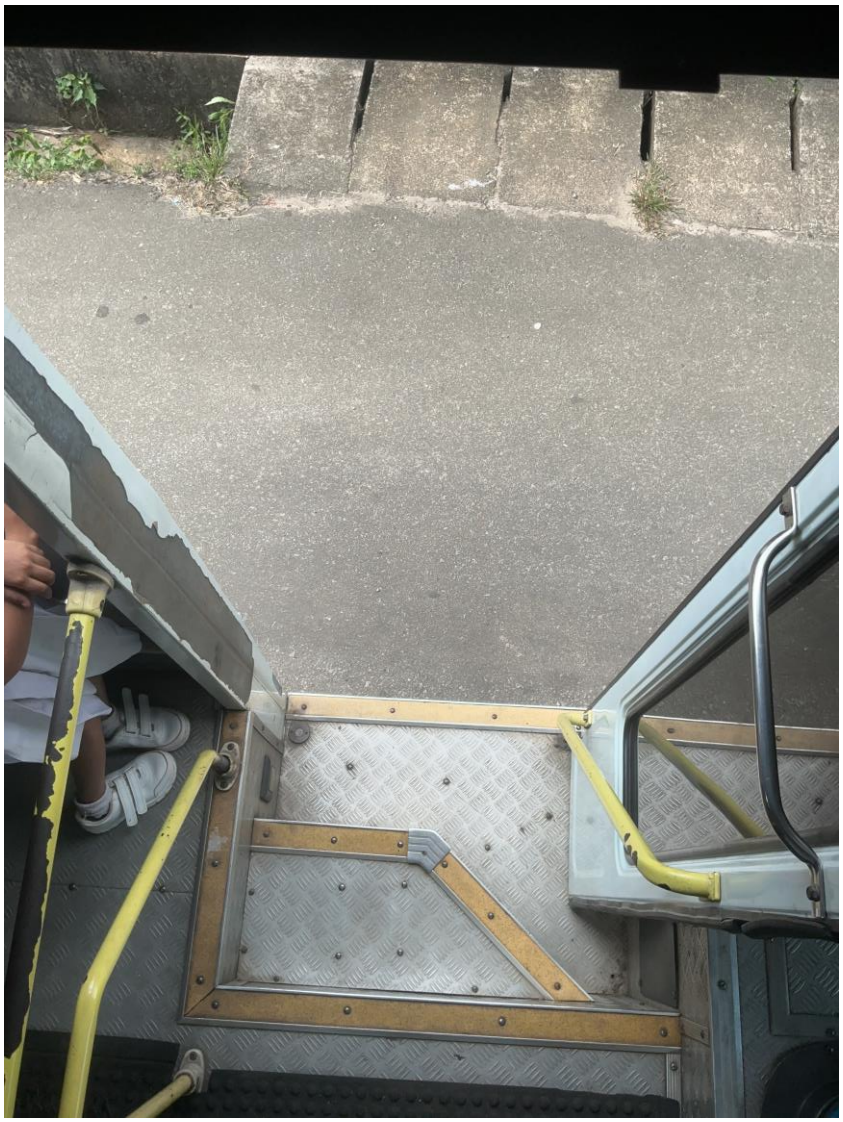


Figure 4.6: Real bus entrance and footboard area

Figure 4.7: Controlled footboard occupancy test on real bus

Figure 4.8: System running during real bus testing

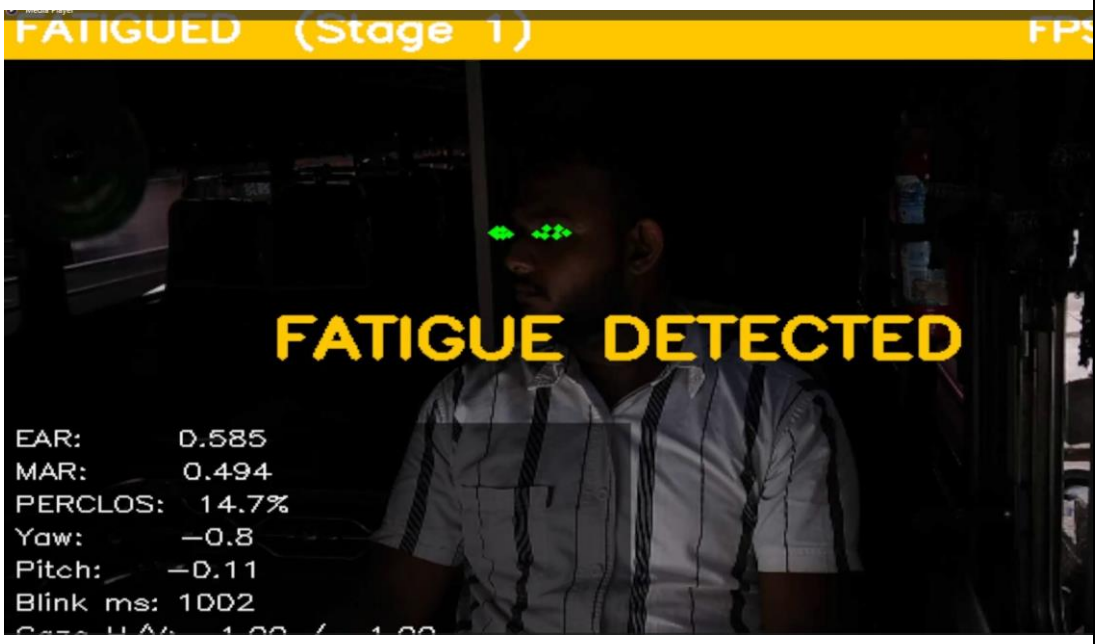


Figure 4.9: Driver monitoring prototype window

Figure 4.10: Driver monitoring false alert or failed detection example

Table 4.1: Prototype Testing Results

Test ID	Test scenario	Observed result
PT-01	No object/person near footboard	SAFE state captured
PT-02	Object/person detected while stopped	WARNING state captured
PT-03	Footboard occupied with movement condition	CRITICAL state captured
PT-04	ESP32 sensor state change	Dashboard/serial monitor evidence captured
PT-05	Python alert communication	Terminal log evidence captured

Table 4.2: Hall Sensor Movement Detection Test Cases

Test ID	Condition	Expected output
HS-01	Shaft or demo motor not rotating	No pulse; bus stopped
HS-02	Magnet passes Hall sensor slowly	Pulse detected; moving
HS-03	Continuous rotation	Repeated pulses; moving
HS-04	Footboard occupied + no pulse	WARNING
HS-05	Footboard occupied + pulse detected	CRITICAL
HS-06	Magnet too far from sensor	Weak or missing pulse; alignment required

Table 4.3: Real Bus Testing Evidence

Evidence ID	Evidence captured	Purpose
RB-01	Real bus entrance / footboard empty	Shows real testing environment
RB-02	Footboard close-up	Shows risk area
RB-03	Object/person near footboard while stopped	Shows controlled unsafe condition
RB-04	System running near the real bus	Shows practical test setup
RB-05	Hall sensor or shaft area if available	Shows movement detection concept placement

Table 4.4: Backend and Communication Evidence

Evidence	Purpose	Status
Backend terminal running	Shows Node.js backend was active	Captured
Python terminal alert/connection log	Shows communication attempt	Captured
ESP32 dashboard/serial monitor	Shows sensor state change	Captured
Arduino webhook/endpoint code	Shows communication configuration	Captured
Database alert row screenshot	Shows stored alert record	Not captured

Table 4.5: Driver Monitoring Prototype Results and Limitations

Feature	Observed issue	Final decision
Face detection	Can work but depends on camera position	Prototype
Looking away/down	Sensitive to head angle and calibration	Prototype
Phone detection	Depends on camera view and model confidence	Prototype
Drowsiness detection	False alerts under lighting/camera changes	Not main validated feature
Calibration	Sensitive to driver movement and environment	Limitation

5 SUMMARY OF INDIVIDUAL CONTRIBUTION

My individual contribution focused mainly on the footboard safety monitoring module and the experimental driver monitoring prototype. The most important part of the work was designing the footboard risk classification logic and improving movement detection after identifying the limitation of GPS.

I worked on the hardware integration of the IR sensors, Hall sensor, ESP32, LM2596 buck converter, DFPlayer module, speaker, push button, and prototype motor demonstration. I also prepared the technical explanation for the Hall sensor method, where magnets attached to the bus propeller shaft generate pulses when the bus moves.

I collected and documented evidence from prototype testing, including SAFE, WARNING, and CRITICAL outputs. I also collected ML data collection evidence for footboard detection and backend/runtime screenshots showing the system running. Driver monitoring was implemented as a prototype, but the final report honestly explains that it was not reliable enough to become the main validated contribution.

Table 5.1: Individual Contribution Summary

Area	Contribution
Footboard safety concept	Designed movement-aware SAFE, WARNING, and CRITICAL logic
IR-based detection	Integrated E18-D80NK IR sensors for step occupancy detection
Movement detection	Improved GPS idea using Hall sensor pulse-based detection
Power design	Used 24V bus battery stepped down to 5V with LM2596 buck converter
Prototype testing	Captured SAFE, WARNING, and CRITICAL testing evidence
ML evidence	Collected footboard dataset and annotation/training evidence
Driver monitoring	Implemented and documented as experimental prototype
Documentation	Prepared diagrams, tables, code evidence, and report structure

6 CONCLUSION AND FUTURE WORK

This research presented the individual footboard safety and experimental driver monitoring component developed under the SISURAKSHA intelligent child safety platform for school buses. The main validated outcome is a footboard safety module that detects occupancy and classifies risk based on whether the bus is moving or stopped.

The improved movement detection method is an important part of the contribution. GPS was considered at first, but it was not suitable for low-speed and short-distance movement decisions. The Hall sensor and propeller shaft magnet method provides a more practical movement signal for this safety problem. The final logic is understandable and directly connected to the real risk: occupied footboard while stopped gives a warning, while occupied footboard while moving gives a critical alert.

The driver monitoring module was implemented but not fully successful. It is included as an experimental prototype because it suffered from false alerts and sensitivity to camera and lighting conditions. Future work should improve the driver monitoring dataset, camera placement, calibration method, and testing under real bus vibration conditions.

Table 6.1: System Limitations and Future Improvements

Limitation	Effect	Future improvement
GPS was not accurate enough	Not suitable for low-speed footboard decisions	Use Hall sensor pulse detection
Hall sensor requires alignment	Weak pulses if magnet is too far	Use fixed bracket and calibration
Database screenshots not captured	Backend evidence limited to logs	Capture stored alert rows later
Driver monitoring false alerts	Not reliable for final deployment	Improve dataset and camera setup
No unsafe moving-bus human test	Critical scenario validated by prototype	Use controlled simulation and safe testing

6.1 Future Enhancements

- Calculate actual bus speed by measuring Hall sensor pulse frequency and calibrating shaft rotation against vehicle speed.
- Improve the Hall sensor mounting bracket to reduce vibration effects and missed pulses.
- Add database-level verification screenshots and full API testing evidence.

- Improve driver monitoring reliability using better camera placement, larger datasets, and real bus vibration testing.
- Develop a mobile dashboard that shows SAFE, WARNING, and CRITICAL states clearly to the driver and school admin.

REFERENCES

[1] U.S. Department of Transportation, National Highway Traffic Safety Administration, National Center for Statistics and Analysis, Traffic Safety Facts 2013–2022 Data: School-Transportation-Related Traffic Crashes, Report DOT HS 813 600, Aug. 2024.

[2] National Highway Traffic Safety Administration, “Planning safer school bus stops and routes,” 2022.

[3] J. F. Dols Ruiz, L. Armesto, V. Girbés, and L. Arnal, “SAFEBUS: A new active safety system for pedestrian detection in public passenger buses,” in Project Management and Engineering Research. Cham, Switzerland: Springer, 2017, pp. 231–244, doi: 10.1007/978-3-319-51859-6_16.

[4] B. P. Y. Loo, Z. Fan, T. Lian, and F. Zhang, “Using computer vision and machine learning to identify bus safety risk factors,” Accident Analysis & Prevention, vol. 185, p. 107017, Jun. 2023, doi: 10.1016/j.aap.2023.107017.

[5] M. W. Raad, M. Deriche, and T. Sheltami, “An IoT-Based School Bus and Vehicle Tracking System Using RFID Technology and Mobile Data Networks,” Arabian Journal for Science and Engineering, vol. 46, pp. 3087–3097, 2021, doi: 10.1007/s13369-020-05111-3.

[6] K. D. Anh, M. V. Tung, V. D. Hanh, L. M. Huong, N. T. Minh, and P. D. Hung, “A Collaborative IoT School Bus Monitoring System,” in Cooperative Design, Visualization, and Engineering, LNCS, vol. 15158. Cham, Switzerland: Springer, 2024, pp. 123–132, doi: 10.1007/978-3-031-71315-6_13.

[7] D. Hercog, T. Lerher, M. Truntić, and O. Težak, “Design and Implementation of ESP32-Based IoT Devices,” Sensors, vol. 23, no. 15, p. 6739, 2023, doi: 10.3390/s23156739.

[8] N. Argirusis, A. Achilleos, N. Alizadeh, C. Argirusis, and G. Sourkouni, “IR Sensors, Related Materials, and Applications,” Sensors, vol. 25, no. 3, p. 673, 2025, doi: 10.3390/s25030673.

[9] M. Akrami, E. Jamshidpour, L. Baghli, and V. Frick, “Application of Low-Resolution Hall Position Sensor in Control and Position Estimation of PMSM—A Review,” Energies, vol. 17, no. 17, p. 4216, 2024, doi: 10.3390/en17174216.

[10] J. Zhong, H. Qian, H. Wang, W. Wang, and Y. Zhou, “Improved real-time object detection method based on YOLOv8: a refined approach,” Journal of Real-Time Image Processing, vol. 22, art. no. 4, 2025, doi: 10.1007/s11554-024-01585-8.

[11] W. Xu, Y. Zhao, X. Du, H. Ji, and L. Xing, “A Study on Bus Passenger Boarding and Alighting Detection and Recognition Based on Video Images and YOLO Algorithm,” Sensors, vol. 26, no. 5, p. 1418, 2026, doi: 10.3390/s26051418.

[12] H. Wang, J. Liu, H. Dong, and Z. Shao, “A Survey of the Multi-Sensor Fusion Object Detection Task in Autonomous Driving,” Sensors, vol. 25, no. 9, p. 2794, 2025, doi: 10.3390/s25092794.

[13] T. Soukupová and J. Čech, “Real-Time Eye Blink Detection using Facial Landmarks,” in Proc. 21st Computer Vision Winter Workshop, 2016.

[14] “PERCLOS-based technologies for detecting drowsiness: current evidence and future directions,” Sleep Advances, vol. 4, no. 1, zpad006, 2023, doi: 10.1093/sleepadvances/zpad006.

[15] E. Michelaraki, C. Katrakazas, S. Kaiser, T. Brijs, and G. Yannis, “Real-time monitoring of driver distraction: State-of-the-art and future insights,” Accident Analysis & Prevention, vol. 192, p. 107241, Nov. 2023, doi: 10.1016/j.aap.2023.107241.

[16] D. Yang et al., “AIDE: A Vision-Driven Multi-View, Multi-Modal, Multi-Tasking Dataset for Assistive Driving Perception,” in Proc. IEEE/CVF International Conference on Computer Vision, 2023.

[17] A. K. Anggara, D. R. Suchendra, P. D. Ibnugraha, S. Siregar, and A. A. G. Agung, “Mobile Phone Usage Warning System for Driver Focus Monitoring,” International Journal of Transport Development and Integration, vol. 9, no. 2, pp. 325–335, 2025.

APPENDICES

Appendix A: Arduino Firmware Evidence

The following appendix space is reserved for the full Arduino firmware or selected code screenshots. The pin mapping should show GPIO 27, GPIO 26, GPIO 25, GPIO 33, GPIO 32, and GPIO 35.

```
#include <WiFi.h>
#include <ESPmDNS.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <HTTPClient.h>

// -- WiFi Credentials --------------------------
const char* WIFI_SSID     = "SLT_FIBER_PYt5z";
const char* WIFI_PASSWORD = "07726Padumapks";



// -- IR Sensor Pins ----------------------------
#define IR_STEP1  27    // top step    — lowest risk
#define IR_STEP2  26    // middle step — medium risk
#define IR_STEP3  25    // bottom step — highest risk

// Enable only the sensors that are physically connected.
// For a one-sensor test on D25, keep Step 1/2 disabled so their pins cannot float.
#define ENABLE_STEP1  false
#define ENABLE_STEP2  true
#define ENABLE_STEP3  false

// Your D25 sensor is behaving as: HIGH = object detected, LOW = clear.
// If a sensor later behaves the opposite way, change both lines to LOW /
INPUT_PULLUP.
#define DETECTED       HIGH
#define IR_INPUT_MODE  INPUT_PULLDOWN

// -- Server + SSE ------------------------------
AsyncWebServer   server(80);
AsyncEventSource events("/events");

// -- Node.js Backend ---------------------------
// Change this to your Node.js server IP
// Use mDNS hostname if available
//const char* NODE_SERVER = "http://192.168.1.102:5000/api/safety/alerts";

// Set this to the laptop IP running footboard safety/riyabeth_pro_safety.py
const char* FOOTBOARD_WEBHOOK_URL = "http://192.168.1.102:5001/ir-webhook";

// -- Sensor state (updated every 50ms) ---------
volatile bool step1 = false;
volatile bool step2 = false;
volatile bool step3 = false;

// Previous state — only POST when changed
bool prevStep1 = false;
bool prevStep2 = false;
bool prevStep3 = false;

// ================================================
// HTML Dashboard
// ================================================
const char HTML_PAGE[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>SISURAKSHA</title>
```

```
  <meta name='viewport' content='width=device-width,initial-scale=1'>
  <style>
    *{box-sizing:border-box}
    body{font-family:Arial,sans-
serif;background:#1a1a2e;color:#eee;margin:0;padding:15px}
    h1{color:#e94560;text-align:center;margin-bottom:5px}
    h2{text-align:center;color:#aaa;font-size:1em;font-weight:normal;margin:0 0 12px}
    #bar{text-align:center;padding:8px;border-radius:8px;margin:8px 0;font-
weight:bold;font-size:1.1em}
    .live{background:#1a3a1a;color:#44ff44}
    .dead{background:#3a1a1a;color:#ff6666}

    /* Step cards */
    .steps{display:flex;gap:10px;margin:15px 0}
    .step{flex:1;border-radius:12px;padding:20px 10px;text-align:center;transition:all
0.15s ease}
    .step-label{font-size:0.85em;color:#aaa;margin-bottom:8px}
    .step-num{font-size:1.8em;font-weight:bold;margin-bottom:6px}
    .step-status{font-size:0.9em;font-weight:bold}

    .clear{background:#16213e;border:2px solid #0f3460}
    .occupied{background:#3a0a0a;border:2px solid #ff4444;animation:pulse 0.5s
infinite alternate}

    @keyframes pulse{from{box-shadow:0 0 5px #ff4444}to{box-shadow:0 0 20px #ff4444}}

    /* Risk banner */
    #risk{text-align:center;padding:12px;border-radius:10px;margin:15px 0;font-
size:1.3em;font-weight:bold;transition:all 0.2s}
    .risk0{background:#1a3a1a;color:#44ff44}
    .risk1{background:#3a2a00;color:#ffaa00}
    .risk2{background:#3a1500;color:#ff6600}
    .risk3{background:#3a0000;color:#ff2222;animation:pulse 0.4s infinite alternate}

    /* Info */
    #info{text-align:center;color:#00b4d8;font-size:0.82em;margin:6px 0}
    #meta{text-align:center;color:#444;font-size:0.75em;margin-top:15px}

    /* Log */
    .card{background:#16213e;border-radius:10px;padding:12px;margin:12px 0}
    h3{color:#fff;background:#0f3460;padding:6px 10px;border-radius:6px;margin:0 0
8px}
    #log{font-family:monospace;font-size:0.82em;color:#aaa;height:120px;overflow-
y:auto}
    .log-entry{padding:2px 0;border-bottom:1px solid #0f3460}
    .log-alert{color:#ff6666}
    .log-clear{color:#44ff44}
  </style>
</head>
<body>
  <h1>SISURAKSHA</h1>
  <h2>Footboard Safety Monitor</h2>

  <div id='bar' class='dead'>Connecting...</div>

  <!-- Step Cards -->
  <div class='steps'>
    <div class='step clear' id='card1'>
      <div class='step-label'>TOP STEP</div>
      <div class='step-num'>1</div>
      <div class='step-status' id='s1txt'>Clear</div>
    </div>
    <div class='step clear' id='card2'>
      <div class='step-label'>MID STEP</div>
      <div class='step-num'>2</div>
      <div class='step-status' id='s2txt'>Clear</div>
    </div>
    <div class='step clear' id='card3'>
      <div class='step-label'>BOT STEP</div>
      <div class='step-num'>3</div>
      <div class='step-status' id='s3txt'>Clear</div>
```

```
</div>
```

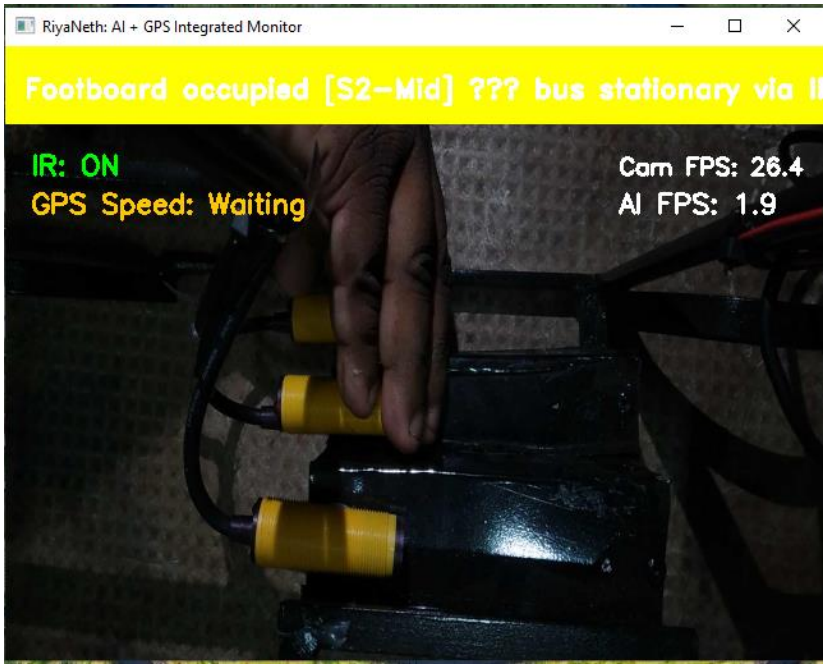
Appendix B: Python Footboard Safety Code Evidence

riyabeth_pro_safety.py 3, M

C: > Users > MB > Documents > GitHub > Sisuraksha > footboard safety > riyabeth_pro_safety.py > _attach_repo_root

```python
import cv2
import threading
import time
import requests
import numpy as np
from dataclasses import dataclass, field
from datetime import datetime
from ultralytics import YOLO

import argparse
import sys
from pathlib import Path
from flask import Flask, request, jsonify


def _attach_repo_root():
    this_file = Path(__file__).resolve()
    for parent in this_file.parents:
        if (parent / "shared_network_config.py").exists():
            root_path = str(parent)
            if root_path not in sys.path:
                sys.path.append(root_path)
            return


_attach_repo_root()

from shared_network_config import (
    build_esp32cam_stream_url,
    build_phone_sensor_url,
    build_phone_video_url,
    load_network_config,
```

Appendix Figure B.1: Python code/runtime evidence 1



Appendix Figure B.2: Python code/runtime evidence 2

```
import { pool } from '../config/postgres.js';
import { spawn } from 'child_process';
import { existsSync } from 'fs';
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const driverHeartbeats = new Map();
const driverSystemEnabled = new Map();
const modelProcesses = new Map();

const FOOTBOARD_ROOT = path.join(__dirname, '../../footboard safety');

const MODEL_CONFIG = {
  pythonCandidates: [
    path.join(FOOTBOARD_ROOT, '.venv/Scripts/python.exe'),
    path.join(FOOTBOARD_ROOT, 'venv/Scripts/python.exe'),
    path.join(__dirname, '../../.venv/Scripts/python.exe'),
  ],
  scriptPath: path.join(FOOTBOARD_ROOT, 'riyabeth_pro_safety.py'),
  cwd: FOOTBOARD_ROOT,
  startupGraceMs: 800,
};

function resolvePythonPath() {
  const overridePath = process.env.FOOTBOARD_PYTHON;
  if (overridePath && existsSync(overridePath)) return overridePath;

  for (const candidate of MODEL_CONFIG.pythonCandidates) {
    if (existsSync(candidate)) return candidate;
  }

  return null;
}

function parseBoolean(value) {
  if (typeof value === 'boolean') return value;
  if (typeof value === 'number') return value !== 0;
  if (typeof value === 'string') {
    return ['1', 'true', 'yes', 'on', 'occupied',
'blocked'].includes(value.trim().toLowerCase());
  }
  return false;
}
```

```
function normalizeHardwareAlert(body) {
  const hasStepPayload = ['s1', 's2', 's3'].some(key =>
Object.prototype.hasOwnProperty.call(body, key));
  if (!hasStepPayload) return null;

  const s1 = parseBoolean(body.s1);
  const s2 = parseBoolean(body.s2);
  const s3 = parseBoolean(body.s3);
  const risk = Number(body.risk ?? (s3 ? 3 : s2 ? 2 : s1 ? 1 : 0));

  let stepLabel = 'Clear';
  if (s3) stepLabel = 'Bottom Step';
  else if (s2) stepLabel = 'Mid Step';
  else if (s1) stepLabel = 'Top Step';

  const status = risk >= 3 ? 'CRITICAL' : risk >= 1 ? 'WARNING' : 'SAFE';
  const alertType = risk > 0 ? `IR Sensors Only (${stepLabel})` : 'IR Sensors
Clear';
  const message = risk > 0
    ? `Footboard ${stepLabel.toLowerCase()} occupied from ESP32 IR sensor`
    : 'Footboard IR sensors clear';

  return {
    alert_type: alertType,
    status,
    speed: 0,
    confidence: risk > 0 ? 1 : 0,
    message,
    sound: risk > 0,
  };
}

export const startModel = (req, res) => {
  const { driver_id, camera_source, camera_url, esp32_cam_ip, phone_ip } =
req.body;
  const driverId = driver_id || 'default';

  if (modelProcesses.has(driverId)) {
    const existingProcess = modelProcesses.get(driverId);
    if (existingProcess && !existingProcess.killed) {
      return res.json({
        success: true,
        running: true,
        message: 'Model is already running',
        pid: existingProcess.pid
      });
    }
  }
```

```
  const pythonPath = resolvePythonPath();
  if (!pythonPath) {
    return res.status(500).json({
      success: false,
      error: 'Python environment not found for footboard safety',
      details: 'Expected footboard safety/.venv/Scripts/python.exe',
    });
  }

  if (!existsSync(MODEL_CONFIG.cwd) ||
!existsSync(MODEL_CONFIG.scriptPath)) {
    return res.status(500).json({
      success: false,
      error: 'Footboard safety script path is invalid',
      details: `script=${MODEL_CONFIG.scriptPath}`,
    });
  }

  try {
    console.log(`[Footboard ${driverId}] Starting model...`);
    console.log(`[Footboard ${driverId}] Python: ${pythonPath}`);
    console.log(`[Footboard ${driverId}] Script:
${MODEL_CONFIG.scriptPath}`);
    console.log(`[Footboard ${driverId}] CWD: ${MODEL_CONFIG.cwd}`);

    const args = [MODEL_CONFIG.scriptPath, '--driver_id', driverId];
    if (phone_ip) args.push('--phone_ip', phone_ip);
    if (esp32_cam_ip) args.push('--esp32_cam_ip', esp32_cam_ip);
    if (camera_source) args.push('--camera_source', camera_source);
    if (camera_url) args.push('--camera_url', camera_url);

    const modelProcess = spawn(pythonPath, args, {
      cwd: MODEL_CONFIG.cwd,
      detached: false,
      stdio: ['ignore', 'pipe', 'pipe'],
      env: { ...process.env, PYTHONIOENCODING: 'utf-8' }
    });

    let startupSettled = false;
    let startupTimer = null;

    const failStartup = (error, details) => {
      if (startupSettled) return;
      startupSettled = true;

      if (startupTimer) {
        clearTimeout(startupTimer);
```

```
      startupTimer = null;
    }

    modelProcesses.delete(driverId);
    driverSystemEnabled.set(driverId, false);

    return res.status(500).json({ success: false, error, details });
  };

  modelProcess.stdout.on('data', (data) => {
    console.log(`[Footboard ${driverId}] ${data.toString().trim()}`);
  });

  modelProcess.stderr.on('data', (data) => {
    console.error(`[Footboard ${driverId} ERR] ${data.toString().trim()}`);
  });

  modelProcess.on('close', (code) => {
    console.log(`[Footboard ${driverId}] Process exited with code ${code}`);
    modelProcesses.delete(driverId);

    if (!startupSettled) {
      failStartup('Footboard safety model exited during startup', `Exit code
${code}`);
    }
  });

  modelProcess.on('error', (err) => {
    console.error(`[Footboard ${driverId}] Error: ${err.message}`);
    modelProcesses.delete(driverId);

    if (!startupSettled) {
      failStartup('Failed to start footboard safety model', err.message);
    }
  });

  startupTimer = setTimeout(() => {
    if (startupSettled) return;
    startupSettled = true;
    startupTimer = null;

    modelProcesses.set(driverId, modelProcess);
    driverSystemEnabled.set(driverId, true);

    res.json({
      success: true,
      running: true,
      message: 'Model started successfully',
```

```
        pid: modelProcess.pid
      });
    }, MODEL_CONFIG.startupGraceMs);
  } catch (error) {
    console.error('Error starting model:', error);
    res.status(500).json({ success: false, error: error.message });
  }
};

export const stopModel = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  const modelProcess = modelProcesses.get(driverId);

  if (!modelProcess) {
    return res.json({
      success: true,
      running: false,
      message: 'Model is not running'
    });
  }

  try {
    console.log(`[Footboard ${driverId}] Stopping model (PID:
${modelProcess.pid})...`);

    if (process.platform === 'win32') {
      spawn('taskkill', ['/pid', modelProcess.pid, '/f', '/t']);
    } else {
      modelProcess.kill('SIGTERM');
    }

    modelProcesses.delete(driverId);
    driverSystemEnabled.set(driverId, false);

    res.json({
      success: true,
      running: false,
      message: 'Model stopped successfully'
    });
  } catch (error) {
    console.error('Error stopping model:', error);
    res.status(500).json({ success: false, error: error.message });
  }
};

export const getModelStatus = (req, res) => {
```

```
  const { driver_id } = req.query;
  const driverId = driver_id || 'default';

  const modelProcess = modelProcesses.get(driverId);
  const isRunning = modelProcess && !modelProcess.killed;

  res.json({
    running: isRunning,
    pid: isRunning ? modelProcess.pid : null,
    driver_id: driverId
  });
};

export const receiveHeartbeat = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  driverHeartbeats.set(driverId, new Date());
  res.status(200).json({ success: true, message: 'Heartbeat received' });
};

export const getSystemStatus = (req, res) => {
  const { driver_id } = req.query;
  const driverId = driver_id || 'default';

  const lastHeartbeat = driverHeartbeats.get(driverId);
  const isEnabled = driverSystemEnabled.get(driverId) !== false;
  let systemStatus = 'offline';

  if (lastHeartbeat) {
    const timeSinceHeartbeat = Date.now() - lastHeartbeat.getTime();
    if (timeSinceHeartbeat < 10000) {
      systemStatus = 'online';
    }
  }

  res.json({
    status: systemStatus,
    enabled: isEnabled,
    driver_id: driverId,
    lastHeartbeat: lastHeartbeat ? lastHeartbeat.toISOString() : null,
    uptime: lastHeartbeat ? Math.floor((Date.now() - lastHeartbeat.getTime()) /
1000) : null
  });
};

export const toggleSystem = (req, res) => {
  const { driver_id, enabled } = req.body;
```

```
  const driverId = driver_id || 'default';

  driverSystemEnabled.set(driverId, enabled);
  console.log(`[Footboard ${driverId}] System ${enabled ? 'ENABLED' :
'DISABLED'}`);

  res.json({
    success: true,
    enabled: enabled,
    driver_id: driverId,
    message: `Safety system ${enabled ? 'enabled' : 'disabled'}`
  });
};

export const createAlert = async (req, res) => {
  try {
    const hardwareAlert = normalizeHardwareAlert(req.body);
    const {
      driver_id,
      timestamp,
      alert_type,
      status,
      speed,
      confidence,
      message,
      sound
    } = hardwareAlert ? { ...req.body, ...hardwareAlert } : req.body;
    const driverId = driver_id || 'default';

    const isEnabled = driverSystemEnabled.get(driverId) !== false;
    if (!isEnabled) {
      return res.status(200).json({ success: false, message: 'System is disabled, alert
not saved' });
    }

    const result = await pool.query(
      `INSERT INTO foot_board_safty (driver_id, timestamp, alert_type, status,
speed, confidence, message, sound)
       VALUES ($1, $2, $3, $4, $5, $6, $7, $8) RETURNING *`,
      [
        driver_id || null,
        timestamp || new Date().toISOString(),
        alert_type || 'Footboard Event',
        status || 'SAFE',
        speed || 0,
        confidence || 0,
        message || 'Footboard safety event received',
        sound || false
```

```
      ]
    );

    console.log(`[Footboard ${driverId}] Alert: ${alert_type || 'Footboard Event'} -
${status || 'SAFE'} (${speed || 0} km/h)`);
    driverHeartbeats.set(driverId, new Date());

    res.status(201).json({ success: true, data: result.rows[0] });
  } catch (error) {
    console.error('Error saving alert:', error);
    res.status(500).json({ error: error.message });
  }
};

export const getAlerts = async (req, res) => {
  try {
    const limit = parseInt(req.query.limit) || 100;
    const { driver_id } = req.query;

    let result;
    if (driver_id) {
      result = await pool.query(
        'SELECT * FROM foot_board_safty WHERE driver_id = $1 ORDER BY
created_at DESC LIMIT $2',
        [driver_id, limit]
      );
    } else {
      result = await pool.query(
        'SELECT * FROM foot_board_safty ORDER BY created_at DESC LIMIT
$1',
        [limit]
      );
    }
    res.json(result.rows);
  } catch (error) {
    console.error('Error fetching alerts:', error);
    res.status(500).json({ error: error.message });
  }
};

export const getCriticalAlerts = async (req, res) => {
  try {
    const { driver_id } = req.query;

    let query = `SELECT * FROM foot_board_safty
      WHERE (status = 'CRITICAL' OR alert_type = 'Danger')`;
    const params = [];
```

```
    if (driver_id) {
      query += ` AND driver_id = $1`;
      params.push(driver_id);
    }
    query += ` ORDER BY created_at DESC LIMIT 50`;

    const result = await pool.query(query, params);
    res.json(result.rows);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};

export const getStats = async (req, res) => {
  try {
    const { driver_id } = req.query;

    let query = `
      SELECT
        COUNT(*) as total_alerts,
        COUNT(CASE WHEN status = 'CRITICAL' THEN 1 END) as
critical_count,
        COUNT(CASE WHEN status = 'WARNING' THEN 1 END) as
warning_count,
        COUNT(CASE WHEN status = 'SAFE' THEN 1 END) as safe_count,
        MAX(created_at) as last_alert_time
      FROM foot_board_safty
      WHERE created_at > NOW() - INTERVAL '24 hours'`;

    const params = [];
    if (driver_id) {
      query += ` AND driver_id = $1`;
      params.push(driver_id);
    }

    const stats = await pool.query(query, params);

    const driverId = driver_id || 'default';
    const lastHeartbeat = driverHeartbeats.get(driverId);
    let systemStatus = 'offline';
    if (lastHeartbeat && (Date.now() - lastHeartbeat.getTime()) < 10000) {
      systemStatus = 'online';
    }

    res.json({
      systemStatus,
      lastHeartbeat: lastHeartbeat ? lastHeartbeat.toISOString() : null,
      ...stats.rows[0]
```

```
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Appendix Figure B.3: controller code

```
import express from 'express';
import {
  receiveHeartbeat,
  getSystemStatus,
  toggleSystem,
  startModel,
  stopModel,
  getModelStatus,
  createAlert,
  getAlerts,
  getCriticalAlerts,
  getStats
} from '../controllers/safetyController.js';

const router = express.Router();

// Heartbeat & Status
router.post('/heartbeat', receiveHeartbeat);
router.get('/status', getSystemStatus);
router.post('/toggle', toggleSystem);

// Model Control (start/stop Python process)
router.post('/model/start', startModel);
router.post('/model/stop', stopModel);
router.get('/model/status', getModelStatus);

// Alerts
router.post('/alerts', createAlert);
router.get('/alerts', getAlerts);
router.get('/alerts/critical', getCriticalAlerts);

// Stats
router.get('/stats', getStats);

export default router;
```

Appendix Figure B.4: route

```
import cv2
import threading
import time
import requests
import numpy as np
from dataclasses import dataclass, field
from datetime import datetime
from ultralytics import YOLO

import argparse
import sys
from pathlib import Path
from flask import Flask, request, jsonify

def _attach_repo_root():
    this_file = Path(__file__).resolve()
    for parent in this_file.parents:
        if (parent / "shared_network_config.py").exists():
            root_path = str(parent)
            if root_path not in sys.path:
                sys.path.append(root_path)
            return

_attach_repo_root()

from shared_network_config import (
    build_esp32cam_stream_url,
    build_phone_sensor_url,
    build_phone_video_url,
    load_network_config,
)

NETWORK_CONFIG = load_network_config()

# --- DEFAULT CONFIGURATION ---
DEFAULT_SERVER_URL =
NETWORK_CONFIG["FOOTBOARD_SERVER_URL"]
DEFAULT_DRIVER_ID = NETWORK_CONFIG["DRIVER_ID"]
DEFAULT_PHONE_IP = NETWORK_CONFIG["PHONE_IP"]
DEFAULT_ESP32_CAM_IP = NETWORK_CONFIG["ESP32_CAM_IP"]
DEFAULT_CAMERA_SOURCE =
NETWORK_CONFIG["FOOTBOARD_CAMERA_SOURCE"]
DEFAULT_ESP_IP = NETWORK_CONFIG["ESP32_IR_IP"]
DEFAULT_WEBHOOK_PORT =
int(NETWORK_CONFIG["FOOTBOARD_WEBHOOK_PORT"])

parser = argparse.ArgumentParser(description="Footboard Safety AI + IR
Failsafe")
```

```
parser.add_argument("--driver_id", type=str, default=DEFAULT_DRIVER_ID)
parser.add_argument("--server_url", type=str,
default=DEFAULT_SERVER_URL)
parser.add_argument("--phone_ip", type=str, default=DEFAULT_PHONE_IP)
parser.add_argument("--esp32_cam_ip", type=str,
default=DEFAULT_ESP32_CAM_IP)
parser.add_argument("--camera_source", choices=["phone", "esp32cam"],
default=DEFAULT_CAMERA_SOURCE)
parser.add_argument("--camera_url", type=str, default="")
parser.add_argument("--esp_ip", type=str, default=DEFAULT_ESP_IP)
parser.add_argument("--webhook_port", type=int,
default=DEFAULT_WEBHOOK_PORT)
parser.add_argument("--display_width", type=int, default=640)
parser.add_argument("--display_height", type=int, default=480)
parser.add_argument("--ai_imgsz", type=int, default=416)
parser.add_argument("--ai_conf", type=float, default=0.25)
parser.add_argument("--ai_interval", type=float, default=0.12, help="Minimum
seconds between YOLO inference passes")

args = parser.parse_args()

SERVER_URL = args.server_url
DRIVER_ID = args.driver_id
PHONE_IP = args.phone_ip
ESP32_CAM_IP = args.esp32_cam_ip
CAMERA_SOURCE = args.camera_source
CAMERA_URL = args.camera_url.strip()
ESP_IP = args.esp_ip
WEBHOOK_PORT = args.webhook_port
DISPLAY_SIZE = (args.display_width, args.display_height)
AI_IMGSZ = args.ai_imgsz
AI_CONF = args.ai_conf
AI_INTERVAL = max(0.03, args.ai_interval)

if CAMERA_URL:
   VIDEO_URL = CAMERA_URL
elif CAMERA_SOURCE == "esp32cam":
   VIDEO_URL = build_esp32cam_stream_url(ESP32_CAM_IP)
else:
   VIDEO_URL = build_phone_video_url(PHONE_IP)
SENSOR_URL = build_phone_sensor_url(PHONE_IP)

# Webhook App for ESP32
app = Flask(__name__)

# Shared variables
current_speed_kmh = 0.0
ir_sensor_state = {"s1": False, "s2": False, "s3": False, "online": False}
```

```
ir_state_lock = threading.Lock()
last_ir_webhook_state = (False, False, False)
last_ir_alert_time = 0.0
IR_WEBHOOK_ALERT_COOLDOWN = 0.8

# Use a session to prevent TCP socket exhaustion (TIME_WAIT)
http_session = requests.Session()

@dataclass
class AiState:
    occupied: bool = False
    max_confidence: float = 0.0
    boxes: list = field(default_factory=list)
    fps: float = 0.0
    updated_at: float = 0.0

ai_state = AiState()
ai_state_lock = threading.Lock()

def parse_bool(value):
    """Accept booleans from JSON, strings, or numeric ESP32 payloads."""
    if isinstance(value, bool):
        return value
    if isinstance(value, (int, float)):
        return value != 0
    if isinstance(value, str):
        return value.strip().lower() in {"1", "true", "yes", "on", "occupied",
"blocked"}
    return False

def get_step_label(s1, s2, s3):
    if s3:
        return "Bottom Step"
    if s2:
        return "Mid Step"
    if s1:
        return "Top Step"
    return "Clear"

# --- SERVER COMMUNICATION FUNCTIONS ---
def send_heartbeat():
    """Send heartbeat to server every 5 seconds"""
    while True:
        try:
            http_session.post(f"{SERVER_URL}/heartbeat", json={"driver_id":
DRIVER_ID}, timeout=2)
        except Exception:
            pass
```

```
        time.sleep(5)

def send_alert(alert_type, status, speed, confidence, message):
    """Send safety alert to server"""
    try:
        payload = {
            "driver_id": DRIVER_ID,
            "timestamp": datetime.now().isoformat(),
            "alert_type": alert_type,
            "status": status,
            "speed": round(speed, 2),
            "confidence": round(confidence, 3),
            "message": message,
            "sound": status in {"CRITICAL", "WARNING"},
            "detection_class": alert_type  # Passing source as detection class
        }
        response = http_session.post(f"{SERVER_URL}/alerts", json=payload,
timeout=2)
        if response.status_code == 201:
            print(f"[OK] Alert sent: {status}")
    except Exception as e:
        print(f"[ERROR] Failed to send alert: {e}")

def send_ir_webhook_alert(s1, s2, s3, risk_level):
    """Forward ESP32 IR changes immediately, even if the camera stream is
unavailable."""
    step_label = get_step_label(s1, s2, s3)
    ir_occupied = s1 or s2 or s3

    if not ir_occupied:
        send_alert(
            "IR Sensors Clear",
            "SAFE",
            current_speed_kmh,
            0.0,
            f"Footboard IR sensors clear. Speed: {current_speed_kmh:.1f} km/h."
        )
        return

    is_moving = current_speed_kmh > 5.0
    status = "CRITICAL" if s3 or is_moving or risk_level >= 3 else "WARNING"
    alert_type = f"IR Sensors Only ({step_label})"
    message = (
        f"Footboard {step_label.lower()} occupied from ESP32 IR sensor. "
        f"Speed: {current_speed_kmh:.1f} km/h."
    )
    send_alert(alert_type, status, current_speed_kmh, 1.0, message)
```

```
# --- WEBHOOK FOR ESP32 EVENTS ---
@app.route('/ir-webhook', methods=['POST'])
def receive_ir_event():
    global ir_sensor_state, last_ir_webhook_state, last_ir_alert_time
    try:
        data = request.get_json(silent=True)
        if data is not None and "s1" in data:
            s1 = parse_bool(data.get("s1", False))
            s2 = parse_bool(data.get("s2", False))
            s3 = parse_bool(data.get("s3", False))
            risk_level = int(data.get("risk", 3 if s3 else 2 if s2 else 1 if s1 else 0) or 0)

            with ir_state_lock:
                ir_sensor_state["s1"] = s1
                ir_sensor_state["s2"] = s2
                ir_sensor_state["s3"] = s3
                ir_sensor_state["online"] = True

            current_state = (s1, s2, s3)
            current_time = time.time()
            if current_state != last_ir_webhook_state and current_time -
last_ir_alert_time > IR_WEBHOOK_ALERT_COOLDOWN:
                send_ir_webhook_alert(s1, s2, s3, risk_level)
                last_ir_webhook_state = current_state
                last_ir_alert_time = current_time
            print(f"[ESP32 PUSH] IR State | Risk: {risk_level} | S1:{s1} S2:{s2}
S3:{s3}")
            return jsonify({"status": "success", "message": "IR State updated"}), 200
        else:
            return jsonify({"status": "error", "message": "Invalid payload"}), 400
    except Exception as e:
        print(f"[ESP32 PUSH ERROR] {e}")
        return jsonify({"status": "error", "message": str(e)}), 500

def run_webhook_server():
    import logging
    log = logging.getLogger('werkzeug')
    log.setLevel(logging.ERROR) # Suppress standard flask output
    app.run(host='0.0.0.0', port=WEBHOOK_PORT, debug=False,
use_reloader=False)

# --- MULTITHREADED SENSOR FETCHING (GPS) ---
def update_sensors():
    global current_speed_kmh
    while True:
        try:
            # Fetch sensor data from IP Webcam app
            response = http_session.get(SENSOR_URL, timeout=0.5)
```

```
            data = response.json()

            # Extract GPS speed (m/s) and convert to km/h (* 3.6)
            if 'gps_speed' in data:
                # Latest entry is at the end of the data list
                speed_ms = data['gps_speed']['data'][-1][1][0]
                current_speed_kmh = speed_ms * 3.6
        except Exception:
            # If GPS signal is lost or network fails
            current_speed_kmh = 0.0
        time.sleep(0.5) # Update speed every 500ms

# --- MULTITHREADED CAMERA CLASS ---
class FastCamera:
    def __init__(self, url):
        self.url = url
        self.use_mjpeg_parser = "/stream" in url or CAMERA_SOURCE ==
"esp32cam"
        self.cap = None
        self.ret = False
        self.frame = None
        self.frame_id = 0
        self.last_frame_time = 0.0
        self.capture_fps = 0.0
        self.frame_lock = threading.Lock()
        self.stopped = False
        if not self.use_mjpeg_parser:
            self.open_capture()
        threading.Thread(target=self.update, daemon=True).start()

    def open_capture(self):
        if self.cap is not None:
            self.cap.release()

        self.cap = cv2.VideoCapture(self.url, cv2.CAP_FFMPEG)
        self.cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)
        self.cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'MJPG'))
        if hasattr(cv2, "CAP_PROP_OPEN_TIMEOUT_MSEC"):
            self.cap.set(cv2.CAP_PROP_OPEN_TIMEOUT_MSEC, 3000)
        if hasattr(cv2, "CAP_PROP_READ_TIMEOUT_MSEC"):
            self.cap.set(cv2.CAP_PROP_READ_TIMEOUT_MSEC, 1000)
        self.ret, first_frame = self.cap.read()
        if self.ret and first_frame is not None:
            with self.frame_lock:
                self.frame = first_frame
                self.frame_id += 1
                self.last_frame_time = time.time()
```

```
def store_frame(self, frame):
    now = time.time()
    with self.frame_lock:
        self.ret = True
        self.frame = frame
        self.frame_id += 1
        if self.last_frame_time > 0:
            instant_fps = 1.0 / max(now - self.last_frame_time, 0.001)
            self.capture_fps = (self.capture_fps * 0.85) + (instant_fps * 0.15)
        self.last_frame_time = now

def update(self):
    if self.use_mjpeg_parser:
        self.update_mjpeg_stream()
        return

    while not self.stopped:
        ret, frame = self.cap.read()
        now = time.time()
        if ret and frame is not None:
            self.store_frame(frame)
        else:
            self.ret = ret
            if now - self.last_frame_time > 2.0:
                print("[Camera] Stream stale - reconnecting...")
                self.open_capture()
            time.sleep(0.03)

def update_mjpeg_stream(self):
    session = requests.Session()

    while not self.stopped:
        try:
            print(f"[Camera] Opening MJPEG stream: {self.url}")
            response = session.get(self.url, stream=True, timeout=(2, 1.5))
            response.raise_for_status()

            buffer = bytearray()
            last_chunk_time = time.time()

            for chunk in response.iter_content(chunk_size=2048):
                if self.stopped:
                    break
                if not chunk:
                    if time.time() - last_chunk_time > 1.5:
                        break
                    continue
```

```
                last_chunk_time = time.time()
                buffer.extend(chunk)

                latest_jpg = None
                while True:
                    start = buffer.find(b"\xff\xd8")
                    if start == -1:
                        if len(buffer) > 1024 * 512:
                            buffer.clear()
                        break

                    end = buffer.find(b"\xff\xd9", start + 2)
                    if end == -1:
                        if start > 0:
                            del buffer[:start]
                        break

                    latest_jpg = bytes(buffer[start:end + 2])
                    del buffer[:end + 2]

                if latest_jpg is not None:
                    image = cv2.imdecode(np.frombuffer(latest_jpg, dtype=np.uint8),
cv2.IMREAD_COLOR)
                    if image is not None:
                        self.store_frame(image)

            response.close()
        except Exception as e:
            if not self.stopped:
                print(f"[Camera] MJPEG stream error: {e}")

        if not self.stopped:
            print("[Camera] Reconnecting MJPEG stream...")
            time.sleep(0.5)

    def get_frame(self):
        with self.frame_lock:
            if self.frame is None:
                return None, self.frame_id, 0.0, self.capture_fps
            return self.frame.copy(), self.frame_id, self.last_frame_time,
self.capture_fps

    def release(self):
        self.stopped = True
        if self.cap is not None:
            self.cap.release()

# --- INITIALIZATION ---
```

```
model = YOLO('best.pt')
model.fuse()

# Use GPU if available (CUDA), with half-precision for speed
import torch
DEVICE = 0 if torch.cuda.is_available() else 'cpu'
USE_HALF = torch.cuda.is_available()  # FP16 only works on GPU
if not torch.cuda.is_available():
    torch.set_num_threads(max(1, min(4, torch.get_num_threads())))

def run_ai_inference(camera):
    """Run YOLO in the background so the preview never waits on inference."""
    last_inference_time = 0.0

    while not camera.stopped:
        now = time.time()
        remaining = AI_INTERVAL - (now - last_inference_time)
        if remaining > 0:
            time.sleep(remaining)

        frame, _, _, _ = camera.get_frame()
        if frame is None:
            time.sleep(0.01)
            continue

        frame = cv2.resize(frame, DISPLAY_SIZE,
interpolation=cv2.INTER_AREA)
        infer_start = time.time()
        try:
            results = model.predict(
                frame,
                imgsz=AI_IMGSZ,
                verbose=False,
                device=DEVICE,
                half=USE_HALF,
                conf=AI_CONF,
                iou=0.45
            )
        except Exception as e:
            print(f"[AI] Inference error: {e}")
            time.sleep(0.2)
            continue

        boxes = []
        yolo_occupied = False
        max_confidence = 0.0

        for r in results:
```

```
            for box in r.boxes:
                class_id = int(box.cls[0])
                label = model.names[class_id]
                confidence = float(box.conf[0])
                if label not in ['Danger', 'Warning']:
                    continue

                yolo_occupied = True
                max_confidence = max(max_confidence, confidence)
                x1, y1, x2, y2 = [int(v) for v in box.xyxy[0].tolist()]
                boxes.append({
                    "label": label,
                    "confidence": confidence,
                    "xyxy": (x1, y1, x2, y2),
                })

        elapsed = max(time.time() - infer_start, 0.001)
        with ai_state_lock:
            ai_state.occupied = yolo_occupied
            ai_state.max_confidence = max_confidence
            ai_state.boxes = boxes
            ai_state.fps = 1.0 / elapsed
            ai_state.updated_at = time.time()

        last_inference_time = time.time()

def draw_ai_boxes(frame, boxes):
    for item in boxes:
        x1, y1, x2, y2 = item["xyxy"]
        label = item["label"]
        confidence = item["confidence"]
        color = (0, 0, 255) if label == "Danger" else (0, 255, 255)
        cv2.rectangle(frame, (x1, y1), (x2, y2), color, 2)
        cv2.putText(
            frame,
            f"{label} {confidence:.2f}",
            (x1, max(20, y1 - 8)),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.55,
            color,
            2,
        )

# Start the Speed Tracker Thread
threading.Thread(target=update_sensors, daemon=True).start()

# Start the Flask Webhook for ESP32 IR Sensor Pushes
threading.Thread(target=run_webhook_server, daemon=True).start()
```

```
# Start the Heartbeat Thread (sends status to server)
threading.Thread(target=send_heartbeat, daemon=True).start()

cam = FastCamera(VIDEO_URL)
threading.Thread(target=run_ai_inference, args=(cam,), daemon=True).start()
prev_time = time.time()
last_display_frame_id = -1
last_repaint_time = 0.0
last_alert_time = 0  # Throttle alerts to avoid spam
ALERT_COOLDOWN = 2  # seconds between alerts

print(f"RiyaNeth Camera Source: {CAMERA_SOURCE}")
print(f"Video URL: {VIDEO_URL}")
if CAMERA_SOURCE == "phone":
    print(f"Phone sensor URL: {SENSOR_URL}")
print(f"IR Sensor Webhook Listening on Port {WEBHOOK_PORT}")
print(f"Configured ESP32 IR IP: {ESP_IP}")
print(f"Display: {DISPLAY_SIZE[0]}x{DISPLAY_SIZE[1]} | YOLO
imgsz={AI_IMGSZ} | conf={AI_CONF:.2f} | AI
interval={AI_INTERVAL:.2f}s")
print("Logic: ALERT if Speed > 5km/h AND (AI sees person OR IR is blocked).")

while True:
    frame, frame_id, frame_time, camera_fps = cam.get_frame()
    if frame is None:
        time.sleep(0.01)
        continue

    is_new_camera_frame = frame_id != last_display_frame_id
    if not is_new_camera_frame and time.time() - last_repaint_time < 0.2:
        time.sleep(0.005)
        continue
    if is_new_camera_frame:
        last_display_frame_id = frame_id

    frame = cv2.resize(frame, DISPLAY_SIZE, interpolation=cv2.INTER_AREA)
    annotated_frame = frame.copy()

    # --- SENSOR FUSION LOGIC ---
    # 1. AI Detection (updated by the background inference thread)
    with ai_state_lock:
        yolo_occupied = ai_state.occupied
        ai_max_confidence = ai_state.max_confidence
        ai_boxes = list(ai_state.boxes)
        ai_fps = ai_state.fps

    draw_ai_boxes(annotated_frame, ai_boxes)
```

```
    # 2. IR Sensor Hardware Detection
    with ir_state_lock:
        current_ir_state = dict(ir_sensor_state)

    ir_occupied = current_ir_state["s1"] or current_ir_state["s2"] or
current_ir_state["s3"]
    ir_danger = current_ir_state["s3"] # Bottom step is immediate danger

    # Combined Safety State
    footboard_occupied = yolo_occupied or ir_occupied

    # Unsafe condition: Movement > 5km/h while steps are occupied (or step 3 is
blocked)
    is_moving = current_speed_kmh > 5.0

    # Determine alert source identity for the dashboard
    detection_source = "Safe"
    max_confidence = 0.0

    if footboard_occupied:
        if yolo_occupied and ir_occupied:
            detection_source = "AI + IR Sensors"
            max_confidence = 1.0
        elif yolo_occupied:
            detection_source = "AI Vision Only"
            max_confidence = ai_max_confidence
        elif ir_occupied:
            detection_source = "IR Sensors Only"
            max_confidence = 1.0 # Hardware blocked

            # Map occupied step for specific messaging
            if current_ir_state["s3"]: detection_source += " (Bottom Step)"
            elif current_ir_state["s2"]: detection_source += " (Mid Step)"
            elif current_ir_state["s1"]: detection_source += " (Top Step)"

    current_time = time.time()

    # --- Build step-specific label for message (S1=Entry, S2=Mid, S3=Bottom) ---
    active_ir_steps = []
    if current_ir_state.get("s1"): active_ir_steps.append("S1-Entry")
    if current_ir_state.get("s2"): active_ir_steps.append("S2-Mid")
    if current_ir_state.get("s3"): active_ir_steps.append("S3-Bottom")
    step_label = ", ".join(active_ir_steps) if active_ir_steps else ""

    # Critical Alert: Moving while occupied OR someone is on the bottom step
(Step 3)
    if (footboard_occupied and is_moving) or ir_danger:
```

```
        overlay_color = (0, 0, 255) # Bright Red

        if ir_danger and not is_moving:
            status_msg = (
                f"Child detected on bottom step ({step_label}). "
                f"Bus is stationary — remove child immediately."
            )
        else:
            step_info = f" [{step_label}]" if step_label else ""
            status_msg = (
                f"Footboard occupied{step_info} at {current_speed_kmh:.1f} km/h"
                f" via {detection_source}. Stop immediately."
            )

        # Send critical alert to server (with cooldown)
        if current_time - last_alert_time > ALERT_COOLDOWN:
            send_alert(detection_source, "CRITICAL", current_speed_kmh,
max_confidence, status_msg)
            last_alert_time = current_time

    # Warning Alert: Occupied (Steps 1 or 2, or AI) but stationary
    elif footboard_occupied:
        overlay_color = (0, 255, 255) # Yellow
        step_info = f" [{step_label}]" if step_label else ""
        status_msg = (
            f"Footboard occupied{step_info} — bus stationary"
            f" via {detection_source}. Monitor situation."
        )
        # Send warning alert to server (with cooldown)
        if current_time - last_alert_time > ALERT_COOLDOWN:
            send_alert(detection_source, "WARNING", current_speed_kmh,
max_confidence, status_msg)
            last_alert_time = current_time

    # Safe
    else:
        overlay_color = (0, 255, 0) # Green
        status_msg = f"Footboard clear. Speed: {current_speed_kmh:.1f} km/h."

    # --- UI RENDERING ---
    # Top Status Bar
    cv2.rectangle(annotated_frame, (0, 0), (DISPLAY_SIZE[0], 60), overlay_color,
-1)
    cv2.putText(annotated_frame, status_msg, (15, 40),
                cv2.FONT_HERSHEY_DUPLEX, 0.7, (255, 255, 255), 2)

    # Hardware / AI Status Indicator
```

```
    cv2.putText(annotated_frame, f"IR: {'ON' if current_ir_state['online'] else
'OFF'}", (20, 100),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0) if
current_ir_state['online'] else (0, 0, 255), 2)
    cv2.putText(annotated_frame, f"GPS Speed: {'Active' if current_speed_kmh > 0
else 'Waiting'}", (20, 130),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0) if
current_speed_kmh > 0 else (0, 200, 255), 2)

    # FPS Display
    curr_time = time.time()
    fps = 1 / max(curr_time - prev_time, 0.001)
    prev_time = curr_time
    last_repaint_time = curr_time
    frame_age = curr_time - frame_time if frame_time else 99.0
    cv2.putText(annotated_frame, f"Cam FPS: {camera_fps:.1f}",
(DISPLAY_SIZE[0] - 160, 100),
                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 2)
    cv2.putText(annotated_frame, f"AI FPS: {ai_fps:.1f}", (DISPLAY_SIZE[0] -
160, 130),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
    if frame_age > 1.0:
        cv2.putText(annotated_frame, "CAMERA STALE", (DISPLAY_SIZE[0] -
180, 160),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

    cv2.imshow("RiyaNeth: AI + GPS Integrated Monitor", annotated_frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        cam.release()
        break

cv2.destroyAllWindows()
```

Appendix Figure B.5: safty.py

Appendix C: Driver Monitoring Code Evidence

```
import { pool } from '../config/postgres.js';
import { spawn, exec } from 'child_process';
import path from 'path';
import { fileURLToPath } from 'url';
import os from 'os';
import { existsSync } from 'fs';

// Get current directory
```

```
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Store last heartbeat time in memory (per driver)
const driverHeartbeats = new Map();

// Store system enabled state (per driver)
const driverSystemEnabled = new Map();

// Store running model processes (per driver)
const modelProcesses = new Map();

// Calibration runtime status (per driver)
const calibrationStatusByDriver = new Map();
const modelStdoutRemainders = new Map();

const CALIBRATION_STATUS_PREFIX = '[CALIB_STATUS]';
const DEFAULT_CALIBRATION_TOTAL_FRAMES = 30;

function createDefaultCalibrationStatus() {
  return {
    status: 'IDLE',
    inProgress: false,
    progressPercent: 0,
    framesCollected: 0,
    totalFrames: DEFAULT_CALIBRATION_TOTAL_FRAMES,
    instruction: '',
    rejectReason: '',
    skipReason: '',
    attempt: 0,
    autoRetryUsed: false,
    source: 'manual',
    stageName: '',
    stagePhase: '',
    acceptedSampleCount: 0,
    skippedFrameCount: 0,
    updatedAt: null,
  };
}

function getCalibrationStatusSnapshot(driverId) {
  return calibrationStatusByDriver.get(driverId) || createDefaultCalibrationStatus();
}

function setCalibrationStatus(driverId, patch) {
  const next = {
    ...getCalibrationStatusSnapshot(driverId),
    ...patch,
```

```
    updatedAt: new Date().toISOString(),
  };
  calibrationStatusByDriver.set(driverId, next);
  return next;
}

function consumeCalibrationLine(driverId, line) {
  if (!line.startsWith(CALIBRATION_STATUS_PREFIX)) return false;

  const payloadText =
line.slice(CALIBRATION_STATUS_PREFIX.length).trim();
  if (!payloadText) return true;

  try {
    const payload = JSON.parse(payloadText);
    const status = typeof payload.status === 'string' ? payload.status :
'IN_PROGRESS';
    const progressPercent = Number(payload.progressPercent ?? payload.progress
?? 0);
    const framesCollected = Number(payload.framesCollected ?? 0);
    const totalFrames = Number(payload.totalFrames ??
DEFAULT_CALIBRATION_TOTAL_FRAMES);
    const inProgress = payload.inProgress ?? ['IN_PROGRESS',
'REJECTED'].includes(status);

    setCalibrationStatus(driverId, {
      status,
      inProgress: Boolean(inProgress),
      progressPercent: Number.isFinite(progressPercent) ? progressPercent : 0,
      framesCollected: Number.isFinite(framesCollected) ? framesCollected : 0,
      totalFrames: Number.isFinite(totalFrames) && totalFrames > 0 ? totalFrames :
DEFAULT_CALIBRATION_TOTAL_FRAMES,
      instruction: typeof payload.instruction === 'string' ? payload.instruction : '',
      rejectReason: typeof payload.rejectReason === 'string' ? payload.rejectReason
: '',
      skipReason: typeof payload.skipReason === 'string' ? payload.skipReason : '',
      attempt: Number.isFinite(Number(payload.attempt)) ?
Number(payload.attempt) : 0,
      autoRetryUsed: Boolean(payload.autoRetryUsed),
      source: typeof payload.source === 'string' ? payload.source : 'manual',
      stageName: typeof payload.stageName === 'string' ? payload.stageName : '',
      stagePhase: typeof payload.stagePhase === 'string' ? payload.stagePhase : '',
      acceptedSampleCount:
Number.isFinite(Number(payload.acceptedSampleCount)) ?
Number(payload.acceptedSampleCount) : 0,
      skippedFrameCount: Number.isFinite(Number(payload.skippedFrameCount))
? Number(payload.skippedFrameCount) : 0,
    });
```

```
  } catch (err) {
    console.warn(`[Driver Monitor ${driverId}] Invalid calibration status payload:
${payloadText}`);
  }

  return true;
}

function processModelStdoutChunk(driverId, chunk) {
  const previous = modelStdoutRemainders.get(driverId) || '';
  const combined = `${previous}${chunk}`;
  const lines = combined.split(/\r?\n/);
  modelStdoutRemainders.set(driverId, lines.pop() ?? '');

  for (const rawLine of lines) {
    const line = rawLine.trim();
    if (!line) continue;
    if (!consumeCalibrationLine(driverId, line)) {
      console.log(`[Driver Monitor ${driverId}] ${line}`);
    }
  }
}

// Model configuration for driver monitoring
const DRIVER_MONITOR_ROOT = path.join(__dirname, '../../Driver
monitering');
const MODEL_CONFIG = {
  pythonCandidates: [
    path.join(DRIVER_MONITOR_ROOT, '.venv/Scripts/python.exe'),
    path.join(DRIVER_MONITOR_ROOT, 'venv/Scripts/python.exe'),
    path.join(__dirname, '../../.venv/Scripts/python.exe'),
  ],
  scriptPath: path.join(DRIVER_MONITOR_ROOT, 'driver_monitoring/main.py'),
  cwd: path.join(DRIVER_MONITOR_ROOT, 'driver_monitoring'),
  startupGraceMs: 800,
};

function resolvePythonPath() {
  const overridePath = process.env.DRIVER_MONITOR_PYTHON;
  if (overridePath && existsSync(overridePath)) return overridePath;

  for (const candidate of MODEL_CONFIG.pythonCandidates) {
    if (existsSync(candidate)) return candidate;
  }

  return null;
}
```

```
// ── Startup cleanup ──────────────────────────────────────────────────────────
// When nodemon restarts the server, previously spawned Python processes
// become orphans (Node.js does NOT kill children on exit on Windows).
// Kill any leftover main.py processes so they don't show a stale window.
function killOrphanedMonitors() {
  const scriptName = 'main.py';
  if (os.platform() === 'win32') {
    // On Windows, use WMIC to find and kill Python processes running main.py
    exec(
      `wmic process where "name='python.exe' and CommandLine like
'%main.py%'" call terminate`,
      (err, stdout) => {
        if (stdout && stdout.includes('ReturnValue = 0')) {
          console.log('[Driver Monitor] Cleaned up orphaned monitor processes on
startup.');
        }
      }
    );
  } else {
    exec(`pkill -f ${scriptName}`, () => {});
  }
}
killOrphanedMonitors();

// POST - Start model process
export const startModel = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  // Check if already running
  if (modelProcesses.has(driverId)) {
    const existingProcess = modelProcesses.get(driverId);
    if (existingProcess && !existingProcess.killed) {
      return res.json({
        success: true,
        running: true,
        message: 'Driver monitoring model is already running',
        pid: existingProcess.pid
      });
    }
  }

  const pythonPath = resolvePythonPath();
  if (!pythonPath) {
    return res.status(500).json({
      success: false,
      error: 'Python environment not found for driver monitor',
```

```
      details: 'Expected Driver monitering/.venv/Scripts/python.exe',
    });
  }

  if (!existsSync(MODEL_CONFIG.cwd) ||
!existsSync(MODEL_CONFIG.scriptPath)) {
    return res.status(500).json({
      success: false,
      error: 'Driver monitor script path is invalid',
      details: `script=${MODEL_CONFIG.scriptPath}`,
    });
  }

  try {
    console.log(`🚗 Starting driver monitoring model for driver ${driverId}...`);
    console.log(`📂 Python: ${pythonPath}`);
    console.log(`📂 Script: ${MODEL_CONFIG.scriptPath}`);
    console.log(`📂 CWD: ${MODEL_CONFIG.cwd}`);

    const modelProcess = spawn(pythonPath, [MODEL_CONFIG.scriptPath], {
      cwd: MODEL_CONFIG.cwd,
      stdio: ['pipe', 'pipe', 'pipe'],
      detached: false,
      env: { ...process.env, PYTHONIOENCODING: 'utf-8' }
    });

    modelStdoutRemainders.set(driverId, '');
    setCalibrationStatus(driverId, createDefaultCalibrationStatus());

    let startupSettled = false;

    const failStartup = (error, details) => {
      if (startupSettled) return;
      startupSettled = true;
      modelProcesses.delete(driverId);
      modelStdoutRemainders.delete(driverId);
      setCalibrationStatus(driverId, {
        ...createDefaultCalibrationStatus(),
        status: 'FAILED',
        inProgress: false,
        instruction: 'Calibration unavailable: monitor failed to start.',
      });
      return res.status(500).json({ success: false, error, details });
    };

    modelProcess.stdout.on('data', (data) => {
      processModelStdoutChunk(driverId, data.toString());
```

```
  });

  modelProcess.stderr.on('data', (data) => {
    console.error(`[Driver Monitor ${driverId} ERR] ${data.toString().trim()}`);
  });

  modelProcess.on('close', (code) => {
    console.log(`[Driver Monitor ${driverId}] Process exited with code ${code}`);
    modelProcesses.delete(driverId);
    modelStdoutRemainders.delete(driverId);

    const previousStatus = getCalibrationStatusSnapshot(driverId);
    if (previousStatus.inProgress) {
      setCalibrationStatus(driverId, {
        ...createDefaultCalibrationStatus(),
        status: 'FAILED',
        inProgress: false,
        instruction: 'Calibration interrupted because monitor stopped.',
      });
    } else {
      setCalibrationStatus(driverId, createDefaultCalibrationStatus());
    }

    if (!startupSettled) {
      failStartup('Model process exited during startup', `Exit code ${code}`);
    }
  });

  modelProcess.on('error', (err) => {
    console.error(`[Driver Monitor ${driverId}] Failed to start:`, err);
    failStartup('Failed to start model process', err.message);
  });

  modelProcesses.set(driverId, modelProcess);

  setTimeout(() => {
    if (startupSettled) return;

    if (modelProcess.exitCode !== null || modelProcess.killed) {
      failStartup('Model process failed to stay alive after launch', `Exit code
${modelProcess.exitCode}`);
      return;
    }

    startupSettled = true;
    res.json({
      success: true,
      running: true,
```

```
        message: 'Driver monitoring model started successfully',
        pid: modelProcess.pid,
      });
    }, MODEL_CONFIG.startupGraceMs);

  } catch (error) {
    console.error('Failed to start driver monitoring model:', error);
    res.status(500).json({
      success: false,
      error: 'Failed to start model',
      details: error.message
    });
  }
};

// POST - Stop model process
export const stopModel = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  if (!modelProcesses.has(driverId)) {
    return res.json({
      success: true,
      running: false,
      message: 'Model is not running'
    });
  }

  try {
    const modelProcess = modelProcesses.get(driverId);

    if (modelProcess && !modelProcess.killed) {
      const pid = modelProcess.pid;

      if (os.platform() === 'win32') {
        // SIGTERM is not real on Windows — use taskkill to force-kill the process
tree
        exec(`taskkill /F /T /PID ${pid}`, (err) => {
          if (err) console.error(`[Driver Monitor] taskkill error for PID ${pid}:`,
err.message);
        });
      } else {
        modelProcess.kill('SIGTERM');
        setTimeout(() => {
          if (!modelProcess.killed) modelProcess.kill('SIGKILL');
        }, 3000);
      }
    }
```

```
    modelProcesses.delete(driverId);
    modelStdoutRemainders.delete(driverId);
    setCalibrationStatus(driverId, createDefaultCalibrationStatus());

    res.json({
      success: true,
      running: false,
      message: 'Driver monitoring model stopped successfully'
    });

  } catch (error) {
    console.error('Failed to stop driver monitoring model:', error);
    res.status(500).json({
      success: false,
      error: 'Failed to stop model',
      details: error.message
    });
  }
};

// GET - Check model status
export const getModelStatus = (req, res) => {
  const { driver_id } = req.query;
  const driverId = driver_id || 'default';

  const modelProcess = modelProcesses.get(driverId);
  const isRunning = Boolean(modelProcess && !modelProcess.killed &&
modelProcess.exitCode === null);

  res.json({
    running: isRunning,
    pid: isRunning ? modelProcess.pid : null
  });
};

// POST - Trigger runtime calibration on a running model process
export const calibrateModel = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  const modelProcess = modelProcesses.get(driverId);
  const isRunning = Boolean(modelProcess && !modelProcess.killed &&
modelProcess.exitCode === null);

  if (!isRunning) {
    return res.status(409).json({
      success: false,
```

```
      error: 'Driver monitor model is not running',
      details: 'Start the model before triggering calibration.',
    });
  }

  if (!modelProcess.stdin || modelProcess.stdin.destroyed ||
!modelProcess.stdin.writable) {
    return res.status(500).json({
      success: false,
      error: 'Model command channel unavailable',
      details: 'Cannot send recalibration command to monitor process.',
    });
  }

  try {
    setCalibrationStatus(driverId, {
      status: 'IN_PROGRESS',
      inProgress: true,
      progressPercent: 0,
      framesCollected: 0,
      instruction: 'Calibration starting. Look forward with a neutral face until it
completes.',
      rejectReason: '',
      skipReason: '',
      attempt: 1,
      autoRetryUsed: false,
      source: 'manual',
      stageName: 'FORWARD',
      stagePhase: 'collect',
      acceptedSampleCount: 0,
      skippedFrameCount: 0,
    });

    modelProcess.stdin.write('recalibrate\n');

    return res.json({
      success: true,
      status: 'IN_PROGRESS',
      message: 'Calibration command sent successfully.',
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      error: 'Failed to trigger calibration',
      details: error.message,
    });
  }
};
```

```
// POST - Retry only the currently paused calibration stage
export const retryCalibrationStage = (req, res) => {
  const { driver_id } = req.body;
  const driverId = driver_id || 'default';

  const modelProcess = modelProcesses.get(driverId);
  const isRunning = Boolean(modelProcess && !modelProcess.killed &&
modelProcess.exitCode === null);
  if (!isRunning) {
    return res.status(409).json({
      success: false,
      error: 'Driver monitor model is not running',
      details: 'Start the model before retrying calibration.',
    });
  }

  const currentStatus = getCalibrationStatusSnapshot(driverId);
  if (currentStatus.status !== 'PAUSED') {
    return res.status(409).json({
      success: false,
      error: 'Calibration stage is not paused',
      details: 'Retry stage is only available when calibration is paused.',
    });
  }

  if (!modelProcess.stdin || modelProcess.stdin.destroyed ||
!modelProcess.stdin.writable) {
    return res.status(500).json({
      success: false,
      error: 'Model command channel unavailable',
      details: 'Cannot send retry command to monitor process.',
    });
  }

  try {
    setCalibrationStatus(driverId, {
      status: 'IN_PROGRESS',
      inProgress: true,
      instruction: currentStatus.instruction || 'Retrying the paused stage. Follow the
stage prompt.',
      rejectReason: '',
      skipReason: '',
      autoRetryUsed: false,
    });

    modelProcess.stdin.write('retry\n');
```

```
    return res.json({
      success: true,
      status: 'IN_PROGRESS',
      message: 'Retry command sent successfully.',
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      error: 'Failed to retry calibration stage',
      details: error.message,
    });
  }
};

// GET - Read latest calibration progress/status
export const getCalibrationStatus = (req, res) => {
  const { driver_id } = req.query;
  const driverId = driver_id || 'default';
  res.json(getCalibrationStatusSnapshot(driverId));
};

// POST - Receive heartbeat from Python model
export const receiveHeartbeat = async (req, res) => {
  const { driver_id } = req.body;

  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }

  const now = new Date();
  driverHeartbeats.set(driver_id, now);

  // Initialize system as enabled if not set
  if (!driverSystemEnabled.has(driver_id)) {
    driverSystemEnabled.set(driver_id, true);
  }

  res.json({
    success: true,
    timestamp: now,
    system_enabled: driverSystemEnabled.get(driver_id)
  });
};

// GET - Get system status
export const getSystemStatus = (req, res) => {
  const { driver_id } = req.query;
```

```
  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }

  const lastHeartbeat = driverHeartbeats.get(driver_id);
  const isEnabled = driverSystemEnabled.get(driver_id) ?? true;

  // System is online if heartbeat received within last 15 seconds
  const isOnline = lastHeartbeat && (Date.now() - lastHeartbeat.getTime() <
15000);

  res.json({
    status: isOnline ? 'online' : 'offline',
    enabled: isEnabled,
    lastHeartbeat: lastHeartbeat || null
  });
};

// POST - Toggle system on/off
export const toggleSystem = (req, res) => {
  const { driver_id, enabled } = req.body;

  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }

  driverSystemEnabled.set(driver_id, enabled);

  res.json({
    success: true,
    enabled: enabled
  });
};

// POST - Create new alert
export const createAlert = async (req, res) => {
  const { driver_id, alert_type, severity, confidence, message, sound,
detection_class } = req.body;

  if (!driver_id || !alert_type || !severity) {
    return res.status(400).json({ error: 'driver_id, alert_type, and severity are
required' });
  }

  try {
    const query = `
      INSERT INTO driver_monitoring (driver_id, alert_type, severity, confidence,
message, sound, detection_class, timestamp)
```

```
      VALUES ($1, $2, $3, $4, $5, $6, $7, $8)
      RETURNING *
    `;

    const values = [driver_id, alert_type, severity, confidence || null, message || null,
sound || false, detection_class || null, Date.now()];
    const result = await pool.query(query, values);

    res.status(201).json(result.rows[0]);
  } catch (error) {
    console.error('Failed to create driver monitoring alert:', error);
    res.status(500).json({ error: 'Failed to create alert' });
  }
};

// GET - Get alerts for driver
export const getAlerts = async (req, res) => {
  const { driver_id, limit = 50 } = req.query;

  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }

  try {
    const query = `
      SELECT id, driver_id, timestamp, alert_type, severity, confidence, message,
sound, detection_class, created_at
      FROM driver_monitoring
      WHERE driver_id = $1
      ORDER BY created_at DESC
      LIMIT $2
    `;

    const result = await pool.query(query, [driver_id, parseInt(limit)]);
    res.json(result.rows);
  } catch (error) {
    console.error('Failed to fetch driver monitoring alerts:', error);
    res.status(500).json({ error: 'Failed to fetch alerts' });
  }
};

// GET - Get critical alerts (DANGER only)
export const getCriticalAlerts = async (req, res) => {
  const { driver_id, limit = 20 } = req.query;

  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }
```

```
  try {
    const query = `
      SELECT id, driver_id, timestamp, alert_type, severity, confidence, message,
sound, detection_class, created_at
      FROM driver_monitoring
      WHERE driver_id = $1 AND severity = 'DANGER'
      ORDER BY created_at DESC
      LIMIT $2
    `;

    const result = await pool.query(query, [driver_id, parseInt(limit)]);
    res.json(result.rows);
  } catch (error) {
    console.error('Failed to fetch critical alerts:', error);
    res.status(500).json({ error: 'Failed to fetch critical alerts' });
  }
};

// GET - Get statistics
export const getStats = async (req, res) => {
  const { driver_id } = req.query;

  if (!driver_id) {
    return res.status(400).json({ error: 'driver_id is required' });
  }

  try {
    const query = `
      SELECT
        COUNT(*) as total_alerts,
        COUNT(CASE WHEN severity = 'DANGER' THEN 1 END) as
danger_count,
        COUNT(CASE WHEN severity = 'WARNING' THEN 1 END) as
warning_count,
        COUNT(CASE WHEN alert_type = 'drowsy' THEN 1 END) as
drowsy_count,
        COUNT(CASE WHEN alert_type = 'phone_use' THEN 1 END) as
phone_count,
        COUNT(CASE WHEN alert_type = 'looking_away' THEN 1 END) as
distracted_count,
        COUNT(CASE WHEN alert_type = 'yawning' THEN 1 END) as
yawning_count
      FROM driver_monitoring
      WHERE driver_id = $1
        AND created_at > NOW() - INTERVAL '24 hours'
    `;
```

```
    const result = await pool.query(query, [driver_id]);
    res.json(result.rows[0]);
  } catch (error) {
    console.error('Failed to fetch stats:', error);
    res.status(500).json({ error: 'Failed to fetch stats' });
  }
};
```

Appendix Figure C.1: Driver monitoring conroller

```
import express from 'express';
import {
  receiveHeartbeat,
  getSystemStatus,
  toggleSystem,
  createAlert,
  getAlerts,
  getCriticalAlerts,
  getStats,
  startModel,
  stopModel,
  getModelStatus,
  calibrateModel,
  retryCalibrationStage,
  getCalibrationStatus
} from '../controllers/driverMonitorController.js';

const router = express.Router();

// Model control
router.post('/model/start', startModel);
router.post('/model/stop', stopModel);
router.get('/model/status', getModelStatus);
router.post('/model/calibrate', calibrateModel);
router.post('/model/calibration/retry', retryCalibrationStage);
router.get('/model/calibration/status', getCalibrationStatus);

// Heartbeat & Status
router.post('/heartbeat', receiveHeartbeat);
router.get('/status', getSystemStatus);
router.post('/toggle', toggleSystem);

// Alerts
router.post('/alerts', createAlert);
router.get('/alerts', getAlerts);
router.get('/alerts/critical', getCriticalAlerts);
```

```
// Stats
router.get('/stats', getStats);

export default router;
```

Appendix Figure C.2: Driver monitoring route

```
import sys
import json
import time
import threading
import queue
from pathlib import Path
import cv2
import numpy as np
import mediapipe as mp
import requests
from datetime import datetime


def _attach_repo_root():
    this_file = Path(__file__).resolve()
    for parent in this_file.parents:
        if (parent / "shared_network_config.py").exists():
            root_path = str(parent)
            if root_path not in sys.path:
                sys.path.append(root_path)
            return


_attach_repo_root()

from shared_network_config import load_network_config

from config import (
    CAMERA_INDEX, FRAME_WIDTH, FRAME_HEIGHT,
    MIN_DETECTION_CONF, MIN_TRACKING_CONF,
    LEFT_EYE, RIGHT_EYE, MOUTH,
    STAGE_DISPLAY, PHONE_ENABLED, IRIS_ENABLED,
    SERVER_ENABLED, HEARTBEAT_INTERVAL_SEC,
ALERT_TIMEOUT_SEC,
    DEBUG_PRINT_EVERY, DRAW_EYE_CONTOURS,
    HEAD_NOD_MIN_PERCLOS, HEAD_NOD_REQUIRE_EYE_FATIGUE,
    CALIBRATION_FRAMES,
```

```
    EAR_STRONG_CLOSED_RATIO,
    YAW_THRESHOLD, PITCH_DOWN_THRESHOLD,
MIRROR_YAW_THRESHOLD,
    DISTRACTION_YAW_THRESHOLD, GAZE_CONFIDENCE_MIN,
    PHONE_LOOK_EAR_OPEN_MARGIN,
)
from face_metrics import get_dominant_ear, get_mar, get_head_pose,
reset_head_pose_smoothing
from iris_gaze import GazeAttentionTracker, get_gaze, get_attention_state
from drowsiness import DrowsinessDetector
from calibration import Calibrator
from state_machine import StateMachine
from phone_detector import PhoneDetector
from signal_quality import build_measurement_quality, normalize_ear


#
═══════════════════════════════════════════════════════════════════
═══════════════════
#  Server Integration Configuration
#
═══════════════════════════════════════════════════════════════════
═══════════════════
NETWORK_CONFIG = load_network_config()
SERVER_URL =
NETWORK_CONFIG["DRIVER_MONITOR_SERVER_URL"]
DRIVER_ID = NETWORK_CONFIG["DRIVER_ID"]

# Alert cooldown tracking
_last_alert_time = {}
ALERT_COOLDOWN = 5  # seconds between same alert type
_alert_queue = queue.Queue(maxsize=64)
_command_queue = queue.Queue(maxsize=16)
_last_heartbeat_error_log = 0.0
_last_alert_error_log = 0.0
_stop_event = threading.Event()  # set to cleanly stop the main loop

CALIBRATION_STATUS_PREFIX = "[CALIB_STATUS]"
CALIBRATION_PROFILE_PATH =
Path(__file__).with_name("driver_monitor_calibration.json")

# Map driver states to alert types for the server
STATE_TO_ALERT = {
    "DROWSY":       {"alert_type": "drowsy",       "severity": "DANGER",
"sound": True},
    "MICROSLEEP":   {"alert_type": "microsleep",   "severity": "DANGER",
"sound": True},
```

```
    "FATIGUED":     {"alert_type": "fatigue",       "severity": "WARNING",
"sound": False},
    "YAWNING":      {"alert_type": "yawning",       "severity": "WARNING",
"sound": False},
    "DISTRACTED":   {"alert_type": "looking_away",  "severity": "WARNING",
"sound": True},
    "LOOKING DOWN": {"alert_type": "looking_away",  "severity":
"WARNING", "sound": False},
    "PHONE IN LAP": {"alert_type": "looking_away",  "severity": "WARNING",
"sound": False},
    "PHONE USE":    {"alert_type": "phone_use",     "severity": "DANGER",
"sound": True},
    "EYES OFF ROAD":{"alert_type": "looking_away",  "severity": "WARNING",
"sound": True},
    "NO FACE":      {"alert_type": "no_face",       "severity": "WARNING",
"sound": False},
}


def save_calibration_profile(calibrator, camera_index, width, height):
    profile = calibrator.export_profile()
    if not profile:
        return False

    payload = {
        "camera_index": camera_index,
        "frame_width": width,
        "frame_height": height,
        "saved_at": datetime.utcnow().isoformat() + "Z",
        "profile": profile,
    }
    try:
        CALIBRATION_PROFILE_PATH.write_text(json.dumps(payload,
indent=2), encoding="utf-8")
        return True
    except Exception as exc:
        print(f"[WARN] Failed to save calibration profile: {exc}")
        return False


def try_load_calibration_profile(calibrator, camera_index, width, height):
    if not CALIBRATION_PROFILE_PATH.exists():
        return False

    try:
        payload =
json.loads(CALIBRATION_PROFILE_PATH.read_text(encoding="utf-8"))
    except Exception as exc:
```

```
        print(f"[WARN] Failed to read calibration profile: {exc}")
        return False

    if payload.get("camera_index") != camera_index:
        return False
    if payload.get("frame_width") != width or payload.get("frame_height") !=
height:
        return False

    return calibrator.load_profile(payload.get("profile"))


def send_heartbeat():
    """Send heartbeat every 5 seconds to keep system status online."""
    global _last_heartbeat_error_log

    if not SERVER_ENABLED:
        return

    while not _stop_event.is_set():
        try:
            response = requests.post(
                f"{SERVER_URL}/heartbeat",
                json={"driver_id": DRIVER_ID},
                timeout=ALERT_TIMEOUT_SEC
            )
            if response.status_code == 200:
                data = response.json()
                ts = datetime.now().strftime('%H:%M:%S')
                print(f"[{ts}] ❤️ Heartbeat OK — system {'enabled' if
data.get('system_enabled') else 'disabled'}")
        except requests.exceptions.RequestException as e:
            now = time.time()
            if now - _last_heartbeat_error_log >= HEARTBEAT_INTERVAL_SEC:
                ts = datetime.now().strftime('%H:%M:%S')
                print(f"[{ts}] ❌ Heartbeat error: {str(e)[:60]}")
                _last_heartbeat_error_log = now
        _stop_event.wait(timeout=HEARTBEAT_INTERVAL_SEC)


def _send_alert_payload(payload, state):
    """Send one alert payload (runs in background thread)."""
    global _last_alert_error_log

    try:
        response = requests.post(f"{SERVER_URL}/alerts", json=payload,
timeout=ALERT_TIMEOUT_SEC)
```

```
        if response.status_code == 201:
            ts = datetime.now().strftime('%H:%M:%S')
            print(f"[{ts}] 🚨 Alert sent: {state} ({payload['severity']})")
    except requests.exceptions.RequestException as e:
        now = time.time()
        if now - _last_alert_error_log >= HEARTBEAT_INTERVAL_SEC:
            ts = datetime.now().strftime('%H:%M:%S')
            print(f"[{ts}] ❌ Alert send failed: {str(e)[:60]}")
            _last_alert_error_log = now


def _alert_sender_worker():
    """Background worker to prevent main-loop stalls on network latency."""
    while True:
        item = _alert_queue.get()
        if item is None:
            break
        payload, state = item
        _send_alert_payload(payload, state)


def send_alert(state, confidence=None, detection_class=None):
    """Send an alert to the server if state is non-normal, with cooldown."""
    global _last_alert_time

    if not SERVER_ENABLED:
        return

    if state not in STATE_TO_ALERT:
        return

    alert_info = STATE_TO_ALERT[state]
    alert_key = f"{alert_info['alert_type']}_{alert_info['severity']}"
    now = time.time()

    # Cooldown check
    if alert_key in _last_alert_time:
        if now - _last_alert_time[alert_key] < ALERT_COOLDOWN:
            return
    _last_alert_time[alert_key] = now

    payload = {
        "driver_id": DRIVER_ID,
        "alert_type": alert_info["alert_type"],
        "severity": alert_info["severity"],
        "confidence": confidence,
        "message": f"Driver state: {state}",
```

```
        "sound": alert_info["sound"],
        "detection_class": detection_class or state,
    }

    try:
        _alert_queue.put_nowait((payload, state))
    except queue.Full:
        # Drop excess alerts rather than stalling the main loop.
        return


def emit_calibration_status(
    status,
    progress=0,
    frames_collected=0,
    total_frames=CALIBRATION_FRAMES,
    instruction="",
    reject_reason="",
    skip_reason="",
    attempt=1,
    auto_retry_used=False,
    source="manual",
    stage_name="",
    stage_phase="",
    accepted_samples=0,
    skipped_frames=0,
    in_progress=None,
):
    """Emit structured calibration status for server-side parsing."""
    if in_progress is None:
        in_progress = status in ("IN_PROGRESS", "REJECTED")

    payload = {
        "status": status,
        "inProgress": bool(in_progress),
        "progressPercent": int(progress),
        "framesCollected": int(max(0, frames_collected)),
        "totalFrames": int(max(1, total_frames)),
        "instruction": instruction,
        "rejectReason": reject_reason,
        "skipReason": skip_reason,
        "attempt": int(max(1, attempt)),
        "autoRetryUsed": bool(auto_retry_used),
        "source": source,
        "stageName": stage_name,
        "stagePhase": stage_phase,
        "acceptedSampleCount": int(max(0, accepted_samples)),
        "skippedFrameCount": int(max(0, skipped_frames)),
```

```
        "timestamp": datetime.utcnow().isoformat() + "Z",
    }
    print(f"{CALIBRATION_STATUS_PREFIX}{json.dumps(payload)}",
flush=True)


def _stdin_command_listener():
    """Listen for runtime commands sent from the Node.js process."""
    while True:
        line = sys.stdin.readline()
        if not line:
            # stdin closed — Node.js process ended, trigger clean shutdown
            _stop_event.set()
            break

        cmd = line.strip().lower()
        if cmd in ("recalibrate", "r"):
            try:
                _command_queue.put_nowait("recalibrate")
            except queue.Full:
                pass
        elif cmd in ("retry",):
            try:
                _command_queue.put_nowait("retry")
            except queue.Full:
                pass
        elif cmd in ("stop", "quit", "exit"):
            print("[INFO] Stop command received — shutting down.")
            _stop_event.set()
            break


# ═══════════════════════════════════════════════════════════════════
# ═══════════════════
#  Initialise MediaPipe Face Mesh
# ═══════════════════════════════════════════════════════════════════
# ═══════════════════
mp_face_mesh = mp.solutions.face_mesh
mp_drawing   = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

face_mesh = mp_face_mesh.FaceMesh(
    max_num_faces=1,
    min_detection_confidence=MIN_DETECTION_CONF,
    min_tracking_confidence=MIN_TRACKING_CONF,
```

```
    refine_landmarks=IRIS_ENABLED,   # skip iris refinement on low-spec mode
when disabled
)


#
═══════════════════════════════════════════════════════════════════════
═══════════════════
#  Helper — draw a dashboard overlay on the frame
#
═══════════════════════════════════════════════════════════════════════
═══════════════════
def draw_dashboard(frame, driver_state):
    """Render metric panel and status bar on the video frame."""
    h, w = frame.shape[:2]
    state   = driver_state["state"]
    color   = driver_state["color"]
    stage   = driver_state["stage"]

    # ── Top status bar ──────────────────────────────────────
    cv2.rectangle(frame, (0, 0), (w, 40), color, -1)
    label = f'{state}  (Stage {stage})'
    cv2.putText(frame, label, (10, 28),
                cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2)

    # ── Metrics panel (bottom-left) ─────────────────────────
    panel_x = 10
    metrics = [
        f"EAR:      {driver_state['ear']:.3f}",
        f"EAR Conf: {driver_state.get('ear_confidence', 0.0):.2f}",
        f"EAR Norm: {driver_state.get('ear_norm', 0.0):.2f}",
        f"MAR:      {driver_state['mar']:.3f}",
        f"PERCLOS:  {driver_state['perclos']:.1f}%",
        f"Yaw:      {driver_state['yaw']:.1f}°",
        f"Pitch:    {driver_state['pitch']:.2f}°",
        f"Nod:      {'YES' if driver_state.get('head_nod_active') else 'No '}
dP={driver_state.get('head_nod_drop', 0.0):.2f}",
        f"Blink ms: {driver_state['avg_blink_ms']:.0f}",
        f"Gaze H/V: {driver_state['gaze_h']:.2f} / {driver_state['gaze_v']:.2f}",
        f"Gaze Cnf: {driver_state.get('gaze_confidence', 0.0):.2f}
Eye:{driver_state.get('dominant_eye', 'na')}",
        f"Quality:  {driver_state.get('quality_level', 'NA')}
{driver_state.get('quality_score', 0.0):.2f}",
        f"Mirror:   {driver_state.get('mirror_elapsed', 0.0):.2f}s",
        f"Phone:    {'YES ' + str(driver_state.get('phone_conf', 0)) if
driver_state.get('phone_detected') else 'No'}",
        f"Reason:   {driver_state.get('reason', '-')[:34]}",
    ]
```

```
    panel_height = len(metrics) * 22 + 20
    panel_y = max(18, h - panel_height)

    # Semi-transparent background
    overlay = frame.copy()
    cv2.rectangle(overlay, (panel_x - 5, panel_y - 15),
                  (panel_x + 395, panel_y + len(metrics) * 22 + 5),
                  (0, 0, 0), -1)
    cv2.addWeighted(overlay, 0.55, frame, 0.45, 0, frame)

    for i, txt in enumerate(metrics):
        cv2.putText(frame, txt, (panel_x, panel_y + i * 22),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.50, (255, 255, 255), 1)

    # ── FPS (top-right) ──────────────────────────────────────
    fps = driver_state.get("fps", 0)
    cv2.putText(frame, f"FPS: {fps:.0f}", (w - 120, 28),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)


def _draw_calibration_basic(frame, progress, instruction="", reject_reason="",
stage_name="FORWARD"):
    """Show calibration progress bar, stage guidance, and any reject reason."""
    h, w = frame.shape[:2]
    cv2.rectangle(frame, (0, 0), (w, 40), (255, 255, 0), -1)
    cv2.putText(frame, "CALIBRATING — LOOK FORWARD", (10, 28),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
    # Progress bar
    bar_w = int((w - 40) * progress / 100)
    cv2.rectangle(frame, (20, 50), (20 + bar_w, 65), (0, 255, 0), -1)
    cv2.rectangle(frame, (20, 50), (w - 20, 65), (255, 255, 255), 1)
    cv2.putText(frame, f"{progress}%", (w // 2 - 20, 62),
                cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 255), 1)
    cv2.putText(frame, "Neutral face: eyes open, mouth closed, no
talking/yawning", (10, 90),
                cv2.FONT_HERSHEY_SIMPLEX, 0.48, (0, 255, 255), 1)
    if reject_reason:
        cv2.putText(frame, f"Calibration retry: {reject_reason[:65]}", (10, 112),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 1)


def draw_eye_contours(frame, landmarks, w, h):
    """Draw small circles around eye landmarks for visual feedback."""
    for idx_list, color in [(LEFT_EYE, (0, 255, 0)), (RIGHT_EYE, (0, 255, 0))]:
        pts = [(int(landmarks[i].x * w), int(landmarks[i].y * h)) for i in idx_list]
        for pt in pts:
            cv2.circle(frame, pt, 2, color, -1)
```

```
def get_face_bounds(landmarks, w, h):
    coords = np.asarray([(lm.x * w, lm.y * h) for lm in landmarks],
dtype=np.float32)
    x_min, y_min = np.min(coords, axis=0)
    x_max, y_max = np.max(coords, axis=0)
    return float(x_min), float(y_min), float(x_max), float(y_max)


def filter_phone_detection_for_driver(phone_result, landmarks, w, h, yaw,
attention_meta=None):
    """
    Reject phone detections that are clearly outside the driver's interaction zone.
    This cuts false positives from dashboard objects and other people entering
frame.
    """
    context = {
        "phone_in_driver_zone": bool(phone_result.get("phone_detected")),
        "phone_filter_reason": "",
    }
    if not phone_result.get("phone_detected") or phone_result.get("bbox") is None:
        return phone_result, context

    x_min, y_min, x_max, y_max = get_face_bounds(landmarks, w, h)
    face_w = max(40.0, x_max - x_min)
    face_h = max(40.0, y_max - y_min)
    face_cx = (x_min + x_max) / 2.0

    px1, py1, px2, py2 = phone_result["bbox"]
    phone_cx = (float(px1) + float(px2)) / 2.0
    phone_cy = (float(py1) + float(py2)) / 2.0

    zone_x_min = max(0.0, face_cx - face_w * 1.10)
    zone_x_max = min(float(w), face_cx + face_w * 1.35)
    zone_y_min = max(0.0, y_min - face_h * 0.15)
    zone_y_max = min(float(h), y_max + face_h * 2.60)

    in_driver_zone = (
        zone_x_min <= phone_cx <= zone_x_max
        and zone_y_min <= phone_cy <= zone_y_max
    )

    mirror_candidate = bool((attention_meta or {}).get("mirror_candidate", False))
    mirror_band = MIRROR_YAW_THRESHOLD <= abs(float(yaw)) <
DISTRACTION_YAW_THRESHOLD
    side_limit = face_w * (1.05 if mirror_candidate and mirror_band else 1.45)
    too_far_side = abs(phone_cx - face_cx) > side_limit
    edge_margin = 0.06 * float(w)
```

```
    clipped_to_edge = float(px1) <= edge_margin or float(px2) >= (float(w) -
edge_margin)

    reject_reason = ""
    if not in_driver_zone:
        reject_reason = "phone box outside driver zone"
    elif too_far_side and mirror_candidate and mirror_band:
        reject_reason = "mirror glance with phone box far from driver"
    elif clipped_to_edge and too_far_side:
        reject_reason = "phone box anchored at frame edge"

    if not reject_reason:
        return phone_result, context

    filtered = dict(phone_result)
    filtered.update({
        "phone_detected": False,
        "confidence": 0.0,
        "bbox": None,
        "class_name": "",
    })
    context["phone_in_driver_zone"] = False
    context["phone_filter_reason"] = reject_reason
    return filtered, context


def draw_calibration(frame, calibrator):
    """Render the staged calibration overlay using the calibrator state."""
    from config import CALIBRATION_PRE_STABLE_FRAMES

    h, w = frame.shape[:2]
    phase_label = str(getattr(calibrator, "stage_phase", "settle") or "settle").upper()
    title = f"CALIBRATING - {calibrator.current_stage_name.replace('_', ' ')}
[{phase_label}]"
    cv2.rectangle(frame, (0, 0), (w, 40), (255, 255, 0), -1)
    cv2.putText(frame, title[:72], (10, 28),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
    bar_w = int((w - 40) * calibrator.progress / 100)
    cv2.rectangle(frame, (20, 50), (20 + bar_w, 65), (0, 255, 0), -1)
    cv2.rectangle(frame, (20, 50), (w - 20, 65), (255, 255, 255), 1)
    cv2.putText(frame, f"{calibrator.progress}%", (w // 2 - 20, 62),
                cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 255), 1)
    if calibrator.instruction:
        cv2.putText(frame, calibrator.instruction[:70], (10, 90),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.48, (0, 255, 255), 1)
    cv2.putText(frame, "Only clean frames count; blinks and jitter are skipped.",
(10, 112),
                cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 255, 255), 1)
```

```
    cv2.putText(
        frame,
        f"Stage samples:
{calibrator.accepted_sample_count}/{calibrator.current_stage_target}  "
        f"Skipped: {calibrator.skipped_frame_count}",
        (10, 134),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.45,
        (255, 255, 255),
        1,
    )
    cv2.putText(frame, f"Stable gate:
{calibrator.stage_stable_count}/{CALIBRATION_PRE_STABLE_FRAMES}",
(10, 156),
                cv2.FONT_HERSHEY_SIMPLEX, 0.45, (255, 255, 255), 1)
    cv2.putText(frame, f"Bad frames: {calibrator.stage_bad_frames}", (250, 156),
                cv2.FONT_HERSHEY_SIMPLEX, 0.45, (255, 255, 255), 1)
    if calibrator.last_skip_reason:
        cv2.putText(frame, f"Skip: {calibrator.last_skip_reason[:60]}", (10, 178),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.45, (120, 220, 255), 1)
    if calibrator.last_reject_reason:
        cv2.putText(frame, f"Reason: {calibrator.last_reject_reason[:60]}", (10, 200),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 1)
    if calibrator.pause_required:
        cv2.putText(frame, "Calibration paused - retry the current stage from the
app", (10, 222),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)


#
═══════════════════════════════════════════════════════════════════════
═══════════════════
#  Main loop
#
═══════════════════════════════════════════════════════════════════════
═══════════════════
# ── Display window name
──────────────────────────────────────────────
WINDOW_NAME = "SISURAKSHA - Driver Monitor"
ALLOW_KEYBOARD_QUIT = sys.stdin.isatty()


def should_quit_from_key(key):
    """Allow quit via ESC always; allow 'q' only in interactive local runs."""
    return key == 27 or (ALLOW_KEYBOARD_QUIT and key == ord('q'))


def update_hold_timer(timer_state, key, active, now):
```

```
    """Track how long a condition has stayed continuously active."""
    if active:
        if timer_state[key] is None:
            timer_state[key] = now
        return now - timer_state[key]

    timer_state[key] = None
    return 0.0


def main():
    # Open webcam
    if isinstance(CAMERA_INDEX, int):
        cap = cv2.VideoCapture(CAMERA_INDEX, cv2.CAP_DSHOW)
        if not cap.isOpened():
            cap.release()
            cap = cv2.VideoCapture(CAMERA_INDEX)
    else:
        cap = cv2.VideoCapture(CAMERA_INDEX)

    cap.set(cv2.CAP_PROP_FRAME_WIDTH, FRAME_WIDTH)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, FRAME_HEIGHT)

    if not cap.isOpened():
        print("[ERROR] Cannot open webcam. Check CAMERA_INDEX in
config.py")
        sys.exit(1)

    print("[INFO] Camera opened successfully. Press 'q' to quit, 'r' to recalibrate.")

    # Listen for runtime commands from Node.js (stdin pipe).
    command_listener_thread = threading.Thread(target=_stdin_command_listener,
daemon=True)
    command_listener_thread.start()

    # Start server-related threads only when enabled.
    if SERVER_ENABLED:
        heartbeat_thread = threading.Thread(target=send_heartbeat, daemon=True)
        heartbeat_thread.start()
        alert_sender_thread = threading.Thread(target=_alert_sender_worker,
daemon=True)
        alert_sender_thread.start()
        print(f"[INFO] Server heartbeat started → {SERVER_URL}")
    else:
        print("[INFO] Server integration disabled (SERVER_ENABLED=False).")

    # Module instances
    reset_head_pose_smoothing()
```

```
    calibrator    = Calibrator()
    drowsiness    = DrowsinessDetector()
    state_machine = StateMachine()
    attention_tracker = GazeAttentionTracker()
    phone_detector = PhoneDetector() if PHONE_ENABLED else None
    event_timers = {
        "mirror_check": None,
        "distracted": None,
        "looking_down": None,
        "high_mar": None,
        "low_ear": None,
    }
    prev_yaw_for_rate = None
    prev_pitch_for_rate = None
    prev_metric_time = None

    calibration_source = "manual"
    calibration_attempt = 1
    calibration_auto_retry_used = False
    manual_retry_required = False
    calibration_active = False
    last_calibration_status = None
    last_retry_event_reason = ""

    def report_calibration_status(status, instruction=None, reject_reason="",
skip_reason="", in_progress=None):
        nonlocal last_calibration_status

        payload = {
            "status": status,
            "progress": 100 if status == "COMPLETED" else calibrator.progress,
            "frames_collected": (
                calibrator.total_frames_required if status == "COMPLETED" else
calibrator.frames_collected
            ),
            "total_frames": calibrator.total_frames_required,
            "instruction": instruction if instruction is not None else
calibrator.instruction,
            "reject_reason": reject_reason,
            "skip_reason": skip_reason,
            "attempt": calibration_attempt,
            "auto_retry_used": calibration_auto_retry_used,
            "source": calibration_source,
            "stage_name": calibrator.current_stage_name if not calibrator.calibrated
else "",
            "stage_phase": calibrator.stage_phase,
            "accepted_samples": calibrator.accepted_sample_count if not
calibrator.calibrated else 0,
```

```
            "skipped_frames": calibrator.skipped_frame_count if not
calibrator.calibrated else 0,
            "in_progress": in_progress,
        }

        if payload == last_calibration_status:
            return

        emit_calibration_status(
            status=payload["status"],
            progress=payload["progress"],
            frames_collected=payload["frames_collected"],
            total_frames=payload["total_frames"],
            instruction=payload["instruction"],
            reject_reason=payload["reject_reason"],
            skip_reason=payload["skip_reason"],
            attempt=payload["attempt"],
            auto_retry_used=payload["auto_retry_used"],
            source=payload["source"],
            stage_name=payload["stage_name"],
            stage_phase=payload["stage_phase"],
            accepted_samples=payload["accepted_samples"],
            skipped_frames=payload["skipped_frames"],
            in_progress=payload["in_progress"],
        )
        last_calibration_status = payload

    def reset_calibration(source="manual"):
        nonlocal calibrator
        nonlocal drowsiness
        nonlocal state_machine
        nonlocal attention_tracker
        nonlocal calibration_source
        nonlocal calibration_attempt
        nonlocal calibration_auto_retry_used
        nonlocal manual_retry_required
        nonlocal calibration_active
        nonlocal last_calibration_status
        nonlocal last_retry_event_reason
        nonlocal event_timers
        nonlocal prev_yaw_for_rate
        nonlocal prev_pitch_for_rate
        nonlocal prev_metric_time

        reset_head_pose_smoothing()
        calibrator = Calibrator()
        drowsiness = DrowsinessDetector()
        state_machine = StateMachine()
```

```
    attention_tracker = GazeAttentionTracker()
    event_timers = {key: None for key in event_timers}
    prev_yaw_for_rate = None
    prev_pitch_for_rate = None
    prev_metric_time = None

    calibration_source = source
    calibration_attempt = 1
    calibration_auto_retry_used = False
    manual_retry_required = False
    calibration_active = True
    last_calibration_status = None
    last_retry_event_reason = ""
    report_calibration_status("IN_PROGRESS")

reset_calibration(source="startup")

prev_time = time.time()
fps = 0.0
last_frame_warn = 0.0
window_initialized = False

while True:
    while True:
        try:
            command = _command_queue.get_nowait()
        except queue.Empty:
            break

        if command == "recalibrate":
            print("[INFO] Recalibrating from app command — look forward...")
            reset_calibration(source="manual")
        elif command == "retry" and calibrator.pause_required:
            print("[INFO] Retrying paused calibration stage...")
            reset_head_pose_smoothing()
            calibration_attempt += 1
            calibrator.acknowledge_retry()
            manual_retry_required = False
            calibration_auto_retry_used = False
            last_calibration_status = None
            last_retry_event_reason = ""
            report_calibration_status("IN_PROGRESS")

    ret, frame = cap.read()
    if not ret:
        now = time.time()
        if now - last_frame_warn >= 2.0:
            print("[WARN] Frame grab failed — retrying")
```

```
            last_frame_warn = now
        continue

    # Resize to configured resolution regardless of camera native resolution.
    # IP webcam streams ignore cap.set() hints and deliver full-resolution frames.
    frame = cv2.resize(frame, (FRAME_WIDTH, FRAME_HEIGHT))

    # Create and size the OpenCV window only after the first valid frame.
    # This avoids showing an empty gray window during camera warm-up.
    if not window_initialized:
        cv2.namedWindow(WINDOW_NAME, cv2.WINDOW_NORMAL)
        cv2.resizeWindow(WINDOW_NAME, FRAME_WIDTH,
FRAME_HEIGHT)
        window_initialized = True

    h, w = frame.shape[:2]

    # Flip horizontally for mirror-like experience on laptop webcam
    frame = cv2.flip(frame, 1)

    # Convert BGR → RGB for MediaPipe
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = face_mesh.process(rgb)

    face_visible = False
    ear = mar = yaw = pitch = perclos = 0.0
    ear_filtered = 0.0
    ear_confidence = 1.0
    avg_blink_ms = 0.0
    gaze_h, gaze_v = -1.0, -1.0
    gaze_meta = {"gaze_confidence": 0.0, "dominant_eye": "none"}
    attention = "UNKNOWN"
    attention_meta = {}
    is_microsleep = False
    microsleep_elapsed = 0.0
    head_nod_active = False
    head_nod_elapsed = 0.0
    head_nod_drop = 0.0
    yaw_rate = 0.0
    raw_reason = ""
    phone_result = {"phone_detected": False, "confidence": 0.0,
                    "bbox": None, "class_name": "", "inference_ms": 0.0}
    phone_context = {"phone_in_driver_zone": False, "phone_filter_reason": ""}

    # ── Phone detection (runs on full frame) ──────
    if phone_detector and phone_detector.enabled:
        phone_result = phone_detector.detect(frame)
```

```
        if results.multi_face_landmarks:
            face_visible = True
            landmarks = results.multi_face_landmarks[0].landmark

            # ── Compute metrics ──────────────────────────────
            left_pts = [(landmarks[i].x * w, landmarks[i].y * h) for i in LEFT_EYE]
            right_pts = [(landmarks[i].x * w, landmarks[i].y * h) for i in RIGHT_EYE]
            left_span = float(np.linalg.norm(np.array(left_pts[0]) -
np.array(left_pts[3])))
            right_span = float(np.linalg.norm(np.array(right_pts[0]) -
np.array(right_pts[3])))

            yaw, pitch, roll = get_head_pose(landmarks, w, h)
            ear, ear_confidence = get_dominant_ear(
                landmarks, w, h, yaw, calibrator.max_left_span,
calibrator.max_right_span
            )
            mar   = get_mar(landmarks, MOUTH, w, h)

            gaze_h, gaze_v, gaze_meta = get_gaze(landmarks, w, h)
            attention, attention_meta = get_attention_state(
                yaw, pitch, gaze_h, gaze_v,
                calibrator=calibrator,
                tracker=attention_tracker,
                gaze_meta=gaze_meta,
                current_time=time.time(),
            )

            # ── Calibration phase ────────────────────────────
            if not calibrator.calibrated:
                if not calibration_active:
                    cv2.putText(frame, "Calibration idle - tap Recalibrate in app", (10,
100),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.62, (0, 255, 255), 2)
                    cv2.putText(frame, "Press 'r' to recalibrate locally", (10, 128),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.55, (255, 255, 255), 2)
                    if DRAW_EYE_CONTOURS:
                        draw_eye_contours(frame, landmarks, w, h)

                    cv2.imshow(WINDOW_NAME, frame)
                    key = cv2.waitKey(1) & 0xFF
                    if should_quit_from_key(key):
                        break
                    elif key == ord('r'):
                        print("[INFO] Recalibrating — look forward...")
                        reset_calibration(source="manual")
                    continue
```

```
            if manual_retry_required:
                draw_calibration(frame, calibrator)
                cv2.putText(frame, "Calibration paused - tap Retry Stage in app", (10,
134),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 0, 255), 2)
                if DRAW_EYE_CONTOURS:
                    draw_eye_contours(frame, landmarks, w, h)

                cv2.imshow(WINDOW_NAME, frame)
                key = cv2.waitKey(1) & 0xFF
                if should_quit_from_key(key):
                    break
                elif key == ord('r'):
                    print("[INFO] Recalibrating — look forward...")
                    reset_calibration(source="manual")
                continue

            calibrator.update(
                ear, yaw, pitch, mar,
                left_span, right_span,
                gaze_h=gaze_h, gaze_v=gaze_v,
                ear_confidence=ear_confidence,
                gaze_confidence=gaze_meta.get("gaze_confidence", 0.0),
            )
            manual_retry_required = calibrator.pause_required
            calibration_auto_retry_used = calibrator.auto_retry_used > 0
            if not calibrator.last_reject_reason:
                last_retry_event_reason = ""

            if calibrator.calibrated:
                calibration_active = False
                last_retry_event_reason = ""
                save_calibration_profile(calibrator, CAMERA_INDEX,
FRAME_WIDTH, FRAME_HEIGHT)
                report_calibration_status(
                    "COMPLETED",
                    instruction="Calibration complete. Monitoring resumed.",
                    in_progress=False,
                )
            elif calibrator.pause_required:
                last_retry_event_reason = calibrator.last_reject_reason
                report_calibration_status(
                    "PAUSED",
                    reject_reason=calibrator.last_reject_reason,
                    skip_reason=calibrator.last_skip_reason,
                    in_progress=False,
                )
            else:
```

```
            if (
                calibrator.last_reject_reason
                and calibration_auto_retry_used
                and calibrator.last_reject_reason != last_retry_event_reason
            ):
                last_retry_event_reason = calibrator.last_reject_reason
                report_calibration_status(
                    "REJECTED",
                    reject_reason=calibrator.last_reject_reason,
                    skip_reason=calibrator.last_skip_reason,
                    in_progress=True,
                )
            else:
                report_calibration_status(
                    "IN_PROGRESS",
                    reject_reason="",
                    skip_reason=calibrator.last_skip_reason,
                )

        draw_calibration(frame, calibrator)
        if DRAW_EYE_CONTOURS:
            draw_eye_contours(frame, landmarks, w, h)

        # Show frame and continue
        cv2.imshow(WINDOW_NAME, frame)
        key = cv2.waitKey(1) & 0xFF
        if should_quit_from_key(key):
            break
        continue

    # ── Post-calibration processing ──────────────
    # Subtract baseline so forward-looking = (0, 0)
    yaw   = yaw   - calibrator.baseline_yaw
    pitch = pitch - calibrator.baseline_pitch

    now = time.time()
    attention, attention_meta = get_attention_state(
        yaw, pitch, gaze_h, gaze_v,
        calibrator=calibrator,
        tracker=attention_tracker,
        gaze_meta=gaze_meta,
        current_time=now,
    )
    phone_result, phone_context = filter_phone_detection_for_driver(
        phone_result,
        landmarks,
        w,
        h,
```

```
            yaw,
            attention_meta=attention_meta,
        )

        yaw_rate = 0.0
        pitch_rate = 0.0
        if prev_metric_time is not None and prev_yaw_for_rate is not None and
prev_pitch_for_rate is not None:
            dt = max(now - prev_metric_time, 1e-3)
            yaw_rate = (yaw - prev_yaw_for_rate) / dt
            pitch_rate = (pitch - prev_pitch_for_rate) / dt
        prev_metric_time = now
        prev_yaw_for_rate = yaw
        prev_pitch_for_rate = pitch

        ear_filtered = drowsiness.filter_ear(ear)
        quality = build_measurement_quality(
            face_visible=True,
            ear_confidence=ear_confidence,
            gaze_confidence=gaze_meta.get("gaze_confidence", 0.0),
            left_span=left_span,
            right_span=right_span,
            calibrator=calibrator,
            yaw_rate=yaw_rate,
            pitch_rate=pitch_rate,
        )
        ear_norm = normalize_ear(calibrator, ear_filtered, yaw,
quality.eye_visibility)
        drowsiness.update_ear_history(ear_filtered, weight=quality.overall)
        drowsiness.process_blink(ear_filtered, now,
blink_threshold=calibrator.blink_threshold)

        perclos      = drowsiness.get_perclos(calibrator.ear_closed_threshold)
        avg_blink_ms = drowsiness.get_avg_blink_ms()
        is_low_ear_fatigue, low_ear_elapsed = drowsiness.check_low_ear_fatigue(
            ear_norm, quality.level
        )
        slow_blink_detected = drowsiness.is_slow_blinking()
        mirror_protected = attention_meta.get("mirror_protected", False)
        eyes_only_mirror = attention_meta.get("eyes_only_mirror", False)
        mirror_soft_protected = (
            (mirror_protected or eyes_only_mirror or
attention_meta.get("mirror_candidate", False))
            and MIRROR_YAW_THRESHOLD <= abs(yaw) <
DISTRACTION_YAW_THRESHOLD
            and not attention_meta.get("mirror_overdue", False)
        )
        gaze_confidence = gaze_meta.get("gaze_confidence", 0.0)
```

```
        phone_like_downlook = (
            attention == "PHONE IN LAP"
            and gaze_confidence >= GAZE_CONFIDENCE_MIN
            and ear_filtered >= (calibrator.ear_fatigue_threshold +
PHONE_LOOK_EAR_OPEN_MARGIN)
        )
        if mirror_soft_protected and abs(yaw) >= MIRROR_YAW_THRESHOLD
and ear_confidence < 0.90:
            drowsiness.low_ear_start = None
            is_low_ear_fatigue = False
            low_ear_elapsed = 0.0
        if phone_like_downlook and ear_confidence >= 0.70:
            drowsiness.low_ear_start = None
            is_low_ear_fatigue = False
            low_ear_elapsed = 0.0
        slow_blink = slow_blink_detected or is_low_ear_fatigue
        head_nod_active, head_nod_elapsed, head_nod_drop =
drowsiness.check_gradual_head_nod(pitch)
        nod_for_microsleep = head_nod_active and not phone_like_downlook

        distracted_elapsed = update_hold_timer(
            event_timers, "distracted",
            (abs(yaw) >= DISTRACTION_YAW_THRESHOLD or attention ==
"EYES OFF ROAD") and not mirror_soft_protected,
            now,
        )
        mirror_elapsed = update_hold_timer(
            event_timers, "mirror_check",
            mirror_soft_protected or attention_meta.get("mirror_candidate", False),
            now,
        )
        looking_down_elapsed = update_hold_timer(
            event_timers, "looking_down",
            pitch > PITCH_DOWN_THRESHOLD or phone_like_downlook or
attention == "PHONE IN LAP",
            now,
        )
        high_mar_elapsed = update_hold_timer(
            event_timers, "high_mar",
            mar > calibrator.mar_yawn_threshold and not mirror_soft_protected and
abs(yaw) < MIRROR_YAW_THRESHOLD,
            now,
        )
        update_hold_timer(
            event_timers, "low_ear",
            ear_norm < 0.82,
            now,
        )
```

```
        closed_for_microsleep = max(
            calibrator.ear_closed_threshold,
            calibrator.blink_threshold - 0.03,
        )

        strong_closed_threshold = min(
            closed_for_microsleep - 0.015,
            calibrator.ear_closed_threshold * EAR_STRONG_CLOSED_RATIO,
        )
        microsleep_support = (
            perclos >= HEAD_NOD_MIN_PERCLOS
            or nod_for_microsleep
            or (slow_blink_detected and not mirror_soft_protected and not
phone_like_downlook)
        )
        if (mirror_soft_protected or phone_like_downlook) and not
nod_for_microsleep and perclos < HEAD_NOD_MIN_PERCLOS:
            microsleep_support = False

        is_microsleep, microsleep_elapsed, microsleep_reason =
drowsiness.check_microsleep(
            ear_filtered,
            ear_norm,
            closed_for_microsleep,
            yaw=yaw,
            ear_confidence=ear_confidence,
            support_present=microsleep_support,
            strong_closed_threshold=strong_closed_threshold,
            mirror_protected=mirror_soft_protected,
            quality_level=quality.level,
        )

        # Gradual head droop can be an additional microsleep cue, but only
        # escalate when eye/fatigue evidence is also present.
        if nod_for_microsleep and not mirror_soft_protected:
            nod_eye_confirmed = (
                is_low_ear_fatigue
                or ear_norm < 0.92
                or perclos >= HEAD_NOD_MIN_PERCLOS
            )
            if (not HEAD_NOD_REQUIRE_EYE_FATIGUE) or
nod_eye_confirmed:
                is_microsleep = True
                microsleep_elapsed = max(microsleep_elapsed, head_nod_elapsed)
                microsleep_reason = "head nod with supporting eye evidence"

        # ── State determination ──────────────────────
```

```
        raw_state, stage, color, raw_reason = state_machine.determine_state(
            face_visible=True,
            ear=ear, mar=mar, yaw=yaw, pitch=pitch,
            perclos=perclos,
            is_microsleep=is_microsleep,
            is_slow_blink=slow_blink,
            attention=attention,
            calibrated=True,
            phone_detected=phone_result["phone_detected"],
            mar_threshold=calibrator.mar_yawn_threshold,
            context={
                "mirror_elapsed": mirror_elapsed,
                "mirror_candidate": attention_meta.get("mirror_candidate", False),
                "mirror_protected": mirror_soft_protected,
                "mirror_soft_protected": mirror_soft_protected,
                "mirror_overdue": attention_meta.get("mirror_overdue", False),
                "eyes_only_mirror": eyes_only_mirror,
                "offroad_by_yaw": abs(yaw) >=
DISTRACTION_YAW_THRESHOLD,
                "distracted_elapsed": distracted_elapsed,
                "looking_down_elapsed": looking_down_elapsed,
                "high_mar_elapsed": high_mar_elapsed,
                "low_ear_elapsed": low_ear_elapsed,
                "ear_confidence": ear_confidence,
                "gaze_confidence": gaze_confidence,
                "phone_like_downlook": phone_like_downlook,
                "yaw_rate": yaw_rate,
                "quality_level": quality.level,
                "quality_reason": quality.reason,
                "quality_score": quality.overall,
                "microsleep_reason": microsleep_reason,
                "phone_in_driver_zone": phone_context.get("phone_in_driver_zone",
False),
            },
        )
        smoothed_state = state_machine.get_smoothed_state(raw_state)

        # Re-fetch color for the smoothed state
        stage = state_machine._state_to_stage(smoothed_state)
        color = STAGE_DISPLAY.get(stage, ("", (255, 255, 255)))[1]

        # ── Console debug (print every 15 frames) ─
        if DEBUG_PRINT_EVERY > 0:
            if hasattr(state_machine, '_dbg_ctr'):
                state_machine._dbg_ctr += 1
            else:
                state_machine._dbg_ctr = 0
            if state_machine._dbg_ctr % DEBUG_PRINT_EVERY == 0:
```

```
                print(f"[DBG] raw={raw_state:<14} smooth={smoothed_state:<14} "
                      f"EAR={ear:.3f}/{ear_filtered:.3f} MAR={mar:.3f}
yaw={yaw:.1f} "
                      f"pitch={pitch:.1f} nod={head_nod_active}
dP={head_nod_drop:.2f} "
                      f"PERCLOS={perclos:.1f}% normEAR={ear_norm:.2f}
q={quality.level} econf={ear_confidence:.2f} gconf={gaze_confidence:.2f} "
                      f"attn={attention:<13} mirror={mirror_elapsed:.2f}s
eyesOnly={eyes_only_mirror} phoneDown={phone_like_downlook} "
                      f"reason={raw_reason} phone={phone_result['phone_detected']}")

            # Draw eye contours
            if DRAW_EYE_CONTOURS:
                draw_eye_contours(frame, landmarks, w, h)

        else:
            # No face detected
            attention_tracker.reset()
            for timer_key in event_timers:
                event_timers[timer_key] = None
            prev_yaw_for_rate = None
            prev_pitch_for_rate = None
            prev_metric_time = None
            ear = 0.0
            ear_confidence = 0.0
            ear_filtered = 0.0
            ear_norm = 0.0
            mar = 0.0
            perclos = 0.0
            yaw = 0.0
            pitch = 0.0
            avg_blink_ms = 0.0
            head_nod_active = False
            head_nod_drop = 0.0
            head_nod_elapsed = 0.0
            gaze_h = 0.0
            gaze_v = 0.0
            gaze_meta = {"gaze_confidence": 0.0, "dominant_eye": "none"}
            attention_meta = {"mirror_elapsed": 0.0}
            attention = "UNKNOWN"
            raw_reason = "no face visible"
            quality = build_measurement_quality(False, 0.0, 0.0, 0.0, 0.0, calibrator,
0.0, 0.0)
            if calibrator.calibrated:
                smoothed_state = "NO FACE"
                stage = 0
                color = (128, 128, 128)
            elif calibration_active:
```

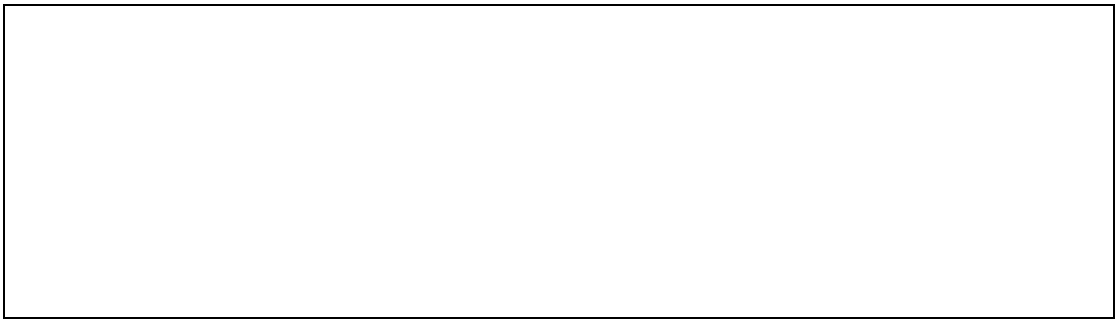
```
            report_calibration_status("IN_PROGRESS", skip_reason="no face
detected")
            draw_calibration(frame, calibrator)
            cv2.putText(frame, "No face detected — position yourself",
                        (10, 100), cv2.FONT_HERSHEY_SIMPLEX,
                        0.6, (0, 0, 255), 2)
            cv2.imshow(WINDOW_NAME, frame)
            key = cv2.waitKey(1) & 0xFF
            if should_quit_from_key(key):
                break
            continue
        else:
            cv2.putText(frame, "Calibration idle - tap Recalibrate in app",
                        (10, 100), cv2.FONT_HERSHEY_SIMPLEX,
                        0.62, (0, 255, 255), 2)
            cv2.putText(frame, "Press 'r' to recalibrate locally",
                        (10, 128), cv2.FONT_HERSHEY_SIMPLEX,
                        0.55, (255, 255, 255), 2)
            cv2.imshow(WINDOW_NAME, frame)
            key = cv2.waitKey(1) & 0xFF
            if should_quit_from_key(key):
                break
            elif key == ord('r'):
                print("[INFO] Recalibrating — look forward...")
                reset_calibration(source="manual")
            continue

    # ── FPS calculation ──────────────────────────────────
    cur_time = time.time()
    fps = 1.0 / (cur_time - prev_time + 1e-6)
    prev_time = cur_time

    # ── Build standardised output dictionary ─────────
    driver_state = {
        "state"        : smoothed_state,
        "stage"        : stage,
        "color"        : color,
        "ear"          : round(ear, 4),
        "ear_confidence": round(ear_confidence, 3),
        "ear_filtered" : round(ear_filtered, 4),
        "ear_norm"     : round(ear_norm, 3),
        "mar"          : round(mar, 4),
        "perclos"      : round(perclos, 2),
        "yaw"          : round(yaw, 2),
        "pitch"        : round(pitch, 4),
        "avg_blink_ms" : round(avg_blink_ms, 1),
        "head_nod_active": head_nod_active,
        "head_nod_drop"  : round(head_nod_drop, 3),
```

```
        "head_nod_sec"   : round(head_nod_elapsed, 2),
        "gaze_h"         : round(gaze_h, 3),
        "gaze_v"         : round(gaze_v, 3),
        "gaze_confidence": round(gaze_meta.get("gaze_confidence", 0.0), 3),
        "dominant_eye" : gaze_meta.get("dominant_eye", "none"),
        "attention"    : attention,
        "mirror_elapsed": round(attention_meta.get("mirror_elapsed", 0.0), 2),
        "quality_level": quality.level,
        "quality_score": round(quality.overall, 3),
        "reason"       : raw_reason,
        "calibrated"   : calibrator.calibrated,
        "face_visible" : face_visible,
        "fps"          : fps,
        "phone_detected" : phone_result["phone_detected"],
        "phone_conf"     : phone_result["confidence"],
        "phone_infer_ms" : phone_result["inference_ms"],
    }

    # ── Draw overlay ──────────────────────────────────
    draw_dashboard(frame, driver_state)

    # ── Send alert to server if state is non-normal ──
    if smoothed_state != "ALERT" and smoothed_state != "CALIBRATING":
        phone_conf = phone_result.get("confidence", 0.0) if
phone_result["phone_detected"] else None
        send_alert(
            smoothed_state,
            confidence=phone_conf,
            detection_class=smoothed_state
        )

    # Microsleep flash warning
    if smoothed_state == "MICROSLEEP":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 0, 255), 12)
        cv2.putText(frame, "!! WAKE UP !!", (w // 2 - 130, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.5, (0, 0, 255), 4)

    # Yawning alert
    if smoothed_state == "YAWNING":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 200, 255), 8)
        cv2.putText(frame, "YAWNING DETECTED", (w // 2 - 180, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 200, 255), 3)

    # Drowsy / Fatigued alert
    if smoothed_state == "DROWSY":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 100, 255), 10)
        cv2.putText(frame, "!! DROWSY !!", (w // 2 - 130, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 100, 255), 3)
```

```
    if smoothed_state == "FATIGUED":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 200, 255), 6)
        cv2.putText(frame, "FATIGUE DETECTED", (w // 2 - 180, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 200, 255), 3)

    # Looking down alert
    if smoothed_state == "LOOKING DOWN":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 140, 255), 8)
        cv2.putText(frame, "LOOK UP!", (w // 2 - 100, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 140, 255), 3)

    if smoothed_state == "PHONE IN LAP":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 140, 255), 8)
        cv2.putText(frame, "PHONE / LAP LOOK", (w // 2 - 170, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 140, 255), 3)

    # Distracted alert
    if smoothed_state == "DISTRACTED":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 0, 255), 8)
        cv2.putText(frame, "EYES ON ROAD!", (w // 2 - 150, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 0, 255), 3)

    # Phone detected alert
    if smoothed_state == "PHONE USE":
        cv2.rectangle(frame, (0, 0), (w, h), (0, 0, 255), 10)
        cv2.putText(frame, "!! PUT PHONE DOWN !!", (w // 2 - 220, h // 2),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 255), 4)

    # Draw phone bounding box if detected
    if phone_detector and phone_result["phone_detected"]:
        PhoneDetector.draw_detection(frame, phone_result)

    cv2.imshow(WINDOW_NAME, frame)

    # ── Key handling ──────────────────────────────────
    key = cv2.waitKey(1) & 0xFF
    if should_quit_from_key(key) or _stop_event.is_set():
        break
    elif key == ord('r'):
        # Recalibrate
        print("[INFO] Recalibrating — look forward...")
        reset_calibration(source="manual")

# Cleanup
if SERVER_ENABLED:
    try:
        _alert_queue.put_nowait(None)
```

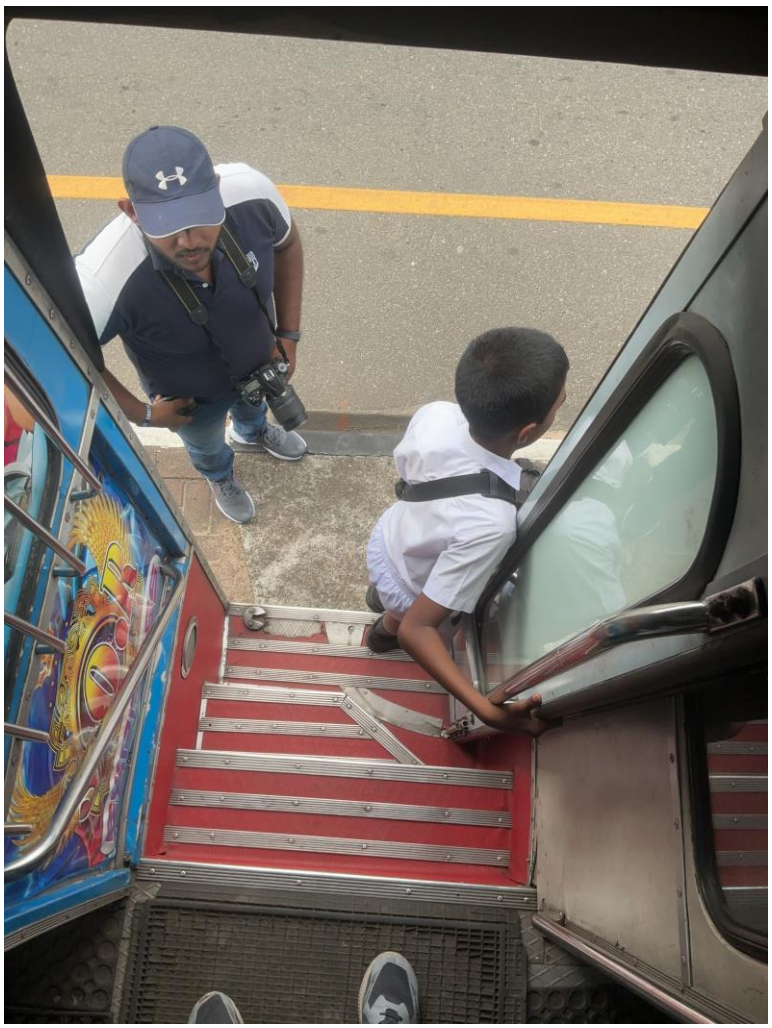
```
        except queue.Full:
            pass

    cap.release()
    cv2.destroyAllWindows()
    face_mesh.close()
    print("[INFO] Shutdown complete.")
    print(f"       Phone detection was: {'ENABLED' if phone_detector and
phone_detector.enabled else 'DISABLED'}")


if __name__ == "__main__":
    main()
```

Appendix Figure C.3: driver monitering main.py



Appendix Figure C.4: Driver monitoring code/runtime evidence 4

Appendix D: Additional Hardware and Testing Evidence

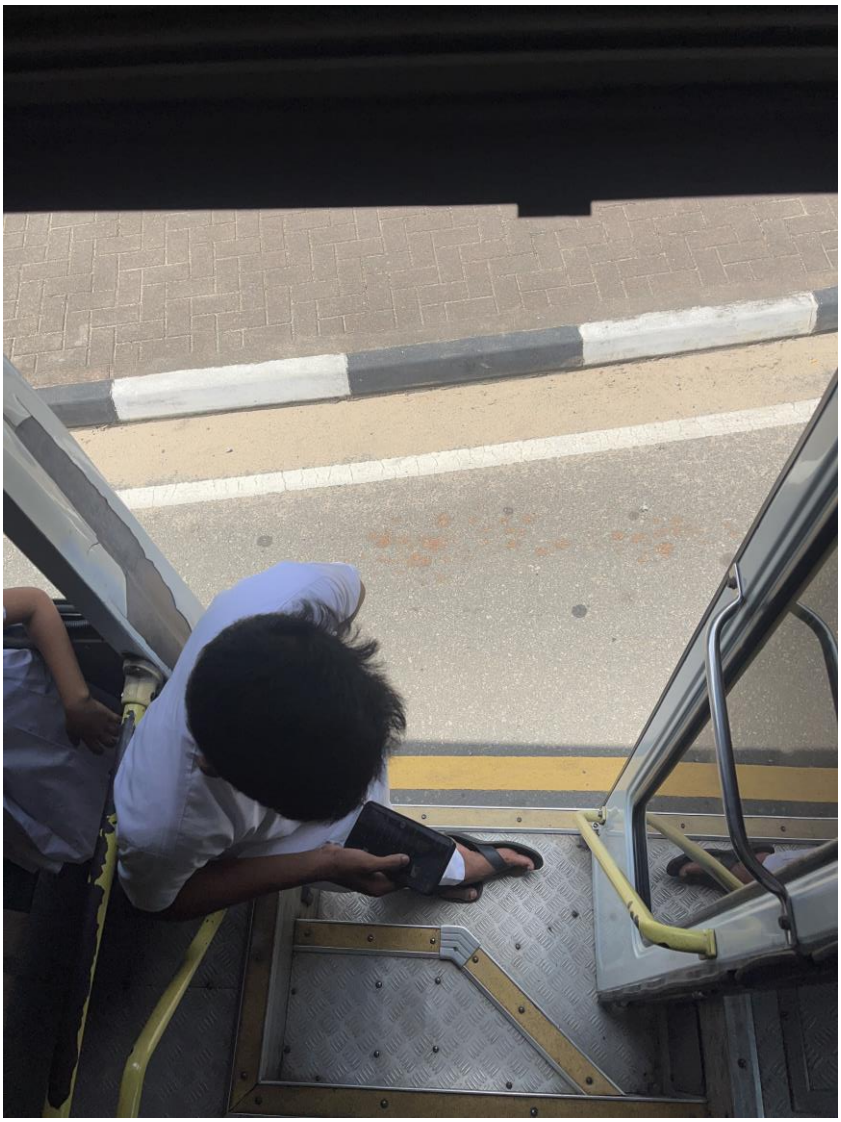


Appendix Figure D.1: Additional evidence placeholder 1

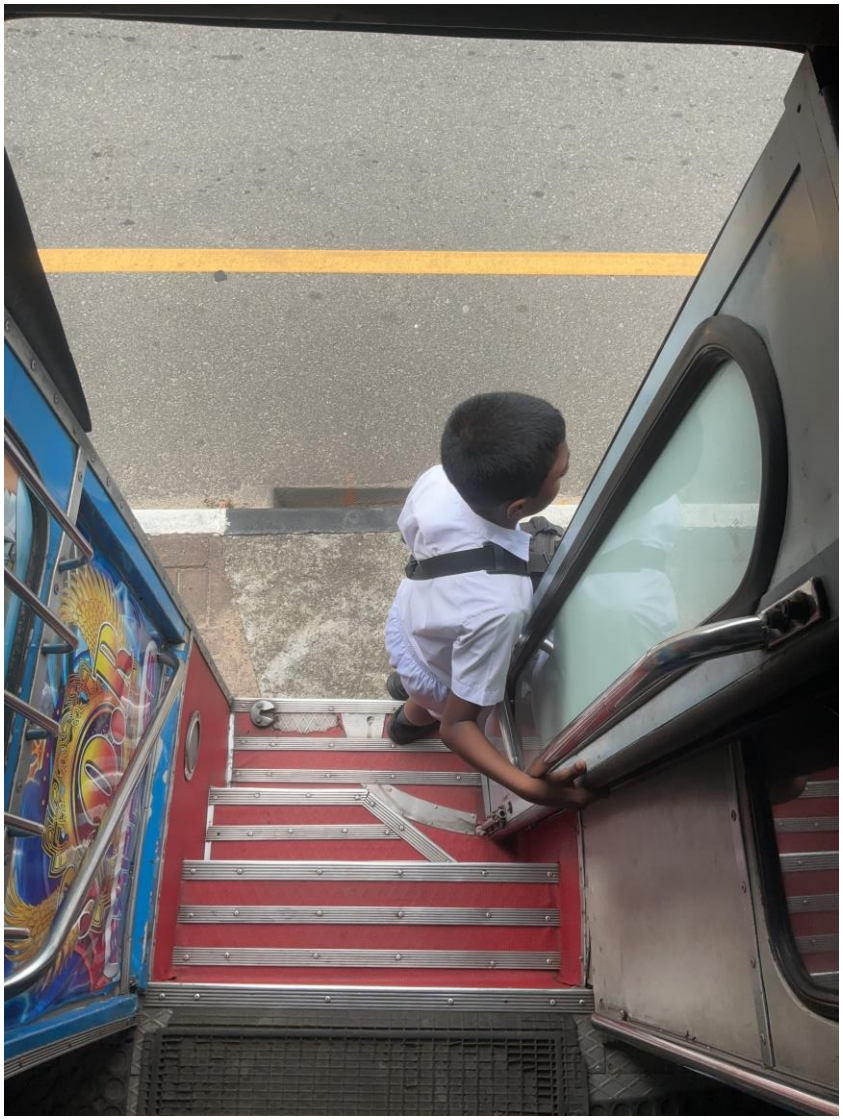
Appendix Figure D.2: Additional evidence placeholder 2

Appendix Figure D.3: Additional evidence placeholder 3

Appendix Figure D.4: Additional evidence placeholder 4



Appendix Figure D.5: Additional evidence placeholder 5



Appendix Figure D.6: Additional evidence placeholder 6