UNIVERSITY OF PISA

Department of Information Engineering

Master's Degree in Artificial Intelligence and Data Engineering

Industrial Applications

Real-time Drowsiness Detection via Distributed Embedded Systems

Development of a non-invasive computer vision-based system for monitoring driver fatigue

Work Group

Daniel Pipitone
Leonardo Cecchini

Project repository: https://github.com/da1pi2/drowsiness-detection

ACADEMIC YEAR 2025/2026

Contents

1 Abstract

This project presents the design and implementation of a driver drowsiness detection system aimed at reducing road accidents caused by fatigue. The system relies on computer vision techniques to analyze facial features in real time, enabling the detection of early signs of driver drowsiness.
Facial landmarks are extracted using MediaPipe [2], providing a robust and lightweight solution suitable for embedded platforms. Based on these landmarks, the Eye Aspect Ratio (EAR) and Mouth Aspect Ratio (MAR) are computed to identify prolonged eye closure and yawning events, which are key indicators of driver fatigue.
The proposed solution adopts a modular and distributed architecture. Visual data acquisition and real-time inference are performed on a Raspberry Pi, while more computationally demanding tasks can be offloaded to a PC server when network connectivity is available. A standalone configuration is also supported, allowing the system to operate entirely on the Raspberry Pi in offline scenarios.

2 Introduction

Driver drowsiness represents a major safety concern in transportation and industrial contexts, contributing significantly to road accidents and reduced operational reliability. Fatigue negatively affects reaction time, attention span, and decision-making abilities, making early detection of drowsiness a critical requirement, especially in long-duration or monotonous driving scenarios.

Recent advances in computer vision and embedded systems have enabled the development of non-invasive driver monitoring solutions based on visual behavior analysis. Compared to physiological approaches, vision-based systems are generally less intrusive, more cost-effective, and easier to integrate into existing infrastructures. However, achieving reliable detection in real-world conditions remains challenging due to variations in lighting, head pose, facial occlusions, and temporary loss of facial visibility.

This project presents a deployable driver drowsiness detection system targeting low-cost embedded hardware. The solution relies on MediaPipe-based facial landmark extraction [2] and on two standard visual indicators: Eye Aspect Ratio (EAR) for eye closure and Mouth Aspect Ratio (MAR) for yawning. The indicators are combined into a single drowsiness score to improve stability compared to single-metric approaches.

Classic ocular measures such as PERCLOS (percentage of eyelid closure over time) are often used in the literature as fatigue proxies [3].

The system is designed to run either fully on a Raspberry Pi or in a client–server setup where heavier processing can be offloaded to a PC when connectivity is available. Calibration, face-loss handling, alerting, and logging are described in detail in the following chapters.

The remainder of this document describes the system architecture, prototype implementation, and demonstration setup, followed by an evaluation of the system behavior under different operational conditions and a discussion of limitations and future improvements.

3 System Description

The proposed drowsiness detection system is designed as a robust and flexible architecture capable of operating both in a distributed client-server configuration and in a fully autonomous standalone mode. The primary design objective is to guarantee continuous driver monitoring under varying connectivity conditions, while preserving consistent detection logic and output metrics across all operating modes.

3.1 Overall Architecture

The system architecture is composed of two main components:

- an embedded edge device based on a Raspberry Pi, responsible for video acquisition and interaction with the driver;
- an optional PC server, which provides additional computational resources and an interactive monitoring interface.

The core pipeline extracts facial landmarks and computes standard visual indicators (EAR and MAR), which are then aggregated into a composite drowsiness score. The same set of outputs is produced in every operating mode, enabling consistent logging and evaluation.
Under normal operating conditions, the system runs in client-server mode: the Raspberry Pi streams video frames continuously to the PC server for processing. In parallel, it sends Raspberry-side telemetry (CPU, RAM, temperature, and camera FPS) once every `camera_fps` (i.e., about once per second), so that performance can be evaluated across operating modes.
When the network is unavailable, the system switches to a standalone mode and executes the full pipeline locally. In standalone deployments, a Streamlit dashboard can also run directly on the Raspberry Pi. Mode transitions are designed to be transparent to the user.

3.2 Operational Modes

3.2.1 Client-Server Mode

In client-server mode, the Raspberry Pi captures video frames from the connected camera and transmits them to the PC server over a network connection. The server performs facial landmark detection and computes the drowsiness-related metrics, including the Eye Aspect Ratio (EAR), the Mouth Aspect Ratio (MAR), and the resulting composite drowsiness score.
The computed metrics, along with the face detection status, are visualized in real time through a dashboard interface. The user interface displays numerical values, graphical overlays, and alert messages, including notifications related to loss of face detection. This configuration enables higher processing throughput and richer visualization capabilities by leveraging the computational power of the PC.

3.2.2 Standalone Mode

When a connection to the server cannot be established or is lost during operation, the system switches to standalone mode. In this configuration, the Raspberry Pi executes the entire analysis pipeline locally.
The standalone mode includes an initial calibration phase, during which a personalized baseline is estimated to adapt the eye-related detection threshold to the specific user. The calibrated parameter is stored in a shared configuration file and reused during subsequent operation. After calibration, the system

performs complete drowsiness detection locally, computing the same set of metrics as in client-server mode, including the composite drowsiness score.
In this mode, the Raspberry Pi also handles face loss detection and generates local alerts when facial landmarks cannot be reliably tracked. Due to hardware limitations, the achievable frame rate is lower than in the distributed configuration; however, the system retains full functional autonomy and continues to log events for post-analysis.

3.3 Automatic Fallback Mechanism

The system continuously monitors network connectivity to determine the appropriate operating mode. If communication with the PC server fails beyond a predefined threshold, the system automatically activates standalone mode. When the network connection is restored, video streaming to the server resumes seamlessly.
Importantly, the system continues to use the user-calibrated detection parameters stored in the shared configuration file after reconnection, ensuring continuity and consistency in detection behavior across mode transitions.

3.4 Configuration and Design Choices

All operational parameters, including default detection thresholds, temporal constraints, camera settings, and connection policies, are centralized in a shared configuration module. While default values are provided to ensure out-of-the-box functionality, user-specific calibration results overwrite the default eye-related threshold through a persistent configuration file.

3.4.1 Default Parameters

Table 3.1 reports the main default configuration parameters.

Parameter	Default	Meaning
PC_SERVER_IP	192.168.1.219	PC server IP address (client–server mode)
PC_SERVER_PORT	5555	PC server port
CONNECTION_TIMEOUT	10	Connection timeout (s)
RECONNECT_DELAY	5	Delay before retrying connection (s)
EAR_THRESHOLD	0.25	Default EAR threshold (used only if calibration is unavailable)
EAR_CONSEC_FRAMES	10	Consecutive frames below threshold to trigger an alert
MAR_THRESHOLD	0.6	Default MAR threshold for yawning
YAWN_CONSEC_FRAMES	8	Consecutive frames above threshold to detect a yawn
CAMERA_WIDTH	320	Camera width (px)
CAMERA_HEIGHT	240	Camera height (px)
CAMERA_FPS	20	Camera FPS target (also telemetry rate basis)
JPEG_QUALITY	70	JPEG compression quality
SHOW_LANDMARKS	True	Draw eye/mouth landmarks
SHOW_EAR_MAR	True	Display EAR/MAR numeric values
DISPLAY_ENABLED	False	Enable on-device display (False for lite/headless)
LOG_EVENTS	False	Enable event logging
LOG_FILE	drowsiness_log.txt	Log file name

Table 3.1: Default configuration parameters (standalone MediaPipe-based version).

4 Algorithm Description

4.1 Facial Landmark Extraction

For each input video frame, facial landmarks are extracted using the MediaPipe framework [2], which provides a dense facial mesh composed of 468 landmarks. This representation allows accurate localization of key facial regions, such as the eyes and mouth, which are required for the subsequent computation of drowsiness-related features.

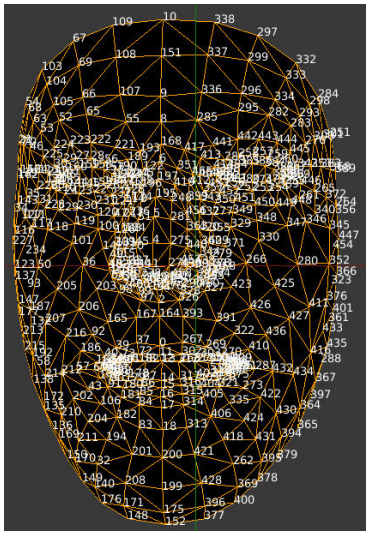


Figure 4.1: Example of MediaPipe Face Mesh landmark detection.

When a face cannot be reliably detected in a frame, a dedicated *face lost* counter is incremented. This mechanism introduces temporal tolerance to short detection failures caused by head movements, partial occlusions, or illumination changes. Only when the absence of facial landmarks persists beyond a short temporal window (approximately one second) is a face loss condition raised, avoiding rapid state oscillations and improving robustness under real-world conditions.

4.2 Eye Aspect Ratio (EAR)

Eye openness is quantified using the Eye Aspect Ratio (EAR), defined as the ratio between vertical and horizontal distances of predefined eye landmarks [1]:

Figure 4.2: Landmark points used to compute EAR (schematic).

$$\text{EAR} = \frac{\|p_1 - p_5\| + \|p_2 - p_4\|}{2\|p_0 - p_3\|}$$

Figure 4.3: Example of the specific eye landmark indices used in MediaPipe Face Mesh.

Lower EAR values correspond to eye closure. Unlike approaches that employ a fixed global threshold, the EAR threshold in this system is dynamically determined. A personalized value is obtained during an initial calibration phase and loaded from a shared configuration file at runtime. If calibration data are not available, a default threshold specified in the system configuration is used as a fallback.
To reduce sensitivity to normal blinking, EAR-based drowsiness detection relies on consecutive-frame validation: a drowsiness condition is triggered only when the EAR remains below the personalized threshold for a predefined number of consecutive frames.

4.3 Mouth Aspect Ratio (MAR)

Yawning behavior is detected using the Mouth Aspect Ratio (MAR), which measures mouth openness through a geometric relationship between selected mouth landmarks. The formulation is derived in the same spirit as EAR [1], using vertical and horizontal mouth distances.

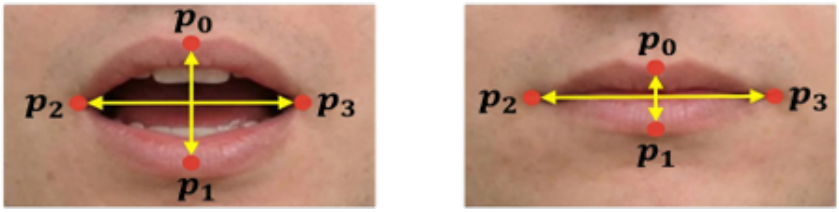


Figure 4.4: Landmark points used to compute MAR (schematic).

$$\text{MAR} = \frac{\|p_0 - p_1\|}{\|p_2 - p_3\|}$$

Sustained MAR values above a predefined threshold indicate yawning events. As with EAR, temporal validation based on consecutive frames is applied to avoid false positives caused by brief or unrelated facial expressions.

4.4 Temporal Thresholding

Temporal thresholding is applied independently to the eye and mouth indicators using dedicated frame counters. Separate counters are maintained for EAR-based and MAR-based events, allowing the system to distinguish between prolonged eye closure and yawning behaviors.
The face loss condition is handled by an additional counter that is logically independent from the eye and mouth counters. Importantly, temporary loss of facial landmarks does not immediately reset the eye and mouth states; only when the face loss counter exceeds its temporal threshold is a face loss alert generated. This design choice prevents unstable behavior and improves continuity in the detection logic.

4.5 Unified Detection Logic

The high-level detection logic, shared across all operational modes, can be summarized as follows:

```
For each frame:
    Detect face and landmarks
    If landmarks are valid:
        face_detected = True
        Compute EAR and MAR
        Update EAR and MAR counters
        If EAR below personalized threshold for N frames:
            Trigger drowsiness event
        If MAR above threshold for M frames:
            Trigger yawning event
    Else:
        face_detected = False
        Update face lost counte
        If face lost persists beyond threshold
            Trigger "face lost" alert
```

Figure 4.5: High-level detection logic (pseudocode).

4.6 Output and Feedback

The algorithm produces a uniform set of outputs in both client–server and standalone modes. The Eye Aspect Ratio (EAR), Mouth Aspect Ratio (MAR), composite drowsiness score, and face detection status are continuously computed and made available to the visualization and logging subsystems.
In client–server mode, these metrics are displayed in real time through an annotated video stream and dashboard interface, including explicit alerts for drowsiness, yawning, and face loss conditions. In standalone mode, the same metrics and events are recorded locally with associated timestamps. Logged drowsiness and yawning events include the corresponding composite score, enabling post hoc analysis and performance evaluation.

5 Prototype and Demo Setup

5.1 Hardware Description and Setup

The prototype is based on a low-cost edge device designed to operate in real-world scenarios such as in-vehicle monitoring. The core hardware component is a Raspberry Pi equipped with a camera module for real-time video acquisition.

5.1.1 Raspberry Pi and Camera Module

The system uses a Raspberry Pi 3 Model B+ as the edge device responsible for video capture and, when required, local processing. The Raspberry Pi is connected to a camera module through the CSI interface, allowing low-latency access to video frames with minimal overhead.
At system startup, the Raspberry Pi performs an automatic calibration phase lasting approximately 10 seconds. During this phase, baseline eye openness statistics are collected in order to personalize the Eye Aspect Ratio (EAR) threshold for the specific user. The calibrated threshold is stored in a shared JSON configuration file, which is subsequently used by both standalone and client–server execution modes.
Only high-level hardware characteristics are reported in this document. Detailed specifications and datasheets are available at the official Raspberry Pi documentation pages:

- Raspberry Pi 3 Model B+: `https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/`
- Camera Module documentation: `https://www.raspberrypi.com/documentation/accessories/camera.html`

5.1.2 Raspberry Pi Software Environment

For the demo and tests, the Raspberry Pi runs a minimal Linux distribution without a desktop environment:

- Release Date: 24 Nov 2025
- Distribution: Debian GNU/Linux 12 (*bookworm*)
- Architecture: `arm64`
- Kernel: Linux `6.12.62+rpt-rpi-v8`
- OS image: Raspberry Pi OS (Legacy) Lite (64-bit)

5.1.3 PC Workstation

A PC workstation is optionally used as a server component in the client–server configuration. In this mode, the Raspberry Pi acts as a video streaming client, while the PC performs computationally intensive tasks such as MediaPipe-based facial landmark extraction, composite drowsiness score computation, and visualization through a dashboard interface.
The use of a PC server is particularly relevant during development, testing, and demonstration phases, as it allows higher frame rates, real-time visualization, and easier inspection of internal metrics and system states.

5.2 Prototype Configuration

The system supports three operational behaviors:

- **Client–Server Mode**: the Raspberry Pi streams video frames continuously to the PC over the network. In addition, it sends Raspberry-side telemetry (CPU, RAM, temperature, and camera FPS) once every `camera_fps` (i.e., roughly once per second). The PC performs facial landmark detection, EAR/MAR computation, composite drowsiness score evaluation, and visualization.
- **Standalone Mode**: when no network connection is available, the Raspberry Pi performs the entire processing pipeline locally, including facial landmark extraction and drowsiness detection. A Streamlit dashboard can optionally run directly on the Raspberry Pi (client version with local dashboard).
- **Hybrid / Fallback Mode**: the system automatically switches between client–server and standalone execution depending on network availability. When connectivity is restored, streaming to the PC resumes without altering the calibrated EAR threshold, which remains loaded from the shared configuration file.

This design guarantees continuous operation and consistent detection behavior even under intermittent network conditions.

5.3 Allocation of Software Modules

The software architecture is modular and explicitly designed to distribute responsibilities between the edge device and the server when available.

Module	**Raspberry Pi**	**PC Server**
Video acquisition	✓	–
Frame transmission (continuous)	✓	–
Raspberry telemetry sampling (CPU/RAM/temp/FPS)	✓	–
Telemetry transmission (every `camera_fps`)	✓	–
MediaPipe Face Mesh	Standalone mode	✓
EAR/MAR computation	Standalone mode	✓
Composite drowsiness score	Standalone mode	✓
Face-lost alert handling	Standalone mode	✓
Visualization dashboard (Streamlit)	Optional	✓
Event logging	✓	✓
Configuration management (calibrated EAR threshold)	✓	✓

Table 5.1: Allocation of software modules in the system architecture.

The same detection logic and configuration files are shared across all execution modes. This ensures that EAR, MAR, composite score computation, and alert conditions remain consistent regardless of where the processing is performed.

6 Performance Evaluation and Optimization

This chapter reports Raspberry-side performance measurements (camera FPS, CPU load, RAM usage, and temperature) to compare how resource usage changes across operating modes and UI configurations. The performance metrics that we decided to use are the following:

- Camera FPS;
- % CPU load on the 4 cores;
- % RAM usage;
- CPU temperature in ºC.

6.0.1 FPS calculation

The FPS (Frames Per Second) value displayed on screen is calculated and updated following a periodic refresh policy rather than being computed on every single frame. Specifically, the FPS update is triggered when `frame_count % config.CAMERA_FPS == 0`, meaning that the calculation occurs approximately once per second (assuming the camera operates at its configured frame rate). This approach is implemented to balance accuracy with computational efficiency: calculating FPS on every frame would introduce unnecessary overhead and could lead to rapidly fluctuating values that are difficult to interpret. By updating the metric periodically at 1-second intervals, the system provides a stable measurement while minimizing the computational cost associated with frequent time measurements and divisions.

6.1 Libraries used

For system resource monitoring, two main libraries are used: **psutil** and **gpiozero**.

6.1.1 psutil

`psutil` (Python System and Process Utilities) is a cross-platform library that provides an interface for retrieving information about running processes and system resource usage. In the code, it is used to monitor:

- **CPU Usage**: The `psutil.cpu_percent(percpu=True)` function returns a list containing the usage percentage for each CPU core. The `percpu=True` parameter specifies that separate values should be obtained for each core, rather than a single average value. The obtained values are then summed using `sum(cpu_usage)` to calculate the total CPU load across all cores.
- **RAM Usage**: The `psutil.virtual_memory().percent` function returns the percentage of RAM currently in use on the system. This method accesses the `virtual_memory()` object, which contains various information about system memory, and extracts the percent attribute representing the memory usage percentage.

6.1.2 gpiozero

`gpiozero` is a library specifically designed for Raspberry Pi that simplifies access to GPIO (General Purpose Input/Output) pins and various hardware components. In this code, it is used to monitor the CPU temperature:

- **CPU Temperature**: The `CPUTemperature()` class provides direct access to the Raspberry Pi's internal temperature sensor. The `.temperature` attribute returns the current CPU temperature in degrees Celsius. The code uses a conditional expression `(if HAS_GPIOZERO else 0.0)` to handle cases where the library is not available (for example, when running on non-Raspberry Pi systems), defaulting to a value of 0.0 in such cases.

6.2 Experimental setup

The experimental tests were conducted on a Raspberry Pi 3 Model B+ using MediaPipe for face detection and drowsiness monitoring. Four different configurations were evaluated to assess the impact of computational load distribution on system performance. Each test recorded the key metrics shown before over a 5-minute period.

6.2.1 Test 1: Raspberry Pi with Dashboard - Standalone Version

In this configuration, the Raspberry Pi handles both face detection processing and the Streamlit dashboard server locally. This represents the most computationally intensive setup as all operations are performed on the single-board computer.

Time	Status	Mode	FPS	CPU (%)	RAM (%)	Temp (°C)
15:26	OK	STNDAL	3.17	174.90	37.53	47.03
15:27	OK	STNDAL	4.95	163.82	38.39	47.43
15:28	OK	STNDAL	4.90	157.56	40.21	48.12
15:29	OK	STNDAL	5.36	158.72	41.82	49.02
15:30	OK	STNDAL	5.82	171.91	43.54	49.44

Table 6.1: Raspberry with Dashboard - Standalone version

The results show highly variable performance with FPS ranging from 3.17 to 5.82, indicating significant processing limitations. The CPU usage is extremely high, with values between 157.56% and 174.90% (summed across all cores), consistently exceeding 150%. This suggests that all available CPU cores are heavily utilized and the system is under severe computational stress. RAM consumption gradually increases from 37.53% to 43.54% over the 5-minute period, while the CPU temperature rises from 47.03°C to 49.44°C.

Analysis
The standalone configuration demonstrates that running both detection and dashboard visualization simultaneously on the Raspberry Pi creates a significant bottleneck. The low FPS values (averaging around 5 FPS) indicate that the system struggles to maintain real-time performance, which could compromise the effectiveness of drowsiness detection in practical applications. The high CPU usage suggests the device is operating near its maximum capacity, leaving little room for additional processes or unexpected load spikes.

6.2.2 Test 2: Raspberry Pi with Dashboard - Client Version

In this setup, the Raspberry Pi performs face detection and updates the UI state every 10 seconds, while the actual Streamlit dashboard rendering is handled by a separate PC server. The Raspberry Pi only sends frames to the server.
Performance improves dramatically compared to Test 1, with FPS increasing from 13.77 to nearly 20 FPS, stabilizing around 19.97 FPS after the first minute. CPU usage drops significantly to a range of 76.79% to 99.30%, settling around 85-90% after initialization. RAM consumption remains stable at approximately

Time	Status	Mode	FPS	CPU (%)	RAM (%)	Temp (°C)
15:33	OK	CLIENT	13.77	76.79	37.10	48.01
15:34	OK	CLIENT	18.68	99.30	36.62	47.66
15:35	OK	CLIENT	19.96	89.17	36.47	47.76
15:36	OK	CLIENT	19.97	86.67	36.50	47.63
15:37	OK	CLIENT	19.97	85.00	36.50	47.28

Table 6.2: Raspberry with Dashboard - Client version

36.5%, a little lower than the standalone version. Temperature also stabilizes around 47-48°C, showing better thermal management.

Analysis
Offloading the dashboard rendering to an external server provides substantial performance benefits. The nearly 4x improvement in FPS (from 5 to 20 FPS) demonstrates that the Raspberry Pi can handle the detection workload efficiently when not burdened with visualization tasks. The reduced CPU usage indicates better resource allocation, while the stable RAM consumption suggests more efficient memory management. This configuration represents a good balance between local processing and distributed computing, making it suitable for real-world deployment scenarios.

6.2.3 Test 3: Raspberry Pi with No Dashboard - Standalone Version

This configuration eliminates the dashboard entirely, with the Raspberry Pi performing only face detection and displaying results via terminal print statements. This represents the minimal overhead scenario for standalone operation.

Time	Status	Mode	FPS	CPU (%)	RAM (%)	Temp (°C)
15:45	OK	STNDAL	5.98	106.02	35.03	46.35
15:46	OK	STNDAL	7.80	110.23	34.59	47.75
15:47	OK	STNDAL	10.12	111.50	34.60	48.17
15:48	OK	STNDAL	11.06	111.26	34.60	48.60
15:49	OK	STNDAL	11.56	110.89	34.60	49.12

Table 6.3: Raspberry with No Dashboard - Standalone version

Results show improved performance compared to Test 1, with FPS starting at 5.98 and gradually increasing to 11.56 over the test period. CPU usage ranges from 106.02% to 111.50%, significantly lower than the dashboard-enabled standalone version but still indicating heavy utilization. RAM consumption remains remarkably stable at around 34.6%, the lowest among all tested configurations. Temperature increases gradually from 46.35°C to 49.12°C.

Analysis
Removing the dashboard overhead doubles the FPS performance compared to Test 1, highlighting the computational cost of real-time visualization on resource-constrained hardware. However, the performance still lags behind the client-server configuration (Test 2), suggesting that even terminal output introduces some overhead. The stable, low RAM usage indicates that the detection pipeline itself is memory-efficient. While this configuration offers better performance than the full standalone version, the 11 FPS average may still be insufficient for applications requiring higher frame rates.

6.2.4 Test 4: Raspberry Pi with No Dashboard - Client Processing

In this final configuration, the Raspberry Pi acts purely as a camera client, capturing frames and sending them to the PC server, which performs all detection processing and maintains the Streamlit dashboard.

Time	Status	Mode	FPS	CPU (%)	RAM (%)	Temp (°C)
16:00	OK	CLIENT	18.09	81.70	34.18	45.41
16:01	OK	CLIENT	19.98	87.50	33.88	46.39
16:02	OK	CLIENT	20.00	80.00	33.80	46.97
16:03	OK	CLIENT	20.00	81.67	33.80	47.25
16:04	OK	CLIENT	20.00	83.33	33.81	47.67

Table 6.4: Raspberry with No Dashboard - Client version (dashboard on the pc server)

This setup achieves the best performance across all metrics. FPS quickly stabilizes at 20 FPS (the FPS set in the configuration) after initialization. CPU usage on the Raspberry Pi is minimal, ranging from 80% to 87.50% and stabilizing around 81-83%. RAM consumption is the lowest observed at approximately 33.8%, and temperature remains stable at 46-47°C.

Analysis
This configuration demonstrates optimal resource utilization by leveraging the Raspberry Pi solely for its camera interface capabilities while delegating compute-intensive tasks to more powerful hardware. The consistent 20 FPS indicates the system operates at the correct frame rate without processing bottlenecks. The significantly reduced CPU usage and lowest RAM consumption confirm that frame capture and network transmission require minimal resources compared to detection processing. This architecture is ideal for distributed systems where multiple Raspberry Pi units could stream to a central processing server.

6.3 Conclusions

The experimental results clearly demonstrate the impact of computational load distribution on system performance. The Raspberry Pi, while capable of running MediaPipe face detection, shows significant limitations when handling both processing and visualization tasks simultaneously.

Key Findings:

- **Standalone with Dashboard** (Test 1): Severe performance bottleneck with 5 FPS and >150% CPU usage, unsuitable for real-time applications.
- **Client-Server with Dashboard** (Test 2): Optimal balance with 20 FPS and manageable 85% CPU usage, suitable for practical deployment.
- **Standalone without Dashboard** (Test 3): Improved but limited performance with 11 FPS and 110% CPU usage, demonstrating visualization overhead.
- **Pure Client Mode** (Test 4): Maximum performance with 20 FPS and minimal resource usage, ideal for distributed architectures.

The client-server architecture (Test 2) emerges as the recommended configuration for real-world drowsiness detection systems, offering the best compromise between performance, resource utilization, and functionality. It maintains near-optimal FPS while providing full dashboard capabilities, making it suitable for applications where both local processing and remote monitoring are required. For scenarios requiring multiple monitoring points, the pure client mode (Test 4) enables efficient scaling by centralizing processing resources while minimizing edge device requirements.

7 Structure of the Demo

This chapter summarizes the main steps required to execute the demo.

7.1 PC Setup

1. Create and activate a Python virtual environment.
2. Install the PC dependencies from `pc_dashboard/requirements_pc.txt`.
3. Identify the PC IP address (same LAN as the Raspberry Pi) and set it in the configuration as `PC_SERVER_IP`.

7.2 Raspberry Pi Setup

1. Ensure the camera is connected and working.
2. Create and activate a Python virtual environment on the Raspberry Pi.
3. Install the Raspberry dependencies from `raspberry/requirements_raspberry.txt`.

7.3 Run the Demo

1. Start the PC dashboard in server mode (Streamlit).
2. Start the Raspberry client (Streamlit):
   - **Client–Server**: video frames are streamed continuously to the PC; Raspberry telemetry (CPU/RAM/temperature/camera FPS) is sent once every `camera_fps` (roughly once per second).
   - **Standalone**: the full pipeline runs locally on the Raspberry Pi (a local Streamlit dashboard can be used).
   - **Hybrid**: the client attempts to connect to the server and automatically falls back to standalone if the connection is unavailable.
3. Run the initial calibration (performed automatically at startup if no calibration file is available) and keep the user in front of the camera during the calibration window.
4. Observe detection metrics (EAR, MAR, composite score) and Raspberry telemetry; verify alerts for eye closure, yawning, and face lost.

7.4 Dashboard

The dashboard is the same for both the standalone and client version, showing the frame (with EAR, MAR and on-screen alerts), previous detection, fps and performance statistics.
The alerts can be seen also on the frame, like in the following examples.

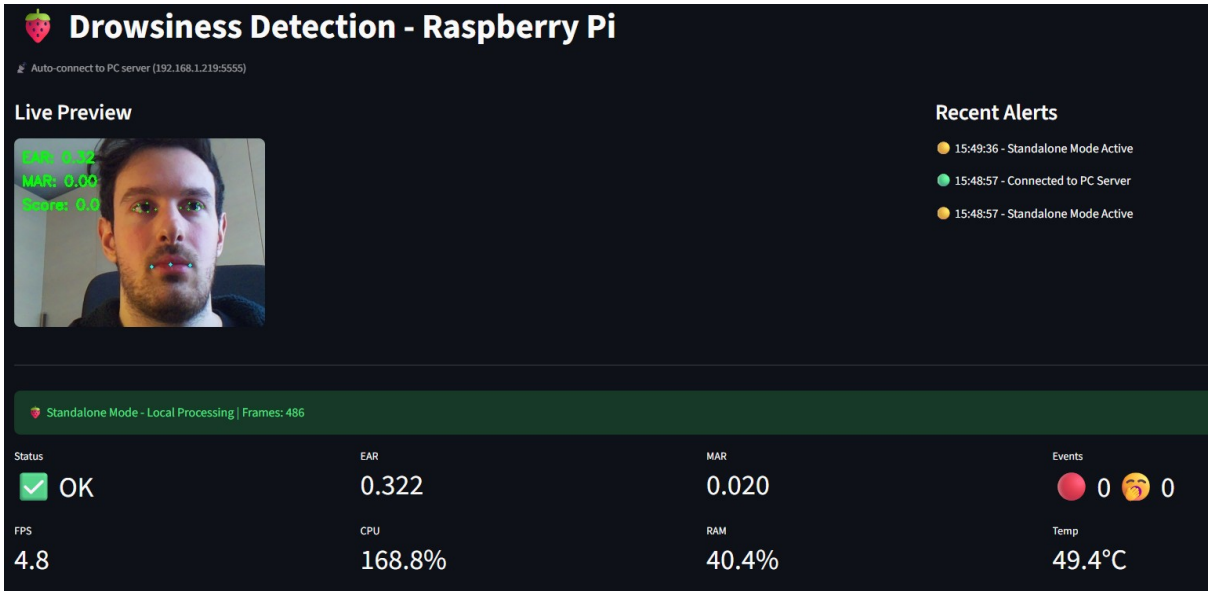
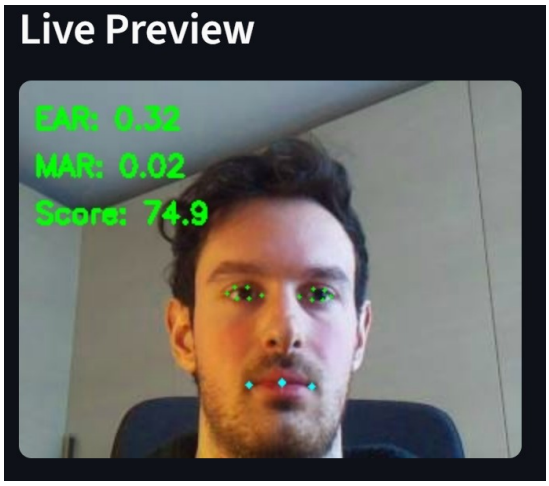
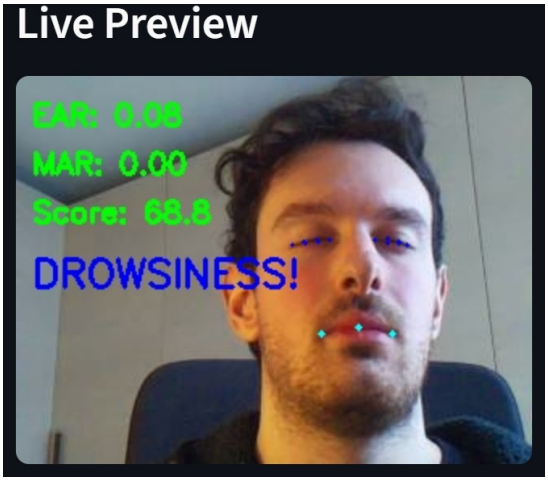


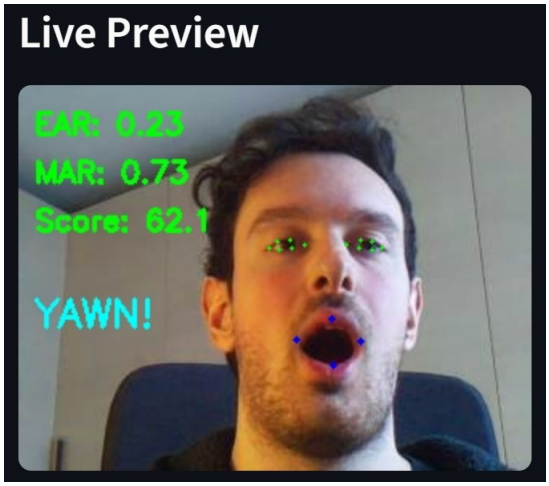
Figure 7.1: Dashboard of the Raspberry in standalone mode



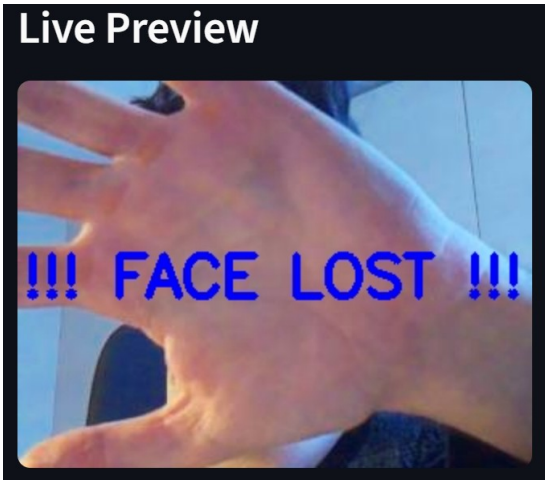
(a) Frame with no alerts



(b) Frame with drowsiness alert



(c) Frame with yawn alert



(d) Frame with ”No Face” detected alert

Figure 7.2: Overview of the detection system alerts.

8 Conclusions

This work presented a lightweight, deployable driver drowsiness detection system based on MediaPipe facial landmarks and simple geometric indicators (EAR and MAR). The solution supports both standalone execution on a Raspberry Pi and a client–server mode where video frames are streamed to a PC for heavier processing and richer visualization.

To evaluate deployability, we also integrated Raspberry-side performance telemetry (CPU, RAM, temperature, and camera FPS), enabling a direct comparison of resource usage across operating modes and UI configurations. Overall, the prototype demonstrates that robust, non-invasive monitoring can be achieved with low-cost hardware while keeping the software architecture modular and extensible.

Future improvements include a more systematic evaluation on public datasets, stronger robustness to occlusions and large head-pose variations, and adaptive policies that use the telemetry signals to automatically tune processing settings (e.g., resolution/FPS) or trigger a fallback strategy when resources become constrained.

References

[1] Tereza Soukupova and Jan Cech. *Eye Blink Detection Using Facial Landmarks*. 21st Computer Vision Winter Workshop (CVWW), Rimske Toplice, Slovenia, 2016.

[2] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. *MediaPipe: A Framework for Building Perception Pipelines*. arXiv:1906.08172, 2019.

[3] David F. Dinges, Malissa M. Mallis, Greg Maislin, and John W. Powell. *Evaluation of Techniques for Ocular Measurement as an Index of Fatigue and as the Basis for Alertness Management*. U.S. Department of Transportation, National Highway Traffic Safety Administration, Report DOT HS 808 762, April 1998.