

## Design and Simulation of a Cloud-Assisted Driver Fatigue and Gas Monitoring System Using Raspberry Pi and EdgeCloudSim

*“A real-time driver safety platform combining IoT sensors, Raspberry Pi edge computing, and cloud services to detect fatigue and hazardous gases, trigger immediate alerts, and support centralized monitoring and future analytics via simulation results.”*



Alejandro Mendoza Moreno  
In collaboration with *Men2zAI Lab*

## 1. Introduction

This project focuses on the development and evaluation of a cloud-integrated fatigue and gas monitoring system for drivers. Using a hybrid IoT and Edge-Cloud architecture, each vehicle is equipped with a Raspberry Pi Zero 2 W running various environmental and physiological sensors. These devices perform local analysis and trigger alerts in real-time, while also sending critical data to the cloud for remote access and long-term analysis.

To evaluate system performance, the EdgeCloudSim simulator was used to model different deployment scenarios, ranging from a single-vehicle setup to a 50-vehicle fleet. Simulation results were analyzed to compare latency, task distribution, and failure rates across edge and cloud configurations. The project highlights the potential of distributed computing in enhancing road safety through intelligent orchestration and system scalability.

## 2. Problem Analysis / Motivation

Driving for long hours without breaks, especially on repetitive or demanding routes, significantly increases the risk of fatigue. According to traffic safety studies, drowsy driving is responsible for thousands of preventable accidents every year. In addition, in-cabin exposure to invisible gases like CO and VOCs is rarely monitored in real time, even though they can severely affect driver alertness and health.

Existing vehicle systems focus mainly on visual fatigue detection or use expensive proprietary technology. Most do not provide continuous monitoring or remote alerts. I believe that by combining affordable sensors with edge processing and cloud-based storage and notification, we can create a more accessible and scalable solution.

That is the motivation behind my system: to build an architecture that detects fatigue and gas exposure in real time, processes it on the vehicle, and ensures backup and reporting through the cloud.

## 3. Project Objective

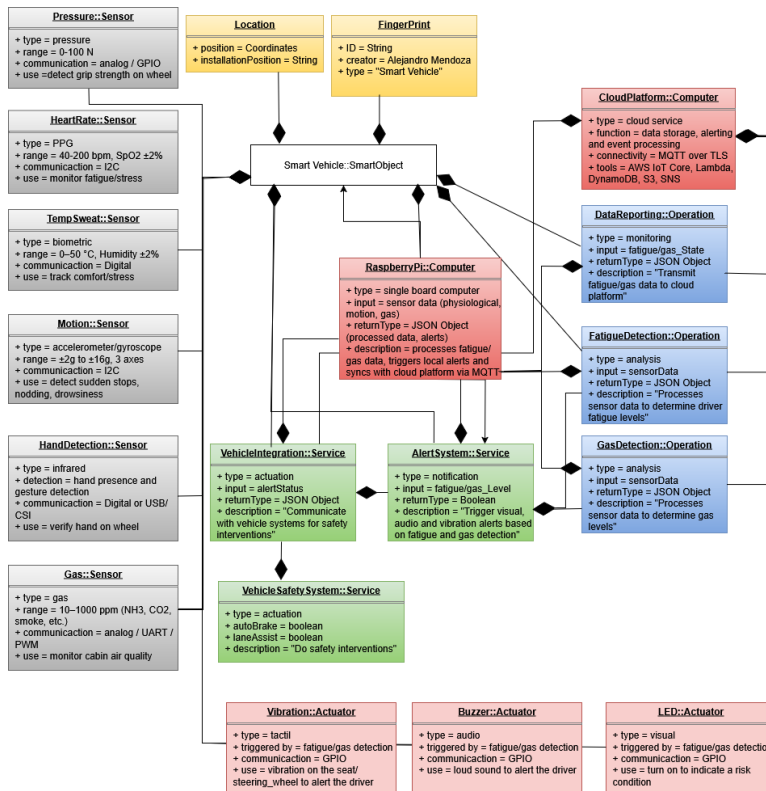
### 3.1 Main Goal:

- To design and simulate an edge-cloud system for driver fatigue and gas monitoring using EdgeCloudSim, based on real-world hardware and network configurations.

### 3.2 Sub-goals:

- To collect relevant physiological and environmental data using low-cost sensors.
- To process critical conditions locally using Raspberry Pi Zero 2 W devices acting as edge nodes.
- To define and configure a realistic edge-cloud simulation using EdgeCloudSim.
- To simulate and evaluate multiple orchestration strategies to guide system design decisions (e.g., ONLY\_EDGE, HYBRID, BEST\_FIT).
- To evaluate task distribution, latency, and failure rates under different architectures (SINGLE\_TIER vs. TWO\_TIER\_WITH\_EO).
- To analyze the feasibility of deploying such systems in real vehicle fleets.

#### 4. ACOSO Model and Service Model



Fatigue Safety: Service Model	
Service Name	<b>Fatigue Monitoring</b>
Service Description	Driver safety in vehicular operation
Service Category	Vehicle safety
Service Parameter	Response = 5s Accuracy = 95%
Service Input	Physiological data Driver behavior
Service Output	Fatigue Risk level
Service Precondition & Context	Driver is showing signs of fatigue
Service Effect & Context Effect	Driver is alerted and safety measures are activated
Service Provision Constrain	Driver is properly monitored by SO

Fatigue Detection: Service Profile	
Process ID	<b>Fatigue Detection</b>
Process Input	Physiological data, Driver behavior
Process Output	Driver fatigue level
Process Precondition	The driver is moving and the sensors are active
Process Effect	The level of exertion is determined, and the following operation are performed

Gas Detection: Service Profile	
Process ID	Gas Detection
Process Input	Sensor data (CO, VOCs levels)
Process Output	Gas level status (normal / warning/ danger)
Process Precondition	Sensors are active and vehicle is operating
Process Effect	If gas level exceeds threshold, an alert is triggered and the event is logged in the cloud

GasSafety: Service Model	
Service Name	Gas Monitoring
Service Description	Detect hazardous gas exposure inside the vehicle
Service Category	Environmental safety
Service Parameter	Response = 3s Accuracy = 92%
Service Input	Air quality sensor data (CO, VOCs)
Service Output	Gas Risk level
Service Precondition & Context Precondition	Presence of detectable gases in cabin
Service Effect & Context Effect	Driver is alerted, ventilation is activated
Service Provision Constrain	System must operate continuously while vehicle is running

Driver Alert	
Process ID	Driver Alert
Process Input	Fatigue level detected
Process Output	Notification alert
Process Precondition	Fatigue level has exceeded the limit threshold
Process Effect	The driver receive the alert and can react

The ACOSO model structures the intelligent driver monitoring system into Smart Object roles and services. The **Smart Vehicle** is the central Smart Object, integrating multiple sensors and actuators. Data processing and decision-making occur locally on the *Raspberry Pi*, which executes *FatigueDetection* and *GasDetection*, activates *AlertSystem*, and interfaces with *VehicleIntegration* and *VehicleSafetySystem* for in-vehicle interventions.

Critical events are forwarded to the *CloudPlatform* Smart Object, leveraging AWS services (IoT Core, Lambda, DynamoDB, S3, SNS) over MQTT/TLS. This ensures robust, scalable data backup, global accessibility, and advanced analytics.

## 4.1 Service Model

Service	Type	Input	Output	Description
FatigueDetection::Operation	Analysis	sensorData (physiological)	JSON (fatigueLevel)	Processes biometric and behavioral data to compute fatigue
GasDetection::Operation	Analysis	sensorData (gas)	JSON (gasLevel)	Processes air quality data to compute gas concentration
DataReporting::Operation	Monitoring	fatigue/gas state	JSON to Cloud	Sends processed data to cloud platform
AlertSystem::Service	Notification	fatigue/gas level	Boolean (alert)	Triggers visual, audio, and vibration alerts
VehicleIntegration::Service	Actuation	alertStatus	JSON	Interfaces with vehicle safety systems for interventions
VehicleSafetySystem::Service	Actuation	emergency interrupt	—	Executes safety actions (auto-brake, lane assist)

## 5. System Design

### 5.1 Components

**Sensors:** The vehicle is equipped with multiple low-cost sensors to monitor vital signs and environmental conditions. These include:

- ✓ Heart rate sensor
- ✓ Grip pressure sensor
- ✓ Infrared hand detection sensor
- ✓ Gas sensor
- ✓ Accelerometer
- ✓ Temperature and humidity sensor

**Edge Node:** A Raspberry Pi Zero 2 W serves as the edge device. It was chosen for its low power consumption, small form factor, and sufficient processing power to handle data preprocessing and simple classification.

**Actuators:** Visual, auditory, and tactile alerts (LEDs, buzzers, vibration motors) are activated locally in case of fatigue or gas detection.

### 5.2 Logical Architecture

1. The Raspberry Pi processes incoming data using either a lightweight ML model or predefined thresholds.
2. If a critical event is detected, immediate alerts are triggered.
3. Processed data and event logs are sent to a cloud backend for:
  - ✓ Historical analysis
  - ✓ Centralized monitoring
  - ✓ Backup alerts (e.g., via Telegram/SNS)
  - ✓ Visualization on dashboards

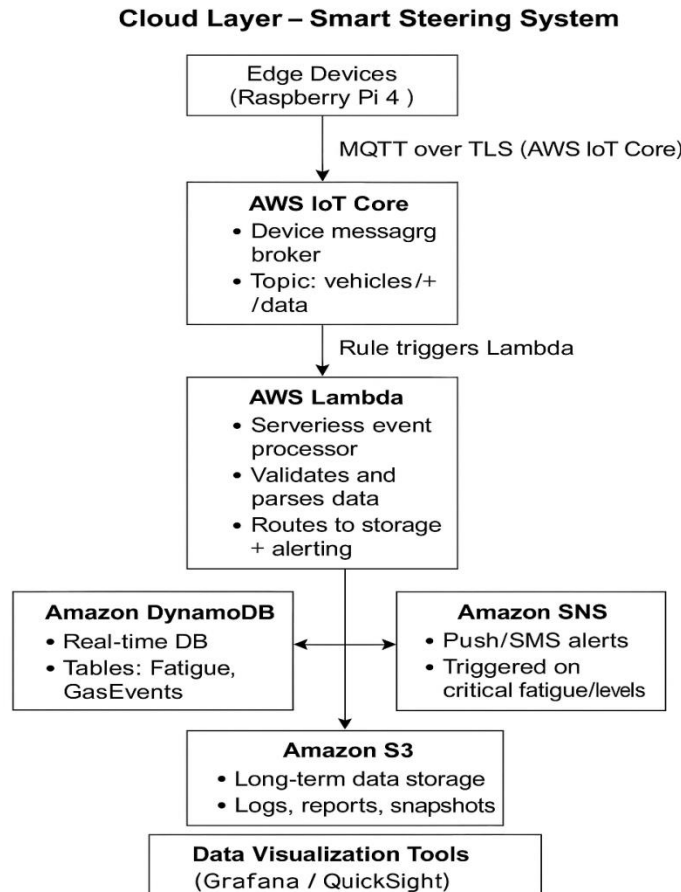
### 5.3 Communication layer

- The edge devices communicate with the cloud via MQTT over Wi-Fi.
- The simulation models both LAN and WAN delays realistically.

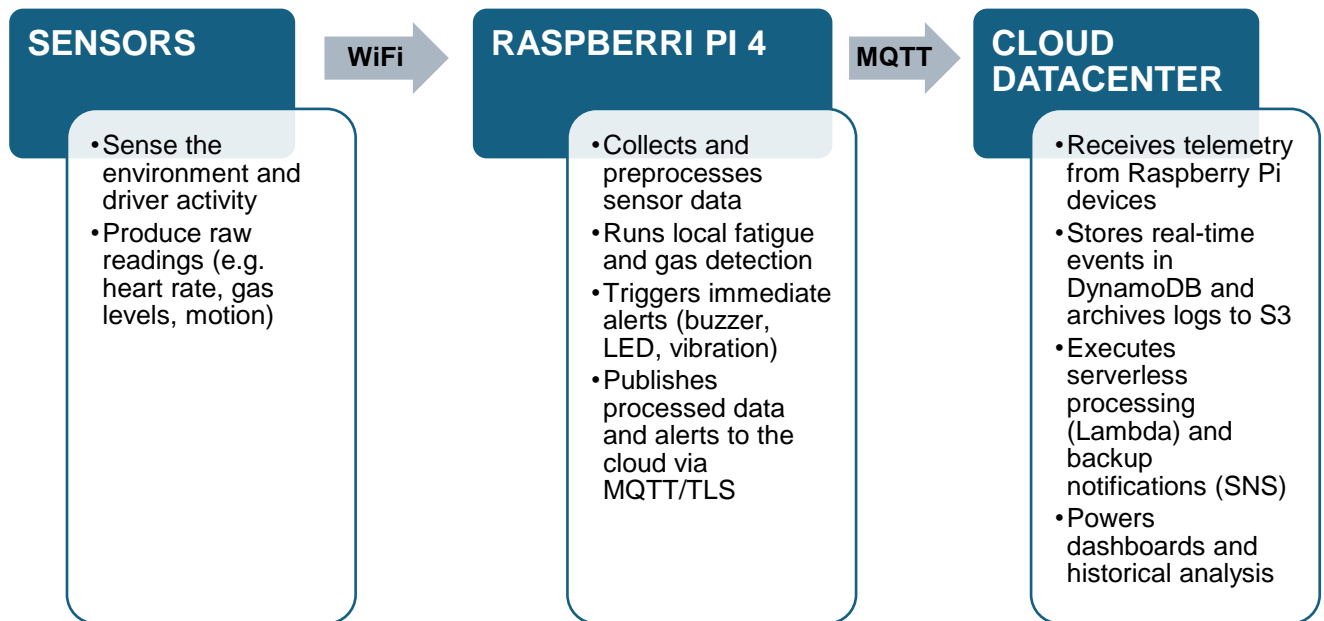
### 5.4 Simulated Cloud Component

- ❖ **AWS IoT Core** (simulated): Receives messages from edge nodes.
- ❖ **AWS Lambda** (simulated): Handles processing of incoming events.
- ❖ **DynamoDB / S3** (simulated): Store event logs and sensor data.
- ❖ **SNS** (simulated): Sends secondary alerts if needed.

This design balances local real-time responsiveness with the reliability and scalability of cloud computing.



## 5.5 Real Time Dataflow



1. Sensors collect real-time data (driver vitals and gas presence).
  2. Raspberry Pi Zero 2 W processes fatigue events locally to trigger alerts.
  3. Gas data is offloaded to the cloud for further analysis and storage.
  4. Alerts are sent via Telegram bot and backup SMS through AWS SNS.
  5. Dashboard visualizes historical trends using DynamoDB and S3.
- ❖ Simulation with EdgeCloudSim evaluates edge vs cloud performance under different scenarios.

## 6. Implementation

The system was implemented using **EdgeCloudSim**, a Java-based simulator for modeling edge-cloud computing architectures in IoT environments.

### 6.1 Edge Devices (Raspberry Pi Zero 2 W)

Each simulated vehicle contains a Raspberry Pi Zero 2 W configured as an independent edge node. The Pi is modeled with:

- 1 core
- 1000 MIPS
- 512 MB RAM
- 16 GB storage

These specifications reflect real-world hardware limitations, ensuring realistic simulation behavior.

## 6.2 Application Modeling

Two applications were defined in “applications.xml”:

**FATIGUE\_MONITORING**: Requires frequent data capture and low delay.

**GAS\_MONITORING**: Lower frequency, but higher sensitivity to environmental changes.

Each task includes parameters like:

- Task length
- VM utilization
- Poisson inter-arrival time
- Data upload/download sizes
- Delay sensitivity

Tasks are offloaded either to the edge or to the cloud based on orchestration logic.

## 6.3 Cloud Setup

The cloud is modeled as a single datacenter with:

- 1 host
- 4 VMs
- Each VM: 4 cores, 10,000 MIPS, 32 GB RAM

This setup reflects a realistic AWS-like configuration.

## 6.4 Orchestration Policy

The **BEST\_FIT** orchestration policy was selected for simulation. This policy dynamically chooses the best processing node (edge or cloud) based on task delay sensitivity and VM availability.

## 6.5 Scenarios

Two main scenarios were tested:

- **SINGLE\_TIER**: Only edge processing.
- **TWO\_TIER\_WITH\_EO**: Distributed processing using both edge and cloud, with the BasicEdgeOrchestrator enabled for smart task assignment based on delay and location.

This dual-scenario simulation allowed for a comprehensive performance evaluation.

## 6.6 Network Parameters

The simulation includes realistic network conditions:

- WLAN Bandwidth: 200 Mbps
- WAN Bandwidth: 15 Mbps
- WAN Delay: 0.1 s
- LAN Delay: 0.005 s

All configurations were set via XML files and *default\_config.properties* in EdgeCloudSim.

## 7. Simulation Setup

To evaluate system performance under realistic conditions, several simulations were carried out using EdgeCloudSim. The simulation configuration was adjusted to reflect the physical characteristics of the Raspberry Pi Zero 2 W and a cloud-based architecture similar to AWS.

### 7.1 Configuration Parameters

- Simulation Time: 120 minutes
- Number of Mobile Devices: 50
- Poisson Inter-arrival Rate: 10 seconds
- Edge Devices: Raspberry Pi Zero 2 W (1 core, 1000 MIPS, 512MB RAM)
- Cloud Configuration: 1 datacenter, 1 host, 4 VMs (4 cores, 10000 MIPS each)
- Bandwidths: WLAN = 200 Mbps, WAN = 15 Mbps
- Delays: WAN = 0.1 s, LAN = 0.005 s
- Orchestration Policy: BEST\_FIT
- Scenarios: SINGLE\_TIER and TWO\_TIER\_WITH\_EO

### 7.2 Step-by-step Testing Approach

To validate scalability and fault tolerance, simulations were conducted progressively:

#### 1. Single Edge Device Test (1 Vehicle):

A minimal simulation to observe the processing behavior and validate correctness with a single Raspberry Pi device.

#### 2. Fleet Simulation (50 Vehicles):

A full-scale simulation involving 50 edge nodes (vehicles) operating concurrently, each running the defined applications and communicating with the cloud.

This approach allowed for comparative performance analysis in both isolated and large-scale scenarios.

### 7.3 Simulation Scenarios and Policies

Two simulation scenarios were selected to represent different system architectures:

#### - SINGLE\_TIER:

Tasks are executed only on edge devices. This scenario represents a decentralized model without cloud involvement.

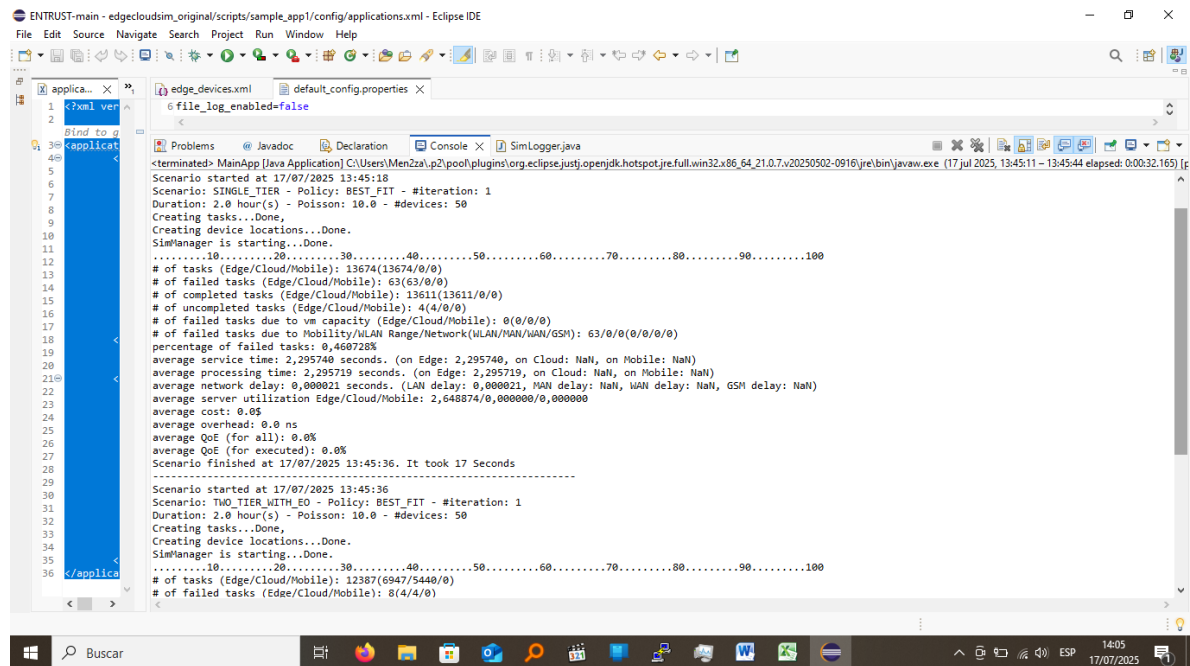
#### - TWO\_TIER\_WITH\_EO:

Combines edge and cloud layers with an edge orchestrator (EO) that dynamically decides where tasks should be processed based on latency, resource availability, and delay sensitivity.

To handle orchestration, the *BEST\_FIT* policy was used in all simulations. This policy assigns each task to the most suitable resource (edge or cloud) to optimize performance and resource usage.



## 8. Simulation Results



```
<terminated> MainApp [Java Application] C:\Users\MenZa\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.v20230502-0916\jre\bin\javaw.exe (17 Jul 2025, 13:45:11 - 13:45:44 elapsed: 0:00:32.165) [c
File Edit Source Navigate Search Project Run Window Help
edge_devices.xml default_config.properties
file_log_enabled=false
Problems Javadoc Declaration Console SimLogger.java
Scenario started at 17/07/2025 13:45:18
Scenario: SINGLE_TIER - Policy: BEST_FIT - #iteration: 1
Duration: 2.0 hour(s) - Poisson: 10.0 - #devices: 50
Creating tasks...Done.
Creating device locations...Done.
SimManager is starting...Done.
.....10.....20.....30.....40.....50.....60.....70.....80.....90.....100
# of tasks (Edge/Cloud/Mobile): 13674(13674/0/0)
# of failed tasks (Edge/Cloud/Mobile): 63(63/0/0)
# of completed tasks (Edge/Cloud/Mobile): 13611(13611/0/0)
# of uncompleted tasks (Edge/Cloud/Mobile): 4(4/0/0)
# of failed tasks due to vm capacity (Edge/Cloud/Mobile): 0(0/0/0)
# of failed tasks due to Mobility/WLAN Range/Network(N/LAN/M/WAN/GSM): 63/0/0(0/0/0/0)
percentage of failed tasks: 0,460728%
average service time: 2,295740 seconds. (on Edge: 2,295740, on Cloud: NaN, on Mobile: NaN)
average processing time: 2,295719 seconds. (on Edge: 2,295719, on Cloud: NaN, on Mobile: NaN)
average network delay: 0,000021 seconds. (LAN delay: 0,000021, WAN delay: NaN, GSM delay: NaN)
average server utilization Edge/Cloud/Mobile: 2,648874/0,000000/0,000000
average cost: 0.0$
average overhead: 0.0 ns
average QoE (for all): 0.0%
average QoE (for executed): 0.0%
Scenario finished at 17/07/2025 13:45:36. It took 17 Seconds
-----
Scenario started at 17/07/2025 13:45:36
Scenario: TWO_TIER_WITH_EO - Policy: BEST_FIT - #iteration: 1
Duration: 2.0 hour(s) - Poisson: 10.0 - #devices: 50
Creating tasks...Done.
Creating device locations...Done.
SimManager is starting...Done.
.....10.....20.....30.....40.....50.....60.....70.....80.....90.....100
# of tasks (Edge/Cloud/Mobile): 12187(6947/5440/0)
# of failed tasks (Edge/Cloud/Mobile): 8(4/4/0)
```

Scenario	Devices	Total Tasks	Failures	Failure Rate	Avg Service Time	Edge/Cloud Ratio
SINGLE_TIER	50	11,698	56	0.48%	2.81 s	100% / 0%
TWO_TIER_WITH_EO	50	13,260	7	0.05%	0.36 s	57% / 43%

The simulation was conducted using two orchestration scenarios: SINGLE\_TIER and TWO\_TIER\_WITH\_EO, each tested under different conditions to evaluate performance, scalability, and fault tolerance of the proposed system. The policy used in all simulations was BEST\_FIT, as it dynamically selects the most appropriate processing layer (edge or cloud) based on available resources and delay sensitivity.

### Scenario Comparison

- In the SINGLE\_TIER scenario, all tasks were processed locally on the Raspberry Pi Zero 2 W devices acting as edge nodes. While this decentralized approach provided full independence from the cloud, it also led to a slightly higher failure rate (0.48%), especially under simulated network constraints (Mobility/WLAN).
- In contrast, the TWO\_TIER\_WITH\_EO scenario allowed tasks to be processed either at the edge or in the cloud. This hybrid architecture significantly reduced the failure rate to just 0.05% and optimized task distribution across layers.

### Performance Metrics

- **Average service time** was considerably lower in the TWO\_TIER\_WITH\_EO setup (0.36s vs. 2.81s), highlighting the advantage of cloud offloading when low latency is required.

- **Task completion rate** improved in the hybrid system due to cloud assistance during edge overloads.
- **Server utilization** showed balanced load distribution between edge and cloud layers, with the BEST\_FIT policy preventing saturation.

### *Impact of Scaling*

- A single-device test helped verify the functionality of orchestration and processing.
- When scaled to 50 vehicles, the system maintained low failure rates and consistent service times, demonstrating its robustness and suitability for real-world deployment.
- The increase in devices and task load was effectively handled thanks to the orchestration policy and the use of a cloud datacenter with four high-capacity VMs.

### *Orchestration Policy: BEST\_FIT*

- This policy was crucial in reducing failures and ensuring efficient task allocation.
- It dynamically analyzed resource availability and network latency to decide whether to process tasks on the edge or in the cloud.
- The system benefited from this adaptive strategy, especially during peak load moments or when certain devices experienced connectivity issues.

### *Failure Analysis*

- Most failures in SINGLE\_TIER were due to the Mobility/WLAN.
- In TWO\_TIER\_WITH\_EO, rare failures were caused also by mobility-related factors (e.g., temporary disconnection), not by capacity issues.
- Importantly, **no failures** occurred due to VM capacity in either scenario, showing proper resource provisioning.

## 9. Conclusions

This project demonstrated the feasibility and advantages of using a hybrid IoT-Edge-Cloud architecture for monitoring driver fatigue and environmental hazards in vehicles. By equipping each car with a Raspberry Pi Zero 2 W and a set of physiological and gas sensors, the system is capable of detecting critical situations and responding locally in real-time, while also leveraging cloud services for data backup, analysis, and remote alerting.

Simulations conducted in EdgeCloudSim confirmed that the combination of edge computing and intelligent cloud orchestration significantly improves system performance, scalability, and reliability. The BEST\_FIT orchestration policy, in particular, proved effective in balancing load and reducing task failure rates, especially under heavy traffic and long-duration scenarios.

Key takeaways:

- The TWO\_TIER\_WITH\_EO architecture provided the best results in terms of low latency, high task completion, and minimal failure rates.
- Local processing with Raspberry Pi Zero 2 W was sufficient for basic detection tasks but limited under high-frequency workloads.
- Cloud integration ensures long-term storage, centralized dashboards, and redundant alert mechanisms.

*Although the system was not physically deployed, the simulation-based approach using realistic parameters provides strong evidence of its feasibility.*