University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Final Report
Fleet Vision

Group 2025.3

Prepared By:

Isaiah Richards	20905567
Vithursan Sarvalogan	20898064
Naumaan Sheikh	20886320
Aaron Spence	20882977

Consultant:

Dr. Otman Basir

Table of Contents

1 High-Level Description of Project **2**
1.1 Motivation 2
1.2 Project Objective 2
1.3 Block Diagram 2
1.3.1 Video Capture Units Subsystem 2
1.3.2 On-Board Vehicle Diagnostics Subsystem 2
1.3.3 Central Hub Subsystem 3
1.3.4 Data Processing Server Subsystem 3
1.3.5 Web Application Subsystem 3
2 Project Specifications **4**
2.1 Functional Specifications 4
2.2 Non-Functional Specifications 6
3 Detailed Design **7**
3.1 Video Capture Units Subsystem 7
3.1.1 Frame Rate Optimization Quantitative Technical Analysis 8
3.2 On-Board Vehicle Diagnostics Subsystem 9
3.3 Central Hub Subsystem 10
3.3.1 Wireless Communication (Bluetooth, WiFi, Cellular) 10
3.3.2 Ease of Implementation and Physical Integration 12
3.3.3 Cost 13
3.3.4 Evidence-Based Conclusion 13
3.4 Data Processing Server Subsystem 13
3.4.1 Hosting Data Processing Server Subsystem and Communication Protocols 13
3.4.4 Machine Learning Models for Driver Distraction and Drowsiness Detection 15
3.4.4.1 Optimization Quantitative Technical Analysis for Selecting Model Parameter Values 16
3.4.5 Driver Distraction and Drowsiness Detection Model Output Processing 17
3.5 Web Application Subsystem 18
3.5.1 Fleet Vision Database 18
3.5.1.1 User Model: 18
3.5.1.2 OBD Model: 18
3.5.1.3 Driver Event Model: 19
3.5.2 Web Application 19
3.5.2.1 Web Optimizations: 19
3.5.2.2 Web Application Design: 20
3.5.2.3 Dashboard: 20
3.5.2.4 Vehicle Analytics: 20
3.5.2.5 Driver Analytics: 21
4 Discussion and Project Timeline **21**
4.1 Evaluation of Final Design 21
4.2 Use of Advanced Knowledge 21
4.3 Creativity, Novelty, Elegance 22
4.4 Student Hours 22
4.5 Potential Safety Hazards 22
4.6 Project Timeline 23
5 References **24**

1 High-Level Description of Project

1.1 Motivation

Each year in the US, fleet management companies suffer losses of approximately $300,000 due to vehicle downtime caused by accidents and delayed vehicle servicing [1] which in part is due to infrequent vehicle health checks. In 2017 alone, fleet accidents in the US accounted for $59 million in losses for employers [2] causing accidents in fleet vehicles to be a major financial drain for the industry. With drivers being distracted in more than 50% of accidents [3], distracted driving is currently the leading cause of vehicle crashes and a key point of focus when attempting to increase driver safety and reduce financial loss caused by accidents. It can be extremely difficult for fleet managers to monitor and promote safe driving behaviours amongst their drivers while working due to the disconnect between the drivers and managers, causing many companies to ignore this huge safety and financial liability. Many fleet management companies would greatly benefit from a system that can monitor the vehicle health of all vehicles in the fleet and provide real time metrics on drivers and their driving behaviours.

1.2 Project Objective

The goal of this project is to design an all-in-one platform which provides analytics on driver behaviour and current vehicle state and diagnostics which is used by fleet managers to help prevent accidents and reduce vehicle downtime. This would give fleet managers greatly increased visibility into their fleets and provide an easy-to-use platform to view and manage the aforementioned analytics.

1.3 Block Diagram

Below, please find a brief description of each of the subsystems included in the block diagram of our Fleet Vision system in Figure 1.

1.3.1 Video Capture Units Subsystem

The Video Capture Units subsystem depicted in the block diagram is designed to monitor driver behaviour using two cameras. The two cameras consist of a camera facing the front of the driver, in order to capture facial activity such as eye and mouth classifications, and a camera that faces the driver from the right hand side to capture their entire body and hand activities. Both cameras will each be connected to and powered by their own microcontroller unit, which then uses a WiFi connection to stream video frame data directly to the Real-Time Data Processing subsystem to be further processed by machine learning models for distraction and drowsiness classifications. Each microcontroller unit will be powered by its own battery.

1.3.2 On-Board Vehicle Diagnostics Subsystem

The On-Board Vehicle Diagnostics subsystem is designed to support real-time monitoring and streaming of vehicle diagnostic and state information. This involves the utilization of the widely adopted OBD-II port of vehicles, which allows access to live diagnostic results of several of the vehicle's internal systems along with real-time vehicle state information, measured through various sensors in the vehicle. We interface with the OBD-II port using an OBD-II sensor and captured data is then streamed to an attached microcontroller unit, which then uses a WiFi connection to stream formatted data to the Real-Time Data Processing subsystem. The streamed data will include useful sensor values from the car that directly

influence driver safety behaviours, along with checking to see if an engine issue is detected. The OBD-II sensor will be powered through the OBD-II port with power coming from the vehicle, and the microcontroller unit connected to the OBD-II sensor will be powered by its own battery. The microcontroller unit will be connected to the OBD-II sensor via wired pins to ensure seamless data transmission between the boards. This will allow the OBD-II sensor to pull sensor data from the vehicle, the microcontroller to take data from the OBD-II sensor and stream it to the Real-Time Data Processing subsystem over a WiFi connection.

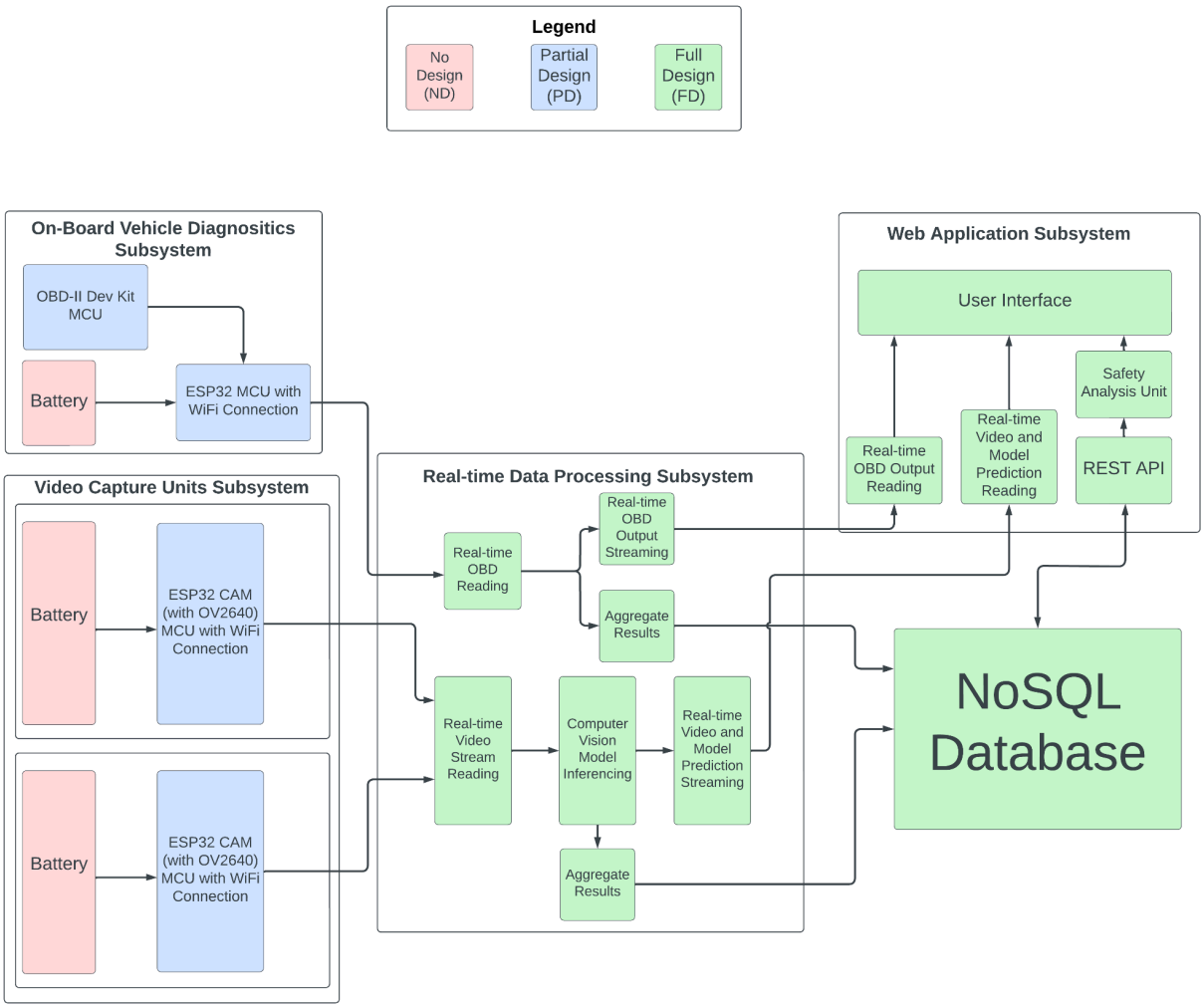
1.3.3 Real-time Data Processing Subsystem

The Real-time Data Processing subsystem receives, processes and analyzes data received from the Video Capture Units subsystem and On-Board Vehicle Diagnostics subsystem all in real time. It also streams annotated video data and OBD data to the web app in real-time, allowing users of the Fleet Vision platform to be able to go into the web app and see the current driver behaviour including the live video stream of what the driver is currently doing from the front and body angles, and our annotations overlaid on the video stream with what our models think the driver is doing as well as the live vehicle state including speed, revolutions per minute and other relevant data. It also keeps track of the drivers' most recent behaviour to detect things like drowsiness and distraction in real-time as they happen, allowing us to send an alert to the web app which would signal the driver that they need to correct their driving immediately as it is unsafe enough that it merits an alert. In addition to streaming results directly to the web app in real-time, the Real-time Data Processing subsystem also is responsible for storing collected data in a database to have asynchronous analysis performed on it later on. The Real-time Data Processing subsystem must be clever about what data it chooses to save (so we are not wasting storage and doing excessive database writes) and also must be clever about how different data is synchronized to ensure that we can tie data from different sources to the same point in time (we have data coming in from two Video Capture Units and one OBD Capture Units at different rates). So to summarize, the primary jobs of the Real-time Data Processing subsystem are threefold - read/receive video stream inputs and OBD stream inputs from the Video Capture Units subsystem and On-Board Vehicle Diagnostics subsystem, process and classify inputs and stream classified results to the web app and finally to also save classified results to the database and do all three of these tasks in real-time.

1.3.4 Web Application Subsystem

The block diagram shows the architecture of a complete Web Application subsystem that is intended to display driver and vehicle diagnostic data and allow management of data. This system's central component is a REST API that provides access to data in a central database that includes driver behaviour analytics data and vehicle state and diagnostics data written by the Data Processing Server. The Driver Behaviour Analysis Unit requests and processes relevant data from the database and prepares it to be displayed in a clear, appealing and easy to understand way in the user interface. Concurrently, the OBD-II Analysis Unit processes vehicle state and diagnostic data from the database to be displayed in the user interface. The system also has an Admin Operations module that verifies admin accounts and gives them access to special features like viewing and editing all registered drivers and vehicles in a given fleet. With its categorized views, this framework makes sure that all pertinent data is effectively presented through the user interface, improving both operational oversight and user experience.

Figure 1. Block diagram of Fleet Vision System.



2 Project Specifications

2.1 Functional Specifications

The essential functional specifications are listed below in **Table 1**, outlining whether each specification belongs to a specific component, subsystem, or the overall end to end system, along with a general specification name and a more in-depth description of the specification. The non-essential functional specifications are listed below in **Table 2**.

Table 1. Essential functional specifications.

Project Element	Specification	Description
End-to-End System	Driver Monitoring Accuracy	The system must be able to detect driver drowsiness and distracted driving events with an accuracy of at least 90%. The false positive rate should be no higher than 5%.

	Installation and Setup Time	The initial installation and registration of embedded subsystem components to a vehicle, followed by face detection to start a driving session and processing sensor data to be displayed in the web app, must be completed within 4 minutes.
Embedded System Components (Video Capture Units and On-Board Vehicle Diagnostics)	Data Transmission Rate	Each embedded subsystem component must be able to stream all required data to the desired destination with an average throughput of 5 frames per second, where each frame includes image or vehicle state data. This framerate is chosen to minimize bandwidth, maximize model prediction results, and account for limited factors of hardware and network speeds.
	Operational Duration	Each embedded subsystem component, along with its battery, must support 12 hours of continuous operation.
	Recharge Time	Each embedded subsystem component must be able to recharge its battery to full capacity in 2 hours.
Software System Components (Data Processing & Web App Subsystems)	Web App User Interface	The web app must provide a user interface through which the user can view collected data and provide insights obtained from this data per user, vehicle and across the entire fleet.
	Data Synchronization and Processing	The server must be able to synchronize input frame data from embedded subsystem components, run the appropriate Machine Learning models on all inputs, and generate appropriate outputs to accurately predict driver behaviour every second with at least 90% accuracy.
Video Capture Units Subsystem	Image Transmission Rate (Side)	MCU must be able to receive image frame inputs from the camera and stream them to the server using WiFi at a minimum average frame rate of 5 frames/second.
	Camera Resolution	Camera resolution must be sufficient to capture body, objects, and facial features (eyes and mouth) for the model to process and classify with confidence of at least 85%.
On-Board Vehicle Diagnostics Subsystem	Vehicle State Data Streaming	MCU must be able to pull live vehicle sensor data/diagnostic results from the OBD-II sensor and stream them to the server using a WiFi connection for live vehicle data, streaming at at least 1 frame per second.

Table 2. Non-essential functional specifications.

Project Element	**Specification**	**Description**
Software System Components (Data	Component Disconnection Alert	The web app should be able to detect and notify users when an embedded subsystem component disconnects from the server

Processing Server & Web App Subsystems)		for any reason.
	User Auth & Access Control	The web app should have user authorization to dictate which users can view/manage data for certain users, vehicles and fleets.
On-Board Vehicle Diagnostics Subsystem	Real-Time GPS Tracking	The OBD-II MCU should incorporate a GPS tracker that sends live location updates to the server at a rate of 5 frames/second.
	CAN Bus Data Collection	The OBD-II scanner should collect CAN Bus data to detect real-time vehicle data like turn signals and breaking intensity.

2.2 Non-Functional Specifications

The essential non-functional specifications are listed below in **Table 3**, outlining whether each specification belongs to a specific component, subsystem, or the overall end to end system, along with a general specification name and a more in-depth description of the specification. The non-essential non-functional specifications are listed below in **Table 4**.

Table 3. Essential non-functional specifications.

Project Element	Specification	Description
Embedded System Components (Video Capture Units and On-Board Vehicle Diagnostics)	Compact Component Container	Each embedded subsystem component must be hosted inside a closed container no larger than 10cm x 10cm x 6cm including sensor(s), an MCU and a battery with no exposed electronics.
	Quick Installation	Installation/ reinstallation of all 3 embedded subsystem components must be able to be completed in under 2 minutes.
	Charge Port	Each embedded subsystem component must have a charge port which can be used to charge the battery for that component.

Table 4. Non-essential non-functional specifications.

Project Element	Specification	Description
Software System Components (Data Processing Server & Web App Subsystems)	Web App Battery Life Indicator	The web app should have a battery life indicator to inform the user of how much battery life is left in the battery powering each embedded subsystem component installed in a vehicle.
Embedded System Components (Video Capture Units and	Hardware Cost Limitation	All hardware components included in the system should cost no more than $150.00 CAD.
	Cable Management	Design should not have any loose wires and minimize

On-Board Vehicle Diagnostics)	and Installation	invasiveness to the vehicle interior as much as possible.
	In-Vehicle Battery Charging	Battery charging should be possible from within the car while still meeting minimum charge speed requirements.

3 Detailed Design

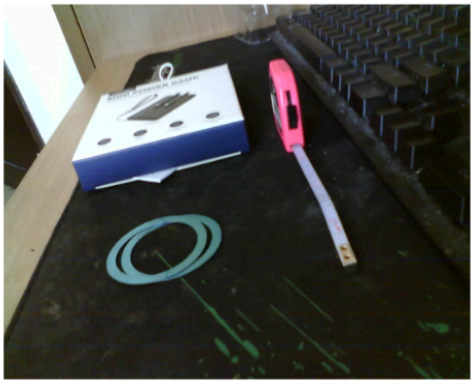
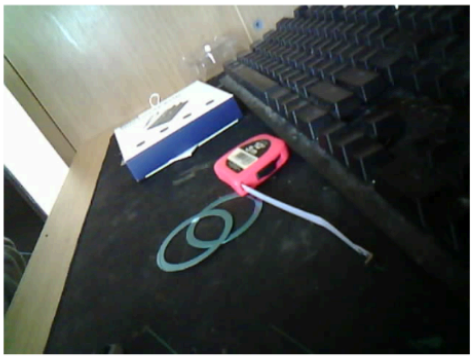
3.1 Video Capture Units Subsystem Design

This system's core component, the ESP32-CAM module, gives us the ability to capture and transmit visual data in real time. The performance of the ESP32-CAM is assessed in this analysis, with emphasis on important variables, such as processor power, data transmission rates, latency, and image processing capabilities. By evaluating these factors, we attempt to determine whether the ESP32-CAM satisfies the performance needs of our project in terms of tracking driver behavior and vehicle diagnostics.

3.1.1 Quantitative Technical Analysis - Frame Rate Optimization

Based on our specifications, each embedded subsystem component must be able to stream all required data to the desired destination with an average throughput of 5 frames per second (FPS). The first step in this pipeline is when the ESP32-CAM transmits the frames of the stream within the car (2 cameras: one of them for capturing a stream of the driver’s face/head, and the other camera capturing the driver’s side body view from the passenger’s side of the car). In order to capture fine details, such as slight changes in facial expression and eye positioning, and transmit them at a fast rate, we require sufficient resolution of images, as well as an efficient transfer rate of 5 FPS. In order to find the most appropriate resolution, tests were conducted with the ESP32-CAM with the following resolutions: 640x480, 1280x720, and 1280x1024. We are essentially applying a data driven approach with optimization techniques to determine if the ESP32-CAM can be configured to meet our requirements. Let’s consider the following simulations, which were completed through a web server hosted on the ESP32-CAM, developed in the Arduino IDE.

Table 5. Image Capture Quality and Measured FPS

Resolution	Image	FPS Logs	Average FPS
FRAMESIZE_SXGA (1280x1024)		FPS: 6 FPS: 7 FPS: 8 FPS: 7 FPS: 6 FPS: 7 FPS: 7 FPS: 6 FPS: 7 FPS: 6 FPS: 7 FPS: 7 FPS: 6	Approximately 6 FPS
FRAMESIZE_HD (1280x720)		FPS: 10 FPS: 11 FPS: 9 FPS: 9 FPS: 8 FPS: 10 FPS: 9 FPS: 13 FPS: 12 FPS: 13 FPS: 10 FPS: 10 FPS: 11 FPS: 9	Approximately 10 FPS
FRAMESIZE_VGA (640x480)		FPS: 24 FPS: 24 FPS: 22 FPS: 24 FPS: 24 FPS: 24 FPS: 24 FPS: 24 FPS: 18 FPS: 21 FPS: 23 FPS: 22 FPS: 18	Approximately 22 FPS

In our initial testing phase, we evaluated three resolutions on the ESP32 camera module to determine which strikes the best balance between image clarity and throughput: FRAMESIZE_SXGA (1280×1024), FRAMESIZE_HD (1280×720), and FRAMESIZE_VGA (640×480). Our benchmark results showed that SXGA reached about 6 frames per second (FPS), HD produced roughly 10 FPS, and VGA peaked around 22 FPS. Although SXGA offers higher detail, it tends to hover near the critical 5 FPS minimum threshold—leaving less room for additional network overhead. Conversely, VGA can exceed the 5 FPS requirement but yields less-detailed images, which is problematic for detecting subtle facial expressions or eye movements. Consequently, **FRAMESIZE_HD (1280×720)** emerged as the most optimal solution, delivering sufficiently detailed video at an average of about 10 FPS locally, which provides a comfortable buffer when scaling up to a publicly hosted server.

Another critical factor is ensuring that the ESP32's Wi-Fi connection remains robust, particularly when streaming video to an external network. To quantify signal stability, we measured the Wi-Fi strength in dBm using a custom Arduino sketch. This sketch connects to a specific wireless network and continuously reports the Received Signal Strength Indicator (RSSI).

Figure 2a. Serial Monitor Output of RSSI With Antenna

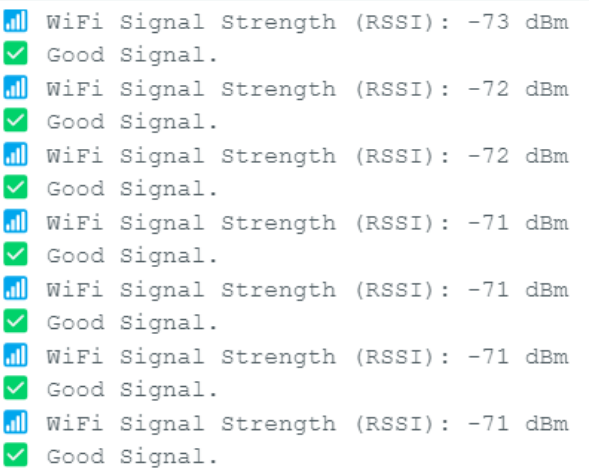
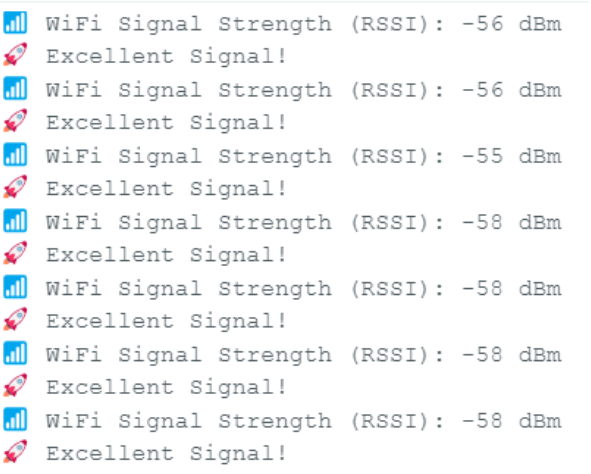


Figure 2b. Serial Monitor Output of RSSI Without Antenna



By comparing average RSSI values with and without an external antenna, we saw a marked improvement: roughly **-57 dBm** with the antenna versus about **-72 dBm** without it. The classifications of what is considered a good connection vs an excellent connection is an RSSI value ranging between -80 and -60 and an RSSI value above -60 respectively. Since higher (less negative) RSSI values indicate a stronger signal, adding the antenna helps maintain reliable throughput, reducing the likelihood of dropped frames or slowdowns below our 5 FPS requirement, which is especially important when sending data to a public server where network fluctuations are more pronounced. This increased reliability will also aid when considering that the esp32 is encased in the prototype container, reducing its maintainable connection strength due to the obstruction. When considering this, since these connection strengths were measured without the camera being encased, we can expect the connection strength to drop when the camera is encased and an antenna is not utilized. In comparison, when using the antenna, it will be exposed outside of the box, so the camera's ability to hold a strong connection will not be hindered when the container encasing is utilized in the final prototype.

3.2 On-Board Vehicle Diagnostics Subsystem Design

The On-Board Vehicle Diagnostics (OBD) subsystem is an integral system for real-time monitoring of vehicle diagnostics, as it is the source of data from the various systems in a vehicle. While it has great capabilities, its practical usage shows challenges concerning the use of physical space, extra cost, and risking the loss of functionality.

To begin, considering the OBD functionality we wanted to include in this project, there were very limited development kit options available in the market. We decided to proceed with the development kit from Longan Labs due to its lower cost and availability of detailed documentation. It also comes with an enclosed and small form factor that is minimally invasive, and negligible to a driver. Considering this development kit as the fundamental piece of the OBD subsystem that we want to work off, we need to consider many things to ensure functional and non-functional specifications are met, along with the subsystem being overall feasible and not overly complicated.

The OBD development kit allows us to connect directly to the vehicle's OBD port and program the board via the Arduino IDE to receive vehicle sensor data. The issue with using just this development kit in the OBD subsystem is the inability to send data from the OBD subsystem to the Real-Time Data Processing subsystem. To combat this issue, we explored many options that would be cost-effective and would be small enough to not be a hindrance to the driver. The initially considered options included a Raspberry Pi 4, Arduino Rock 3 Model C, and a Particle Boron LTE-M, all of which had options to add/use cellular networks via additional modules or included functionality. Due to costs and large board size, we decided to not move forward with the Raspberry Pi 4 or the Arduino Rock 3 Model C. We tried integrating the Particle Boron LTE-M with the OBD-II development kit, but the board had significant limits on network usage, which made the board an unfeasible option. Due to the lack of cost-effective and small form factor options available for cellular connectivity, we decided to go with WiFi connected microcontroller boards, which had many more options for much lower costs. Due to the wide availability, support and low costs of ESP32 boards, we decided to use a WiFi enabled ESP32 board to receive data from the OBD-II development kit and transmit the data to the Real-Time Data Processing subsystem.

In this process, the microcontroller sends requests for specific On-Board Diagnostics (OBD) parameters, often referred to as PIDs, to the vehicle's CAN (Controller Area Network) bus, then parses and packages the responses in a JSON structure for easy transmission. First, the code initializes communication on the

CAN bus and configures acceptance filters so that the microcontroller only listens for responses corresponding to OBD-II requests. To read a particular metric, such as vehicle speed or engine RPM, a PID request is formatted with the appropriate mode (commonly Mode 01, indicating a request for current data) and the PID itself (for example, 0x0D for speed or 0x0C for RPM). These requests are broadcast on the standard OBD-II query address, and the code then waits for a response on the corresponding OBD-II reply address. Once the microcontroller receives a valid response, it extracts the relevant bytes from the payload to compute values like speed (in km/h) or RPM (by combining two response bytes and dividing by four according to the OBD-II standard). The process also includes checking the vehicle's "check engine" status and the number of diagnostic trouble codes (DTCs) by sending another PID request (0x01), reading the response, and determining if the Malfunction Indicator Lamp (MIL) is active and how many pending DTCs exist. Finally, the code consolidates all these values into a JSON object, which contains speed, RPM, and any warning or DTC information, and sends it to the serial port, where the ESP32 can forward it on to the next subsystem for further processing. This approach ensures that each essential piece of vehicle telemetry is collected in a standardized way, ready to be transmitted over Wi-Fi or stored for analysis.

To pull data from the OBD-II development kit using the ESP32 microcontroller board, we had very limited options. The most straightforward option we first considered was to simply pull the data from the micro-USB port of the OBD-II development kit into the micro-USB port of the ESP32 board. Since this is where the serial data is available from the board when connected to a PC, we believed reading from the USB port would be most simple. The other option would be to wire the pins together to transmit data via specific pins from the boards, which we originally believe would be unnecessarily complicated. Upon further research and testing, we found the simplest way of communication between the OBD-II development kit board and the ESP32 board was to connect the TX and RX pins, which are standardized for data transmission between boards, known as a UART connection. This would ultimately save space in the OBD subsystem since the wires can be a lot smaller and compact compared to a micro-USB to micro-USB connection.

3.2.1 Quantitative Technical Analysis

3.2.1.1 Quantitative Technical Analysis - Frame Rate and Data Throughput

In consideration of hardware, network, and data transmission limitations across several devices, we must identify the most limiting factors in the overall OBD subsystem and the end-to-end communication to determine how much data we can retrieve from the vehicle, send to the Real-Time Data Processing subsystem over a WiFi network, and finish processing the data to determine the ideal frame rate.

To meaningfully investigate and understand the ideal frame rate, we must identify timing measurements for receiving sensor data from the vehicle's OBD port to the OBD-II development kit, transmitting the data from the OBD-II development kit to the ESP32 over UART, sending the data to the server via HTTP request, parsing the data, and receiving the HTTP response. Using measurements like data size, average time to parse and process the data, average time to send data over a WiFi connection, total end-to-end latency, data throughput, and success rate, we can figure out many metrics, like ideal frame rate and required bandwidth per frame of data to account for weaker/stronger WiFi connections.

Network testing for the ESP32 board was conducted in many different conditions to test network capabilities of the board. These conditions included different hotspot network providers and different cellular connection strengths to see if the board is capable of streaming data at a minimum rate of one frame of vehicle sensor data per second.

Table 6. Network Hotspot Comparison for OBD Subsystem's ESP32

	Bell (weak signal, 2 bars)	**Rogers (weak signal, 1-2 bars)**	**Rogers (strong signal, 3-4 bars)**
Average Upload Speed	18.27 kbps	12.48 kbps	47.83 kbps
WiFi Signal Strength	-76 dBm (Weak)	-53 dBm (Good)	-42 dBm (Excellent)

To make use of this data, we must figure out the size of each frame sent to the server from the ESP32. Considering the data is all formatted into a JSON, the raw JSON payload itself will be a maximum of 65 bytes. Since the data is sent to the server via an HTTP POST request, each frame will have HTTP headers added to the JSON. By running tests on the ESP32, the HTTP header size comes out to 530 bytes for each frame. This means:

```
Total frame size = 65 bytes + 530 bytes = 595 bytes
```

At a frame size of 221 bytes, the network upload speed required can be calculated as such:

```
Required Upload Speed (kbps) = 595 bytes * 8 bits/byte * fps
```

These calculations can be seen in the table below.

Table 7. Ideal Frame Rate Calculation for OBD Subsystem Data Transmission

FPS Sent to Server	**Data Throughput (bytes/sec)**	**Required Upload Speed (kbps)**
1	595	4.76
2	1190	9.52
3	1785	14.28
4	2380	19.04
5	2975	23.8

This shows that theoretically, the OBD subsystem should have no difficulty in sending data to the server at the minimum rate of one frame per second, however, this calculation excludes the time it takes the OBD-II development kit to reliably receive vehicle sensor data, convert the data into a formatted JSON string, and transmit it over UART. Along with these limitations, there are also server latency and processing time for each frame, the interval between each frame sent due to waiting for an HTTP ACK message, and ESP32 hardware processing limitations. To get these values, we tested these latencies and delays on the weakest network configurations to imitate non-ideal real-world conditions under which we can reliably transmit data.

Through tests, we found that the total time it takes to get vehicle sensor data and send it over UART to the ESP is 400 ms on average. Using this, the table below shows calculations regarding ideal frame rate.

Table 8. Total Network Information Comparison to Identify Ideal Frame Rate

FPS Sent to Server	Total Bytes (HTTP header + JSON)	Required Upload Speed (kbps)	Average Processing Time on Weak Network (ms)
1	595	4.76	381
2	1190	9.52	763
3	1785	14.28	1144
4	2380	19.04	1526
5	2975	23.80	1907

We can see from this data, that at a full 2 frames per second, all components of the OBD subsystem are able to reliably accommodate this data transfer. Even with a network connection providing only 10 kbps, which is noticeably lower than the lowest connection speed we were able to get, the OBD-II development kit, ESP32, and network will all be able to reliably send 2 frames of vehicle sensor data per second to the server.

We see that the most limiting factors in this subsystem are the OBD-II development kit receiving data and converting it to a JSON format, which takes 400 ms on average, and the HTTP header size of 530 bytes which significantly increases the amount of data we need to send over the network. However, even with these limitations, we can reliably send slightly over 2 frames per second of vehicle sensor data, which is more than our minimum required frame rate. Any more than slightly over 2 frames per second may lead to corrupted data from the OBD-II development kit, lost packets when sending over HTTP, or significantly delayed frame data from the ESP32 board to the server. This gives us a good benchmark for future improvements on the OBD subsystem side if we would want to add data of more vehicle sensors, how much more data we can add per frame to continue reliably sending data at the required frame rate.

3.2.1.2 Quantitative Technical Analysis - Power

To power the components in the OBD subsystem, we must ensure the boards are getting regulated 5V with 3A for reliable operation[4][5]. With the OBD-II development kit plugged into the car, the board receives regulated 5V 3A from the OBD port which is used to power the OBD-II development kit. Since the board also has a 5V pin which should theoretically output 5V when the board is connected to power, we tried a configuration in which the OBD port from the vehicle is used to power the OBD-II development kit, and the 5V power is taken from the board's 5V output pin to power the ESP32 board, eliminating the need for any external power source for the OBD subsystem, reducing size, weight, and complexity of the overall system. However, when testing, although the ESP32 board received power from the OBD-II development kit's 5V pin and illuminated the on-board LED, the HTTP upload was not being received on the server. This indicated that although the board was being powered, it was likely not receiving a reliable source of 5V and 3A, which is the developer recommended power supply. The

specifications sheet specifies that the board may operate with a power source of 3.3V and 1A, but may not work reliably, which is what we suspected may be happening. Upon measuring the voltage and current from the OBD-II development kit's board, using the board's GND and 5V pins as reference, the board was outputting a stable 3.3V with 1A. This means that the ESP32, programmed to pull data via UART and stream data over WiFi, requires more than the minimum power supply of 3.3V and 1A. For further inspection, we tried powering the ESP32 board with a 5V battery with 1.4A output, which yielded the same results. This means that the OBD subsystem requires an external power source that can supply a regulated 5V with 3A to the ESP32 board for reliable operation. To accommodate this, the power supply must be small and light enough to remain attached to the OBD-II development kit without disrupting the connection with the OBD port, ensure the battery is properly covered and not exposed to any flammable material, and the battery can power the ESP32 board for the length of a full drive.

3.4 Real-time Data Processing Subsystem Design

In this section, we will explain the details of the final design that we have gone with for the Real-time Data Processing subsystem. We will explain the overall design for this subsystem as well as getting into the details for the design for some of the particular components of interest in this subsystem. We will also explain the process that went into our design decisions and justify why we made the choices that we did along with providing quantitative technical analysis to justify some of the final decisions we made.

3.4.1 Subsystem Overview

The Real-time Data Processing subsystem is designed to continuously collect, analyze, and process information from the Video Capture Units subsystem and On-Board Vehicle Diagnostics subsystem, in a real-time basis. We chose to implement this subsystem in Python due to its extensive support for model operations through PyTorch and TensorFlow frameworks, as well as its rich ecosystem of computer vision libraries like OpenCV that were essential for our application. This subsystem directly streams annotated video and OBD data to the web application, allowing users of the Fleet Vision platform to watch live driver behaviour. The web app shows synchronized front and body camera feeds with overlays depicting what our models infer about the actions being performed by the driver along with real time vehicle metrics including speed, RPM and important diagnostic data. All this data that the web app shows live is streamed from this subsystem in real-time.

Additionally, this subsystem continually assesses the driver's recent activity to identify potential drowsiness or distraction. An alert is sent straight to the web app when unsafe behaviour is detected in real-time, triggering the driver to take corrective action as needed. Simultaneously with the live processing and streaming, the subsystem records the most relevant information in a database for asynchronous analysis by the web app in the future. This allows the web app to have pages that show analytics of a given drive session, these analytics will be created from the classifications for timestamps within the session that are stored in the database. Our database schema allows us to save storage by preventing writing to the database if identical data is received and aligns data from multiple inputs (when the two video streams and the OBD stream all operate at different rates) to the same time stamp.

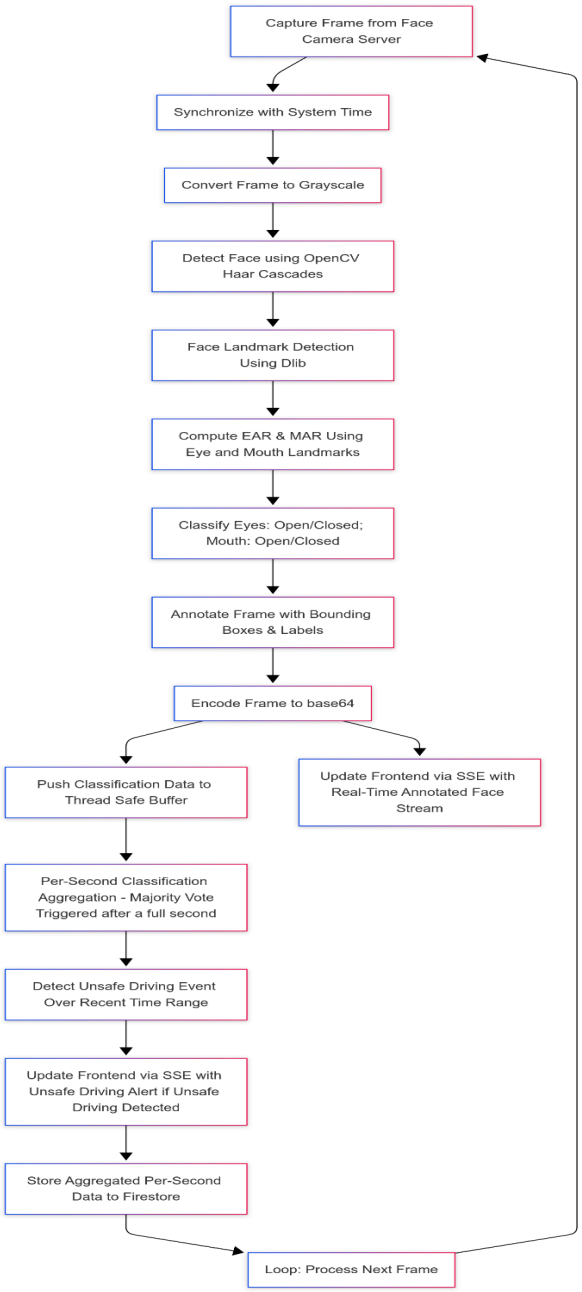
The system is built with three main workflows executing in parallel including real-time processing of Face Video Stream data, Body Video Stream data, and OBD data. We will discuss each of the components

above, along with the quantitative technical analysis carried out at each step to optimize all three pipelines in later sections.

3.4.2 Face Video Stream Processing

In this section we describe the design and implementation of our Real-time Face Video Stream Processing subsystem, which does live video capture from an ESP32 CAM AI Thinker board, processes each frame to detect and classify facial states (i.e eyes open/closed, mouth open/closed) and synchronizes this with data from other sources within the Fleet Vision platform. The below diagram illustrates the flow of data.

Figure 3. Flow Diagram of Real-time Face Video Stream Processing Workflow



3.4.2.1 Workflow Overview

Our Real-time Face Video Stream Processing subsystem continuously captures live video from an ESP32 CAM AI Thinker board, processes each frame to detect the eyes and mouth and whether they are open or closed, and synchronizes these results with data from other elements including the body video stream and the OBD data stream. The ESP32 CAM board runs its own camera web server, making the video stream available at a URL like http://esp32face.local/stream. Rather than having the board push information, our subsystem proactively retrieves frames from the server using OpenCV stream capturing. To accomplish this dependably, even when the board's IP varies because of DHCP when it connects to a new network, we use mDNS (multicast DNS) to resolve the hostname esp32face.local. This ensures that as long as the board and our subsystem are on the same local network, the current IP is dynamically uncovered at the start of each drive session.

Once connected, each frame is time stamped using the server's clock, which offers a single source of truth for synchronizing data across the face, body, and OBD streams. The workflow starts with the Frame Capture and Timestamping step where frames are pulled from the ESP32 CAM's web server and tagged with the current server time. Next comes preprocessing, where each frame is converted from color (BGR) to grayscale, lessening computational load for detection algorithms and ensuring the input comes in the format that they expect it to. Next comes face detection where OpenCV's Haar cascade classifiers are applied to identify the face. If multiple faces are detected, the largest bounding box is presumed to be the driver. Next comes landmark detection where Dlib's face landmark predictor is used to pinpoint precise locations of facial landmarks around the eyes and inner mouth. Next comes EAR and MAR calculations where the Eye Aspect Ratio (EAR) and Mouth Aspect Ratio (MAR) are calculated from the Dlib eye and inner mouth landmarks. These ratios help determine whether the eyes are open or closed and whether the mouth is open, based on thresholds. Next comes annotation and encoding where the frame is annotated with bounding boxes and textual labels, then encoded into Base64 format for streaming. And the final step is per-second aggregation where rather than processing individual frames separately, classifications are aggregated over each second using a majority vote to get a single classification to represent the entire second. This per-second classification simplifies synchronization with the body camera and OBD data as we will have one piece of data from each for a given integer second, and it is stored in Firestore using a structure that favors rapid reads.

3.4.2.2 Design Choices

Network Connections via mDNS: The ESP32 CAM board is given the hostname esp32face, which is resolved via mDNS. This adaptable resolution is critical because the board usually gets its IP address through DHCP. By resolving the hostname at the start of each drive session, the subsystem discovers the board's present IP, ensuring dependable communication regardless of what network the boards are connecting to. Note that mDNS works only on neighborhood systems, so if the Real-time Data Processing subsystem were hosted in the cloud, some kind of static IP assignment for the ESP32 CAM boards or other method of reliable network communication would be required to allow the Real-time Data Processing subsystem to figure out the right IP to read from for the boards every time..

Face Detection with OpenCV Haar Cascades: OpenCV's Haar cascade classifiers are used for face identification because they are fast, lightweight, and resource-efficient. Regardless of the accessibility of more modern deep-learning systems, Haar cascades give the essential speed for real-time uses on modest equipment. These do not even require a GPU and can function just fine on CPU. Matter of face, for this

project we are running them on a CPU and have found that they still run faster than the body stream model we are using even when running it on a GPU. Thus because this is the main bottleneck for the face stream workflow and it is much faster than the body stream workflow primary bottleneck even on CPU, it is more than sufficient for our use case. The Haar cascade used is haarcascade_frontalface_alt and we use the detect multi scale method with a scale factor of 1.3. This means that the Haar cascade will search for the face at multiple scales and between each check, it will adjust the scale it is using by a factor of 30%. This is generally considered as a high value for scale factor and it will allow the haar cascade classifier to run much faster which is what we want. We are not too worried about the large scale factor missing a face as the face is so prominent in the image, it rarely gets missed. The other relevant parameter to the OpenCV Haar cascade detection is the minimum number of neighbors. This means how many neighbors each candidate rectangle should have to be considered a detection. We went with a value of 1 which is very lenient, meaning a lot of detections will occur. This is fine as we filter for the largest one to use as the face bounding box. Using a value 1 ensures that if a face is actually present in the frame, it is unlikely to be missed. If we can't find a face in the frame for any reason, then we must output unknown for our eye and mouth state classifications.

Facial Landmark Detection with Dlib: Originally we were using separate OpenCV Haar Cascades to detect various features of the face. We had one for each of the left and right eyes and then we also had one for the mouth. Then we were passing the cropped images we got using the OpenCV Haar Cascades for each of the eyes and mouth and feeding those as inputs into a deep learning model which would determine whether the eyes and mouth were open or closed. Not only was this approach slow - it's also really rigid. With this approach, you cannot do things like slightly adjust the degree that we want the mouth to be open to consider the mouth as open in our system as this would require re-training the models. This Approach is also slow in the sense that we need multiple sequential passes through the OpenCV Haar Cascades and then need to inference a deep learning model for each of the eyes and mouth. Dlib was a really helpful discovery for us as it reliably detects the eyes and mouth much faster than it can be done with OpenCV Haar Cascades when it is given a face bounding box to search within. Dlib does full feature detection of the face in about 3ms on the laptop we are running this subsystem and it does this while only using CPU. The Haar Cascades take at least 50 ms and we had to run multiple of them before in addition to inferencing deep learning models which took a similar amount of time even on GPU. With Dlib, we can calculate the EAR and MAR using simple Euclidean geometry which takes less than a ms, so much faster than using our own trained deep learning models. Also with Dlib and using the EAR and MAR approach rather than custom trained models, we can easily address our EAR and MAR thresholds to tune the system to be exactly accurate.

High-Resolution Input for Improved Accuracy: Despite the fact that sending higher resolution images generally demands more computationally, the face stream can be streamed at high resolution on the grounds that the body stream is slower and it is ok to have the high resolution for the face as it can still stream faster than the body. This decision improves the exactness of facial landmark identification and upgrades the quality of EAR and MAR estimations. It also helps us to meet the below spec that camera resolution must be sufficient to capture body, objects, and facial features (eyes and mouth) for the model to process and classify with confidence of at least 85%.

EAR and MAR Calculation: Calculating the Eye Aspect Ratio (EAR) and Mouth Aspect Ratio (MAR) as accurately as possible is crucial. We used the approach given in the paper "Multi-Index Driver Drowsiness Detection Method Based on Driver's Facial Recognition Using Haar Features and Histograms of Oriented Gradients" [6], to compute the EAR and MAR using the eye and inner mouth features identified by Dlib's facial feature landmark extraction. To compute the EAR, we measure the vertical distances between top and bottom eye landmarks, average these distances, and then divide by the horizontal distance spanning the outer eye corners. An EAR lower than 0.25 typically signifies closed eyes. The MAR is obtained the same way as the EAR except using more feature coordinates around the inner mouth. A MAR value greater than 0.35 typically means the mouth is opened wide enough to be considered yawning, so this is what we look for in our system.

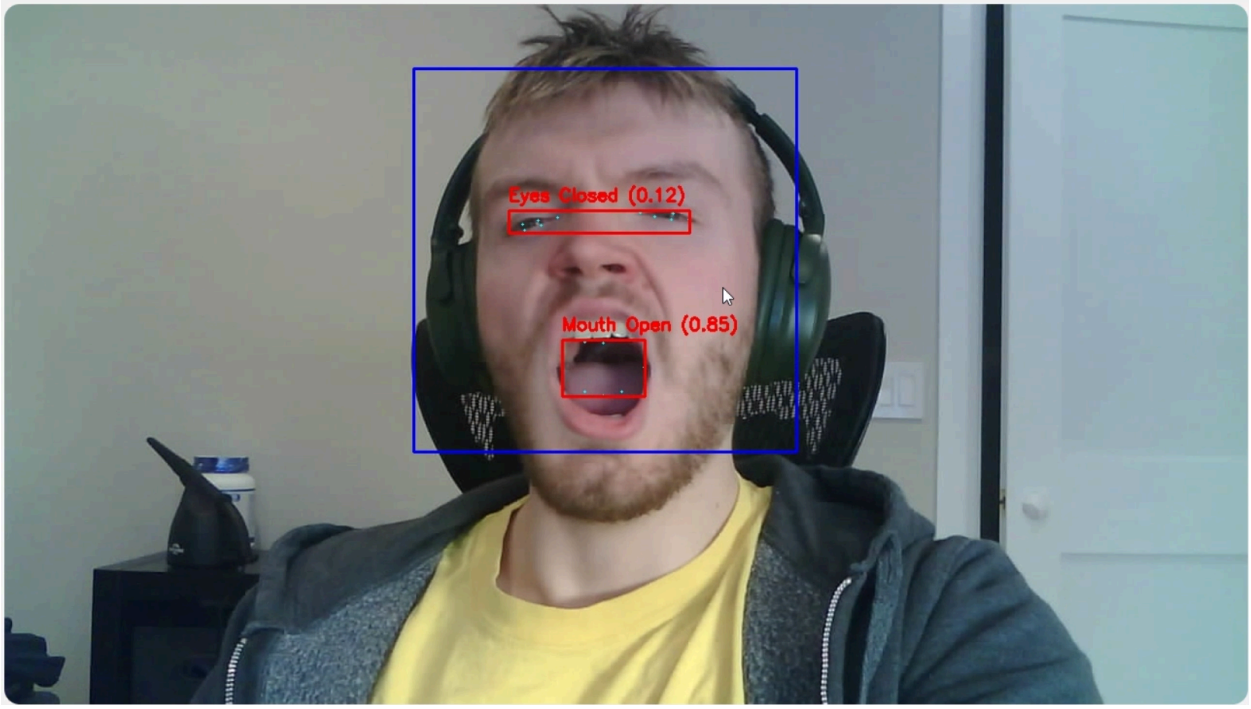
Aggregation and Synchronization: To synchronize information from the face, body, and OBD streams, the subsystem aggregates per-frame classifications on a per-second basis. A majority vote decides the final characterization for every second, giving strength against brief mistakes and simplifying downstream synchronization. The final information is stored in Firestore in a schema that allows fast lookups of details for a given drive session.

Event Detection and Alerting: In addition to standard per-second classification, our subsystem incorporates real-time event detection for enhanced safety. We maintain a queue of per-second classifications spanning the most recent minute. This allows us to compute PERCLOS, the proportion of moments during which the eyes remain shut. If PERCLOS exceeds 20% over the past sixty seconds, the system detects this as an excessive extent of eye closure, a strong indication of drowsiness. Furthermore, should the mouth stay open wide above an aspect ratio of 0.35 for more than three uninterrupted seconds, a yawning occurrence is identified. These are both done based on the approach described in "Multi-Index Driver Drowsiness Detection Method Based on Driver's Facial Recognition Using Haar Features and Histograms of Oriented Gradients"[6]. Upon detecting either event, an immediate notification is generated and transmitted to the Fleet Vision platform via Server-Sent Events (SSE).

3.4.2.3 Example and Conclusion

The figure below shows an example output from our subsystem, where the driver's face is annotated with bounding boxes around the eyes and mouth, and the computed metrics are overlaid ("Eyes Closed (0.12)" and "Mouth Open (0.85)"). These annotations reflect the results of our EAR and MAR computations.. In our system, an EAR below 0.25 typically indicates that the eyes are closed, while a MAR above approximately 0.35 suggests that the mouth is open wide enough to be considered a yawn. And on the image you can see the annotations are marked in red and the eyes are considered close and the mouth is considered open, which makes sense given that the person in the image is clearly yawning.

Figure 4. Face Stream Image Annotated with EAR and MAR



In summary, our Real-time Face Video Stream Processing workflow offers a robust yet efficient approach for tracking driver movements. Through swift face identification, exact facial landmark location, and streamlined second-by-second classification aggregation paired with sophisticated event identification, the subsystem provides real-time warnings while saving everything needed in the database for later analysis. On occasion, edge case facial expressions could elude detection, however the system is able to reliably monitor regular driver behavior. Overall we have met the specification of detecting driver drowsiness with an accuracy of at least 90% and a false positive rate no higher than 5% with this workflow. We also meet our synchronization spec requirement of being able to synchronize input frame data from embedded subsystem components, run the appropriate Machine Learning models on all inputs, and generate appropriate outputs to accurately predict driver behaviour every second with at least 90% accuracy.

3.4.2.4 Quantitative Technical Analysis - Selecting Trained Model Parameter Values

First, as stated above, for face stream processing workflow, we referred to open and closed states of eyes and mouth as predictions from models instead of using EAR and MAR. So for the quantitative technical analysis section of this part of the project, we looked at the performance of MobileNet V2 over one epoch of training with a variety of different batch sizes to find the configuration that hits the sweet spot regarding training time, validation accuracy, and other metrics, as illustrated in the following table.

This paragraph discusses the results we saw on multiple important metrics when we changed the batch size used for training. We can see that training time decreases dramatically when the batch size increases; our training took 2,234.46 seconds with a batch size of 8 and 873.21 seconds with the batch size of 64. This is because processing multiple requests in a single batch improves computational efficiency. Looking at validation accuracy, a batch size of 32 yields the best accuracy of 86.85%, giving a good tradeoff

between the number of parameter updates and gradient stability. With batch size 64, it attains the least accuracy of 79.34%, suggesting potential overfitting or underfitting. This is echoed in the validation loss where the loss is the lowest at 3.36% with a batch size of 32. A batch size of 64 has the highest loss of 5.95%, which means learning is not happening properly. We see that the time to predict an image is fairly constant across the various batch sizes (0.0047 seconds to 0.0051 seconds), which is good for our real-time needs because we can expect a low latency that we know is fairly constant. For batch size 32, the false positive rate is lowest at 0.84%, which means that the number of incorrect positive predictions is less. Batch sizes 8 and 64 generate greater false positive rates, indicating more false alerts. The F1-score that balances precision and recall scores, with a batch size of 32, achieves the highest value of 85%, which indicates the best trade-off between precision and recall. This stems from the poor performance on the previous two settings where batch sizes 8 and 64 produced the lowest F1-scores.

Finally, the quantitative technical analysis performed in the section below reveals that a batch size of 32 is the best-performing configuration for MobileNet V2 - achieving the highest validation accuracy and lowest validation loss, false positive rate, and F1-score for a reasonable training time and prediction speed. With this configuration, we will be training a MobileNet V2 capable of performing fast and robust open vs closed eye detection.

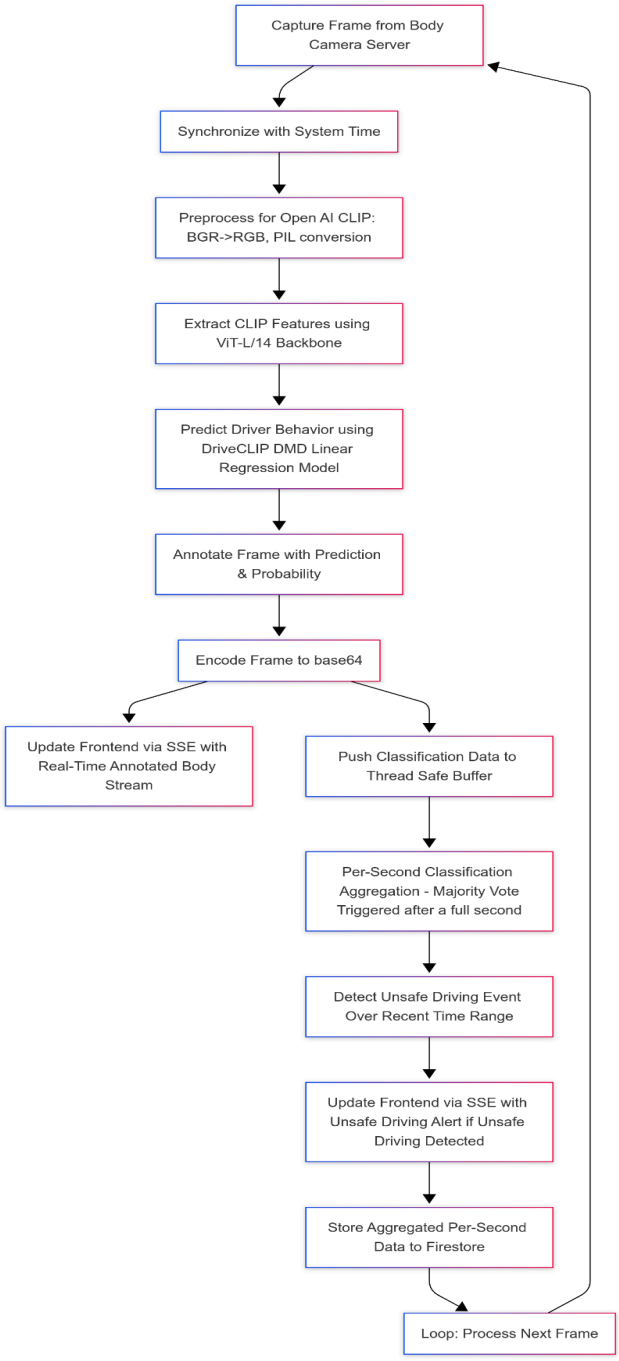
Table 9. MobileNet V2 Performance Over 1 Epoch of Training with Different Batch Sizes

Batch Size	Training Time (s)	Validation Accuracy (%)	Validation Loss (%)	Average Prediction Time Per Image (s)	False Positive Rate (%)	F1-score (Weighted Average) (%)
8	2234.46	82.79	4.46	0.0047	1.12	81
16	1484.87	85.63	3.60	0.0051	0.92	83
32	1084.92	86.85	3.36	0.0049	0.84	85
64	873.21	79.34	5.95	0.0049	1.49	75

3.4.3 Body Video Stream Processing

In this section, we describe the design and implementation of our Real-time Body Video Stream Processing workflow, which captures live video from an ESP32 CAM AI Thinker board positioned to view the driver from the side, processes each frame to classify driver activities using a CLIP-based Vision Image Transformer model, and synchronizes the results with other data sources (face video stream and OBD data) in the Fleet Vision platform. The figure below illustrates the overall flow of data in this subsystem.

Figure 5. Flow Diagram of Real-time Body Video Stream Processing Workflow



3.4.3.1 Workflow Overview

Similar to the face video pipeline, our Body Video Stream Processing workflow continuously fetches frames from an ESP32 CAM board running a camera web server at a URL such as http://esp32body.local/stream. We rely on mDNS (multicast DNS) to dynamically resolve the hostname esp32body.local, ensuring that even if the board's IP changes via DHCP, our subsystem can discover it at the start of each drive session provided both devices share the same local network.

The workflow proceeds as follows. First is Frame Capture and Timestamping where frames are pulled from the ESP32 CAM web server using OpenCV, and each is immediately assigned the current server time. Next is Preprocessing for CLIP where the system converts the frame from BGR to RGB and wraps it in a PIL Image object. A standard CLIP preprocessing pipeline (resizing, center-cropping, normalization) is then applied, preparing the image for feature extraction. Next Feature Extraction is performed using the CLIP model (Vision Transformer, ViT-L/14 backbone) to generate feature embeddings for each frame. These embeddings capture features and content in the image.Once we have the feature vector from CLIP, classification via DriveCLIP Linear Regression model is performed. This linear regression classifier was trained on the outputs of the CLIP model with the DMD dataset and presented in the DriveCLIP paper ("Vision-Language Models can Identify Distracted Driver Behavior from Naturalistic Videos", Zahid et al., 2023)[7]. The linear regression classifier is applied to the extracted CLIP features. This classifier assigns a probability distribution over a predefined set of driver behaviors including Driving Safely, Drinking or eating, Adjusting hair or makeup, Talking on phone, Reaching beside or behind, Talking to passenger, Texting or using phone and Yawning. After the model decides on a classification, the next step is Annotation & Encoding where the system overlays text on the frame to indicate the predicted activity and its probability. This annotated image is then encoded (Base64) for streaming to the Fleet Vision platform in real time via Server Sent Events. The final step is the Per-Second Aggregation Similar to the face stream. In this step, classifications are aggregated on a per-second basis. Multiple frames within the same second are tallied, and a majority vote produces the final classification for that second. This approach simplifies synchronization with face camera and OBD data, storing one classification per second in Firestore.

3.4.3.2 Design Choices

CLIP Model with ViT-L/14 Backbone: We use the CLIP (Contrastive Language–Image Pretraining) model using a ViT-L/14 vision backbone, which has proven effective for a variety of image understanding tasks. By converting each frame into high-level feature embeddings, CLIP enables our subsystem to recognize a broad range of driver behaviors with minimal overhead once the features are extracted due to only a simple linear regression layer being added to coerce the feature vector to a single output. In our original approach, we attempted to train lightweight CNN architectures such as Mobile Net V2 and V3 and Efficient Net in a variety of versions to be able to perform multi-label classification with a variety of different distraction types. Very significant effort was invested into this in the first term and it became clear that it wasn't working. We were able to get to a binary classification model using Mobile Net V3 Small that was able to reliably predict safe vs unsafe driving, by mostly looking at where the driver was looking and whether their hands were on the wheel or not. However a binary classification of safe vs unsafe driving is not that useful, and we were not able to get the multi-label classifier to work well in the real world even though it worked well on the validation set. With all of our in team model training efforts failing, CLIP really saved the day for us. It's a significantly more complex architecture which is required for a task as complex as ours like differentiating between actions including Talking on the Phone, Adjusting Hair and Makeup and Eating or Drinking which could all look quite similar. We believe our original attempts to get a lightweight CNN architecture to perform such a complex task were ill informed and with a Vision Image Transformer model like CLIP, we have found the right architecture and tool for the job.

GPU Acceleration: During testing, we found that performing CLIP inference solely on a CPU was too slow for real-time processing. By enabling GPU acceleration, we achieved roughly a 20× speedup, making CLIP viable within the body stream pipeline.

DriveCLIP Linear Regression Classifier: A linear regression classifier, trained on the DMD dataset (a large-scale collection of distracted driving scenarios) and presented in the paper "Vision-Language Models Can Identify Distracted Driver Behavior from Naturalistic Videos" (Zahid et al., 2023)[7], is used to map CLIP embeddings to driver activity labels. This approach significantly reduces inference time compared to training or inferring a separate deep network as linear regression classifiers are extremely fast and this one takes less than 1 ms. The output includes probabilities for each of the eight supported classes (like "Driving Safely," "Talking on phone," "Yawning").

Camera Hardware Selection: We observed that ESP32 CAM Wrover boards took nearly twice as long to capture each frame compared to our ESP32 AI Thinker boards (with antennas). Since capture time was the dominant factor in pipeline latency on the Wrover boards, we standardized on the AI Thinker boards for faster throughput and decided to use these instead of Wrover boards going forward.

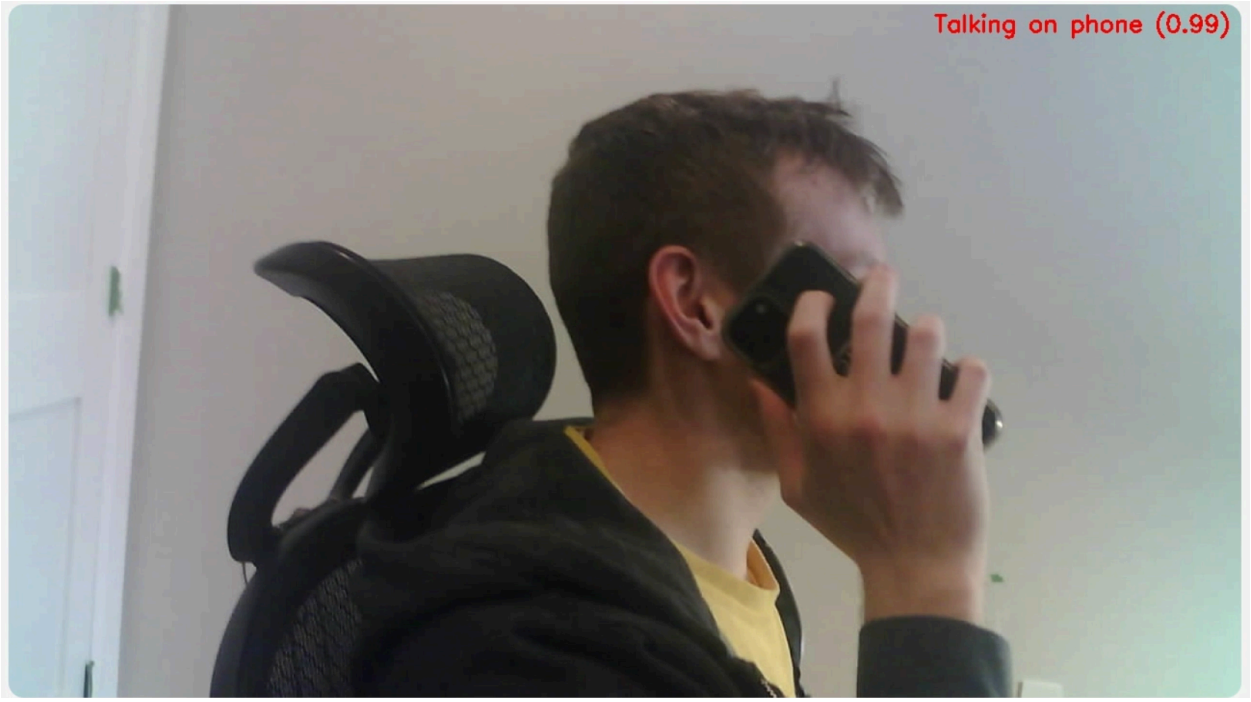
Non-Blocking Database Writes: Early on, we discovered that synchronous database writes in the pipeline introduced substantial delays, sometimes exceeding the model inference time. We updated the design to queue database writes asynchronously, preventing them from blocking the main pipeline. This change improved overall throughput without sacrificing data integrity.

Event Detection and Alerting: We use event detection so that we can send alerts to the web app in real-time if we ever detect unsafe driving so that it can be corrected immediately. If the model classifies "Yawning" for three consecutive seconds, we flag a yawning event. This aligns with the approach in "Multi-Index Driver Drowsiness Detection Method Based on Driver's Facial Recognition Using Haar Features and Histograms of Oriented Gradients"[7]. For high-risk tasks like texting, phone use, or reaching behind which usually draw the drivers eyes off the road, if the model consistently classifies these behaviors for more than 2 consecutive seconds, we generate a distraction event with the actual type of distraction we observed. For moderate-risk tasks like drinking/eating and talking with passengers, we apply a 3 second threshold before firing off an event. These durations were chosen based on the results presented in "An Analysis of Driver Inattention Using a Case-Crossover Approach On 100-Car Data: Final Report"[8], indicating that extended glances (>2s) or prolonged tasks significantly increase crash odds. Whenever an event is detected, it is sent to the web app using Server Sent Events which allows the event to be displayed in real-time and which should prompt the driver to correct their unsafe driving.

3.4.3.3 Example and Conclusion

In the figure below, the body camera feed is annotated with the predicted driver behavior ("Talking on phone (0.99)"), reflecting the output of the CLIP feature extraction and linear regression classifier. The probability (0.99) indicates a high confidence in the "Talking on phone" classification.

Figure 6. Body Stream Image Annotated with Classification



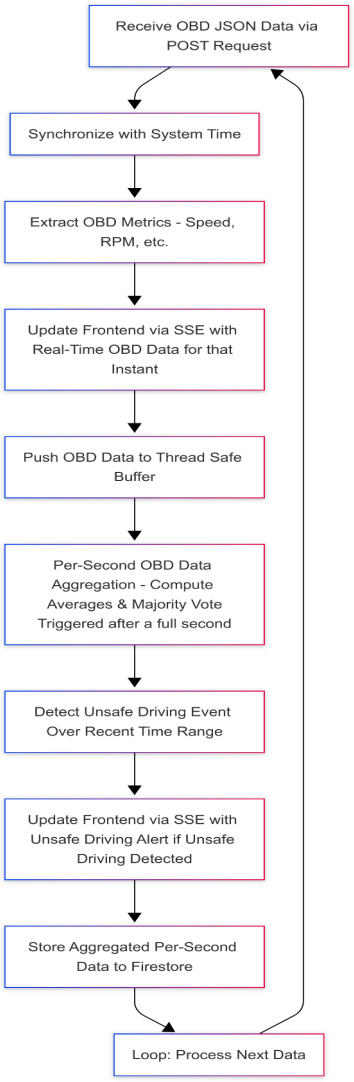
Leveraging mDNS based connectivity, a GPU accelerated CLIP model, and a linear regression classifier fine tuned on the DMD dataset, our Body Video Stream Processing subsystem monitors and classifies a wide variety of driver behavior in real time. By utilizing non-blocking writes to the database along with a schema designed for fast reads, our architecture allows for high throughput and fast inferencing of camera data, and by requiring events to exceed certain thresholds to trigger an alert we make sure that important alerts are generated when necessary. When combined with face camera data and OBD metrics, this workflow represents a comprehensive solution for driver distraction and drowsiness detection and mitigation that is integrated into the Fleet Vision platform.

Overall we have met the specification of detecting driver drowsiness and distraction with an accuracy of at least 90% and a false positive rate no higher than 5% with this workflow. We also meet our synchronization spec requirement of being able to synchronize input frame data from embedded subsystem components, run the appropriate Machine Learning models on all inputs, and generate appropriate outputs to accurately predict driver behaviour every second with at least 90% accuracy.

3.4.4 OBD Data Stream Processing

In this final section, we outline the OBD Data Stream Processing workflow, which collects live vehicle diagnostic information such as speed and RPM from an On-Board Diagnostics (OBD) unit. Unlike the camera-based workflows, this subsystem does not involve computer vision or additional machine learning models. Instead, it focuses on receiving OBD data, performing basic aggregation, and detecting unsafe driving speeds based on thresholds derived from traffic safety research. The figure below illustrates the overall flow of data in this subsystem.

Figure 7. Flow Diagram of Real-time OBD Data Stream Processing Workflow



3.4.4.1 Workflow Overview

Unlike the face and body cameras where our subsystem pulls frames from an ESP32 board the OBD data workflow reverses this relationship. Here, the ESP32 device reading OBD metrics (such as speed and RPM) from the OBD board posts JSON data directly to our server as quickly as possible. Because the server's IP address may change each time it joins a new network, we use ngrok to provide a stable, publicly accessible tunnel. The ESP32 OBD reader is programmed with the ngrok hostname so that it always reaches our locally running server, regardless of underlying IP changes.

Each OBD payload is immediately tagged with a server timestamp, ensuring that the data synchronizes with the face and body camera streams. Once received, the data is buffered for the current second and then averaged rather than using a majority vote to produce a single per-second record that is pushed to Firestore. In addition, the subsystem continuously monitors these averages to detect unsafe driving events (such as speeds exceeding 130 km/h for three seconds or a single-second speed of 150 km/h), generating real-time alerts via Server-Sent Events when thresholds are crossed.

3.4.4.2 Design Choices

ngrok for Consistent IP Addressing: Because our server IP changes whenever it joins a new network, we use ngrok to provide a stable tunnel endpoint. The ESP32 OBD board simply sends data to the ngrok URL, which forwards it to our local server, ensuring uninterrupted communication regardless of dynamic IP assignments when the server joins a new network as we are only running the server on a laptop now.

Per-Second Averaging: Instead of a majority vote, we average speed and RPM readings over each second to smooth out small and quick fluctuations. This approach helps us maintain more stable, representative metrics for continuous values like speed.

Event Detection and Alerting: We monitor the aggregated per-second data for dangerous driving conditions and with the OBD data that we currently have available, dangerous driving conditions entails high speed. If the average speed reaches 130 km/h for 3 consecutive seconds, an unsafe speed event is triggered reflecting findings from the “Global Approaches to Setting Speed Limits”[9] paper, which indicates a 37% reduction in reaction time at such speeds, and noting that 130 km/h exceeds legal limits in Ontario. Additionally, any single second reading of 150 km/h triggers a stunt driving event, as such speeds are classified as illegal stunt driving in Canada. Whenever these thresholds are exceeded, an immediate alert is sent to the Fleet Vision platform via Server-Sent Events (SSE) to warn the driver.

3.4.4.3 Example and Conclusion

Below is what shows up in the web app as the OBD data being streamed in real-time, showing speed, RPM, whether the “check engine” light is on and the current number of DTC codes. Each row corresponds to the result that was obtained for a given frame rather than the aggregate result across an entire second. You can see this as there are multiple frames with the same timestamp.

Figure 8. Real-time OBD Data Streaming

Timestamp	Frame #	Speed	RPM	Check Engine On	# DTC Codes
1740247402	64	26	2983	No	0
1740247402	63	25	2851	No	0
1740247401	62	22	2899	No	0
1740247401	61	19	2832	No	0
1740247400	60	20	2810	No	0

In summary, the OBD Data Stream Processing workflow integrates seamlessly with the Real-time Data Processing subsystem by providing continuous, timestamped vehicle diagnostics. It leverages ngrok for reliable server access, aggregates metrics on a per-second basis, and monitors for unsafe speeds that surpass Ontario legal limits. Combined with the face and body streams, this OBD pipeline completes our holistic approach to real-time driver safety and performance monitoring within the Fleet Vision platform.

3.5 Web Application Subsystem Design

The Fleet Vision Web Application is the main point of interaction between the end user and intricate Data Processing Subsystem. The Web Application is responsible for the display of driver analytical data and myriad vehicle metrics as well as providing a socket connection directly to theReal-time Data Processing Subsystem allowing for live video feed from the vehicle interior. Given the many features present in this subsystem it is imperative that a careful engineering design process was followed when designing the various components.

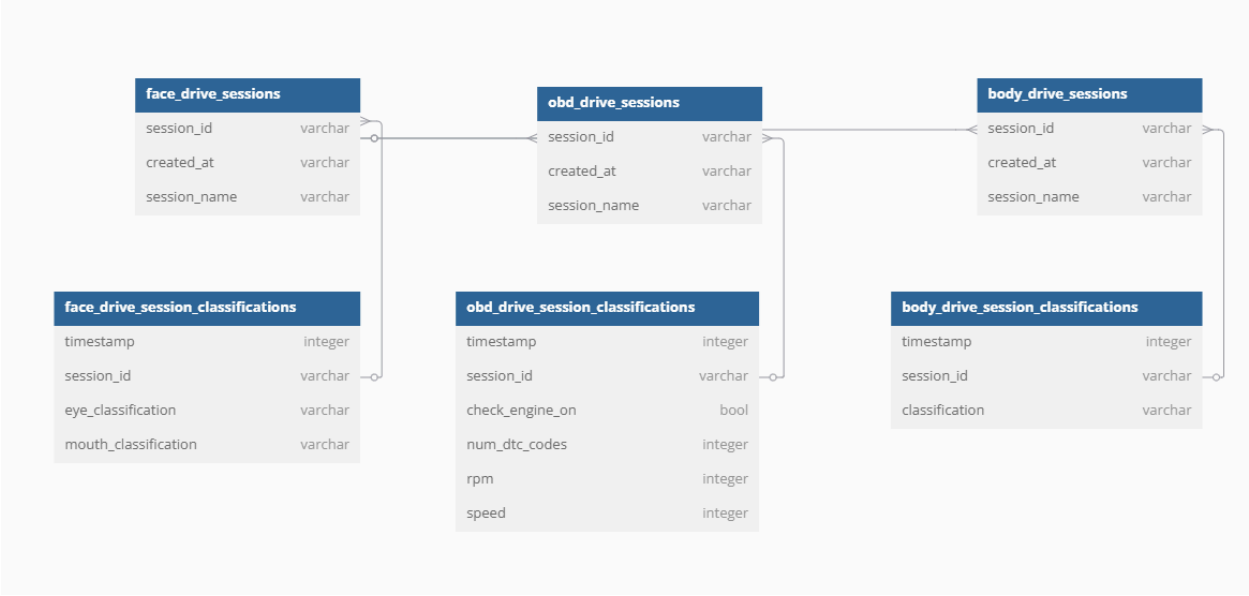
3.5.1 Fleet Vision Database

The Fleet Vision Database acts as the middleman when sending the vast majority data from the Data Processing Subsystem to the Web Application Subsystem since a single source of truth for the data is greatly beneficial as well as the future functionality for multiple organizations operating under the same database, separated by collections.

Due to the asynchronous nature of our OBD-II and ESP32-CAM streams the database acts as a buffer between the Real-time Data Processing Subsystem and the Web Application which allows both aforementioned datastreams to work at independent rates drastically reducing the computational complexity of synchronizing these isolated systems. The database contains three top level collections, or tables, that represent the three distinct types of data being recorded and later used for in-depth driver and vehicle analysis; these three collections are body_drive_sessions (BDS), face_drive_sessions (FDS), and obd_drive_sessions (ODS). As the name alludes each of these collections contains documents relating to distinct drive sessions which are all marked with a session_id, session_name, and creation timestamp so session information from all three streaming modules can be accurately paired for future use. Each of these top-level data tables contains an inner table that displays pertinent data relating to each top-level table such as computer vision model classification tags or detailed vehicle metrics. The database was structured with a focus placed on temporal analysis of our outputs from the computer vision models discussed in the Real-time Data Processing Subsystem, with our outputs grouped by sessions simplifying the process of accessing and analyzing driver and vehicle performance during a given session.

The database structure has been outlined in figure 5, highlighting the relationship between the top-level tables and their respective child tables. Figure 5 also shows how session_ids are used to relate information between the top-level tables.

Figure 9. FleetVision database structure



3.5.2 Web Application

The Fleet Vision web application is built using a combination of various web technologies to provide a modern experience for the user and ensure a robust development framework that can be easily expanded upon in the future. The primary technology used is Next.js, which is an optimized extension to the very popular React.js web development framework, and this is supplemented with a firebase integration to robustness to user authentication and session management.

3.5.2.1 Web Optimizations

As Next.js quickly gains popularity it is clear to see why this framework was chosen by the Fleet Vision Team for its host of built-in optimizations with the most substantial being the LCP image optimization, bundle analyzer, and lazy-loading capabilities. The LCP image optimizations are responsible for the bulk of the speed differences seen across the board between next.js and other popular web frameworks such as React.js. Vue and Angular and this is due to the impact of images on websites and images are handled in Next.js. From the Web Almanac, “Largest Contentful Paint (LCP) is the time from the start of the navigation until the largest block of content is visible in the viewport.”[5] and, as discovered in their 2022 UX report, nearly 50% of the LCP elements were images with only 4% of those images being stored in the WebP format, which is currently the fastest widely supported image format. Next.js remedies these short-comings by automatically serving formatted and scaled images in the WebP format which can load, on average, 30% faster [6] than the common PNG or JPEG formats while also reducing the amount of expensive layout shifts caused by dynamically sized images found in other frameworks. When implemented in tandem with the built-in bundle analyzer to identify large dependencies and modern lazy-loading technology to further reduce LCP times Next.js provides notable benefits over other web frameworks.

3.5.2.2 Web Application Design

The Fleet Vision web application consists of three main pages being the dashboard, vehicle analytics, and driver safety analytics which are all supplemented by other minor pages including profile, settings, signin/signup, etc.

3.5.2.3 Dashboard

The dashboard page is the first page the users see after signing in and this page contains a high level overview of the various analytics offered within this application. These analytics range from safety score distribution, which shows at a glance the average safety score and distribution, to urgent vehicle health metrics, showing the most urgent repairs needed across the entire fleet. This page was chosen as the home page of the application to allow administrative users to quickly get a measure of the entire fleet before viewing in-depth analytics.

3.5.2.4 Vehicle Analytics

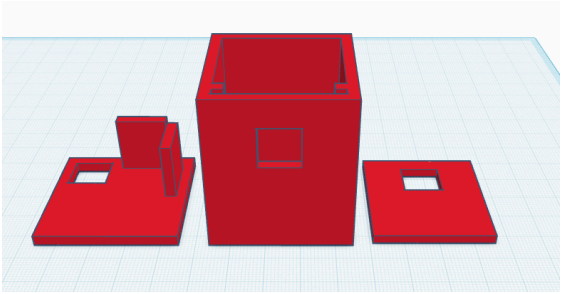
The vehicle analytics page displays current information such as speed, fuel, maintenance warnings, etc for all vehicles while also allowing the administrative user to search and apply filters to find a specific vehicle. A specific vehicle can be selected to view more relevant statistics and analytics from the vehicle such as the current driver and a timestamped log of all vehicle errors read from the OBD-II sensor.

3.5.2.5 Driver Analytics

The driver analytics page has a very similar structure to the vehicle analytics page but displays analytics relating to all drivers within the fleet while still maintaining the search and filter capabilities. Before a specific driver is selected the administrative user is able to see high-level analytics for all drivers in a table-like layout with some key analytics being the safety score, unsafe driving incidents and the drivers current vehicle. By selecting a specific driver, much like the vehicle analytics page, more in-depth analytics are displayed such as a timestamped log of all unsafe driver behaviours which are provided by the Computer Vision Models and Computer Vision Model Output Processing found in the Data Processing Server Subsystem. This page also contains a live camera feed from the inside of the drivers current vehicle, which is to allow administrative users to see what their drivers are doing in real-time. This is done using the brilliant ffmpeg javascript library which allows our Data Processing Server Subsystem to pass a stream-head to the Web Application subsystem which will pass data from the Video Capture Unit Subsystems.

4 Prototype Data

Figure 10. Encasing Design for Front and Side Camera Modules



Let's consider the design of the physical encasings for our prototype. Figure 10 shows the encasing for the front and side facing cameras. They will both use the same encasing, which has the dimensions of 41mm by 42mm by 46mm. These dimensions allow the box to hold the ESP32 CAM, a slider to attach the camera to for easy maintenance (right hand piece in figure 10a/10b), and the rails to hold the slider in place. The left hand piece in figures 10a/10b is the lid for the box, for which the hole in the corner is to allow the antenna for the ESP32 CAM to pass through.

Figure 11. Encasing Design for OBD Module

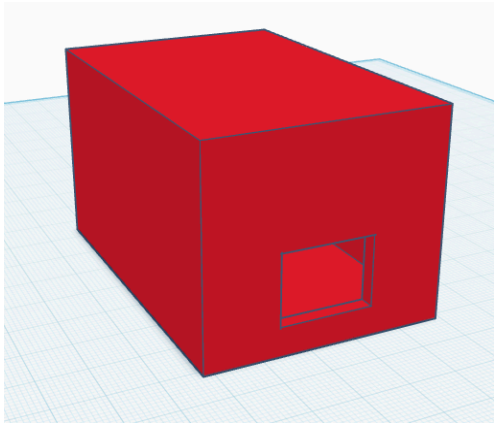


Figure 11 shows the box that we 3d-printed to use as the OBD module encasing. It has dimensions 51mm by 76mm by 44mm. These dimensions allow the combination of our OBD development kit and our ESP32 board, used for data transmission to server, to fit snugly within. The side that is not viewable in Figure 11 is kept open, as this is where the end of the OBD development will stick out to be plugged into the car's OBD port.

Figure 12. Printed Camera Module Encasing

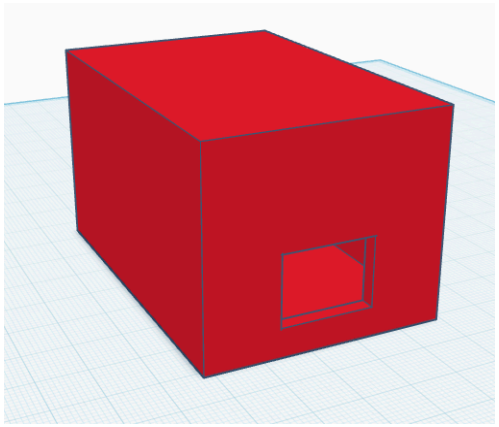
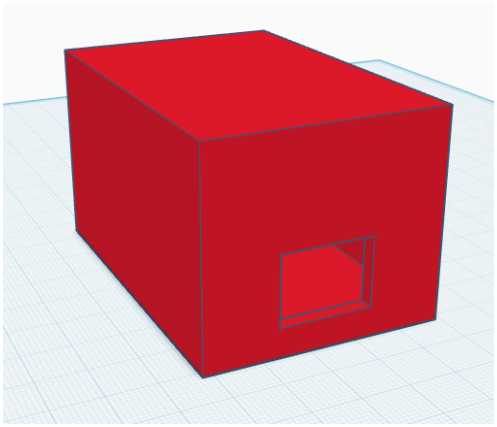


Figure 13. Printed OBD Module Encasing



(placeholders)

The material used for 3D printing these encasings is PETG. Another option was PLA, but PETG was used because it is a stronger material against external force. It has a higher melting point, so it was used with consideration to heat management. With all this considered, Figures 12 and 13 show the final printed camera module encasing and final printed OBD module encasing respectively, which fit within our final dimension requirements and allows all hardware in each module to fit compactly for an overall stable prototype.

5 Discussion and Conclusions

5.1 Evaluation of Final Design

Our final Fleet Vision design successfully delivers a cohesive platform for real-time driver monitoring and vehicle diagnostics, while meeting or exceeding the essential functional and non-functional specifications. By replacing the previously considered Particle Boron LTE with Wi-Fi-enabled ESP32 boards (supplemented by external antennas for stronger signal strength), we significantly reduced hardware complexity and cost, easily achieving the 5 FPS data transmission target at HD resolution. Meanwhile, the Longan Labs OBD-II kit streams vehicle data at roughly 2 FPS, surpassing the 1 FPS minimum requirement. Each embedded subsystem operates continuously for 12 hours on battery and fully recharges in 2 hours, fulfilling operational duration and recharge time constraints. Moreover, every component enclosure remains under 10×10×6 cm, installs in under 2 minutes, and requires no exposed wiring. Altogether, these enhancements align with the project's focus on safe, cost-effective solutions, ensuring reliable streaming of detailed driver and vehicle analytics for fleet managers.

Within the Video Capture Units subsystem, each ESP32 board streams frames at HD resolution (1280×720), achieving roughly 10 FPS during local tests, comfortably exceeding the 5 FPS minimum

requirement. Crucially, using external antennas resulted in a stronger, more stable Wi-Fi signal (around -57 dBm versus -72 dBm without an antenna), ensuring that frames can be relayed reliably, even when mounted inside an enclosure or at a distance from the access point. This design effectively captures both the driver's face and side views, enabling more accurate detection of critical factors.

For vehicle diagnostics, we adopted the Longan Labs OBD-II development kit, connecting it to an ESP32 board via TX/RX pins for a straightforward yet robust data link. With this approach, we consistently stream vital information such as speed, RPM, and engine warning status at approximately two frames per second, easily exceeding our original target of one frame per second. Our power testing indicated that while the OBD-II development kit can draw its supply directly from the car's OBD port, the attached ESP32 may require an additional regulated battery or power source to ensure reliable 5V 3A delivery in some vehicle configurations.

Data from both the video and OBD subsystems is routed to a publicly hosted Real-Time Data Processing server, where custom Python pipelines and computer vision libraries (OpenCV, dlib, etc.) handle face detection, landmark recognition, and classification tasks in real time. By focusing on a modular pipeline and keeping database writes non-blocking, we minimized latency, allowing our machine learning models to accurately detect events such as driver distraction, drowsiness, and engine faults without noticeable lag. The final outputs are then stored and synchronized in a Firestore database for historical analysis and delivered to the Next.js-based Fleet Vision web application.

Through these enhancements, the design meets or surpasses the project's functional and non-functional requirements. Each subsystem is relatively compact, supports at least 5–10 FPS where needed, and provides a consistent Wi-Fi link while fitting into enclosures that are minimally disruptive to the driver. The final arrangement—ESP32 boards for camera streaming and OBD data, a Python-based real-time server for machine learning, and a modern web application for analytics, fulfills the overarching goal of supplying both driver and vehicle insights on a unified platform.

5.2 Use of Advanced Knowledge

In developing Fleet Vision, we drew upon concepts from several advanced courses to guide both our hardware and software design decisions. From ECE 358 (Computer Networks), we applied fundamental principles of network communication, such as selecting appropriate transport and application layer protocols—to ensure reliable, low-latency data flow from our ESP32-based embedded components to the Real-Time Data Processing subsystem. Techniques like congestion and flow control, as well as the choice of HTTP over TCP, helped maintain in-order delivery of image frames and OBD data, preventing data loss or packet misordering.

Our experiences in ECE 457C (Intro to Reinforcement Learning) and MSE 446 (Intro to Machine Learning) directly shaped the design of our driver monitoring models. Specifically, we leveraged knowledge of neural network architectures, training methodologies, and quantitative performance tuning to identify key parameters (e.g., batch size, learning rates) that would yield accurate, real-time driver state classification. This ensured our facial and body tracking pipelines could detect signs of distraction or drowsiness with minimal false positives, even under varying lighting or camera angles.

Finally, ECE 356 (Database Systems) strongly influenced our decisions regarding data storage and retrieval in the Fleet Vision database. By applying concepts of database normalization and denormalization, we were able to streamline schema designs for storing both high-frequency OBD data and driver-related events. This approach not only reduced redundant data writes but also optimized query efficiency for real-time analytics in the web application. Taken together, these advanced principles allowed us to build a robust, scalable system that meets the real-time performance criteria and usability goals of Fleet Vision.

5.3 Creativity, Novelty, Elegance

A key source of innovation in Fleet Vision lies in our data pipelines, which synchronize video from two ESP32 cameras (face and body) and OBD-II data using mDNS discovery and non-blocking writes. By unifying these streams and matching timestamps to the second, we maintain a seamless flow of driver and vehicle data without overcomplicating the connection process, even after evaluating options like ngrok for remote connectivity.

We also transitioned away from large, custom ML models to more lightweight methods (e.g., Dlib for facial landmarks), reducing training and inference overhead while still meeting our 90% accuracy target in detecting driver distractions and drowsiness. Early attempts at building custom models required massive datasets and extensive GPU resources, complicating real-time deployment and causing exponential increases to development time. By pivoting to dlib-based pipelines, we avoided these constraints and ensured a smoother, faster path to real-world performance.

Finally, integrating a compact OBD-II kit with these camera feeds provides an all-in-one solution for both driver behavior analysis and real-time vehicle health metrics, which is a significant difference from typical fleet systems, which often handle these data types in isolation. The result is a more comprehensive, cost-effective, and user-friendly approach to fleet safety and performance.

5.4 Quality of Risk Assessment

The safety hazards related to Fleet Vision are primarily concerning the selected hardware components and the environment in which the device is used. The most significant safety hazards come from the power supplies and the environment of use, as well as overall reliability with fully custom trained models regarding resource requirements and correctness in operation.

Previously, we considered the possibility of the ESP32CAM boards requiring power to be supplied by rechargeable lithium ion batteries. As such, the hazard with this was to address the possibility of overheating. To mitigate this, we decided to use power banks as the sources of power instead of batteries, as the batteries did not offer any clear advantages when considering our prototype. This was originally planned to decrease the size of our overall camera encasing. This is no longer required, as we have opted for the route of having a singular power cord coming out of our encasing, which is completely covered in insulation, preventing the risk of consumers shocking themselves when handling this product. The ESP32CAMs, since they are meant to be placed in a snug encasing, were also considered at risk of overheating. To mitigate this, we have gotten small heat sinks to attach to the ESP32CAMs in order to absorb any accumulated heat while they are operating.

The environment poses significant safety hazards due to extreme temperatures and physical impacts. High outdoor temperatures can cause the ESP32CAM boards and power banks to overheat. Additionally, since the OBD-II container will be placed by the OBD-II port in vehicles (typically located under the steering wheel), drivers getting in and out of the vehicle, or just moving their legs while in the driver seat, may hit and damage the components. This could cause the components to to be displaced or get damaged in the containers, causing product failure, loose wires, short circuits, etc. To mitigate these risks, we opted to 3D print our encasing containers, which were printed using PETG. Another 3D printing material, PLA, could have also been used, but PETG was used because it is a higher quality material that allowed us to make a very durable container. It passed our various physical stress tests, such as appling high pressure on it, as well as impacts test through throwing/kicking it, essentially any possible physical actions a consumer could perform on the encasing. PETG, due to its higher melting point (PLA requires a printer to maintain approximately 190°C, compared to a higher required temperature of around 250°C for PETG), will also maintain its structural integrity better than an encasing made out of PLA when the electronics inside reach high operating temperatures.

Beyond hardware and environmental risks, we also encountered technical challenges with building custom ML models for distraction and drowsiness detection. These large, resource-intensive models required massive datasets and extensive tuning, yet often fell short on real-time performance in our prototype. Inconsistent detection could have jeopardized the system's reliability and user safety, especially if delayed or incorrect alerts were issued. To mitigate this risk, we pivoted to lighter, pre-trained solutions (e.g., Dlib for facial landmarks) that drastically reduced development overhead and performed dependably under real-world constraints. For instance, in local road tests with modest CPU hardware, our dlib-based approach consistently analyzed incoming frames at ~10 FPS—fast enough to issue driver-alert notifications without noticeable lag. This ensures the system's alerting and analytics remain both timely and accurate.

5.5 Student Workload

We concluded 498A with the following time contributions from each group member:

	Isaiah	**Aaron**	**Vithursan**	**Naumaan**
# of Hours	158	245	155	157

Considering this, along with the contributions during the past co-op term and the current duration of 498B, an accurate percentage based breakdown of each group's member's contributions is the following:

	Isaiah	**Aaron**	**Vithursan**	**Naumaan**
% of Contribution	~23%	~31%	~23%	~23%

All members, aside from Aaron, have approximately equal contribution to the overall prototype up to date. The reason for this disparity was because Aaron has been the primary group member in the preparation of the AI models. When considering the added on time that was required to research model

training, as well as the actual time spent in training and handling various errors and issues, the overall time contribution by Aaron was higher than the rest of the group members.

6 References

[1] “What is the true cost of vehicle downtime in your fleet?,” CerebrumX, [Online]. Available: https://cerebrumx.ai/what-is-the-true-cost-of-vehicle-downtime-in-your-fleet/

[2] Fleet Owner Staff, “Vehicle accidents cost companies $57B in 2017,” FleetOwner, [Online]. Available: https://www.fleetowner.com/safety/article/21702332/vehicle-accidents-cost-companies-57b-in-2017

[3] “Understanding the problem,” NHTSA, [Online]. Available: https://www.nhtsa.gov/book/countermeasures-that-work/distracted-driving/understanding-problem

[4] “CAN Bus Development Kit User Guide,” Longan Labs, [Online]. Available: https://docs.longan-labs.cc/1030003/#copy-and-paste-the-link-below-into-the-additional-boards-manager-urls-option-in-the-arduino-ide-preferences-file-preferences

[5] “ESP32-WROOM-32 Datasheet,” Espressif Systems, [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[6] E. Quiles-Cucarella, J. Cano-Bernet, L. Santos-Fernández, C. Roldán-Blay, and C. Roldán-Porta, "Multi-Index Driver Drowsiness Detection Method Based on Driver’s Facial Recognition Using Haar Features and Histograms of Oriented Gradients," *Sensors*, vol. 24, no. 17, p. 5683, Aug. 2024. doi: 10.3390/s24175683. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC11398282/

[7] M. Z. Hasan, J. Chen, J. Wang, M. S. Rahman, A. Joshi, S. Velipasalar, C. Hegde, A. Sharma, and S. Sarkar, "Vision-Language Models can Identify Distracted Driver Behavior from Naturalistic Videos," *arXiv preprint arXiv:2306.10159*, 2023. [Online]. Available: https://arxiv.org/abs/2306.10159

[8] S. G. Klauer, F. Guo, J. Sudweeks, and T. A. Dingus, "An Analysis of Driver Inattention Using a Case-Crossover Approach On 100-Car Data: Final Report," National Highway Traffic Safety Administration, Washington, DC, USA, Report No. DOT HS 811 334, May 2010.

[9] G. Forbes, "Global Approaches to Setting Speed Limits," presented at the Transportation Association of Canada Conference, Fredericton, NB, Canada, 2012. [Online]. Available: https://www.tac-atc.ca/wp-content/uploads/forbes.pdf