Automotive Monitoring System for Reducing Emissions and Improving Fuel Economy

William Griffin ([redacted])

The University of the West of England | Computer Science B.Sc.

GitHub link:
https://github.com/willgriffin111/Automotive-Monitoring-System

Abstract

For this project, a smart automotive monitoring system was developed to provide drivers with insights into their driving behaviour, with the aim of improving fuel efficiency and reducing emissions. The system collects real-time data using an OBD-II interface, GNSS module, and IMU sensor to monitor metrics such as vehicle speed, RPM, throttle position, location, and driving dynamics including harsh braking and acceleration.

This report documents the development of both the embedded system and a desktop application, built using the Svelte and Tauri frameworks. A literature review was conducted to explore existing automotive monitoring technologies. Insights from this research informed the system requirements and guided the overall design.

The system was then successfully implemented and thoroughly tested. The report concludes with an evaluation and reflection on the implementation process, discussing key outcomes and areas for future improvement.

Report Word Count:
Background section: 2,894 words
All other sections: 8,784 words
Total word count: 11,678 words

Acknowledgement

I would like to acknowledge the support and guidance of my supervisor, [redacted], as well as the encouragement from my friends and family throughout the development and documentation of this project.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Automotive transport is crucial but it brings financial and environmental costs. As car numbers rise, pollution and fuel consumption also increase, impacting budgets and carbon footprints. Improving driving habits such as avoiding rapid acceleration, harsh braking and idling reduces these effects.

Technologies including OBD-II, GNSS, and IMUs help drivers monitor and adjust behaviour to lower fuel consumption and emissions. OBD-II provides real-time engine data, GNSS offers location insights and IMUs detect aggressive driving events.

This report explores the design, implementation and testing of a device that improves fuel economy by providing drivers with actionable performance metrics. By fine-tuning habits, users can reduce fuel consumption, lower emissions and reduce fuel costs.

1.1 Aims

The project aims to develop a device that connects to a vehicle's OBD-II port to monitor driver behaviour and vehicle performance. The device will use engine data, GNSS, and IMU sensors to generate reports that help improve driving habits and fuel economy. The results will be displayed in a user-friendly app.

1.2 Objectives

These are the broad objectives for the project:

1 Gather real-time engine data from the OBD-II port, track location with GNSS, and monitor driving dynamics with an IMU.
2 Process the data to identify driving patterns that impact fuel efficiency and vehicle performance.
3 Generate reports with actionable insights to help drivers optimise fuel economy.
4 Create an app to present the reports and feedback to the driver.

2 Literature Review

This literature review provides a foundation of knowledge for the technologies used in this project. It starts with existing automotive monitoring systems, moves on to the background of the problem and concludes with an overview of the relevant software frameworks.

2.1 Existing Automotive Monitoring Systems

Modern automotive monitoring systems use various technologies to track vehicle performance and driver behaviour. These systems collect real-time data to improve safety, fuel efficiency, and vehicle emissions. Key automotive monitoring technologies include On-Board Diagnostics II (OBD-II), Global Navigation Satellite System (GNSS) technology, and Inertial Measurement Units (IMUs). Below is an overview of some of the primary systems in use today.

2.1.1 OBD-II

“The On-Board Diagnostic System (OBD II) was established in the United States in 1996 by the Society of Automotive Engineers (SAE) to standardise vehicle emissions monitoring and ensure Environmental Protection Agency (EPA) compliance. These standards mandate a 16-pin OBD II port in vehicles, through which diagnostic data from the Electronic Control Unit (ECU) is accessible. This development led to OBD II scanning tools, which interface with vehicles via this port to monitor parameters like fuel efficiency, oxygen sensor outputs, and vehicle speed”, as defined by Malekian R. et al. (2017).

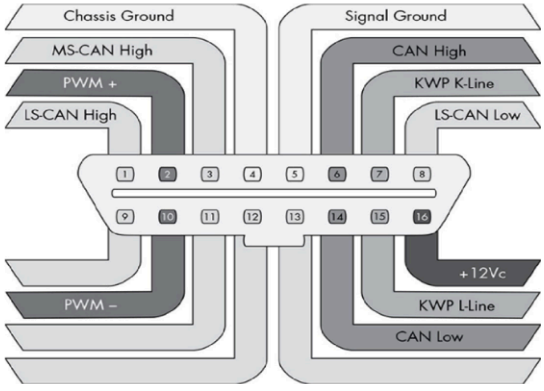


Figure 1 Data link connector. Kumar, R. & Jain, A. (2022)

The OBD-II system operates through a standard diagnostic port, typically under the dashboard, which connects directly to the vehicle’s Electronic Control Unit (ECU). This port provides a standardised communication protocol. Malekian R. et al. (2017) said, “A scanning tool typically requests information from the ECU by sending a message containing a hexadecimal code

associated with a specific parameter. The message would then get interpreted according to one of five OBD II signalling protocols". The standard from Anon (2002) SAE Technical Standard states that the protocols are SAE J1850 (VPW and PWM), ISO 15765, ISO 1941-2 and IS0 142300-4. These protocols determine how data is exchanged between the ECU and diagnostic tools, ensuring compatibility across various vehicle manufacturers.

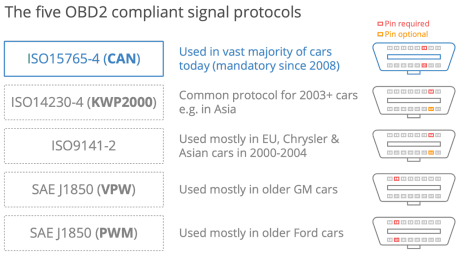


Figure 2. OBD-II Car manufactures communication protocols. (CSS Electronics, 2024)

OBD-II communicates with the ECU using two main code types. As stated by Rimpas et al., 2020 these are "1. Diagnostic Trouble Code (DTC): Each code is used to describe an issue, for example, Pxxxx is a powertrain code error and Cxxxx a chassis error. 2. Parameter ID (PID): Codes used to require data from the ECU, like RPM in idle speed." Interpreting these PID values enables driving behaviour monitoring, including acceleration, fuel efficiency, and speed patterns, offering analysis of vehicle performance and driving style.

Mode Number	Mode Description
01	Current Data
02	Freeze Frame Data
03	Diagnostic Trouble Codes
04	Clear Trouble Code
05	Test Results/Oxygen Sensors
06	Test Results/Non-Continuous Testing
07	Show Pending Trouble Codes
08	Special Control Mode
09	Request Vehicle Information
0A	Request Permanent Trouble Codes

Table 1: OBD Commands. (SparkFun Electronics, no date.)

The following table provides examples of commonly accessed Parameter IDs (PIDs) for Mode 01 (Current Data), used in OBD-II systems to retrieve real-time vehicle diagnostics.

PID Code	**Data Retrieved**	**Description**
0C	Engine RPM	Shows the current engine speed
0D	Vehicle Speed	Indicates the speed of the vehicle
0F	Intake Air Temperature	Measures the temperature of air entering

11	Throttle Position	Indicates the throttle opening percentage
1F	Run Time Since Engine Start	Time engine has been running since start
2F	Fuel Level Input	Displays the level of fuel
31	Distance Travelled with Malfunction Lamp (MIL) On	Total distance driven with MIL on
46	Ambient Air Temperature	Shows the ambient air temperature
5C	Engine Oil Temperature	Displays the temperature of engine oil

Table 2: Example PID codes

While OBD-II is designed to standardise diagnostics across different vehicle manufacturers, this can lead to compatibility challenges. Different manufacturers use multiple OBD-II protocols, which means certain OBD-II scanners may not work universally across all vehicles.

Sim and Sitohang (2014) describe a widely adopted solution for interfacing with OBD-II systems: “All five protocols of OBD-II can be implemented in hardware or software. The most popular implementation of these communication protocols is hardware-based, using the ELM327 chip. The ELM327 chip translates OBD-II communication signals into simple text data, which can be accessed through an RS232 port.” The ELM327 makes interfacing with OBD-II systems straightforward, as it converts complex protocols into readable text and supports all five OBD-II standards.

2.1.2 Global Navigation Satellite System

Global Navigation Satellite System (GNSS) technology calculates a device's location by triangulating signals from multiple satellites. A GNSS module in a vehicle constantly refreshes the location data with any change in position and speed. This information can be stored and later processed to map the vehicle's route. Moreover, combining GNSS with Onboard Diagnostics can offer additional insights, such as whether a driver is speeding or idling the vehicle in traffic.

Multiple GNSS technologies, such as GPS (American), GLONASS (Russian), Galileo (European), and BeiDou (Chinese), provide precise positioning services. As Cai and Gao (2012) state, “the precise point positioning (PPP) technique is mainly limited to the use of GPS measurements.” This project will primarily use GPS due to its global availability and precision.

GNSS has limitations, and one major concern is its accuracy. Tang et al. (2015) stated, “The quality of tracking data is often low due to the geometrical effects of urban canyons on the accuracy of GPS ranging, thereby causing tracking points to deviate from the original roads.” This is quite common in mountainous and heavily built-up areas.

A technique now known as Dead Reckoning (DR) has been developed to deal with this. As stated by Nascimento et al. (2018), “The process of transforming the measurements from the

sensor of the vehicle into an estimated position is called Dead Reckoning (DR). The localisation DR technique computes the current position of the vehicle based on its last location estimation." This method uses data from additional sensors, commonly IMUs, to estimate the vehicle's position by calculating its speed, direction, and elapsed time since the last known position.

DR is useful in environments with unreliable GNSS signals, such as tunnels or dense urban areas. Once GNSS becomes available again, it recalibrates the estimated position to correct potential drift errors, as shown below.

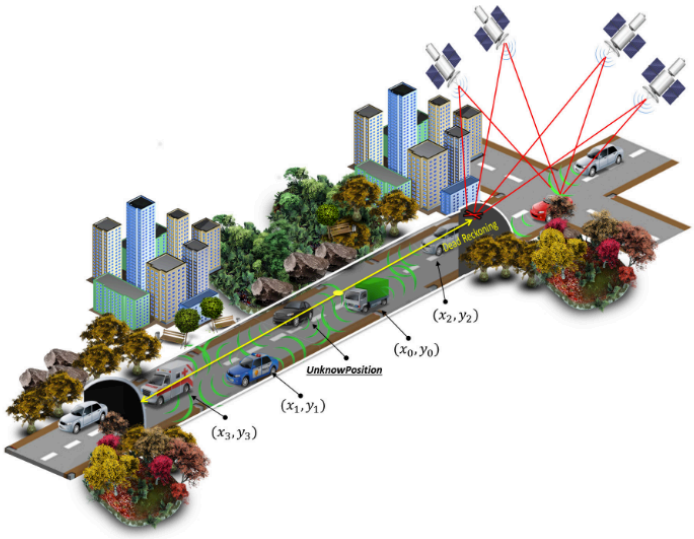


Figure 3: Dead reckoning illustration. Nascimento et al. (2018)

2.1.3 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) comprises accelerometers, gyroscopes, and magnetometers that can measure 9 degrees of freedom. For automotive tracking, IMUs can track the forces on the vehicle, including harsh acceleration, decelerations, and cornering.

One of the main challenges with IMUs is sensor drift. As Zhao (2018) states, "IMUs suffer from integration drift. Small errors appear during the measurement of acceleration and angular velocity, and they will progressively be integrated into larger errors in velocity and angle, which are compounded into still greater errors in position."

A solution involves using algorithms to mitigate drift. One widely used approach is the Kalman Filter and Extended Kalman Filter, which combines data from multiple sensors to produce accurate and reliable state estimates. As Franklin (2020) states, the Kalman Filter can "Use Inertial Measurement Unit (IMU) sensors to estimate an object's location, velocity, and acceleration; and use those estimates to control the object's next moves.". Przybylski (2024) noted that the difference between the two types of Kalman Filter is that the "Standard Kalman

filter is used for linear data only." As vehicles navigate in a three dimensional environment, an Extended Kalman filter would be the correct choice.

For instance, Zhao (2003) has demonstrated the benefit of integrating GNSS data with Dead Reckoning using an Extended Kalman Filter. Field trials conducted within Greater London achieved positioning errors below 90%. Extended Kalman Filter can maintain positioning accuracy within 50 meters 99.9% of the time, whereas only 90% coverage could be achieved using only GNSS.

2.2 Economic and environmentally friendly driving style

Driving behaviour significantly affects fuel consumption, with aggressive driving patterns often leading to inefficient fuel usage. Factors such as rapid acceleration, frequent braking, and excessive idling can increase fuel consumption. This is backed by Ma, Y. & Wang, J. (2022): "Correlation results indicate that the differences in fuel consumption incurred by various driving behaviours, even in the same traffic conditions, can be as much as 29% for a light-duty truck and 15% for a passenger car."

Starting with harsh acceleration and frequent braking, these are the most impactful behaviours on fuel consumption. Rapid acceleration requires the engine to increase its workload, using more fuel. According to Fuel Economy of Road Vehicles (2012), avoiding unnecessary acceleration and deceleration can "save between 2% and 10% of fuel". Research by Ziakopoulos, A. (2024) highlights the importance of such practices, stating that "higher traffic occupancy, traffic speed, and gradient values increase the probability of harsh braking (HB) occurrence."

Driving at high speeds significantly increases fuel consumption due to increased engine load and aerodynamic drag, which require more fuel to overcome. According to Fuel Economy of Road Vehicles (2012), "as speed rises above 90 km/h (about 56 mph), fuel economy decreases rapidly".

Excessive idling, including waiting in traffic or leaving the engine running while stationary (e.g., level crossing), also increases fuel economy as the engine is using unproductive fuel. In a study by Rahman, et al. (2013, p.10), they found that: "In the USA only, idling causes more than $1 billion increase in fuel consumption per annum." and "Fuel consumption during idling was as high as 1.85 gal/h.". To reduce these effects, engines should be switched off where possible. Additionally, many modern vehicles have 'Start/Stop Technology' that automatically shut off the engine when stationary, reducing fuel waste and emissions.

Another significant factor contributing to reduced fuel economy is using air-conditioning systems in vehicles. A study by Lee, J. et al. (2013) highlighted the substantial impact of the air-conditioning compressor on fuel consumption, stating that "the most important component affecting the fuel consumption is the compressor, which caused a fuel consumption increase of 77–89%." This increase is from the energy required to power the compressor, which places an additional load on the engine, especially during high ambient temperatures.

By incorporating these techniques, drivers will increase fuel consumption and reduce emissions. For instance, an AA study in 2017 found that drivers "saved an average of 10% on their weekly fuel bills. The best saved an impressive 33%."

2.3 Fuel Consumption and Emissions: Financial and Environmental Consequences

According to the Department for Transport (2024), "motor vehicle traffic on Great Britain's roads increased by 2.2% between 2022 and 2023, reaching 330.8 billion vehicle miles". Additionally, a report by Yurday (2024) states that the average fuel efficiency for cars in the UK is "38.8 miles per gallon".

Based on these figures, it can be estimated that approximately 8.53 billion gallons of fuel were consumed on UK roads in 2023 (calculated from Department for Transport data). The average fuel price in the UK for 2023 at £6 per gallon (Department for Energy Security and Net Zero, 2024) translates to a total national fuel expenditure of around £51.18 billion. For the average UK driver, who covers 7,000 miles annually (Department for Transport, 2024), this amounts to approximately 191 gallons of fuel consumed each year, costing around £1,082 in fuel expenses. These statistics show the significant financial burden of vehicle use for individuals.

High fuel consumption not only places a financial burden on individuals but also significantly contributes to environmental issues such as carbon dioxide emissions and climate change. In 2021, domestic transport in the UK was responsible for emitting 109 million tonnes of carbon dioxide equivalent (MtCO2e), making transport the largest emitting sector, contributing 26% of the country's total emissions (Department for Transport, 2023). Cars alone contributed to 57% of transport emissions. Therefore, improving fuel efficiency is both an economic and environmental priority.

2.4 Technology Stack

In this section, I will discuss the technology stack of the project.

2.4.1 Hardware Platform: Adafruit QT Py ESP32-S2

The Adafruit QT Py ESP32-S2 is a compact microcontroller designed for projects requiring Wi-Fi connectivity, sensor integration, and data handling. According to Adafruit (2021), "QT Py comes with 4 MB flash and 2 MB PSRAM, allowing it to buffer massive JSON files for parsing." This makes it well-suited for applications requiring large data storage and processing. Additionally, Adafruit highlights its communication options: "Two I²C ports - one on the breakout pads and another with a STEMMA QT plug-and-play connector - along with hardware UART and hardware SPI available on high-speed SPI peripheral pins." These features ensure

seamless integration with various sensors, making the QT Py ESP32-S2 suitable for automotive applications.

The QT Py ESP32-S2 is the central controller managing data from the OBD-II system, GNSS module and IMU sensor. It processes real-time data on vehicle performance, location tracking and driving behaviour while having the capability to store the collected information on an SD card for later analysis. The board's Wi-Fi connectivity allows for remote monitoring and data transmission. It's compact size ensures it can be discreetly installed in tight spaces within the vehicle, while its low power consumption minimises strain on the car's electrical system.

2.4.2 Software Frameworks

In this subsection, I will discuss the software frameworks of this project.

2.4.2.1 Front-End Framework: Svelte

Front-end frameworks are critical in developing scalable and maintainable user interfaces. Timbó (2023) states, "Front-end frameworks are sets of pre-written code that provide developers with a scalable and maintainable structure for creating user interfaces more efficiently. They contain HTML, CSS and JavaScript components that developers can reuse in other projects, helping to keep the codebase consistent and organised." Svelte distinguishes itself by compiling components at build time, resulting in minimal runtime overhead.

In the context of automotive monitoring systems, Svelte is suitable for data-intensive applications, as backed up by MDN contributors (2024): "If you are building data-visualizations that need to display a large number of DOM elements, the performance gains that come from a framework with no runtime overhead will ensure that user interactions are snappy and responsive." In automotive applications, these features are vital for presenting large quantities of data, such as GPS routes or fuel efficiency metrics.

2.4.2.2 Cross-Platform Application Framework: Tauri

Tauri is an emerging framework designed to create lightweight, secure desktop applications. Unlike traditional frameworks such as Electron, Tauri leverages native WebView technology, significantly reducing the application footprint. Tauri uses the operating system's default webview rather than bundling Chromium with the application. According to the Aptabase Team (2023), "This means that on macOS, your app will use WebKit (Safari's engine), on Windows, it'll use WebView2 (based on Chromium), and on Linux, it'll use WebKitGTK (same as Safari's). The end result is an extremely small app that feels super fast."

Tauri's design and security features make it ideal for building applications for automotive monitoring systems. Its efficiency on low-processing-power devices and cross-platform capabilities allow for complete deployment.

2.4.2.3 Mapping Framework: Leaflet

Interactive mapping is vital in automotive monitoring systems because it visualises the vehicle's route. Swadia (2021) states, “Leaflet.js is a popular, lightweight, and open-source JavaScript library for creating interactive maps that work well across major desktop and mobile platforms.”

It can also be expanded with plugins to add overlays for various uses, such as showing speed limits. Its functionality is further enhanced by integrating with external data sources such as the OpenStreetMap or Overpass APIs. These integrations allow for dynamic fetching of speed limits along routes and offer real-time alerts if the driver exceeds the speed limit.

2.6 Data Processing

Data processing is essential for transforming raw data into actionable insights. This section explores data storage and calculating fuel consumption.

2.6.1 Data Storage

Efficient data storage is crucial for managing the large volumes of data generated during vehicle operation. JSON (JavaScript Object Notation) is widely used due to its lightweight nature, readability, and compatibility with modern tools. According to the ECMA-262 3rd Edition standard (1999), “JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.” It has two primary structures - “a collection of name/value pairs” and “an ordered list of values” - making it suitable for efficiently organising OBD-II, GPS, and IMU data.

2.6.2 Calculating Fuel Consumption

Fuel efficiency can be calculated using data from the Mass Air Flow (MAF) sensor and the Vehicle Speed Sensor (VSS). As Kwiatkowski (2023) explains, “Modern vehicles use oxygen sensors to maintain an air-to-fuel ratio of 14.7:1, representing 14.7 grams of air for every gram of fuel.” He further states, “Most vehicles do not have a fuel flow sensor; therefore, you need to use the Mass Air Flow (MAF) sensor and the Vehicle Speed Sensor (VSS) to calculate miles per gallon (MPG).”

The MAF value, measured in grams per second, is converted into gallons per hour (GPH). According to Kwiatkowski (2023), “Divide the MAF by 14.7 to get grams of gas per second, then by 454 for pounds per second, and by 6.701 for gallons per second. Multiply by 3600 to get gallons per hour. The math expression for GPH is: MAF * 0.0805.”

Simultaneously, VSS data, given in kilometres per hour, is converted into miles per hour (MPH) using a factor of 0.621317. As Kwiatkowski (2023) notes, “To calculate MPG, divide MPH by GPH. The final math expression for MPG will be: VSS * 7.718 / MAF.”

The formulas for these calculations are (Kwiatkowski, 2023):

$$GPH = MAF * 0.0805$$

$$MPH = VSS * 0.621317$$

$$MPG = \frac{VSS*7.718}{MAF}$$

When adapting Kwiatkowski's (2023) formula for calculating average MPG over an entire journey, the formula becomes:

$$MPG = \frac{\sum(VSS*0.621317*\Delta t)}{\sum(MAF*0.0805*\Delta t)}$$

3 Requirements

In this section, I will state the requirements for this project using the MoSCoW method.

As stated by Tudor & Walter (2006) defines MoSCoW as: “M(ust have): essential; S(hould have): important but not critical; C(ould have): beneficial but non-essential; W(ont have): discussed but not included.”

3.1 Functional Requirements

Requirement ID	Priority (MoSCoW)	Description	Research (If applicable)
FR-01	Must	The embedded system must gather real-time engine RPM (Revolutions per Minute), Speed (KPH and MPH), throttle position, and MAF (Mass Airflow sensor) from the OBD-II port.	See literature review (Section 2.1.1).
FR-02	Must	The embedded system must calculate real-time and average fuel consumption (MPG).	As backed up by Wikiapedia (2024), “Miles per gallon (mpg) are commonly used in the United States, the United Kingdom, and Canada”
FR-02	Must	The embedded system must track vehicle location and time.	See literature review (Section 2.1.2).
FR-03	Must	The embedded system must be able to calculate vehicle position without a satellite signal.	See literature review (Section 2.1.2).
FR-04	Must	The embedded system must gather data on forces like harsh braking, acceleration and cornering.	See literature review (Section 2.1.3).
FR-05	Must	The embedded system must log gathered data on an SD card in JSON format.	See literature review (Section 2.6.1).

FR-06	Must	The embedded system must have a button to activate and deactivate logging.	N/A
FR-07	Must	The embedded system must have a way to indicate when calibration/initialisation is complete and if logging is active.	N/A
FR-08	Must	The app must display the past route driven on a map with points of inefficient driving behavior marked.	See literature review (Section 2.4.2.3).
FR-09	Must	The app must provide graphs of collected data for analysis.	As stated by Eberhard, K., (2023), "The strengths of data visualization lie in its ability to reduce cognitive load, allowing users to process and interpret large volumes of information quickly and accurately."
FR-10	Must	The app must display the real time driving metrics on a map with points of inefficient driving behavior marked.	See literature review (Section 2.4.2.3)
FR-11	Must	The app must be able to manage the files saved on the SD Card.	See literature review (Section 2.6.1).
FR-12	Must	The app must be able to display the speed in KPH and MPH.	N/A
FR-13	Should	The embedded system should transmit live real-time data for remote monitoring.	N/A
FR-14	Could	The interface could cache the map for offline uses.	N/A

FR-15	Could	The app could provide visual and auditory alerts for speeding or harsh driving.	N/A
FR-16	Won't	The app won't integrate with third-party apps for data sharing (e.g., Google Maps).	N/A

Table 3: Functional Requirements

3.2 Non-functional Requirements

Requirement ID	Priority (MoSCoW)	Description	Research (If applicable)
NFR-01	Must	The app must have an intuitive and responsive user interface.	N/A
NFR-02	Must	The app should be compatible with macOS and Windows. [1]	See literature review (Section 2.4.2.2).
NFR-03	Must	The system should be compatible with all OBD-II protocols. [2]	The protocols have been outlined in the literature review (see Section 2.1.1).
NFR-04	Must	The system must be housed in a durable enclosure.	N/A
NFR-05	Must	The app must follow GDPR (General Data Protection Regulation).	HM Government (n.d.) notes users have rights to know how data is used, access it, correct inaccuracies, request deletion, and restrict processing.

[1] NFR-02 was empirically tested using my personal macOS laptop and university Windows desktop machines.
[2] NFR-03 was not fully tested (Addressed in evaluation)

NFR-06	Must	The enclosure must have a secure mounting mechanism.	N/A
NFR-07	Must	The system must store and retain data even in case of power loss.	N/A
NFR-08	Should	The system should transmit live data within 3 second latency.	N/A
NFR-09	Should	The enclosure should include a cooling mechanism, such as a fan and heat dissipation vents, keeping the internal temperature below 85 degrees celsius.	As stated by Microchip Technology Inc (2010), OBD Solutions (2012) and u-blox (2018) the maximum operating temperature is “85 degrees celsius.”
NFR-10	Could	The data stored on the SD card could be encrypted.	N/A
NFR-11	Won’t	The app should integrate voice feedback for real-time driving guidance.	N/A

Table 4: Non-functional Requirements

3.3 Use case Diagram

A use case diagram is a graphical depiction of a user's possible interactions with a system. (Wikipedia, no date).

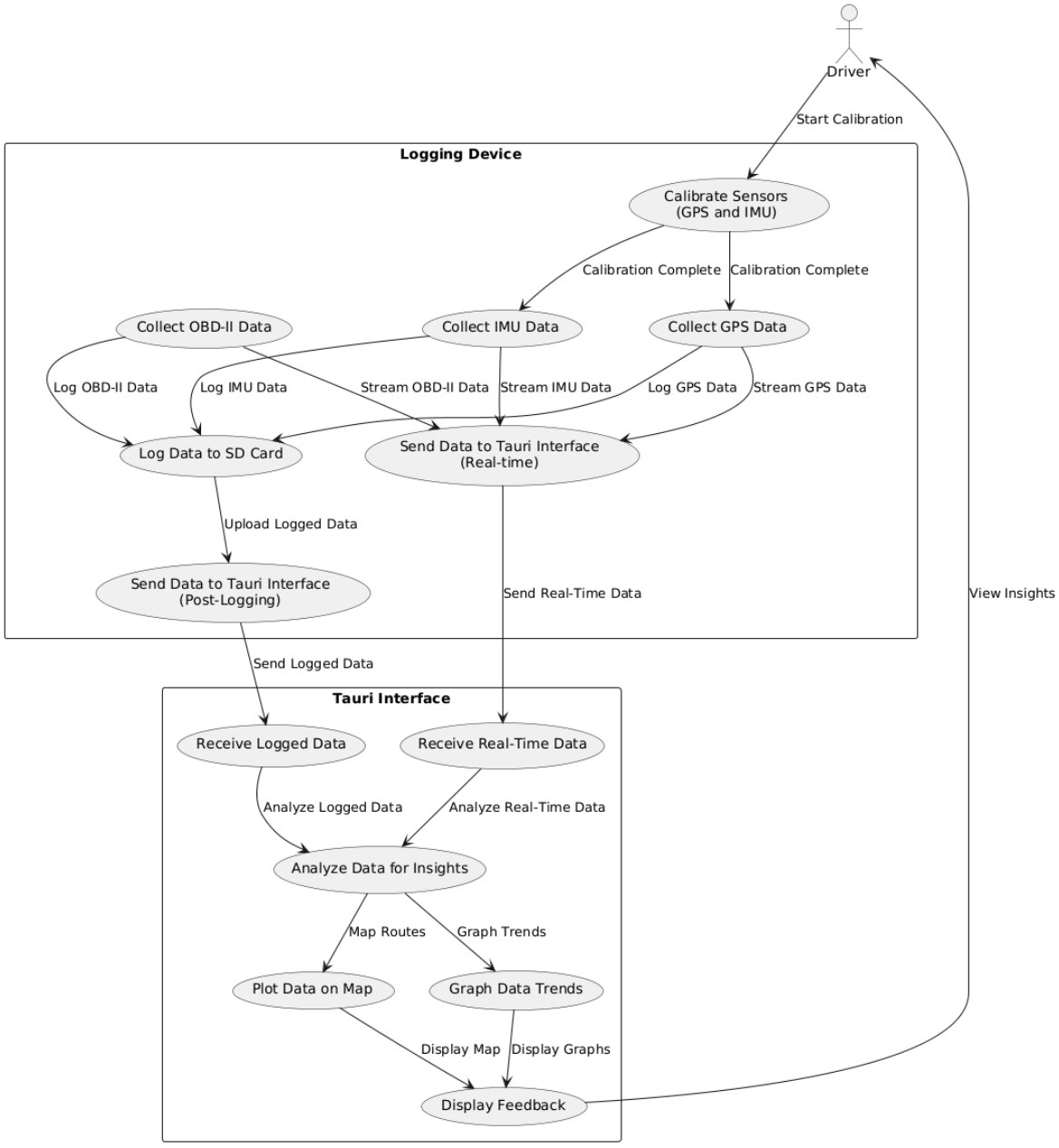


Figure 4: Use case Diagram

3.4 Activity Diagram

An activity diagram shows system behavior by describing action sequences (IBM, 2021).

3.4.1 Embedded System

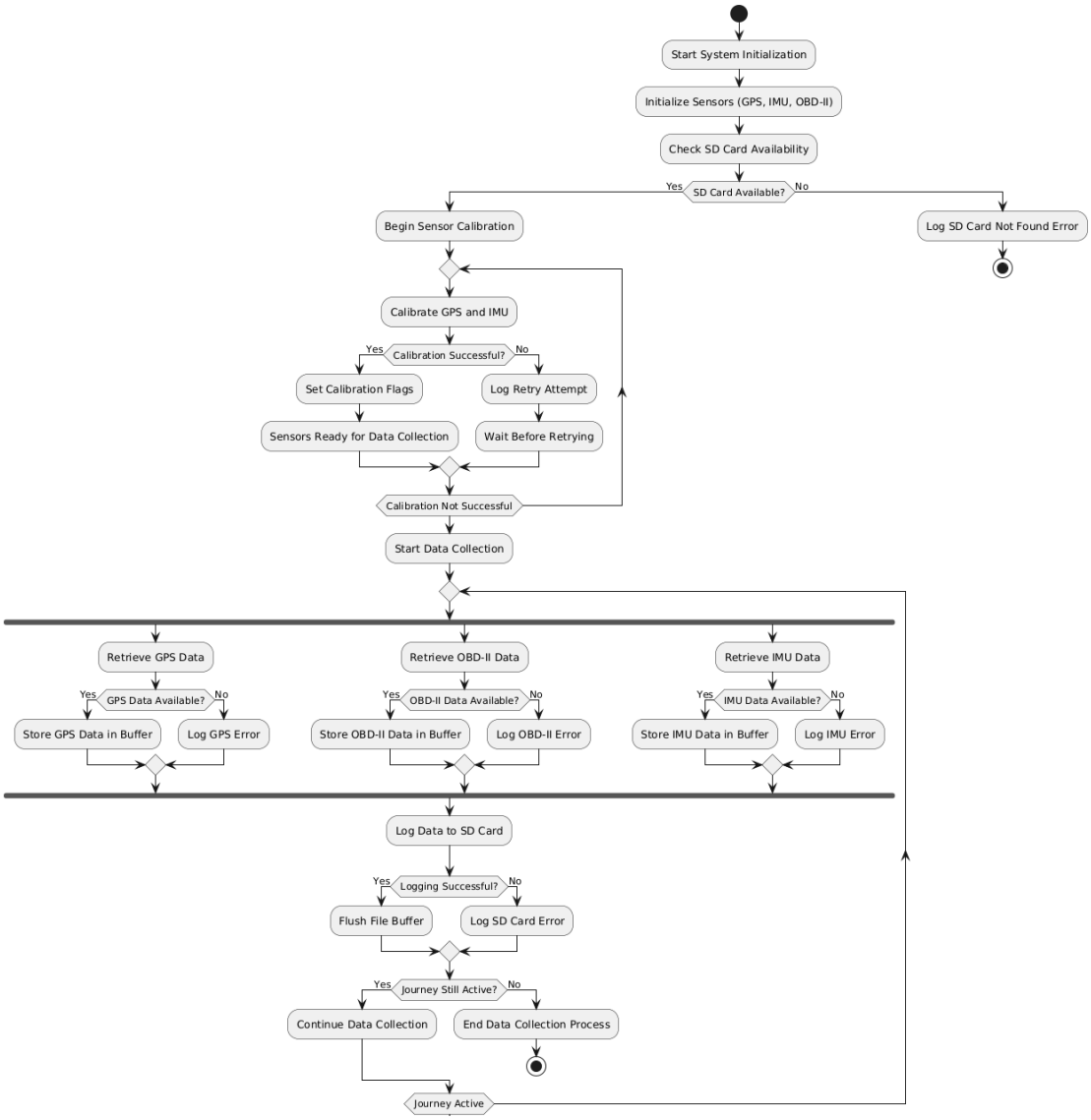


Figure 5: Activity diagram - Embedded System

3.4.2 Desktop Interface

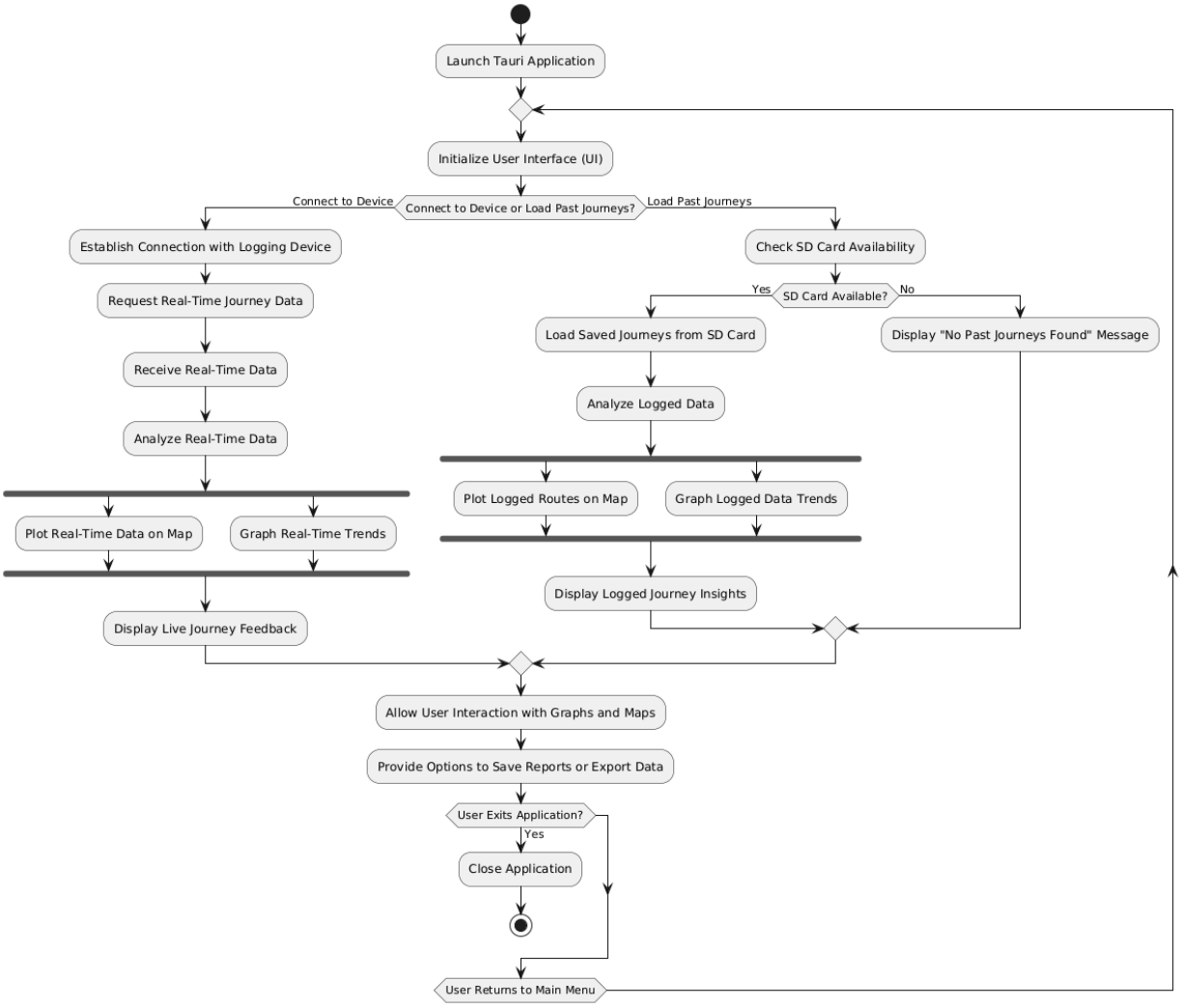


Figure 6: Activity Diagram - Desktop Interface

4 Methodology

This project follows the Agile Development Methodology. As stated Robinson, S. (2024), “Agile is a type of software development methodology that anticipates the need for flexibility and applies a level of pragmatism to the delivery of the finished product.” By adopting Agile principles, this project can adapt to evolving requirements, ensuring that development is responsive to new features. Regular testing and iterative development help to identify and address potential issues early.

The Agile Manifesto outline four values (Beck et al., 2001):

- “Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan”

The Scrum framework is the chosen Agile framework for this project. As described by Drumond, C. (2025) “The scrum framework outlines a set of values, principles, and practices that scrum teams follow to deliver a product or service. It details the members of a scrum team and their accountabilities, ‘artifacts’ that define the product and work to create the product, and scrum ceremonies that guide the scrum team through work.”

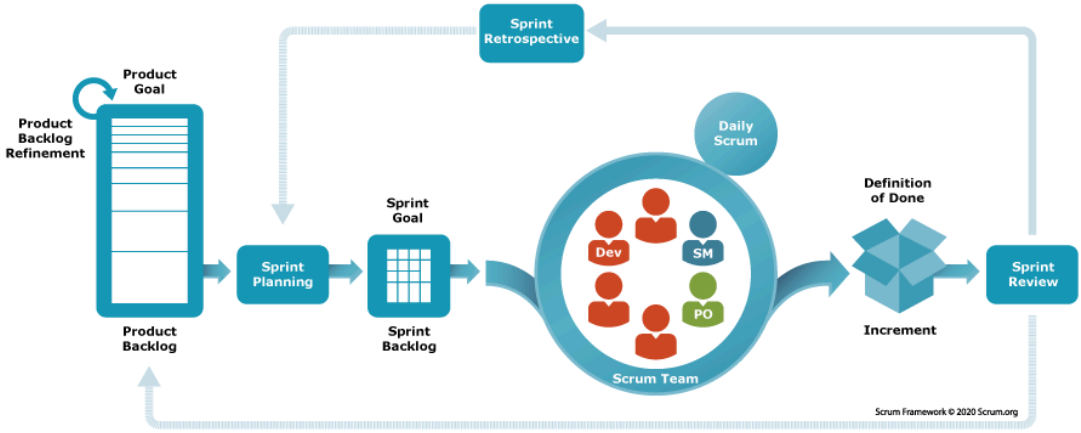


Figure 7: Scrum flow (Scrum.org, 2022)

4.1 Sprint Planning

Scrum breaks down work into sprints, short development cycles where specific tasks from the product backlog are completed. The product backlog contains all tasks needed to develop the project, with priorities typically set by a Product Owner. I will take on that role in this project, selecting and organising tasks for each sprint.

Sprint planning ensures each sprint has clear objectives and a manageable workload. Once a sprint is completed, the progress is reviewed. Usually, this would be presented to the Product Owner, but in this case, it will be demonstrated to my project supervisor. A breakdown of tasks assigned to each sprint can be found in the Requirements section.

4.2 Tests

After each sprint, tests are carried out to ensure the system functions as expected. Testing is divided into embedded system testing, interface testing and enclosure testing.

4.2.1 Embedded System Testing

The embedded system testing checks functionality, accuracy, and reliability.

- Unit Tests:
    - Test individual sensor readings (OBD-II, GNSS, IMU) for expected outputs.
    - Validate fuel efficiency calculations.
    - Testing logging functionality
- Empirical Testing:
    - Compare OBD-II speed and RPM with vehicle dashboard.
    - Validate GNSS accuracy against a reference device (Google Maps).
    - Check IMU response to acceleration, braking and cornering.

4.2.2 Interface Testing

Interface testing checks usability and data presentation.

- Unit Tests:
    - Validate that individual UI components render correctly.
- Widget Tests:
    - Ensure data displays correctly within maps and graphs.
- Integration Tests:
    - Ensure data displays from the embedded system to the interface.
    - Verify cross-platform compatibility and data integrity.

4.2.3 Enclosure Testing

Empirical tests evaluate the enclosure's performance under typical vehicle conditions.

.

- Thermal Tests:
    - Monitor internal temperature under heat (e.g., placed in the sun).
- Vibration and Shock Tests:
    - Test that internal components remain secure during driving
    - Ensure the enclosure stays fixed in place while driving.
- Access Tests:
    - Confirm that IO is accessible without disassembly.

4.2.4 Sprint One: Embedded system

This sprint aims to develop the software and a prototype for the hardware for logging driving data. The requirements to be implemented in this sprint are:

- FR-01
- FR-02
- FR-03
- FR-05
- FR-06
- FR-07
- NFR-03
- NFR-07
- NFR-08
- NFR-10

4.2.5 Sprint Two: User Interface and Data Transmission

This sprint aims to develop the application to display the collected data on an interface and handle data transmission from the embedded system. The requirements to be implemented in this sprint are:

- FR-08
- FR-09
- FR-10
- FR-11
- FR-12
- FR-13
- FR-14
- FR-15
- NFR-01
- NFR-02
- NFR-05
- NFR-07
- NFR-08

4.2.6 Sprint Three: Embedded system enclosure

This sprint aims to develop housing for the embedded system. The requirements to be implemented in this sprint are:

- NFR-04
- NFR-06
- NFR-09

4.3 Project Timeline

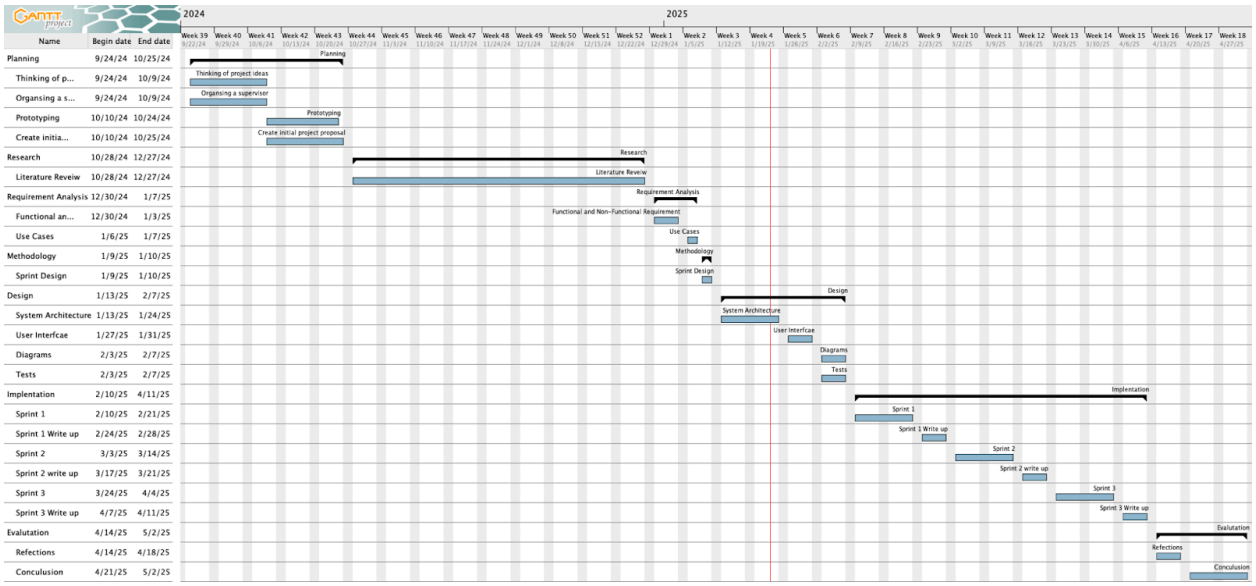


Figure 8: Gantt chart

5 Product Design

This section details the design and architecture of the Automotive Monitoring System, which combines hardware, software and user interface elements to meet the project's requirements.

The design incorporates the following key components:

- Hardware Design: Selection and integration of components.
- Data Communication: Design for interface access logged JSON files
- User interface Design: A desktop application visualising data through interactive maps and graphs.
- Enclosure: A durable, compact housing for the embedded system.

5.1 Hardware Design

This system integrates OBD-II, GNSS, IMU, and an SD card for vehicle data logging. The SparkFun OBD-II UART enables vehicle communication, the u-blox NEO-M8U provides GNSS with dead reckoning and the IMU captures motion data. An SD card stores all collected data for analysis.

The circuits will initially be designed on breadboards, and once finalised, they will be soldered onto a perforated circuit board.

Below is my circuit design, made in Fritzing, followed by an explanation of each component.

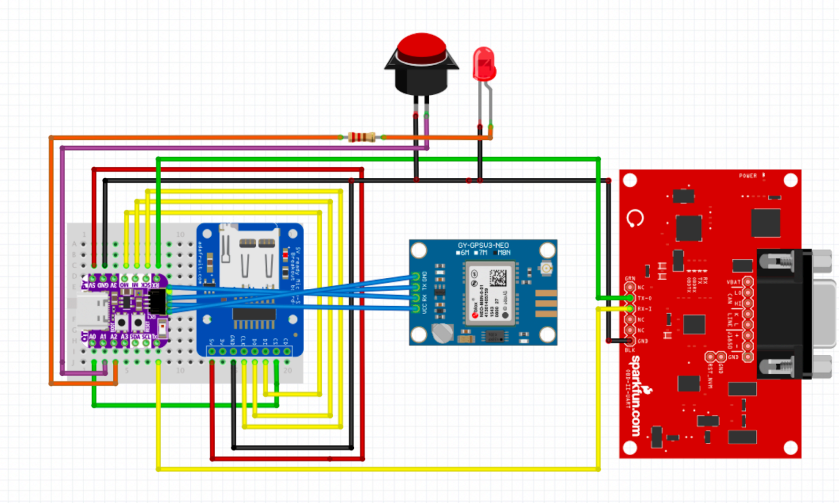


Figure 9: Hardware Design

5.1.1 OBD-II UART

I will be interfacing with the vehicle's OBD-II port using the SparkFun OBD-II UART module. As its name suggests, this module uses UART (Universal Asynchronous Receiver-Transmitter) communication to exchange data between the vehicle and the microcontroller.

As stated by SparkFun (no date), “The OBD-II UART board has both the STN1110 and the MCP2551 chips populated on it, allowing the user to access both CAN and OBD-II protocols.”. Sparkfun further states, “STN1110 is an OBD to UART interpreter that can be used to convert messages between any of the OBD-II protocols currently in use, and UART. It is fully compatible with the de facto industry standard ELM327 command set.”.

The MCP2551 chip, on the other hand, is a high-speed CAN transceiver. As stated by Microchip Technology Inc. (2010), “The MCP2551 device provides differential transmit and receive capability for the CAN protocol controller and is fully compatible with the ISO-11898 standard. It will operate at speeds of up to 1 Mb/s.”. It serves as a bridge between the physical CAN bus signals and the digital signals required by the STN1110 for processing.

Together, the STN1110 and MCP2551 on the OBD-II UART module can communicate with a wide range of OBD-II-compliant vehicles, regardless of the protocol in use. Furthermore, the STN1110’s compatibility with the ELM327 command set simplifies the process of requesting specific PIDs to retrieve diagnostic data.

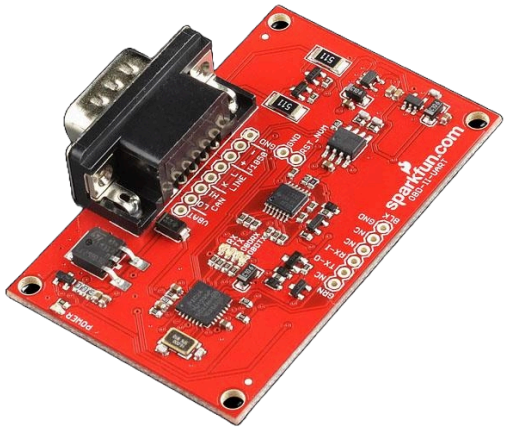


Figure 10: SparkFun OBD-II UART
(Mouser Electronics, 2025)

5.1.2 GNSS & IMU

The system uses the SparkFun NEO-M8N. This module provides longitude and latitude using Satellite constellations. When calibrated, its onboard IMU can also use Dead Reckoning.

As U-blox (2018) highlights, the NEO-M8N facilitates, “UART, USB, DDC (I2C compliant), and SPI interface options provide flexible connectivity,” allowing for integration with various systems. The module also features a Qwiic connector, enabling quick and reliable interfacing with other devices without soldering.

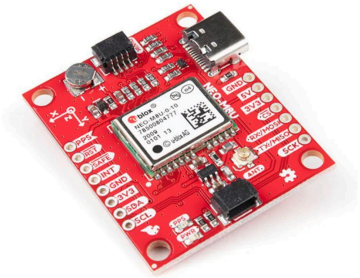


Figure 11: SparkFun NEO-M8U
(Mouser Electronics, 2025)

5.1.2.1 GNSS Module

The U-blox NEO-M8U features Untethered Dead Reckoning (UDR) technology. As backed up by U-blox (2018), “The NEO-M8U module introduces U-blox’s Untethered Dead Reckoning (UDR) technology, which provides continuous navigation without requiring speed information from the vehicle.”

The NEO-M8U integrates GNSS data with IMU data to provide accurate, real-time positioning. According to U-blox (2018), “The combination of GNSS and sensor measurements enables accurate, real-time positioning at rates up to 30 Hz.” Additionally, the module offers multiple connectivity options. As u-blox (2018) explains, “includes GPS ,Galileo, GLONASS, BeiDou, and QZSS.”

The NEO-M8U features a u.FL antenna connector. For this project, I am using a DollaTek 28dB high-gain GPS active antenna to gain reliable signal reception. Furthermore, this antenna includes an integrated low-noise amplifier (LNA) to enhance weak signals.

Figure 12: DollaTek 28dB GPS Active Antenna (DollaTek, nd)

In order for the Dead Reckoning to work the sensor requires calibration. As stated by U-blox (2018) this is done is three phases:

Phase	Procedure	Indicator of Success
IMU initialization	After receiver cold start or first receiver use, turn on car engine and stay stationary under good GNSS signal reception conditions for at least 3 minutes.	IMU initialization status (imuInitStatus) is flagged as 2:INITIALIZED in the UBX-ESF-STATUS message.
INS initialization (position and velocity)	Once IMU is initialized, stay stationary under good GNSS signal reception conditions until a reliable GNSS fix is achieved.	GNSS 3D fix achieved, good 3D position accuracy (at least 5 m), high number of used SVs (check UBX-NAV-PVT message).
IMU-mount alignment initialization	Start driving at a minimum speed of 30 km/h and perform a series of approximately 10 left and right turns (at least 90-degree turns). Each turn should be as if driving in a sharp roundabout. This step can be skipped if automatic IMU-mount alignment is turned off.	IMU-mount alignment status (mntAlgStatus) is flagged as 2:INITIALIZED in the UBX-ESF-STATUS message, and the IMU-mount alignment status (status) is flagged as 3:COARSE ALIGNED in the UBX-ESF-ALG message.
Wheel-tick sensor initialization	Drive for at least 500 meters at a minimum speed of 20 km/h. To shorten this calibration, drive at a higher speed (~50 km/h) for at least 10 seconds under good GNSS visibility.	Wheel-tick sensor initialization status (wtInitStatus) is flagged as 2:INITIALIZED in the UBX-ESF-STATUS message.
INS initialization (attitude)	Drive straight for at least 100 meters at a minimum speed of 40 km/h.	INS initialization status (insInitStatus) is flagged as 2:INITIALIZED in the UBX-ESF-STATUS message.

Table 5: NEO-M8U Calibration procedure (u-blox, 2018)

5.1.2.2 IMU Module

The NEO-M8U module has a 6-axis Inertial Measurement Unit (IMU), which includes an accelerometer and gyroscope. The specific IMU used within the module can vary depending on the firmware version and hardware.

According to the U-blox datasheet (U-blox, 2023), the NEO-M8U may feature IMU sensors from ST, Bosch, or InvenSense, such as the LSM6DS3, BMI160, or MPU6500. After asking on the U-blox forum (U-blox, 2024), I found that it “typically used the Bosch BMI160”, as referenced in datasheet page 366.

5.1.3 SD Card

The SD card module is an essential component for storing collected data. The chosen module is the Adafruit MicroSD Card Breakout Board, communicating using SPI.

To ensure reliable communication, the board includes a CD74HC4050 level shifter that improves signal quality. SD cards are highly sensitive to signal integrity, as they require precise, edge-triggered transitions. As Ladyada (2013) notes, “The newest cards are edge-triggered and require very 'square' transitions.” The level shifter mitigates these issues, ensuring stable data transfer.

The SD card is formatted with the FAT32 file system to be compatible with the microcontroller.

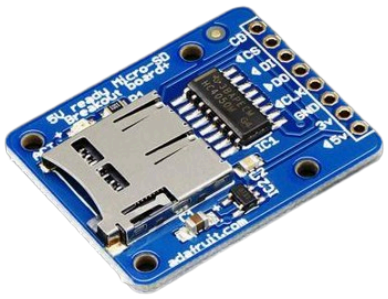


Figure 13: Adafruit MicroSD Card Breakout Board
(Mouser Electronics, 2025)

5.1.4 Embedded System Control

The embedded system is operated using a latching button that controls data logging. The button has a built-in LED to indicate calibration and logging status. Pressing the button initiates data logging.

Figure 14: Latching on/off button (The Pi Hut, 2025)

5.1.5 Cooling

The selected fan is the Noctua NF-A4x10 5V, chosen for its low power consumption and efficient airflow.

Figure 15: Noctua fan (Amazon, 2025)

This fan operates at 0.25 watts, which when calculated using the formula:

$$Current(I) = \frac{Power(P)}{Voltage(v)}$$

$$I = \frac{0.25W}{5V} = 50mA$$

A current draw of 50mA makes it an ideal option, as it can run directly from the microcontroller's power output without requiring an additional power source.

5.2 Data communication

This sub-section details how the embedded systems and interface will communicate.

5.2.1 HTTP-Based Web Server

The embedded system and the interface communicate using an HTTP (Hypertext Transfer Protocol) web server access point using RESTful architecture. This setup enables the embedded system to host a web server that acts as an access point, handling the transfer of JSON files through HTTP requests.

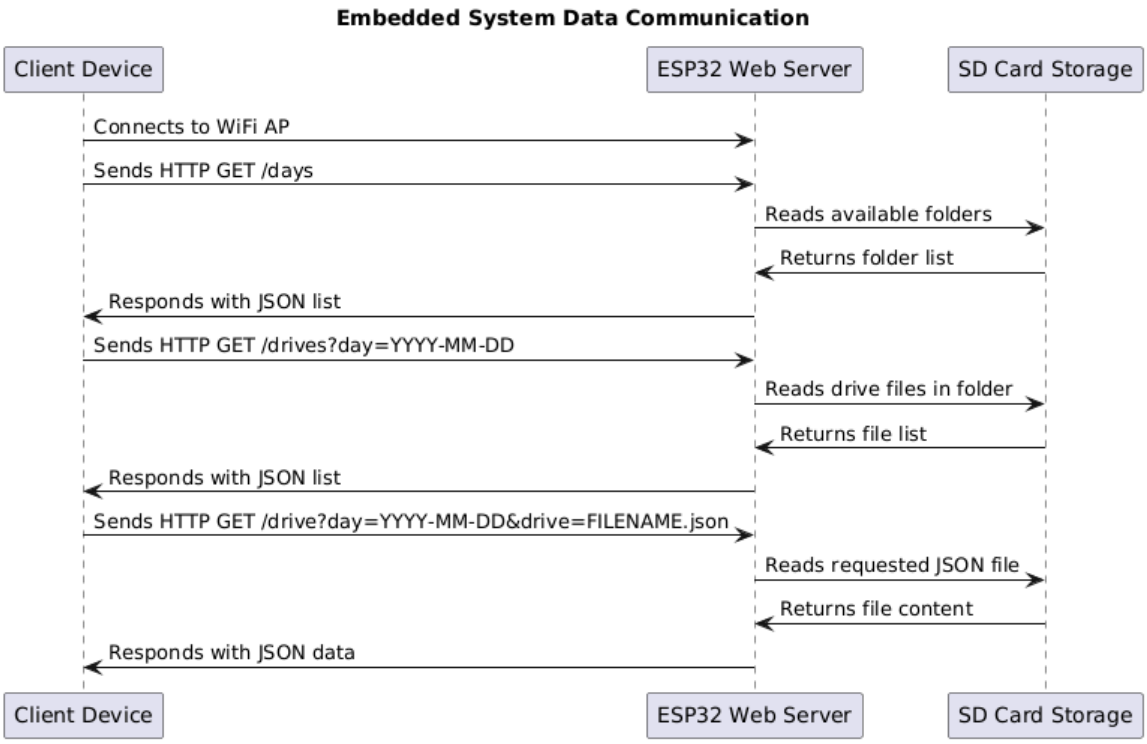


Figure 16: Data communication - Sequence Diagram

5.2.2 API Endpoints

The web server transmits data through API endpoints:

Endpoint	**Description**
/Index	Verifies server connectivity.
/days	Lists dates with recorded driving sessions.
/drives?day={YYYY-MM-DD}	Lists drives for a specific day.
/drive?day={YYYY-MM-DD}&drive={ DRIVETIME}	Retrieves the complete dataset for a session.
/live	Provides real-time access to the latest JSON file.
/sdinfo	Check SD card storage.
/delete	Allows removal of individual drives or entire days.

Table 6: API endpoints

5.2 User Interface

The UI (User Interface) follows Galitz's principles: "people want to do, not learn to do," showing key functions first and hiding advanced ones. Design consistency: "similar components have a similar look, similar uses, and operate similarly", and supports error recovery, as "to forgive is good design" (Galitz, W. O., 2007).

5.2.1 Components and Features

The application consists of three primary views:

- Post-Drive Analysis Interface – Displays detailed data from completed drives.
- Real-Time Analysis Interface – Provides live vehicle data during the drive.
- Settings Interface – Allows users to configure preferences and manage data.

5.2.1.1 Post-drive interface

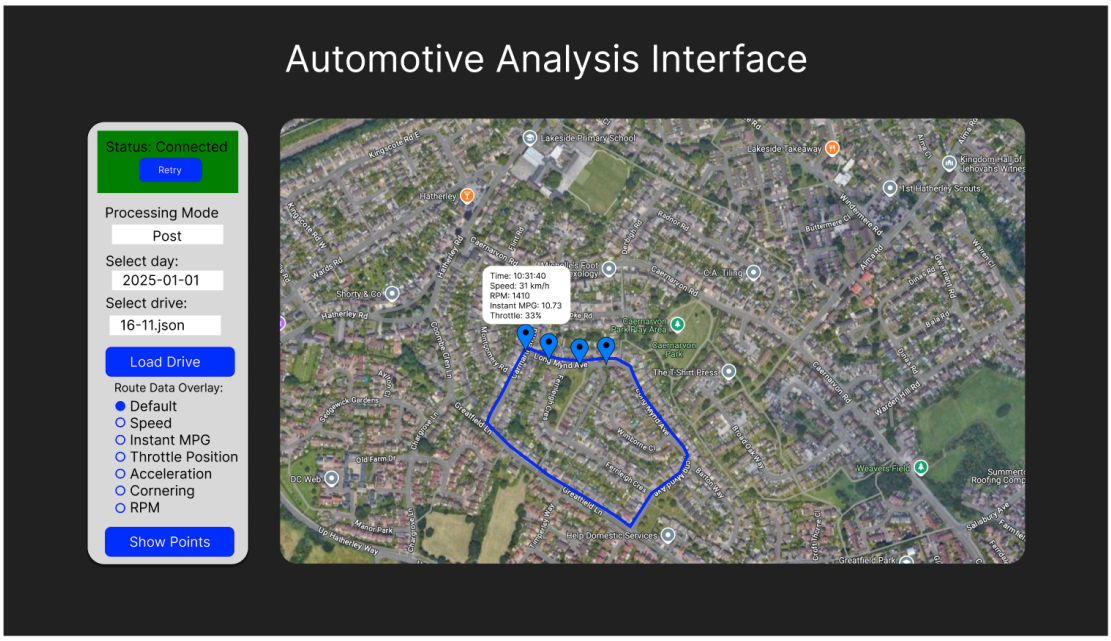


Figure 17: Post-drive interface design

5.2.1.2 Real-time interface

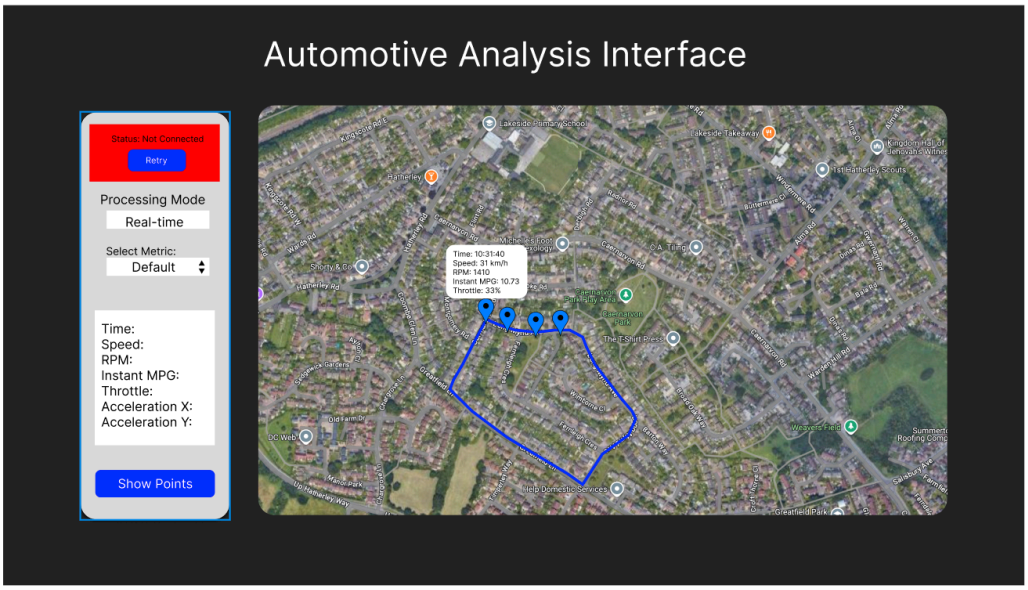


Figure 18: Real-time interface design

5.2.1.3 Settings interface

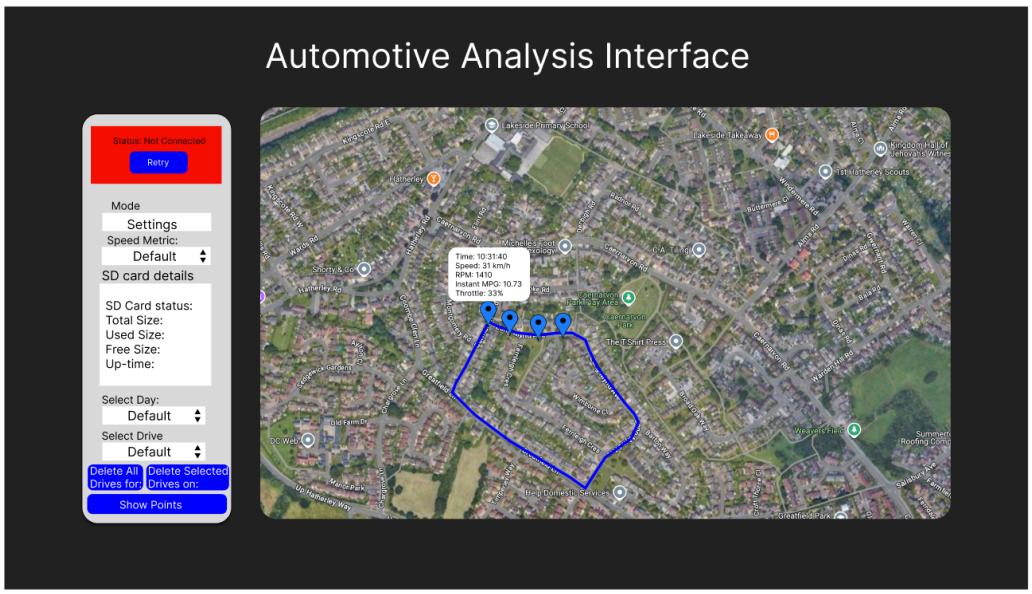


Figure 19: Settings interface

5.3 Enclosure

The enclosure will be made from Medium-Density Fiberboard (MDF). It will be put together using finger joints, where interlocking cuts along the edges slot together. This increases the surface area, making the structure stronger when glued.

A removable lid will provide access to the internal components. There will also be cutouts in the enclosure for external connections, with specific openings for the USB-C port that powers the device, a 15-pin connector for the OBD interface, and a button to control logging. These cutouts will be positioned so that everything is easily accessible.

To ensure the enclosure stays secure when placed on the dashboard, the bottom will be fitted with a grippy material. This will prevent it from sliding around while the vehicle is in motion, keeping it stable even on bumpy roads.

For cooling a 5v fan will be installed to ensure that the components inside stay cool with suitable ventilation.

The enclosure will be designed using Fusion 360, allowing for modelling before manufacturing. Once finalised, the individual panels will be cut using a laser cutter.

Below is a mood board illustrating the design inspiration, material choices and construction techniques for the enclosure.

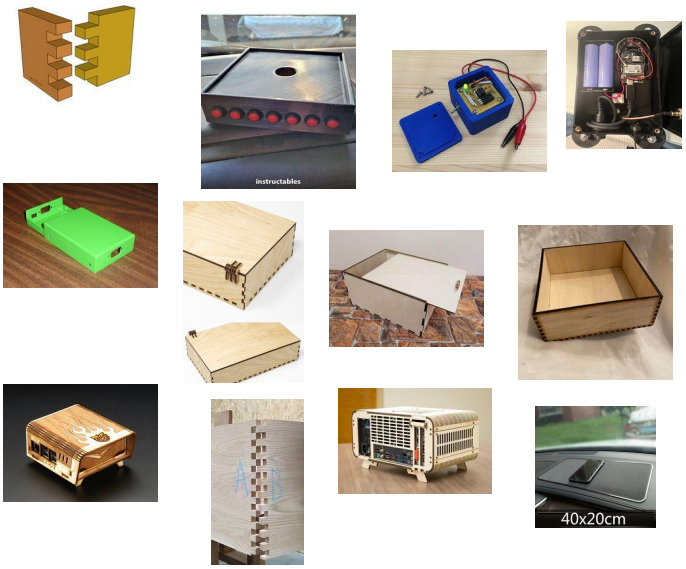


Figure 20: Enclosure mood board

6 Implementation

This section describes how the project was implemented.

6.1 Sprint One: Embedded System

This sprint focused on developing the Embedded Systems hardware prototype and the software.

6.1.1 Embedded System Hardware

The hardware components for the project were sourced from Mouser Electronics, allowing all parts to be ordered at once at a competitive price.

To begin assembly I soldered pins onto the microcontroller and the SparkFun OBD-II UART board. I then placed the microcontroller into a mini breadboard and wired the microcontroller to the OBD-II UART board using the TX and RX pins(the module's TX connected to the microcontroller's RX, and vice versa).

Next, the GNSS module, SparkFun NEO-M8U, was connected using its Qwiic connector to the microcontroller and the external antenna was attached to the module via the UF2 connector.

The microSD card module, which came with pre-soldered pins, was placed into the mini breadboard and wired to the microcontroller's SPI pins (MCK, MI, MO), and the Chip select was wired to Analog Pin 0.

The button was then added and was wired to Analog Pin 1, while the LED was connected to Analog Pin 2 with a 220-ohm pull-down resistor.

The components were mounted onto a wood block, with the OBD-II module and GNSS module secured using nylon standoffs. This setup allowed for initial testing before transitioning to a more permanent design.

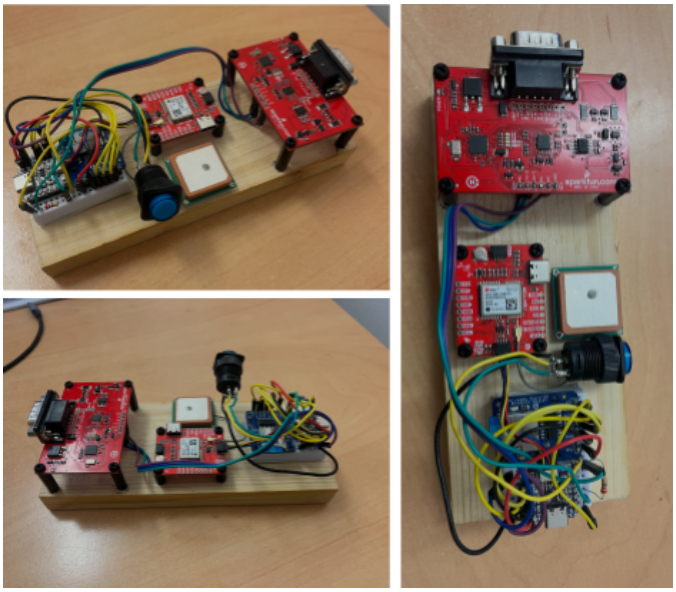


Figure 21: Embedded system prototype

6.1.1 Embedded System Setup and Initialisation

The microcontroller was programmed using PlatformIO, an open-source environment for developing embedded software, as shown in Figure 22.

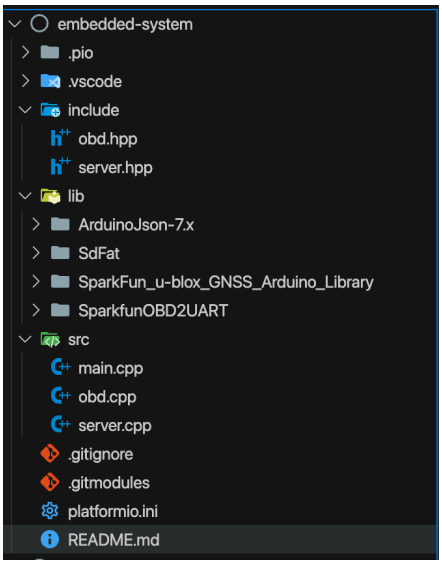


Figure 22: New Embedded system project structure

The table below describes the purpose of each folder/file:

Name	**Purpose**
include/	Contains header files (.h) that define functions used throughout the project.
lib/	Contains external libraries and custom modules.
src/	Contains the main source code for the system functionality
platormio.ini	Configuration file specifying target hardware, frameworks, and dependencies.

Table 7: PlatformIO folder/file purpose

The microcontroller makes use of several libraries for communications as well as a Real-Time Operating System (RTOS). The libraries are as follows:

Library	**Rational**	**Source**
Arduino	Provides core functions for pin control, timing, and serial communication.	GitHub
Wire	Enables I2C communication with GNSS and IMU sensors.	GitHub
SPI	Handles SPI communication with the SD card module.	GitHub
SparkfunOBD2UART	Retrieves vehicle diagnostics (speed, RPM, throttle, MAF) via UART.	GitHub
SparkFun_u-blox_GNSS_Arduino_Library	Communicates with NEO-M8U over I2C for location and dead reckoning.	GitHub
SdFat	Manages SD card storage with high-speed read/write.	Github
ArduinoJson	Structures sensor data into JSON for transmission/storage.	GitHub
WiFi	Enables wireless connectivity for data transmission.	GitHub

Table 8: Embedded System Initialised Libraries

6.1.1.1 Modules Setup and Initialisation

Now that the PlatformIO environment is set up, the next step is to initialise the external modules.

First, the latching push button is initialised with the button pin as an input and the LED pin as an output. The LED is initially off.

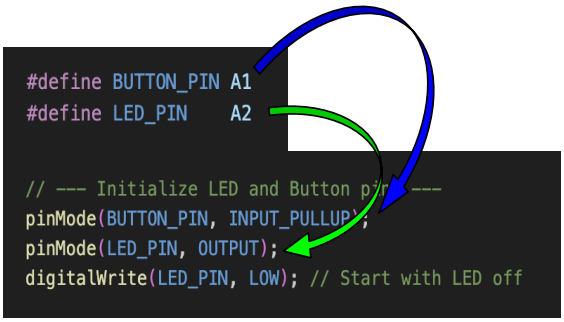


Figure 23: Button initialise

Next, the SD card module is initialised next. The communication starts with SD.begin, where the chip select pin is set and the SPI clock speed is set to 1 MHz.

```
// --- SD Card Initialization ---
if (!SD.begin(SD_CS_PIN, 1000000)) {
    Serial.println("SD card initialization failed!");
} else {
    Serial.println("SD card initialized successfully.");
}
```

Figure 24: SD card initialise

Moving onto the OBD-II UART module and the NEO-M8U. These are only initialised when the button is pressed for the first time (Section 6.1.2). This allows the device to still serve server clients when the system is not connected to a vehicle.

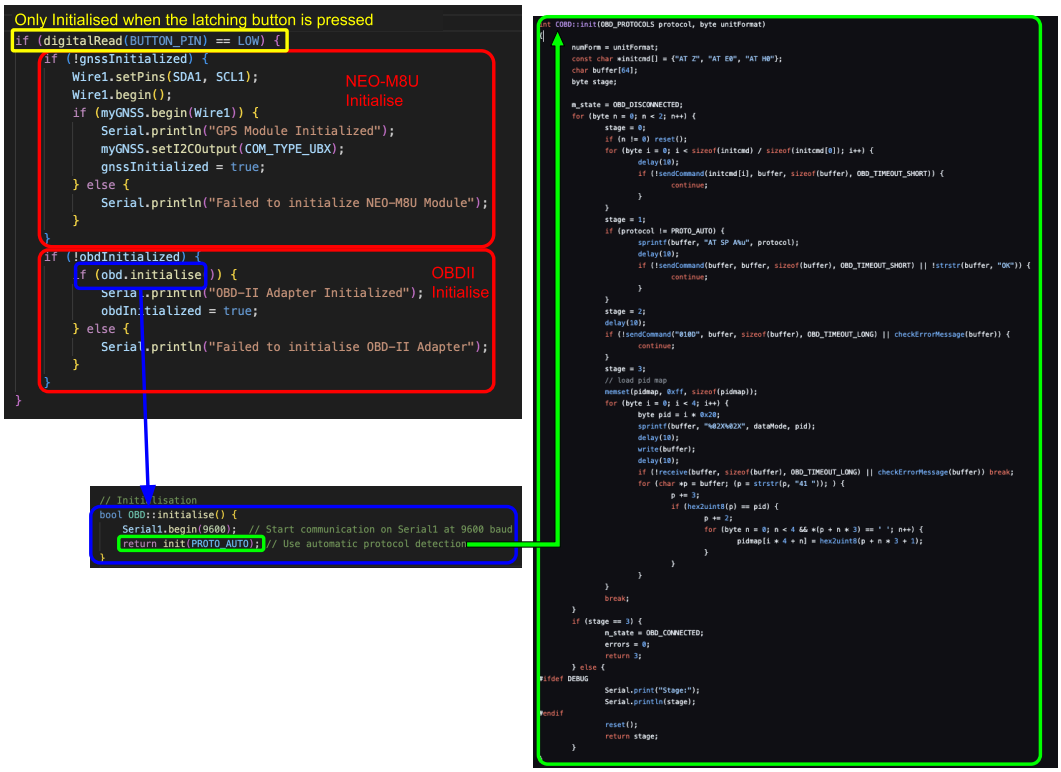


Figure 25: OBD-II UART and NEO-M8U initialise

The OBD-II UART module communicates with the microcontroller via a UART interface at a 9600 baud rate.

The module initialises via initialise(), which calls init(), from the SparkFun library for automatic protocol detection. The init() function resets the OBD-II module (PID-ATZ), disables echo (PID-ATE0), removes headers (PID-ATH0), and enables auto protocol detection (PID-ATSP0). It then verifies communication, and if successful, the module state is set to OBD_CONNECTED.

The NEO-M8U initialisation uses the Wire library, set to channel one with the correct I2C bus (Qwiic connector), as shown in Figure 25.

6.1.1.2 Server Setup and Initialisation

```
// WiFi Access Point Credentials
const char* ssid = "MyESP32AP";
const char* password = "12345678";

// Setup: Initialization & Task Creation
void setup() {
    Serial.begin(115200);
    delay(1000);
    Serial.println("Initialising System...");

    WiFi.softAP(ssid, password);
    Serial.println("AP IP address: " + WiFi.softAPIP().toString());
```

Figure 26: Server initialise

Now that all external modules have been initialised, the next step is server setup. This involves configuring the access point's SSID and password and enabling the microcontroller as a SoftAP (software-enabled access point).

6.1.1.3 NEO-M8U Calibration

Once the initialisation is complete, the NEO-M8U can be calibrated for dead reckoning.

```
void calibrateGNNS() {
    Serial.println("Starting GNSS Calibration...");
    // Loop until calibration is achieved.
    while (!isCalibrated) {
        if (myGNSS.getEsfInfo()) {
            int fusionMode = myGNSS.packetUBXESFSTATUS->data.fusionMode;
            Serial.print("Fusion Mode: ");
            Serial.println(fusionMode);
            if (fusionMode == 1) {
                Serial.println("Calibrated!");
                isCalibrated = true;
            } else {
                Serial.println("→ Initialising... Perform calibration maneuvers.");
            }
        } else {
            Serial.println("Failed to retrieve ESF Info. Retrying...");
        }
        delay(1000);
    }
    Serial.println("Calibration Complete!");
}
```

Figure 27: NEO-M8U Calibration

The calibrateGNSS() function handles this by continuously checking the module's Extended Sensor Fusion (ESF) status. If the response is valid, it checks the fusionMode value. A value of 1 means the system is fully calibrated, and the function exists.

6.1.2 Core functionality

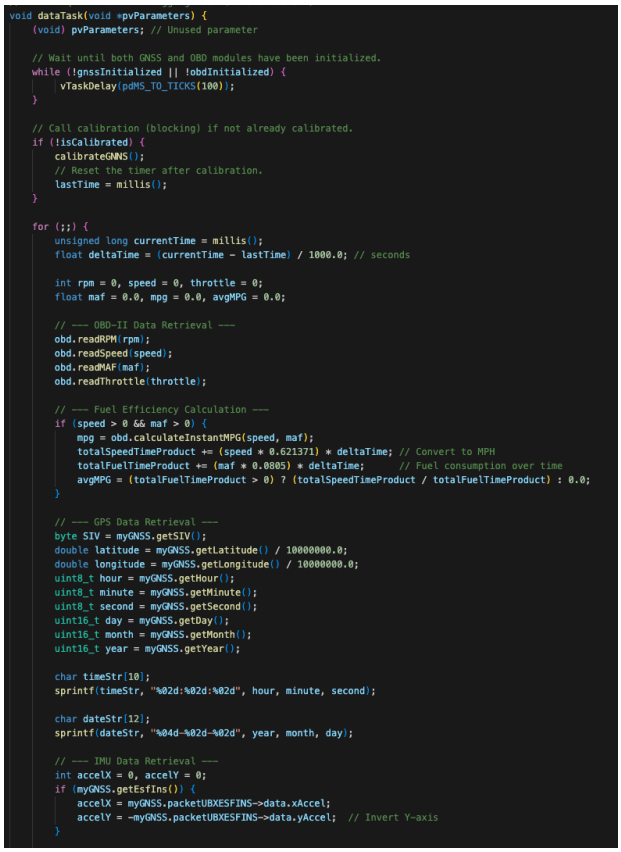
The microcontroller uses a dual-core architecture with FreeRTOS to handle data acquisition and server requests concurrently. Initially, I made it as a sequential execution, but it caused client requests to time out from the server while logging was enabled; this was due to sensor response times. Now, Core 1 manages sensor data collection and processing, while Core 0 handles client requests and logging control. A mutex is used to ensures safe access to the SD card, as Core 1 writes data and Core 0 reads it. Without this synchronisation, simultaneous read/write access led to deadlock errors.

Starting with data acquisition on Core 0. This task is created using xTaskCreatePinnedToCore(), pinning the task exclusively to Core 1.

```
// --- Create Data Task on Core 1 ---
xTaskCreatePinnedToCore(
    dataTask,      // Task function.
    "Data Task",   // Name of task.
    16384,         // Stack size.
    NULL,          // Parameter.
    1,             // Task priority.
    NULL,          // Task handle.
    1              // Run on Core 1.
);
```

Figure 28: Create Data Task on Core 1

The dataTask() collects data from all the sensors and logs it to the SD card. The functionality is shown below.



```
void dataTask(void *pvParameters) {
    (void) pvParameters; // Unused parameter

    // Wait until both GNSS and OBD modules have been initialized.
    while (!gnssInitialized || !obdInitialized) {
        vTaskDelay(pdMS_TO_TICKS(100));
    }

    // Call calibration (blocking) if not already calibrated.
    if (!isCalibrated) {
        calibrateGNNS();
        // Reset the timer after calibration.
        lastTime = millis();
    }

    for (;;) {
        unsigned long currentTime = millis();
        float deltaTime = (currentTime - lastTime) / 1000.0; // seconds

        int rpm = 0, speed = 0, throttle = 0;
        float maf = 0.0, mpg = 0.0, avgMPG = 0.0;

        // --- OBD-II Data Retrieval ---
        obd.readRPM(rpm);
        obd.readSpeed(speed);
        obd.readMAF(maf);
        obd.readThrottle(throttle);

        // --- Fuel Efficiency Calculation ---
        if (speed > 0 && maf > 0) {
            mpg = obd.calculateInstantMPG(speed, maf);
            totalSpeedTimeProduct += (speed * 0.621371) * deltaTime; // Convert to MPH
            totalFuelTimeProduct += (maf * 0.0805) * deltaTime;      // Fuel consumption over time
            avgMPG = (totalFuelTimeProduct > 0) ? (totalSpeedTimeProduct / totalFuelTimeProduct) : 0.0;
        }

        // --- GPS Data Retrieval ---
        byte SIV = myGNSS.getSIV();
        double latitude = myGNSS.getLatitude() / 10000000.0;
        double longitude = myGNSS.getLongitude() / 10000000.0;
        uint8_t hour = myGNSS.getHour();
        uint8_t minute = myGNSS.getMinute();
        uint8_t second = myGNSS.getSecond();
        uint16_t day = myGNSS.getDay();
        uint16_t month = myGNSS.getMonth();
        uint16_t year = myGNSS.getYear();

        char timeStr[10];
        sprintf(timeStr, "%02d:%02d:%02d", hour, minute, second);

        char dateStr[12];
        sprintf(dateStr, "%04d-%02d-%02d", year, month, day);

        // --- IMU Data Retrieval ---
        int accelX = 0, accelY = 0;
        if (myGNSS.getEsfIns()) {
            accelX = myGNSS.packetUBXESFINS->data.xAccel;
            accelY = -myGNSS.packetUBXESFINS->data.yAccel;  // Invert Y-axis
        }
```

Figure 29: Data task

The data task function initialises the GNSS and OBD-II modules, blocking function execution until both are ready. If the GNSS isn't calibrated, calibrateGNSS() is called. Once calibrated and initialised, OBD data is retrieved, and fuel efficiency is calculated. GNSS data, including coordinates and satellite count, is then collected. Lastly, IMU acceleration data is acquired, with Y-axis inversion applied for correct orientation.

Once the data is collected, it can be logged and organised on the SD card in JSON format. As seen below:

```
// Log only if calibration is done, the button is pressed and a journey is active.
if (loggingActive) {
    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000))) { // Protect SD access
        if (firstLog) {
            sprintf(folderName, "%04d-%02d-%02d", year, month, day);
            if (!SD.exists(folderName)) SD.mkdir(folderName);
            sprintf(fileName, "%s/%02d-%02d.json", folderName, hour, minute);
            logFile = SD.open(fileName, O_RDWR | O_CREAT | O_AT_END);
            if (logFile) {
                Serial.printf("Log file created: %s\n", fileName);
                firstLog = false;
            } else {
                Serial.println("Failed to create log file.");
            }
        }

        if (logFile) {
            StaticJsonDocument<256> jsonDoc;
            jsonDoc["gps"]["time"] = timeStr;
            jsonDoc["gps"]["latitude"] = latitude;
            jsonDoc["gps"]["longitude"] = longitude;
            jsonDoc["obd"]["rpm"] = rpm;
            jsonDoc["obd"]["speed"] = speed;
            jsonDoc["obd"]["maf"] = maf;
            jsonDoc["obd"]["instant_mpg"] = mpg;
            jsonDoc["obd"]["throttle"] = throttle;
            jsonDoc["obd"]["avg_mpg"] = avgMPG;
            jsonDoc["imu"]["accel_x"] = accelX;
            jsonDoc["imu"]["accel_y"] = accelY;

            if (serializeJson(jsonDoc, logFile) == 0) {
                Serial.println("Failed to serialize JSON.");
            } else {
                logFile.println();
                Serial.println("\nData logged.");
            }
            logFile.flush();
        } else {
            Serial.println("Log file not open. Retrying...");
            logFile = SD.open(fileName, O_RDWR | O_CREAT | O_AT_END);
        }
        xSemaphoreGive(sdMutex); // Release SD Mutex
    } else {
        Serial.println("SD mutex timeout, skipping write...");
    }
}

lastTime = currentTime;
vTaskDelay(pdMS_TO_TICKS(1000));  // 1-second delay (non-blocking)
```

Figure 30: OBD-II data retrieval functions 2

Continuing the dataTask(), it also manages data logging, ensuring it occurs only when enabled. To prevent concurrent access to the SD Card, it acquires the sdMutex. If this is the first log entry, a directory and log file are created using the timestamp as the filename, and the firstLog flag is disabled to prevent recreation of the file. A 256-byte JSON document is then initialised to store the data. The JSON is serialised and written to the file with a newline appended for readability. Finally, the data is ‘flushed’ to the SD card to save all buffered content before releasing the mutex.

To interface with the OBD-II, I have implemented a custom class. The functions are:

- Public
    - initialise - Initalise OBD-II.
    - readRPM - Returns RPM Integer.
    - readSpeed - Returns KPH Integer.
    - readMAF - Reuters MAF float.

    - readThrottle - Returns throttle position percentage integer.
    - calculateInstantMPG - Returns calculated instant MPG.
    - calculateAverageMPG - Returns calculated average MPG.
- Private
    - sendPIDCommand - Sends a hexadecimal PID code.
    - parseHexValue - Parses hexadecimal response to integer.

The OBD-II data retrieval calls the following functions from 'obd.hpp':

```cpp
//------------------------------------------------------------------------------
// sendPIDCommand(pid, response, bufsize)
//   - Sends an OBD-II PID request string (e.g., "010C").
//   - Waits up to OBD_TIMEOUT_LONG ms for a response.
//   - Fills the provided buffer with the reply
// Parameters:
//   - pid:      String of the PID command.
//   - response: char array to receive the reply.
//   - bufsize:  Size of the response buffer.
// Returns:
//   - true if any bytes were received before timeout, false on timeout.
//------------------------------------------------------------------------------
bool OBD::sendPIDCommand(const char* pid, char* response, int bufsize) {
    unsigned long startTime = millis();
    write(pid);  // Transmit the PID request

    // Loop until we receive data or timeout
    while (millis() - startTime < OBD_TIMEOUT_LONG) {
        int len = receive(response, bufsize, OBD_TIMEOUT_LONG);
        if (len > 0) {
            return true;  // Successful read
        }
    }
    // Timeout without data
    return false;
}

//------------------------------------------------------------------------------
// parseHexValue(response, startIndex, length)
//   - Parses a substring of ASCII hex digits into an integer.
// Parameters:
//   - response:    Response string.
//   - startIndex:  Index in the string where hex digits begin.
//   - length:      Number of hex characters to parse.
// Returns:
//   - Integer value of the parsed hex substring.
//------------------------------------------------------------------------------
int OBD::parseHexValue(const char* response, int startIndex, int length) {
    // strtol will stop after 'length' hex digits or at NUL
    return strtol(&response[startIndex], NULL, 16);
}

//------------------------------------------------------------------------------
// readRPM(rpm)
//   - Queries PID 0x0C to get engine RPM.
//   - Response format: "41 0C AA BB", where RPM = ((AA*256) + BB) / 4.
// Parameters:
//   - rpm: Reference to an int to store the result.
// Returns:
//   - true on success (rpm set), false otherwise.
//------------------------------------------------------------------------------
bool OBD::readRPM(int& rpm) {
    char response[64];
    if (sendPIDCommand("010C", response, sizeof(response))) {
        // Find the "41 0C" header in the reply
        char* ptr = strstr(response, "41 0C");
        if (ptr) {
            int A = parseHexValue(ptr, 6, 2);  // Byte A at offset 6
            int B = parseHexValue(ptr, 9, 2);  // Byte B at offset 9
            rpm = ((A * 256) + B) / 4;         // Compute RPM
            return true;
        }
    }
    return false;
}

//------------------------------------------------------------------------------
// readSpeed(speed_kph)
//   - Queries PID 0x0D to get vehicle speed in km/h.
//   - Response: "41 0D AA", where speed_kph = AA.
// Parameters:
//   - speed_kph: Reference to an int to store the speed.
// Returns:
//   - true on success, false otherwise.
//------------------------------------------------------------------------------
bool OBD::readSpeed(int& speed_kph) {
    char response[64];
    if (sendPIDCommand("010D", response, sizeof(response))) {
        char* ptr = strstr(response, "41 0D");
        if (ptr) {
            speed_kph = parseHexValue(ptr, 6, 2);  // Single byte value
            return true;
        }
    }
    return false;
}
```

Figure 31: OBD-II data retrieval functions 1

The sendPIDCommand function sends a PID request to the OBD-II module and waits for a valid response within 10000ms. It returns true if data is received; otherwise, it returns false.

The parseHexValue function converts hexadecimal string values from the OBD response into integers.

To retrieve engine RPM, PID 010C is sent, and the response bytes are parsed. The first byte (A) is the MSB (most significant bit), and the second (B) is the LSB(least significant bit), forming a 16-bit integer: (A * 256) + B. The value is then divided by four per the OBD-II standard. (ISO, 2006, Table B.13, p. 123).

$$RPM = \frac{(A * 256 + B)}{4}$$

To retrieve vehicle speed, PID 010D requests vehicle speed in KPH. The response is then converted to an integer.

The function readMAF sends PID 0110 to request MAF data. The OBD-II response consists of two bytes: the first (A) is the MSB, and the second (B) is the LSB. The value is calculated as (A * 256) + B and divided by 100 per OBD-II standards (International Organization for Standardization, 2006, Table B.17, p. 123).

$$MAF = \frac{A * 256 + B}{100}$$

The function readThrottle sends PID 014A to retrieve the absolute throttle position as a percentage. The OBD-II returns one byte, where 0x00 represents 0% and 0xFF represents 100%. The returned byte is multiplied by 100 and divided by 255 to find the percentage. (International Organization for Standardization, 2006, Table B.14, p. 123).

$$Throttle = \frac{(value * 100)}{255}$$

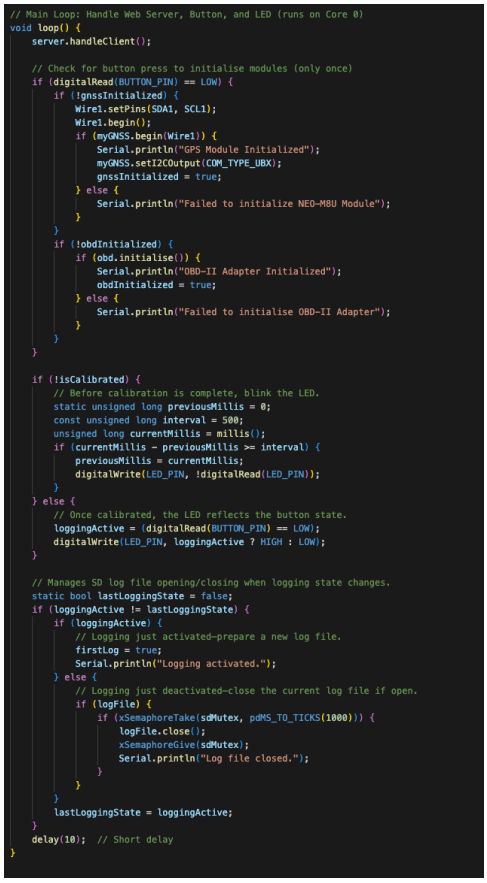
CalculateInstantMPG converts speed from KPH to MPH, computes fuel consumption in GPH, and calculates instantaneous MPG (calculation in section 2.6.2).

CalculateAverageMPG uses total speed-time and fuel-time products to compute average MPG, returning 0.0 if fuel consumption is zero (calculation in section 2.6.2).

```
//------------------------------------------------------------------------------
// readMAF(maf)
//   - Queries PID 0x10 to get mass air flow sensor data (g/s).
//   - Response: "41 10 AA BB", where maf = ((AA*256) + BB) / 100.
// Parameters:
//   - maf: Reference to a float to store grams per second.
// Returns:
//   - true on success, false otherwise.
//------------------------------------------------------------------------------
bool OBD::readMAF(float& maf) {
    char response[64];
    if (sendPIDCommand("0110", response, sizeof(response))) {
        char* ptr = strstr(response, "41 10");
        if (ptr) {
            int A = parseHexValue(ptr, 6, 2);
            int B = parseHexValue(ptr, 9, 2);
            maf = ((A * 256) + B) / 100.0;  // Convert to g/s
            return true;
        }
    }
    return false;
}

//------------------------------------------------------------------------------
// readThrottle(throttle)
//   - Queries PID 0x4A for absolute throttle position (%).
//   - Response: "41 4A AA", where position = (AA * 100) / 255.
// Parameters:
//   - throttle: Reference to an int to store percentage (0–100).
// Returns:
//   - true on success, false otherwise.
//------------------------------------------------------------------------------
bool OBD::readThrottle(int& throttle) {
    char response[64];
    if (sendPIDCommand("014A", response, sizeof(response))) {
        char* ptr = strstr(response, "41 4A");
        if (ptr && strlen(ptr) >= 8) {
            // Extract two hex digits after the header
            char hexVal[3] = { ptr[6], ptr[7], '\0' };
            int raw = strtol(hexVal, NULL, 16);
            throttle = (raw * 100) / 255;  // Scale to percent
            return true;
        }
    }
    Serial.println("Failed to parse absolute throttle data.");
    return false;
}

//------------------------------------------------------------------------------
// calculateInstantMPG(speed_kph, maf)
//   - Converts speed (km/h) to mph and MAF (g/s) to gallons per hour (gph).
//   - Computes instant MPG = mph / gph.
// Parameters:
//   - speed_kph: Vehicle speed in km/h
//   - maf:       Mass air flow in grams/sec
// Returns:
//   - Instant MPG as a float, or 0.0 if inputs invalid.
//------------------------------------------------------------------------------
float OBD::calculateInstantMPG(int speed_kph, float maf) {
    if (speed_kph > 0 && maf > 0) {
        float mph = speed_kph * 0.621317;     // Convert to miles/hour
        float gph = maf * 0.0805;             // Convert g/s to gal/hr
        return mph / gph;
    }
    return 0.0;
}

//------------------------------------------------------------------------------
// calculateAverageMPG(totalSpeedTimeProduct, totalFuelTimeProduct)
//   - Computes average MPG over a trip given the sum of (speed * delta_t)
//     and sum of (fuel_flow * delta_t).
// Parameters:
//   - totalSpeedTimeProduct: Sum of (speed * time interval)
//   - totalFuelTimeProduct:  Sum of (fuel flow * time interval)
// Returns:
//   - Average MPG, or 0.0 if no fuel consumed.
//------------------------------------------------------------------------------
float OBD::calculateAverageMPG(float totalSpeedTimeProduct, float totalFuelTimeProduct) {
    if (totalFuelTimeProduct > 0) {
        return totalSpeedTimeProduct / totalFuelTimeProduct;
    }
    return 0.0;
}
```

Figure 32: OBD-II data retrieval functions 2

The second task runs in the primary loop function, which is allocated to core 0. This handles the server, logging and OBD-II/GNSS initialisation (Figure 33)



```
// Main Loop: Handle Web Server, Button, and LED (runs on Core 0)
void loop() {
    server.handleClient();

    // Check for button press to initialise modules (only once)
    if (digitalRead(BUTTON_PIN) == LOW) {
        if (!gnssInitialized) {
            Wire1.setPins(SDA1, SCL1);
            Wire1.begin();
            if (myGNSS.begin(Wire1)) {
                Serial.println("GPS Module Initialized");
                myGNSS.setI2COutput(COM_TYPE_UBX);
                gnssInitialized = true;
            } else {
                Serial.println("Failed to initialize NEO-M8U Module");
            }
        }
        if (!obdInitialized) {
            if (obd.initialise()) {
                Serial.println("OBD-II Adapter Initialized");
                obdInitialized = true;
            } else {
                Serial.println("Failed to initialise OBD-II Adapter");
            }
        }
    }

    if (!isCalibrated) {
        // Before calibration is complete, blink the LED.
        static unsigned long previousMillis = 0;
        const unsigned long interval = 500;
        unsigned long currentMillis = millis();
        if (currentMillis - previousMillis >= interval) {
            previousMillis = currentMillis;
            digitalWrite(LED_PIN, !digitalRead(LED_PIN));
        }
    } else {
        // Once calibrated, the LED reflects the button state.
        loggingActive = (digitalRead(BUTTON_PIN) == LOW);
        digitalWrite(LED_PIN, loggingActive ? HIGH : LOW);
    }

    // Manages SD log file opening/closing when logging state changes.
    static bool lastLoggingState = false;
    if (loggingActive != lastLoggingState) {
        if (loggingActive) {
            // Logging just activated—prepare a new log file.
            firstLog = true;
            Serial.println("Logging activated.");
        } else {
            // Logging just deactivated—close the current log file if open.
            if (logFile) {
                if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000))) {
                    logFile.close();
                    xSemaphoreGive(sdMutex);
                    Serial.println("Log file closed.");
                }
            }
        }
        lastLoggingState = loggingActive;
    }
    delay(10);  // Short delay
}
```

Figure 33: Manage server and logging task

The function handles incoming web server requests and processes HTTP requests from clients (server.handleClient()).

Before calibration, the button LED blinks every 500ms, tracking the last toggle time for consistency. It toggles at 500ms intervals, updating previousMillis after each toggle. After calibration, the LED reflects the button state.

The function tracks changes in the button state . If logging starts, firstLog is set to true, preparing a new log file. If logging stops, the system acquires sdMutex, closes the log file if it is open, releases sdMutex, and updates lastLoggingState.

6.1.3 Test results

The tests are listed in table 13, with the results in table 16 (Appendix). Unit tests were done (test/test_main.cpp) using Unity Testing framework, and empirical Testing.

All tests passed successfully during this sprint. Vehicle data was empirically verified against dashboard readings. MAF and throttle position couldn’t be directly validated, as vehicles don’t

display these values. Instead, I used a Bluetooth ELM327 interface with the 'Car Scanner' app to compare MAF readings.

Testing dead reckoning was challenging due to the need to block the satellite signal. I resolved this by wrapping the GNSS antenna in tin foil, which successfully triggered the fallback and validated dead reckoning functionality.

Overall these tests allowed me to thoroughly validate Sprint-1 features and ensure all requirements were met. In hindsight, developing a custom testing harness could have ensured more complete coverage (Addressed in Evaluation).

6.2 Sprint two: User Interface and Data Communication

This sprint focused on developing the User interface and connecting the front end with the embedded system.

6.2.1 Server Implementation

The microcontroller hosts a HTTP web server on the IP 192.168.1.4 and port 80. I have implemented functions to set up the server and handle each endpoint, as explained below.

The setupServer() function initialises the web server, assigns request handlers, configures WiFi transmission power and starts the server with server.begin().

```
void setupServer() {
    Serial.println("Setting up Web Server...");

    server.on("/", HTTP_GET, handleRoot);
    server.on("/days", HTTP_GET, handleDays);
    server.on("/drives", HTTP_GET, handleDrives);
    server.on("/drive", HTTP_GET, handleDrive);
    server.on("/live", HTTP_GET, handleLiveData);
    server.on("/sdinfo", HTTP_GET, handleSDInfo);
    server.on("/delete", HTTP_OPTIONS, handleDeleteOptions);
    server.on("/delete", HTTP_DELETE, handleDelete);
    WiFi.setTxPower(WIFI_POWER_19_5dBm);  // Increase signal strength

    server.begin();
    Serial.println("Web server started.");
}
```

Figure 34: Setup server function

The handleRoot function verifies server connectivity, responding with "Connected".

```
void handleRoot() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    server.send(200, "text/plain", "Connected");
}
```

Figure 35: Manage server and logging task

The handleDays function lists recorded days by iterating through SD card directories. Before accessing the filesystem, it acquires the sdMutex. It then opens the root directory and iterates through its contents, filtering out hidden and system-related entries. Each valid folder name is added to a JSON array before the mutex is released and the response is sent.

```
//-------------------------------------------------------------------------------
// Handler for GET /days
// Lists top-level directories (YYYY-MM-DD folders) as JSON array
//-------------------------------------------------------------------------------
void handleDays() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    Serial.println("Listing available days...");

    // Lock SD card for safe multi-thread access
    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000)) == pdTRUE) {
        FsFile root = SD.open("/");
        if (!root) {
            // Couldn't open root directory
            Serial.println("Failed to open root directory");
            server.send(500, "text/plain", "Failed to open root directory");
            xSemaphoreGive(sdMutex);
            return;
        }

        String json = "[";
        bool first = true;

        // Iterate each entry in root
        while (true) {
            FsFile entry = root.openNextFile();
            if (!entry) break;                    // No more entries

            if (entry.isDir()) {
                char name[32];
                entry.getName(name, sizeof(name));
                // Skip hidden dirs starting with '.'
                if (name[0] != '.') {
                    if (!first) json += ",";
                    json += "\"" + String(name) + "\"";
                    first = false;
                }
            }
            entry.close();
        }
        json += "]";
        root.close();

        // Send JSON list of day folders
        server.send(200, "application/json", json);
        xSemaphoreGive(sdMutex);
    } else {
        // Mutex lock timed out
        Serial.println("SD Mutex timeout in handleDays()");
        server.send(500, "text/plain", "SD card access timeout");
    }
}
```

Figure 36: Handle days server function

```
//------------------------------------------------------------------------------
// Handler for GET /drives?day=YYYY-MM-DD
// Lists all JSON files (drives) under the specified day folder
//------------------------------------------------------------------------------
void handleDrives() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    if (!server.hasArg("day")) {
        // Missing required query parameter
        server.send(400, "text/plain", "Missing 'day' parameter");
        return;
    }

    String day = server.arg("day");
    Serial.print("Listing drives for day: ");
    Serial.println(day);

    String path = "/" + day;
    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000)) == pdTRUE) {
        FsFile dir = SD.open(path.c_str());
        if (!dir || !dir.isDir()) {
            // Folder doesn't exist
            Serial.println("Day folder not found");
            server.send(404, "text/plain", "Day folder not found");
            xSemaphoreGive(sdMutex);
            return;
        }

        String json = "[";
        bool first = true;
        while (true) {
            FsFile entry = dir.openNextFile();
            if (!entry) break;

            // Only include files (skip sub-directories)
            if (!entry.isDir()) {
                char name[32];
                entry.getName(name, sizeof(name));
                if (name[0] != '.') {
                    if (!first) json += ",";
                    json += "\"" + String(name) + "\"";
                    first = false;
                }
            }
            entry.close();
        }
        json += "]";
        dir.close();

        // Return JSON array of drive filenames
        server.send(200, "application/json", json);
        xSemaphoreGive(sdMutex);
    } else {
        Serial.println("SD Mutex timeout in handleDrives()");
        server.send(500, "text/plain", "SD card access timeout");
    }
}
```

Figure 37: Handle drives server function

The handleDrives function retrieves available drive logs for a specified day. Using the provided day parameter, it constructs the corresponding directory path from the request. The function then opens the folder, using the sdMutux, scans for files, and filters out sub-directories and hidden files. The filenames, representing recorded drive sessions, are compiled into a JSON array and returned as the response.

The handleDrive function retrieves the contents of a specific log file. It verifies day and drive parameters, constructs the file path, and reads its contents, holding the sdMutex. To prevent unauthorised access, it blocks filenames beginning with ".". The file is then streamed in 512 byte chunks and sent as a JSON response.

```
//------------------------------------------------------------------------------
// Handler for GET /drive?day=YYYY-MM-DD&drive=FILE.json
// Streams the contents of a specific drive file
//------------------------------------------------------------------------------
void handleDrive() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    if (!server.hasArg("day") || !server.hasArg("drive")) {
        server.send(400, "text/plain", "Missing 'day' or 'drive' parameter");
        return;
    }
    String day   = server.arg("day");
    String drive = server.arg("drive");

    // Prevent path traversal
    if (day.startsWith(".") || drive.startsWith(".")) {
        server.send(403, "text/plain", "Access forbidden");
        return;
    }

    String path = "/" + day + "/" + drive;
    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000)) == pdTRUE) {
        FsFile file = SD.open(path.c_str(), O_READ);
        if (!file) {
            server.send(404, "text/plain", "Drive file not found");
            xSemaphoreGive(sdMutex);
            return;
        }

        // Send headers, then stream file in chunks
        server.sendHeader("Content-Type", "application/json");
        server.setContentLength(file.size());
        server.send(200);

        const size_t bufSize = 512;
        uint8_t buf[bufSize];
        while (file.available()) {
            size_t n = file.read(buf, bufSize);
            server.sendContent((const char*)buf, n);
        }
        file.close();
        xSemaphoreGive(sdMutex);
    } else {
        Serial.println("SD Mutex timeout in handleDrive()");
        server.send(500, "text/plain", "SD card access timeout");
    }
}
```

Figure 38: Handle drive server function

```
//------------------------------------------------------------------------------
// Handler for GET /live
// Finds the most recent date folder and drive file, then streams its contents
//------------------------------------------------------------------------------
void handleLiveData() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    Serial.println("Fetching latest drive data...");

    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000)) == pdTRUE) {
        // 1) Scan root for latest YYYY-MM-DD folder
        FsFile root = SD.open("/");
        String latestDay;
        while (true) {
            FsFile e = root.openNextFile();
            if (!e) break;
            if (e.isDir()) {
                char n[32];
                e.getName(n, sizeof(n));
                // Simple lexicographical compare for YYYY-MM-DD
                if (strlen(n)==10 && n[4]=='-' && n[7]=='-'
                    && (latestDay.isEmpty() || String(n) > latestDay)) {
                    latestDay = n;
                }
            }
            e.close();
        }
        root.close();

        if (latestDay.isEmpty()) {
            server.send(404, "text/plain", "No log data found");
            xSemaphoreGive(sdMutex);
            return;
        }

        // 2) Scan that folder for latest HH-MM-SS*.json file
        FsFile dayDir = SD.open(("/" + latestDay).c_str());
        String latestDrive;
        while (true) {
            FsFile e = dayDir.openNextFile();
            if (!e) break;
            if (!e.isDir()) {
                char n[32];
                e.getName(n, sizeof(n));
                // Check for time-formatted name
                if (strlen(n)>=12 && n[2]=='-' && n[5]=='-' && strstr(n, ".json") &&
                    (latestDrive.isEmpty() || String(n) > latestDrive)) {
                    latestDrive = n;
                }
            }
            e.close();
        }
        dayDir.close();

        if (latestDrive.isEmpty()) {
            server.send(404, "text/plain", "No latest drive data found");
            xSemaphoreGive(sdMutex);
            return;
        }

        // 3) Stream that latest file
        FsFile file = SD.open(("/" + latestDay + "/" + latestDrive).c_str(), O_READ);
        if (!file) {
            server.send(404, "text/plain", "Latest drive file not found");
            xSemaphoreGive(sdMutex);
            return;
        }
        server.sendHeader("Content-Type", "application/json");
        server.setContentLength(file.size());
        server.send(200);
        const size_t bufSize = 512;
        uint8_t buf[bufSize];
        while (file.available()) {
            size_t n = file.read(buf, bufSize);
            server.sendContent((const char*)buf, n);
        }
        file.close();
        xSemaphoreGive(sdMutex);
    } else {
        Serial.println("SD Mutex timeout in handleLiveData()");
        server.send(500, "text/plain", "SD busy, try again later");
    }
}
```

Figure 39: Handle live-data server function

The handleLiveData function identifies the most recent log file by scanning for date-named directories and selecting the latest. It then locates the most recent file within that directory and reads its contents with the sdMutex. The response is streamed as a JSON to the client.

The handleSDInfo function retrieves the SD card status and calculates the total, used, and available storage as well as ESP32's uptime. It extracts filesystem details, including the SD card's total size and free space in bytes, then computes the used storage, whilst holding the sdMutex. This information is formatted as a JSON and sent to the client.

```
//------------------------------------------------------------------------------
// Handler for GET /sdinfo
// Reports SD card health and sizes, plus ESP32 uptime
//------------------------------------------------------------------------------
void handleSDInfo() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    Serial.println("Fetching SD diagnostics...");

    if (xSemaphoreTake(sdMutex, pdMS_TO_TICKS(1000)) == pdTRUE) {
        String json = "{";
        FsVolume* vol = SD.vol();  // Get volume metadata
        if (!vol) {
            // No card detected
            json += "\"sd_status\":\"Not detected\"";
        } else {
            // Calculate sizes
            uint32_t spc = vol->sectorsPerCluster();
            uint32_t cc  = vol->clusterCount();
            uint32_t fc  = vol->freeClusterCount();
            uint64_t total = (uint64_t)cc * spc * 512;
            uint64_t free  = (uint64_t)fc * spc * 512;
            uint64_t used  = total - free;

            json += "\"sd_status\":\"OK\","
                    "\"total_size\":" + String(total/1024.0/1024.0,2) + ","
                    "\"used_size\":"  + String(used /1024.0/1024.0,2) + ","
                    "\"free_size\":"  + String(free /1024.0/1024.0,2);
        }

        // Append ESP32 uptime in seconds
        json += ",\"esp32_uptime_sec\":" + String(millis()/1000) + "}";
        server.send(200, "application/json", json);
        xSemaphoreGive(sdMutex);
    } else {
        Serial.println("SD Mutex timeout in handleSDInfo()");
        server.send(500, "text/plain", "SD card access timeout");
    }
}
```

Figure 40: Handle SD info server function

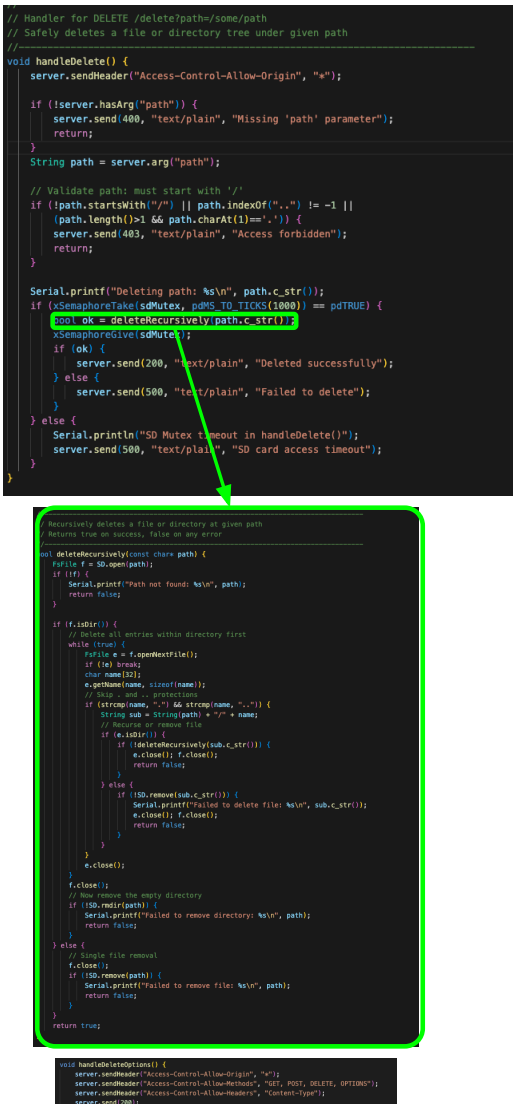


Figure 41: Handle Delete server function

The handleDelete function deletes files or directories. It verifies the path parameter and prevents unauthorised access by blocking ".." or paths starting with ".". The deleteRecursively function iterates through directory contents, deleting files and sub-directories before removing the main folder. It acquires a semaphore lock to ensure safe deletion.

The deleteRecursively function handles recursive deletion, opening a directory and iterating through its contents. It deletes files individually and is recursive for sub-directories. Once empty, the directory is removed. If deleting a single file, it attempts direct removal using SD.remove().

The handleDeleteOptions function sets HTTP headers for CORS, allowing DELETE requests from external clients. It responds with HTTP 200.

6.2.2 User Interface

The User Interface is created using Svelte and is packaged using Tauri.

6.2.2.1 Setup

The project is managed using npm, a package manager for JavaScript, which handles dependencies and builds for the application. Using npm I initialised the project:

1. **Installing Prerequisites**

   Before initialising the project, the following dependencies were installed**:**

Name	Rational	Command
Node.js & npm	Manages dependencies, runs Svelte.	Install from nodejs.org
Rust	Compiles Tauri backend.	curl https://sh.rustup.rs
Tauri CLI	Builds and packages Tauri.	cargo install tauri-cli
Tauri API	Connects Svelte to Tauri.	npm install @tauri-apps/cli @tauri-apps/api
Vite	Fast Svelte build tool.	Installed with npm create vite@latest my-app --template svelte

Table 9: App dependencies

2. **Initialising the Project**

```
npm create vite@latest my-tauri-svelte-app --template svelte
cd my-tauri-svelte-app
npm install
```

   This command initialises a new Svelte project with Vite

3. **Adding Tauri to the Project**

   Once the Svelte project was set up, Tauri was initialised using:

```
npx tauri init
```

4. **Running the Development Server**

   Once the setup was complete, the application was started in development mode using:

```
npm run tauri dev
```

This creates the following environment as shown in figure x, and explained in table x

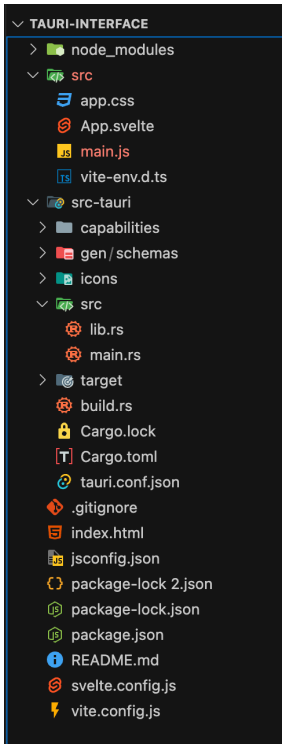


Figure 37: Tauri and Svelte environment

Name	Purpose
node_modules/	Stores npm dependencies.
src/ (app.css, app.svelte)	Holds Svelte components & logic.
src-tauri/ (capabilities/, gen/, icons/, target/, build.rs, tauri.conf.json...)	Contains Tauri backend files.
src-tauri/src/	Stores Rust source files.
index.html	Main HTML entry point.
jsconfig.json	Configures JS/TS support.
package-lock.json	Locks dependency versions.
package.json	Defines dependencies & scripts.
svelte.config.js	Configures Svelte.
vite.config.js	Configures Vite.

Table 10: Tauri/svelte folder/file purpose

With the Interface environment set up, the next step was importing the necessary modules:

```
import { onMount, tick } from "svelte";
import L from "leaflet";
import "leaflet/dist/leaflet.css";
import Chart from "chart.js/auto";
import { confirm } from '@tauri-apps/plugin-dialog';
```

Figure 38: Interface modules

Name	Purpose	Source
tick	Ensures UI updates after state changes.	Built-in (Svelte)
Leaflet	Renders interactive maps for GPS tracking.	npm install leaflet
Chart.js	Displays real-time charts for vehicle data.	npm install chart.js
Confirm	Manages user confirmation dialogs.	npm run tauri add dialog

Table 11: Interface imports

6.2.2.2 Interface Implementation

The interface consists of three display modes, all sharing a common structure. These displays are:

1. Post-Analysis Display – This is for reviewing past driving data.
2. Live Data Display – This is used to monitor real-time vehicle data.
3. Settings Display – This is for configuring preferences and managing data storage.

The following sections describe their fundamental interface features, followed by an in-depth look at each display tab.

6.2.2.3 Fundamental Interface Features

The interface layout is consistent across all three windows. It features a map display on the right, an options panel on the left and charts below. The options panel allows users to select displayed data and manage connection status.

The interface's core is an interactive map implemented using Leaflet. The map initialises via Svelte's onMount() function. It starts with a default view centered at [0, 0] and a zoom level of 2.

```
// Initialise the map
onMount(() => {
  map = L.map("map").setView([0, 0], 2);
  L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png", {
    attribution:
      '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors'
  }).addTo(map);
});
```

Figure 39: Map initialisation

Leaflet uses a tile-based system, loading sections dynamically as users pan and zoom. The tile layer is sourced from OpenStreetMap.

The options panel on the left updates based on user selection. It uses Svelte's bind:value to update selectedDay and on:change to trigger handleProcessingModeChange. If "Settings" is selected, fetchDeviceInfo retrieves relevant device details.

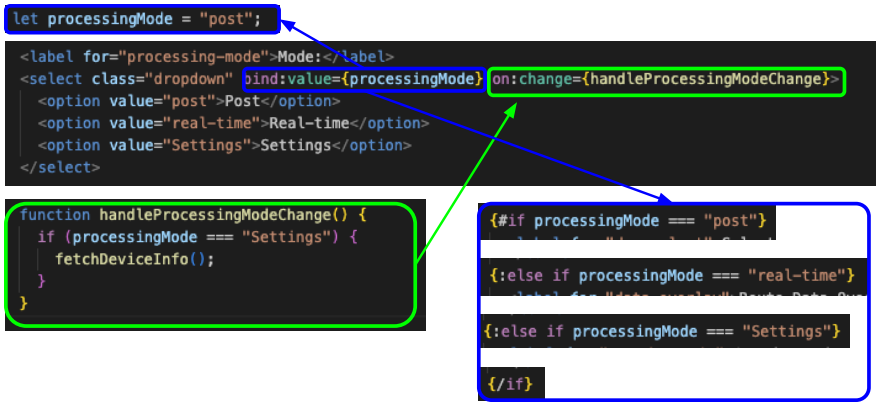


Figure 40: Options panel update

The system checks the connection on launch or “Retry” press, sending a request to the root endpoint, using checkConnection(). If successful, message updates to “Connected,” and data retrieval starts; otherwise, it stays “Not Connected.”

The interface dynamically displays the connection status: green for success, red for failure.

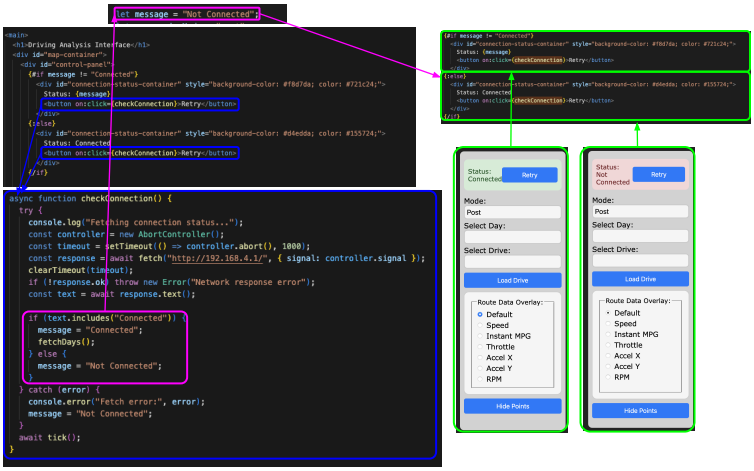


Figure 41: Interface Check connection function

6.2.2.4 Post analysis display

The Post-Analysis Display lets users review driving data by selecting a date and drive, then pressing "Load Drive" to display the route.

Logged data points appear along the route with a toggle to hide/show them. Users can customise the display with a heat map for metrics like Speed, Instant MPG, Average MPG, Throttle Position, RPM, Acceleration (X-axis), and Cornering (Y-axis).

Starting with drive selection:

Figure 42: Selecting Drive

The process begins with checkConnection(). If successful, fetchDays requests http://192.168.4.1/days, storing the JSON response. Svelte's {#each} block populates a dropdown. Selecting a day updates triggers fetchDrives function, which fetches drives from http://192.168.4.1/drives?day={selectedDay}. The response loads available drives into the drives dropdown using {#each}.

Now that the drive is selected, it needs to be loaded onto the map

The "Load Drive" button triggers loadDrive(), requesting http://192.168.4.1/drive?day=${selectedDay}&drive=${selectedDrive}. The NDJSON response is converted to JSON, and the map centres on the first recorded latitude and longitude.

updateRouteLine() draws the route with heatmap-style colours based on the selected metric. It does this by using the maximum and minimum values and setting the colour within the range.

updateMarkers() adds markers with driving metrics, visible only if showPoints is true, toggled via "Show/Hide Points".

Figure 43: Load drive functionality

The updateCharts() function extracts data from routeData, clears old charts, converts acceleration to m/s², and uses createChart() to generate eight charts with Chart.js.

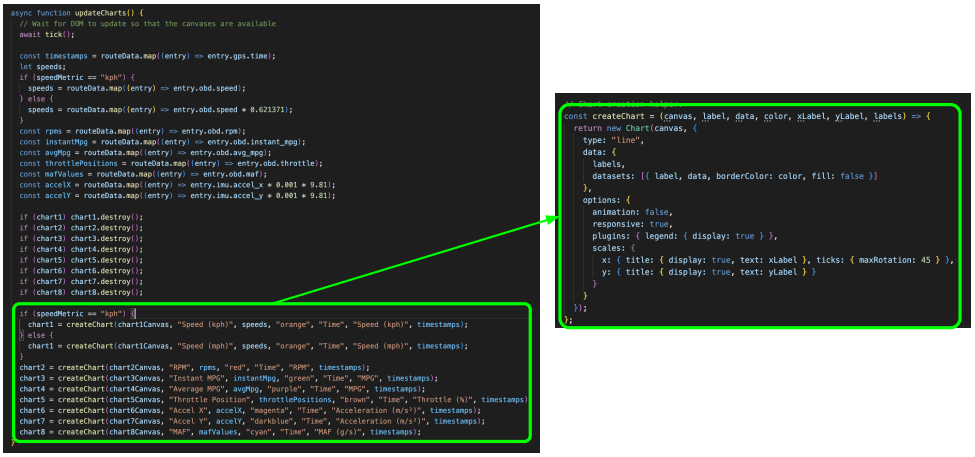


Figure 44: Chart creation

6.2.2.5 Real-time display

The live data display functionality handles real-time visualisation of vehicle data, dynamically updating metrics and map elements as new data is received.

```
// Manage live polling based on processing mode.
$: {
  if (processingMode === "real-time") {
    routeData = [];
    updateRouteLine();
    updateMarkers();

    if (!liveDataInterval) {
      checkConnection();
      fetchLiveData();
      liveDataInterval = setInterval(fetchLiveData, 2000);
    }
  } else {
    if (liveDataInterval) {
      clearInterval(liveDataInterval);
      liveDataInterval = null;
    }
  }
}
```

```
let liveTime = "";
let liveSpeed = "";
let liveRPM = "";
let liveInstantMPG = "";
let liveAvgMPG = "";
let liveThrottle = "";
let liveAccelX = "";
let liveAccelY = "";
```

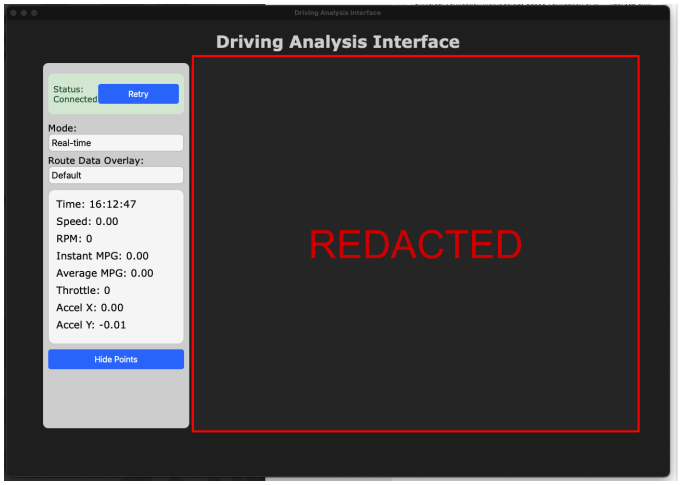
```
async function fetchLiveData() {
  try {
    console.log("Fetching live data...");
    const response = await fetch("http://192.168.4.1/live");
    if (!response.ok) throw new Error("Failed to fetch live data");
    const text = await response.text();
    const liveData = text
      .split("\n")
      .filter(line => line.trim() !== "")
      .map(line => JSON.parse(line));

    if (liveData.length > 0) {
      // Append new data if it's new
      if (
        routeData.length === 0 ||
        routeData[routeData.length - 1].gps.time !== liveData[0].gps.time
      ) {
        routeData = [...routeData, ...liveData];
        const latestPoint = liveData[liveData.length - 1].gps;
        map.setView([latestPoint.latitude, latestPoint.longitude], 16, { animate: true });
      } else {
        console.log("Duplicate data");
      }
    }

    await tick();

    if (routeData.length > 0) {
      const latest = routeData[routeData.length - 1];
      liveTime = latest.gps.time;
      if (speedMetric == "kph") liveSpeed = latest.obd.speed;
      if (speedMetric == "mph") liveSpeed = (latest.obd.speed * 0.621371).toFixed(2);
      liveRPM = latest.obd.rpm;
      liveInstantMPG = latest.obd.instant_mpg.toFixed(2);
      liveAvgMPG = latest.obd.avg_mpg.toFixed(2);
      liveThrottle = latest.obd.throttle;
      liveAccelX = (latest.imu.accel_x * 0.001 * 9.81).toFixed(2);
      liveAccelY = (latest.imu.accel_y * 0.001 * 9.81).toFixed(2);
    }

    updateRouteLine();
    updateMarkers();
    updateCharts();
  } catch (error) {
    console.error("Error fetching live data:", error);
  }
}
```

Figure 45: Real-time data functionality

When processingMode is "real-time," the system resets routeData, updates the map, and refreshes markers. If live data polling isn't running, it verifies the connection, fetches initial data via fetchLiveData(), and starts polling every two seconds.

fetchLiveData() requests data from http://192.168.4.1/live, converts NDJSON to JSON. The map re-centres on the latest coordinates. It updates speed, RPM, fuel efficiency (instant/average MPG), throttle position, and acceleration (m/s²). Finally, updateRouteLine(), updateMarkers(), and updateCharts() refresh the map, markers, and charts in real time.

The data is displayed to the interface



[3]

Figure 46: Real-time data display

[3] The unloaded tile in Figure 46 is discussed in the Evaluation section.

6.2.2.6 Settings display

When processingMode is set to "Settings," the interface provides options for configuring speed metrics, viewing SD card details and managing stored drive data.

Users can choose between miles per hour (MPH) or kilometers per hour (KPH) for speed display using a drop-down menu. This selection updates the speedMetric variable, which is referenced whenever speed values are shown.

The fetchDeviceInfo() function retrieves SD card details from http://192.168.4.1/sdinfo and converts the response into JSON. When successful, it replaces the default "Loading..." placeholder with storage information. If the request fails, all values are set to "Fail".

```
let speedMetric = "mph";

let sdStatus = "Loading...";
let totalSize = "Loading...";
let usedSize = "Loading...";
let freeSize = "Loading...";
let upTime = "Loading...";
```

```
function handleProcessingModeChange() {
  if (processingMode === "Settings") {
    fetchDeviceInfo();
  }
}

async function fetchDeviceInfo() {
  try {
    console.log("Fetching SD card info...");
    const response = await fetch("http://192.168.4.1/sdinfo");
    if (!response.ok) throw new Error("Failed to fetch SD info");

    const data = await response.json();
    sdStatus = data.sd_status;
    totalSize = data.total_size.toFixed(2) + " MB";
    usedSize = data.used_size.toFixed(2) + " MB";
    freeSize = data.free_size.toFixed(2) + " MB";
    upTime = data.esp32_uptime_sec + " Seconds";
  } catch (error) {
    console.error("Error fetching SD info:", error);
    sdStatus = "Fail";
    totalSize = "Fail";
    usedSize = "Fail";
    freeSize = "Fail";
    upTime = "Fail";
  }
  await tick();
}
```

```
{:else if processingMode === "Settings"}
  <label for="speed-metric">Speed metric: </label>
  <select name="speed-metric" class="dropdown" bind:value={speedMetric}>
    <option value="mph">MPH</option>
    <option value="kph">KPH</option>
  </select>
  <label for="sdinfo">SD card details: </label>
  <div class="sd-info-box">
    <p><b>SD Card Status:</b> {sdStatus}</p>
    <p><b>Total Size:</b> {totalSize}</p>
    <p><b>Used Size:</b> {usedSize}</p>
    <p><b>Free Size:</b> {freeSize}</p>
    <p><b>Up-time:</b> {upTime}</p>
  </div>

  <div class="delete-management">
    <div>
      <label for="delete-day-select">Select Day:</label>
      <select id="delete-day-select" class="dropdown" bind:value={selectedDay} on:change={fetchDrives}>
        {#each days as day}
          <option value={day}>{day}</option>
        {/each}
      </select>
    </div>
```

Figure 47: Settings display (fetchDeviceInfo())

For data management, users can browse and select specific drives using the same functions in post-analysis mode and delete drives or entire day folders.

Figure 48: Delete days/drives

The deleteDayFolder() function enables bulk deletion of all drives stored on a selected day. After prompting the user for confirmation, it sends a DELETE request to: http://192.168.4.1/delete?path=/<selectedDay> If the deletion is successful, the system refreshes the list of available days and associated drives to reflect the changes.

The deleteDriveFile() function is used to remove a specific drive file within a selected day's folder. Once the user confirms the action, it sends a DELETE request to: http://192.168.4.1/delete?path=/<selectedDay>/<selectedDrive> to delete the file. Upon successful deletion, the system updates the list of available drives.

6.2.3 Compiling the Application

To compile the application into executables I ran the following commands:

Mac (.dmg)

```
npm run tauri build
```

Windows (.exe)

```
rustup target add x86_64-pc-windows-gnu
```

```
npm run tauri build -- --target x86_64-pc-windows-gnu
```

6.2.4 Test results

The tests are listed in table 14, with the results in table 17 (Appendix). Tests were run using playwright testing library and covered Unit, Widget and integration tests (tauri-interface/tests/).

Overall, all tests passed successfully during this sprint. While testing edge cases, initial large file integration tests failed due to attempting to buffer entire files in memory before sending them to the endpoint. On large files, this exceeded the microcontroller's RAM and caused failures. To resolve this, I implemented a chunked streaming approach (GitHub Commit), allowing large files to be uploaded in smaller parts, solving this issue.

I experimented with several testing libraries, including Jest and Vitest, but ultimately chose Playwright for its end-to-end testing capabilities. Playwright enabled me to verify widget rendering, test dynamic UI elements, and fully validate all API endpoints to ensure data updates were correctly reflected in the app.

I found that the microcontroller and OBD module heated up significantly when exposed to direct sunlight on the dashboard (addressed in the evaluation).

Overall, these tests allowed me to thoroughly validate Sprint-2 features and ensure all requirements were met.

xx6.3 Sprint Three: Embedded system enclosure and circuit design

During this sprint, I focused on transferring the components from the prototype wooden block to perfboards (perforated boards) and designing an enclosure to house the entire system.

6.3.1 Circuit

To begin, I measured the internal dimensions of the enclosure based on the layout of the prototype wood block. After I disassembled the prototype for permanent assembly on the perfboards.

I used two 5 x 7 cm perfboards, stacked on top of each other with M3 nylon standoffs to secure them together. The lower perfboard houses the microcontroller and SD card breakout board. The upper perfboard holds the GNSS module, which is mounted using nylon standoffs, and the antenna is hot-glued down. As shown in Figure 49.

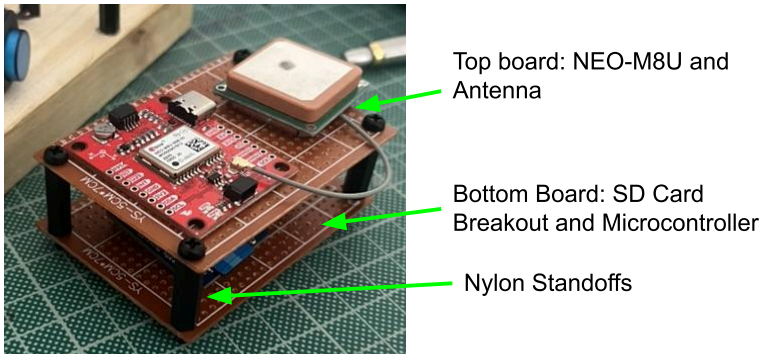


Figure 49: Circuit stack

I then soldered all the connections to the microcontroller, secured the GNSS module using nylon standoffs, and fixed the antenna in place with hot glue.

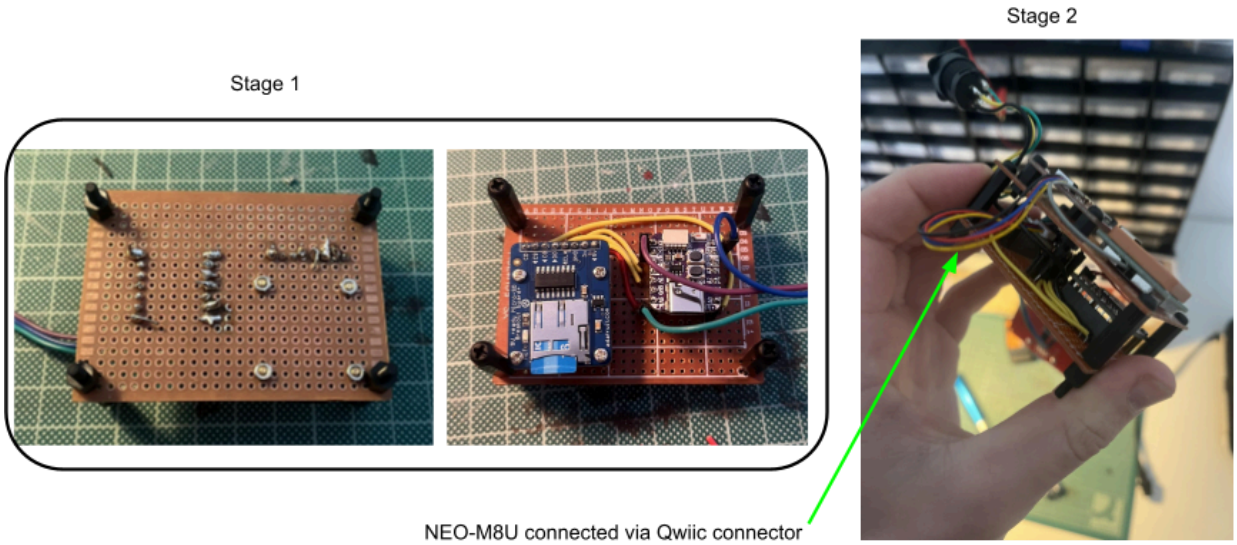


Figure 50: Soldered circuit stack

The OBD-II Module is also secured using the Nylon stand-offs and the connector has been hot-glued to remain connected.

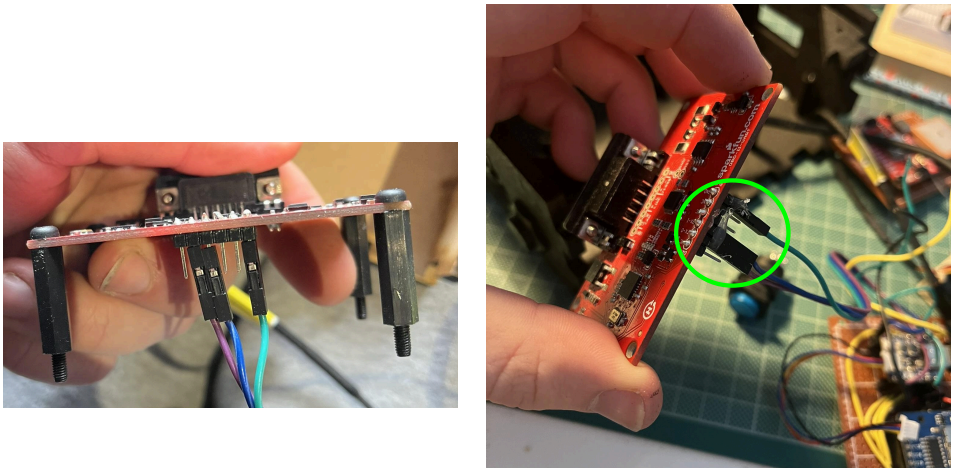


Figure 51: OBD-II Connection

Using this setup, I can maximise the enclosure space whilst being able to securely mount the components inside the enclosure.

6.3.2 Enclosure

The enclosure was designed using a template found on Boxes.py (Hackerspace Bamberg, 2025). This template allowed me to create a laser-cuttable enclosure with interlocking finger joints and a removable lid. The design was then customised to accommodate the embedded system components, including ventilation slots for heat dissipation and a hole for system power input.

The final measurements for the enclosure were:

- Width (X): 157 mm
- Depth (Y): 107 mm
- Height (H): 56 mm

Once the design was completed, the enclosure was exported as a DXF file for laser cutting, as seen in Figure 52.

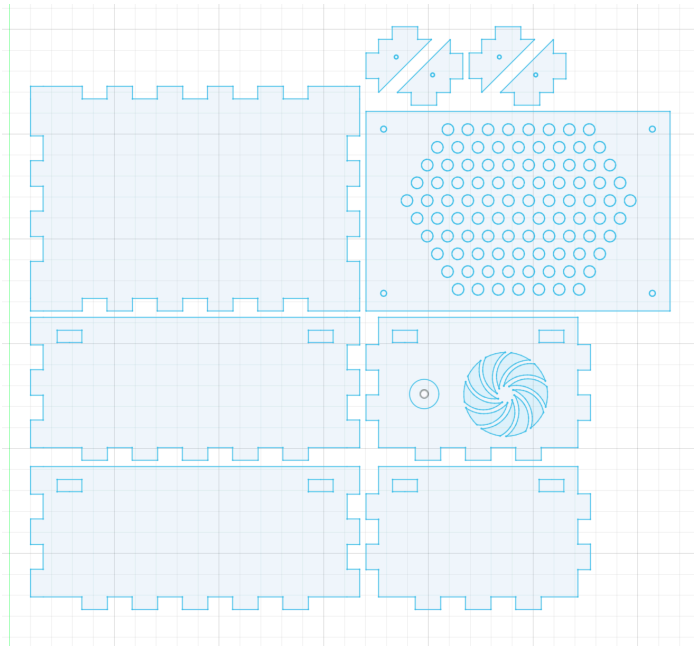


Figure 52: Enclosure design

I added the cutouts for the OBD-II module and push button after laser cutting to ensure they were in the correct position and did not conflict with other components inside the enclosure.

Once the pieces were cut, I began assembling the enclosure. This proved challenging initially due to the burn tolerances from the laser cutting process, which caused some parts to fit too tightly. To resolve this, I filed down the edges. After these adjustments, the enclosure was constructed, as seen in Figure 53.

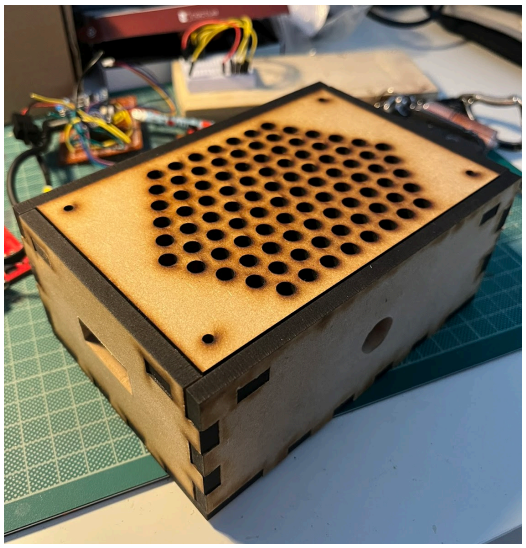


Figure 53: Constructed enclosure

With the enclosure assembled, I aligned the OBD-II module interface with the end piece and marked the area that needed to be cut to ensure a proper fit.

Figure 54: OBD-II Interface measurements

To create the cutout, I drilled two holes on either side of the marked area and used a file to remove the excess material, as shown in Figure 54.

I used the same process to create a cutout for the push button on the front plate. I also aligned the fan with its designated cutout on the left-hand end plate, marking the positions for the mounting holes. Using a 3mm drill bit, I drilled four holes for the fan mounting.

To secure the enclosure, I glued the finger joints together using wood glue on all joints. I then used a clamp to hold the structure in place while the glue dried. Additionally, I used M3 nylon screws to fasten the lid in place.

Finally, to enhance the enclosure's appearance and provide a more professional finish, I spray-painted it black.

6.3.3 Integration of Circuit and Enclosure

To integrate the perforated circuit board stack into the enclosure, I first positioned the components inside and marked the locations for drilling the mounting holes. Using a 3mm drill bit, I drilled the marked holes. This drill bit size was ideal, as it allowed the screw threads to securely grip.

The stand-off screws used for securing the circuit are 6mm in length, which is the same depth as the MDF material. This presented a challenge, as the screws would fully penetrate the wood.

To solve this, I added a nut onto the stand-off thread, reducing the thread length and preventing it from going through the material. Additionally, when drilling the holes for the stand-offs, I ensured only to drill 3mm deep.

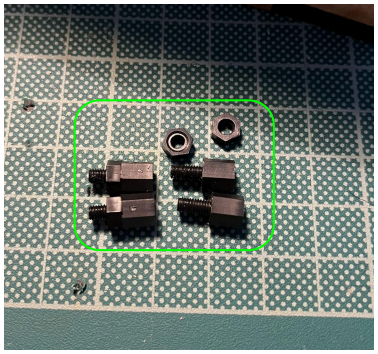


Figure 55: Thread length solution

I encountered a similar issue with the fan mounting, where the screw thread was too long. To resolve this, I trimmed the screw thread to the appropriate length, ensuring a proper fit.

Lastly, I purchased a USB-C panel mount cable to allow external power access to the system. The female end of the cable was screwed on to the exterior of the enclosure, providing a power input, while the male end was connected directly to the microcontroller inside the enclosure. This setup allows for easy and secure power access without needing to open the enclosure.

Figure 56: USB C Panel Mount Cable (Ebay, 2025)

6.3.4 Enclosure mounting

To keep the enclosure stable during driving, I place it on a non-slip mat, which provides enough grip to keep the enclosure secure while driving, as shown below:

Figure 57: Enclosure mounting

6.3.5 Test results

The tests are listed in table 15, with the results in table 18 (Appendix)

There were fewer tests to perform in this sprint as it focused primarily on hardware. The tests I conducted were designed to ensure the enclosure met all requirements including durability, temperature, secure mounting, and external accessibility.

On reflection, these tests allowed me to validate the enclosure's real-world suitability and ensure it could withstand typical automotive conditions.

7 Evaluation

I began with the literature review by researching existing automotive monitoring technologies and the problem background. With guidance from my supervisor, I then investigated the most suitable technology stack for implementation for both my app and embedded system.

In hindsight, using Tauri and Svelte was an excellent choice, as Tauri enabled me to leverage my existing experience in web development, while Svelte allowed for the integration of dynamic components into the application and improved my technical ability. These tools were also utilised as my supervisor advised me they would be valuable tools to have experience with for the future.

On the other hand, for the embedded system, the initial microcontroller chosen was QT-PY ESP32 S2, chosen for its availability, but during development I found that the single-core processor could not handle logging and serving clients concurrently. To solve this, I upgraded to the QT-PY ESP32 S3, during sprint-2, which had a dual-core processor which, when using RTOS, was capable of running the system. I found this very beneficial, as it gave me the opportunity to explore and learn how to use an RTOS.

Within the Literature Review I explored some ideas that I did not fulfil:

1. I explored the impact of engine idling on fuel consumption and initially planned to identify instances of unnecessary idling. However, further research revealed that accurately detecting unnecessary idling, such as distinguishing it from idling at traffic lights, would be highly complex (address in Section 8).
2. I researched how to manually use Dead Reckoning using a Kalman filter. This was not necessary as I selected the NEO-M8U which had these calculations embedded in. This was extremely beneficial as I was limited on microcontroller processing power.

In terms of the requirements, while I aimed to have as many MUST requirements as possible, it was not feasible due to time constraints. I completed all except FR-14 and FR-15 due to technical difficulties (addressed in section 7.1). Remarkably, I achieved NFR-03, which relates to supporting a wide range of OBD-II protocols. This was enabled by the SparkFun OBD-II UART module. Although I was only able to test the system with two vehicles, the module's datasheet indicates compatibility with all OBD-II protocols. Comprehensive verification would require a dedicated testing harness, which was beyond this project's scope.

Next was the methodology section. Each sprint was initially planned based on similarities in the requirements. However, upon reflection, I found that sprint two was significantly heavier than sprints one and three. In hindsight, I should have introduced an additional sprint between sprints two and three. This would have helped balance the workload and accounted for the extra time spent on implementing data communication.

Lastly, was the product design and implementation section. The design was the most beneficial section, as it allowed me to discuss my design ideas with my supervisor and to develop a plan. Once designed, I started development. The part I'm most proud of is the data communication

system, which enables the embedded system to communicate with the user interface. This feature required a great deal of consideration during the design phase. Initially, I planned to use BLE (Bluetooth Low Energy) for communication but after further research, I found it was not powerful enough to reliably transfer large volumes of data. Switching to Wi-Fi proved to be the right decision, offering stable performance with minimal processing overhead.

The cooling system was a late addition, introduced after I noticed a significant temperature increase in the components during Sprint two testing, especially when exposed to direct sunlight. The added cooling system was then tested in Sprint three which solved this issue.

To handle communication, my supervisor set up a Discord server, which made it easy to ask questions with fast responses and allowed me to learn from my peers' challenges. We also had regular face-to-face meetings to discuss my progress, where I received valuable feedback.

The most beneficial feedback I received was regarding my enclosure. I initially thought 3D Printing would be the most suitable construction method, but after discussing with my supervisor, he recommended laser cutting with finger joints. This was due to using nylon standoff which required being screwed into a surface and 3D printing threads or tapping was not viable and would affect the component mounting strength. As testing proved, the standoffs were an ideal decision as they can securely hold the components when driving. Furthermore, the chosen MDF is a much more rigid material, making for an overall sturdier enclosure.

Although I only spoke with my second marker once during the Project-in-Progress day, the feedback was still valuable. She challenged some of my design ideas, helping me ensure that I had covered all aspects and thoroughly considered my design decisions.

7.1 Limitations and solutions

Below are some issues I encountered that could have been resolved but were not possible within my time frame or due to technical limitations:

Issue	Solution
Offline map tile display (FR-14): Not implemented due to storage and download limits.	The default Leaflet tile provider prohibits bulk downloading (OSMF OWG, n.d.). An alternative would be using a paid service like Mapbox, which supports offline usage, and dynamically handling the map tiles though this exceeds the scope and budget of the project. Furthermore this would significantly increase the size of the application.
Auditory alerts for speeding or harsh driving (FR-15): Auditory alerts not implemented due to hardware constraints and conflicts	A simple buzzer could be integrated into the embedded system to provide basic alerts without requiring audio playback. However, this approach was reconsidered, as auditory alerts may distract the driver.

with SD card access for sound file storage.	
Real-time speed limit data: Unavailable due to lack of internet access while using the access point for system communication.	Use an API like OpenStreetMap to access real-time speed limits, but this would require an active internet connection. To solve this, the embedded system communication would need to be redesigned

Table 12: Project limitations and solutions

8 Further Work and Conclusion

During development, there were some improvements that were either outside the scope or were not achievable within my time frame. These would have improved the system's overall functionality listed below:

1. **Encryption**: Currently, anyone can access data on the SD card with the password to the access point. To solve this, I could implement encryption for stored data using API keys. This ensures that, even if someone connects to the access point, the data remains unreadable. The embedded system would handle encryption and the interface would perform decryption using a shared secret key stored securely in both systems. A crypto library such as TinyAES could be used on the embedded side. However, it would be challenging to implement due to processing power constraints on the microcontroller.
2. **UI device internet access:** Currently, the UI device cannot access an internet connection while connected to the access point, stopping any new map tiles from loading and making real-time speed limit data inaccessible. This limitation can be addressed using one of the following methods:
    a. **Shared internet connection -** The microcontroller and the UI device connect to the same external Wi-Fi network. This allows the microcontroller to serve data while the UI maintains internet access.
    b. **Cellular Mobile Data -** By integrating a mobile data module, the system can access the internet independently and share this connection through its access point. This allows the UI to stay connected and access online resources while communicating with the embedded system.
3. **Idling Detection:** The current system cannot detect idling and differentiate between necessary and unnecessary idling. This could be solved in future work by training a machine learning model on GNSS and vehicle data to recognise patterns of valid stops (e.g. traffic lights) versus unnecessary idling.

In conclusion, the system successfully achieved its primary objective, which is to design and develop an automotive monitoring system aimed at reducing emissions and improving fuel economy. The final implementation is capable of logging driving data and communicating with an app which can deliver actionable feedback to the user. However, there remains room for further improvement. With additional development and refinement, the system has the potential to evolve into a widely accessible product that could benefit companies and individuals.

9 Appendix

9.1 Sprint one tests

Test ID	Test Type	Test Description	Test Steps	Expected Results	Requirements Tested
T001	Unit	Verify button is not pressed	Read button state	HIGH	FR-06
T002	Unit	Turn LED on and verify state	Set LED to HIGH, read state	LED is HIGH	FR-07
T003	Unit	Turn LED off and verify state	Set LED to LOW, read state	LED is LOW	FR-07
T004	Unit	Verify GNSS initialisation	Initialise GNSS module	Initialisation successful	FR-03
T005	Unit	Retrieve GNSS location data	Read latitude and longitude	Values within valid ranges	FR-03
T006	Unit	Verify IMU data availability	Retrieve IMU acceleration data	Values within expected range	FR-04
T007	Unit	Verify dead reckoning when no satellites available	Check GNSS output with 0 satellites	Lat/Lon should be 0.0	FR-04
T008	Unit	Verify OBD initialisation	Call obd.initialize()	Returns true or false	FR-01
T009	Unit	Read engine RPM	Retrieve RPM from OBD	RPM > 0 if engine is on	FR-01
T010	Unit	Read vehicle speed	Retrieve speed from OBD	Speed >= 0	FR-01

T011	Unit	Read mass air flow (MAF)	Retrieve MAF from OBD	MAF > 0 if engine is running	FR-01
T012	Unit	Read throttle position	Retrieve throttle position from OBD	Throttle between 0-100%	FR-01
T013	Unit	Calculate instant MPG	Call calculateInstantMPG (100, 10)	MPG > 0 for valid inputs	FR-02
T014	Unit	Calculate average MPG	Call calculateAverageMPG(1000, 10)	MPG > 0 for valid inputs	FR-02
T015	Unit	Initialise SD card	Call SD.begin()	Initialization successful	FR-05
T016	Unit	Create directory	Create folder on SD	Directory exists	FR-05
T017	Unit	Create JSON file	Create file on SD	File is created successfully	FR-05
T018	Unit	Write to JSON file	Write test data to JSON file	Data written successfully	FR-05
T019	Unit	Read from JSON file	Read data from file	Expected data is read	FR-05
T020	Unit	Delete JSON file	Delete test file from SD	File no longer exists	FR-05, NFR-05
T021	Unit	Delete directory	Remove test folder	Folder no longer exists	FR-05, NFR-05
T022	Unit	Simulate full SD-card write failure	Repeatedly open/write to a test file until SD.open() or file.print() returns false	The code detects “disk full” (open/write returns false) and handles it gracefully	FR-05

T023	Unit	Power-loss during JSON write	Begin writing partial JSON, call SD.begin() mid-write to simulate power-reset, then close and remount	Filesystem remains mountable and the file still exists after remount	FR-05, NFR-07
T024	Unit	Concurrent SD-card read/write access	While appending JSON lines to a file, intermittently open it for read and check its size	No lockup; reader opens successfully each time and reports size > 0	FR-05
T025	Empirical	Verify GNSS location accuracy	Compare GNSS readings to google maps	Error margin within ±5m	FR-03
T026	Empirical	Verify vehicle speed accuracy	Compare OBD speed to dashboard reading	Difference within ±2 km/h	FR-01
T027	Empirical	Verify RPM readings	Compare OBD RPM to vehicle tachometer	RPM values match within 100 RPM	FR-01
T028	Empirical	Verify throttle position accuracy	Press accelerator and compare OBD values	OBD throttle matches pedal position	FR-01
T029	Empirical	Verify MAF sensor readings	Compare OBD MAF data to existing OBD code reader	MAF values align with expected ranges	FR-01
T030	Empirical	Verify MPG calculation	Compare OBD-calculated MPG to vehicle calculated MPG	Calculated MPG within ±5%	FR-02
T031	Empirical	Verify IMU acceleration accuracy	Compare IMU acceleration data to accelerations and decelerations	IMU values correspond correct	FR-04

T032	Empirical	Verify SD card data logging and accuracy	Compare logged timestamps with vehicle clock	Timestamp error within ±3 seconds	FR-05, NFR-08

Table 13: Sprint one tests

9.2 Sprint two tests

Test ID	Test Type	Test Description	Test Steps	Expected Results	Requirements Tested
T033	Unit	Verify UI renders correctly	Load page and check if key elements are present	All UI elements are displayed as expected	NFR-01
T034	Unit	Verify "post" mode UI elements	Select "post" mode and check visible elements	Elements specific to "post" mode are present	FR-10
T035	Unit	Verify "real-time" mode UI elements	Select "real-time" mode and check visible elements	Elements for real-time data are visible	FR-10
T036	Unit	Verify "Settings" mode UI elements	Select "Settings" mode and check visible elements	Settings panel elements are displayed	FR-11
T037	Unit	Verify speed metric toggle	Switch between KPH and MPH display modes and observe speed reading	Speed updates and displays correctly in selected unit	FR-12
T038	Unit	Verify toggle points button changes text	Click "Hide Points" button and check if it updates	Button text changes to "Show Points"	FR-08

T039	Widget	Verify connection status updates correctly	Click retry button and check connection status	Status updates to "Connected"	FR-10
T040	Widget	Verify drop-down selection changes view	Change modes using the dropdown	UI updates correctly for each mode	FR-10
T041	Widget	Verify live data container updates correctly	Switch to real-time mode and observe data changes	Live data updates dynamically	FR-10
T042	Widget	Verify settings dropdown options persist	Change a setting and reload the page	Setting remains as selected	FR-11
T043	Widget	Verify graph display of collected data	Open post view in the app and check if speed, RPM, throttle, and MPG are plotted	Graphs are rendered correctly with appropriate axes and data points	FR-09
T044	Integration	Verify /days API endpoint returns valid response	Send GET request to /days	Returns status 200 with a valid list	FR-11, FR-13
T045	Integration	Verify /drives API endpoint returns valid response	Send GET request to /drives?day=test	Returns status 200 with a valid list	FR-11, FR-13

T046	Integration	Verify /drive API endpoint returns valid data	Send GET request to /drive?day=test&drive=dummy.json	Returns status 200 with properly formatted JSON	FR-11, FR-13
T047	Integration	Verify /sdinfo API endpoint returns diagnostics	Send GET request to /sdinfo	Returns status 200 with storage and uptime details	FR-11, FR-13
T048	Integration	Verify drive file exists before deletion	Send GET request to /drive?day=test&drive=dummy.json	Returns status 200 if file exists	FR-11, FR-13, NFR-05
T049	Integration	Verify drive file deletion	Send DELETE request to /delete?path=/test/dummy.json	Returns status 200 with "Deleted successfully" message	FR-11, FR-13, NFR-05
T050	Integration	Verify drive file is not found after deletion	Send GET request to /drive?day=test&drive=dummy.json after deletion	Returns status 404 (file not found)	FR-11, FR-13, NFR-05
T051	Empirical	Verify cross-platform compatible	Run the application executable on both macOS and Windows systems	Application launches and functions correctly without errors on both platforms	NFR-02

Table 14: Sprint two tests

9.3 Sprint three tests

Test ID	Test Type	Test Description	Test Steps	Expected Results	Requirements Tested
T052	Empirical	Verify enclosure temperature remains within safe range	Park vehicle in direct sunlight for 2+ hours and monitor internal temperature using an external thermometer	Temperature stays below 85°C	NFR-09
T053	Empirical	Verify components remain secure during driving	Mount enclosure in vehicle, drive over varied surfaces, then inspect internal components	No components are loose or damaged	NFR-04
T054	Empirical	Verify enclosure stays fixed in place during motion	Drive with sudden stops and over bumps; inspect mounting after drive	Enclosure remains firmly mounted	NFR-06
T055	Empirical	Verify external IO access without disassembly	Attempt to plug into USB and OBD connector while enclosure is closed	All IO ports accessible without opening case	NFR-04

Table 15: Sprint three tests

9.4 Sprint one Results

Test ID	Result	Comments
T001	Pass	Tests passed as expected
T002	Pass	Tests passed as expected
T003	Pass	Tests passed as expected
T004	Pass	Tests passed as expected
T005	Pass	Tests passed as expected

T006	Pass	Tests passed as expected
T007	Pass	Tests passed as expected
T008	Pass	Tests passed as expected
T009	Pass	Tests passed as expected
T010	Pass	Tests passed as expected
T011	Pass	Tests passed as expected
T012	Pass	Tests passed as expected
T013	Pass	Tests passed as expected
T014	Pass	Tests passed as expected
T015	Pass	Tests passed as expected
T016	Pass	Tests passed as expected
T017	Pass	Tests passed as expected
T018	Pass	Tests passed as expected
T019	Pass	Tests passed as expected
T020	Pass	Tests passed as expected
T021	Pass	Tests passed as expected
T022	Pass	Tests passed as expected
T023	Pass	Tests passed as expected
T024	Pass	Tests passed as expected
T025	Pass	Tests passed as expected
T026	Pass	Tests passed as expected
T027	Pass	Tests passed as expected
T028	Pass	Tests passed as expected
T029	Pass	Tests passed as expected
T030	Pass	Tests passed as expected

T031	Pass	Tests passed as expected
T032	Pass	Tests passed as expected

Table 16: Sprint one tests results

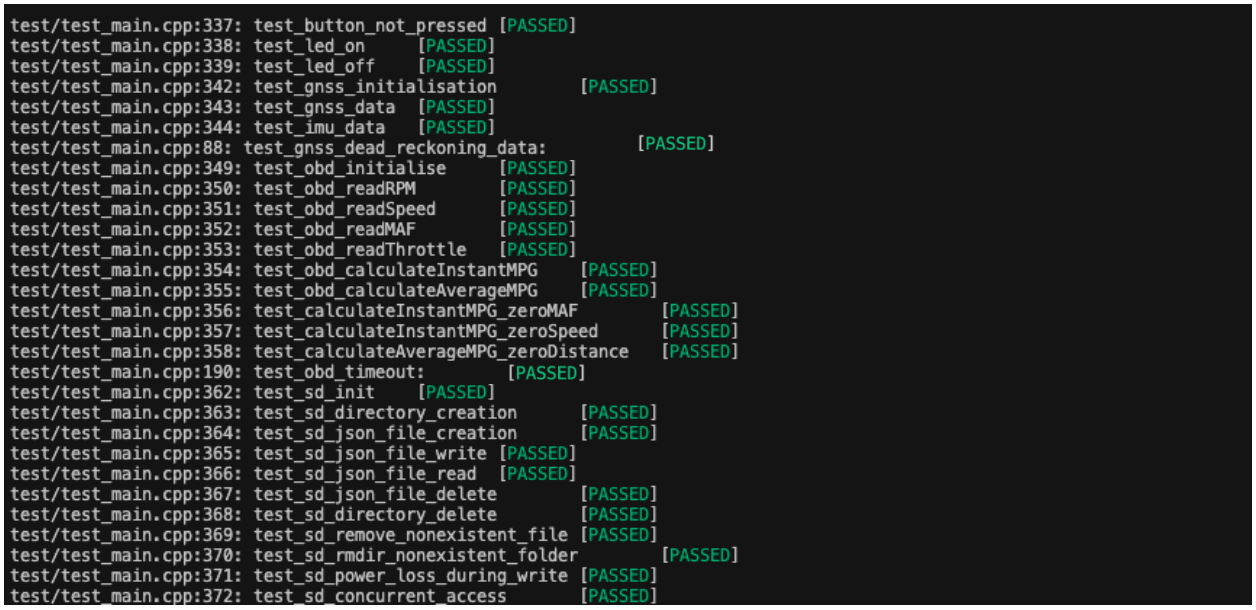


Figure 58: Sprint one tests evidence

9.5 Sprint two Results

Test ID	Result	Comments
T033	Pass	Tests passed as expected
T034	Pass	Tests passed as expected
T035	Pass	Tests passed as expected
T036	Pass	Tests passed as expected
T037	Pass	Tests passed as expected
T038	Pass	Tests passed as expected
T039	Pass	Tests passed as expected
T030	Pass	Tests passed as expected
T041	Pass	Tests passed as expected

T042	Pass	Tests passed as expected
T043	Pass	Tests passed as expected
T044	Pass	Tests passed as expected
T045	Pass	Tests passed as expected
T046	Pass	Tests passed as expected
T047	Pass	Tests passed as expected
T048	Pass	Tests passed as expected
T049	Pass	Tests passed as expected
T050	Pass	Tests passed as expected
T051	Pass	Tests passed as expected

Table 17: Sprint two test results

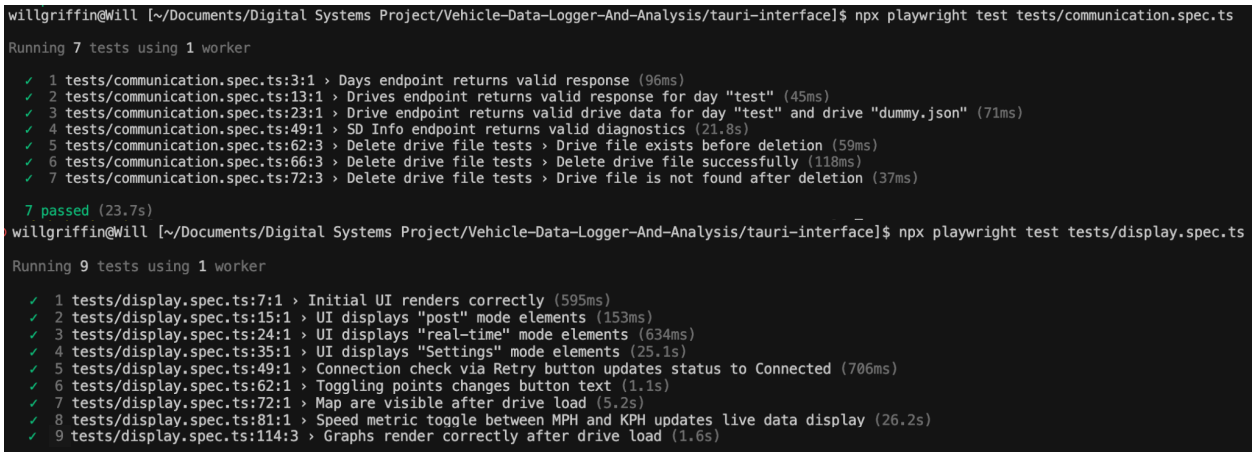


Figure 59: Sprint two tests evidence

9.6 Sprint three results

Test ID	**Result**	**Comments**
T052	Pass	Tests passed as expected
T053	Pass	Tests passed as expected
T054	Pass	Tests passed as expected

T055	Pass	Tests passed as expected

Table 18: Sprint three test results

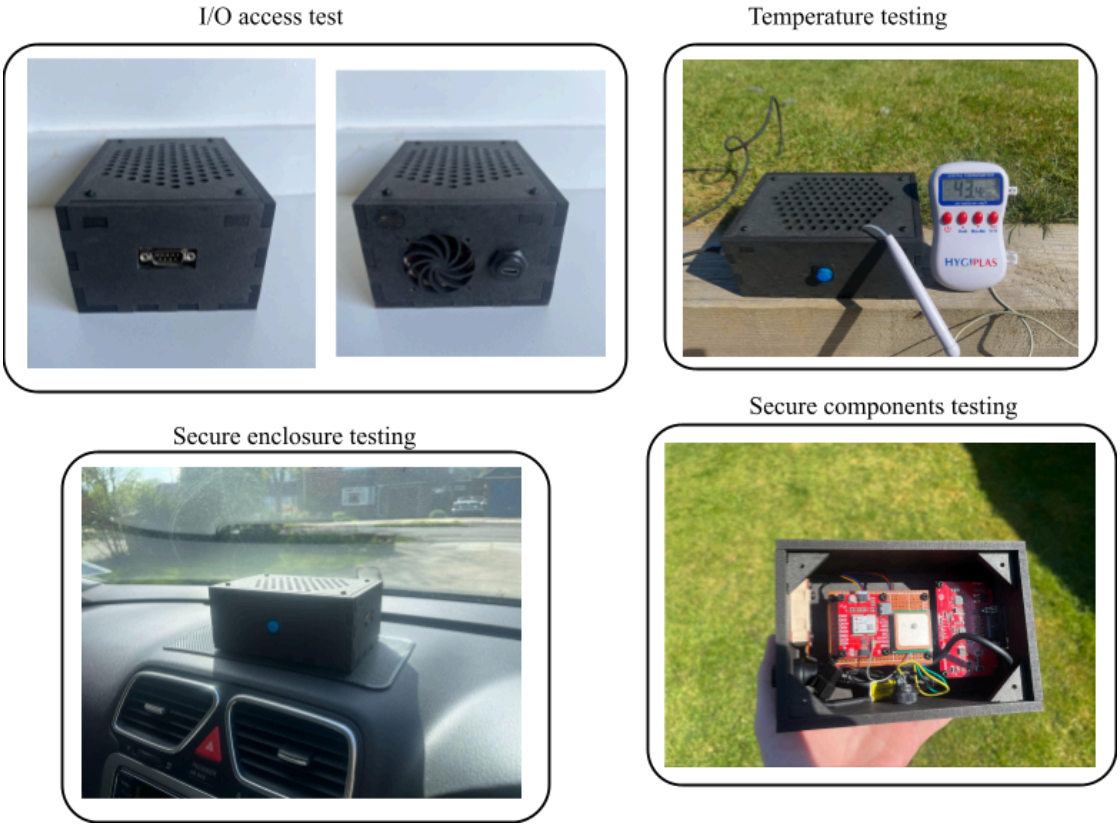


Figure 60: Sprint three tests evidence

10 References

Adafruit (2021) Adafruit QT Py ESP32-S2 WiFi Dev Board with STEMMA QT. Available at: https://www.adafruit.com/product/5325 (Accessed: 2 January 2025).

Amazon (2025) [Noctua NF-A4x10 5V]. Available at: https://www.amazon.co.uk/dp/B00NEMGCIA (Accessed: 18 March 2025).

Anon (2002) SAE Technical Standard. [Online]. 400 Commonwealth Drive, Warrendale, PA, United States: SAE International.

Anon (2012) Fuel Economy of Road Vehicles. [Online]. Paris: OECD Publishing.

Aptabase Team (2023) 'Why I chose Tauri instead of Electron', Aptabase. [online] Available at: https://aptabase.com/blog/why-chose-to-build-on-tauri-instead-electron (Accessed: 26 November 2024).

Automobile Association Developments Ltd. (2024) How to drive economically. Available at: https://www.theaa.com/driving-advice/fuels-environment/drive-economically (Accessed: 15 November 2024).

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001) Manifesto for Agile Software Development. [online] Available at: https://agilemanifesto.org/ [Accessed January 2025].

Cai, C. and Gao, Y. (2013) 'Modeling and assessment of combined GPS/GLONASS precise point positioning', GPS Solutions. [Online] 17(2), 223–236.

CSS Electronics (2024) OBD2 Explained - A Simple Intro. [online] Available at: https://www.csselectronics.com/pages/obd2-explained-intro (Accessed: 5 November 2024).

Department for Energy Security and Net Zero (2013) Weekly road fuel prices. [online] Last updated 22 October 2024. Available at: https://www.gov.uk/government/statistics/weekly-road-fuel-prices (Accessed: 24 October 2024).

Department for Transport (2013) Vehicle mileage and occupancy. [online] Last updated 28 August 2024. Available at: https://www.gov.uk/government/statistical-data-sets/nts09-vehicle-mileage-and-occupancy (Accessed: 24 October 2024).

DollaTek (n.d.) DollaTek 28dB High Gain GPS Active Antenna Ceramic Patch Internal Module Navigation Connector with Cable. Available at: https://www.amazon.co.uk/dp/B07MHGPT8L?ref_=ppx_hzsearch_conn_dt_b_fed_asin_title_1 (Accessed: 24 February 2025).

Drumond, C. (2025) What is scrum and how to get started. Available at: https://www.atlassian.com/agile/scrum (Accessed January 2022).

Eberhard, K. (2023) 'The impact of data visualization on decision making: A systematic review', Information & Management, 60(2), p.103674. Available at: https://ijrpr.com/uploads/V6ISSUE3/IJRPR41136.pdf (Accessed: 7 April 2025).

ECMA International (2017) ECMA-404: The JSON Data Interchange Standard. Geneva: ECMA International. Available at: https://www.json.org/json-en.html (Accessed: 2 January 2025).

eBay (2025) USB C Panel Mount Cable. Available at: https://www.ebay.co.uk/itm/185715395904 (Accessed: 18 March 2025).

Franklin, W. (2020) The Kalman Filter. Available at: https://thekalmanfilter.com/kalman-filter-explained-simply/ (Accessed: 22 November 2024).

Galitz, W. O. (1998) The essential guide to user interface design: an introduction to GUI design principles and techniques. Program: electronic library and information systems, 32(3), p.334–335.

Hackerspace Bamberg (2025) Electronics Box Generator. Available at: https://boxes.hackerspace-bamberg.de/ElectronicsBox (Accessed: 18 March 2025).

IBM (2021) Activity diagrams. Rational Software Architect 9.6.1. Available at: https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-activity (Accessed: 6 February 2025).

International Organization for Standardization (ISO) (2006) ISO 15031-5: Communication between vehicle and external equipment for emissions-related diagnostics - Part 5: Emissions-related diagnostic services. Available at: https://share.qclt.com/%E6%B1%BD%E8%BD%A6%E8%AF%8A%E6%96%AD%E5%8D%8F%E8%AE%AE2/ISO-15031-5[1].pdf (Accessed: 27 February 2025).

Kumar, R. and Jain, A. (2022) 'Monitoring and Remote Data Logging of Engine Operation via On Board Diagnostic Port', in 2022 Fifth International Conference on Computational Intelligence and Communication Technologies (CCICT). [Online]. IEEE. pp. 550–555.

Kwiatkowski, R. (2023) Real-Time Fuel Consumption Monitoring. Windmill Software. Available at: https://www.windmill.co.uk/fuel-monitoring-obd.html (Accessed: 2 January 2025).

Ladyada (2013) Micro SD Card Breakout Board Tutorial. [online] Adafruit. Available at: https://learn.adafruit.com/adafruit-micro-sd-breakout-board-card-tutorial (Accessed: 28 January 2025).

Lee, J. et al. (2013) 'Effect of the air-conditioning system on the fuel economy in a gasoline engine vehicle', Proceedings of the Institution of Mechanical Engineers. Part D, Journal of Automobile Engineering. [Online] 227(1), 66–77.

Ma, Y. and Wang, J. (2022) 'Personalized Driving Behaviors and Fuel Economy Over Realistic Commute Traffic: Modeling, Correlation, and Prediction', IEEE Transactions on Vehicular Technology. [Online] 71(7), 7084–7094.

Malekian, R. et al. (2017) 'Design and Implementation of a Wireless OBD II Fleet Management System', IEEE Sensors Journal. [Online] 17(4), 1154–1164.

MDN contributors (2024) Getting started with Svelte. [online] Mozilla Developer Network. Available at: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Svelte_getting_started (Accessed: 26 November 2024).

Microchip Technology Inc. (2010) MCP2551 High-Speed CAN Transceiver Data Sheet. DS21667F. Available at: https://cdn.sparkfun.com/assets/0/7/5/8/c/MCP2551.pdf (Accessed: 27 January 2025).

Mouser Electronics (2025) Electronic Components Distributor. [online] Available at: https://www.mouser.co.uk/ [Accessed January 2025].

Nascimento, P. P. L. L. do et al. (2018) 'An Integrated Dead Reckoning with Cooperative Positioning Solution to Assist GPS NLOS Using Vehicular Communications', Sensors (Basel, Switzerland). [Online] 18(9), 2895–.

OBD Solutions (2012) STN1110 multiprotocol OBD to UART interpreter: datasheet. Revision B. Available at: https://cdn.sparkfun.com/datasheets/Widgets/stn1110-ds.pd f (Accessed: 7 April 2025).

OpenStreetMap Foundation Operations Working Group (OSMF OWG) (n.d.) Tile Usage Policy. Available at: https://operations.osmfoundation.org/policies/tiles/ [Accessed 31 March 2025].

Przybylski, S. (2024, 9 January) The math behind Extended Kalman Filtering. Available at: https://medium.com/@sasha_przybylski/the-math-behind-extended-kalman-filtering-0df981a87453 (Accessed: 7 January 2025).

Rafael Timbó (2023) Front End Frameworks: What They Are, and Best Options. [online] Revelo. Available at: https://www.revelo.com/blog/front-end-frameworks (Accessed: 26 November 2024).

Rahman, S. M. A. et al. (2013) 'Impact of idling on fuel consumption and exhaust emissions and available idle-reduction technologies for diesel vehicles – A review', Energy Conversion and Management. [Online] 74, 171–182.

Rimpas, D. et al. (2020) 'OBD-II sensor diagnostics for monitoring vehicle operation and consumption', Energy Reports. [Online] 6(3), 55–63.

Scrum.org (2022) What is Scrum. [Online] Available at: https://scrumorg-website-prod.s3.amazonaws.com/drupal/inline-images/2021-01/scrumorg-scrum-framework-3000.png (Accessed January 2022).

Sim, A. X. A. and Sitohang, B. (2014) 'OBD-II standard car engine diagnostic software development', in 2014 International Conference on Data and Software Engineering (ICODSE). [Online]. IEEE. pp. 1–5.

SparkFun (no date) OBD-II UART Hookup Guide. Available at: https://learn.sparkfun.com/tutorials/obd-ii-uart-hookup-guide (Accessed: 30 October 2024).

Swadia, S. (2021) A Beginner's Guide to Creating a Map Using Leaflet.js. [online] SitePoint. Available at: https://www.sitepoint.com/leaflet-create-map-beginner-guide/ (Accessed: 26 November 2024).

Tang, L. et al. (2015) 'Lane-Level Road Information Mining from Vehicle GPS Trajectories Based on Naïve Bayesian Classification', ISPRS International Journal of Geo-Information. [Online] 4(4), 2660–2680.

The Pi Hut (2025) 16mm Illuminated Pushbutton - Blue Latching On/Off Switch. Available at: https://thepihut.com (Accessed: 6 March 2025).

Tudor, D. & Walter, G. A. (2006) 'Using an agile approach in a large, traditional organization', in AGILE 2006 (AGILE'06). [Online]. 2006 IEEE. p. 7 pp. – 373.

u-blox. (2018) NEO-M8U: u-blox M8 Untethered Dead Reckoning module including 3D inertial sensors - Data Sheet. UBX-15015679 - R06. Available at: https://cdn.sparkfun.com/assets/6/d/7/c/6/NEO-M8U_DataSheet__UBX-15015679_.pdf [Accessed 28 January 2025].

u-blox (2023) u-blox 8 / u-blox M8 Receiver description: Including protocol specification v15-20.30,22-23.01. UBX-13003221 - R28. Early production information. Available at: https://content.u-blox.com/sites/default/files/products/documents/u-blox8-M8_ReceiverDescrProtSpec_UBX-13003221.pdf?utm_content=UBX-13003221#page=366 [Accessed 30 January 2023].

u-blox (2024) Question about IMU in NEO-M8U module. u-blox Community Portal. Available at: https://portal.u-blox.com/s/question/0D5Oj00000iWMIwKAO/question-about-imu-in-neom8u-module [Accessed 24 February 2025].

Wikipedia. (2024) Fuel economy in automobiles. Available at: https://en.wikipedia.org/wiki/Fuel_economy_in_automobiles [Accessed 7 April 2025].

Wikipedia (n.d.) Use case diagram. [Online] Available at: https://en.wikipedia.org/wiki/Use_case_diagram [Accessed January 2022].

Yurday, E. (2024) Average MPG for Cars UK (2024). Updated 3 January. Available at: https://www.nimblefins.co.uk/cheap-car-insurance/average-mpg (Accessed: 5 November 2024).

Zhao, J. (2018) 'A Review of Wearable IMU (Inertial-Measurement-Unit)-based Pose Estimation and Drift Reduction Technologies', Journal of Physics: Conference Series. [online] 1087(4). Available at: https://www.proquest.com/docview/2572552414/abstract/15F9D13F9204492DPQ/1 (Accessed: 19 November 2024).

Ziakopoulos, A. (2024) 'Analysis of harsh braking and harsh acceleration occurrence via explainable imbalanced machine learning...', Accident Analysis and Prevention. [Online] 207, 107743.

Zhao, L., Ochieng, W. Y., Quddus, M., and Noland, R. B. (2003) 'An Extended Kalman Filter Algorithm for Integrating GPS and Low Cost Dead Reckoning System Data for Vehicle Performance and Emissions Monitoring', Journal of Navigation. 56(2), pp. 257–270. Available at: https://www.researchgate.net/publication/48352906_An_Extended_Kalman_Filter_Algorithm_for_Integrating_GPS_and_Low_Cost_Dead_Reckoning_System_Data_for_Vehicle_Performance_and_Emissions_Monitoring (Accessed: 7 January 2025).