ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΙΓΑΙΟΥ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
ΠΛΗΡΟΦΟΡΙΑΚΩΝ ΚΑΙ
ΕΠΙΚΟΙΝΩΝΙΑΚΩΝ
ΣΥΣΤΗΜΑΤΩΝ

Driver profiling and Gas station store classification simulation

Προσομοίωση Προφίλ Οδηγών και Ταξινόμησης Πρατηρίων Καυσίμων

του Χαλκιαδάκη Μιχαήλ
3212018241

Επιβλέπων καθηγητής:	Αλεξόπουλος Χαράλαμπος
Τριμελής επιτροπή:	Χ. Αλεξόπουλος, Κ. Μαλιάτσος, Γ. Χαραλαμπίδης

ΟΚΤΩΒΡΙΟΣ 2025

https://github.com/mchalkiadakis/driver-profiling-analysis-tool

Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων
Εργαστήριο Πληροφοριακών Συστημάτων
83200 Καρλόβασι, Σάμος

Ονομ/μο Φοιτητή:	Χαλκιαδάκης Μιχαήλ A/M, Email: icsd18241@icsd.aegean.gr, 3212018241
Επιβλέπων καθηγητής:	Αλεξόπουλος Χαράλαμπος alexop@aegean.gr
Τριμελής επιτροπή:	Χ. Αλεξόπουλος, Κ. Μαλιάτσος, Γ. Χαραλαμπίδης

Table of Figures

Table of Contents

ΤΥΠΟΣ ΕΡΓΑΣΙΑΣ	**ΠΕΡΙΓΡΑΦΗ - ΟΔΗΓΙΕΣ**	**ΠΑΡΑΔΕΙΓΜΑ**
Ερωματολόγιο	Φτιάχνω ερωτηματλογιο (google form) - διαδίδω - μαζεύω 100 απαντήσεις - οπτικοποιώ αποτελέσματα - γράφω κείμενο http://www.infosoc.gr/infosoc/el-GR/services/elibrary/reports_list/. ΠΡΟΣΟΧΗ στην πόλωση του δείγματος - να μην έχω μόνο απαντήσεις από φοιτητές !! Να υπάρχουν πλήρη ΔΗΜΟΓΡΑΦΙΚΑ στοιχεία: ηλικία, επίπεδο μόρφωσης, επάγγελμα, τόπος διαμονής κλπ για να μπορούν να γίνουν σχετικές ομαδοποιήσεις στα αποτελέσματα. Για τη ΔΙΑΔΟΣΗ του ερωτηματολογίου χρησιμοποιείστε (πλέον των κοινωνικών σας δικτύων) και : μεγάλα Blogs στην ελλάδα που σας επιτρέπουν να γράψετε, Linkedin και Facebook groups, Twitter με hashtag #έρευνα #μελέτη, το site / τα sites του φορέα που εμπλέκεται (στείλτε τους μία απλή επιστολή). Παραδίδεται word και αρχείο με απαντήσεις (google form / google sheet). Αν παραδοθεί και site (blogger / wordpress) αποτελεί επιπλέον στοιχείο βαθμολογίας.	bit.ly/1ILj2mC
Data Journalism	Βρίσκω δεδομένα - οπτικοποιώ (live chart) - γραφω "story" - κάνω blog publish. www.statistics.gr, www.geodata.gr, www.data.gov.gr, www.engagedata.eu ΣΥΓΚΡΙΤΙΚΑ ΜΕΤΑΞΥ ΧΩΡΩΝ. Παραδίδεται word και web site (blogger / wordpress)	http://t-government.blogspot.gr/2011/06/tax-offices-productivity-in-greece-for.html
Αναμορφώνω μία διαδικασία (τύπος)	Στοχεύω στη σχεδίαση μίας νέας διαδικασίας με την οποία κάποια υπηρεσία κράτους - πολίτη ή κράτους - επιχείρησης θα υλοποιείται σε 1 ηλεκτρονική "στάση" (one-stop) και σε 1 δευτερόλεπτο (άμεση εκτέλεση), με ελάχιστο δυνατό κόστος. Για να επιτευχθεί αυτό, πρέπει να προβλέψετε και να προδιαγράψετε τα απαραίτητα πληροφοριακά συστήματα και τα μηνύματα μεταξύ τους για την επίτευξη διαλειτουργικότητας. Παραδίδεται Word και Powerpoint.	https://eclass.icsd.aegean.gr/modules/document/file.php/ICSD250/%CE%A3%CE%B7%CE%BC%CE%B5%CE%B9%CF%8E%CF%83%CE%B5%CE%B9%CF%82%20-%20%CE%94%CE%B9%CE%B1%CE%BB%CE%AD%CE%BE%CE%B5%CE%B9%CF%82/eParavolo.pdf
Ανάπτυξη πρωτότυπης εφαρμογής	Σχεδίαση και υλοποίηση σε επίπεδο πρωτοτύπου μίας διαδικτυακής εφαρμογής που σχετίζεται με δημόσιες υπηρεσίες ή κοινωνικές λειτουργίες και ανάγκες. Παραδίδεται word, πρωτότυπη εφαρμογή και παρουσίαση – video (ppt ή screen capture/voice over)	https://pithos.okeanos.grnet.gr/public/6USETteMzky5rAz2Tkt7o7
Μελέτη Διαδικτυακών Τόπων	Ποιο URL έχουν, πόσες σελίδες έχουν, τι γλώσσες υποστηρίζουν (σε ποιο βάθος), τι υπηρεσίες παρέχουν (σε ποια επίπεδα 1-2-3-4-5), αν έχουν search, αν έχουν χάρτη του διαδικτυακού τόπου - ένδειξη σελίδας, αν έχουν πρόσβαση από κινητά (με κατάλληλο rendering), αν έχουν οργανόγραμμα, αν έχουν email επικοινωνίας με όλα τα στελέχη, αν έχουν χάρτη φυσικής πρόσβασης, αν έχουν ώρες λειτουργίας, αν έχουν ανοικτά δεδομένα (στατιστικά στοιχεία μελέτες, βιβλιοθήκη, κλπ), αν έχουν forum ηλεκτρονικού διαλόγου, αν έχουν facebook-twitter-youtube-linkedin (με πόσους followers στο καθένα), αν έχουν νέα, πότε έγινε η τελευταία ενημέρωση στα νέα, αν έχουν πρόγραμμα εκδηλώσεων, πότε έγινε η τελευταία ενημέρωση στα κοινωνικά δίκτυα. Παραδίδεται word και αρχείο επεξεργασίας αποτελεσμάτων (google sheet / excel). Αν παραδοθεί και site (blogger / wordpress) αποτελεί επιπλέον στοιχείο βαθμολογίας.	https://pithos.okeanos.grnet.gr/public/EG289lNlom7HxgTZFxgLE7

ΚΡΙΤΗΡΙΑ ΒΑΘΜΟΛΟΓΗΣΗΣ ΕΡΓΑΣΙΩΝ

Πρωτοτυπία: πόσο είδατε με καινοτομικό τρόπο το θέμα σας, πόσο είναι «δική» σας η εργασία, πόσο ξεπεράσατε τις οδηγίες

Πληρότητα: συνολικό μέγεθος εργασίας, πόσο καλύψατε τις παραπάνω οδηγίες, πόσο καλύψατε το θέμα σας, αριθμός και ποιότητα αναφορών

Ποιότητα: σωστή μεθοδολογία, σωστή γλώσσα και επιχειρήματα, απουσία ορθογραφικών και συντακτικών λαθών, καλό formatting

Περίληψη

Η παρούσα διπλωματική εργασία ασχολείται με τον σχεδιασμό και την ανάπτυξη ενός λογισμικού συστήματος που προσομοιώνει την οδηγική συμπεριφορά και αξιολογεί πρατήρια καυσίμων μέσα σε ένα πλήρως εικονικό περιβάλλον οχήματος. Το σύστημα συνδυάζει εξομοίωση OBD-II, GPS, τεχνικές μηχανικής μάθησης και διαδικτυακά APIs, με στόχο να αναπαραστήσει τη λειτουργία ενός «έξυπνου» συνδεδεμένου αυτοκινήτου. Ο βασικός σκοπός είναι να μελετηθεί ο τρόπος οδήγησης, να αξιολογηθούν οι στάσεις ανεφοδιασμού ως αξιόπιστες ή όχι και να εξαχθούν χρήσιμα συμπεράσματα χωρίς τη χρήση πραγματικού αυτοκινήτου ή εξοπλισμού.

Στο κέντρο του συστήματος βρίσκεται ο εξομοιωτής ELM327, υλοποιημένος σε Python, ο οποίος δημιουργεί δεδομένα όπως ταχύτητα, στροφές κινητήρα, ροή αέρα και σχέση κιβωτίου. Τα δεδομένα αυτά συνδυάζονται με συντεταγμένες GPS που παράγονται τεχνητά και αποστέλλονται μέσω Flask-SocketIO για επικοινωνία σε πραγματικό χρόνο ανάμεσα στο backend και τη διαδικτυακή εφαρμογή. Επιπλέον, το σύστημα αξιοποιεί εξωτερικές υπηρεσίες, όπως το vPIC API για πληροφορίες οχημάτων, το OpenWeatherMap για λήψη καιρικών δεδομένων και το Overpass API για εντοπισμό κοντινών πρατηρίων καυσίμων.

Η ανάλυση της οδηγικής συμπεριφοράς πραγματοποιείται με βάση κανόνες και γραφικές αναπαραστάσεις που καταγράφουν πώς αλλάζει ο τρόπος οδήγησης με τον χρόνο. Στη συνέχεια, εφαρμόζεται ο αλγόριθμος K-Means clustering για να ομαδοποιηθούν τα δεδομένα σε διαφορετικά προφίλ οδηγού. Το σύστημα εξετάζει επίσης την κατανάλωση καυσίμου για να υπολογίσει την αξιοπιστία κάθε πρατηρίου, αναγνωρίζοντας ποια προσφέρουν πιο σταθερά ή αποδοτικά ανεφοδιαστικά μοτίβα.

Η διαδικτυακή διεπαφή του συστήματος επιτρέπει στον χρήστη να βλέπει σε πραγματικό χρόνο την ταχύτητα, τις στροφές κινητήρα, την κατανάλωση, τη θέση του οχήματος στον χάρτη και το τρέχον προφίλ οδήγησης. Χάρη στην αρθρωτή του δομή, το σύστημα μπορεί εύκολα να επεκταθεί ώστε να συνδεθεί με πραγματικές συσκευές, όπως ένα Raspberry Pi με προσαρμογέα OBD-II, ή να χρησιμοποιηθεί σε μία πιο αποκεντρωμένη αρχιτεκτονική κατανεμημένων συστημάτων. Το έργο αυτό προσφέρει έναν απλό, ευέλικτο και οικονομικό τρόπο για τη μελέτη και αξιολόγηση της οδηγικής συμπεριφοράς μέσα από προσομοίωση, αποτελώντας μια χρήσιμη βάση για έρευνα, εκπαίδευση και περαιτέρω ανάπτυξη σε πραγματικές συνθήκες.

Summary

This thesis describes the design and development of a software-based system for driver profiling and fuel station evaluation, built entirely in a simulated vehicle environment. The system combines an OBD-II emulator, real-time GPS simulation, machine learning methods, and web-based APIs to mimic how a connected vehicle works. Its main goal is to study driving behavior, rate fuel stops as reliable or unreliable, and provide useful insights all without using any physical car hardware.

At its core, the system uses the Python-based ELM327 emulator, which simulates standard vehicle data such as speed, RPM, mass air flow (MAF), and gear position. This simulated data is paired with a GPS path generator and sent through Flask-SocketIO, which allows fast, real-time communication between the backend and the web interface. The platform also connects to external data sources like the NHTSA vPIC API for vehicle details, OpenWeatherMap for weather-based scoring, and the Overpass API to find fuel stations along planned or custom routes.

Driving behavior is analyzed using a mix of rule-based checks and graph-based models to track changes over time. The system collects data samples and runs them through K-Means clustering to group driving patterns into different states. Fuel station reliability is calculated using patterns in simulated fuel consumption, helping detect both good and potentially problematic service points.

The web interface provides an interactive dashboard showing live speed, RPM, fuel efficiency, map location, and the driver's current profile state. This setup makes it easy to see driving style, environmental conditions, and fuel stop quality in real time.

Because of its modular design, the system can be adapted for real hardware, such as connecting a Raspberry Pi to an actual OBD-II adapter, or extended for use in autonomous driving simulations and fleet management. This makes it a practical tool for research, teaching, and early-stage development, offering a low-cost and repeatable way to test ideas before moving to real-world trials.

1. Introduction

1.1 Background and Motivation

Over the past decade, the convergence of vehicle diagnostics, sensor technology, and software platforms has enabled the development of increasingly sophisticated driver behavior analysis systems. Traditionally, these systems relied on embedded telematics hardware or aftermarket solutions such as black boxes and On-Board Diagnostics (OBD-II) scanners to collect data about a vehicle's performance and the behavior of its driver. These technologies have proven valuable in contexts such as insurance risk assessment, fleet management, and eco-driving optimization.

However, the adoption of hardware-based systems faces limitations. They incur installation and maintenance costs, present data access challenges, and are constrained by hardware availability.To address these limitations, platforms based on software implementation have emerged that simulate or emulate vehicle behavior using standardized automotive protocols, such as ELM327 over OBD-II. When combined with network communication, geographic routing, and external APIs, these platforms offer a low-cost and highly customizable environment for experimentation and analysis. They also do not offer the possibility of a rough evaluation of the quality of the gas stations. There are many gas stations that offer lower quality fuel, mixed with detergents and corrosion inhibitors that reduce the efficiency of the combustion.Gas stations can also use several deceptive methods to make customers believe their tank is full while actually delivering less fuel than paid for. One common trick is **pump calibration tampering**, where the meter is adjusted so it displays more liters than it actually dispenses, making the gauge rise as expected but with less actual volume. Another method is **a**ir mixing or micro-foaming, where small amounts of air are injected into the fuel stream; the pump counts the air bubbles as fuel, and once the bubbles dissipate in the tank, you're left with less gasoline even though the fuel gauge has moved upward. Others may manipulate **fuel temperature** or use flow restrictor**s** to create the illusion of normal delivery speed and volume. In all these cases, the car's gauge reads only the rising liquid level, so the driver sees a "full" tank without realizing they received less fuel energy content than expected. [9]

This thesis was motivated by the need for a scalable, hardware-independent platform capable of performing basic driver profiling and fuel station evaluation. Such a system should emulate realistic vehicle behavior, gather and process telemetry data, and provide live feedback to users. Moreover, it should support integration with external data sources (e.g., weather APIs, vehicle databases, route planners) and offer analytical capabilities using both rule-based logic and unsupervised learning.

1.2 Problem Statement

There is a growing need for intelligent systems that assess not only driving performance but also external factors such as fuel stop reliability and environmental efficiency. Most current systems rely on static thresholds or isolated data sources.

They do not integrate multi-source information (e.g., GPS, MAF, weather) within a simulated loop.

In this context, the problem is defined as: how can we build an emulation-based application that performs driver behavior profiling, fuel consumption analysis, and fuel stop reliability evaluation in real time, without requiring access to a physical vehicle?

1.3 Thesis Objectives

The main goal of this thesis is to design and implement an emulation-driven platform that enables:

- **Driver profiling** based on OBD-II simulated data (speed, RPM, acceleration, fuel consumption).
- **Fuel station evaluation** based on efficiency after refueling and spatial proximity.
- **Real-time visualization and feedback** through a web-based dashboard.
- **Integration with external APIs** such as Overpass, NHTSA vPIC, and OpenWeatherMap.
- **Behavioral clustering and state transition mapping** using machine learning (KMeans and graph modeling).

Secondary goals include ensuring modularity for extensibility, creating reproducible test scenarios, and maintaining low computational and infrastructure costs.

1.4 System Overview

The proposed application consists of a backend engine written in Python, a frontend interface built using web technologies (HTML, JavaScript, Leaflet.js), and a communication layer using Flask-SocketIO for real-time data exchange. A simulated route (based on set GPS waypoints) is used with the help of OSRM API. Along this path, virtual driving behavior is emulated using an ELM327-compatible OBD-II emulator that responds to PID queries such as 010C (RPM), 010D (speed), and 015E (fuel rate). These data points are then processed and sent to the frontend for live display.

Driver samples (timestamped measurements of speed, RPM, and fuel consumption) are collected periodically and analyzed to determine the driver's behavioral state. The analysis pipeline includes both heuristic profiling rules and unsupervised clustering, producing qualitative profiles like "Aggressive", "Relaxed", or "Optimal".

Fuel stops are dynamically discovered using the Overpass API. When a simulated refueling event occurs, the system locks the state and monitors post-refueling efficiency. The fuel station is then classified as "Reliable" or "Unreliable" based on how far the vehicle travels relative to expected performance.

1.5 Structure of the Thesis

The structure of this thesis is as follows:

- **Section 2: Field Analysis** provides a survey of related work in driver profiling, telematics, and fuel consumption modeling. It also positions the current system within both international and national research contexts.
- **Section 3: Methodology** describes the design principles, software architecture, API integrations, and clustering models used in the system.
- **Section 4: Future Work** proposes directions for extending the application.
- **Section 8: References** lists the academic and technical sources cited throughout the thesis.

2. Field Analysis

2.1 Introduction to the Domain

The automotive industry is undergoing a digital transformation driven by the integration of software, sensor networks, and intelligent analytics. Among the key trends in this space are driver profiling, eco-driving optimization, predictive maintenance, and data-based insurance models. All of these depend on the ability to accurately capture and interpret vehicular telemetry data such as speed, acceleration, gear shifts, and fuel consumption in real time.

Traditionally, such data have been gathered using embedded vehicle hardware or aftermarket devices connected via the On-Board Diagnostics II (OBD-II) interface. However, these methods are often costly, require physical access to vehicles, and limit the reproducibility of testing conditions. In contrast, recent developments in OBD-II emulation and real-time GPS simulation allow researchers to simulate entire driving scenarios, enabling full-stack testing of intelligent vehicle systems without the constraints of physical deployment.

This thesis builds upon these innovations by introducing a software system that performs **driver profiling**, **fuel consumption analysis**, and **fuel station reliability**

evaluation using simulated vehicle data. It addresses a critical research and engineering need: how to develop, test, and iterate over complex vehicle analytics pipelines in a controlled, reproducible, and fully software-driven environment.

2.2 Industrial Applications and Real-World Systems

In the commercial world, vehicle telemetry is a cornerstone of modern fleet management, insurance risk modeling, and smart mobility services. Major companies have developed proprietary platforms that collect and analyze vehicle data for operational optimization and risk mitigation:

- **Fleet Telematics Platforms**: Tools such as *Verizon Connect*, *TomTom Webfleet*, and *Geotab* collect high-resolution data on vehicle usage, route adherence, driver safety, and fuel efficiency. These platforms are widely adopted by logistics and delivery companies to reduce costs and improve service levels.
- **Insurance Telematics**: Companies like *Progressive* (Snapshot), *State Farm* (Drive Safe & Save), and *Allianz* offer driver-based insurance pricing models. These rely on behavioral data such as harsh braking, rapid acceleration, cornering force, and nighttime driving patterns to calculate a risk-adjusted premium.
- **Connected Cars and OEM Integrations**: Modern vehicles often come with built-in telematics units (e.g., Tesla's remote diagnostics, BMW ConnectedDrive) that allow real-time communication with backend analytics systems.

While powerful, these systems are **closed-source**, often prohibitively expensive, and not designed for experimentation.

In contrast, the system developed in this thesis allows researchers to simulate complete trips, control environmental and operational parameters, and monitor all aspects of vehicle performance through emulated sensors.

2.3 National Context: Research and Deployment in Greece

In Greece, telematics research is primarily concentrated within academic institutions and specialized technology companies. Applications include intelligent transport systems, traffic prediction, and fleet optimization often supported by European funding programs.

Despite progress, real-world deployment of driver profiling and eco-driving analytics remains limited. Most commercial telematics deployments in Greece focus on vehicle tracking, basic maintenance alerts, and theft prevention. Very few systems attempt to perform **behavior-based scoring**, **machine-learning-based driving style detection**, or **service point evaluation** (such as fuel station reliability) as done in this thesis.

As a result, this project contributes not only to the international body of research but also fills a technological gap within the Greek context, offering a model for how

software emulation and intelligent analytics can be combined for research, teaching, and potential commercial exploration.

2.4 Emulation vs. Real-World Platforms

A key design choice in this project was to simulate vehicle data using software rather than collecting it through physical sensors or embedded automotive hardware. This decision significantly shaped the system's architecture and development process. Instead of working directly with a vehicle or embedded ECU, the system uses the open-source **ELM327 emulator**, which mimics OBD-II PID (Parameter ID) responses in a configurable and scriptable way. This allowed for the creation of a virtual vehicle environment where behavior such as RPM, speed, fuel rate, and gear state could be controlled and observed in real time.

The primary advantage of emulation lies in its accessibility. Physical automotive platforms require vehicles, sensors, diagnostic tools, and often specialized access to CAN bus systems. By contrast, emulation removes these barriers entirely. It eliminates hardware costs, enables rapid prototyping, and avoids the safety risks involved in testing real-world driving scenarios. For example, it is possible to simulate extreme conditions such as aggressive acceleration, sudden deceleration, or low-fuel warnings without putting anyone in danger or relying on track tests.

Another important benefit is repeatability. In real-world tests, environmental variables such as traffic, weather, or road conditions introduce noise and unpredictability. In an emulator, every test run can follow the exact same sequence of inputs, allowing for consistent comparison, debugging, and validation of algorithms. This is particularly valuable when developing systems for driver profiling, where minor variations in input data can affect the classification results.

Furthermore, emulation is highly adaptable. The ELM327 emulator can simulate vehicles of different types, including gasoline, diesel, and hybrid powertrains, by changing the simulated PID response logic. This makes it possible to analyze how different fuel types or engine behaviors affect fuel consumption, emissions, or driver behavior, without having access to those actual vehicle models. When paired with external APIs such as NHTSA’s vPIC API for real vehicle specifications, or OpenStreetMap and Overpass for spatial data the system gains even more realism and context-awareness, despite not being physically installed in a car.

However, emulation does come with limitations. Most notably, it lacks sensor noise, hardware failure modes, and generally physical constraints. In real cars, sensor data is influenced by mechanical wear, latency, electrical noise, and environmental factors like temperature or road friction. An emulator can only produce idealized data unless these effects are artificially modeled. Additionally, factors such as tire condition, brake performance, steering accuracy, or driver fatigue are outside the scope of our purely digital simulation.

Another drawback is that validation in the real world is still necessary. While emulation is an excellent tool for prototyping and testing logic, it cannot replace real-

world testing when it comes to deployment or certification. A system trained or tested purely on synthetic data might not perform reliably when exposed to real-life imperfections, making it crucial to eventually verify performance with real sensor inputs.

Despite these challenges, emulation remains a powerful approach for academic research, system design, and early-stage development. It accelerates the development lifecycle, enables experimentation without risk, and provides a controlled environment where hypotheses can be tested rapidly. Especially for student projects, R&D environments, or AI training pipelines, vehicle emulation offers a low-cost, high-flexibility alternative to physical prototypi**ng.**

2.5 Related Scientific Work

Driver Profiling Systems

One of the most directly relevant works is **SenseFleet** by Castignani et al. (2015), which introduced a smartphone-based driver behavior analysis system. SenseFleet uses fuzzy logic to classify events such as sharp turns, harsh acceleration, and sudden stops. These classifications are then weighted based on trip context (e.g., time of day, weather) to generate a final driving score. The platform also includes a calibration phase, adjusting its scoring model to the typical behavior of the driver.

This thesis builds upon similar concepts but replaces smartphone sensors with **emulated OBD-II signals**, offering a richer and more realistic representation of vehicle dynamics. It also introduces a **graph-based profiling method**, inspired by research on **driver state transitions**, where clusters of driving behavior (e.g., speeding, stable cruising, aggressive acceleration) are modeled as nodes in a transition graph. This allows the detection of repeating patterns, abrupt transitions, and long-term driver tendencies.

Fuel Consumption Modeling

Works like **Navneeth et al. (2020)** and **Garg & Singla (2015)** use OBD-II signals such as RPM, vehicle speed, and Mass Air Flow (MAF) to estimate real-time fuel consumption. These studies validate that even with a limited set of PIDs, it is possible to estimate consumption with sufficient accuracy to support eco-driving feedback systems. The current thesis adopts these models and expands upon them by applying them **during a simulated trip**, validating them against expected distance-to-fuel ratios.

Graph-Based Profiling

The thesis also employs methods from graph theory, such as the construction of **driver behavior graphs** using state clustering and edge transition weights. This technique has been shown in other works to provide useful abstractions for long-term behavioral tracking and anomaly detection. In this system, the graphs are generated

during the trip and updated live, offering real-time profiling rather than post-trip analytics.

2.6 Technologies and Tools Used

To build this vehicle simulation and analysis platform, a carefully selected set of tools, libraries, and APIs was used. Each component played a specific role in enabling real-time data flow, emulation, visualization, and driver profiling, all within a lightweight and modular architecture.

- **ELM327-emulator**
  This open-source emulator replicates the behavior of an ELM327 OBD-II interface, which is commonly used in automotive diagnostics. It allows the system to simulate realistic PID responses (e.g., speed, RPM, fuel rate) over a TCP socket, without requiring a physical vehicle or OBD hardware. This tool serves as the foundation for vehicle data generation and is highly configurable, enabling the simulation of various engine types and driving conditions.

- **Flask-SocketIO**
  Flask is a lightweight Python web framework, and when paired with SocketIO, it enables **real-time, bidirectional communication** between the server (backend) and the web client (frontend). This is essential for delivering live GPS updates, fuel metrics, driver feedback, and map interactions in real time. SocketIO was chosen over traditional REST endpoints to ensure low-latency, event-driven updates between components.

- **Leaflet.js**
  Leaflet is a powerful JavaScript library for interactive maps. It provides the frontend interface with a dynamic map view, showing the vehicle's real-time position, fuel station markers, and any refueling events. Its lightweight nature and OpenStreetMap integration make it ideal for custom simulations and real-time visualization in a browser without the need for commercial map services.

- **Overpass API**
  The Overpass API is a query interface for OpenStreetMap data. In this project, it is used to locate nearby **fuel stations** (i.e., nodes tagged with `"amenity"="fuel"`) based on the geographic bounds of the simulated route. These stations are displayed on the map and used as potential refueling locations, integrating the simulation with real-world spatial data.

- **NHTSA vPIC API**
  Provided by the U.S. National Highway Traffic Safety Administration (NHTSA), this API decodes VINs (Vehicle Identification Numbers) to return real-world vehicle information such as **fuel type**, **engine details**, and **manufacturer data**. It allows the system to automatically adjust simulation parameters (e.g., stoichiometric ratio, fuel density) based on the actual vehicle entered by the user, improving realism and user customization.

- **KMeans (from scikit-learn)**
  KMeans is a machine learning clustering algorithm used here to analyze and classify **driver behavior patterns** based on speed and acceleration data. The platform collects time-series samples during the simulation and applies clustering to identify different driving states or habits (e.g., aggressive, relaxed). These clusters form the basis for behavior profiling and graphical state modeling.

- **NetworkX**
  This Python library is used to construct **state-transition graphs** from the KMeans clustering results. Each node in the graph represents a behavioral state, and edges indicate transitions between them during the simulation. This graphical representation helps visualize how a driver moves between behaviors over time, adding depth to the driver profiling analysis.

2.7 Contribution and Research Gap

What makes this thesis different is the fact that it combines several elements into one system: driver profiling, route simulation, fuel consumption modeling, and even checking how reliable fuel stations are. Most existing systems usually focus on just one or two of these areas, and they often depend on expensive hardware or closed platforms that are not easy to access. Here, everything is done in software, making the whole setup easier to reproduce and experiment with.

The system shows how a software-only approach can still handle intelligent vehicle data analytics effectively. By using an OBD-II emulator, GPS simulation, and online APIs, it builds a connected-vehicle environment without needing real cars or telematics devices. At the same time, it mixes synthetic data with real-world information, since APIs for vehicle specifications, weather, and fuel stations are directly built into the simulation.

On the research side, the project goes beyond simple rule-based checks for driver behavior. It also uses graph analysis and clustering methods to spot driving patterns in a way that gives more depth and accuracy. Finally, the thesis introduces a new idea: evaluating fuel stations based on how fuel efficiency changes after a refuel. Instead of just looking at price or volume, this approach focuses on performance, giving a fresh perspective.

In recent years, several academic and commercial solutions have been proposed for monitoring driver behavior and generating risk or efficiency profiles. These solutions form the current state of the art and provide a useful background against which the contributions of this thesis can be compared.

One well-known academic approach is **SenseFleet**, a platform that leverages mobile phone sensors such as accelerometer and GPS to capture and analyze driving events. Using fuzzy logic, the system detects risky behaviors including harsh braking or

speeding, and adapts its thresholds to different drivers and road conditions. Its applications are mainly in fleet management, eco-driving, and usage-based insurance, while also supporting CO_2 reduction initiatives [11].

Another relevant line of work is based on mobile applications connected to **OBD-II adapters**. For example, Eren et al. [12] proposed a smartphone app that combines GPS traces with vehicle parameters such as RPM, throttle, and speed. Machine learning techniques are then applied to classify drivers into behavioral categories and detect unsafe patterns. Such solutions show how low-cost hardware (smartphones plus OBD-II dongles) can provide both diagnostic insights and driver profiling.

In the commercial sector, **Usage-Based Insurance (UBI)** products are now common. Programs such as Progressive's Snapshot or Allstate's Drivewise rely on telematics devices or smartphone apps to monitor habits like harsh acceleration, speeding, or nighttime driving. These behaviors are aggregated into risk scores that directly influence the driver's insurance premium. Studies have shown that telematics data provides strong predictive value for insurance risk selection and pricing [13].

Car manufacturers are also heavily investing in telematics. **OEMs** such as Tesla, BMW, and Toyota integrate connected vehicle platforms that collect real-time data for diagnostics, predictive maintenance, and advanced driver-assistance system (ADAS) feedback. These systems are tightly integrated into the vehicle's architecture and serve as a foundation for connected mobility ecosystems [14].

While these systems are powerful and already demonstrate strong capabilities in telematics and driver analytics, the present thesis deliberately takes a different approach. Instead of relying on physical vehicles, expensive data loggers, or proprietary platforms, the work presented here develops a **fully software-based emulation environment**. This environment is capable of reproducing OBD-II signals with high fidelity, allowing researchers to test driver profiling techniques without the need for an actual car. Such an approach brings several advantages. It significantly reduces costs, since no dedicated hardware or on-road data collection campaigns are required. It also improves reproducibility, as every experiment can be repeated under identical conditions. Furthermore, it enhances safety, because potentially risky driving behaviors (such as harsh braking or sudden accelerations) can be studied in a purely virtual setting, without endangering drivers or other road users.

A second key contribution is the introduction of an **innovative link between driver profiling and fuel station evaluation**. While most existing platforms focus exclusively on detecting driving styles or calculating risk scores, the system described here also considers how refueling patterns and fuel quality affect the driver's behavior and vehicle performance. By comparing the expected and actual driving range after each refueling event, the system provides insights into whether a station consistently delivers the promised fuel volume or quality. This dimension has been largely overlooked in existing literature and commercial applications, yet it directly impacts both economic efficiency for the driver and environmental sustainability.

Finally, unlike SenseFleet or many insurance-driven telematics systems that analyze short-term driving sessions or focus only on individual trips, the architecture developed in this thesis is **extensible toward long-term behavioral monitoring**. At the current stage, the system focuses on single-session profiling, but its modular design allows for the integration of a database to persist driver data across multiple simulations. This would make it possible, in future work, to identify gradual changes in a driver's style over weeks or months. For example, with session-to-session logging, the system could detect whether a driver becomes progressively more aggressive, whether fuel efficiency improves with experience, or whether external factors such as traffic congestion or time of day influence long-term habits. This forward-looking perspective positions the system not only to classify driving behaviors at a single point in time but also to eventually track their evolution, which would be essential for applications in personalized feedback, adaptive training, or even future autonomous driving support.

In this way, the system presented here not only builds upon existing ideas but also moves beyond them by focusing on reproducibility, integration of fuel quality evaluation, and adaptability to both academic and applied contexts.

In today's market there are several well known telematics and driver profiling platforms that show the state of the art but also highlight some clear limits for research. **Verizon Connect** for example is a big fleet management solution that offers real time tracking, driver scorecards, fuel usage reports and route optimization. It mainly works with proprietary OBD gateways and OEM integrations, and while it is enterprise grade in scale and dashboards, the platform is closed and very hardware dependent, which makes it not really suitable for experiments. **Geotab** is another fleet oriented system that combines high frequency CAN/OBD data with driver coaching and fuel analytics. It has a marketplace of APIs and add-ons that makes it flexible, but again the devices are licensed and there is vendor lock-in, so quick prototyping is difficult. **Samsara** focuses more on connected operations, with GPS/ELD compliance, safety events and even distraction detection with cameras, and has a strong user experience, but the event scoring is basically a black box and the reliance on hardware fleets makes repeatable testing hard. **TomTom Webfleet** is also popular especially in Europe, with eco-driving scores, fuel/CO_2 reporting and tachograph features, but its analytics are not open and access to data is paywalled behind devices. In the insurance domain **Octo Telematics** provides risk scoring, crash detection and actuarial tools based on OBD dongles, smartphones or OEM data, but the models are closed and only accessible through insurer programs, so there is little room for exploration. Finally, **Cambridge Mobile Telematics (CMT)** takes a more smartphone-based approach with the DriveWell SDK, using phone sensors for driving scores and distraction detection. It is easy to deploy, sometimes even without extra hardware, but phone sensors only approximate vehicle dynamics and the backend models remain opaque.

3. Methodology

In order to build a system capable of profiling driving behavior and evaluating the reliability of fuel stations without relying on real-world vehicle data, I decided to create a complete emulation-based framework. This system simulates a moving vehicle by combining a GPS route, emulated OBD-II data, and real-time communication between the backend and the user interface. All vehicle parameters, such as speed, RPM, gear, and fuel consumption, are generated dynamically and adjusted over time to imitate realistic driving conditions.

To make the simulation feel more realistic and tailored, I implemented a feature that allows the user to enter a **Vehicle Identification Number (VIN)** at the beginning of the simulation. The VIN is a globally unique code assigned to every vehicle and can be used to identify its manufacturer, model, fuel type, and other characteristics.

When the user submits a VIN through the frontend, the backend sends a request to the **vPIC (Vehicle Product Information Catalog) API** provided by the **National Highway Traffic Safety Administration (NHTSA)**. This API returns a structured JSON response that includes dozens of fields describing the vehicle.

From this response, I extract key fields that are useful for simulation purposes. The most important among these is the **fuel type**. Based on the reported value, the system sets internal parameters that affect fuel consumption modeling. For example:

- If the **FuelTypePrimary** field indicates `"Gasoline"`, the following values are used:
    - **Stoichiometric Air-Fuel Ratio (AFR):** `14.7`
    - **Fuel density:** `720 g/L`
- If the field indicates `"Diesel"`, then:
    - **AFR:** `14.5`
    - **Fuel density:** `840 g/L`

These parameters are important because they directly affect the **fuel consumption calculation** based on MAF (Mass Air Flow). The formula used to estimate fuel

consumption depends on both the AFR and the fuel density, so using vehicle-specific values ensures that the simulation adapts to different vehicle types.

If the API fails to return a valid fuel type (for example, if the VIN is not recognized or the field is missing), the system/simulation falls back to a **default configuration**:

Once the backend receives the vehicle details, it starts simulating the drive along a predefined route in this case, a trip across Crete (BOAK) using GPS coordinates generated with the OSRM routing API. As the simulated vehicle travels along the route, the system continuously generates OBD-II data and sends it to the frontend, where it is displayed in real time.

To simulate the OBD-II interface, I used the open-source **ELM327-emulator**. This tool pretends to be a real ELM327 device (An ELM327 device is a small microcontroller-based adapter that acts as a bridge between a vehicle's OBD-II diagnostic port and external devices such as laptops, smartphones, or diagnostic scanners. It interprets the standardized OBD-II commands defined by SAE J1979 and translates them into a simple serial communication protocol, usually accessible over USB, Bluetooth, or Wi-Fi. This allows users to request information such as RPM, vehicle speed, fuel trims, or error codes directly from the car's ECU)and responds to diagnostic commands over TCP. I created a Python class that connects to this emulator and sends standard commands like "010C" (for RPM) and "010D" (for speed). The emulator replies in hexadecimal, and I then convert those values into readable numbers. For example, RPM is calculated based on how fast the simulated car is moving and is adjusted slightly using random factors to make it feel more natural.

Fuel data was another key element. To calculate fuel consumption, I used two approaches. First, I relied on the MAF (Mass Air Flow) values returned by the emulator through PID 0110. The Mass Air Flow (MAF) sensor measures the amount of air entering the engine in grams per second (g/s). Since modern gasoline engines operate at a near-constant stoichiometric air–fuel ratio (AFR) of approximately 14.7:1 by mass, the MAF value can be used to infer the fuel mass flow rate. Dividing the MAF reading by the AFR gives the fuel flow in grams per second. To convert this to liters per hour (L/h), the result is divided by the density of gasoline (typically around 710 g/L) and then multiplied by 3600 to scale from seconds to hours. The general formula is:

Fuel Flow (L/h)=MAF (g/s)AFR (g air/g fuel)×1Fuel Density (g/L)×3600\text{Fuel Flow (L/h)} = \frac{\text{MAF (g/s)}}{\text{AFR (g air/g fuel)}} \times \frac{1}{\text{Fuel Density (g/L)}} \times 3600Fuel Flow (L/h)=AFR (g air/g fuel)MAF (g/s)×Fuel Density (g/L)1×3600

This method is widely used in on-board diagnostics and telematics systems where direct fuel flow measurements are not available, as it allows continuous, real-time estimation of fuel consumption using only standard OBD-II data from PID 0110.

Based on that [15], I estimated the vehicle's fuel consumption in liters per hour. Secondly, I also tracked fuel level as a percentage of the tank. At the beginning of the simulation, the tank starts at 100%, and as the vehicle moves, it burns fuel depending on speed, distance, and consumption rate. If the driver (or the user of the interface) triggers a refueling event near a fuel station, the system locks in and starts monitoring the efficiency of that refuel.

One of the interesting parts of this methodology was that I didn't have to depend on physical hardware. All of the vehicle responses are simulated by the emulator, and I had complete control over what values it returns and when. This meant I could easily test different driving scenarios fast driving, eco-driving, sudden stops, or even refueling halfway through a trip. It also helped me ensure that the system behaves consistently during testing.

To bring everything together, I used **Flask** as the backend web server and **Flask-SocketIO** to handle real-time communication. Every time the backend gets a new set of sensor values from the emulator, it sends them to the frontend using a WebSocket event. The frontend then updates the user interface with the latest values for example, showing the current speed, gear, fuel rate, and driver profile state on the screen. This interaction happens around 10 times per second, which is fast enough to feel live and smooth.

Overall, the methodology combines simulation, live data processing, and user interaction in a way that resembles real-time telematics systems. The advantage of doing everything in software is that it gives full flexibility to test complex behaviors without the limitations of physical devices.

Continuing from the previous setup, once the vehicle simulation is running and telemetry data is being generated, the next major goal of the system is to determine what kind of driving behavior is being exhibited during the simulated trip. To do this, I implemented a mechanism for driver profiling based on both real-time analysis and machine learning techniques, specifically clustering.

At each time step in the simulation loop which runs roughly every 100 milliseconds the backend collects a snapshot of the current driving data. This includes key metrics such as vehicle speed in km/h, RPM, fuel consumption in liters per 100 km, and a timestamp. These values are stored in memory in a list of "samples" that represent how the driver has been behaving over a short period, usually a few seconds.

What I found early on was that having just raw numbers doesn't immediately help classify driving style. For example, a speed of 80 km/h could be perfectly normal on a highway but aggressive in an urban area. So instead of using the raw values directly, I started focusing on **changes** in these values specifically acceleration and variability. For each pair of consecutive samples, I calculate the difference in speed and divide it by the time difference to estimate acceleration. This gives a more accurate sense of how suddenly the vehicle is changing speed, which is one of the main indicators of aggressive driving.

Once enough samples are collected usually over a window of 5 to 10 seconds the system starts performing profiling. The first method I implemented was a **rule-based** classification system. It works by checking certain thresholds. For instance, if the maximum observed acceleration in the current window is above 3.0 m/s^2 and fuel consumption is unusually high (say, more than 12 L/100 km), the system labels that driving period as "Aggressive." On the other hand, if acceleration is very smooth (below 1.0 m/s^2) and the fuel consumption is low (under 8 L/100 km), it is marked as "Relaxed." Anything in between is considered "Optimal." This logic is very basic but useful because it gives immediate feedback without needing complex calculations.

However, I wanted the system to be more flexible and capable of learning patterns automatically. To achieve that, I implemented a second profiling method based on **unsupervised machine learning** using the KMeans clustering algorithm from the scikit-learn library. The idea behind this approach is to group similar driving patterns into clusters without manually labeling them. For this to work, each driving sample is transformed into a two-dimensional feature vector: one axis is speed, and the other is acceleration. These vectors are collected and fed into KMeans, which then tries to organize them into groups based on similarity.

I experimented with different numbers of clusters and eventually settled on three, since they tend to roughly correspond to aggressive, relaxed, and normal driving states. After KMeans assigns each sample to a cluster, I then track how the driver moves between clusters over time. For example, if a driver goes from relaxed to aggressive and back to relaxed, the system records this as a state transition. To model these transitions, I used a simple **directed graph** where each node is a cluster and edges represent the number of transitions between them. This allowed me to create a basic "behavior graph" of the driver during a trip, showing how stable or erratic their style was.

This clustering-based method is especially powerful because it doesn't require hardcoded rules. It learns from the data itself. It can also adapt depending on how the emulator is configured. For instance, if I simulate a very fuel-inefficient vehicle, the clustering still works it just shifts the baseline of what counts as high consumption.

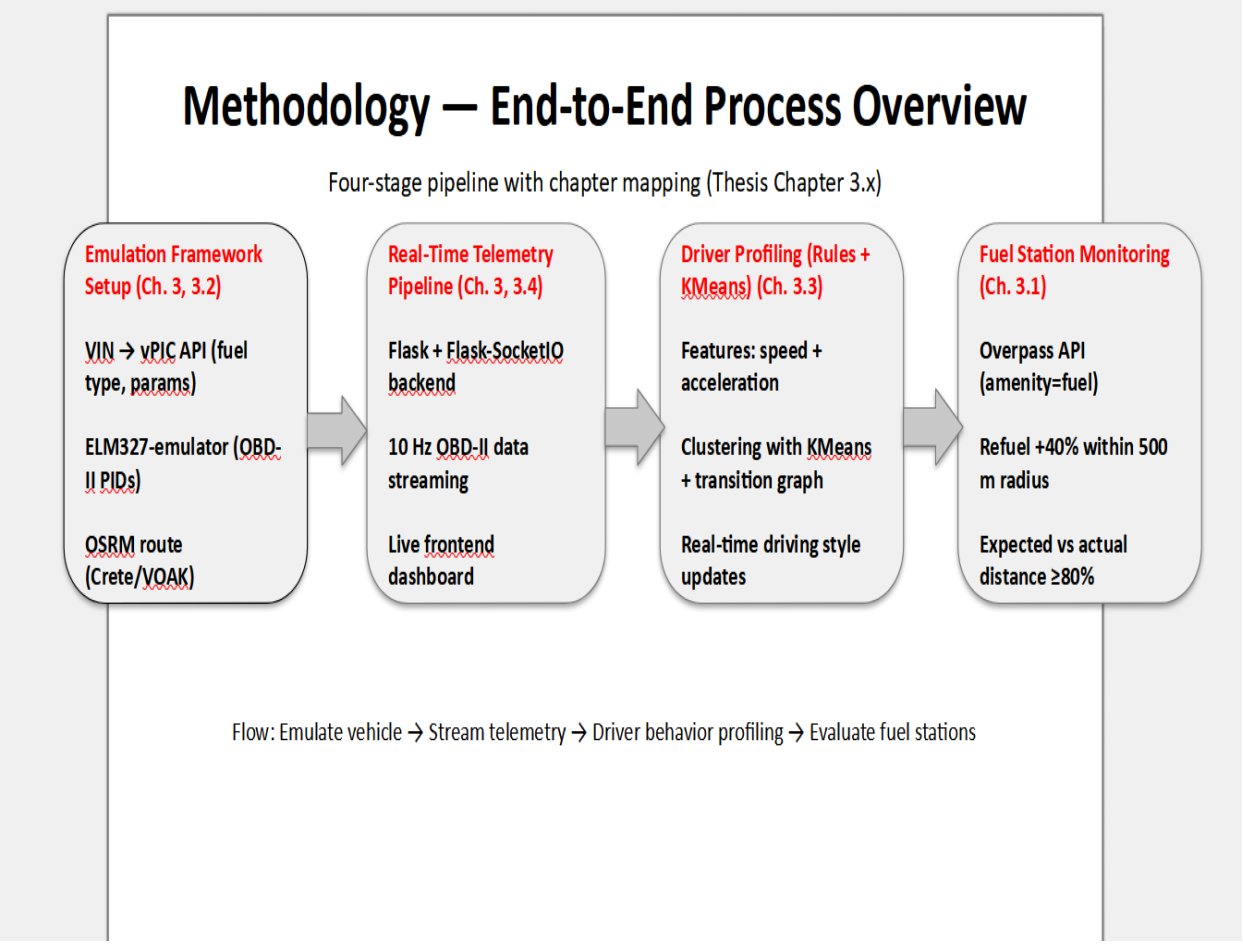
Each time a profiling update is performed (usually every couple of seconds), the current graph is analyzed, and a summary is generated. This summary is then sent to the frontend as part of a WebSocket message, which updates the "Driver Profile" field in the interface. This gives the user live feedback about their simulated driving behavior.

In addition to this, I made sure that the clustering system resets itself periodically so that each segment of the trip is treated independently. That way, if the user starts driving aggressively in one part of the route and then switches to a calmer style later, the system captures that transition and doesn't average it out across the whole journey.

One of the most challenging parts of implementing this system was tuning the parameters. Small changes in the sample window size, clustering features, or threshold values could cause the profiling to feel either too sensitive or too unresponsive. Through trial and error.I adjusted the system to be responsive while still avoiding noisy fluctuations.For example I decided to use three clusters for the driver profiling instead of a larger number. The main reason is that three groups are easy to understand and match the kinds of driving styles I wanted to capture: relaxed, normal, and aggressive. If I tried to use more clusters, the results often became messy, with small groups that didn't really mean anything useful for the analysis. Since the system runs in real time, I also needed the classification to be quick and stable so the driver profile wouldn't jump around too much. With three clusters the results are both interpretable and consistent, while still showing clear differences in driving style.

To help with debugging and tuning, I also added a logging system that prints profiling information to the backend console. This includes current cluster centers, number of samples in each cluster, and the current behavior label. These logs were extremely useful during development and allowed me to visualize how the driver model evolved over time.

In summary, the profiling part of the methodology combines both a simple rules-based classifier for immediate response and a more sophisticated clustering-based method for pattern analysis. Together, these approaches provide a strong foundation for understanding and reacting to driver behavior in a simulated environment.



The diagram above illustrates the overall methodology adopted in this thesis. It is structured into four main stages, each corresponding to a subsection of Chapter 3. The process begins with the *Emulation Framework Setup* (Section 3.2), where the system initializes vehicle parameters, VIN-based configuration, and the OBD-II emulator. It then moves to the *Real-Time Telemetry Pipeline* (Section 3.4), which ensures continuous data streaming between the backend and the user interface. Next, *Driver Profiling* (Section 3.3) applies both rule-based thresholds and KMeans clustering to classify and analyze driving behavior in real time. Finally, *Fuel Station Monitoring* (Section 3.1) evaluates the reliability of refueling points based on expected versus actual driving range. Together, these four stages form a complete emulation-based environment for driver behavior analysis and fuel station evaluation.

4.Development and validation of services

4.1 Fuel Station Monitoring and Evaluation

Based on the research gap identified in Section 2.7, several user requirements were defined for the system. Existing commercial solutions such as Verizon Connect, Geotab or Octo Telematics are powerful but they all share common limitations: they

rely heavily on proprietary hardware, their algorithms are black-box and cannot be easily inspected, and they do not provide a low-cost reproducible environment for experimenting with new ideas. To address these gaps, the following requirements were established:

- **Hardware independence**
  Derived from: Market solutions (e.g., Verizon, Geotab, Octo) rely on proprietary OBD gateways or dongles. This creates cost, vendor lock-in, and barriers for experimentation.
  Requirement: The system must run in a software-only environment, avoiding reliance on external dongles or closed OEM platforms.
- **Reproducibility and control**
  Derived from: Commercial tools are tied to real vehicles and fleet operations, where conditions (traffic, weather, driver mood) cannot be repeated exactly.
  Requirement: The system should ensure controlled and repeatable test runs by using an emulator and fixed scenarios.
- **Transparent driver profiling**
  Derived from: Market systems (e.g., Samsara, Octo, CMT) provide driver scores but use opaque scoring models.
  Requirement: The system should implement profiling logic that is transparent and modifiable, combining rule-based thresholds with open machine learning methods (e.g., KMeans, graphs).
- **Fuel station evaluation**
  Derived from: No reviewed solution explicitly links refueling events to fuel efficiency or service reliability. This represents an unaddressed gap.
  Requirement: The system should classify fuel stations as “reliable” or “unreliable” by comparing expected vs. actual post-refuel range.
- **Integration with real-world data sources**
  Derived from: Commercial solutions integrate live data but through proprietary APIs. Academic prototypes often ignore real context.
  Requirement: The system should integrate open APIs (vPIC, Overpass, OpenWeatherMap, OSRM) to ensure realism while remaining open and reproducible.
- **Real-time visualization and interactivity**
  Derived from: Market dashboards are powerful but inaccessible for research. Academic works often lack user-facing interfaces.
  Requirement: The system should provide a lightweight web dashboard with live telemetry, driver state, and refueling interaction, so users can experience the simulation in real time.

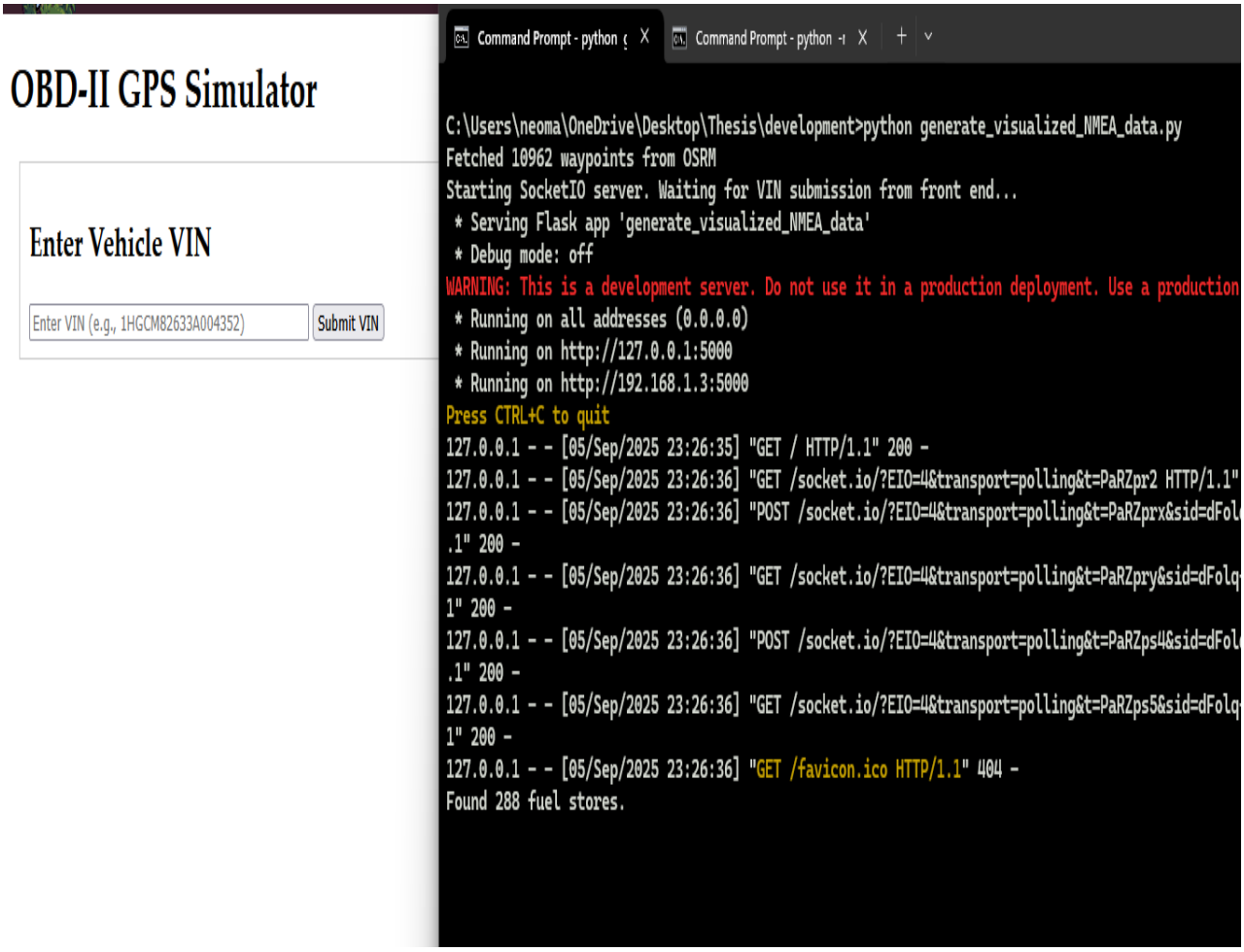
These requirements guided the design and implementation of the artifact and ensured that the final system directly addresses the gaps identified in the literature and market analysis.

4.2 Fuel Station Monitoring and Evaluation

Designing a system that can evaluate the quality or "reliability" of a fuel station based solely on simulated vehicle data was both an interesting challenge and a creative opportunity. Since I didn't have access to real-world telemetry from fuel sensors or onboard diagnostics, I had to find a way to approximate this type of assessment using virtual indicators mainly, the **expected distance the vehicle should travel after a refill** versus the actual distance it covered. The concept was simple: a good fuel refill should allow the vehicle to cover a certain number of kilometers based on known consumption rates. If it didn't, something was off perhaps the fuel wasn't effective, or the tank wasn't properly filled.

The first step was to **collect fuel station data** from the surrounding area along the predefined GPS route. To do this, I used the **Overpass API**, which allows querying OpenStreetMap data. I constructed a bounding box using the latitude and longitude values of the route waypoints and queried for all nodes tagged with `amenity="fuel"`. The response returned a list of stations with their GPS coordinates and optional metadata like station name. This meant that no matter where the simulation ran, the system could dynamically discover real fuel stations from open data sources.

Each of these stations was stored as a dictionary object inside a list. In addition to latitude, longitude, and name, I also added a placeholder field called `reliable`, which initially had the value `None`. This field would later be updated with either `True` or `False` depending on whether the station was deemed reliable after evaluation.



It is worth noting that in order to conserve resources the fuel stores and the waypoints are searched for and loaded (queried) in the VIN submission page, before the user has started the main, more demanding part of the simulation.

Once the stations were loaded, the next problem was figuring out **when and where the user could refuel**. On the frontend, I provided a “Refuel (+40%)” button.I chose to implement the refuel action as a fixed +40% increase to the fuel tank instead of a full refill. The main reason was to keep the simulation more realistic and flexible. In real life, drivers do not always refuel to 100%; they often add a partial amount depending on cost, urgency, or availability. By using 40%, the system could test how efficiency changes after a noticeable but not complete refuel, while still leaving room for multiple refueling events during a single trip. This value was large enough to be clearly detected by the system but small enough to avoid ending the simulation prematurely with a permanently full tank.

When clicked, it emitted a `refuel_request` event to the backend, along with the vehicle's current latitude, longitude, and the total distance it had traveled so far.

When the backend received this event, it needed to validate whether the vehicle was close enough to any known fuel station. I did this using the **haversine formula**, which

calculates the distance between two points on Earth based on their lat/lon coordinates. If the distance from the vehicle's current location to a station was less than 500 meters, I considered that the vehicle was eligible to refuel at that location.

If a valid refuel station was found, the backend created a **monitoring block** using a dictionary called `monitoring_data`. This contained key fields such as:

- The name and coordinates of the fuel station
- How much fuel was in the tank before refueling
- The amount of fuel added (40% of full capacity)
- The new fuel level
- The total route distance at the time of refueling

Once refueling was confirmed, I locked the system into **"monitoring mode"** to prevent further refuels until this one was evaluated. This meant that for the next few simulation steps, every frame update checked how much of the newly added fuel had been consumed and how far the car had gone since the refill. These values were incrementally updated inside the simulation loop.

For practical reasons especially to keep the simulation short during tests I decided that once the vehicle had used **just 2% of the added fuel**, I would perform an evaluation. In a real-world system, you would wait much longer, but for testing purposes this short threshold allowed me to observe many station evaluations during a single test run.

The evaluation formula was based on a straightforward comparison:

- I calculated the **expected distance** using the formula:

expected_distance = added_fuel * fuel_efficiency * 1000

where fuel_efficiency was measured in kilometers per liter (e.g., 10 km/L), and the *1000 factor converted km to meters.The expected distance is currently calculated on a fixed fuel_efficiency a improvement for the project would be to tie it to speccific car metrics, likely using maf which we already calculate.

I then compared this expected distance to the **actual distance traveled** since the refill..If the actual distance was **at least 80%** of the expected one, the station was considered "reliable." Otherwise, it was marked "unreliable."

The decision to use 80% as the threshold was based on the idea that some variance is always expected in real-world driving road incline, acceleration, stop-and-go traffic, and other factors can all reduce fuel efficiency slightly. But if the vehicle didn't even reach 80% of the expected range, that strongly suggested that something went wrong either too little fuel was added, or it wasn't being used efficiently.

Once this check was done, I updated the `reliable` field of the fuel station object and sent a message back to the frontend via SocketIO. The map marker representing the station was then updated red if unreliable, green if reliable. This gave the user visual confirmation of the result.

To preserve the evaluation, I also saved the results to a text file. The filename was generated using the station's name (or a default label if no name was available), replacing spaces with underscores. The content of the file included:

- Station name
- Final evaluation (reliable/unreliable)
- Fuel before and after refuel
- Fuel added
- Distance traveled on that fuel
- Expected distance vs actual distance

This file output serves as a log and could be used for debugging, report generation, or even training data in a future version of the system.

An additional design consideration was to prevent overlapping refueling sessions. If the user clicked “Refuel” multiple times during monitoring, it could corrupt the measurement logic. To avoid this, I set a global `refueling_locked` flag, which disabled refueling until the current evaluation finished. Once the monitoring criteria were met and the evaluation complete, I reset the `monitoring_data` and unlocked the system for the next refuel.

Through this part of the system, the simulation became more than just a real-time dashboard it also started generating insights based on user behavior and system dynamics. The ability to classify service points like fuel stations introduces a kind of **environmental intelligence**, which could be extremely useful in future telematics platforms, especially those aimed at eco-routing or driver assistance.

Although the evaluation algorithm is relatively simple, it establishes a solid framework for more advanced logic. For example, a future version could incorporate pricing data, altitude, traffic conditions, to identify stations that regularly result in poor performance.

4.3 Emulator Realism and Assumptions

One of the key decisions in this thesis was to base the entire system on a software emulator specifically, the ELM327-emulator rather than interfacing with a real vehicle or using an actual hardware-based OBD-II adapter. While this choice was largely motivated by practicality (no need for vehicle access, safety, and full control),

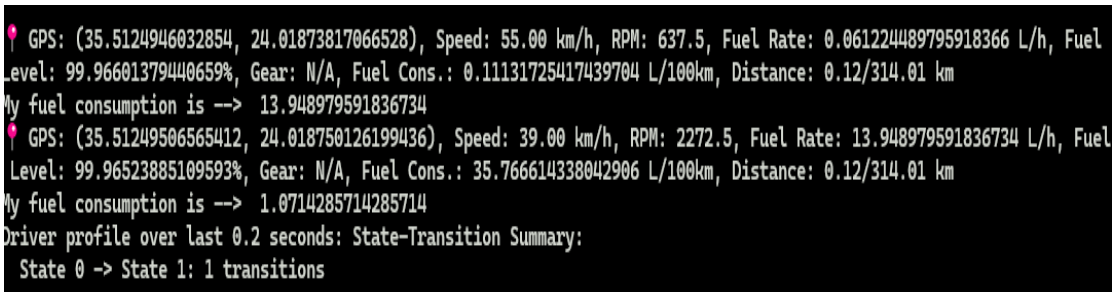
it naturally raises the question of **realism**: How accurately can an emulator replicate the behavior of a real vehicle?

Protocol and Interface Accuracy

From a communications standpoint, the ELM327-emulator behaves almost identically to a real ELM327 OBD-II adapter. It understands and replies to standard OBD-II PID commands like `010C` (engine RPM), `010D` (vehicle speed), or `0110` (Mass Air Flow). These commands are the same ones used by actual vehicle diagnostic tools and telematics platforms, and the format of the emulator's replies is indistinguishable from that of a real device hexadecimal strings formatted with carriage returns, precisely matching the ELM327 specification.

The interaction flow is also realistic: my backend sends commands through a TCP socket to the emulator in the same way that a diagnostic scanner would send commands over a serial or Bluetooth link. The emulator processes each command and returns a response, which is parsed and decoded into meaningful values (RPM, speed, etc.) in real time. This mimics the entire lower-level protocol layer and ensures that the software architecture is fully compatible with a real-world ELM327 adapter.

Logging system



To support debugging and evaluation throughout the development process, I included a logging mechanism that continuously prints the internal state of the simulation to the command line. This approach provided me with direct, real-time feedback on how the system behaved at every step, without the need for external monitoring tools. The log outputs include raw telemetry values such as GPS coordinates, vehicle speed, RPM, fuel rate, estimated fuel consumption (in L/100 km), current fuel level, and gear position. These values are printed at a frequency of around 10 Hz, giving a live stream of the simulated driving session.

In addition to basic telemetry, the logging system also reports derived metrics. For example, every time the Mass Air Flow (MAF) is processed, the system calculates an instantaneous fuel consumption rate, which is printed in parallel with the other data. This allowed me to cross-check whether the formulas used for converting MAF values into liters per hour and liters per 100 km were behaving realistically.

The log also integrates the driver profiling output. As shown in the screenshot, after a small profiling window (in this case 0.2 seconds), the system produces a "State

Transition Summary." This summary indicates how many times the driver transitioned between different behavioral states identified by the KMeans clustering. For instance, in the example shown, the log captures one transition from State 0 to State 1, representing a detected change in driving style between two consecutive clusters.

This logging system played a dual role. On the one hand, it was a development tool, helping me identify bugs or unrealistic values (such as extreme RPM spikes or negative fuel levels). On the other hand, it also acted as a form of validation, allowing me to verify that the backend logic for telemetry, clustering, and fuel monitoring was working as intended. This incremental visibility was crucial while building the system, since it ensured that each module behaved correctly before integrating it into the full pipeline.

Value Simulation and Parameter Modeling

While the emulator accurately replicates the communication protocol of a real ELM327 device, the realism of the data it returns such as speed, RPM, or fuel rate depends entirely on the logic behind the simulation. Unlike an actual vehicle, where values are generated by physical sensors and engine control units, my system uses programmable logic to produce data that behaves like real driving conditions.

To make the simulation feel authentic, I designed these values based on well-established relationships in automotive physics. For instance, RPM values are generated in proportion to the simulated vehicle's speed, using assumed gear ratios. I also introduced slight random fluctuations or "jitter" to avoid the overly perfect behavior that would otherwise break immersion. Similarly, Mass Air Flow (MAF) is estimated in a way that reflects how an engine actually works: when RPM and throttle increase, more air enters the engine, and thus, more fuel is burned.

Fuel consumption is calculated from MAF readings using standard formulas widely cited in both academic and automotive literature. The system also simulates speed changes with smooth acceleration and deceleration curves to reflect natural driving behavior.

Although this approach doesn't reach the complexity of an engine simulator, it strikes a practical balance producing realistic, consistent, and believable telemetry that is sufficient for behavioral analysis and real-time monitoring.

Real-Time Behavior and Timeliness

Beyond data content, **timing is another critical part of realism**. Real OBD-II devices operate on a continuous polling model, where data is updated at frequencies ranging from 1 Hz (for slow diagnostics) to ~10 Hz (for real-time streaming). My backend replicates this by sending new commands every 100 milliseconds and immediately processing the responses. This aligns with how vehicle data loggers or

telematics devices behave in production environments, where WebSocket feeds or MQTT brokers stream this data to servers for processing.

In this system, the backend maintains a loop that generates and updates data at roughly **10 Hz**, which matches the polling frequency of commercial vehicle telemetry systems. This allows the frontend to display smooth transitions in values and helps the driver profiling module detect subtle behavior changes in near real time.

Key Assumptions and Their Justification

To ensure simulation realism without needing overly complex physics modeling, I made the following **deliberate assumptions**:

- **Fuel efficiency** is fixed at 10 km/L a middle-ground value representing compact vehicles under moderate conditions.
- **Tank capacity** is 50 liters common for small to mid-sized cars.
- **Air-fuel ratio (stoichiometric ratio)** is 14.7 for gasoline consistent with combustion chemistry.
- **Fuel density** is set to 720 g/L based on known gasoline properties.
- **Gear shifts** are not explicitly modeled, but are indirectly handled through speed-RPM curves.
- **No terrain elevation or traffic modeling** is applied, keeping the road model flat and constant-speed unless overridden by input.

These simplifications were necessary to make the system testable, repeatable, and explainable and they align well with the **goals of this project**, which are focused on software logic and behavioral modeling, not thermodynamic simulation.

Practical Value and Comparability

The emulator provides a testbed that is representative enough to validate features like:

- Driver classification using clustering
- Distance-based fuel station evaluation
- Fuel tracking and consumption monitoring
- Real-time data streaming over a web interface

It allows controlled experiments where the only changing variable is **driver input or fuel station behavior**, which is ideal for validating new ideas or algorithms. If we had used a real car, results could have been distorted by unpredictable real-world effects, and repeatability would be a challenge.

That said, this simulation environment provides a solid foundation for future experiments using real OBD-II data. Once the system has been fully tested and validated in the emulator, the same software stack could be connected to a physical

ELM327 Bluetooth adapter and deployed in an actual vehicle. Doing so would allow for direct comparison between simulated and real-world data, offering the opportunity to fine-tune the RPM, MAF, and fuel consumption equations based on real measurements. It would also make it possible to observe the impact of real-world sensor noise or communication glitches factors that are difficult to fully simulate. Most importantly this setup would help verify whether the logic used to evaluate fuel station reliability continues to hold up under real driving conditions.

4.4 Driver Profiling Using Clustering

While rule-based methods provide a simple way to classify driving behavior, they tend to be inflexible and fail to capture more nuanced or transitional driving patterns. To improve upon this, I implemented a more adaptive profiling mechanism based on **unsupervised machine learning**, using the **KMeans clustering algorithm**. This approach enables the system to group similar driving behavior patterns without requiring predefined labels or thresholds.

Data Collection and Feature Engineering

During the simulation, vehicle data is sampled at a high frequency approximately every 100 milliseconds. Each sample records the current vehicle **speed** (in km/h), a **timestamp**, and the calculated **acceleration** (in m/s^2), derived by comparing consecutive speed values over time. Acceleration is particularly valuable as a behavioral metric, as it reveals how abruptly the vehicle is changing speed, which is a common indicator of aggressive or erratic driving.

Each sample is transformed into a two-dimensional feature vector:

Feature Vector=[Speed,Acceleration]\text{Feature Vector} = \left[\text{Speed}, \ \text{Acceleration} \right]Feature Vector=[Speed,Acceleration]

These vectors are collected over a short time window, typically 5 to 10 seconds, forming a snapshot of the driver's recent behavior.

Clustering with KMeans

Once a sufficient number of feature vectors is accumulated, the data is passed to the **KMeans algorithm**, which attempts to partition the observations into a fixed number of clusters. After experimenting with several values for k, I selected **three clusters**, as this provided a reasonable balance between granularity and interpretability. In practice, these clusters tended to correspond to intuitive behavioral states commonly changing between clusters would mean erratic behavior.

KMeans calculates the **centroids** of each cluster and assigns each sample to the closest centroid based on Euclidean distance. This classification is done in real time, and the resulting cluster labels are used to infer the current driving style.

Modeling Transitions with Graphs

To go beyond static classification, I implemented a dynamic **state-transition model** using a directed graph structure. Each node in the graph represents one of the three clusters (behavior states), and edges denote transitions between them. Edge weights track the frequency of transitions, providing a visual and data-driven summary of how the driver moves between behavioral states over time.

This method captures more than just the type of behavior; it reveals the **stability or volatility** of driving. For example, a stable driver might remain within a single cluster for an extended duration, while a volatile driver would show frequent transitions across multiple clusters.

Real-Time Processing and Reset Logic

To ensure that different segments of a simulated trip are evaluated independently, the clustering model is reset periodically typically every few seconds. This prevents behavior in one phase of the journey from influencing classification in another, such as when transitioning from urban to highway driving. After each reset, a new cluster model is constructed from recent data, and the frontend interface is updated with the current behavioral summary.

This update is transmitted via a WebSocket message, allowing the user interface to reflect changes in driving behavior in near real time.

Evaluation and Limitations

The clustering-based profiling system has shown itself to be much more flexible and insightful than a simple rule-based approach. Instead of relying on hard-coded thresholds, it is able to automatically discover patterns in how the driver behaves and group them into meaningful categories. This makes the system responsive to a wide variety of situations: sudden accelerations are quickly detected, calm cruising is consistently recognized, and transitions between the two are captured in real time. The result is a richer and more dynamic picture of driver behavior than what fixed rules could provide.

That said, there are some important limitations to this method. One challenge comes from the assumptions built into the KMeans algorithm itself. KMeans treats each cluster as if it were roughly spherical in shape and of similar size in the feature space. In practice, this means the algorithm expects that “normal” driving behavior should form a neat, roundish group of points, and that “aggressive” or “relaxed” behaviors should form clusters of a similar spread. Real world driving data is rarely this tidy: one driver might spend long periods cruising calmly (creating a dense, compact cluster), while their aggressive bursts of acceleration might be rarer but spread out over a wider range of values. In such cases, KMeans can oversimplify the data or misclassify edge cases.

Another limitation is that the algorithm does not naturally account for the *order* in which events occur. Each data point each combination of speed and acceleration is

considered independently, without knowledge of whether it came before or after another. While the transition graph that I build on top of the clustering does capture these sequences, the clustering step itself has no memory. This can sometimes cause it to miss context that might be important, such as a short acceleration that is only meaningful if it follows heavy braking.

KMeans also requires the number of clusters to be specified in advance. In this project, I chose three clusters, which worked well for balancing detail with interpretability. But this choice may not generalize across all vehicles or driving environments. Some datasets might naturally fall into two categories (calm vs. aggressive), while others might reveal four or five distinct styles. More adaptive methods, such as density-based clustering or dynamic selection of k, could provide greater flexibility in future work.

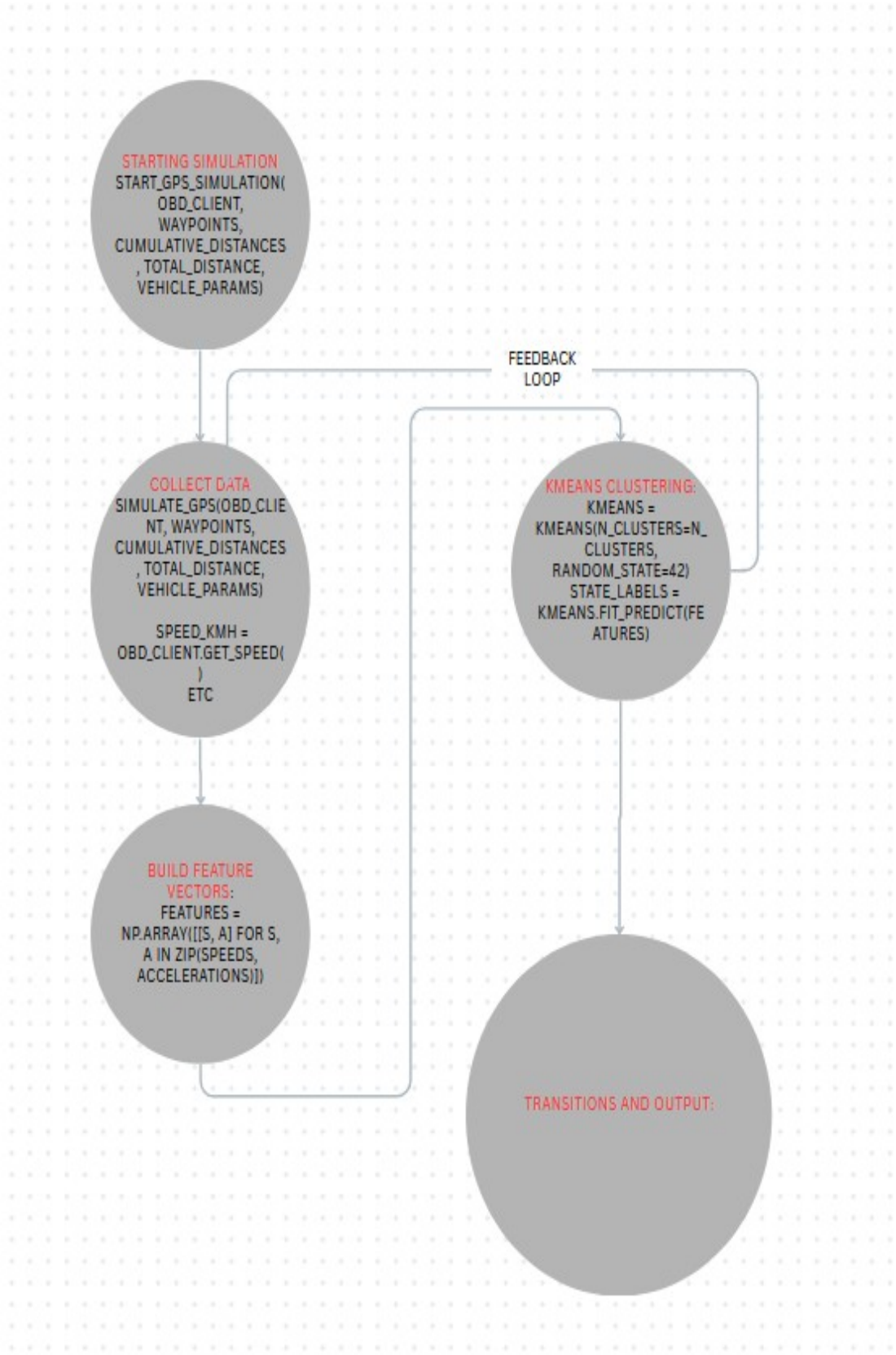


Figure 3: Driver Profiling Pipeline Flowchart

Despite these limitations,i think that K clustering offers significant value in the scope of this thesis

This diagram illustrates the real-time data processing pipeline used for driver behavior profiling. The system begins by continuously collecting key vehicle metrics such as speed and calculating acceleration between samples. These values are stored along with timestamps to form a temporal dataset. Once a sufficient number of samples is accumulated, the data is passed into the KMeans clustering algorithm, which groups the driving behavior into one of several predefined clusters based on speed and acceleration. The system then constructs a transition graph to monitor behavioral shifts over time and generates a textual summary that is sent to the frontend. This flow ensures that driver behavior is continuously analyzed and visualized throughout the simulation. The feedback loops in the diagram represents the continuing feeding of data from our emulator (in a real world scenario these data would come frokm the car interface).

Below I am devoting a section of this thesis explaining the implementation in a code snippet – explanation format:

```
sample = {
    'timestamp': time.time(),
    'speed': speed_kmh,
    'rpm': rpm,
    'fuel_consumption': fuel_consumption_l_per_100km
}
driver_samples.append(sample)
```

Explanation:
This snippet captures a single snapshot of the vehicle's state during the simulation loop. The `timestamp` records when the sample was taken, and the other fields store the speed (in km/h), engine RPM, and estimated fuel consumption in L/100km. These samples are appended to the `driver_samples` list, which serves as the input for the clustering algorithm. The integration of KMeans clustering into the driver profiling system adds significant value to the simulation by enabling dynamic, data-driven classification of behavior. This method complements the earlier rule-based approach and forms a core part of the system's ability to analyze and respond to simulated driving data in real time. It also establishes a scalable foundation for future enhancements involving more advanced machine learning models or real-world data integration.

```
dt = samples[i]['timestamp'] - samples[i-1]['timestamp']
speed_prev = samples[i-1]['speed'] / 3.6  # Convert to m/s
speed_curr = samples[i]['speed'] / 3.6
acceleration = (speed_curr - speed_prev) / dt
```

Explanation:
Acceleration is calculated based on the change in speed over time. Speeds are

converted from km/h to m/s before computing acceleration in m/s². This value is a key feature in driver profiling as it captures sudden changes in vehicle motion.

```
features = np.array([[s, a] for s, a in zip(speeds, accelerations)])
kmeans = KMeans(n_clusters=3, random_state=42)
state_labels = kmeans.fit_predict(features)
```

Explanation:
Once enough samples are collected, the system builds a 2D feature array combining speed and acceleration values. This array is passed into the `KMeans` algorithm, which clusters the samples into behavioral states. Each sample is then labeled according to its cluster.

```
for i in range(1, len(state_labels)):
    prev_state = state_labels[i-1]
    curr_state = state_labels[i]
    if G.has_edge(prev_state, curr_state):
        G[prev_state][curr_state]['weight'] += 1
    else:
        G.add_edge(prev_state, curr_state, weight=1)
```

Explanation:
This loop creates a directed graph where each node represents a driving state (cluster). Transitions between clusters are tracked by adding or updating edges in the graph. This structure helps visualize how stable or erratic the driver is.

To better illustrate the clustering-based approach used for driver profiling, i generated representative examples (proof of concept images) that simulate both stable and erratic driving behaviors. Each data point in the following graphs corresponds to a moment in time during a simulated trip, characterized by the vehicle's speed and acceleration. Using KMeans clustering, i organized this data into behavioral states that reflect distinct driving patterns. These clusters help us visualize how consistent or variable a driver is over time, which is crucial for understanding overall driving style.

Each point is a driving snapshot (speed vs. acceleration).

- Points are color-coded by KMeans cluster i.e., driving states.
- Shows how the driver’s behavior groups into distinct patterns.
- Shows how the driver’s behavior groups into distinct patterns

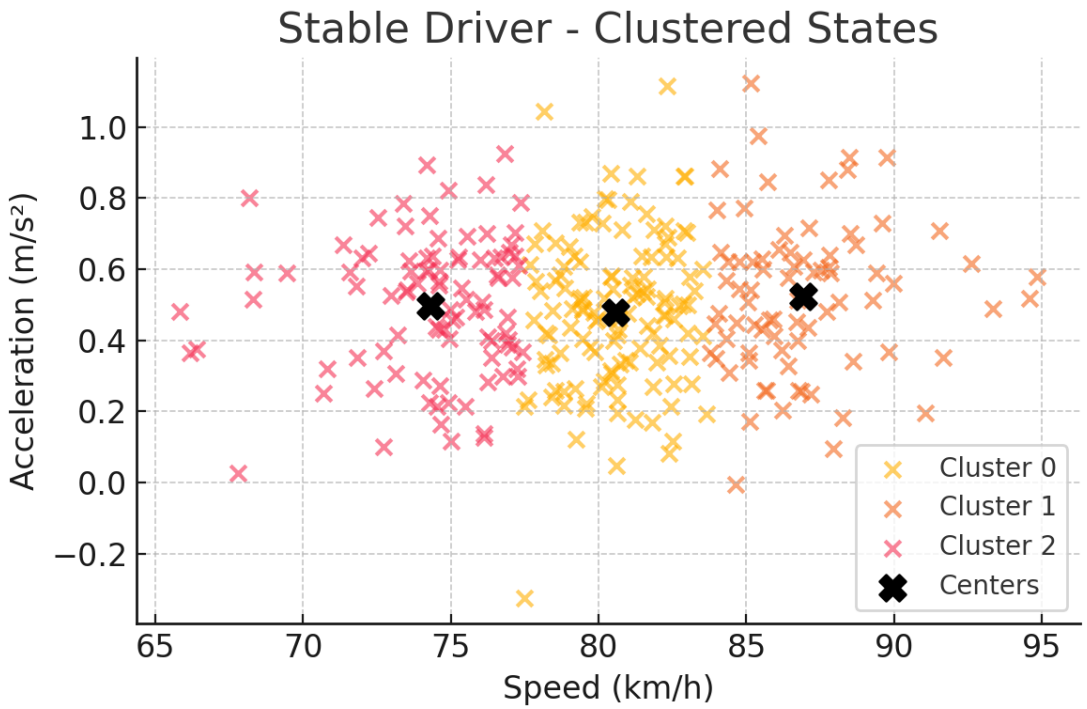


Figure 4: Stable driver example K-Means clustering

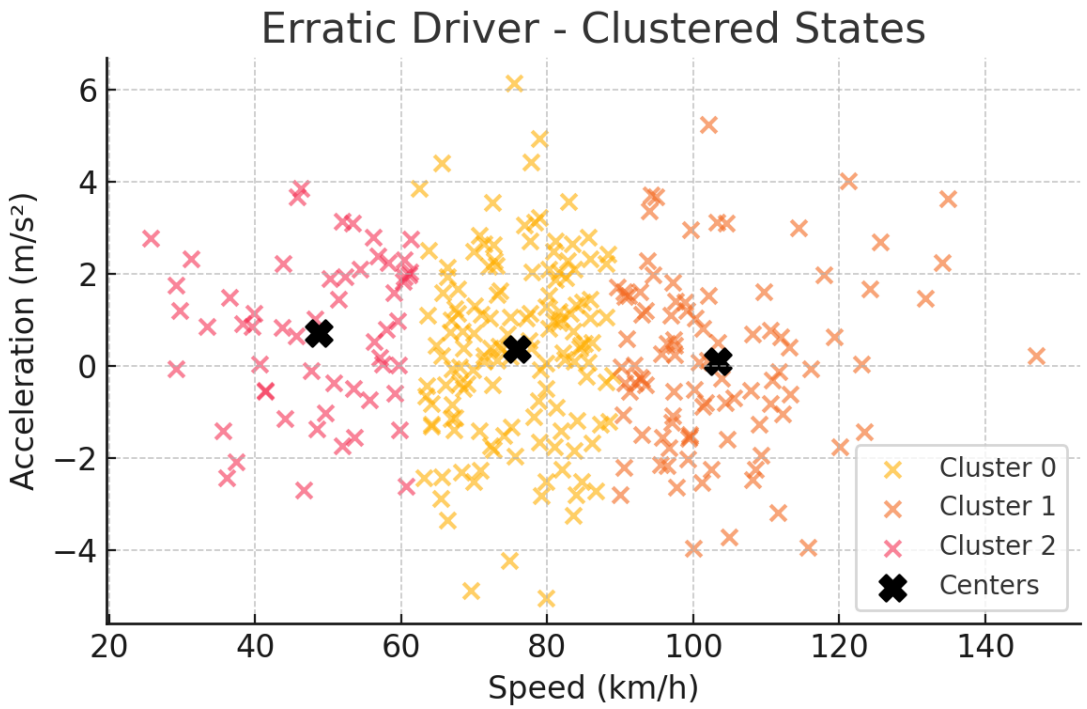


Figure 5: Erratic driver example K-Means clustering

As observed in the visualizations, a stable driver exhibits tightly grouped clusters, indicating a narrow range of speed and acceleration values and thus a steady, predictable driving style. In contrast, the erratic driver graph shows clusters that are more widely scattered, reflecting greater fluctuations in driving behavior. This erratic pattern suggests frequent transitions between aggressive acceleration, deceleration, and speed changes. These differences become even more meaningful when integrated into a state-transition graph, allowing us to monitor how often a driver switches between behavioral states and assess the overall stability of their driving profile.

4.5 Real-Time Communication and User Interface Integration

A key aspect of the system's design was ensuring that the simulation could be experienced in real time. The goal wasn't just to calculate values in the backend, but to present them immediately to the user through a live dashboard, giving the feeling of an actual driving session. To achieve this, I integrated **Flask** as the main web

server and used **Flask-SocketIO** to handle **real-time communication** between the backend and the frontend.

Backend-to-Frontend Messaging

The backend runs a loop that queries the emulator for sensor values roughly every 100 milliseconds. These values include key metrics such as speed, RPM, fuel consumption, fuel level, and driver behavior. Once collected and processed, the backend emits this data to the frontend using **WebSocket events**. Specifically, the `gps_update` event is triggered and sent to all connected clients. This event includes a structured JSON object containing all the current telemetry values, which are then displayed on the screen without requiring the page to reload.

This architecture allows the interface to feel responsive and alive for example, when the simulated vehicle speeds up, the speed value on the screen updates almost instantly. Similarly, the fuel level decreases in real time, and any changes in driving behavior are immediately reflected in the "Driver Profile" field.

VIN Submission and Initialization

At the start of a simulation, the user is prompted to enter a **Vehicle Identification Number (VIN)**. When the user clicks the "Submit VIN" button, the frontend sends a `vin_submission` event to the backend, along with the provided VIN. The backend then queries the vPIC API, retrieves vehicle parameters, and initializes the simulation accordingly.

Once this process is complete, the frontend transitions to the main simulation interface and starts listening for real-time data updates via the `gps_update` WebSocket channel.

User Controls and Refueling Interaction

In addition to displaying live data, the frontend also includes **interactive controls**. These include:

- The **Refuel (+40%)** button, which emits a `refuel_request` event when clicked.
- Input fields to add custom **waypoints**, allowing the user to modify the route dynamically.
- A map rendered using **Leaflet.js**, which updates the vehicle's GPS position as it moves.

When a refueling request is made, the backend checks whether the vehicle is near any known fuel stations. If it is, the system enters monitoring mode and emits a `refuel_status` event to inform the frontend that evaluation is in progress. Once the evaluation is complete, another `refuel_status` message is sent, and the frontend updates the station's marker color accordingly (green for reliable, red for unreliable).

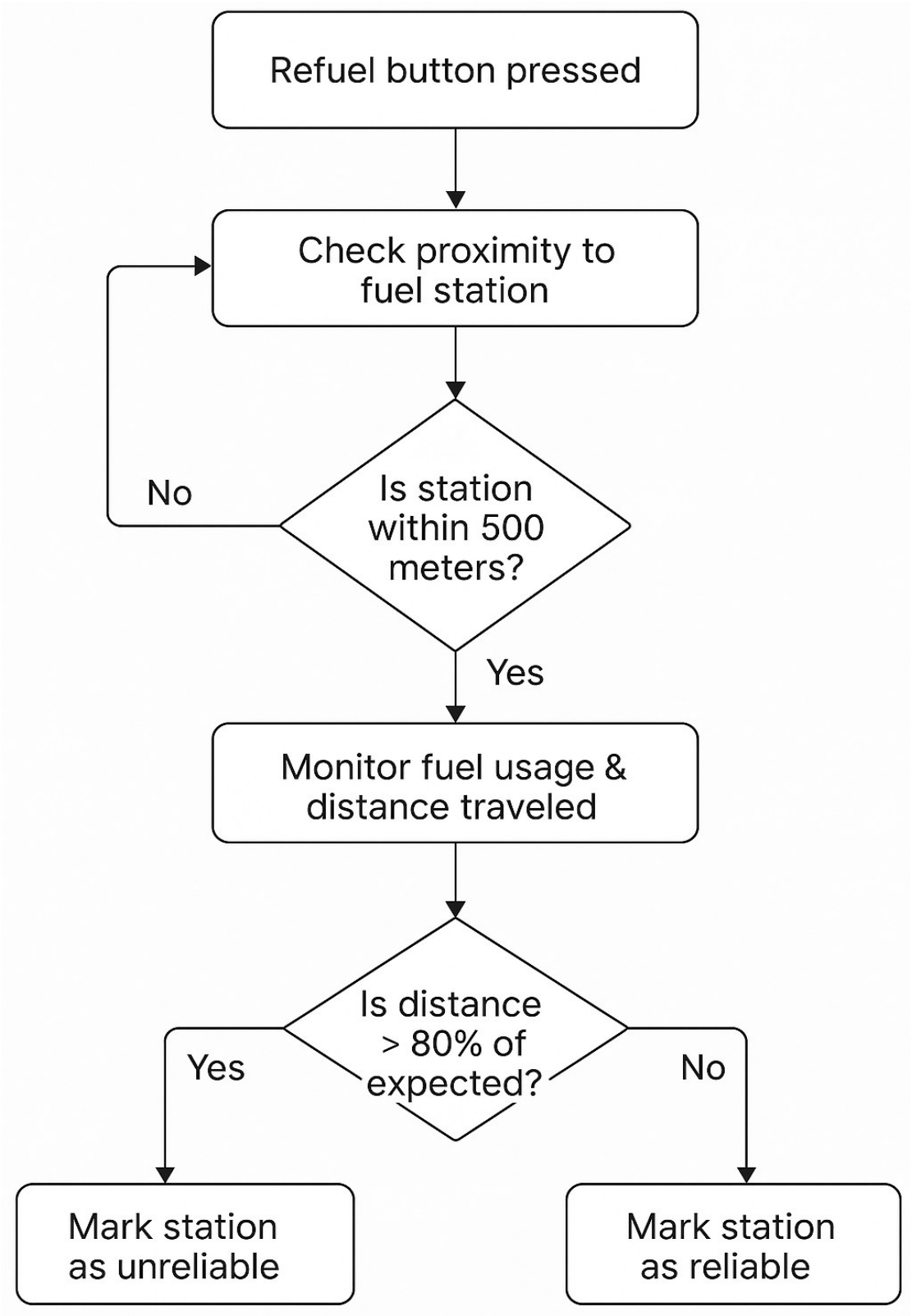


Figure 6: Fuel monitoring feature flowchart

Additionally, if a fuel station's status is updated during monitoring, a `fuel_station_update` event is sent, prompting the frontend to update the map marker in real time.

Visual Feedback and User Experience

The frontend provides a clean, minimal interface that includes:

- Live telemetry values (RPM, speed, fuel level, gear, fuel consumption)
- A map showing the vehicle's current location
- Buttons and fields for interacting with the simulation
- Status messages for refueling events or errors

This combination of real-time data updates and direct user input creates an interactive simulation that feels responsive and immersive. Although the system is entirely software-based, the integration of Flask-SocketIO with a dynamic frontend makes it possible to emulate the experience of driving and monitoring a connected vehicle.

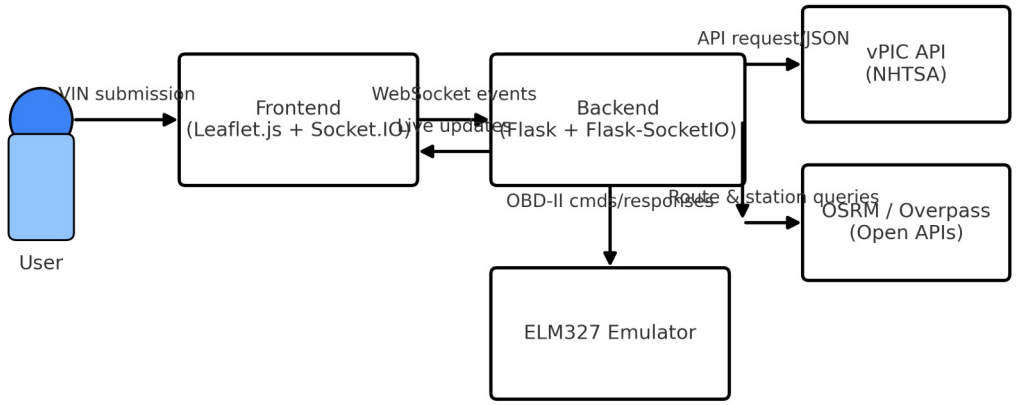


Figure 7: Architecture diagram

This architecture built around real-time WebSocket communication allows the simulation to go beyond a simple data generator. It becomes an interactive telematics platform, complete with live feedback, user-driven events, and behavior monitoring. By designing the frontend and backend to work seamlessly together, I was able to

create a responsive system where users can both observe and influence the simulation as it runs.

A key part of making this system feel interactive and realistic was designing it to operate in real time. It wasn't enough to compute values on the backend and save them to a log the goal was to allow the user to observe and interact with the simulation as if they were driving a real connected vehicle. To accomplish this, the system uses Flask as the core web server, while Flask-SocketIO handles real-time communication between backend and frontend via WebSocket events.

Every 100 milliseconds, the backend gathers fresh data from the emulator. This includes the current speed, engine RPM, fuel consumption, fuel level, and the assigned driver profile based on recent behavior. These values are packaged into a structured JSON object and sent to all connected clients using the `gps_update` event. The frontend listens for this event and updates the display immediately without refreshing the page or making explicit polling requests. As a result, whenever the simulated vehicle speeds up or consumes fuel, those changes are reflected live in the interface.

Here is a small portion of the backend code that emits telemetry data:

```
data = {
    'speed': current_speed,
    'rpm': current_rpm,
    'fuel_level': current_fuel_level,
    'fuel_consumption': current_fuel_rate,
    'profile': current_driver_profile,
    'lat': lat,
    'lon': lon
}
socketio.emit('gps_update', data)
```

On the frontend side, the JavaScript client listens for these updates using:

```
socket.on('gps_update', function(data) {
                    document.getElementById('speed').innerText     =
data.speed.toFixed(2);
    document.getElementById('rpm').innerText = data.rpm;
                document.getElementById('fuel_level').innerText    =
data.fuel_level.toFixed(1);
             document.getElementById('driver_profile').innerText   =
data.profile;
    gpsMarker.setLatLng([data.lat, data.lon]);
```

```
    map.panTo([data.lat, data.lon]);
});
```

This mechanism allows for near-instantaneous synchronization between the simulation backend and the live dashboard. Users experience smooth updates of vehicle speed, RPM changes, and shifting driver behavior classification giving the illusion of a live telemetry feed. The responsiveness of the system is further improved by keeping the update rate consistent at 10 Hz, which is a common frequency used in commercial-grade vehicle telematics.

At the beginning of each simulation, the user enters a Vehicle Identification Number (VIN) into a form field and clicks "Submit." The frontend then emits a `vin_submission` event to the backend. The backend queries the vPIC API, retrieves vehicle-specific metadata like fuel type, and uses that to initialize the simulation's internal parameters. Once that's done, the frontend transitions from the input view to the main simulation interface, which immediately starts receiving and displaying real-time data from the backend.

In addition to passive updates, the user can also interact with the simulation. A "Refuel (+40%)" button is available, which sends a `refuel_request` event to the backend. The backend checks whether the simulated vehicle is close enough (within 500 meters) to a known fuel station. If so, the system locks into a monitoring mode and begins evaluating the effectiveness of that refuel. During this time, refueling is disabled, and the frontend reflects this change by updating the button state and showing status messages. Once the fuel evaluation completes, a `refuel_status` message is sent to unlock the button, and a `fuel_station_update` message is triggered to update the map marker color (green for reliable, red for unreliable).

All of this is displayed through a clean interface that shows live vehicle telemetry, a GPS map, and real-time behavior updates. Users can also manually add waypoints or observe the vehicle move across the map via Leaflet.js, which renders each new GPS coordinate in sync with backend updates. Thanks to the real-time nature of Flask-SocketIO and careful frontend optimization, the experience feels smooth, responsive, and immersive.

From a performance perspective, the choice to emit updates at 10 Hz ensures that the interface remains responsive without overwhelming the browser or SocketIO queue. During testing, this rate proved effective for catching acceleration spikes, monitoring driver state transitions, and tracking rapid changes in fuel consumption.

The overall architecture creates a feedback loop between the user and the system. Actions taken on the frontend, such as VIN submission or refueling, produce visible and immediate effects. Meanwhile, backend telemetry streams maintain an up-to-date reflection of simulated vehicle behavior. Although the entire system is software-

based, this real-time design makes it feel interactive and dynamic, closely approximating the experience of working with a live vehicle interface.

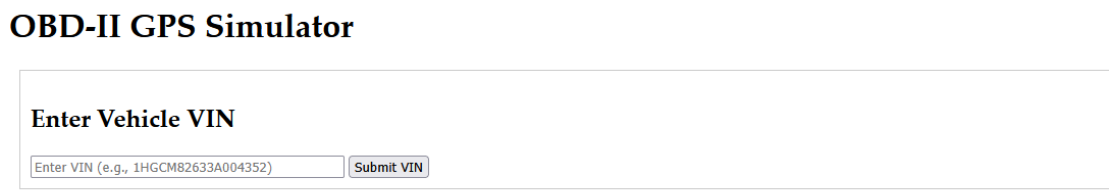


Figure 8: FIrst screen of program (Inserting VIN)

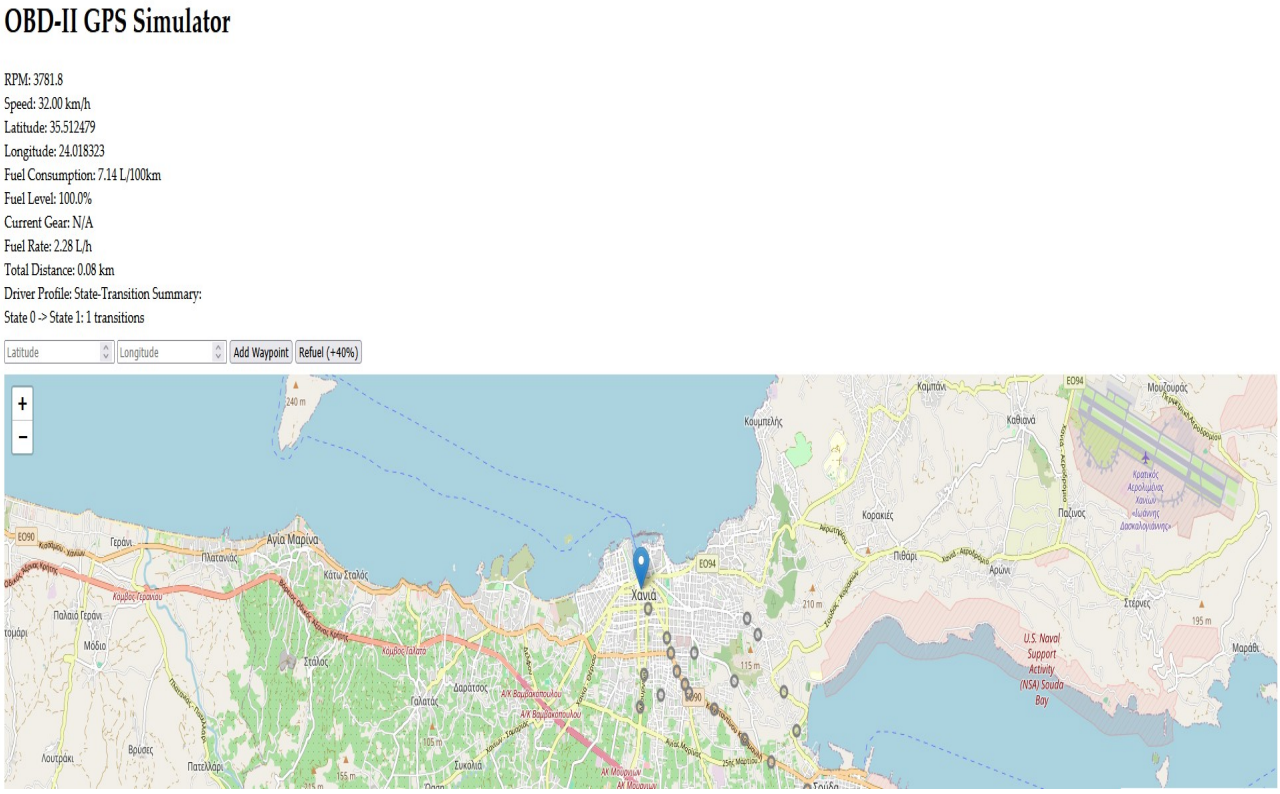


Figure 9: Interface

One of the central components of the system is the main simulation loop, which is responsible for coordinating all real-time updates, behavior analysis, and data transmission. Running at a fixed interval of approximately 100 milliseconds, this loop acts as the heartbeat of the simulation. Each cycle begins by sending OBD-II commands to the emulator to retrieve current readings such as vehicle speed, RPM, and MAF (Mass Air Flow). These values are parsed, processed, and then used to update internal metrics, including fuel level and gear estimation.

This loop also plays a critical role in driver profiling. It continuously collects time-stamped snapshots of the current state and appends them to a buffer used by the clustering and rule-based profiling logic. Every few iterations, the buffer is analyzed to detect behavior transitions (e.g., aggressive to relaxed) using either threshold logic or KMeans-based clustering, depending on which method is active. The results are sent to the frontend in real time, keeping the user informed of their current simulated driving state.

In addition to telemetry and profiling, the loop manages refueling logic. If the system is in monitoring mode following a refueling event, the loop accumulates the distance traveled and fuel burned since the refill. Once a minimal threshold (e.g 2% of the added fuel) is reached, it triggers the station evaluation logic and unlocks refueling for the next session.

From a design perspective, structuring the application around a single, centralized loop simplifies synchronization and makes timing behavior predictable. Rather than relying on multiple asynchronous callbacks or complex multi-threaded scheduling, this approach ensures that all data processing steps remain tightly coordinated. It also makes the system easier to debug and extend in the future. Since all telemetry, profiling, and evaluation updates happen in the same place, it becomes straightforward to insert new logic (e.g., road event simulation or adaptive fuel modeling) without disrupting the overall flow. A critical part of the system's fuel station evaluation logic involves comparing the **actual distance** traveled by the vehicle after a refueling event with the **theoretical expected distance** it should have covered, based on the amount of fuel added and the assumed fuel efficiency. In this design, a fuel station is marked as *reliable* if the vehicle travels at least **80%** of the expected distance. Otherwise, the station is classified as *unreliable*.

The selection of this 80% threshold was a carefully considered design decision, balancing **real-world variability**, **simulation constraints**, and **practical usability**. In a perfect world with ideal conditions and no noise, a vehicle consuming fuel at a constant rate would travel exactly the expected distance (e.g., 10 km per liter × fuel added). However, real-world driving is rarely perfect. Factors like acceleration spikes, engine load, slight elevation changes, or inconsistent throttle use all cause natural fluctuations in fuel consumption even in identical driving conditions. Thus, expecting the full theoretical range to always be achieved is not realistic.

To account for these inevitable deviations, I introduced a **margin of tolerance** in the form of an 80% floor. This means the station is considered reliable if the vehicle can cover at least 80% of the distance it theoretically should have covered on the newly added fuel. This 20% buffer is meant to absorb minor inconsistencies without misclassifying a generally effective station as unreliable.

In simulation, this threshold also helps smooth out any artificial variance introduced by random fluctuations in fuel rate or RPM. For example, if a vehicle refuels and receives 20 liters of fuel (with a default efficiency of 10 km/L), the expected range would be 200 km. If the car covers at least 160 km before that fuel is used up, the system considers that result acceptable. This avoids false negatives that could arise from simulated jitter or short-term inefficiencies.

I also considered other thresholds (like 90% or 70%) during early testing. A 90% cutoff was too strict even slight variance in driving style or fuel calculations led to frequent false “unreliable” results, which felt unrealistic. On the other hand, a 70% threshold was too lenient, allowing even clearly inefficient stations to be labeled as reliable. Through experimentation, 80% proved to be a balanced point that aligned with intuitive user expectations while still being sensitive enough to catch refueling inconsistencies.

Furthermore, this design choice introduces a framework that could be expanded in the future. For instance, different vehicle types (e.g., hybrids or trucks) could have different tolerances. The threshold could also be dynamically adjusted based on road conditions or driving style. In future real-world implementations, machine learning models could even learn an adaptive deviation margin based on historical data.

In conclusion, the 80% deviation threshold serves as a flexible and empirically justified rule-of-thumb for evaluating refueling performance. It respects the complexity of real-world driving while maintaining a practical balance in a simulated environment. It’s not a perfect metric, but it introduces a structured, testable standard for how the system distinguishes reliable from unreliable fuel stations which is essential to the integrity of the station monitoring subsystem.

4.6 Archtecture

The architecture of the system follows a simple but flexible client–server style. At the center there is the backend, developed in Python with Flask and Flask-SocketIO, which works as the main coordinator of the whole application. On one side it communicates with the data source of the vehicle, which in the current version is the ELM327 emulator, while in the future the same code can be almost directly connected to a real OBD-II adapter with only small changes. On the other side the backend communicates with the frontend web application, where the user can see the live telemetry, the route on the map, and interact with basic controls such as VIN submission or a refueling action. In addition, the backend also makes calls to some external APIs, like the NHTSA vPIC service for decoding the VIN and getting the correct fuel type, or the Overpass API for finding nearby fuel stations on the route.

This separation of roles makes the architecture easier to understand and maintain. We can think of three main roles: the data source (vehicle or emulator), the processing and analytics (backend), and the visualization and interaction (frontend). The external APIs bring realism and help the system adapt to different vehicles or routes without rewriting the main logic. The frontend runs directly in the browser with HTML, JavaScript and Leaflet.js for the map. Its role is to visualize the telemetry values that are updated roughly ten times per second and also to provide the user interactions. For instance, every time the “Refuel” button is pressed or a VIN is submitted, the frontend emits a SocketIO event back to the backend.

The backend is structured around a continuous simulation loop that runs every 100 milliseconds. Inside this loop the system queries the OBD client for new values, computes derived metrics like acceleration or fuel flow based on MAF, and stores them in buffers that are later used by the driver profiling service. Profiling is done in two different ways: with a simple rule-based approach that uses thresholds, and with a clustering approach using KMeans for more flexibility. At the same time, the backend also runs the fuel station monitoring service, which checks if a refuel took place and evaluates the reliability of the station by comparing the expected distance with the actual one travelled by the simulated vehicle. Once all the processing is complete, the backend broadcasts the updated values to the frontend through SocketIO so that the user interface updates in real time.

When the simulation starts, the frontend sends the VIN to the backend. Immediately the backend queries vPIC and sets internal parameters such as the air-fuel ratio and the density of the fuel depending on the type of vehicle. After that the backend initializes the route using OSRM and begins the main loop. Every cycle, new values are retrieved from the emulator and processed. The backend keeps sending updates to the frontend, which displays them almost instantly. If the user requests a refuel, the backend checks whether the car is close enough to a known fuel station, locks the monitoring mode, and after some distance decides if the station is reliable by applying the 80% threshold rule. Once the evaluation is complete, the backend emits the results back and the frontend updates the station marker to green or red.

At the moment, all components run on a single machine: the emulator, the backend, and the frontend in the browser. Still, the design allows the backend to be deployed on a Raspberry Pi inside a real vehicle, connected to an actual OBD-II dongle. In this case, the communication logic and data contracts remain the same, the only difference is the transport method (a TCP socket in the emulator versus a serial or Bluetooth connection in the car). Because the architecture is modular, new elements can be added later without breaking the system. For example, more advanced machine learning models could replace KMeans, or additional APIs like weather services could be integrated without needing to redesign the core loop.

The rationale behind adopting this layered architecture is to maximize modularity and reproducibility. By separating data acquisition, processing, and visualization, the system remains flexible enough to swap out emulated data sources with real OBD-II

inputs without requiring structural changes. Furthermore, the client–server model makes the platform deployment-agnostic: it can run entirely on a single laptop during development or be distributed between an in-vehicle unit (e.g., Raspberry Pi with OBD-II dongle) and a remote monitoring interface. This flexibility ensures that the same architecture can support both controlled academic experiments and early-stage real-world applications

5.Integration with Real Vehicles

While the present work used a software emulator to generate OBD-II signals, the designed architecture can be directly adapted to a real vehicle environment with minimal adjustments. The key modification lies in replacing the emulated data source with a real OBD-II connection, while keeping the processing and visualization pipeline unchanged.

Hardware Setup

1. **Vehicle Interface (OBD-II Port):**
   All modern cars (post-2001 in Europe, post-1996 in the U.S.) are equipped with an OBD-II diagnostic port. This port provides access to standardized parameters such as vehicle speed, RPM, fuel level, throttle position, and diagnostic trouble codes.
2. **OBD-II Adapter:**
   A standard ELM327-compatible dongle (USB, Bluetooth, or Wi-Fi) can be plugged into the OBD-II port. For research use, USB or Wi-Fi dongles are often preferred due to more stable data rates compared to Bluetooth.
3. **Computing Unit:**
   - A **laptop** can be used in the vehicle for initial testing.
   - For permanent installation, a **Raspberry Pi** or similar single-board computer can host the backend services. This unit connects to the OBD-II adapter and runs the same Python backend used in the emulator environment.
   - Power can be supplied from the car's 12V outlet or directly wired via a DC-DC converter.
4. **Networking:**
   The computing unit can open a local Wi-Fi hotspot or connect to cellular data for external access. This enables real-time communication between the backend and a dashboard device (tablet, phone, or in-car display).

Software Setup

1. **Data Acquisition:**
   Instead of reading from the emulator, the backend imports the `python-OBD` library, which allows direct communication with the ELM327 dongle.

Figure 10: ELM327 dongle

2. Backend Integration:
The collected live data is fed into the same **Flask-SocketIO backend** used in the simulation. No major changes are required, since both the emulator and a real adapter expose OBD-II values in the same format.

3. Analysis Pipeline:
The clustering (KMeans) and driver profiling logic run unmodified, as the data structure remains identical. The only difference is that values are now sourced from the car in real-time rather than from emulated streams.

Frontend Visualization:
The web-based frontend continues to receive events via SocketIO. Dashboards, graphs, and alerts are displayed on any connected device (laptop, tablet, or smartphone).

5.1 Considerations for Real-Car Integration

Although the present system has been developed and validated in a software-based emulation environment, certain practical issues must be taken into account by anyone attempting to connect it to a physical vehicle in the future. First, real OBD-II adapters may use different transport layers (USB, Bluetooth, or Wi-Fi), each requiring compatible drivers and libraries (e.g., `pyserial` for COM ports). Timing is also

more restrictive in real conditions: the ECU may stop responding if requests are too frequent or improperly terminated. Therefore, adaptive delays and error-handling mechanisms must be integrated. Finally, safety considerations must be emphasized: live connections to a moving vehicle can interfere with diagnostic functions or even affect performance if intrusive commands are sent. For this reason, early tests should be performed only with non-critical read-only queries, ideally with the car stationary, before scaling to on-road scenarios.

With that in mind, I believe we could still make a technical dive on what are some expected modifications that will be needed if anyone tries to use the current code in a real car.

At present, the system communicates with the ELM327 emulator over a TCP socket, which allows reproducible testing without physical hardware. For example, the current implementation connects to the emulator running locally on port 35000 :

```
client = OBD2EmulatorClient(host='localhost', port=35000)
client.connect()
response = client.send_command("010C")  # Request RPM
print(response)
```

This design works seamlessly with the emulator, which guarantees synthetic responses for all supported PIDs. However, when connecting to a real vehicle, several considerations arise. First real ELM327 adapters are typically accessed via **serial interfaces** (e.g., `COM3` on Windows or `/dev/ttyUSB0` on Linux) or Bluetooth, not TCP. Instead of sockets, a library such as `pyserial` would be required:

```
import serial
ser = serial.Serial("COM3", baudrate=38400, timeout=1)
ser.write(b"ATZ\r")          # Reset the adapter
print(ser.readline())        # Read the response
```

It is also worth mentioning that while the emulator always responds with valid data, a real ECU may return `NO DATA` if a PID is unsupported. For example, querying RPM (`010C`) on the emulator produces a predictable synthetic value, but the same request on a real car may fail if the ECU does not support the parameter.

6. Conclusions

From the experiments, one of the most important findings is that clustering methods such as KMeans can detect and separate different driving patterns in a way that is much more flexible compared to using simple fixed rules. While the rule-based logic was useful for quick tests (like aggressive vs relaxed driving), the clustering approach

provided a richer and more adaptable way to understand the driver's style over time. This showed that even with limited features such as speed and acceleration, it is possible to construct a meaningful profile of the driver.

The use of external APIs added another important layer of realism. By connecting to OSRM for routes, Overpass for fuel stations, and vPIC for real vehicle parameters, the system didn't just simulate an abstract car, but could adapt to real geographic and vehicular data. For example, fuel efficiency calculations could change depending on whether the VIN represented a gasoline or diesel car, and refueling logic could be tied to actual fuel stations along a real map of Crete. This makes the simulation not only more realistic but also more practical for future research or even teaching purposes.

The main contribution of this work is showing that software emulation can be a very effective testbed for mobility-related research. It allows for reproducible experiments, low cost, and no risks since nothing dangerous is actually happening on the road. Of course, there are still clear limitations: the emulator cannot capture real sensor noise, hardware errors, or environmental conditions like weather or tire wear. Because of that, it cannot completely replace field testing. However, the architecture was designed in such a way that switching from an emulator to a real OBD-II adapter would be straightforward. This creates a clear path from simulation to real-world validation, where results can be compared and the simulation can be adjusted accordingly.

In the bigger picture, this thesis contributes to the wider field of smart mobility and intelligent transportation by showing how driver profiling, fuel evaluation, and live data visualization can be done with minimal infrastructure. It makes this type of research more accessible, especially for students, researchers, or small labs who don't have access to fleets of vehicles or expensive telematics equipment. Personally, I believe the system I built can serve as a solid foundation for further work, whether that's integrating real data, adding more advanced machine learning, or expanding it into larger smart city contexts.

Even though there were challenges along the way, such as tuning thresholds, dealing with clustering stability, or ensuring the emulator produced believable data, the project proved that meaningful insights can be extracted even in a purely digital environment. This mix of challenges and achievements makes me confident that software-based emulation is not only a useful academic tool but also a stepping stone towards more advanced applications in mobility research.

7 Future Work

Although the current system provides a solid foundation, several areas remain open for future development. One promising direction is the integration of real OBD-II data using Bluetooth or USB adapters. This would allow direct validation of the simulated values and help refine the equations for fuel consumption, RPM, and airflow to better reflect actual engine dynamics.

Another extension would be the use of more advanced machine learning models. While KMeans provided a simple and interpretable way to identify behavioral clusters, future work could explore methods such as Gaussian Mixture Models, Hidden Markov Models, or even deep learning approaches that incorporate temporal dynamics. This would make it possible to capture more subtle driving styles and provide more nuanced behavioral insights.

On the application side, expanding the refueling evaluation module into a comprehensive fuel efficiency advisor could be highly valuable. By considering additional variables such as road grade, traffic patterns, or even dynamic fuel pricing, the system could evolve into a decision-support tool for both drivers and taxi fleet managers. Several hard-coded parameters could change to simulate more accurately specific simulation scenarios, the code base is designed with extensibility in mind.

Finally, deploying the platform in a connected mobility context—where multiple vehicles exchange data with each other and with infrastructure—would open the door to collaborative driver profiling and large-scale studies of mobility behavior. Such extensions would align the project more closely with the goals of smart cities and intelligent transportation systems.

7.1 Expanding the System with a Shared Database and Distributed Architecture

A logical next step for improving this project would be to create a shared database that lets multiple drivers exchange information about fuel station reliability. At the moment, each user's evaluation is stored locally, so the system only knows about one driver's experience. However, if every driver's data could be uploaded and stored in a central database, the whole platform could evolve into a crowdsourced tool. This would allow users to see which fuel stations are consistently reliable or not, based on hundreds of refueling sessions instead of just one.

The database could include information such as station locations from the Overpass API, average reliability ratings, and summaries of the drivers' profiles based on KMeans clustering (for example, whether someone drives in a relaxed or aggressive way). Using a proper database system like PostgreSQL or MongoDB would make it possible to store, index, and query all this data efficiently. On the frontend map, each station could display its reliability score in real time, such as "90% reliable, based on 150 drivers."

Privacy would also be an important aspect. Personal data about drivers should be anonymized, and evaluations could be weighted based on the consistency of each driver's behavior. That way, the results would be fair and resistant to outliers caused by irregular driving conditions. A few techniques like data averaging, anomaly detection, and confidence scoring could make the results much more trustworthy.

Another big improvement would be to redesign the platform as a distributed system. Right now, everything runs on a single computer the simulator, the analysis, and the APIs. In a more realistic setup, a small device like a Raspberry Pi could be installed in

a car to collect live OBD-II data using a Python library such as `python-OBD`. The Pi would send this data over Wi-Fi or cellular connection to a remote server, where the heavy computations would take place. The server would be responsible for clustering driver behavior, evaluating refueling events, communicating with APIs like vPIC or OSRM, and storing everything in the shared database.

This kind of architecture would make the system scalable and much closer to a real-world connected vehicle environment. The car would only handle the lightweight task of collecting and sending data, saving energy and reducing cost, while the cloud server would manage the processing for many drivers at once. The main challenges would be ensuring low-latency communication, dealing with unstable network connections, and making sure no data is lost when the connection drops. These could be solved by using reliable data transfer methods like WebSocket or MQTT, and by buffering data locally until it can be uploaded safely.

Overall, adding a shared database and distributed architecture would transform this project from a single-user simulation into a collaborative platform. It would allow large-scale driver profiling, community-based fuel station evaluation, and possibly even contribute to smarter transport and energy policies in the future.

Going distributed also opens the door to fancier AI. KMeans is great for clustering driving styles, but it's a bit basic it assumes neat clusters and doesn't track how behavior changes over time. On a server with more power, we could try:

- **Gaussian Mixture Models (GMM)**: Unlike KMeans, GMMs can model non-spherical clusters and assign probabilistic memberships, capturing more nuanced driving behaviors. For example, a driver might exhibit a mix of relaxed and aggressive traits, which GMMs could quantify with confidence scores.
- **Hidden Markov Models (HMM)**: HMMs could model temporal dependencies in driving behavior, recognizing sequences of actions (e.g., frequent braking followed by acceleration) that KMeans overlooks. This would improve the system's ability to predict long-term behavioral trends.
- **Deep Learning Models**: Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) networks could analyze time-series telemetry data to detect complex patterns, such as fatigue-induced behavior changes or context-specific driving styles (e.g., urban vs. highway). These models require significant computational resources, making a server-based approach ideal.
- **Reinforcement Learning (RL)**: An RL agent could be trained to provide personalized eco-driving recommendations, optimizing for fuel efficiency or safety based on real-time telemetry and historical driver data stored in the database.
- **Anomaly Detection**: Advanced AI could identify outliers in fuel station performance (e.g., stations consistently delivering low-quality fuel) or driver

behavior (e.g., sudden erratic driving indicating distraction), enhancing safety and reliability insights.

These models would leverage the centralized database to train on aggregated multi-driver data, improving accuracy and generalizability..

References

Driver Profiling and Behavior Analysis

1. Castignani, G., Derrmann, T., Frank, R., & Engel, T. (2015). *Driver behavior profiling using smartphones: A low-cost platform for driver monitoring*. IEEE Intelligent Transportation Systems Magazine, 7(1), 91–102. https://doi.org/10.1109/MITS.2014.2328673
2. Zhang, Z., & Li, S. (2017). *Graph-based driver profiling using driving behavior pattern mining*. Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), 4262–4269. https://doi.org/10.1109/BigData.2017.8258452
3. https://www.ibm.com/think/topics/k-means-clustering

Fuel Consumption Modeling

3. Navneeth, N., Biju, B., & Sundaramoorthy, M. (2020). *Fuel consumption prediction using OBD II parameters and multiple regression models*. Materials Today: Proceedings, 37, 2341–2346. https://doi.org/10.1016/j.matpr.2020.07.523
4. Garg, R., & Singla, P. (2015). *Modeling and prediction of fuel consumption using OBD II data*. International Journal of Scientific & Engineering Research, 6(5), 1126–1131.

OBD-II Emulation and Data Acquisition Tools

5. Ircama. (n.d.). *ELM327-emulator: A Python emulator of the ELM327 OBD-II adapter*. GitHub. https://github.com/ircama/ELM327-emulator

APIs and Standards

6. U.S. Department of Transportation, National Highway Traffic Safety Administration. (n.d.). *vPIC API*. https://vpic.nhtsa.dot.gov/api/
7. OpenStreetMap contributors. (n.d.). *Overpass API*. https://overpass-api.de/
8. OpenWeatherMap. (n.d.). *Current Weather Data API*. https://openweathermap.org/api

---

Fuel Theft

9. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5057775

10.https://www.researchgate.net/publication/285614280_Assessing_the_impact_of_driving_behavior_on_instantaneous_fuel_consumption

11.Khandoker, M.R., et al. *SenseFleet: Mobile Sensing for Fleet Management and Driving Behavior Analysis*. (Findable in IEEE or ACM)

12.Eren, H., et al. *Driver Behavior Analysis with Smartphone Sensors and Machine Learning*. *Sensors*, 2012.

13.Baecke, P., Bocca, L. *The Value of Vehicle Telematics Data in Insurance Risk Selection*. *Decision Support Systems*, 2017.

14.Ghosh, A., et al. *Connected Vehicles, Telematics, and Insurance: An Overview*. *Transportation Research Part C*, 2019.

15. https://github.com/ircama/ELM327-emulator [used this implementation of the elm 327 emulator]