Groundwork of P.I.E.N.O.

Marco Savarese
271055@studenti.unimore.it

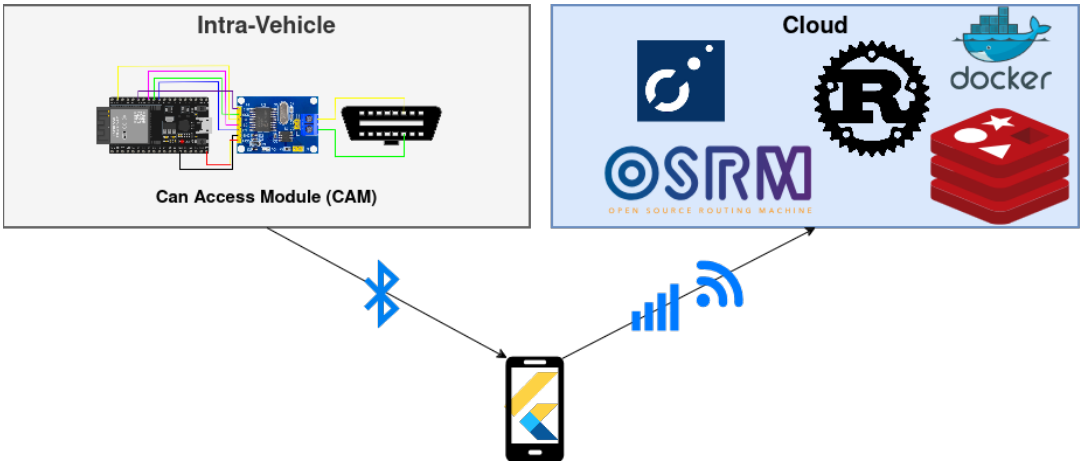

Figure 1: Complete view of the P.I.E.N.O. system.

Abstract

P.I.E.N.O. (Petrol-Filling Itinerary Estimation aNd Optimization) is a modular, cross-platform system designed to optimize vehicle refueling through real-time telemetry, fuel price forecasting, and route scoring. By reverse-engineering CAN Bus messages and integrating them with cloud-based forecasting models, PIENO creates a digital twin of the vehicle to provide intelligent and cost-effective refueling suggestions. The framework combines a CAN Access Module, a Flutter mobile app, a scalable microservice backend, and time-series models trained on public fuel price data using Prophet. This work demonstrates how data-driven decisions in everyday mobility can be enabled by merging in-vehicle data with external economic signals.

Keywords

AI, Cloud Computing, Data Mining, ITS, SDV, Smart Mobility, Smart Transportation, Sustainable Transport, Time Series Forecasting, V2X

1 Introduction

These recent times, marked by uncertainty, have brought numerous challenges, among which climate change, driven by anthropogenic emissions, stands out as one of the most difficult to address. In particular, emissions from inefficient transportation systems and non-renewable energy production are significant contributors to air pollution and are recognized as major health risk factors worldwide [26]. While long-term strategies, such as transitioning to electric vehicles or enhancing low-emission public transport infrastructure, show promise [10], they are often limited by financial and logistical constraints.

In the short term, however, modern technology can help improve how we travel. Emerging systems like Vehicle-to-Everything (V2X) and cloud computing allow for data collection from vehicles and support smarter, data-driven decision-making. These technologies form the foundation of what are known as Intelligent Transportation Systems (ITS).

P.I.E.N.O. [20] (also PIENO, from now on) was developed with these principles in mind. It targets a frequently overlooked but impactful aspect of driving: when and where to refuel. Poor refueling decisions often result in higher costs and inefficient routes. PIENO addresses this by enabling smarter refueling through intelligent data fusion. By combining internal vehicle data (such as fuel level, fuel type, and tank size) with external data sources (such as fuel station prices and locations), it recommends the optimal refueling options based on proximity and cost-effectiveness.

To achieve this, PIENO integrates OBD-II diagnostics, CAN Bus signal processing, and cloud computing to construct a virtual representation, or digital twin [6], of the vehicle. This digital model supports personalized, adaptive recommendations and encourages users to adopt more fuel-efficient behaviors. With a modular design aligned with Software-Defined Vehicle (SDV) principles, PIENO is scalable, adaptable to a wide range of vehicles, and accessible even to users with limited technical expertise.

2 Related Work

Recent advances in ITS have introduced paradigms such as SDVs, where software plays a central role in defining vehicle functionality and user experience[1]. SDVs have been explored for next-generation

Tesina d'esame Big Data & Text Analysis, March,2025, Modena, MO

[1]SDV definition proposed by BOSCH: https://www.bosch-mobility.com/en/mobility-topics/software-defined-vehicle/

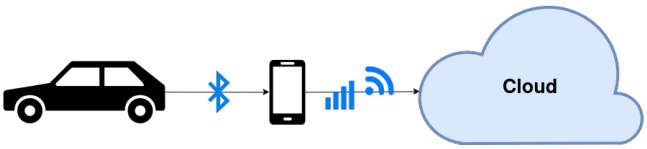

Figure 2: General view of the system.

OS design [1] and applications like FOTA [15], with the goal of harmonizing E/E architectures across OEMs.

Distinct from SDV are Software-Defined Internet of Vehicles (SDIV) and SDN-based vehicular networks, which focus on inter-vehicle aspects such as routing [7], RSU design [19], and QoS optimization [12, 17, 27]. However, these works typically treat vehicles as network nodes, neglecting challenges in bridging intra-vehicle data acquisition and external cloud services.

Some cooperative ITS applications—such as platooning [8, 11] and eco-routing [13, 25]—begin to explore this integration, but are often OEM-specific or scenario-limited. Fuel monitoring with AI, as in [23], similarly lacks a broader infrastructure integration.

PIENO addresses this gap by offering a modular, SDV-ready framework that unifies intra-vehicle telemetry with cloud-based forecasting. Unlike prior work, it enables personalized, predictive fuel optimization across diverse vehicle platforms, combining route planning with real-time and forecasted fuel prices to support sustainable, cost-effective mobility.

3 PIENO in a nutshell

P.I.E.N.O. integrates intra-vehicle data acquisition, BLE-based mobile interfacing, and cloud computation to optimize fuel stops (Figure 2). Its design is OEM-agnostic and fuel-type independent. The framework comprises:

- A CAN Access Module (CAM) built from MCP2515 [16], ESP32 [5], and an OBD-II [22] interface.
- A cross-platform Flutter app acting as a bridge and UI.
- A cloud-native backend featuring microservices in Rust and Python.
- An AI-powered time series forecasting component for fuel price forecasting (Prophet [24]).
- A C++ routing engine for distance calculation (OSRM [14]).

The CAM collects the vehicle's fuel level data and transmits it to the cloud through the user's smartphone. Once received, the cloud system processes this information, alongside fuel price forecasts generated by the Prophet model, to determine the optimal refueling time and station. The results are then presented to the user via the mobile application, which also enables navigation to the selected fuel station. Thanks to its modular and OEM-agnostic architecture, the PIENO system can be easily adapted to a wide variety of vehicle models, fuel types, and market contexts.

The source code for the parts can be found in our repository on GitHub [21].

3.1 CAN Access Module

The CAM (Figure 3) was prototyped using:

- **ESP32**: BLE and Wi-Fi capable microcontroller,

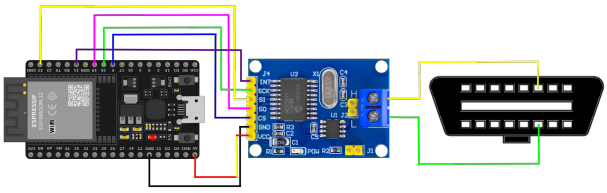

Figure 3: Can Access Module (CAM).

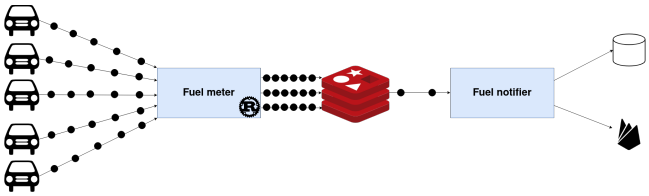

Figure 4: Decoupled fuel level acquisition architecture overview.

- **MCP2515**: CAN transceiver,
- **OBD-II**: Access point to the vehicle network.

This component is crucial for the reverse engineering of the CAN messages, that will be further analyzed in Section 4.

3.2 Mobile App (Flutter)

For the mobile application, the choice fell on Flutter due to its ability to efficiently build high-performance, cross-platform apps [9]. Supporting both Android and iOS, and potentially other systems like embedded Linux, Flutter maximizes accessibility and simplifies development by allowing developers to maintain a single code base across platforms. It is currently the most widely used framework in its category, having surpassed React Native [28].
The main features offered by the car are:

- Car registration and fuel tracking.
- Daily price forecasts via Prophet integration.
- Refuel suggestions with cost-saving metrics.

3.3 Cloud Backend

The backend infrastructure consists of a standard REST API. Particular attention was given to the potential high frequency of fuel level updates coming from many vehicles. To prevent performance issues caused by this heavy data flow, two decoupled microservices were developed (Figure 4):

- **Fuel Meter**, which exposes an HTTP API to receive BLE data and pushes all requests to a Redis queue,
- **Fuel Notifier**, which retrieves data from Redis, writes it to the main SQL database, and notifies users via Firebase Cloud Messaging.

Microservice	Average Req/sec
Login	974.2
Fuel-meter	**2084.81**

Table 1: Backend performance evaluation.

Figure 5: Test subjects.

3.3.1 Backend Throughput. To validate the effectiveness of the backend optimizations, an HTTP load-testing tool was used.[2] The goal was to measure how many requests the Fuel Meter microservice could handle per second. For comparison, we tested the login service, which performs an SQL query and a hash computation, under the same conditions. The results, shown in Table 1, were impressive: Redis decoupling more than doubled the throughput under load.

3.4 Fuel Station Scoring

To determine the most cost-effective fuel station considering both price and distance, a simple scoring formula was used:

$$\text{Score} = A + B.$$

Where:

$$A = \text{cons}_{\text{km}} \cdot \text{distance}, \quad B = (1 - \text{fuel\_level}) \cdot \text{price}_{lt}.$$

This formula balances the fuel consumption required to reach a given station (A) with the cost of completing the refueling process at that station (B).

4 CAN Reverse Engineering

While the OBD-II port is a standardized (and mandatory) diagnostic interface for vehicles, with the CAN Bus [2] serving as the main protocol for formatting and exchanging messages between car components, the same cannot be said about the content of these messages. Automotive standards, such as SAE-J1979, attempt to provide a consistent mapping between message IDs and the information being transmitted over the CAN Bus. However, due to a variety of reasons, ranging from security concerns to economic interests, different vehicle manufacturers implement their own proprietary message formats. As a result, the same type of data, such as fuel level, may correspond to different CAN IDs depending on the OEM.

For the proof of concept of PIENO, two cars were selected: a 2007 Citroën C3 and a 2017 Citroën C3 Aircross (Figure 5). The process, in theory, is quite straightforward:

(1) Using the CAN Access Module (Figure 3), we connect to the 6^{th} and 14^{th} pins of the OBD-II port, following the standard pinout [4]. The CAM is flashed with firmware that dissects each CAN message into its components: the ID and four data bytes. It then prints them to a serial monitor.

```
void loop() {
    struct can_frame frame;
    if (mcp2515.readMessage(&frame) == MCP2515::ERROR_OK) {
        Serial.println("Message received");
        Serial.print("ID: ");
        Serial.println(frame.can_id);
        Serial.print("Byte1: ");
        Serial.println(frame.data[0]);
        Serial.print("Byte2: ");
        Serial.println(frame.data[1]);
        Serial.print("Byte3: ");
        Serial.println(frame.data[2]);
        Serial.print("Byte4: ");
        Serial.println(frame.data[3]);
        //...
        Serial.print("Time: ");
        Serial.println(millis());
  }
}
```

Listing 1: First general code.

(2) A PC connected to the CAM exposes a serial interface and redirects the CAN Bus traffic to a file.
(3) The standard ID (01 2F in hexadecimal) is used to filter out the fuel level and send it to the cloud.

However, after extensive traffic capture sessions on the 2017 C3 Aircross, no message matched the standard ID, indicating that Citroën does not adhere to the standard. Therefore, the correct fuel level message had to be reverse engineered.

4.1 Reverse Engineering Pipeline

One important observation is that, in a CAN Bus message, the relevant information is often contained in just one of the four data bytes, while the remaining bytes may serve as checksums or control values.

Assuming the fuel level data is indeed transmitted over the CAN Bus and not via a proprietary protocol, the reverse engineering process was carried out as follows:

(1) **Capture CAN traffic** during a €10 refuel event.
(2) **Filter out** CAN IDs whose data bytes have a maximum value of zero, low variance, or do not increase during the refueling.
(3) **Plot the four data bytes** over time to visually correlate them with the expected fuel level behavior.

To perform this analysis, a Python script was developed using the `pandas` and `matplotlib` libraries (full script available in Appendix A). This script allowed us to accelerate the reverse engineering process, narrowing the possibilities down to three candidate IDs (Figure 6–8).

As shown in the graphs, the byte that best corresponds to the fuel level is Byte 4 of ID 1554 (Figure 7). Interestingly, it appears that Citroën transmits the fuel level directly as a value between

[2] https://github.com/mcollina/autocannon.git

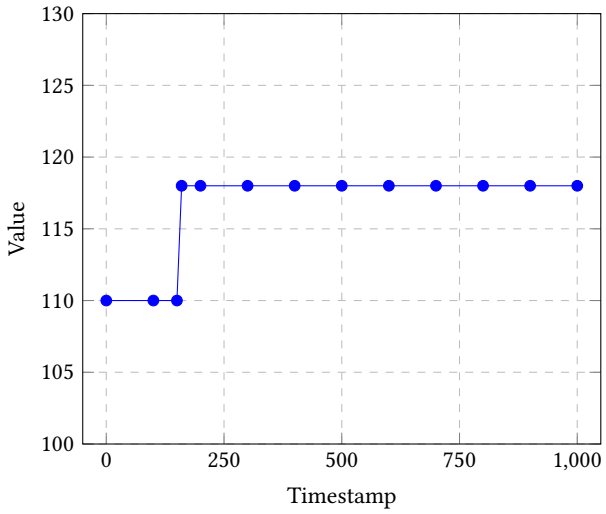

Figure 6: ID 1426 - Byte 2.

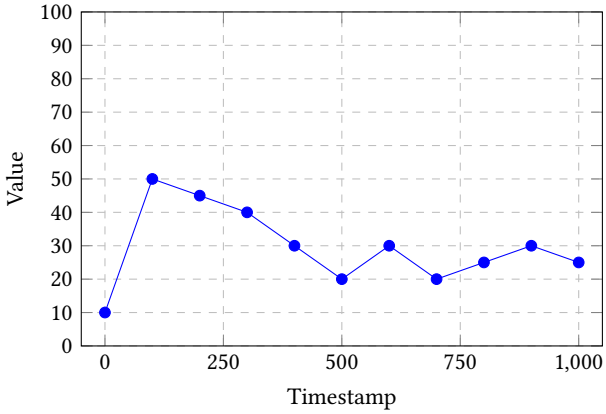

Figure 7: ID 1554 - Byte 4.

0 and 100 in Byte 4. This deviates from the SAE-J1979 standard, which defines the fuel level data in Byte 4 to be scaled as follows:

$$\frac{100}{255} \cdot A.$$

After identifying the correct data, we repeated the test on the other Citroën vehicle, this time filtering only messages with ID 1554. We began the test with a fuel level at 40% and filled the tank completely. Additionally, we updated the microcontroller's code to automatically filter and process only the ID 1554 messages:

```
void loop(){
  struct can_frame frame;
  if (mcp2515.readMessage(&frame) == MCP2515::ERROR_OK) {
    Serial.println("Message received");
    if(frame.can_id == 1554) {
      /* same debug prints ... */
      int currFuelLevel = frame.data[3];
      if (currFuelLevel != fuelLevel){
        fuelLevel = currFuelLevel;
        pCharacteristic->setValue(fuelLevel);
        pCharacteristic->notify();
```

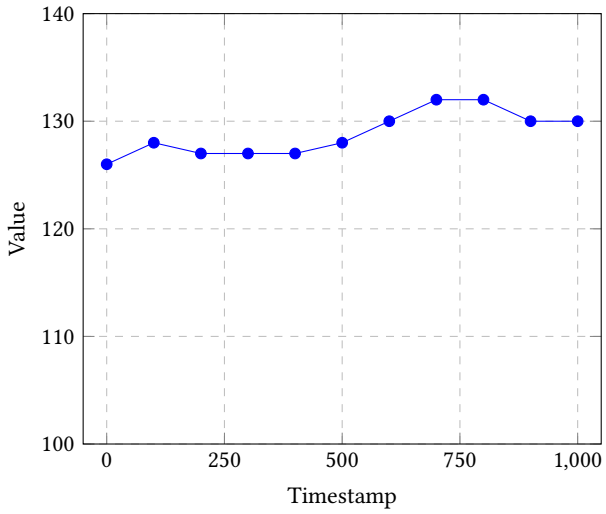

Figure 8: ID 813 - Byte 1.

```
        }
      }
    }
  }
```

Listing 2: Specific code for Citroën.

As it can be seen in Figure 9, the same value on the 2007 Citroën C3 follows the same pattern: the value being on ID 1554 and not needing any transformation to be read. In addition, it seems that the adoption by the 2007 C3 of a digital meter makes the value more stable, whereas the electrical meter of the 2017 C3 Aircross while being more precise brings to many fluctuations around the threshold value.

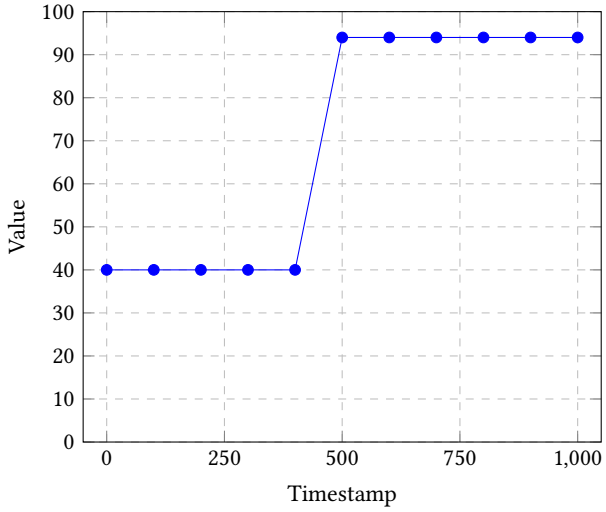

Figure 9: ID 1554 - Citroën C3 2007.

The reverse engineering process used to acquire data from these two vehicles could be extended to other makes and models by

introducing a configuration and calibration feature in both the CAM and the bridge. This enhancement would enable users of known vehicle models to access the correct data automatically, while also simplifying the reverse engineering process for users of new or unsupported vehicles. As a result, PIENO would become effectively independent of both the OEM and fuel type.

5 Prophet Forecasting

Fuel price trends in Italy are made available by the Ministry of Enterprises and Made in Italy[3]. The dataset, available in CSV format, spans fuel price records from 2015 to 2025, divided by quarter. Each entry includes the region, fuel type, price, and corresponding date. The goal of the forecasting model is to help users anticipate whether fuel prices will increase or decrease in the coming days, allowing for more informed decisions on when to refuel.

We trained the model multiple times across different time ranges, ultimately finding that data from 2021 to 2024 produced the most reliable results: this period helps avoid distortions caused by the COVID-19 pandemic, resulting in a more accurate and representative predictive model. Unfortunately, the dataset suffers from inconsistent formatting:

- Mixed date formats (e.g., "01-05-2022" and "01/05/2022"),
- Mixed numeric encodings (e.g., "1.729" and "1,719"),
- Inconsistency in data reporting, that will be further explored in the model validation (Subsection 5.3).

5.1 Data Cleaning Pipeline

To prepare the dataset, we start from the government-provided CSV files. From those, we generate a SQLite [18] database that will then allow us to query the entire available data in a more powerful way.

In particular, we create a table with the columns "Region", "Fuel", "Price" and "Submit_date" and add all data to this table in this format.

In the end, to obtain the training data, what we had to do, starting, from this data, is to:

(1) Unify all date fields to `datetime` objects,
(2) Map ambiguous fuel names (e.g., "Benz" to "Benzina"),
(3) Group by `Region, Fuel, Date` and average daily prices.

5.2 Prophet Training

Prophet was selected for its robustness on irregular daily data [24]. We performed stratified sampling to create training segments of historical prices.

We train multiple Prophet model instances, each predicting fuel price trends for a specific fuel type in a specific region. This is possible thanks to the *group by* operation described in the previous section.

Given a dataframe `df` containing price trends for a specific fuel type in a specific region, sample training code is shown below:

```
df['ds'] = pd.to_datetime(df['Submit_date'])
df['y'] = df['Price']
model = Prophet()
model.fit(df[['ds', 'y']])
```

Listing 3: Training Meta Prophet on regional fuel data

[3]Accessible at https://www.mimit.gov.it/it/open-data/elenco-dataset/carburanti-archivio-prezzi

5.3 Prophet Validation

To validate the model performances, we split data at random time points and test the model's ability to forecast the trend for the following day. Predictions were evaluated as binary classification (*price up* vs *price down*) rather than regression accuracy. The average trend-matching accuracy was **75%**, outperforming a Random Forest [3] model (Table 2), which was used in previous iterations (PIENO V0).

A major challenge in validating the forecasting model lies in the inconsistency of the official data reporting. The open fuel price datasets provided by the Italian government may include retroactive updates, i.e., price values for past dates can be revised at any time after their initial publication. This characteristic introduces uncertainty into the historical record, making it difficult to guarantee a stable training and validation environment.

To ensure both reproducibility and temporal consistency, we adopted a conservative validation strategy: for each target date, only the dataset versions released up to the following day are considered. This restriction avoids the use of any future information that would not have been available at prediction time, thereby simulating a realistic deployment scenario.

While this approach likely underestimates the model's true performance, since it operates on potentially incomplete or outdated records, it preserves methodological integrity and ensures that our evaluation does not benefit from information leakage.

Model	Highest Accuracy (%)
Avg. accuracy of time series prediction	58
Random forest (PIENO V0)	67
Prophet (PIENO)	75

Table 2: Accuracy of different models.

To further assess the model's practical utility, we evaluated its performance on a real-world prediction task. As illustrated in Figure 10, Prophet successfully anticipated the direction of price changes for six out of seven consecutive days during the week of July 15–21, 2024. This result confirms the model's ability to generalize beyond the training period and provides encouraging evidence of its effectiveness in short-term fuel price forecasting under realistic constraints.

Metric	Score
Mean Square Error	0.00136
Avg. difference from ground truth	1.90%
Precision	80%
Recall	80%

Table 3: Comparison with ground truth.

6 Conclusion

This work presented PIENO, an end-to-end system for intelligent fuel itinerary optimization based on real-time vehicle telemetry and

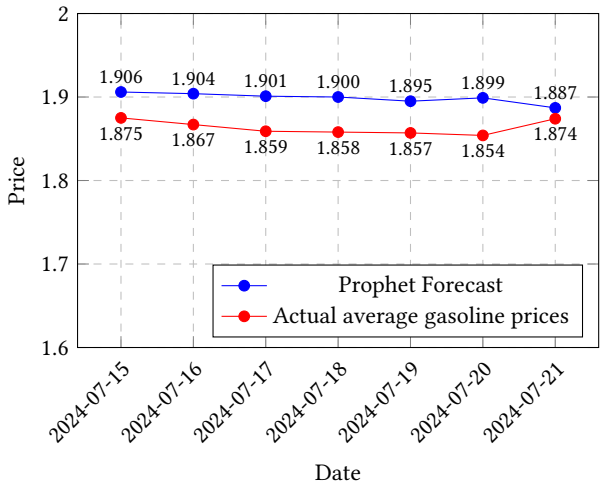

Figure 10: Prophet's forecast related to the actual average gasoline price.

economic forecasting. Through CAN Bus reverse engineering, we extracted fuel-level data from Citroën vehicles and demonstrated how this process can be extended to other car models. Our Prophet-based forecasting engine achieved high accuracy in predicting short-term fuel price trends, and the backend infrastructure demonstrated strong scalability under simulated loads.

PIENO exemplifies how modern software-defined vehicle paradigms, combined with open data and lightweight AI models, can bring tangible improvements to everyday driving. With its modular architecture, the system is designed for flexibility and extensibility, making it suitable for broader adoption across different vehicle makes and models. Future work will focus on expanding compatibility with more manufacturers, refining prediction models using more granular data, and integrating the system into mainstream navigation applications to promote the widespread adoption of intelligent mobility solutions.

References

[1] Jan Becker. 2022. Operating System for Software-defined Vehicles. *ATZelectronics worldwide* 17, 5 (01 May 2022), 40–45. https://doi.org/10.1007/s38314-022-0756-6

[2] Robert Bosch GmbH. 1991. *CAN Specification Version 2.0.* Stuttgart, Germany.

[3] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[4] Mark du Preez, Johan Zandin, et al. 2022. OBD II diagnostic interface pins and signals. https://pinoutguide.com/CarElectronics/car_obd2_pinout.shtml. Accessed: 2025-05-13.

[5] Espressif Systems. 2023. *ESP32 Series Datasheet.*

[6] Aidan Fuller, Zhong Fan, Charles Day, and Chris Barlow. 2020. Digital Twin: Enabling Technologies, Challenges and Open Research. *IEEE Access* 8 (2020), 108952–108971. https://doi.org/10.1109/ACCESS.2020.2998358

[7] Kayhan Zrar Ghafoor, Linghe Kong, Danda B. Rawat, Eghbal Hosseini, and Ali Safaa Sadiq. 2019. Quality of Service Aware Routing Protocol in Software-Defined Internet of Vehicles. *IEEE Internet of Things Journal* 6, 2 (2019), 2817–2828. https://doi.org/10.1109/JIOT.2018.2875482

[8] Francesco Giannini, Giuseppe Franzè, Francesco Pupo, and Giancarlo Fortino. 2023. A Sustainable Multi-Agent Routing Algorithm for Vehicle Platoons in Urban Networks. *IEEE Transactions on Intelligent Transportation Systems* 24, 12 (2023), 14830–14840. https://doi.org/10.1109/TITS.2023.3305463

[9] S Gowri, C Kanmani Pappa, T Tamilvizhi, Leema Nelson, and R Surendran. 2023. Intelligent Analysis on Frameworks for Mobile App Development. In *2023 5th International Conference on Smart Systems and Inventive Technology (ICSSIT).* 1506–1512. https://doi.org/10.1109/ICSSIT55814.2023.10060902

[10] Greenpeace Southeast Asia and Centre for Research on Energy and Clean Air (CREA). 2020. Aria tossica: il costo dei combustibili fossili. Media briefing.

[11] Ge Guo and Qiong Wang. 2019. Fuel-Efficient En Route Speed Planning and Tracking Control of Truck Platoons. *IEEE Transactions on Intelligent Transportation Systems* 20, 8 (2019), 3091–3103. https://doi.org/10.1109/TITS.2018.2872607

[12] Wei Huang, Lianghui Ding, De Meng, Jenq-Neng Hwang, Yiling Xu, and Wenjun Zhang. 2018. QoE-Based Resource Allocation for Heterogeneous Multi-Radio Communication in Software-Defined Vehicle Networks. *IEEE Access* 6 (2018), 3387–3399. https://doi.org/10.1109/ACCESS.2018.2800036

[13] Junyoung Kim and Changsun Ahn. 2020. Real-Time Speed Trajectory Planning for Minimum Fuel Consumption of a Ground Vehicle. *IEEE Transactions on Intelligent Transportation Systems* 21, 6 (2020), 2324–2338. https://doi.org/10.1109/TITS.2019.2917885

[14] Dennis Luxen and Christian Vetter. 2011. Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Chicago, Illinois) *(GIS '11).* Association for Computing Machinery, New York, NY, USA, 513–516. https://doi.org/10.1145/2093973.2094062

[15] Asad Waqar Malik, Anis U. Rahman, Arsalan Ahmad, and Max Mauro Dias Santos. 2022. Over-the-Air Software-Defined Vehicle Updates Using Federated Fog Environment. *IEEE Transactions on Network and Service Management* 19, 4 (2022), 5078–5089. https://doi.org/10.1109/TNSM.2022.3181027

[16] Microchip Inc. 2003. *Stand-Alone CAN Controller with SPI Interface.*

[17] Sangjin Nam, Hyogon Kim, and Sung-Gi Min. 2021. Simplified Stream Reservation Protocol Over Software-Defined Networks for In-Vehicle Time-Sensitive Networking. *IEEE Access* 9 (2021), 84700–84711. https://doi.org/10.1109/ACCESS.2021.3088288

[18] Michael Owens. 2006. *Introducing SQLite.* Apress, Berkeley, CA, 1–16. https://doi.org/10.1007/978-1-4302-0172-4_1

[19] Mohammad Ali Salahuddin, Ala Al-Fuqaha, and Mohsen Guizani. 2015. Software-Defined Networking for RSU Clouds in Support of the Internet of Vehicles. *IEEE Internet of Things Journal* 2, 2 (2015), 133–144. https://doi.org/10.1109/JIOT.2014.2368356

[20] Marco Savarese, Antonio De Blasi, Carmine Zaccagnino, and Carlo Augusto Grazia. 2024. P. I. E. N. O.–Petrol-Filling Itinerary Estimation aNd Optimization. *IEEE Access* 12 (2024), 158094–158102. https://doi.org/10.1109/ACCESS.2024.3483959

[21] Marco Savarese, Antonio De Blasi, and Carmine Zaccagnino. 2024. P.I.E.N.O. https://github.com/P-I-E-N-O.

[22] Gabriel Signoretti, Marianne Silva, Alexandre Dias, Ivanovitch Silva, Diego Silva, and Paolo Ferrari. 2019. Performance Evaluation of an Edge OBD-II Device for Industry 4.0. In *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0&IoT).* 432–437. https://doi.org/10.1109/METROI4.2019.8792885

[23] Haochen Sun, Fazhan Tao, Zhumu Fu, Aiyun Gao, and Longyin Jiao. 2023. Driving-Behavior-Aware Optimal Energy Management Strategy for Multi-Source Fuel Cell Hybrid Electric Vehicles Based on Adaptive Soft Deep-Reinforcement Learning. *IEEE Transactions on Intelligent Transportation Systems* 24, 4 (2023), 4127–4146. https://doi.org/10.1109/TITS.2022.3233564

[24] Sean J. Taylor and Benjamin Letham. 2018. Forecasting at Scale. *The American Statistician* 72, 1 (2018), 37–45. https://doi.org/10.1080/00031305.2017.1380080 arXiv:https://doi.org/10.1080/00031305.2017.1380080

[25] Panagiotis Typaldos, Ioannis Papamichail, and Markos Papageorgiou. 2020. Minimization of Fuel Consumption for Vehicle Trajectories. *IEEE Transactions on Intelligent Transportation Systems* 21, 4 (2020), 1716–1727. https://doi.org/10.1109/TITS.2020.2972770

[26] World Health Organization. 2022. Air Pollution. https://www.who.int/news-room/fact-sheets/detail/ambient-(outdoor)-air-quality-and-health. Accessed: 2024-07-20.

[27] Bin Xia, Fanyu Kong, Jun Zhou, Xiaosong Tang, and Hong Gong. 2020. A Delay-Tolerant Data Transmission Scheme for Internet of Vehicles Based on Software Defined Cloud-Fog Networks. *IEEE Access* 8 (2020), 65911–65922. https://doi.org/10.1109/ACCESS.2020.2983440

[28] Carmine Zaccagnino. 2020. *Programming Flutter.* The Pragmatic Bookshelf, Raleigh, NC.

Appendix A: CAN Reverse Engineering script

The following code was used to plot the different messages coming from the CAN IDs.

```
import matplotlib.pyplot as plt
from collections import defaultdict
import numpy as np

filename = "can_log.txt"
target_byte = "Byte4"
```

```

with open(filename, 'r') as file:
    lines = file.readlines()

records_by_id = defaultdict(list)
current_record = {}

for line in lines:
    line = line.strip()
    if line.startswith("ID:"):
        if current_record:
            records_by_id[current_record["ID"]].append(
      current_record)
            current_record = {}
        current_record["ID"] = int(line.split(":")[1])
    elif line.startswith("Byte"):
        key, value = line.split(":")
        current_record[key] = int(value)
    elif line.startswith("Timestamp:"):
        current_record["Timestamp"] = int(line.split(":")[1])

if current_record:
    records_by_id[current_record["ID"]].append(current_record)

```

```
candidate_ids = []

for can_id, records in records_by_id.items():
    byte_series = [r.get(target_byte, 0) for r in records]

    if np.var(byte_series) > 1.0 and np.max(byte_series) > 0:
        candidate_ids.append((can_id, byte_series))

for can_id, byte_series in candidate_ids:

    plt.figure(figsize=(8, 4))
    plt.plot(range(len(byte_series)), byte_series, label=f'{
      target_byte}')
    plt.title(f'CAN ID: {can_id} - {target_byte}')
    plt.xlabel('Message Index')
    plt.ylabel('Byte Value')
    plt.ylim(0, 80)
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Listing 4: Training Meta Prophet on

The value of target_byte is changed in order to analyze each byte of any ID at a time.