
V2X Communications

- Networks and Distributed Systems -

Project-Oriented Study in an External Organization

Gerardo Eugenio Martínez Ramos

Title:

V2X Communications

Theme:

Analysis of vehicle telematics

Project Period:

Fall Semester 2024

Participant:

Gerardo Eugenio Martínez Ramos

Supervisors:

Peter Falkesgaard Ørts

Israel Leyva Mayorga

Page Numbers: 27**Date of Completion:**

January 6, 2025

Abstract:

This report delves into the development, evaluation, and optimization of Vehicle-to-Everything (V2X) communication systems, focusing on AutoPi's telematics solutions. Through an in-depth exploration of network protocols, data logging, and communication performance, the study addresses challenges in real-time data transmission, GPS accuracy, and data latency. Practical contributions include establishing a robust logger configuration for heterogeneous vehicular environments and enhancing communication reliability. Insights are provided to improve user experiences and IoT applications in intelligent transportation systems, with future work extending to CO2 emissions reporting and other areas of opportunity.

Contents

1	Introduction	2
1.1	AutoPi devices	2
1.2	Main responsibilities	3
1.3	AutoPi Mini	6
2	State of the art	9
2.1	On Board Diagnostics II	9
2.2	CAN Protocol	10
2.3	MQTT	10
2.4	IoT devices in V2X ecosystems	11
3	Methods	12
3.1	Requirements analysis	12
3.2	Evaluation metrics	16
3.3	Mini device tests	17
4	Results	18
4.1	Data logger accuracy	18
4.2	Data latency	18
5	Conclusion	22
	Bibliography	23
A	Worksheets	24
A.1	September	24
A.2	October	25
A.3	November	26
A.4	December	27

Preface

This report describes in detail the 9th semester project done at AutoPi, an external organization based in Aalborg. This was done as part of the Computer Engineering curriculum at Aalborg University.

This project has been a pivotal part of my academic and professional journey, providing a unique opportunity to participate in solving a real-world problem. It allowed me to deepen my understanding of embedded systems, Internet-of-Things (IoT) technologies, and the integration of software and hardware. Furthermore, this experience has significantly improved my problem-solving and teamwork skills, preparing me for future challenges in the field of computer engineering.

Working with AutoPi was a very rewarding experience. I express my sincere gratitude to the AutoPi team for the opportunity to work on this project and for their great support throughout the process. The collaborative nature of the project gave me a great insight into the professional work culture, especially within the context of an international organization. This time also offered a perspective on how engineering challenges are approached and solved in a different country, broadening my understanding of global engineering practices.

In this report, the reference sources will appear according to the numeric reference source method, which means that the first source will be assigned [1] and the second [2]. If multiple sources are used throughout a paragraph, the sources will be listed at the end of the paragraph [1,2].

Abbreviations that are not commonly known will be written in full the first time they are mentioned, followed by the abbreviation in parentheses. For later uses, the abbreviation will be used explicitly.

Aalborg University, January 6, 2025

Acronyms

API Application Programming Interface. 4, 17

CAN Controller Area Network. 3, 5, 9, 13, 22, 24, 25

CRUD Create, Read, Update and Delete operations. 25–27

DTCs Diagnostic Trouble Codes. 2

EV Electrical Vehicle. 17–22, 25

FURPS Functionality, Usability, Reliability, Performance, and Supportability. 16, 22

GPS Global Positioning System. 20, 21

ICE Internal Combustion Engine. 22

IoT Internet-of-Things. iii, 7, 11, 22

JSON JavaScript Object Notation. 17

MQTT Message Queueing Telemetry Transport. 9, 12, 13, 17, 19, 22

OBD2 On Board Diagnostics II. 2, 9

QA Quality Assurance. 17, 18, 20

QoS Quality of Service. 22

UI User Interface. 4, 12

V2X Vehicle-to-Everything. 2, 9, 11

Introduction

AutoPi.io has delivered IoT telematics solutions for more than 9 years, consisting of a unique combination of customized hardware and software, namely AutoPi devices and AutoPi Cloud. Both allow their customers to manage their assets and address their vehicle-related requirements, which span various use cases such as car leasing, supply chain logistics, public transportation, among several others [1]. To accomplish the different goals each customer may have, AutoPi has designed different types of devices that can be seen from the perspective of the Vehicle-to-Everything (V2X) Communication field. Most of the various topics in V2X fall within the scope of networks and distributed systems.

During the internship at AutoPi, I collaborated as a software developer by testing and maintaining existing features, and participating in the design of new ones as well. On the hardware side I contributed by updating the software in charge of initial configuration for devices, performing tests, and providing support for some customers. My experience at the organization was deeply valuable since it gave me deeper knowledge on how edge solutions are designed, implemented, and the subsequent steps in the lifecycle of these types of products which play an important role in today's world. The report is structured as follows:

- The remainder of the introduction provides further details about the organization's devices and an overview of all the tasks done during the internship. The section ends by describing some of the challenges faced and introducing the central problem studied during this time.
- Section 2 presents the state-of-the-art methods to provide a better understanding of the V2X context in which the central problem revolves.
- The Methods section elaborates on the AutoPi Mini device, since most of my tasks were related to enhancing its performance. The central problem approached and the steps taken to solve it.
- The last section shows the obtained results, a discussion, and conclusion remarks.

1.1 AutoPi devices

The devices were designed to communicate with vehicles compliant with the On Board Diagnostics II (OBD2) standard, which has been in use in the US since 1996 and mandatory in Europe in 2001, with many more regions adopting it since then [2]. This standardized system monitors and reports information about the performance and health of a vehicle's engine and other of its components.

Once attached to a vehicle, the device reports the data to the AutoPi Cloud to securely collect, store, and analyze data. Key features include real-time vehicle diagnostics, GPS tracking, remote control, automated alerts, and data recording. Some parameters it monitors are engine temperature, Diagnostic Trouble Codes (DTCs), fuel pressure, intake air temperature, and engine oil temperature [3]. Currently there are three types of devices:

- AutoPi TMU CM4, a robust telematics unit that uses a RaspberryPi compute module [4], [5]. This device is aimed at users who wish to extend functionality by implementing custom code on top of the compute module.
- Autopi CAN FD Pro, for users who have a deep understanding of the Controller Area Network (CAN) protocol and have very specific needs. [6]
- AutoPi Mini, solves most telematics use cases. [3]

Although each device has a different purpose, they all share a common ground. I participated in activities related to each of them.

1.2 Main responsibilities

On the development side, management groups pending activities for the cloud platform, devices, and customer support in different categories or projects. These projects are made up of many tasks such as implementing new features, bug fixing, documentation or enhancements. Every two weeks the tasks are prioritized and put in one of the following categories:

- Hardware: anything related to a device's physical components.
- Core: source code for device functionalities, it interacts directly with the vehicle.
- Backend: source code for the backend component of AutoPi Cloud. It is the link between AutoPi Cloud and AutoPi devices.
- Frontend: user interface for AutoPi Cloud and other internal applications.

Documentation and customer support were also recurring tasks which could fall in any of these categories. The next subsections give an overview of all the tasks done during the internship grouped by project. The annex A contains a detailed summary for the activities done in chronological order.

1.2.1 Hardware and Core

Squid3 improvements

Every TMU device must go through a *checkout process* to ensure it works properly and all configurations are correctly set. Typically, this process is carried out simultaneously for multiple devices at a checkout station. As a result, it is critical for *checkout operators* an intuitive user interface that displays the different states a device goes during the process, in order to detect issues. To simplify the operator tasks, an internal application was developed by AutoPi, namely *Squid3*. In general, this application performs the following steps:

1. Confirm a correct voltage is applied to the device and check the hardware components.
2. Load operating system on the TMU device.
3. Run automated tests on the device.
4. Inform success or error messages to the operator.
5. Register the device on the cloud platform.

The devices are connected to a network switch and a process with a web interface enables the operator to interact with the device. My contribution in this project consisted in improving the User Interface (UI) to visualize in what state each device is during the process. Another enhancement was automating the deployment of this UI in order to enable the web application in different checkout stations.



Figure 1.1: Squid3 improvements progress in a test environment

1.2.2 Backend and Frontend

SIM provider API integration

Most devices come with a SIM card that is managed by AutoPi, and handling all connections with the network operator can be cumbersome and repetitive. I was given the task of making an Application Programming Interface (API) integration with the SIM provider. The purpose of this is two-fold first, to give the users more information on how much data their device was consuming, and, on the other hand, to simplify management tasks and allow alerts implementations for events such as SIM card change, excessive data usage, and SIM status (active or disabled).

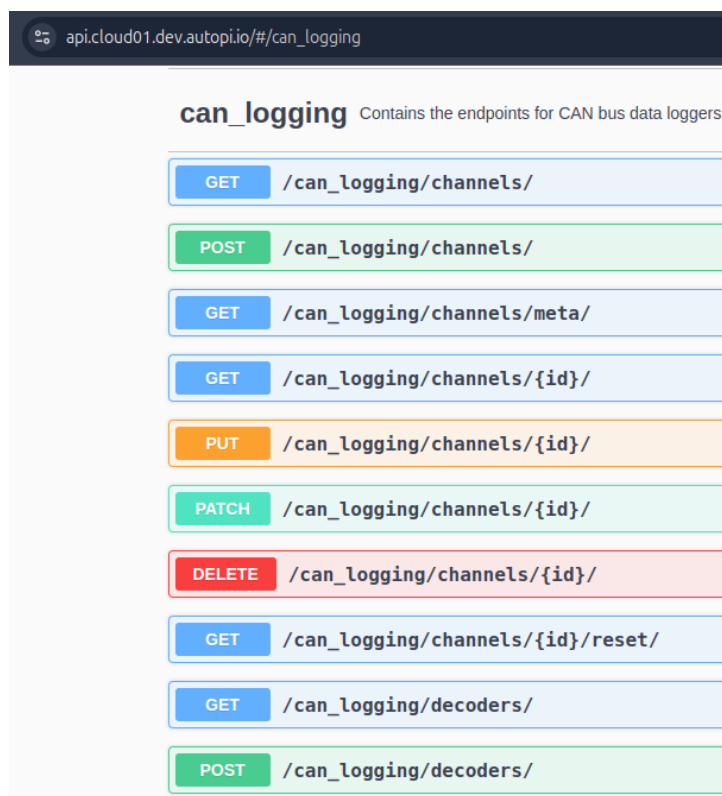


Figure 1.3: API specification backend endpoints in test environment

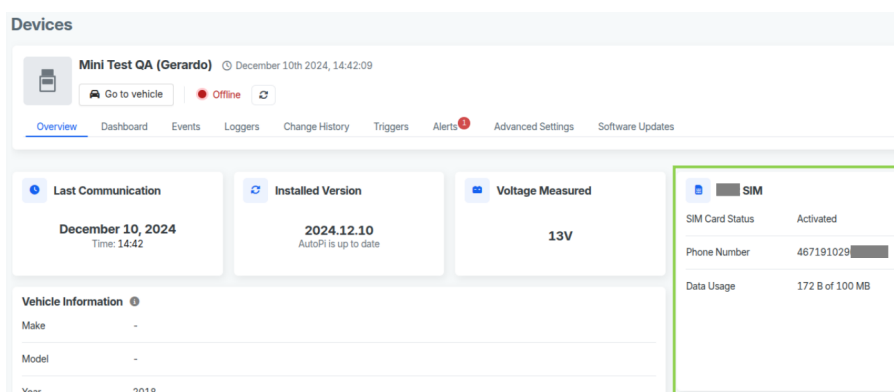


Figure 1.2: SIM integration progress in test environment

CAN logging endpoints

The newest AutoPi device is targeted at specialized users interested in events communicated on the CAN bus for all types of vehicles. Logging this information is complicated due to several factors like proprietary CAN configurations, large amount of traffic on the bus, memory and connectivity constraints, etc.

Basically, core processes collect and store CAN bus events that can then be read by AutoPi Cloud. My contribution to this project was implementing the back-end endpoints that provide data for the front-end.

1.2.3 Documentation and customer support

Documentation was a recurrent activity, while technical support, on the other hand, was more sporadic depending on incoming requests from customers. In general, the tasks performed can be summarized as follows.

- Updated technical manuals for users who needed to modify their devices. In particular, some are required to improve the network signal in areas with low connectivity by attaching an external antenna.
- Documented internal processes such as device configuration, how to set up an environment for local testing, among others.
- Scripting to address customer requirements or proof of concepts. For instance, a customer needed to download their vehicle data in a particular format to train a machine learning model that could identify issues with a vehicle's battery.

All the described tasks required a team effort where my co-workers at AutoPi guided me through all of the processes. As a result, I have gained a solid understanding of how the devices work and how to solve different types of issue on the platform. Of all the ongoing projects during my internship, the AutoPi Mini stood out because it presented many engineering challenges and involved all the tasks described previously.

1.3 AutoPi Mini

The primary goal of this device is to save time for the user by offering good functionality with minimal configuration right out of the box. To guarantee this ease of use, thorough testing of different settings and initial conditions is essential. For example, the device must perform reliably in different vehicle types, electric or powered by an internal combustion engine. To accomplish this, the current architecture is designed to support regular firmware updates and establish communication channels for reporting the vehicle's data readings.

1.3.1 System overview

Similarly to the TMU device, each Mini device goes through a provisioning process at an assembly plant. After completion of this process, the device is shipped to a customer and connected to a vehicle, where it reads different signals and transmits the relevant data to the AutoPi Cloud. This is depicted in the diagram 1.4, each box represents a participating entity in the network, and the text next to the connecting edges represents the protocol used between entities to communicate.

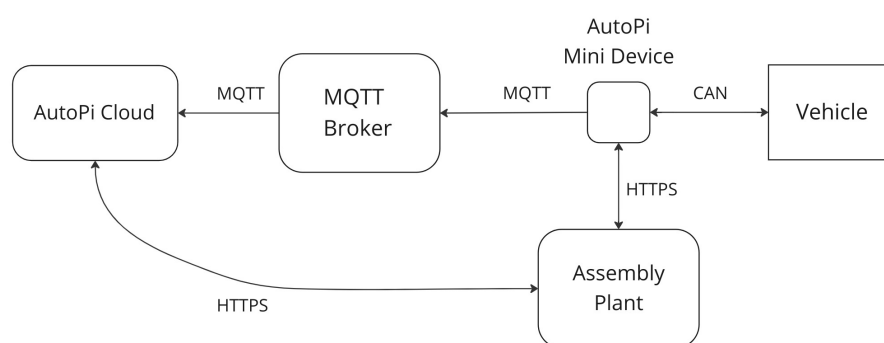


Figure 1.4: Communication network for an AutoPi Mini

The protocols and entities will be described further in the next sections of the report. Another important aspect of working with IoT devices is the ability to test them without having to set up all the entities in the network. To do it, a local debugging environment is needed, whether the purpose is to perform quality assurance inspection or to reproduce specific issues reported by customers.

1.3.2 Debugging devices

Although devices share similar hardware components, their configuration processes differ significantly. For example, interacting with these components may require drivers that are compatible only with specific operating systems or rely on proprietary tools. Once the appropriate debugging environment was set up, the next step was to test new device features or investigate a specific issue reported by a customer. Figure 1.5 shows an example of how to send signals from an emulator to an AutoPi device.

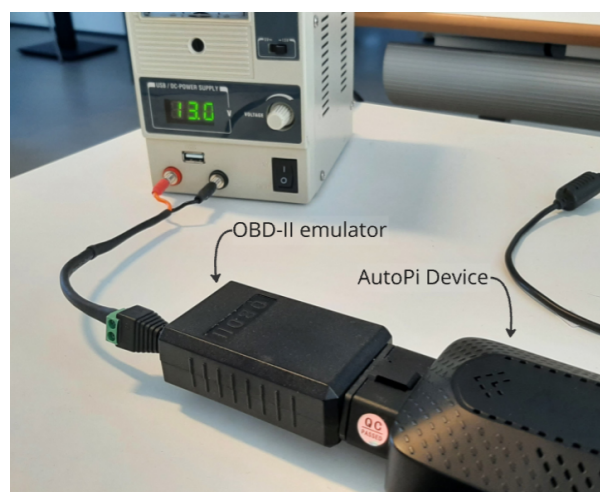


Figure 1.5: Setup for device testing

After tests were performed with an emulator, connecting the device to a vehicle is necessary to verify that the rest of the entities in the system work as expected. The following list describes some of the challenges faced while working with the AutoPi Mini:

1. Configuration

- (a) From factory settings, the device can detect more than 200 different signals from a vehicle by means of a *logger*. However, not all loggers work on different vehicle makes.

2. Debugging and Testing

- (a) There is no standard way to emulate data signals from electrical or internal combustion engine vehicles. The codes used by different vehicle manufacturers need to be reversed engineered. This makes debugging a convoluted process, and figuring out a general solution is practically impossible due to all the different possible configurations and customer specific needs.

3. Communication with other systems

- (a) As stated above, vehicle data signals are not standardized between manufacturers, which can cause data loss when another system is parsing it.

- (b) An unreliable network connection or a faulty configuration can result in a bad experience for the user, as the data is expected to be available as close to real time as possible.

The main goal is to provide users with the best experience when using an AutoPi Mini, and to achieve that, the previous challenges can be translated into solving the next two main problems.

Central problems:

1. Determine which configurations work best on different vehicles. Or, alternatively, what configuration provides the most information from a vehicle?
2. How to identify and mitigate bottlenecks in the communication network?

Before exploring the specifics of these problems, it is of great value to establish more terminology and context to gain a clearer perspective for identifying potential solutions. The next chapter outlines current trends and context, serving as a foundation for the Methods section, where the problems will be tackled with a technical approach.

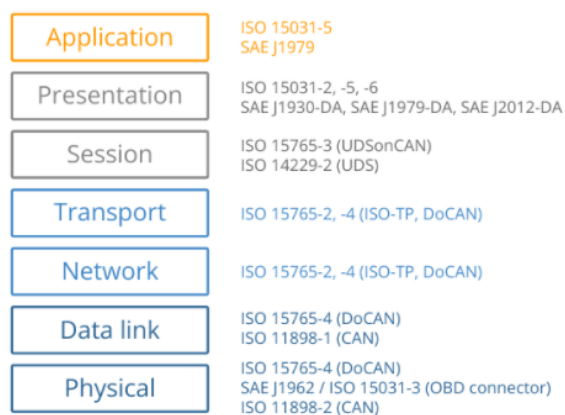
State of the art

V2X communication represents a transformative technology in the automotive and transportation sectors, enabling vehicles to interact with other vehicles (V2V), infrastructure (V2I), pedestrians (V2P), and the network (V2N). As a cornerstone of intelligent transportation systems (ITS), V2X leverages a combination of advanced protocols, communication models, and IoT-enabled devices to ensure seamless, real-time data exchange. This section delves into the foundational components and technologies that underpin V2X, structured around the OSI model to emphasize the layered approach in communication design. Explores the role of OBD2 in providing essential vehicle data at the physical and data link layers, the CAN bus and protocol at the network layer, and the Message Queueing Telemetry Transport (MQTT) protocol at the application layer for lightweight and efficient messaging. The last subsection provides a list of current challenges regarding IoT devices in V2X ecosystems. This general overview provides a comprehensive understanding of the state of the art in V2X communications.

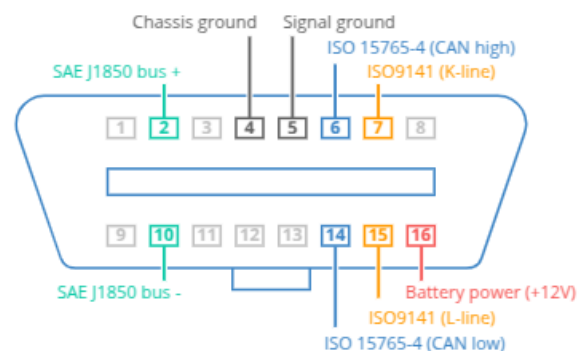
2.1 On Board Diagnostics II

OBD2 is a standardized system used in vehicles for self-diagnosis and reporting of engine and other system issues. Mandated in the U.S. since 1996, OBD-II provides real-time data and standardized diagnostic trouble codes (DTCs) that technicians can use to identify and address vehicle problems. The system interfaces with a vehicle's engine control unit (ECU) and monitors components such as the engine, emissions systems, and transmission. It also allows external devices, such as scan tools, to retrieve data through a standardized connector, making diagnostics and maintenance more efficient. [7].

7 layer OSI model | **OBD2** (on CAN) standards



(a) OSI model.



(b) OBD connector.

Figure 2.1: OBD diagrams taken from [8]

2.2 CAN Protocol

The Controller Area Network (CAN) protocol is a robust communication protocol widely used in embedded systems, particularly in automotive, industrial, and IoT applications. It facilitates efficient and reliable data exchange between microcontrollers and devices without needing a host computer. A CAN bus is the physical layer implementation of the protocol, where multiple nodes (devices) are connected through a two-wire twisted pair, enabling communication. [9] Some of its key features are the following:

- Data transmission: CAN operates at speeds up to 1 Mbps, supporting real-time communication.
- Error detection: The protocol has built-in error detection and handling mechanisms.
- Broadcast messaging: CAN uses a message-based protocol, where all nodes receive all messages but process only those of interest.
- Scalability: Multiple devices can be connected to a single CAN bus with minimal impact on performance.

2.3 MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish/subscribe messaging protocol designed for low-bandwidth, high-latency, and unreliable networks. Initially developed by IBM in the late 1990s, MQTT is widely used in IoT (Internet of Things) applications due to its simplicity and efficiency. [10, 11, 12]

Key Features

- Publish/Subscribe Model: Enables decoupled communication between devices.
- Lightweight Protocol: Optimized for constrained devices and networks.
- QoS Levels: Provides three levels of Quality of Service for message delivery:
 - Level 0: At most once (fire and forget).
 - Level 1: At least once (acknowledged delivery).
 - Level 2: Exactly once (guaranteed delivery).
- Retained Messages: Stores the last known message for new subscribers.
- Last Will and Testament (LWT): Notifies subscribers about unexpected disconnections.

Structure

- Broker: Central server responsible for routing messages.
- Clients: Devices or applications that publish or subscribe to topics.

Current Implementations

- Mosquitto: An open-source MQTT broker written in C.
- HiveMQ: A commercial-grade MQTT broker for enterprise IoT use cases.
- EMQX: A scalable and open-source MQTT broker supporting millions of connections.

- AWS IoT Core: A managed cloud-based implementation of MQTT.

2.4 IoT devices in V2X ecosystems

IoT devices play a crucial role in V2X communication by enabling real-time interaction between vehicles and their environment. They collect, process, and transmit data essential for ensuring safety, efficiency, and scalability in intelligent transportation systems. The following list mentions some of the current challenges in this field.

- Power consumption in IoT devices
- Security and privacy concerns
- Reliability in High-speed and dense environments
- Cost and infrastructure limitations
- Context awareness and decision making
- Standardization and regulatory concerns
- Heterogeneity of IoT devices
- Limited range and coverage

In particular, these three topics are the most related to this project:

1. Latency and real-time processing
2. Network congestion scalability
3. Localization and synchronization issues

Methods

The purpose of this section is to give a precise description of the central problems presented at the end of Section 1.3. First, the communication network presented in 1.4 is re-examined with more thorough descriptions of the events occurring between entities. From there, a requirement analysis is performed in order to establish performance metrics to evaluate different device configurations. The last subsection describes the devices, environments, and general steps used for testing.

3.1 Requirements analysis

The communication flow from Mini to the cloud platform is as follows.

1. The device transmits and receives CAN signals to and from the vehicle.
2. The device sends stored data to the MQTT broker.
3. The broker forwards notifications to the Device Agent, which parses and stores incoming data to display in the UI.

When a new configuration, feature, or fix needs to be published to Mini devices:

1. The new settings are sent from the Config Agent towards the assembly plant, and any firmware updates are also retrieved to update the information on the cloud platform accordingly.
2. The configurations are pushed to specific devices using the assembly plant's REST API. Not all devices are updated at the same time (or with the same firmware) due to the variety in customer requirements.
3. The AutoPi Mini is updated, and it confirms the success or failure back through the assembly plant's REST API.

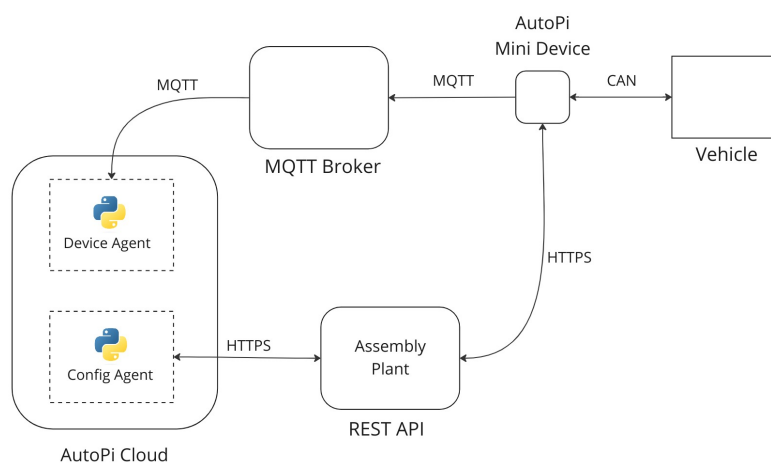


Figure 3.1: General architecture for Mini device

Currently, a Mini device can detect more than 200 CAN signals, each of which can be acquired and transmitted later to the message broker using different time intervals. In addition, the transmission can be done in batches or individually. The process in charge of handling this is called a *data logger*. Moreover, there is a very large set of vehicle and machinery technicalities to consider, for instance, logging events such as detecting if the engine was turned on or if the acceleration is above a certain value. In short, the search space for the ideal logger configuration is very large and difficult to explore, as the number of test devices is also limited.

The next figures give an idea of how different loggers can be enabled in a Mini device, an example of the data that is received by the MQTT broker, and how to adjust the frequency for each logger.

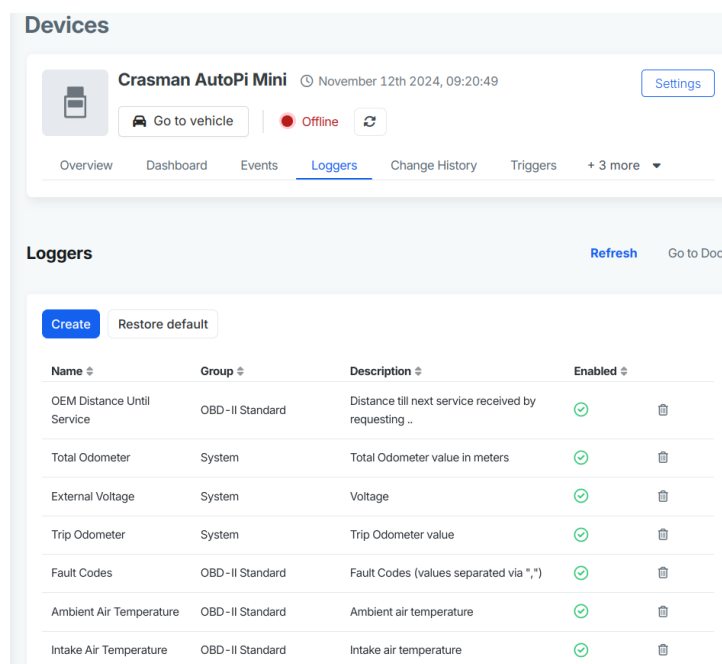


Figure 3.2: Selecting vehicle loggers

```
{
  "state": {
    "reported": {
      "ts": 1700813013000, "pr": 0,
      "latlng": "56.842227,9.894260",
      "alt": 0, "ang": 0, "sat": 0,
      "sp": 0, "evt": 0, "66": 13011
    }
  }
}
```

Figure 3.3: JSON payload example

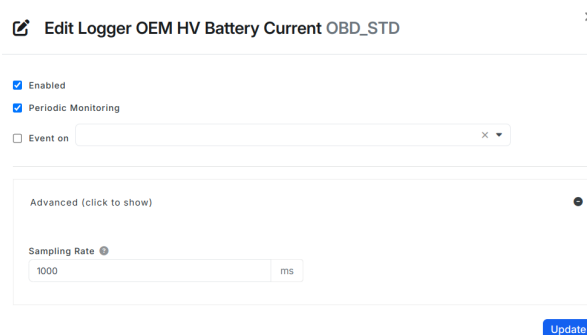


Figure 3.4: Specifying logger frequency and events in the cloud platform

As stated in the end of the introduction 1, the main objectives are to find the best logger configuration and to reduce bottlenecks in the communication network. These have an impact in different aspects, such as the location precision in vehicle routes or the time it takes for the device to send all the data it has collected. Examples of these problems were noticed early while investigating the loggers and are shown in the figures 3.5 and 3.6.

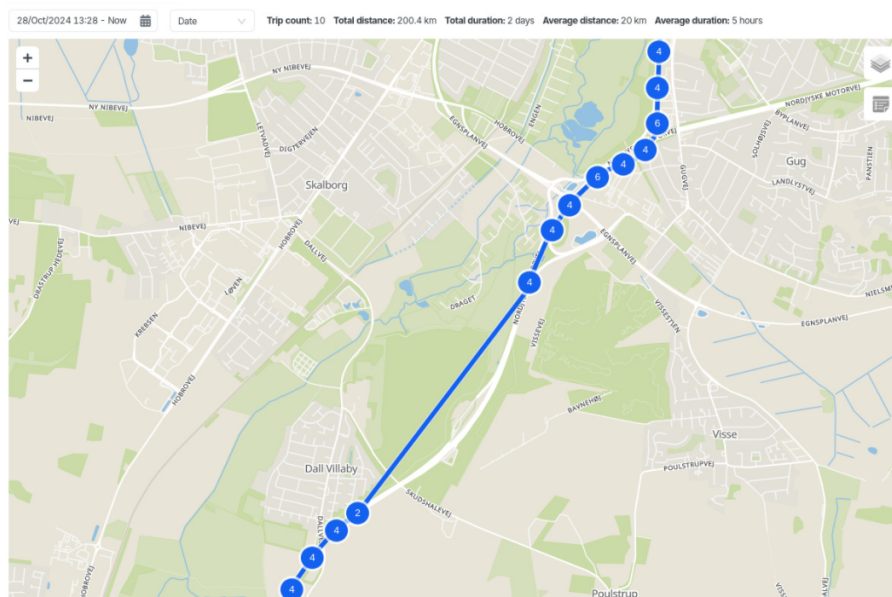


Figure 3.5: Example of location precision issues.

In figure 3.5 the route displayed on a map for a vehicle trip shows a sequence of blue points for every position reported by the Mini device. Notice that there is a visible gap between two of these points, which gives the appearance of the vehicle going off the main road. Although this could be corrected in cases where it is “evident” on which road the vehicle was on, it is very difficult to correct the route and infer the position of missing points if the route has many possible paths.

On the other hand, figure 3.6 shows a data transmission bottleneck for a device on November 13th. Bottlenecks can be characterized by *latency*, which in this case is defined as the number of seconds it takes for the data to be received on the cloud platform after being collected by the Mini device in the vehicle. The plot shows how the latency remains below 2,000 seconds (or 33 minutes) from 0 through hour 16, and then at hour 16 there is a very considerable spike. These latency values are very high and can significantly affect the user experience.

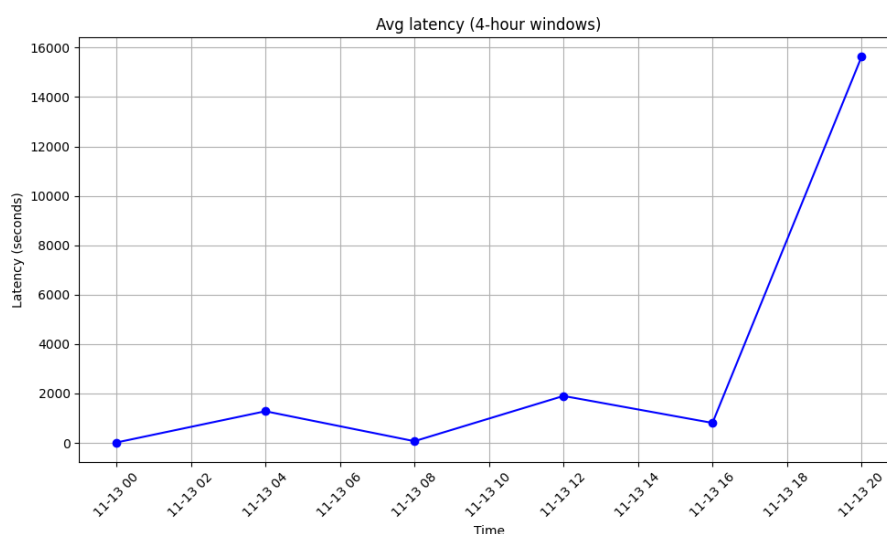


Figure 3.6: Example of latency issues.

Next, in Figure 3.7, a set of plots illustrates the values collected for different data loggers. Each plot shows on the Y-axis the range of values collected during a specific time interval. Notice that there are several loggers that exhibit oversampling.

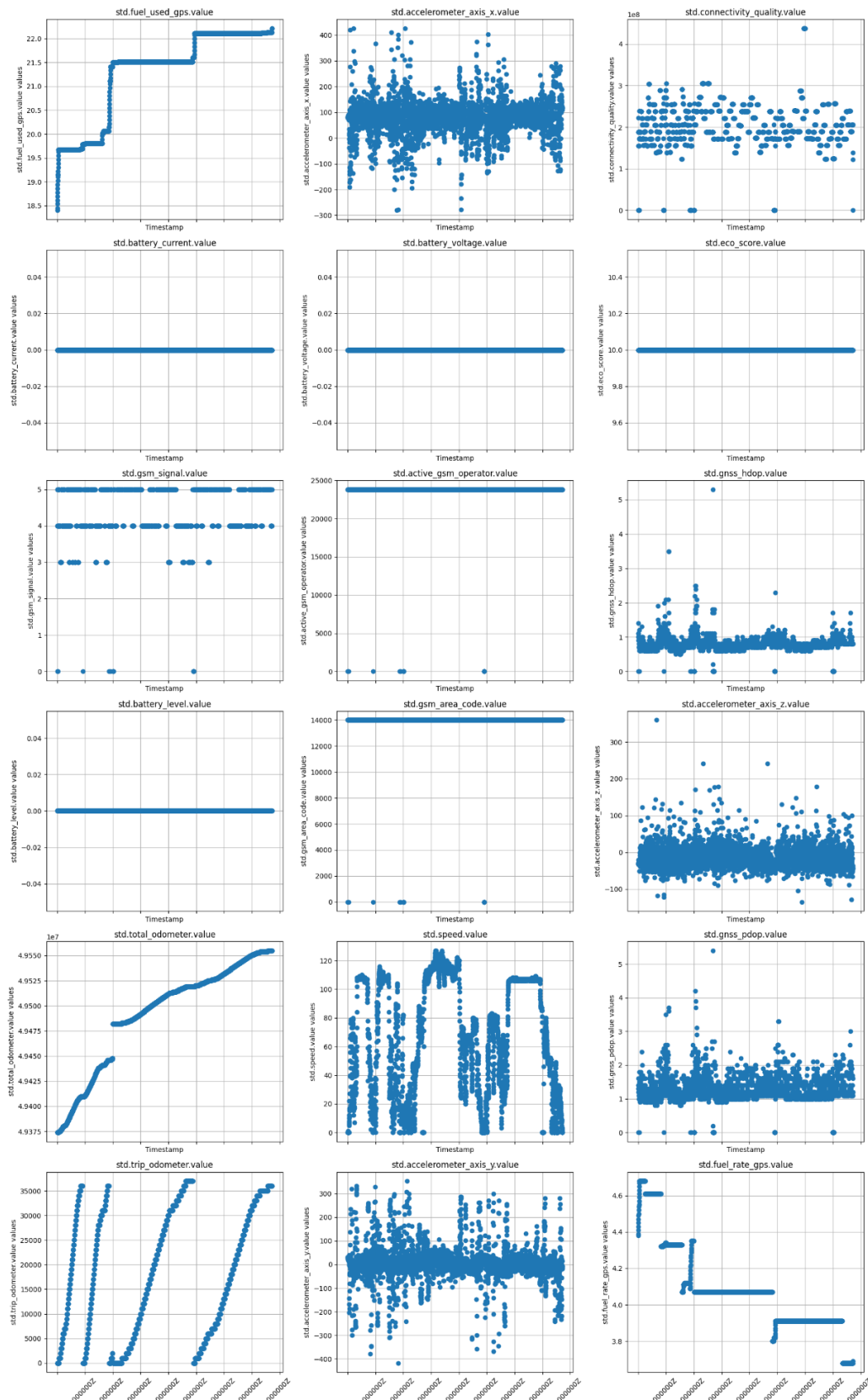


Figure 3.7: Standard parameters example

The main objectives have different criteria that can be evaluated; to display them in a concise manner, the Functionality, Usability, Reliability, Performance, and Supportability (FURPS) model was used to generate the following table.

FURPS Aspect	Logger Accuracy	Data Latency	Main Entities
Functional	Correct values stored and minimal under-sampling/oversampling.	Near-real-time data transmission to the cloud.	Vehicle sensors, data logger, communication modules, cloud platform.
Usability	Intuitive interfaces to configure loggers.	Real-time dashboards for latency monitoring.	User interface (UI), analytics tools.
Reliability	Fault-tolerant logging; data integrity verification.	Stable transmission with fallback mechanisms.	Data storage systems, network infrastructure.
Performance	Consistent sampling rates; high fidelity to source data.	150s transmission latency for collected data.	Communication protocols (e.g., 5G, DSRC), cloud processing units
Supportability	Diagnostic tools; adaptable to multiple protocols.	Extensible system; robust troubleshooting tools.	Developers, maintenance teams, documentation systems.

Table 3.1: FURPS Analysis for Logger Accuracy and Data Latency

Due to time and resource constraints, **only functional and performance aspects** were considered in the scope of this report. That, and also the other FURPS aspects, imply modifications to more entities of the system besides the AutoPi Mini, such as the cloud platform or the network infrastructure.

3.2 Evaluation metrics

To evaluate functionality and performance improvements, the following metrics are defined.

1. **Logger accuracy:** detect the highest number of working data loggers that provide:
 - (a) **Functionality:** collected values from data loggers make sense; this means that they are in an appropriate range according to their description (e.g. temperature, speed). In addition, there should be minimal oversampling for each data logger.
 - (b) **Performance:** Optimal sampling rates, high fidelity to source data.
2. **Data latency:** The average latency across all data loggers within a device will be measured, with the following considerations:
 - (a) **Functionality:** Near-real-time data transmission to the cloud.
 - (b) **Performance:** ideally transmission latency below 200s.

3.3 Mini device tests

The data collected for this report came from three Mini devices, namely device A, device B and device C, each of them attached to a different type of Electrical Vehicle (EV). On the cloud platform side, a test environment and a production environment were used. This means that there were two separate AutoPi Cloud entities receiving the data from the MQTT broker (as described in 3.1). Devices A and B were registered in the test or Quality Assurance (QA) environment and were connected to assets owned by AutoPi, while device C was registered in production since its owner is a customer participating in an early release of the Mini device.

The described setup poses some limitations, particularly for device C since the production environment is also used by many AutoPi customers. On another note, physical interaction was easy to achieve for devices A and B when it was necessary (e.g. to inspect device logs), but for device C this was not the case since the customer was located in another country. For this reason, each device served a different purpose:

1. Device A was used to investigate the effect of varying the sampling frequency for each logger, and to detect which loggers returned values with an EV.
2. Device B was used mainly to investigate GPS precision (by other members of the team) and to replicate *working* or *faulty* configurations found in A.
3. Device C was focused on investigating the acquisition of values related to the vehicle's battery. During the tests, device C was tested in two different vehicles.

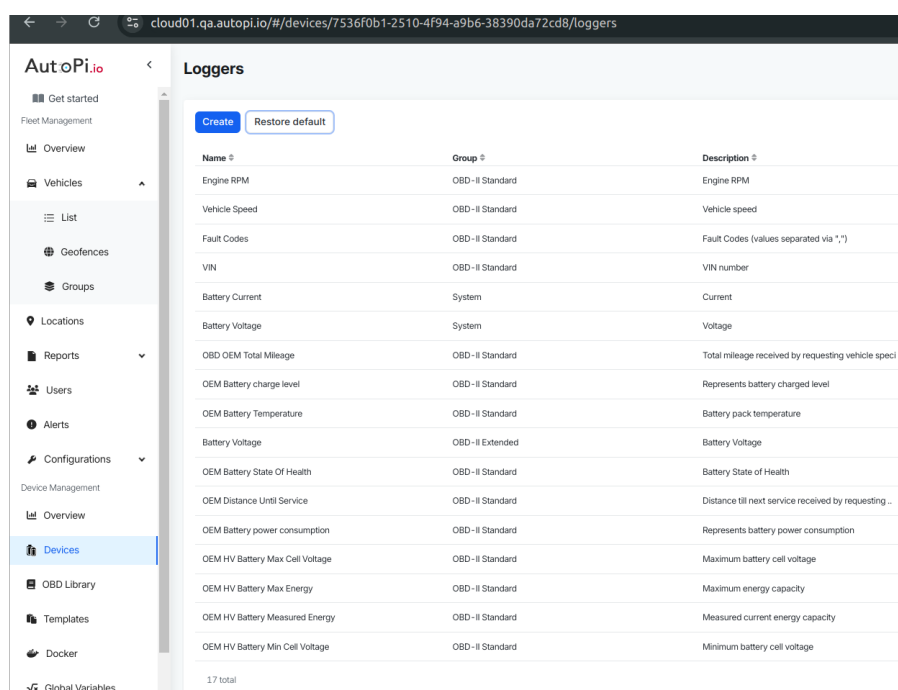
It is important to note that the trips done by the vehicles associated to devices A and B were mostly periodic, i.e., the routes remained fairly constant day after day with little variation. Device C had no routes repeating at all; this is important because the configuration of a logger reacts to the behavior of a vehicle. Another observation is that the QA environment has the most recent code version of the AutoPi Cloud platform, which can offer some advantages over the production environment. All functionalities of the platform are tested in QA before releasing features or corrections into production. In short, production can be more stable, but new features or corrections are available sooner in QA.

In general, once the configuration was modified for any device, it was necessary to wait at least one to two days to see the effects on the logger accuracy or data latency. Getting this information was not an easy task because the telematics data is stored in JavaScript Object Notation (JSON) format, which makes parsing data between entities easy, but can generate some complications when trying to perform queries or aggregation operations. Basically, to investigate the effect that different configurations had on devices and the system, first the "raw" JSON data were retrieved in batches from the cloud storage due to the API request rate limits that prevent the infrastructure servers from being stressed. From there, the data was transformed to CSV format, and then an exploratory analysis was done.

Results

4.1 Data logger accuracy

Of the complete set of data loggers, only 59 of them were detected to return values for any EV considered in the tests. The only way to verify this was to enable the data logger (as shown in 3.2) and wait at least a couple of days to be *almost* certain that the value is not actually reported by the vehicle. Once a steady set of loggers were reporting values, they were analyzed to ensure that their range was correct by performing multiple exploratory data analysis as described in Figure 3.7.



Name	Group	Description
Engine RPM	OBD-II Standard	Engine RPM
Vehicle Speed	OBD-II Standard	Vehicle speed
Fault Codes	OBD-II Standard	Fault Codes (values separated via ",")
VIN	OBD-II Standard	VIN number
Battery Current	System	Current
Battery Voltage	System	Voltage
OBD OEM Total Mileage	OBD-II Standard	Total mileage received by requesting vehicle speci...
OEM Battery charge level	OBD-II Standard	Represents battery charged level
OEM Battery Temperature	OBD-II Standard	Battery pack temperature
Battery Voltage	OBD-II Extended	Battery Voltage
OEM Battery State Of Health	OBD-II Standard	Battery State of Health
OEM Distance Until Service	OBD-II Standard	Distance till next service received by requesting ..
OEM Battery power consumption	OBD-II Standard	Represents battery power consumption
OEM HV Battery Max Cell Voltage	OBD-II Standard	Maximum battery cell voltage
OEM HV Battery Max Energy	OBD-II Standard	Maximum energy capacity
OEM HV Battery Measured Energy	OBD-II Standard	Measured current energy capacity
OEM HV Battery Min Cell Voltage	OBD-II Standard	Minimum battery cell voltage

17 total

Figure 4.1: Final set of 17 loggers selected

Figure 4.1 displays the default configuration currently used by Mini devices, which was selected in large part after the tests performed. Varying the number of enabled loggers in a device had no side effects on logger accuracy or data latency. However, the factor that produced the worst effect on the device and on the cloud platform was increasing the logger frequency, as shown in the next subsection.

4.2 Data latency

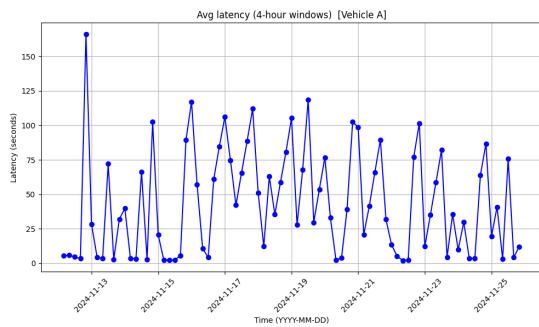
Device A

This device transmitted data to the QA environment, which made it more comfortable to try different configurations, as any errors could be fixed faster. As seen in figure 3.3, each data logger reports the time the data is collected (along with its respective values) and then the

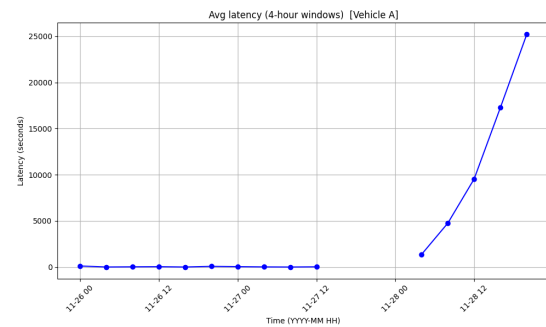
cloud platform registers the timestamp when the payload was stored. The difference between these timestamps is defined as latency. So then, the average latency is calculated for all the payloads received and resampled in windows of 4 hours to improve the visualization, which correspond to the grid of plots in figure 4.2. These plots show how increasing the data logger frequency considerably affected latency.

The first plot 4.2a, shows the initial configuration that even if it provided low latency values (almost all below 150s) the data loggers did not include specific EV parameters (e.g. battery state of health). Next, in figure 4.2b the configuration was modified around November 27 to include the desired parameters, and the effects are visible starting from November 28. The data gap in this plot is due to the device being offline during manual configuration.

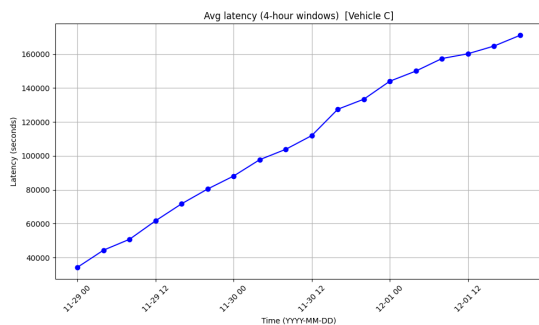
After November 28, latency follows an increasing function pattern (4.2c) until December 3 when it starts to decrease and stabilizes again with values under 150 s until December 9 (4.2d). The reason why the latency was so high, reaching delays of up to 2 days, was the oversampling caused by the high frequency. In the beginning, I thought it could be the MQTT broker that could not keep up with the high volume of data collected, a rough estimate for device A indicates there are around 11 million records from November 12 to December 1. However, this possibility was discarded since only device A was experiencing this delay and it did not occur with other devices in the same environment.



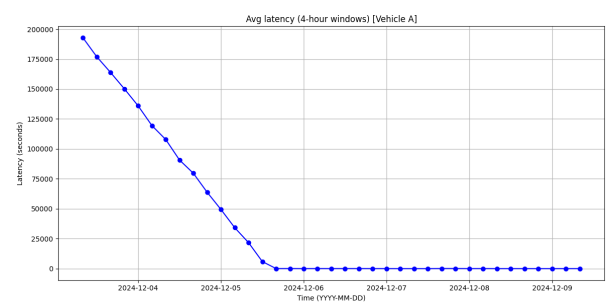
(a) November 12 to November 25



(b) November 26 to November 28



(c) November 29 to December 2



(d) December 3 to December 9

Figure 4.2: Latency results for vehicle A

After some guidance from my colleagues, we realized that the problem was that the Mini device does not support increasing the sampling frequency to the limit of 1 value every second (1000ms as shown in 3.4) for more than 4 loggers at the same time. Doing so started to saturate the device's memory with pending messages to be send and could potentially start degrading the device's capabilities for GPS precision or to correctly detect vehicle events.

During the time interval where the latency increased considerably (November 26 to December 3), I had also been varying the frequency for different combinations of 59 loggers. After realizing that no more than 4 loggers could be set with the maximum frequency, the configuration was adjusted, and after a couple of days the device was able to stabilize.

Device B

This device also transmitted to the QA environment; however, its configuration was kept more stable compared to A, since it was used to investigate some Global Positioning System (GPS) problems. Some large spikes can be observed on the following dates: November 13, November 28, December 6 and December 9. November 13 and 28 correspond to reproducing configurations from device A into B to confirm the source of the problem was in fact logger frequency and not related to other matters.

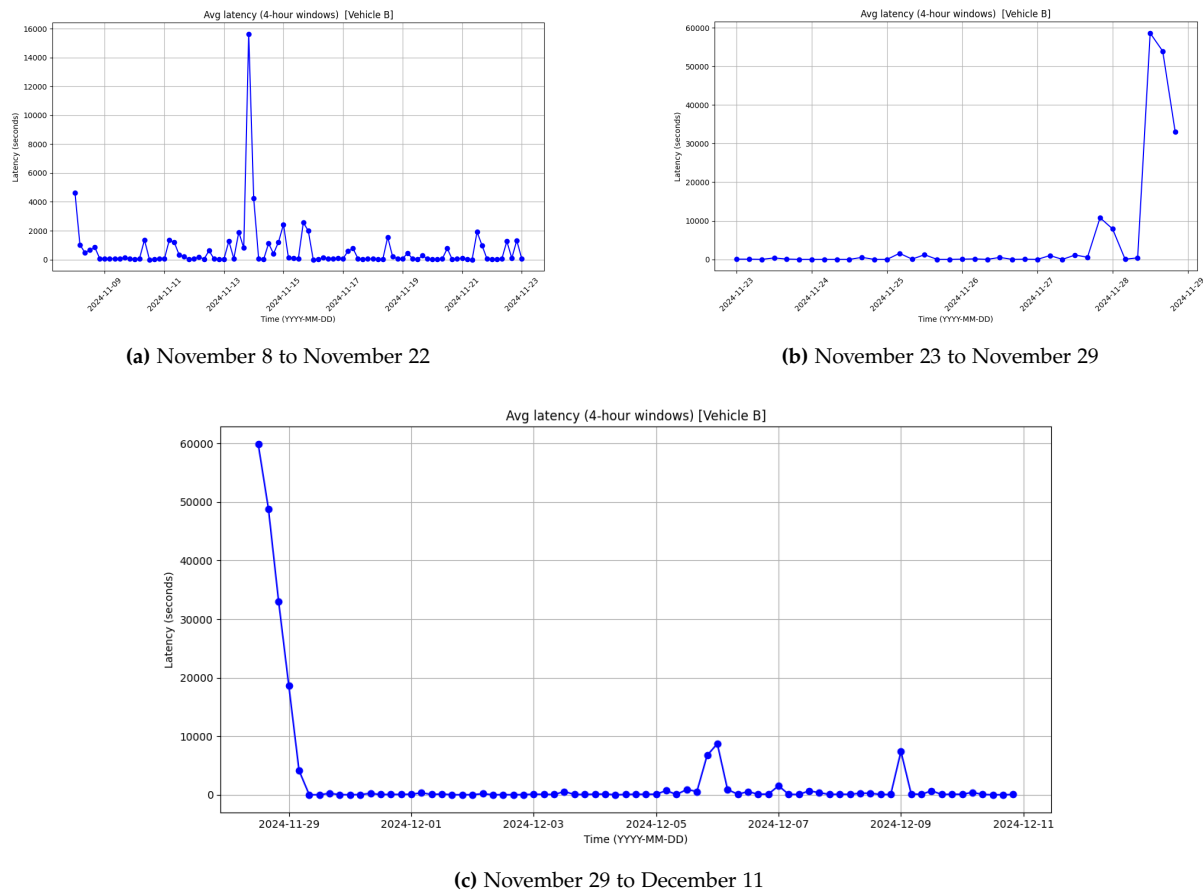


Figure 4.3: Latency results for vehicle B

The cause for the spikes in latency observed during the month of December have not yet been explained. This is an ongoing task at AutoPi where I am participating.

Device C

Lastly, device C is owned by a customer who is doing research in EV parameters. It was tested in one type of vehicle from early November until November 16, then there was a pause in testing, and then they were resumed around November 21. The large gaps in data are explained either by the test pauses done by the customer. They would usually test the device for a period of time and then inspect the generated data.

The high latency values in their case is explained due to the fact that they increased the logger frequency for all the EV data loggers since they were interested in having the best data resolution. However, this brought about problems with their GPS precision, and also the data would take a very long time to be available on the cloud platform. Once the cause was identified with the analysis from the other two devices, the configuration was adjusted and the device started to stabilize after November 25.

The intermittence in the values between November 25 to December 3 is explained by the fact that most data loggers were deactivated, and only EV loggers with minimal sampling frequency were kept. After 3 December, the device stabilized with the new default configuration described in 4.1.

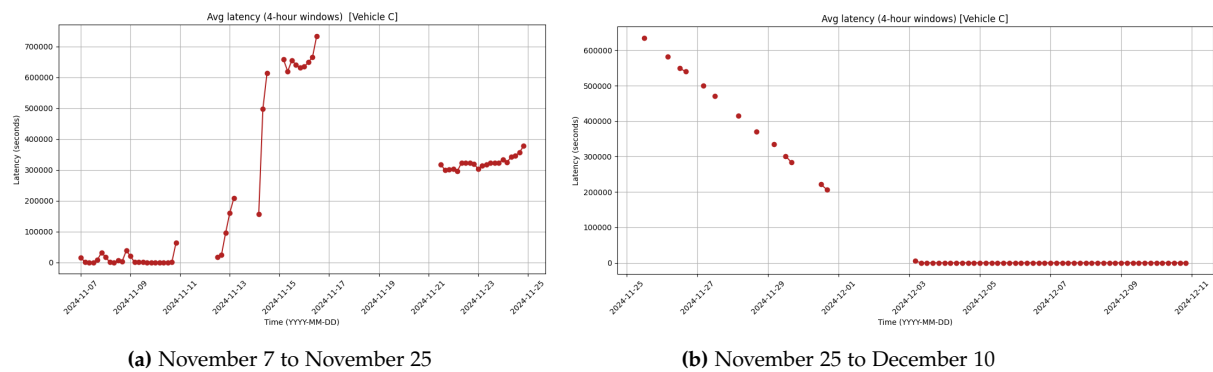


Figure 4.4: Latency results for vehicle C

These tests were quite different from the other two since they took place in the production environment and inspecting logs directly on the device was not possible since to do so the device needs to be connected to an emulator in a local environment. Most of the troubleshooting had to be done remotely which also posed a great challenge.

Conclusion

My experience with AutoPi was very enriching, after it I can confidently say that I have a solid understanding of communication network setups involving IoT devices, and, on the other hand, what aspects to keep in mind when working with the different protocols involved. The hardest part for me was working with the CAN protocol, but it was a great way to gain more experience with it and see the potential it has for future applications.

In hindsight, it would have been better to explore the configurations of the data loggers with a more systematic approach. In particular, having a way to emulate EV or Internal Combustion Engine (ICE) codes, or alternatively replaying a CAN dump would also have been of great value, since it would have saved many days of work. However, I also see value in the way the tests were carried out, as it improved my collaboration and problem-solving skills.

For the logger accuracy metric, in particular, a statistical method such as domain frequency analysis would have been a more robust way to ensure that no undersampling or oversampling occurred and to fine-tune frequency better. Regarding data latency, I am confident that the configuration currently set for Mini devices give it a better performance, but there is still a lot of room for improvement, for example, the MQTT broker currently has a level 0 Quality of Service (QoS), and another detail is that all AutoPi Cloud entities (from different environments) are listening to all the broker messages when this could be simplified to optimize resources.

There is plenty of work that can be done; In my opinion, there is great value in focusing on the following topics.

- CO₂ reporting by fleet
This is a topic of interest among AutoPi customers. Using data collected from the CAN bus reports can be generated, which can include a simple algorithm to generate recommendations to end users. This can later be integrated into decarbonization (or carbon accounting) tools that have gained popularity in recent years.
- FURPS optimizations
There are plenty of optimizations that came up from doing the FURPS model analysis. Some examples could include implementing QoS levels appropriately to balance delivery guarantees and performance and optimizing payload sizes to reduce bandwidth usage. Furthermore, fine-tuning the keep-alive interval, employing efficient topic hierarchies, and using lightweight encryption methods can improve both the performance and security of the MQTT network.

Bibliography

- [1] AutoPi. *AutoPi | Our history*. <https://www.autopi.io/about-us/>. (Accessed on 03/10/2024). Jan. 2023.
- [2] AutoPi. *What is OBD2 and How Does it Work?* <https://www.autopi.io/blog/what-is-obd-2/>. (Accessed on 15/10/2024). Jan. 2024.
- [3] AutoPi. *AutoPi Mini*. <https://www.autopi.io/hardware/autopi-mini/>. (Accessed on 15/10/2024). Jan. 2024.
- [4] AutoPi. *AutoPi Telematics Unit CM4*. <https://www.autopi.io/hardware/autopi-tmu-cm4/>. (Accessed on 15/10/2024). Jan. 2024.
- [5] RaspberryPi. *RaspberryPi Compute Module 4*. <https://www.raspberrypi.com/products/compute-module-4/?variant=raspberry-pi-cm4001000>. (Accessed on 15/10/2024). Jan. 2024.
- [6] AutoPi. *AutoPa CAN-FD Pro*. <https://www.autopi.io/hardware/autopi-canfd-pro/>. (Accessed on 15/10/2024). Jan. 2024.
- [7] U.S. Environmental Protection Agency. *Regulations for On-Board Diagnostics (OBD) Systems for Light-Duty Vehicles and Trucks*. <https://www.epa.gov>. 1994.
- [8] CSS Electronics. *OBD2 Explained - A simple intro*. <https://www.csselectronics.com/pages/obd2-explained-simple-intro>. (Accessed on 15/11/2024). Jan. 2024.
- [9] Bosch. *CAN Protocol Overview*. <https://www.bosch.com>. Accessed: January 2, 2025.
- [10] Andy Stanford-Clark and Arlen Nipper. *MQTT Version 3.1.1*. OASIS Standard Specification. 2014. URL: <https://mqtt.org/>.
- [11] HiveMQ. *HiveMQ - MQTT Broker for IoT Applications*. 2025. URL: <https://www.hivemq.com/>.
- [12] Eclipse Foundation. *Eclipse Mosquitto - An Open Source MQTT Broker*. 2025. URL: <https://mosquitto.org/>.



Worksheets

A.1 September

Week	Task Group(s)	Detail
35	Internal	<ul style="list-style-type: none">- Introductions and onboarding meetings with marketing and development teams to understand AutoPi's internal organization.- Configuration of local environments for development and testing.- Introduction to <i>Squid3</i> project for device checkout.
36	Backend Frontend	Reviewed internal documentation and started working on improvements for the Squid3 project. In short, the enhancements consisted in adding state, device ID, timers and other useful information needed by checkout operators.
37	Backend Core Frontend	<ul style="list-style-type: none">- Code review for changes applied to Squid3, implemented corrections from supervisor's feedback.- Started testing functionalities in AutoPi Cloud in order to detect bugs.- Configuration of an OBD-II emulator to test CAN loggers in the cloud platform.- Introduction to API testing in the backend for the cloud platform.
38	Backend Documentation Frontend Support	<ul style="list-style-type: none">- Implemented fixtures for API tests in backend (Python and Typescript).- Reviewed user permissions for API endpoints.- Updated internal documentation to facilitate developers access to the different testing environments that allow to investigate issues in devices or in the cloud platform.- Implemented a bash script to automate deployment process of Squid3 application.- Created a short guide to help a customer install an external antenna for their TMU devices.- Assisted a customer in updating their TMU device configuration in order to enable CAN loggers.- Started investigation to learn how to generate CO₂ reports from vehicle data.- Reviewed SIM provider documentation in order to integrate the SIM usage information into AutoPi Cloud.

A.2 October

Week	Task Group(s)	Detail
39	Backend	<ul style="list-style-type: none"> - Started SIM provider API integration. - Finished backend API tests update.
40	Backend Support	<ul style="list-style-type: none"> - Code review for API integration, implemented corrections and started design for alerts related to data usage and SIM meta-data. - CO₂ reports investigation. - Introduction to AutoPi Mini device, reviewed internal documentation and set up local environment for development and testing. - Assisted a customer having issues reading EV and non-EV parameters with the Mini device (support ticket #2).
41	Backend Core Frontend	<ul style="list-style-type: none"> - Started working on updates for the AutoPi Mini internal documentation. System diagrams, debugging set up, and other development topics. - Provided a working configuration to close support ticket #2. The fix consisted in increasing the sampling frequency to collect vehicle data.
42	Backend Documentation Frontend Support	<ul style="list-style-type: none"> - AutoPi Mini project: revision and updates for internal documentation. Updated frontend to hide settings that should be available only for AutoPi developers. - SIM provider project: implemented corrections obtained from code review. - CAN data logging project: introduction to new backend endpoint requirements.
43	Backend Documentation Frontend Support	<ul style="list-style-type: none"> - CAN data logging project: started implementation for backend endpoints. CAN logger channel endpoints for Create, Read, Update and Delete operations (CRUD). - AutoPi Mini project: revision and updates for internal documentation. - Customer support: implemented a script in Python to allow a customer to query data records for longer periods of time. The records were collected from a Mini device. - CAN data logging project: code review and updates for channel endpoints.

A.3 November

Week	Task Group(s)	Detail
44	Backend internal	<ul style="list-style-type: none"> - CAN data logging project: API serializers for channel endpoints - QA meeting - AutoPi Mini project: updated internal documentation from supervisor feedback. Loggers testing with OBD2 emulator.
45	Backend Frontend	<ul style="list-style-type: none"> - AutoPi Mini project: implemented script to download and preprocess vehicle data records. - CAN data logging project: updated CRUD endpoints for channel and logger groups.
46	Backend Core Frontend Support	<ul style="list-style-type: none"> - AutoPi Mini project: exploratory data analysis for data acquired from devices. Researched optimal sampling frequency for loggers, to do this I configured a Mini device and tested it in a coworker's vehicle. Updated internal documentation. - CAN data logging project: started design and implementation for metadata and default values endpoints. - Support ticket #3: investigated GPS accuracy and data latency for a customers Mini device.
47	Backend Documentation Frontend Support	<ul style="list-style-type: none"> - AutoPi Mini project: more exploratory data analysis, improved preprocessing script. - CAN data logging project: updates and corrections for API endpoints and database model. - Support ticket #3: continued investigation related to GPS accuracy and data latency.

A.4 December

Week	Task Group(s)	Detail
48	Internal	<ul style="list-style-type: none"> - SIM provider project: started internal documentation and added corrections obtained from code review with company supervisor. - AutoPi Mini project: updated frontend for cloud platform by hiding advanced settings that should only be visible to AutoPi users. - CAN data logging project: worked on corrections for endpoints and finished implementation for defaults and metadata groups. Started implementation for decoders CRUD endpoints group.
49	Backend Frontend	<ul style="list-style-type: none"> - SIM provider project: tested API integration in QA environment. Made annotations for enhancements and to fix errors found in the implementation. - AutoPi Mini project: reviewed vehicle data generated from past weeks. - CAN data logging project: corrections for endpoints. Finished implementation for decoders endpoints.
50	Documentation	- Worked on AAU project report.
51	Documentation	- Worked on AAU project report.
52	Documentation	- Worked on AAU project report.