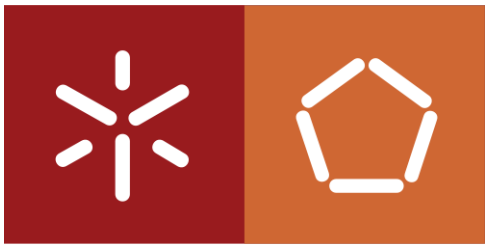

Universidade do Minho
Escola de Engenharia

Projeto em Engenharia Informática

Technical Report

PEILoad

RideCare

Master's Degree in Informatics Engineering
2020/2021

BOSCH

February 1, 2021

TECHNICAL REPORT

RideCare

Universidade do Minho
Escola de Engenharia

Document ID	RT-20201218-PEI2020
Version	1.0
Access	Restricted
Date of issue	February 1, 2021
Authors	André Coutinho (pg39284) Gabriela Martins (a81987) João Costa (a81822) José Pinto (a81317) Leandro Costa (pg41083) Luís Correia (a81141) Paulo Martins (pg17918) Pedro Barbosa (a82068) Rafaela Silva (a79034) Rui Santos (pg41850) Tiago Fontes (a80987)
Remittee	PEI2020

Contents

1 Introduction

The rapid growth of population within cities is directly related to the reduced urban mobility capacity, which negatively impacts the quality of life in cities, restricting mobility options.

Moreover, multiple factors make the usage of private vehicles, in a citizen mobility context, less appealing. As an example, maintenance expenses and hideous traffic can sometimes outshine the convenience that is using a private car at any time.

These factors, combined with the evolution of technology, led to the emergence of new mobility services such as **ride halling** (e.g. Uber or Lyft) and **car sharing** (eg. Sixt and ShareNow) which provide new options for urban mobility, demoting the use of a personal vehicle.

One concern of these service providers is the assurance of the integrity state of their vehicles, as well as the safety and comfort experienced by the customers. With car-sharing services, there's no straightforward form to guarantee that the established usage guidelines are being complied with. Therefore, there's no way of tracking the security and minimum comfort conditions of the vehicles between usages without using employees who can make the trip for an on site assessment. Thus, it is of extreme relevance to adopt a monitoring system that assures the integrity of both customers and vehicles as well as improve business value.

1.1 Opportunities

It is estimated that the number of users of car-sharing services worldwide in 2025 will surpass 36 million [1], resulting in companies that provide this type of services having an increase of

demand and customer engagement.

Such an exponential growth in users turn this market into a promising new trend. As far as it is known, there are no such systems that provide real-time monitoring of the integrity of a fleet of vehicles in a car-sharing context. The existing alternative solutions are based on manual inspections of the vehicles between usages, by a human inspector. That being said, there is a clear opportunity for innovation.

Imagine a significant flow of users using the same car-sharing service vehicle in a given day. Every time there is a switch of customers, the service provider should conduct a local inspection of the vehicle to assure its conditions are the same as before the usage. This process would be far from ideal, as with these kinds of services, the vehicle can be parked at several places in the city, making it impractical to assure someone is nearby to conduct the inspection. What if there was a more convenient alternative?

1.2 The idea

With RideCare, it is intended to give car-sharing service providers a system capable of monitoring, detection, and notification on the occurrence of anomalous events inside the vehicles.

This way, it is possible to provide added assurance to companies in regards to customer accountability towards damages inflicted due to non-compliance with established standards, such as the ban on smoking cigarettes.

In addition, another benefit of the adoption of such a system is the increased sense of security for both users and companies, since it can detect security compromising disasters, for example, detect a fire through smoke.

1.3 Domain Diagram

During the initial analysis and planning phases, a domain diagram was developed in order to illustrate the relevant conceptual classes for the project, so a better understanding of the field of work is provided.

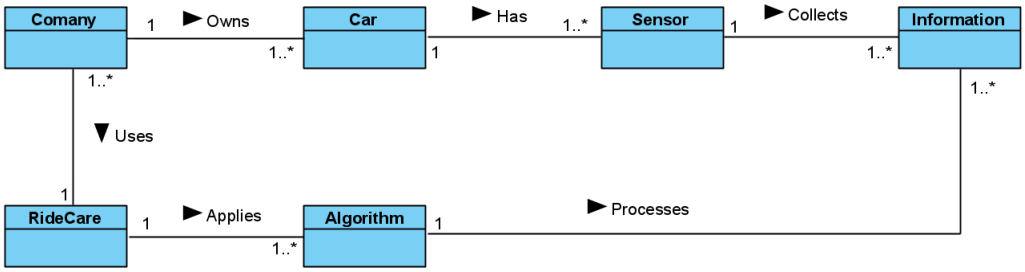

Figure 1.1: Domain Diagram.

1.4 Use-cases diagram

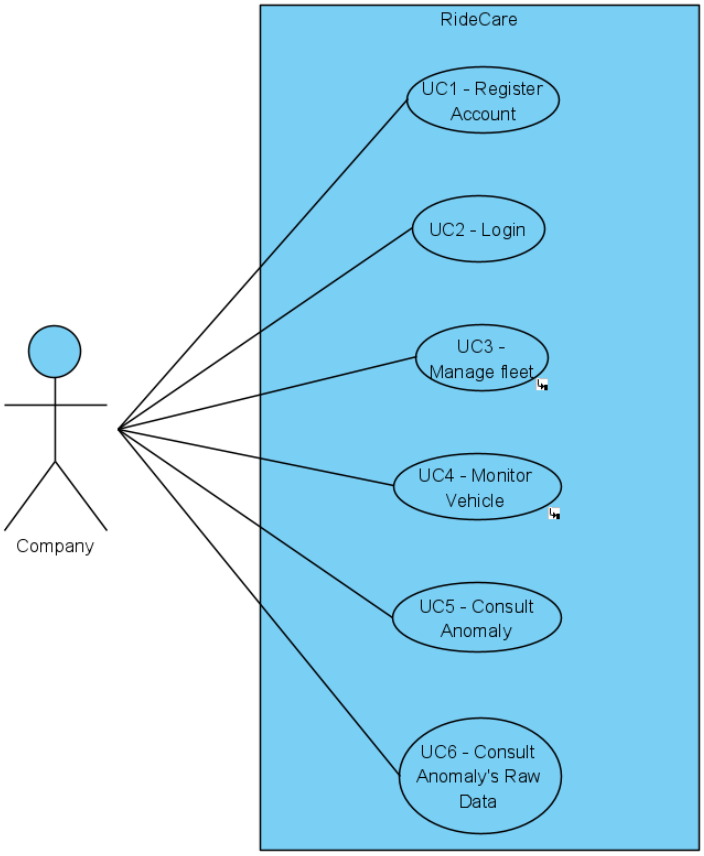

Figure 1.2: Use Case Diagram.

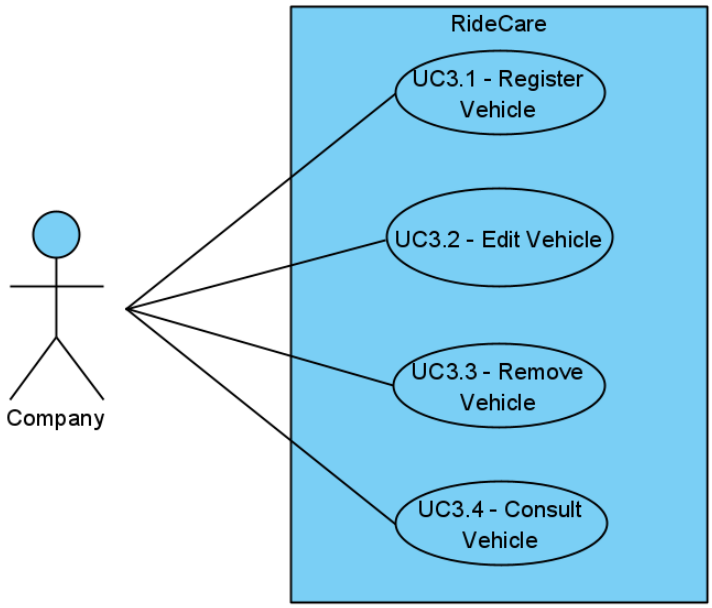

Figure 1.3: Use Case Diagram "Manage Fleet".

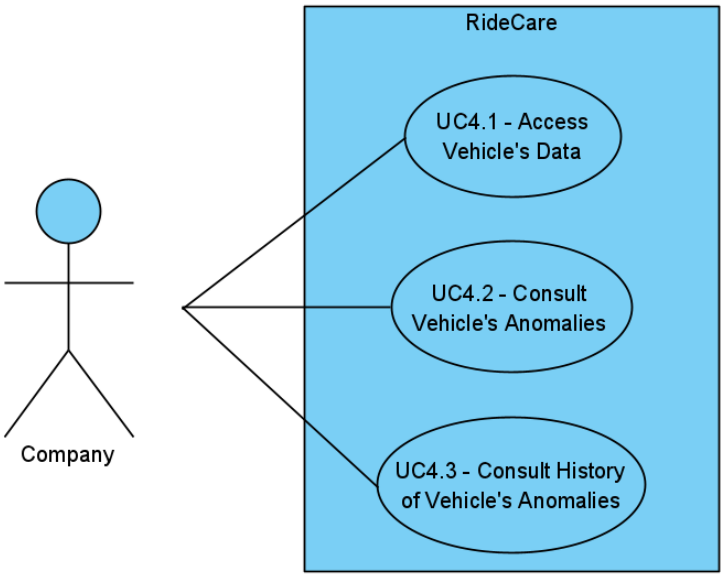

Figure 1.4: Use Case Diagram "Monitor Vehicle".

1.5 Use Cases

Use Case:	UC3.1 - Register Vehicle	
Actor:	Client	
Precondition:	User Authenticated	
Postcondition:	Vehicle registered in the system	
Normal case	Actor input	System response
	1. User selects "MyCars" option in the application menu	
		2. Displays "MyCars" interface
	3. User selects "New Car" option	
		4. Displays form
	5. User fills the form and submits data	
		6. System validates the data
		7. Registers the new vehicle and informs the user
Exception stage 6 (invalid data)		6.1 Informs the user that invalid data was inserted
		6.2 Returns to point 4

Figure 1.5: Description of use case "Register Vehicle".

Use Case:	UC3.2 - Edit vehicle	
Actor:	Client	
Precondition:	User authenticated and vehicle registered	
Postcondition:	Vehicle with updated information	
Normal case	Actor input	System response
	1. User selects "MyCars" option in the application menu	
		2. Displays "MyCars" interface
	3. User selects "Edit" option	
		4. Displays form
	5. User edits and submits data	
		6. System validates the data
		7. Registers the new data and informs the user
Exception stage 6 (invalid data)		6.1 Informs the user that invalid data was inserted
		6.2 Returns to point 4

Figure 1.6: Description of use case "Edit Vehicle".

Use Case:	UC4.2 - Consult vehicle anomalies	
Actor:	Client	
Precondition:	User Authenticated and anomalies registered	
Postcondition:	Not applicable	
Normal case	Actor input	System response
	1. User selects "MyCars" option in the application menu	
		2. Displays "MyCars" interface
	3. User selects a vehicle	
		4. Displays info about the vehicle alongside the anomalous events

Figure 1.7: Description of use case "Consult vehicle anomalies".

1.6 Functional requirements

In order to delimit and better perceive the functional requirements that the system encompasses, this section will present the list of requirements, from a car-sharing service provider user and system's perspective. Each one of the requirements is displayed through the use of **Volere Cards** [2] which provide a detailed insight of its general description, validation criteria, priority, and more.

1.6.1 User functional requirements

Firstly, the user requirements are presented in the form of a list in order to provide a more general overview, followed by the complete description of each individual requirement.

- **User Requirements**
  1. Register account
  2. Log in
  3. Register vehicle
  4. Edit vehicle
  5. Remove vehicle
  6. Check vehicles
  7. Obtain vehicle data
  8. Check real-time anomalous events from a vehicle
  9. Check history of anomalous events from a vehicle
  10. Check all anomalous events from every vehicle
  11. Check raw data collected from each vehicle

Requirement #: 1 **Requirement Type:** 9

Event/Use Case #: 1

Description: The user must be able to register in the system.

Justification: It is necessary to identify the users, since they will have confidential information in the system.

Validation criteria: The home page of the system should contain a section for registering users and the registration data must be stored correctly.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 2 **Requirement Type:** 9

Event/Use Case #: 2

Description: The user must be able to login in the system.

Justification: Access to the system is vital to using its features.

Validation criteria: The data entered by the user must be validated by an authentication mechanism.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 3 **Requirement Type:** 9

Event/Use Case #: 3.1

Description: The user must be able to register vehicles.

Justification: Vehicle registration is essential for the presentation of possible anomalous events in its interior.

Validation criteria: The vehicle registration will be stored correctly.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 4 **Requirement Type:** 9

Event/Use Case #: 3.2

Description: The user must be able to edit vehicles information.

Justification: The user must be able to edit information relating his vehicle(s).

Validation criteria: Updating vehicle's information in the system.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 5 **Requirement Type:** 9

Event/Use Case #: 3.3

Description: The user must be able to remove vehicles from his fleet.

Justification: The user must be able to remove his/her vehicle(s) if, for example, he/she wishes to stop the car sharing of the vehicle(s).

Validation criteria: Removal of the vehicle from the system.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 6 **Requirement Type:** 9

Event/Use Case #: 3.4

Description: The user must be able to consult his vehicle(s).

Justification: Make the list of the user's vehicles registered in the system available to him.

Validation criteria: Displays all vehicles stored in the system.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 7 **Requirement type:** 9

Event/Use Case #: 4.1

Description: The user must be able to obtain real time data related to his vehicles.

Justification: Give real-time user access to the data gathered from the crawler systems installed in all their vehicles.

Validation criteria: Perform test scenarios, in order to infer whether the system presents vehicle data correctly.

Client satisfaction: 5 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 8 **Requirement type:** 9

Event/Use Case #: 4.2

Description: The user must be able to consult anomalous events from specific vehicles in real time.

Justification: Ensure that the user is aware of the anomalous events that are occurring in his vehicle(s) in real time.

Validation Criteria: Display of a vehicle's anomalous events detected by the system.

Client Satisfaction: 1 **Client Dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 9 **Requirement Type:** 9

Event/Use Case #: 4.3

Description: The user must be able to consult the history of anomalous events of his vehicle(s).

Justification: Allow the user to access previous anomalous events to which a given vehicle has been subjected, in order to infer its conditions over time.

Validation Criteria: Display of the history of anomalous events detected by the system.

Client Satisfaction: 1 **Client Dissatisfaction:** 5

Priority: Medium **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 10 **Requirement type:** 9

Event/Use Case #: 5

Description: The user must be able to consult anomalous events of his vehicles in real time.

Justification: Ensure that the user is aware of all anomalous events that are occurring in his vehicle(s) in real time.

Validation Criteria: Display of all anomalous events detected by the system.

Client Satisfaction: 1 **Client Dissatisfaction:** 5

Priority: High **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

Requirement #: 11 **Requirement type:** 9

Event/Use Case #: 6

Description: The user must be able to consult the raw data which generated the anomalous event.

Justification: Provide the user with the reason for the anomalous event, allowing an analysis of the concrete values.

Validation criteria: Tests the occurrence of anomalous events and validate the values presented.

Client Satisfaction: 1 **Client Dissatisfaction:** 5

Priority: Medium **Conflicts:** None

Support documentation: Use case diagram, chapter 1.4

History: Last modification: 05/11/2020

1.6.2 System functional requirements

- **System Requirements**

  1. Collect data
  2. Store data
  3. Process collected data
  4. Detect anomalous events
  5. Report anomalous events
  6. Consume data from the crawling systems installed on the vehicles
  7. Provide instant detection of anomalous events
  8. Ensure communication between components
  9. Provide graphical representation of anomalous events

Requirement #: 1 **Requirement Type:** 9

Description: The system must be able to collect the data acquired by the sensors.

Justification: The collection of data is essential for the entire pipeline to be covered to detect anomalous events.

Validation criteria: Verification of the collection through test data, under random conditions.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 2 **Requirement Type:** 9

Description: The system must be able to store the acquired data by the sensors.

Justification: It is relevant to have possession of the history of sensory values of the vehicles. Either from a perspective of consultation or in the context in which the constant acquisition of new data makes it possible to improve the anomaly detection algorithm.

Validation criteria: Realization of test cases to verify if the data storage in the system is correct.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 3 **Requirement Type:** 9

Description: The system must be able to process the data acquired.

Justification: The data must undergo a treatment stage since these operations can be very beneficial for the anomalous events classification process.

Validation criteria: Use of standardization techniques, standardization, removal of outliers, and other relevant methods.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 4 **Requirement Type:** 9

Description: The system must be able to detect anomalous events.

Justification: Since the main objectives are the detection of anomalous events inside vehicles in the context of car sharing, the system must be able to detect them.

Validation criteria: Conducting test cases to infer if the system detects anomalous events, such as tobacco smoke inside a vehicle, for example.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 5 **Requirement Type:** 9

Description: The system must be able to report anomalous events.

Justification: Ensure that users are aware of anomalous events detected by the system.

Validation criteria: Realization of test cases to find out if the system reports anomalous events upon detection.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 6 **Requirement Type:** 9

Description: The system must be able to consume the data from the crawling systems installed on the vehicles in real-time.

Justification: Provide the user with data on all his vehicles in real-time so that he can follow their whereabouts.

Validation criteria: Realization of test cases to verify if the system obtains the vehicle data correctly.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 7 **Requirement Type:** 9

Description: The system must be able to provide the instant detection of anomalous events.

Justification: The presentation of the instant of the anomalous event detection is essential so that the user can identify the user who was using the vehicle. Moreover, to take action as soon as possible.

Validation criteria: Realization of test cases to infer if the system reports the correct detection date.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 05/11/2020

Requirement #: 8 **Requirement Type:** 9

Description: The system must be able to guarantee communication between its components.

Justification: Provide information transmission between the various components.

Validation criteria: The data collection system must connect with the various components through the 4G system.

Client satisfaction: 1 **Client dissatisfaction:** 5

Priority: High **Conflicts:** None

History: Last modification: 06/11/2020

Requirement #: 9 **Requirement Type:** 9

Description: The system must be able to provide a graphical representation of the anomalous events detected.

Justification: Provide the user with a graphical and intuitive representation of the detection of anomalous events.

Validation criteria: The system should present a graph representing the detected events.

Client satisfaction: 3 **Client dissatisfaction:** 5

Priority: Low **Conflicts:** None

History: Last modification: 06/11/2020

1.7 Non functional requirements

In the following subsections, the non-functional requirements of this system will be presented, framed in the sections that the team considered appropriate. They will be, as with functional requirements, presented through the Volere Card templates.[2].

1.7.1 Look and Feel Requirements

Requirement # 1

Description: The system must provide a intuitive interface.

Justification: Provide the user simple and easy learning interface.

Validation criteria: A group of potencial users should evaluate the level of intuitiveness of the interface.

Priority High

History: Last modification: 05/11/2020

1.7.2 Usability and Humanity Requirements

Requirement # 1

Description: The system must help the user not inserting erratic data (email verification, NIFs, etc).

Justification: Making the system more appealing while ensuring consistent and valid data entry.

Validation criteria: The system should validate the insertion of data which is invalid in terms of its format and length, if this is fixed.

Priority Medium

History: Last modification: 05/11/2020

Requirement # 2

Description The system should require few steps for the user to realize the main functionalities.

Justification: Provide the user with a simple and appealing experience.

Validation criteria: The average user should need four or fewer clicks to perform the desired tasks.

Priority Medium

History: Last modification: 05/11/2020

Requirement # 3

Description The system must use different colors to present success and failure, namely green and red, respectively.

Justification: Simplify the user experience, with particular relevance for users with visual problems.

Validation criteria: A group composed of potential users should evaluate the simplicity of using the system based on the colors used.

Priority High

History: Last modification: 05/11/2020

1.7.3 Performance Requirements

Requirement # 1

Description The in-vehicle system must withstand internet connection failures.

Justification: Provide a reliable and fault-tolerant system.

Validation criteria: The system must store the captures during the internet connection failure and reestablish communication when connection gets back on.

Priority High

History: Last modification: 05/11/2020

1.7.4 Maintenance and Support Requirements

Requirement # 1

Description The system must operate on Google Chrome and Mozilla Firefox.

Justification: Ensure that must users can use the system.

Validation criteria: Verification of system support by web browsers.

Priority High

History: Last modification: 05/11/2020

Requirement #: 3 **Requirement Type:** 14

Description System update takes place during the night of local time.

Justification: So as not to disturb normal user usage, system updates coincide with the time when the number of users is smaller.

Validation criteria: System update to include new features is performed during the night of local time.

Client satisfaction: 2 **Client dissatisfaction:** 4

Priority: Medium

History: Last modification: 05/11/2020

1.7.5 Security Requirements

Requirement #: 1 **Requirement Type:** 15

Description The system should only allow access to the data to the data owner.

Justification: In order to ensure data privacy and integrity, only data owners and administrators should have access to their confidential data.

Validation criteria: A registered user can access his own data.

Client satisfaction: 4 **Client dissatisfaction:** 5

Priority: High

History: Last modification: 05/11/2020

Requirement #: 2 **Requirement Type:** 15

Description The system must prevent the input of malicious data.

Validation criteria: The system must check the validation of data entered on forms so that it is protected against invalid data entered which could harm the system.

Client satisfaction: 4 **Client dissatisfaction:** 4

Priority: Medium

History: Last modification: 05/11/2020

Requirement # 3

Description The system must require acceptance of its terms and conditions of use.

Justification: The user must be aware that he/she consents to the use of his/her data.

Validation criteria: The system must request permission from the user to collect their data for statistical purposes before any other action.

Client satisfaction: 4 **Client dissatisfaction:** 5

Priority: Medium

History: Last modification: 05/11/2020

Requirement #: 4

Description: The system should notify its customers and users to update its terms and conditions of use.

Justification: If the terms and conditions change, the user must be notified of the changes to which he was subjected.

Client Satisfaction: 4 **Client Dissatisfaction:** 5

Priority: Medium

History: Last modification: 05/11/2020

1.8 Structure of the report

This technical report is composed by five main sections. Firstly, in Chapter 2, the applicational structure is exposed, where all the subsystems of the developed system are detailed.

Then, in Chapter 3, on a higher abstraction level, the implemented final infrastructure is presented, as well as all of the mechanisms introduced in order to assure high availability and fail tolerance. Subsequently, in Chapter 4, all the aspects regarding the client application interface are addressed.

Finally, in Chapter 5, it is presented a brief summary of the results of the project and exposed aspects considered relevant for its continuity.

2 Applicational Structure

Since this is a project carried out in a business context, a diagram specifying the components to be deployed was provided to us at an early stage. There are two main domains of components, the ones present in the vehicles (**FieldCrawler** and **AlertAI**) and the cloud-based ones (**DataLake**, **Backend web**, **Frontend**, **AlertAI Cloud**). Collaborating together, they form the RideCare system.

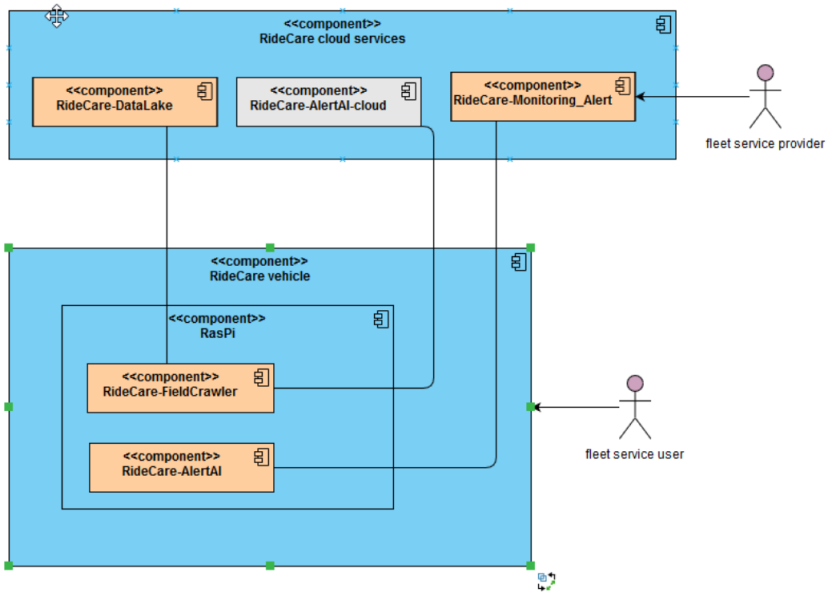

Figure 2.1: Component diagram provided by Bosch

2.1 FieldCrawler

The FieldCrawler subsystem is responsible for periodic real-time data acquisition from the sensors SDS011 (2.1.3), BME680 (2.1.4), and GPS Receiver G-MOUSE VK-162 (2.1.5) connected to the *raspberry pi 4* [1] microcomputers. The gathered data had the purpose of providing training data for the development of the machine learning algorithms, resulting in more than 10.000 samples in several situations being collected. For that purpose, scripts were developed, making use of libraries that allow the reading of the values measured by the sensors. Since Python was chosen to develop the scripts, the libraries used to access the sensor's values were py_sds011 [2] for the first sensor, adafruit_bme680 [3] for the second one, and pyembedded [4] for the last one. Even though there are multiple other compatible libraries, the ones selected promised stability and a low margin of error.

When the FieldCrawler subsystem is activated, firstly, it sends to the web backend the information regarding the moment when it is turned on. This data allows the user to have information about the use of the vehicle when it is online and offline. Once activated, this system periodically collects data from the sensors, with a customizable frequency. After collecting data, it goes through pre-labeling processes, placing the data in a specific defined format (Figure 2.7), to be posteriorly consumed by other components.

Once new data is gathered, it is sent to the alertAI module. This subsystem receives every raw data collected to apply Machine Learning algorithms and associate it with the corresponding anomaly classification. In situations where anomalous events are effectively detected, the AlertAI notifies the web back-end so it persists the anomaly information and the client application is able to report these anomalies in real-time. Besides the data acquisition and classification, this component also owns the responsibility to send the collected data to other cloud-based subsystems (DataLake and AlertAI Cloud) through an intermediary, the API Gateway (2.2).

Regarding the cloud-based subsystems, data is sent to the DataLake (2.3) in order to have all the raw data persisted, which is very valuable for further algorithms development. In the case of the AlertAI Cloud, the data sent will be classified by alternative Machine Learning algorithms (not present in the vehicle).

[1] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/
[2] https://pypi.org/project/py-sds011/
[3] https://pypi.org/project/adafruit-circuitpython-bme680/
[4] https://pypi.org/project/pyembedded/

2.1.1 Communication between FieldCrawler and other components

The communication between FieldCrawler and AlertAI occurs at a direct method invocation level since both components are present within the same device and were developed using the same programming language. On the other hand, all communication from the device to the interior of the cloud-based infrastructure is carried out through an intermediate component, a single controlled entry point (API Gateway (2.2)).

Offline mode

Since the FieldCrawler is a system present in the vehicle itself, the communication with the other components of the RideCare system occurs through the 4G connection provided inside the cars.

However, in order to avoid the cases where the device is in an area without mobile network coverage, a Redis-based job queuing system (Python-rq [5]) was implemented. This system accumulates the requests (jobs) until the connection is resumed. Whenever a request needs to be made, if, for whatever reason, an acknowledgment of reception of data is not received, the request will get stored and will be redone by a worker at a later time, when the internet connection is resumed. Thus, the high availability of the system is assured since there is no longer tight coupling between the FieldCrawler and Internet connection.

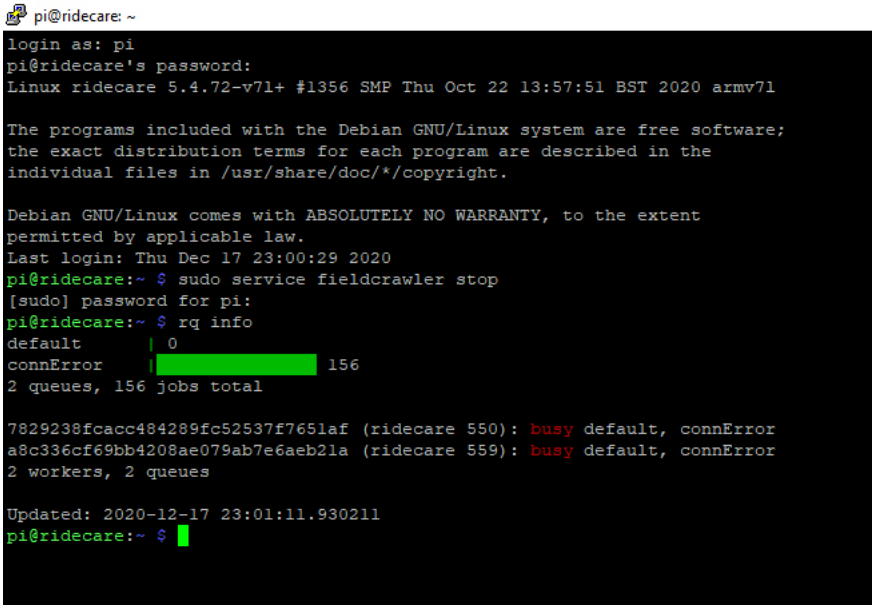

Figure 2.2: Accumulated jobs when there is no connection to the 4G network

[5] https://python-rq.org/

2.1.2 Other considerations

Regarding the possibility of sending fraudulent data by using other vehicles license's ID, this situation was handled beforehand through the implementation of a token-based authentication system (better described in section 2.2). During the registration process of a vehicle in the client application, a unique device ID is inserted, which identifies the installed system in a given car. In the situation of pretending to send data from a given false vehicle, these will be prevented due to authentication failure.

In other to facilitate the gathering of new data, during the development phase, system services were created in order to assure that once a device is turned on, the applications will automatically start gathering and classifying data, as well as the initialization of the necessary workers for the queuing system. There was also effort put into security matters of the in-car system, such as firewall configurations, brute force attacks protection, and more.

2.1.3 Sensor SDS011

SDS011 is a quality of the air sensor capable of measuring particle's concentration metrics, such as $\mathrm{PM}_{2.5}$ (particles with less than 2.5 micrometers of diameter) and PM_{10} (particles with a 2.5 and 10 micrometers range of diameter), which enables the ability to measure fine dust that may indicate the presence of smoke. Such is possible through the principle of laser mirroring. Commonly used in quality of the air measurement contexts in urbanized areas, whose reference values are presented in Figure 2.4, the data gathered from this sensor enabled the development of the Machine Learning algorithms.

Figure 2.3: SDS011 sensor

AQI Category	PM2.5 (ug/m3)	PM10 (ug/m3)	Health Impact
Good (0-50)	0-30	0-50	Minimal
Satisfactory (51-100)	31-60	51-100	Minor Breathing discomfort to sensitive people.
Moderately polluted (101-200)	61-90	101-250	Breathing discomfort to asthma patents, elderly and children.
Poor (201-300)	91-120	251-350	Breathing discomfort to all
Very poor (301-400)	121-250	351-430	Respiratory illness on prolonged exposure.
Severe (401-500)	250+	430+	Health impact even on light physical work. Serious impact on people with heart/lung disease.

Figure 2.4: Air quality reference values ($PM_{2.5}$ e PM_{10})

As mentioned above, we chose the py_sds011 library to read the values collected by the sensor. To extract the values using this library, the following code represents a simple example of how it is done:

```
# connects to the SDSO11 sensor through the mentioned port
sensor = py_sds011.SDS011(port, use_query_mode)

# reads the measurement
measurement = sensor.query()

# keeps the measurement
data = {"pm2.5": measurement[0],
        "pm10" : measurement[1],
       }
```

2.1.4 Sensor BME680

BME680 is a sensor developed by Bosch that is capable of measuring temperature, humidity, pressure and gas (VOC - Volatile Organic Compounds) values, which allows it to monitor metrics associated with air quality. Highlighted by its low energetic consumption and multiple possible applications, in smart homes and smart energy contexts, among other scenarios, this sensor provided valuable data for the development of the machine learning algorithms.

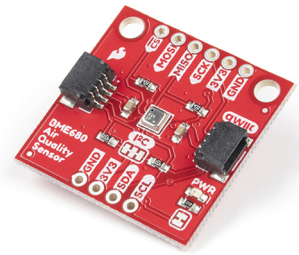

Figure 2.5: Sensor BME680

To read the values from this sensor, it needs to be connected to the raspberry, either by using a qwiic connect system or by welding wires and connect them directly to the microcomputer's pins. Then the I2C bus must be enabled within the raspberry in order to allow multiple devices to be connected. At last, the adafruit_bme680 and board packages must be installed. The following represents a simple way of using the library to gather the data from the sensor:

```
# connects to the BME680 sensor through the board
sensor = adafruit_bme680.Adafruit_BME680_I2C(
I2C(board.SCL, board.SDA))

# reads and keeps the values
data = {"temperature": sensor.temperature,
        "gas": sensor.gas,
        "humidity": sensor.humidity,
        "pressure": sensor.pressure,
        "altitude": sensor.altitude,
       }
```

2.1.5 GPS RECEIVER G-MOUSE VK-162

GPS Receiver G-MOUSE VK-162 is a GPS receptor module composed of an antenna and a USB interface. From it, it is possible to obtain data regarding geographic location, latitude and longitude. Usually, this type of device is connected to port "/dev/ttyACM0".

Figure 2.6: *GPS Receiver G-MOUSE*

As said before, we chose the pyembedded library to read the GPS receiver values. These values can be acquired as presented below:

```
# connects to the GPS receiver through the mentioned port
gps = GPS(port, baud_rate=9600)

# reads the values
coordinates = gps.get_lat_long()
```

2.1.6 *Raw Data*

The acquired data from the sensors, before any labeling or processing, is known as raw data. These collections of data are composed of an id, vehicle identifier, timestamp of the collection, vehicle location, and data from the SDS011 and BME680 sensors. For the persistence of these collections of raw data, a datalake 2.3 was developed.

Primarily, this kind of data proved to be essential for the training of Machine Learning algorithms. During the production phase, these collections of raw data are stored to allow a constant evolution of the existing algorithms and support the development of alternative ones.

The data periodically sent through the network to the DataLake has the following structure:

```
{
    "id": 5882,
    "carId": "66-ZZ-66",
    "timeValue": "2020-12-17 23:00:30",
    "carLocation": "41.5608 -8.3968",
    "temperature": 20.234296875,
    "pm10": 14.2,
    "pm25": 6.3,
    "altitude": -53.09144555572551,
    "gas": 112302,
    "humidity": 62.787795346847886,
    "pressure": 1019.6430957564247
}
```


Figure 2.7: Data row format

2.1.7 Installation instructions

The main requirement is a device that features compatible interfaces with the sensors used (BME680, SDS011, and GPS Receiver G-MOUSE VK-162). Moreover, this device should have installed all the libraries and dependencies mentioned above, as well as the creation of the necessary services for the automatic start. A script was created to execute all the necessary steps required for the start of the application. The following command executes the script.

```
$ ./install_fieldcrawler.sh
```

The next time the system boots, FieldCrawler will start collecting and sending data and the AlertAI module will assess the presence of anomalies and act accordingly.

2.1.8 Final Setup

The setup used in the vehicle for the development of the project was the following:

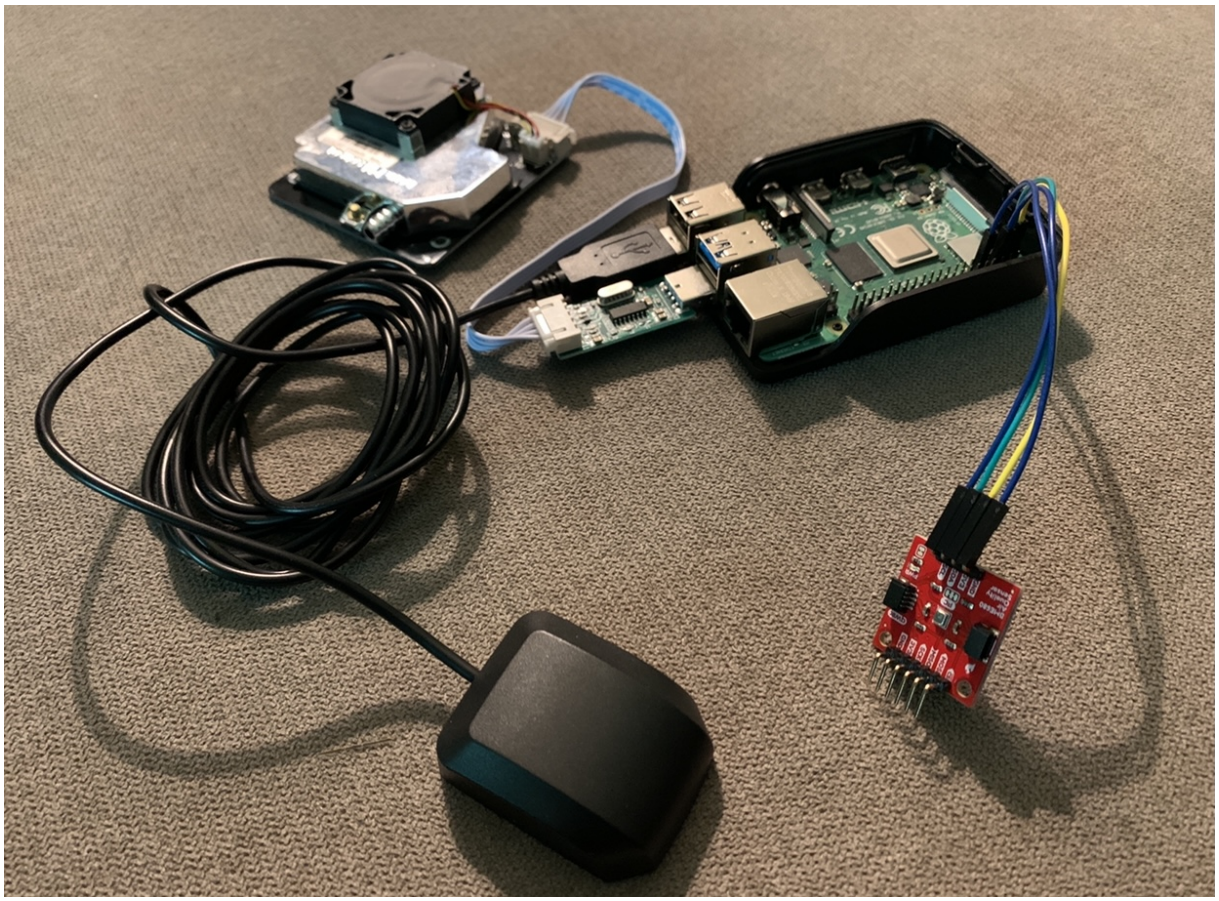

Figure 2.8: Setup used in the vehicles

2.2 *API Gateway*

The API Gateway, developed using the Flask framework, acts as an intermediary between the cloud-based subsystems (DataLake, web backend and AlertAI Cloud) and the subsystems in the vehicle (FieldCrawler and AlertAI). We also developed this system in order to implement an authentication mechanism and provide a single controlled access point.

2.2.1 Authentication

Whenever the FieldCrawler subsystem collects data, it makes HTTP requests to the API Gateway, and then, this component will check whether they meet the authentication requirements. If so, the HTTP requests will pass through to the API Gateway. Otherwise, requests are not satisfied due to failed authentication.

To this end, token-based authentication was implemented, in which all requests from the car

system, whether the destination is the DataLake, AlertAI, or web backend, are validated through the HTTPTokenAuth module, from the library flask_httpauth.

Regarding the token request, the FieldCrawler initially requests it to API Gateway through basic authentication (HTTPBasicAuth module from the library flask_httpauth). The username is the license plate, and the password is the unique device id associated with the vehicle. Both parameters are sent to the API Gateway with encryption applied (AES block encryption in GCM mode) to ensure confidentiality. This way, it is only allowed to obtain the token from permitted entities and its expiration time is ten minutes.

To clearly understand our authentication system, observe Figure 2.9.

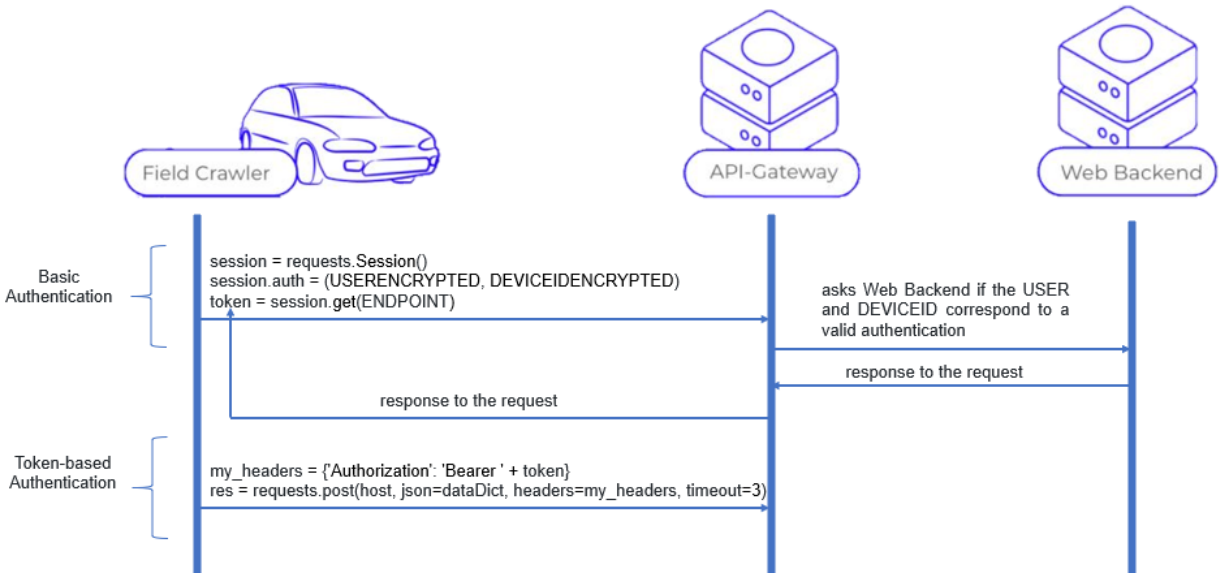

Figure 2.9: Authentication system implemented in the API Gateway

2.2.2 Job queuing system

Since this component corresponds to the only possible entry point to access the cloud-based infrastructure, there is a possibility that some of the requests will exceed their timeout period due to a possible system overload. That is possible in situations where there are hundreds of thousands of vehicles registered and in use.

Although this academic context only has a maximum of three simultaneous vehicles, we decided to implement a job queuing system in order to solve this hypothetical inconvenience. This decision takes into consideration the awareness that, in a real scenario, the load would be exponentially greater. As with FieldCrawler (2.1), we used Python RQ. Although these are different use cases, we consider that both situations are valid and justify their use.

In the context of the API Gateway, it ensures that all requests to this component will be eventually satisfied, with no risk of it being overloaded and failing due to this situation. However, in a certain way, it can be seen as a trade-off between high availability and consistency, which in this context, is given priority to availability. Additionally, the queuing system provides a web interface where it is possible to observe the jobs on hold. It is also possible to view the status of the active workers, among other information.

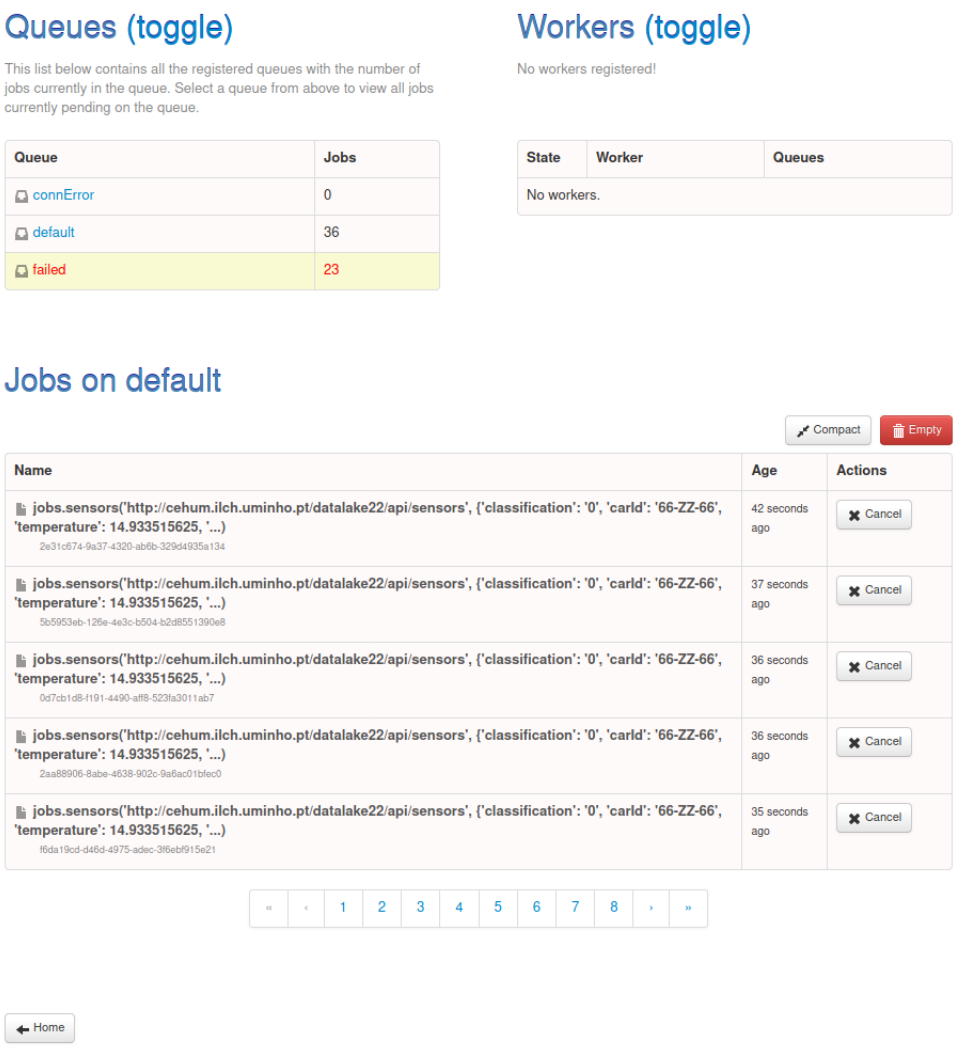

Figure 2.10: Jobs accumulated in extreme load situations in the API Gateway

2.2.3 Deploy

Since we chose to deploy this system in a docker container, we developed a script to build and run the Dockerfile corresponding to the API Gateway. The following command runs it.

```
$ ./run_apiGateway.sh
```

Once the container is created and run, the **docker start** and **docker stop** commands start and stop the instance, respectively.

2.3 *Data Lake*

The Datalake subsystem is responsible to store and provide the raw data that came from the FieldCrawler systems installed in vehicles. This repository of raw data, following a SQL schema, stores all the past and new data gathered by the vehicles. During its development, this component had the purpose of providing training data for the development of the machine learning algorithms. Subsequently, during a production phase, it continues to store the collected data with the purpose of later data querying and further development of the existent machine learning algorithms and alternative ones.

The structure of the data during the development and production phases were different. Initially, since the machine learning algorithms weren't developed, the data was sent with a hard-coded classification and a description of the gathering conditions. This manual data insertion before sending it was necessary so the machine learning team could develop their supervised learning models. Then, during the production phase, this manual classification was naturally removed and not sent with the sensorial data, since it is now classified by our machine learning algorithms.

In order to query the raw data stored in the datalake, this subsystem provides a collection of endpoints that can be accessed by the Web Backend, requested from the client application (e.g. sensorial data before and after a given anomaly, boot times, etc.). Several filters and custom queries can also be applied to the data present in the datalake. In regards to data formats, this component offers the possibility of requesting the data according to JSON or CSV formats, which are useful for sending data to the web application, and for the developing of the machine learning algorithms, respectively.

In accordance with the user requirements, some endpoints were created in order to provide data to be represented in the format of charts, with the evolution of the sensorial data within an anomaly, or provide information regarding the usage of the vehicle, which is displayed in the client application.

The database used initially was based on SQLite, a simple application-embedded server-less system. However, for production, this solution could become a bottleneck, since it is not a tool suitable for a high-load of users, as it lacks scalability. For production, and since Google Cloud was being used, a cloud-managed MySQL database was created. Google Cloud promises high availability, dynamic arrangement, and all sorts of necessary management. Full advantage

was taken from all these conveniences. For better data visualization and quick analysis of possible misleading data, a web interface, based on web2py for the datalake was developed, which enables a convenient direct way of querying the data gathered.

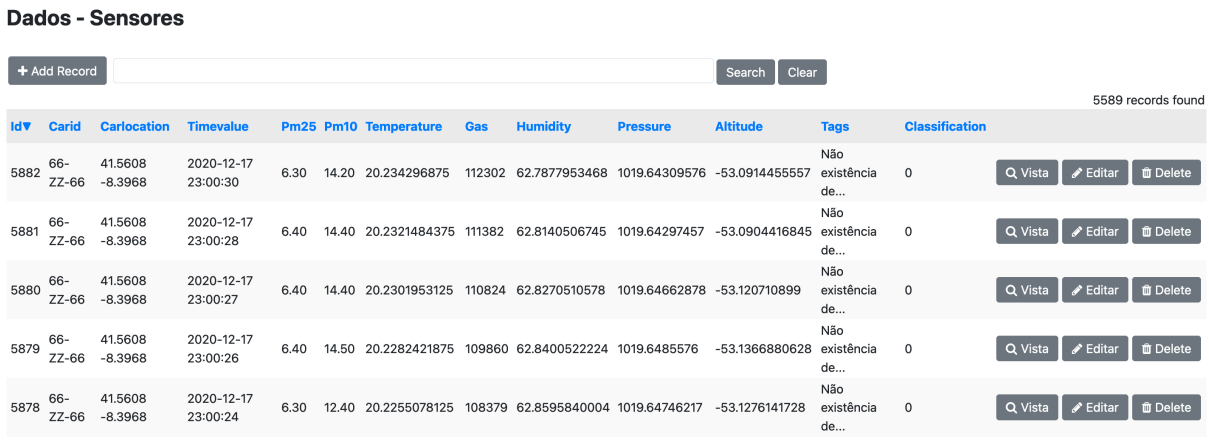

Figure 2.11: Web interface for the visualization of the training data

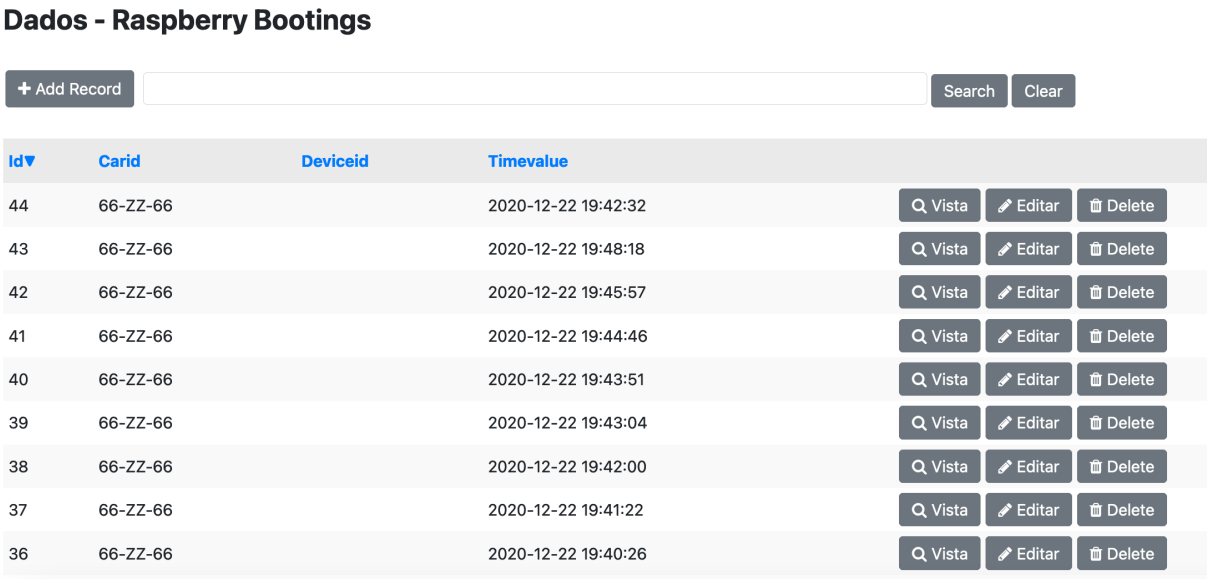

Figure 2.12: Boot times table

Figure 2.13: Example of a custom query inside the datalake

The following endpoints made all the required information available for other components to consume:

- /datalake/sensors/all/json
- /datalake/sensors/all/csv
- /datalake/sensors/last/json
- /datalake/raspberry/status/<carId>
- /datalake/history/<carId>/<timestamp>/<limit>
- /datalake/boots
- /datalake/boots/<carId>

2.3.1 Deploy

Since a docker container was used, it is only required to build the Dockerfile available within the root directory of the Datalake and then initialize it using the just created image. For the automation of both processes, a script was created and its execution is the only required step to get the datalake running.

```
$ ./run_datalake.sh
```

Once the container is created and running, the docker start and docker stop commands will start and stop the instance, respectively. The running container is then accessed through via port 80 from the exterior.

2.4 *Website Backend*

To support the existence of the website, it was necessary to implement a backend responsible for storing data about the users, their cars and the anomalies that have occurred. In addition to the website, the backend interacts with the FieldCrawler, the DataLake and the AlertAI Cloud.

From the FieldCrawler it receives the anomalous measurements that it detects.

The interaction with the DataLake consists of obtaining data about normal measurements that happened before or after an anomaly. These values could be stored directly on the backend but that would imply receiving, processing and storing all normal measurements, as we can't predict when an anomaly will occur. Given that the DataLake already stores all measurements, the small cost of obtaining some measurements from it occasionally is preferable.

A new requirement was introduced in the later stages of development. It specified providing the user with information about alternative classifications from the algorithms running in the cloud. To achieve this functionality, the backend requests the needed data from the AlertAI Cloud.

Storing these values directly in the backend would imply significant costs of receiving, processing, and storing.

For the implementation of the backend, the Spring Boot framework was used. This was chosen because it is familiar to the members of the sub-team responsible for the development of this component. Besides, it allows rapid and efficient development, facilitating tasks such as integration with the database and an embedded Tomcat server.

Since the relational model was adequate to represent the data, we opted to use MySQL.

To make the connection between Spring Boot and the MySQL Database, Spring Data JPA was used, which is easily integrated with Spring Boot. Spring Data JPA is an abstraction from JPA (Java Persistence API), facilitating access and interaction with MySQL data.

2.4.1 Data Model

The Data Model was auto-generated with the help of the Spring annotations. Each Java class annotated with **@Entity** represents a class with information that should be persisted, representing a table on the Database.

The following figure represents the Data Model.

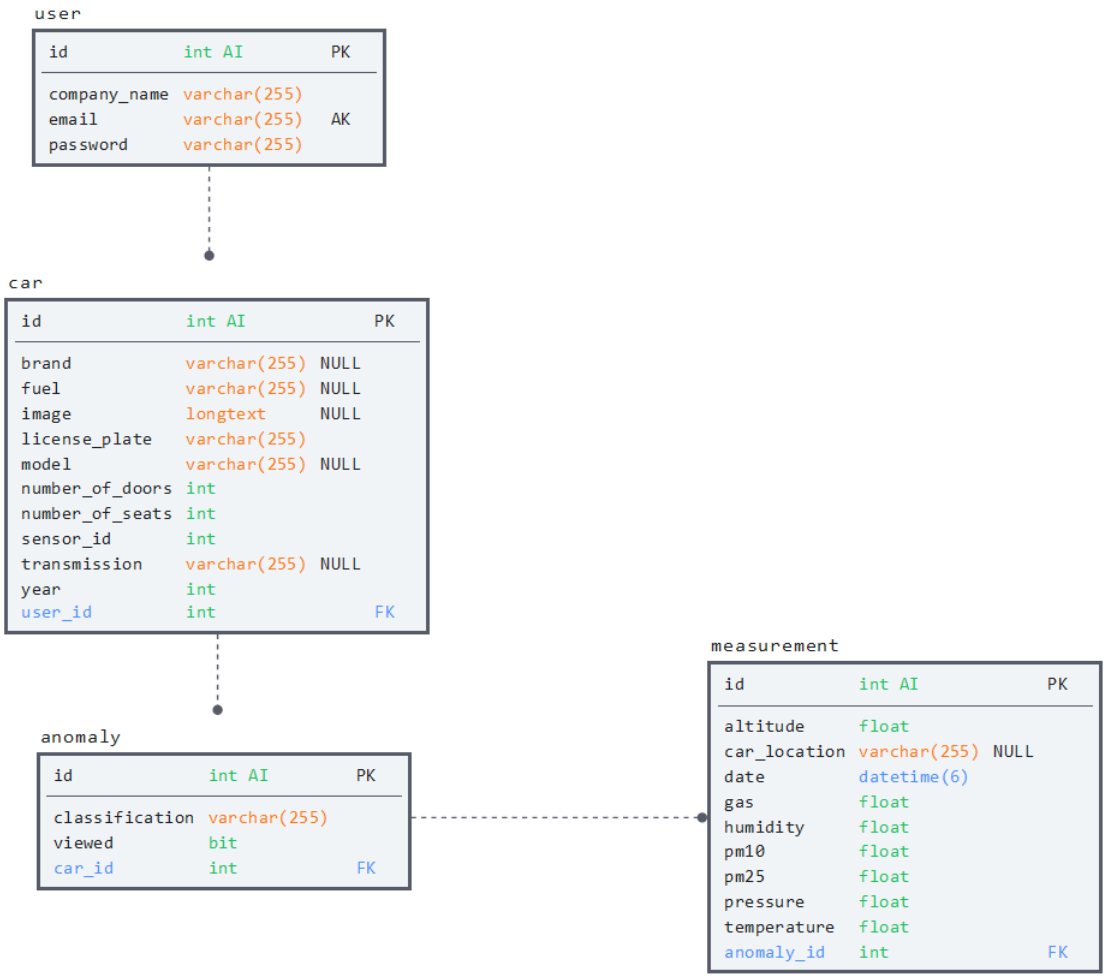

Figure 2.14: Data Model

From the Data Model we can conclude that:

- A user can have several cars registered;
- Each car can have anomalies associated with it;
- Each anomaly has one or more associated measurements, that contain all the information collected by the sensors;

2.4.2 REST API

To access all the endpoints in the application it is necessary to be authenticated. Otherwise, access is denied.

When a user successfully authenticates it is returned a token. This is a JWT token that allows the user to access all the features of the application and modify their data.

User

The application provides several endpoints for the user, such as authentication, registration, update, and deletion. A user can also retrieve information about himself.

Car

The application allows the user to add a new car to his fleet, update the information of an existing car or delete it.

It also allows access to the current state of the car.

Anomalous Events

There are several endpoints that provide access to the anomalous events, according to different parameters. These include car, date, or if the event was already viewed by the user or not.

2.4.3 Notifications

When a new anomaly occurs, the user should receive a notification acknowledging the occurrence of this new anomaly. WebSockets were used for this functionality.

WebSockets allow the connection between a web browser and a server, establishing a bi-directional, full-duplex, persistent connection, which allows both to send messages over the channel. The WebSocket connection stays open until the client or the server decides to close the connection.

For this functionality, the connection will only be unidirectional, with only the backend sending messages to the frontend.

Another way to implement the notifications would be using long-polling to check if there are new notifications which the user should be aware of. The problem with this solution is that it is more intense on the server, making lots of requests, which could increase the response time of the server. For that reason, it was decided that WebSockets was the best option.

When a new anomaly occurs, a new message is sent containing the notification to the Websocket with the topic **/queue/{companyName}**, where companyName is the name of the company of the car where the new anomaly occurred.

2.4.4 Deployment

For the deployment, it was used Ansible, a tool that allows the provisioning, configuration management, and deployment of applications automatically, without the need to do everything manually every time there is a change in the applications.

This tool is composed of a set of files. For this deployment, it was used a playbook and a role for each virtual machine. In the playbook, it is declared what virtual machines, and respective configurations, are going to be created in the cloud, as well as the connections between virtual machines.

In each role, it is specified what each instance needs to do to be up and running.

Mostly each role installs the necessary software, sets permissions, builds a docker image concerning the service, and runs it in a container, making it available over port 80 of the virtual machine.

With this, Ansible is in charge of creating four machines in the cloud, if they don't already exist. One of the machines is for the database, another for the load balancer, and two for the backend, for performance and availability reasons. This can be seen in the deployment diagram in the figure 3.1.

The load balancer, NGINX in this case, redirects the orders it receives towards one of the backends, depending on the load each one is processing. An exception is made for WebSocket connections and messages indicating the occurrence of a new anomaly. They are both redirected to the same backend instance, as the arrival of such a message is what triggers a notification.

The backend also makes available **Swagger**, an interface that describes the existing *endpoints* and allows interaction with the application. This is very useful in terms of development and debugging, not only to the team that developed the backend but also to the teams responsible for other components, as it describes the REST API to be used.

2.5 *Frontend App*

This web client was developed with Vue, a JavaScript framework, and Vuetify, a UI framework. This choice of technology was based on five main factors:

- Learning curve;
- Speed of development;
- Speed of the app;

- Code structure;
- Documentation.

With Vue and the help of its intuitive and complete documentation, a developer can easily and quickly start building an application and is, therefore, a good choice for MVPs and small applications. The ease of learning makes it accessible to inexperienced programmers and also allows more experienced programmers to quickly enter the project. [6]

The structure of the code is also a great advantage in a project since the cleaner it is, the more maintainable it is. In Vue, there is a separation of responsibilities between Structure (HTML), Functionality (JavaScript), and Style (CSS), which largely contributes to this factor.

Since Vuetify is built on Vue, it is highly integrated with the framework. Due to the speed of styling compared to other tools or pure CSS, Vuetify was the most natural choice for this project.

2.5.1 Live notifications

Whenever an event is detected by the sensors, it is shown in the SPA as an alert so that the user can be immediately aware of the state of the vehicle which triggered it.

WebSockets are needed to implement this feature, however, not all versions of browsers support the technology. To get around this issue, an emulator was used: SockJS. But in order to correctly communicate with the server-side, due to the technology in which it is implemented, support for the STOMP protocol is required. This obliges the use of a client-side technology that supports it, in our case, webstomp-client.

Implemented correctly, these technologies provide a live smooth connection to the back-end services which in turn receive the necessary information about the detected events.

2.5.2 Authentication

All customers wishing to use the application will need an active account. To obtain it, the application provides access forms using the back-end and JWT to ensure the most secure and user-friendly experience.

2.6 AlertAI

RideCare takes advantage of real-time classification algorithms, based on classic Machine Learning methods to detect anomalous situations inside shared vehicles.

From labelled and unlabelled data, several algorithms from Supervised and Unsupervised Learning paradigms were tested and insights were taken during that extensive exploratory work. Our best algorithm achieved more than 97% accuracy on all predictions made, for all scenarios considered. These results were obtained under controlled capture scenarios, taking into account real environments inside vehicles.

Additionally, other supervised learning algorithms, which performed well during our research process, are hosted by a cloud application -AlertAI Cloud, allowing alternative predictions for the raw data collections.

The whole process mentioned above can be found in another document, called "Technical Report - AlertAI", detailing all stages of research and all informations about its development.

3 Final infrastrucutre

Once the system's components were identified, its specification was made through modeling techniques. However, as the development was pursued, the architecture suffered changes. The end result of the system's architecture features high availability mechanisms and asynchronous communication for task completion assurance, custom entry points, load balancing, and more. The final infrastructure implemented is specified in the following deployment diagram:

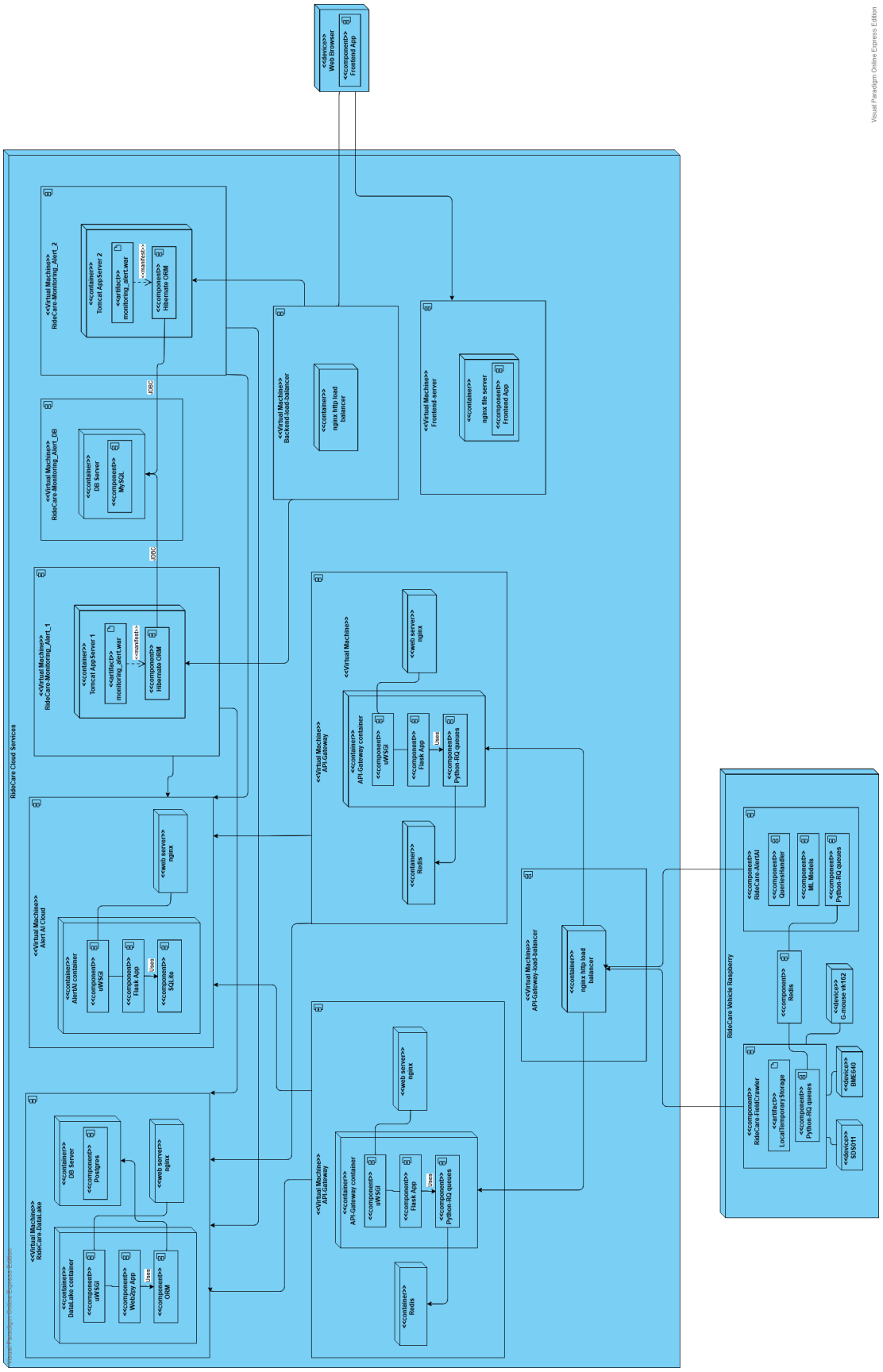

Figure 3.1: Diagrama de deployment

3.0.1 Cloud-based deployment

During the development phase, functionality tests were made using local machines. However, since the components consume data from each other, a transition to a cloud-based infrastructure was quickly necessary. By using Google Cloud's compute engine, an Infrastructure-as-a-service platform that provides full control of virtual machines hosted on the cloud, the components could be accessed from anywhere. By defining a static external IP, data collected by the in-vehicle system could finally be sent, in real-time, to the datalake, instead of saving it locally.

Then, all the developed components were hosted on the cloud and collaboration started to take place. The FieldCrawler can now send the gathered data to the datalake, the AlertAI can send the evaluated anomalies to the web backend and AlertAI cloud for alternative classifications, and the backend can request for the data present in the datalake AlertAI cloud.

Once the core collaboration was defined, special attention to the quality attributes was given. Regarding security matters, an API Gateway was created with the objective of having a single controlled entry point for the systems present in the vehicles. Moreover, it provided the agility to implement an authentication system that can detect fraud data sent (see section 2.2.1). However, since it is the entry point for every single vehicle system, it constitutes a critical single point of failure and a possible bottleneck. In order to address these issues, a replica of the component was introduced and both copies were placed behind a load balancer. Since every request is stateless and self-contained, the replication did not come with overheads. This way, a single point of failure is eliminated and the load is now distributed across two instances. The same principle was introduced into the web backend, which was also duplicated and placed behind a load balancer. Now, the client application is more safeguarded from possible downtime, since it now implies that two machines would have to fail simultaneously, contributing to a high percentage of overall uptime. Further contributing to high availability, a task queuing system was implemented in three of the components, the FieldCrawler, AlertAI, and API Gateway (as detailed in previous sections). This system assures that a given task is executed, even if only eventually, providing assurance that the task will be fulfilled, which is fundamental for this particular use case, in which anomalous data must be delivered. Regarding one of the most critical components, given that Google Cloud was being used, their built-in high available database options were used for the datalake database. Google cloud promises full management of the instance, automatic database provisioning, storage capacity management, among other relevant aspects. Full advantage of these advantages was taken.

Having all these aspects into consideration, the final cloud-based infrastructure is composed of the following virtual machine instances.

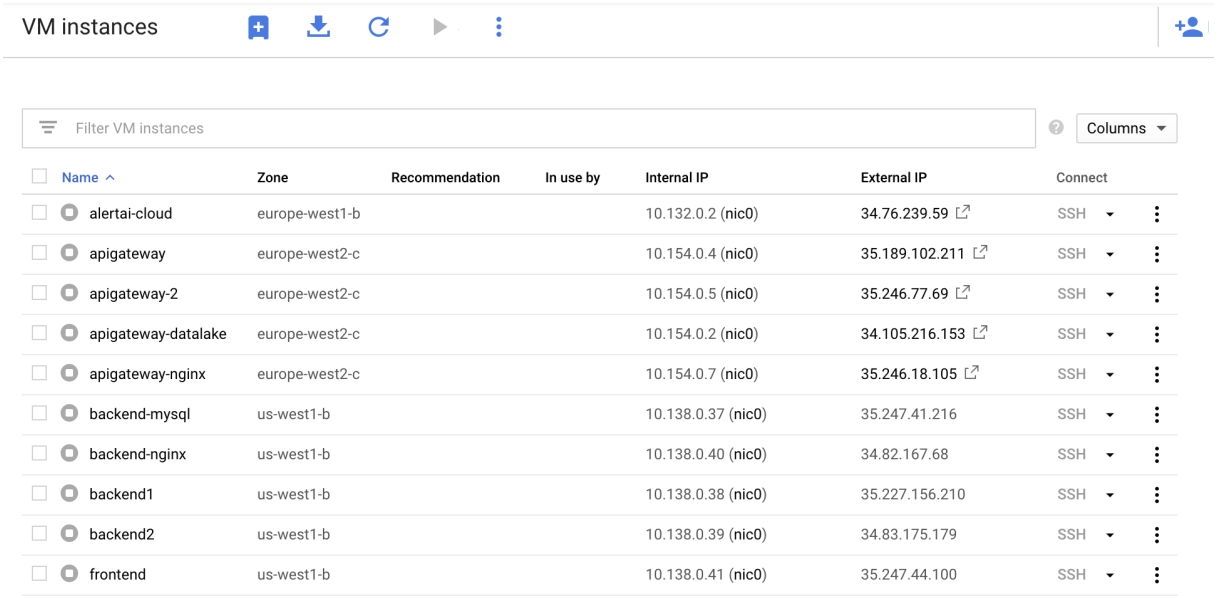

Figure 3.2: VM instances created inside Google Cloud

Inside each one of the virtual machines, docker containers were used in order to promote portability and enable a simpler deployment pipeline. By using docker, all the configurations and dependencies are packed together, assuring interoperability in any host. Moreover, each service has a custom script that builds and starts the necessary containers. Additionally, every machine has a system service that automatically starts the required dependencies on boot, so all it is needed to start the system is to turn on the instances.

4 Interface

4.1 *Mockups*

As a result of an initial prototyping phase, mockups of relatively high fidelity were developed in order to analyze and infer the suitability of a possible solution of what will be the end-user interface. AdobeXD, an advanced UI/UX design and prototyping tool for applications, was used for this.

During the course of the project, the mockups underwent some fine-tuning as the changes were being decided, both by the mentor and the development team. The initial mockups developed are presented below:

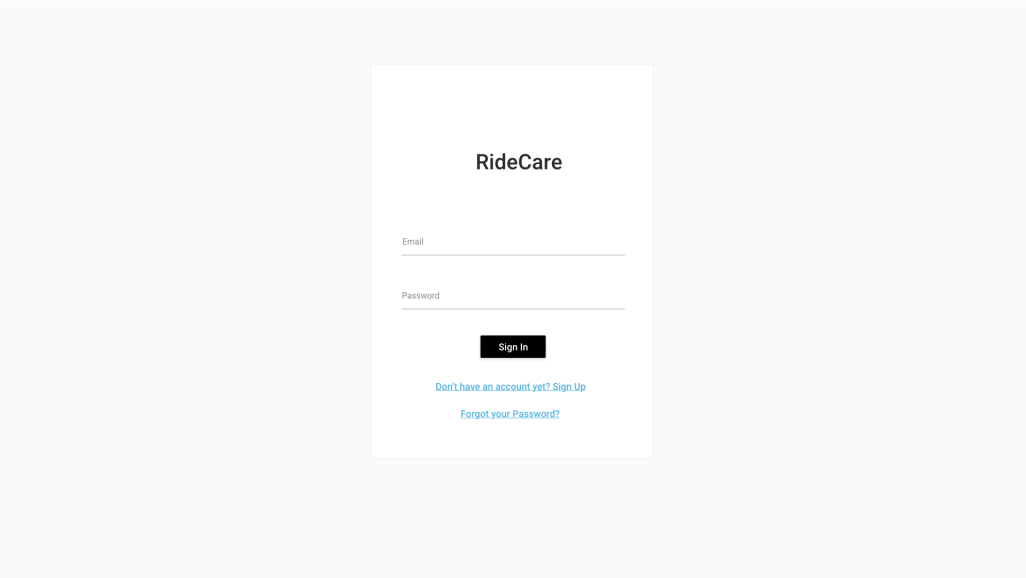

Figure 4.1: Mockup - Sign In

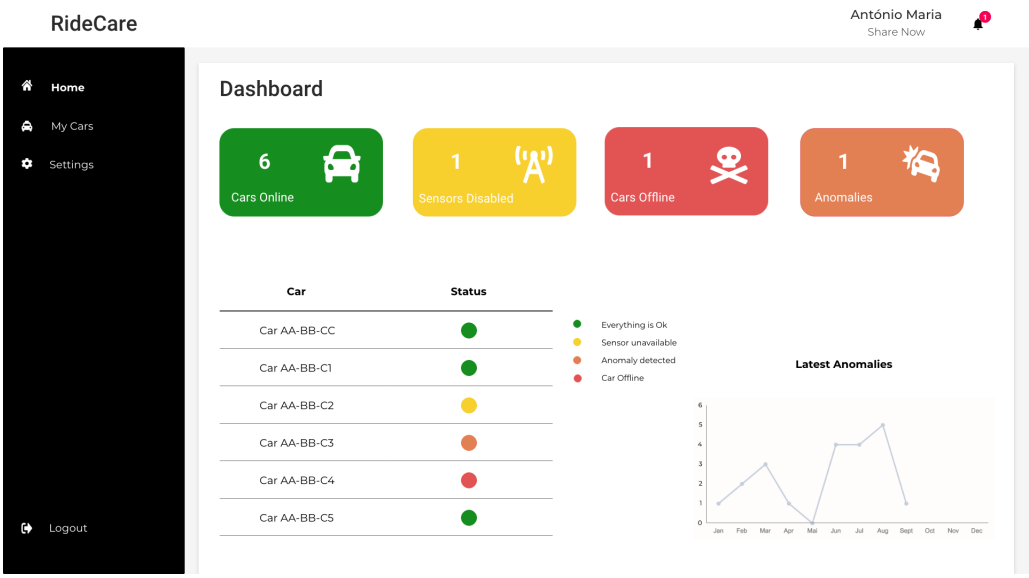

Figure 4.2: Mockup - Dashboard

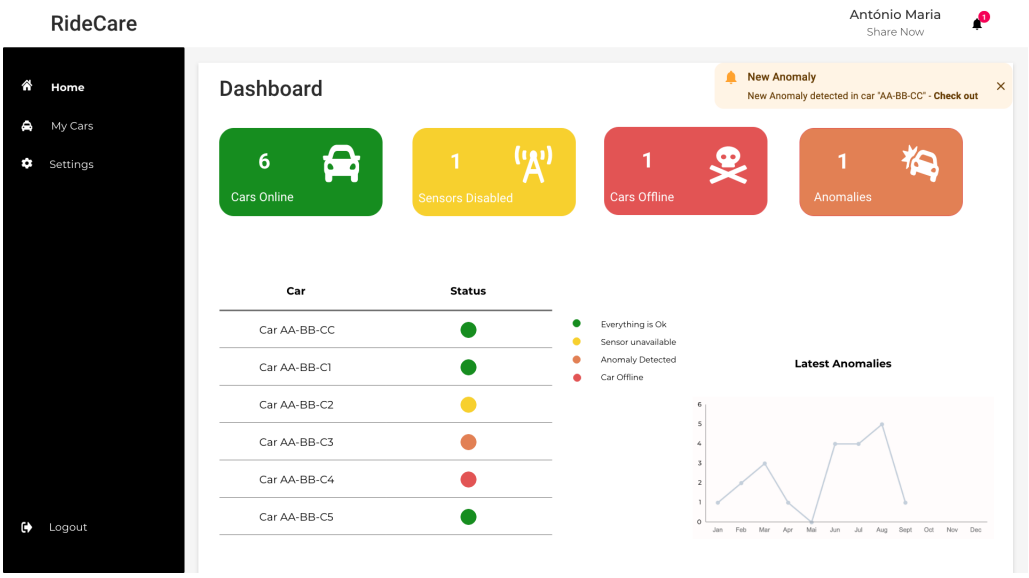

Figure 4.3: Mockup - Dashboard with anomaly notification

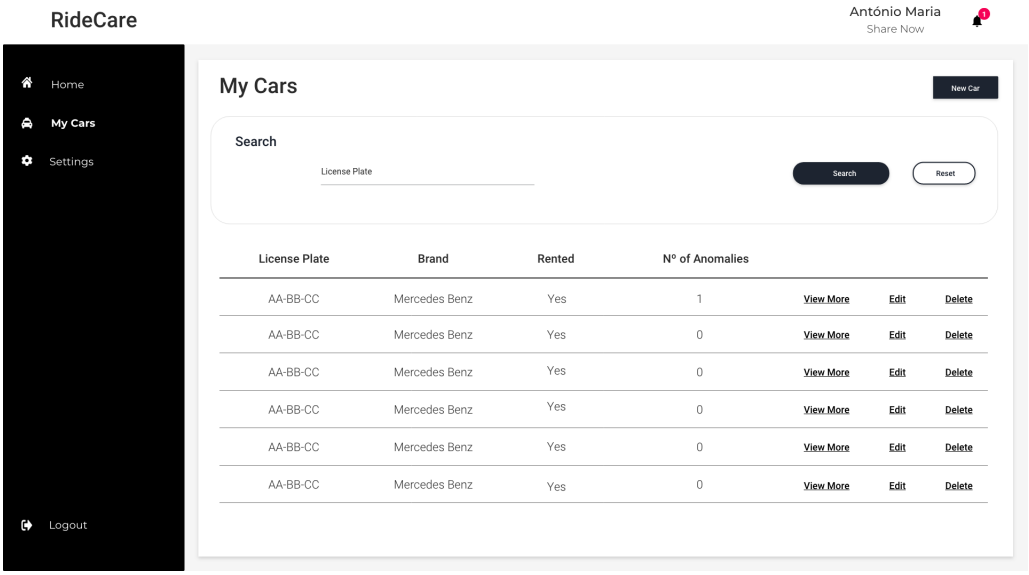

Figure 4.4: Mockup - All user vehicles

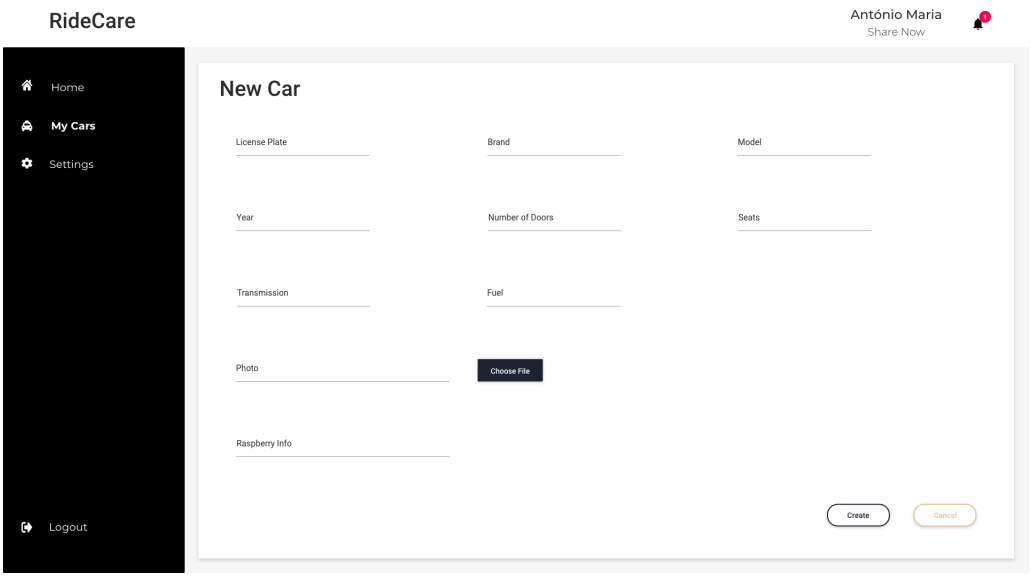

Figure 4.5: Mockup - Registration of a new vehicle in the application

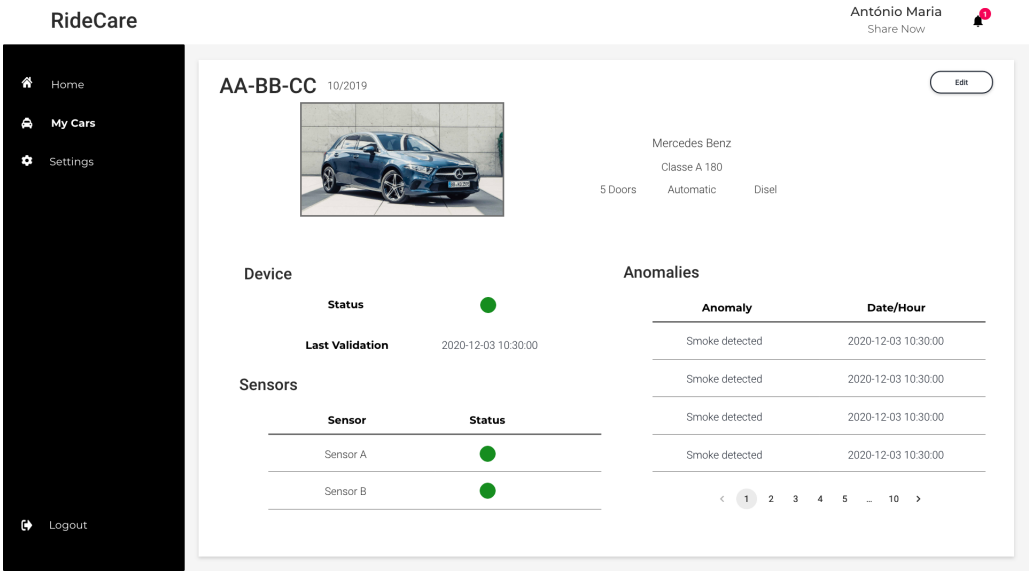

Figure 4.6: Mockup - Details of a user's vehicle

4.2 *Single Page Application*

The final result of the Single Page Application will be shown in the following figures:

4.2.1 Authentication

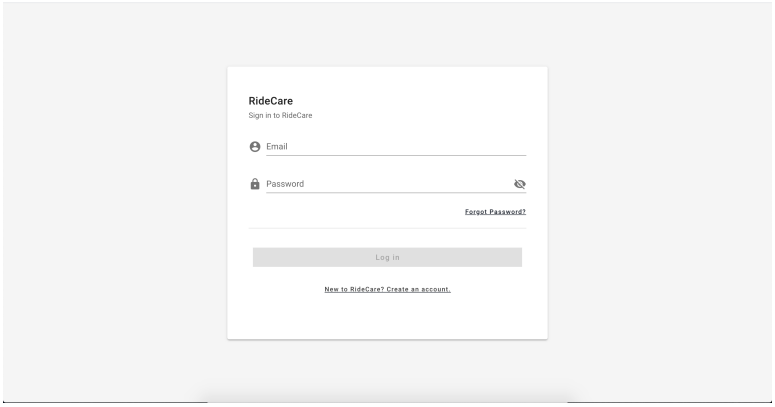

Figure 4.7: Sign In page

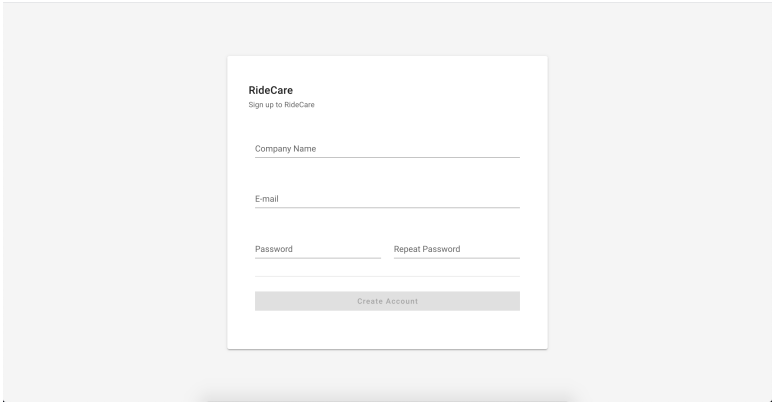

Figure 4.8: Sign Up page

4.2.2 Dashboard

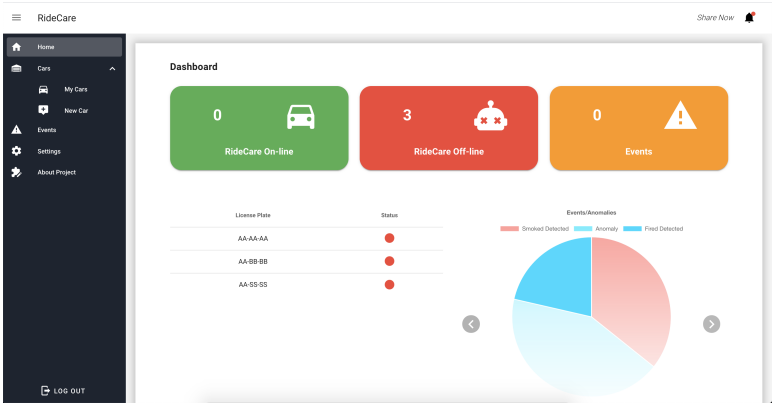

Figure 4.9: Dashboard view with a Pie chart showing the amount of events detected

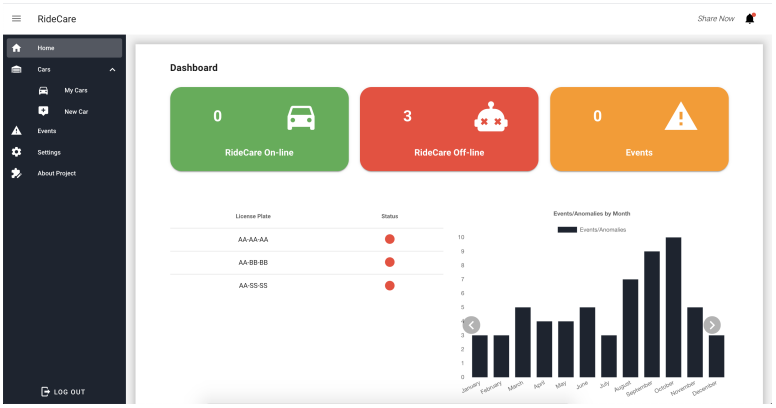

Figure 4.10: Dashboard view with a Bar graph showing the amount of events detected per month during the current year

4.2.3 Vehicles

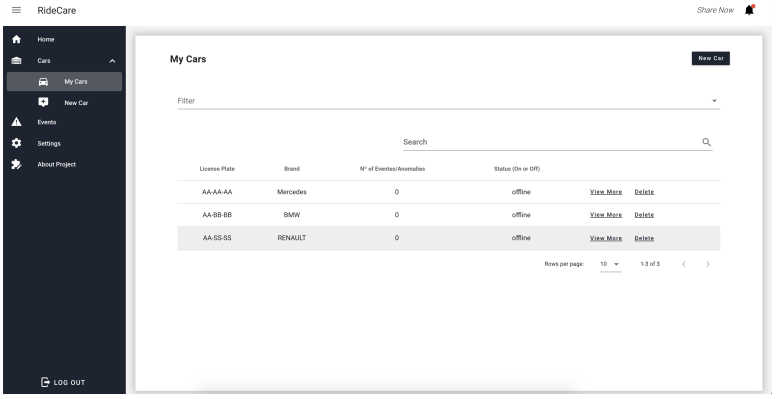

Figure 4.11: Overview of all user vehicles

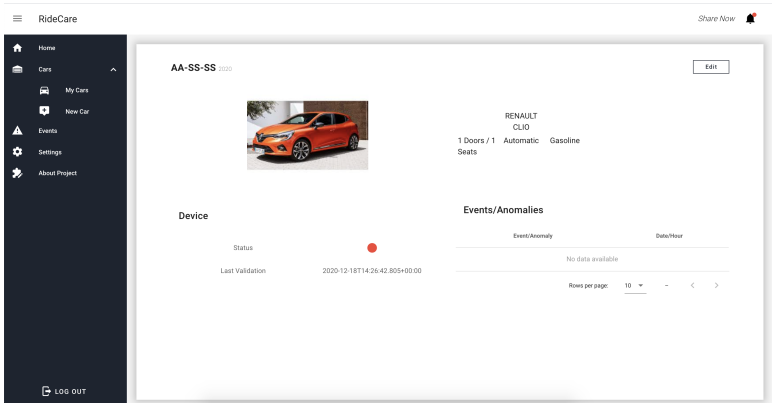

Figure 4.12: Details of a user vehicle

Registration of a new vehicle

For the registration of a new vehicle in the application several fields are needed, which are divided by steps, in order to facilitate data entry.

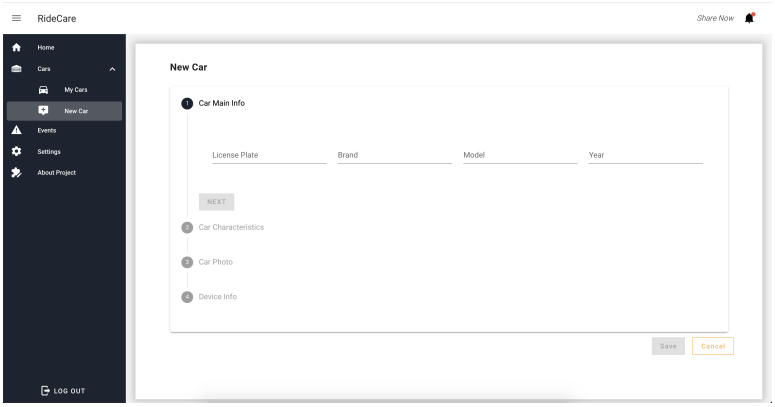

Figure 4.13: General vehicle information

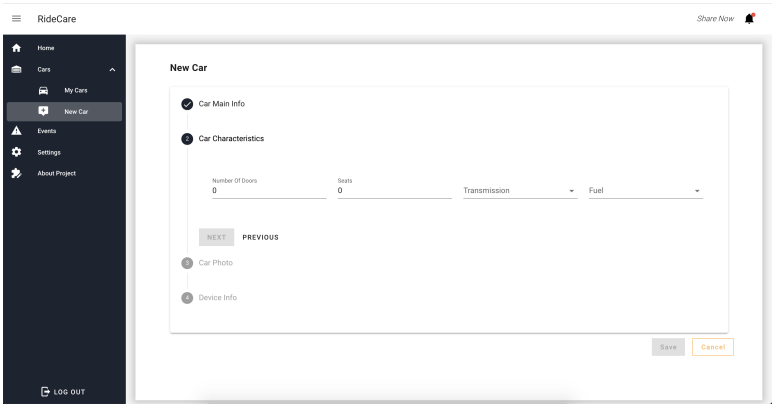

Figure 4.14: Additional vehicle features

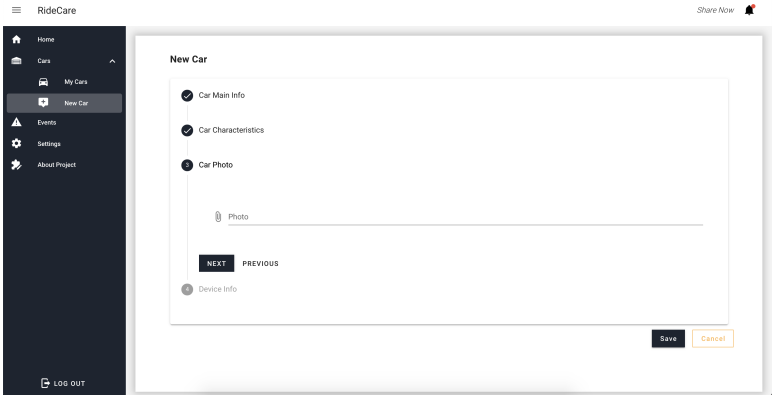

Figure 4.15: Vehicle photo - Optional

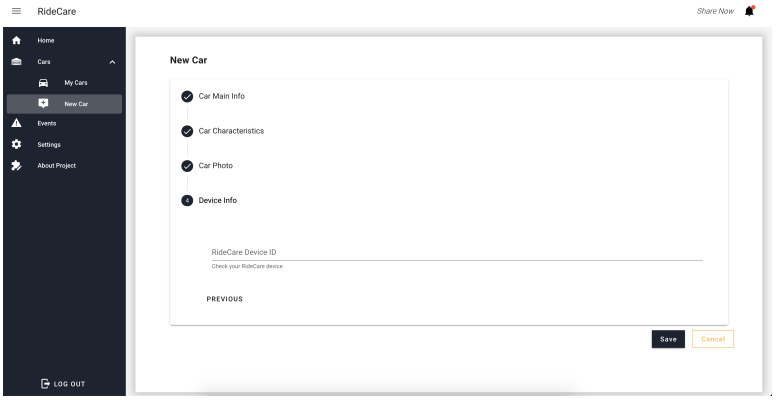

Figure 4.16: RideCare device key

4.2.4 Events / Anomalies

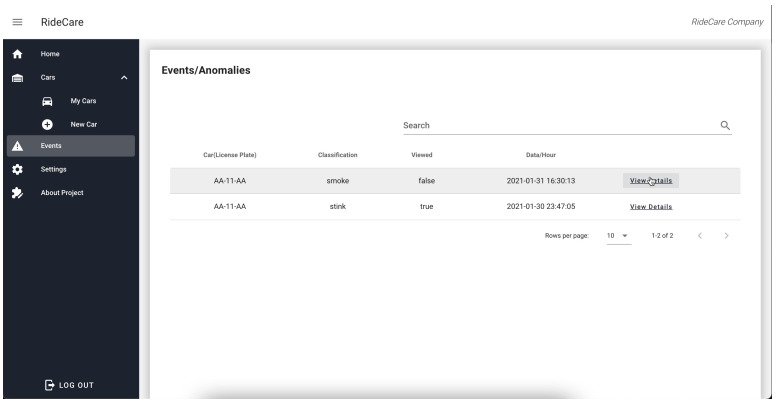

Figure 4.17: List of all events detected in the user's vehicles

Figure 4.18: Details of an anomaly

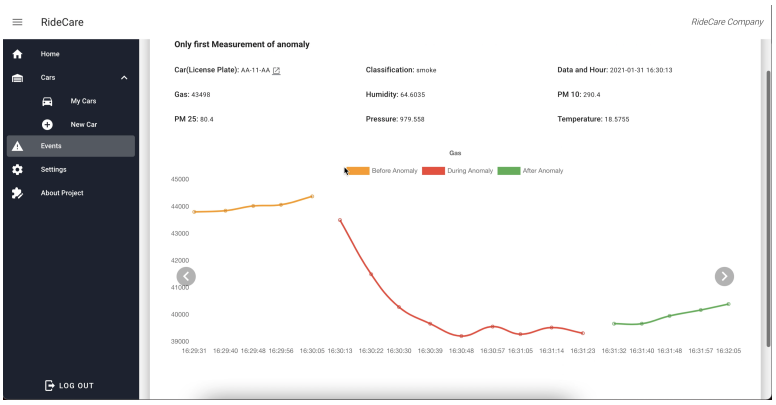

Figure 4.19: Details of an anomaly - Graphics

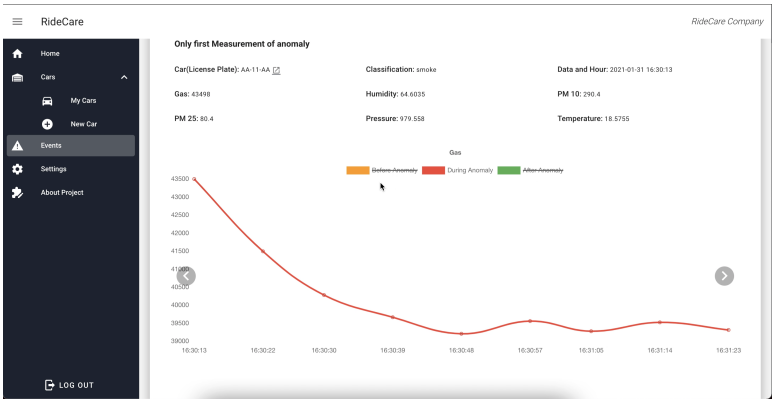

Figure 4.20: Details of an anomaly - Graphics Detail (During Anomaly)

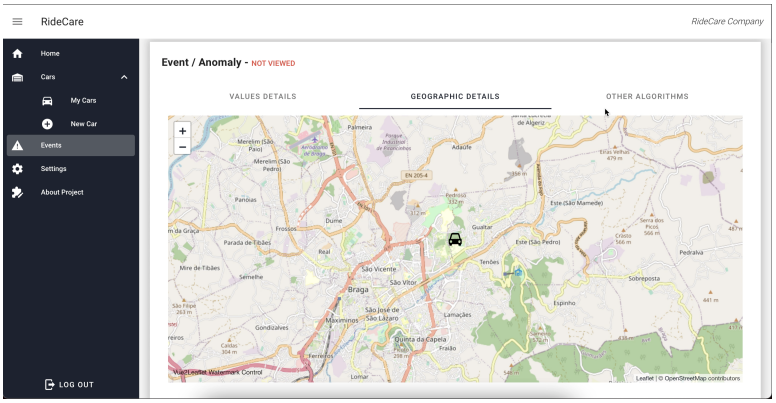

Figure 4.21: Location of where the anomaly took place

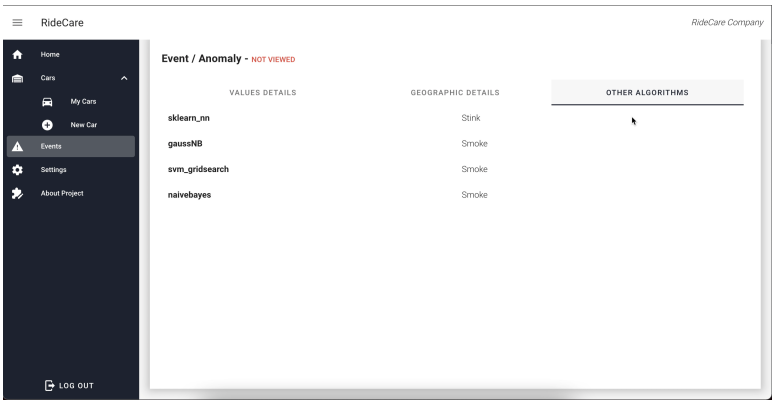

Figure 4.22: Other Classifications of an anomaly

4.2.5 Settings

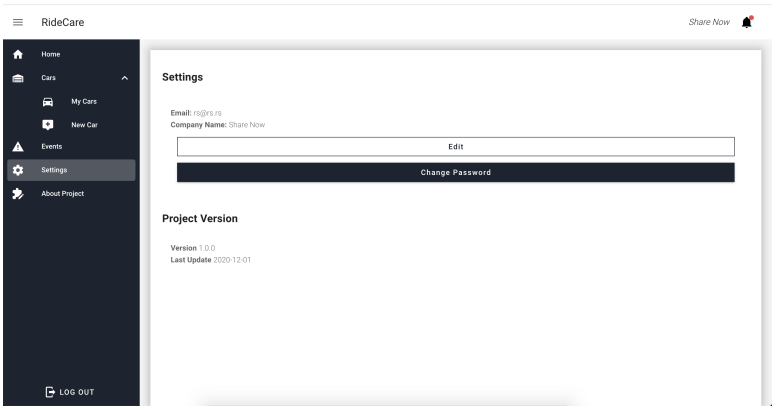

Figure 4.23: User profile

4.2.6 About the project

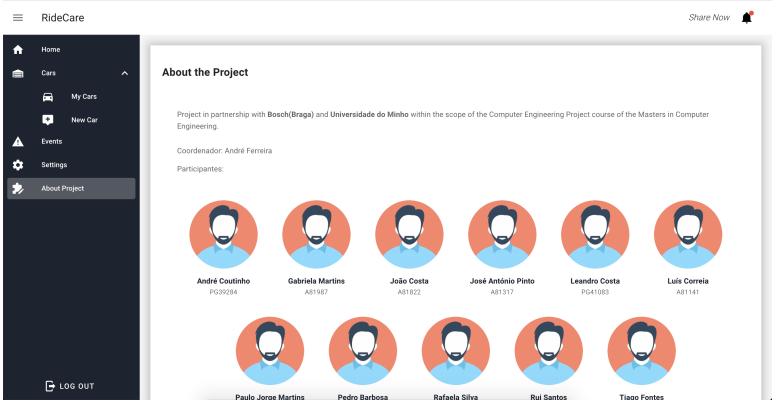

Figure 4.24: Page dedicated to the project and the development team

5 Conclusion

Throughout the conceptualization process and the implementation of the various components that make up the entire system, the group felt the need to make key decisions in terms of the communication, architecture, and structure of the various existing subsystems.

In this sense, the requirements definition was an important starting point, as well as structuring of the mechanisms which enable its implementation.

After this step, the selected anomalies to take into consideration for the detection were delimited. Currently, the system is capable of detecting 3 different situations: normal, smoke, and stink (presence of bad smells). Preceding any further development, the sensors in possession were explored, integrated with the microcomputers and the functionality was tested, in order to be able to retrieve and data from them.

Subsequently, the FieldCrawler 2.1 and Datalake 2.3 were implemented. These components enabled both the gathering and storage of the data and suffered multiple upgrades throughout the development, whose functionality was extended. Then, a data collection plan was developed for each one of the defined anomalies. Although some pandemic-related difficulties have arisen, the group was able to overcome the challenges and a considerable amount of samples were gathered in real scenarios. More than 10 000 samples were gathered from several different scenarios. This data enabled the exploratory study which followed.

Therefore, extensive work in terms of statistical analysis of the gathered data took place. Creating a context for each of the scenarios (anomalous or non-anomalous), in association with intense study of the values was relevant to understand the influence that several factors had. These factors, such as the intensity of the smoke, its distance, or even the sensibility of the

sensors can have an impact on the gathered data and that margin was analyzed and taken into consideration. Following this, a whole implementation pipeline of Machine Learning algorithms was developed with the detection and classification of anomalies insight. This process included data treatment, the development of multiple algorithms, as well as its optimization. Since it was known beforehand that the AlertAI would dictate the quality of the whole system, a significant amount of the team's attention was focused on this component and its development.

Given that the team was split in smaller work groups, multiple other components were being developed in parallel. A web backend and a client application were developed in order to provide a form for users to interact with the data and fulfill the ultimate purpose of the whole system: provide a monitoring system for a fleet of vehicles.

A complex flow of information starts every time a new sample is collected. Due to the high user load, additional components, such as an API gateway were developed in order to assure non-functional requirements.

Being a distributed system, high availability is crucial in this context, since an error during the communication between the components could possibly result in loss of data, and subsequently not detecting an anomalous situation. For correct function assurance, several mechanisms were implemented, such as the introduction of queuing systems in the vehicle system, which handles situations where there's no network coverage, or replication or servers, which improve the overall availability of the system. Both functionalities and quality attributes were valued within the context of this project.

In regards to possible future work, the group could improve some aspects in each component, by providing a more robust testing environment or enabling a more automated deployment pipeline. Nonetheless, the system works as a whole and all its promised functionalities are present in the final product.

Bibliography

[1] Number of car sharing users worldwide from 2006 to 2025 [Online]. Disponível em https://ai-guru.de/deep-reinforcement-learning-and-doom/. [Acedido em 5 de novembro de 2020].

[2] Volere Requirements Specification https://www.volere.org/templates/volere-requirements-specification-template/ [Acedido em 9 de novembro de 2020].

[3] Sathya, R. & Abraham, Annamma. (2013). Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification. International Journal of Advanced Research in Artificial Intelligence. 2. 10.14569/IJARAI.2013.020206.

[4] A. Singh, N. Thakur and A. Sharma, "A review of supervised machine learning algorithms," 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, 2016, pp. 1310-1315.

[5] Khanam, Memoona & Mahboob, Tahira & Imtiaz, Warda & Ghafoor, Humaraia & Sehar, Rabeea. (2015). A Survey on Unsupervised Machine Learning Algorithms for Automation, Classification and Maintenance. International Journal of Computer Applications. 119. 34-39. 10.5120/21131-4058.

[6] React vs Vue: What is the best choice for 2020? [Online] https://www.mindk.com/blog/react-vs-vue/ [Accessed on 12 November 2020]