Politecnico di Torino

Master of Science in Computer Engineering

Master Thesis

Rethinking Automotive Software Development: Exploring Software Defined Vehicle and its potential

Supervisors
prof. Danilo Bazzanella
dott.sa Piera Limonet

Candidate
Lorenzo Sciara

Academic Year 2023-2024

Summary

This thesis project delves into the analysis of contemporary connected vehicle platforms, focusing on the benefits and challenges associated with these advanced solutions and emphasising aspects of safety and flexibility. A key trend in the current automotive sector is the prospect of transforming the car from a hardware-focused product to a software-driven device. The technology of choice for leading software development and production companies driving this change is the Software Defined Vehicle (SDV).

The primary objective of the thesis is to apply this paradigm to the development of a simulator for a vehicle control unit responsible for collecting telemetric data from the vehicle. The implementation of the simulator involves an in-depth analysis of the drawbacks of the automotive software production industry and the advantages of the Software Defined Vehicle solution. The simulator implementation also includes the creation of a scaled-down version of a connected vehicle platform, storage infrastructure and example application.

Using the Amazon Web Services (AWS), an environment in the cloud is established for the development of the necessary software for the operation of the vehicle control unit. Development of the vehicle control unit simulator is carried out, including client connectivity to interact with the cloud platform, telemetry generation, logic for remote operations, and optional applications. The final phase involves testing the simulator on compatible hardware to validate its functionality and performance.

The successful completion of this project in collaboration with Storm Reply, not only highlights the potential of the software-defined vehicle paradigm as a leading force in the future of the automotive sector, but also explores the economic, safety and security benefits associated with its adoption, paving the way for significant progress in the field and ensuring an advanced and safe end-user experience.

Contents

List of Figures

List of Tables

Listings

Acronyms

3PAO	Third-Party Assessment Organization 37
ABS	Anti-lock Braking System 74
AD	Automated Driving 25
ADAS	Advanced Driver Assistance Systems 25
AICPA	American Institute of Certified Public Accountants 37
AM	Application Messages 26, 27
Amazon EC2	Amazon Elastic Container Cloud 46–48, 62, 64
Amazon ECR	Amazon Elastic Container Registry 46, 48, 66, 67
Amazon S3	Amazon Simple Storage Service 44, 45, 66, 70
Amazon VPC	Amazon Virtual Private Cloud 47, 62
AMI	Amazon Machine Images 46, 47, 62
API	Application Programming Interface 62, 64, 68–71
APN	AWS Partner Network 36
AWS	Amazon Web Services 18–20, 24, 26, 30, 32–47, 49–51, 59, 72, 73, 76, 78
AWS CDK	Cloud Development Kit 41
AWS CLI	AWS Command Line Interface 40
AWS IAM	AWS Identity and Access Management 42, 46
AZ	Availability Zones 19
CAIQ	Consensus Assessments Initiative Questionnaire 39
CDK	Cloud Development Kit 33, 59–61, 64, 71, 74
CSA	Cloud Security Alliance 39
ECU	Electronic Control Unit 41
FedRAMP	Federal Risk and Authorization Management Program 37
FISMA	Federal Information Security Management Act 37
FOTA	Firmware-Over-The-Air 24
GDPR	General Data Protection Regulation 39
HIPAA	Health Insurance Portability and Accountability Act 39

HITRUST CSF	Health Information Trust Alliance Common Security Framework 39
HTTP	Hypertext Transfer Protocol 62, 69
IaaS	Infrastructure as a Service 33
ID	Identification 60
IEC	International Electrotechnical Commission 38
IoT	Internet of Things 16, 26, 33, 36, 40–42, 45, 49, 59
IP	Internet Protocol 52, 63, 64, 74
ISO	International Organization for Standardization 17, 31, 38, 39
IT	Information Technology 17, 18, 32–38, 79
IVI	In-Vehicle Infotainment 25
MCU	MicroController Unit 25
MQTT	Message Queuing Telemetry Transport 25–27, 41, 42, 51, 59, 77
NIST	National Institute of Standards and Technology 17, 32, 34, 37, 39
OTA	Over-The-Air 25, 26, 49, 50, 52, 70
PaaS	Platform as a Service 33
PC	Personal Computer 40, 51
PCI	Payment Card Industry 39
PII	Personally Identifiable Information 38
PoC	Proof of Concept 40, 41, 48, 49, 74, 78, 79
QMS	Quality management System 38
QoS	Quality of Service 26
SaaS	Software as a Service 33, 34
SDK	Software Development Kit 40, 41, 46
SDV	Software Defined Vehicle 14, 16, 17, 20, 22, 24, 25, 27–32, 40, 49–51, 76, 78, 79
SOAFEE	Scalable Open Architecture for Embedded Edge 20, 30, 31
SOC	System and Organization Controls 37
SQL	Structured Query Language 71
SRM	Shared Responsibility Matrix 39
SSH	Secure Socket Shell 47
STAR	Security, Trust, Assurance, and Risk 39
TCU	Telematic Control Unit 16, 47, 48, 50, 53, 54, 56–59, 63, 65, 70–77, 79

TN Topic Name 26

URL Uniform Resource Locator 53, 69

Chapter 1

Introduction

"We really designed the Model S to be a very sophisticated computer on wheels. We view this the same as updating your phone or your laptop. Tesla is a software company as much as it is a hardware company. A huge part of what Tesla is, is a Silicon Valley software company." - Elon Musk [14]

With these words, Elon Musk, the visionary entrepreneur behind many of the most innovative companies in today's business landscape and co-founder of one of the most, progressive automotive companies of our time, Tesla, highlighted how the automotive industry is changing dramatically over time, transforming today's cars and vehicles from objects where the fundamental part consists of mechanical components to ones where the main focus lies in the simplicity of hardware and the innovation of software.

This shift in paradigm, which is now a reality in the automotive industry, requires significant effort, especially from a security perspective. While a cyber vulnerability in a traditional device like a laptop or smartphone may result in data loss, vulnerabilities in a vehicle's computer system, where software is a fundamental element, can have tragic and even life-threatening consequences. For this reason, addressing security from the design stage is one of the primary objective of this paper.

To understand and address the Software Defined Vehicle (SDV), the latest form of software integration in the automotive industry, it is important to examine the industry's dynamics and software production processes. This introduction provides an overview of the automotive context in which the project is located. It then details the role of the project partner company, a leader in software consulting and development and a partner of major automotive companies. Finally, in conclusion of this chapter, the thesis's key objectives and the practical project that will support this work are described, providing an overview of the entire thesis project and a description of the practical validation demo that was executed on a physical device using the various elements of the built project itself.

1.1 Automotive Context

The automotive industry has stood out for decades as a continuously growing sector, playing a significant role both as an employer for millions of people and as an investor in the research and development of cutting-edge technologies in many fields, including mechanics, materials, and software. Thanks to the presence of the largest automotive companies across Europe, there is a great deal of knowledge in this sector, which represents one of the most crucial areas for the European Union's economy. As can be seen from the table below 1.1, the production of total vehicles worldwide has been continuously growing, with the exception of two periods: following the financial crisis of 2007-2008 and following the pandemic of 2020, both events having a very strong impact on the entire global economy and which have had effects on many sectors. In any case, it can be noted that automotive production has resumed strong growth in the last two years.

Table 1.1. World automobile production in million vehicles [11]

Year	Production (millions)	Change
2007	73 266 061	+ 05.80 %
2008	70 520 493	- 03.70 %
2009	61 791 868	- 12.40 %
2010	77 857 705	+ 26.00 %
2011	79 989 155	+ 03.10 %
2012	84 141 209	+ 05.30 %
2013	87 300 115	+ 03.70 %
2014	89 747 430	+ 02.60 %
2015	90 086 346	+ 00.40 %
2016	94 976 569	+ 04.50 %
2017	97 302 534	+ 02.36 %
2018	95 634 593	- 01.71 %
2019	91 786 861	- 05.20 %
2020	77 621 582	- 16.00 %
2021	80 145 988	+ 03.25 %
2022	85 016 728	+ 06.08 %
2023	93 546 599	+ 10,03 %

In the ever-expanding landscape of the automotive industry, a new frontier has been added in recent years, that is the software development, which first arrived in the luxury car markets as optional and marginally relevant systems in the vehicle, and then spread to all types of vehicles. Today, the current challenges for automotive companies extend far beyond the traditional areas of mechanical or material engineering to reach a total and fundamental involvement in the study and innovation of software and hardware components for vehicle construction.

A look at the intricate network of different components in today's cars, as shown in Figure 1.1, reveals that a vehicle is actually composed of a mosaic of dozens of different systems, which in turn are composed of dozens of processors that interact

with each other at different levels. For these reasons, today's cars have have earned the moniker of *'Computers on Wheels'*.

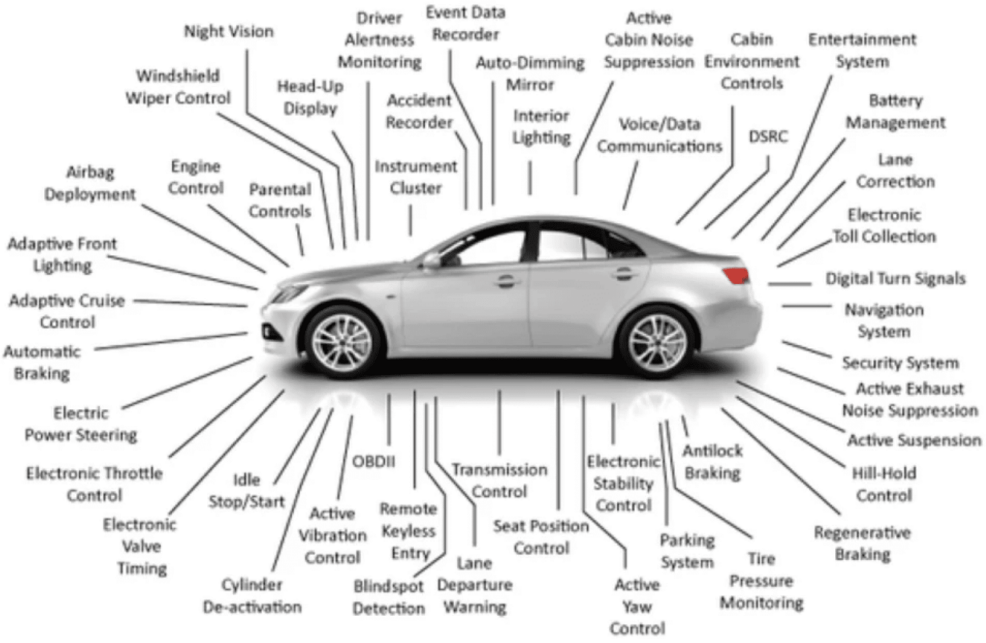

Figure 1.1. An incomplete overview of computers in a modern car [1]

This paradigm shift has been driven in part by the introduction of autonomous driving. To ensure maximum safety, a car must be equipped with dozens of sensors that can constantly collect data about what is happening to the vehicle and its surroundings. The number of these telemetry devices, which are nothing more than specialized Internet of Things (IoT) devices for the automotive world, also known as Telematic Control Unit (TCU), is expected to grow as autonomous driving technologies advance. In addition to data collection, another key issue is the data analysis of information collected from the devices. Modern vehicles are caught between low-power systems for maximum vehicle efficiency and high-performance systems for analyzing the collected data and making excellent decisions in a short time.

However, the proliferation of processors within vehicles, orchestrating communication to manage diverse components, presents a formidable challenge; each component often integrates a processor with unique logics, diverging from the logics embedded in processors of other components. Complicating matters further, these components are frequently supplied by companies with proprietary management logics, not readily accessible to the automotive companies themselves.

In addressing this intricate scenario, the transformative concept of a SDV comes to the forefront. Defined as "any vehicle that manages its operations, adds functionality, and enables new features primarily or entirely through software" [15], the notion of SDV, with all the associated technologies, offers a comprehensive solution to the challenges posed by the intricate interplay of software and hardware in modern vehicles.

One of the main benefits of this innovation in the automotive industry is the ability to have easily manageable systems. In the past, due to their high level of specialization for performance and low power consumption, automotive computer systems were developed and tested directly on the devices themselves, often in a manual way. This resulted in a large consumption of resources and a waste of time. Today, the goal is to have cloud infrastructures that ensure a more agile development and testing process due to the presence of general-purpose systems in the vehicle.

Consequently, the use of SDV aims to completely separate software and hardware, allowing the production of high-level software on entirely generalized hardware systems. This results in significant savings in terms of time and money for hardware production, along with providing an advantage in terms of security due to the simplification of software.

Another very important aspect of SDV, which will be analyzed in the following chapters, is that since a SDV is by definition characterized by the ability to dynamically and flexibly update software, this solution offers significant security advantages in several aspects:

1. **Human Safety Critical Security:** From the moment that a vehicle can be classified as safety critical (as it is reported in the standard International Organization for Standardization (ISO) 26262-1:2018 of the ISO society where is said that "safety is one of the key issues in the development of road vehicles" [16]), the elimination of software vulnerabilities related to the vehicle's systems is crucial for the overall safety of the vehicle itself.

2. **Intrinsic Software Security:** This approach allows for the prevention and resolution of vulnerabilities unknown at the time of software design, contributing to ensuring a high standard of security. For example, as demonstrated by National Institute of Standards and Technology (NIST) in the research on the Analysis Of The Impact Of Software Complexity [17], the increase in software complexity in different cases results in less analyzable programs. In some instances, the same vulnerability analysis tool may detect vulnerabilities, while in others, analyzing the same code, it may not.

Effectively navigating the development of SDV technology necessitates a collaborative approach across diverse companies, particularly in the realms of hardware and cloud computing. For this reason, many software, hardware, and automotive companies are involved in the development of this innovation, which aims to become a standard in vehicle production for the entire automotive industry.

In order to carry out the research and analysis of the new technologies described above, as well as to get involved in the practical side of things, it was essential to find a company that had both the Information Technology (IT) skills needed to interface with cloud technologies and experience in the world of automotive manufacturing and software production. The partner company with which the practical design and implementation of the working explanatory example was carried out is introduced below.

1.2 Partner Company

Leveraging extensive experience in the cloud industry and fostering deep-rooted relationships within the automotive sector, *Storm Reply* stands out as the ideal choice to lead the project discussed in this thesis. A key player in the *Reply* group, *Storm Reply* specializes in designing and implementing innovative Cloud-based solutions and services [18].

With a broad client base spanning multiple sectors, particularly the automotive industry, the company's expertise played a pivotal role in fully understanding the project's context and internal dynamics. This extensive knowledge provided the cornerstone for the development of a tangible example of the infrastructure.

Figure 1.2. Logo of the partenr company of the project

Among the main customers in the automotive world of the consulting company, we can mention *Ferrari*, one of the most important companies in motor sports competitions and in the production of luxury cars, and *Stellantis*, one of the biggest giants in the automotive industry as well as in the global market. Although different, these two companies interact with *Storm Reply* to take advantage of its great knowledge in the cloud world and in the management of Amazon Web Services (AWS) services. Thanks to the connection with these important companies it was possible to receive essential information for the thesis work.

One great advantage of collaborating with this company is the wide availability of resources, both material and, above all, in terms of experience. As a large IT consultancy company, *Reply* is divided into many sub-business units, that makes possible the interaction with various realities, ranging from embedded to low-level development, network and security infrastructure management, and web services management. Furthermore, there is a research and development section called Area 42 where entities can interact and influence each other to create innovative projects.

A point of pride for *Storm Reply* is its recognition as an AWS Premier Consulting Partner since 2014, ranking among the top Amazon Partners globally. This distinctive characteristic underscores the decision to develop the infrastructure using AWS services.

According to the official AWS description page [19] the *AWS Cloud* spans 102 *Availability Zones* within 32 geographic *Regions* around the world and servs 245 countries and territories. With millions of active customers and tens of thousands of partners globally, AWS has the largest and most dynamic ecosystem. AWS is evaluated as a leader in the 2022 *Gartner Magic Quadrant for Cloud Infrastructure and Platform Services* (a series of market research reports published by IT consulting firm *Gartner* that rely on proprietary qualitative data analysis methods to demonstrate market trends, such as direction, maturity and participants), placed highest in *Ability* to *Execute* axis of measurement among the top 8 vendors named in the report.

Figure 1.3. The Gartner Magic Quadrant for Cloud Infrastructure and Platform Services [2]

The infrastructure exhibits several key attributes contributing to its robustness and efficiency:

- **Security:** The infrastructure undergoes 24/7 monitoring to ensure the confidentiality, integrity, and availability of data. All data flowing across the AWS global network is automatically encrypted at the physical layer before leaving secured facilities.

- **Availability:** To ensure high availability and isolate potential issues, applications can be partitioned across multiple Availability Zones (AZ)s within the same region, creating fully isolated infrastructure partitions.

- **Performance:** *AWS Regions* offer low latency, low packet loss, and high overall network quality. This is achieved through a fully redundant 100 GbE fiber network backbone, often providing terabits of capacity between *Regions*.

- **Scalability:** The *AWS Global Infrastructure* allows companies to take advantage of the virtually infinite scalability of the cloud. This enables customers to provision resources based on actual needs, with the ability to instantly scale up or down according to business requirements.

- **Flexibility:** The *AWS Global Infrastructure* provides flexibility in choosing where and how workloads are run, whether globally, with single-digit millisecond latencies, or on-premises.

- **Global Footprint:** AWS boasts the largest global infrastructure footprint, continually expanding at a significant rate.

Thanks to its expertise and qualifications, *Storm Reply* is able to provide the above features to its customers, offering a comprehensive consultancy service for managing cloud infrastructures.

1.3 Thesis Objective

In the automotive context, the use of SDV plays a crucial role in terms of cost, innovation and safety. The objectives of the thesis are intertwined with the opportunities offered by SDV technology, for instance addressing the primary challenge of overcoming the current difficulties associated with the presence of different specialized hardware platforms on the same vehicle, to make the vehicle a more efficient and safer device based on software as a fundamental element.

One of the main objectives of this thesis is to propose the opportunity offered by SDV solution capable of eliminating various phases of the software production pipeline. This would result in significant time and cost savings, enabling the investment of these resources in other areas.

From a practical standpoint, the project's goal is to provide, through the use of AWS services, a cloud infrastructure capable of managing the SDV both in terms of software production and data analysis.

The work begins with an overview of the state of the art of software development in the automotive world, comparing the goals of the future with the techniques used in the past. For this purpose, the weaknesses of the sector are explained in order to highlight the advantages of SDV technology. Next, some definitions of the technologies that can bring the development of a paradigm shift in SDV benefits are provided. Finally, an example of an initiative proposed as a first attempt to standardize the SDV concept is presented, that is the Scalable Open Architecture for Embedded Edge (SOAFEE) project.

The work continues with an introduction to cloud computing, which is the programming approach that accompanied the entire project from beginning to end. The characteristics of this technique are analyzed, especially the advantages it can bring, and the example of how AWS manages to best enhance the potential of cloud computing is shown. Special attention is given to the security aspect, as it is a fundamental objective of the thesis topic, but also central element of the idea behind the cloud development of the AWS company and the project partner company.

Moving towards the description of the implementation of the practical project, it is possible to arrive at the exploration of the AWS services used. In order to make the realization of the project possible, as described several times throughout the thesis, it is necessary to rely on this type of service. With the aim of introducing the characteristics of the services used, an extensive descriptive list is provided.

The last part of the thesis deals with the actual implementation of the project, with the purpose of providing a concrete and working example of what has been

described in the previous chapters. The implementation begins with the creation from scratch of a device capable of simulating telematic data, develops in the construction of a cloud infrastructure with the services mentioned above, and ends with the exploration of the tool used for data analysis.

Finally, to conclude the research, the results of the final presentation of the operation of the whole system on a real device are shown. In addition, a final evaluation of the whole project is made and the future possibilities opened by the work are shown.

Chapter 2

State-of-the-Art Analysis

The following chapter constitutes an in-depth exploration of current technologies and methodologies within the automotive industry, with a specific focus on the complexity of vehicular software development. Firstly, the current automotive landscape will be examined, providing a detailed insight into challenges associated with software development in vehicles.

Subsequently, through meticulous analysis of scientific publications, technical reports, and practical implementations, the chapter delves into the radical transformation of the automotive sector facilitated by the concept of SDV. This technology, crucial for technological progress and vehicular safety, will be explored from various perspectives. Particularly, the synergy between cloud computing, software, and hardware will be investigated, highlighting solutions proposed by major industry players and analyzing their applications, benefits, and limitations.

The objective is to offer a comprehensive overview of current dynamics, emphasizing the pivotal role of SDV in the evolution of the automotive industry.

2.1 Context

In the past, the automotive industry advanced primarily through the development of technologies in mechanical engineering, focusing on perfecting combustion engines. Nowadays, the paradigm has radically changed due to multiple factors, including electrification, automation, shared mobility, and connected mobility.

Software technology development in the automotive field can be metaphorically compared to what has happened in smartphone development, as highlighted in the manifesto document regarding *Bosch*'s SDV [20].

The ultimate goal is to achieve simple and user-friendly devices that fully meet the user's needs. Currently, many customers express dissatisfaction because their cars do not offer the same functionality and ease of use common in smartphones. Many ask the question about how is possible that their $50,000 car can't perform the same tasks as their $300 smartphone.

A key difference between the automotive and smartphone industries is the level of complexity, which brings with it a number of issues.

2.1.1 difficulties

It is possible to analyse in depth the problems of the current automotive software that is being developed via four main difficulties:

- **Specialized Hardware:** Today's vehicles are still complex systems of systems. Each subsystem in a car, from brakes to transmission, is a complex entity, supplied by a different manufacturer and integrated with a unique software architecture. The level of complexity and the need for seamless interoperability between systems far exceeds that of today's smartphones.
- **Time:** The software production pipeline involves many development and testing steps with a not inconsiderable amount of time spent on each one. This is greatly increased by the presence of different components, so development time must be considered for each different unit of the system.
- **Cost:** The complexity of the software systems in vehicles entails very high costs, aggravated by the fact that the test phase is often carried out directly on the boards (for hardware requirements), which means a much longer production process, especially in the event of errors.

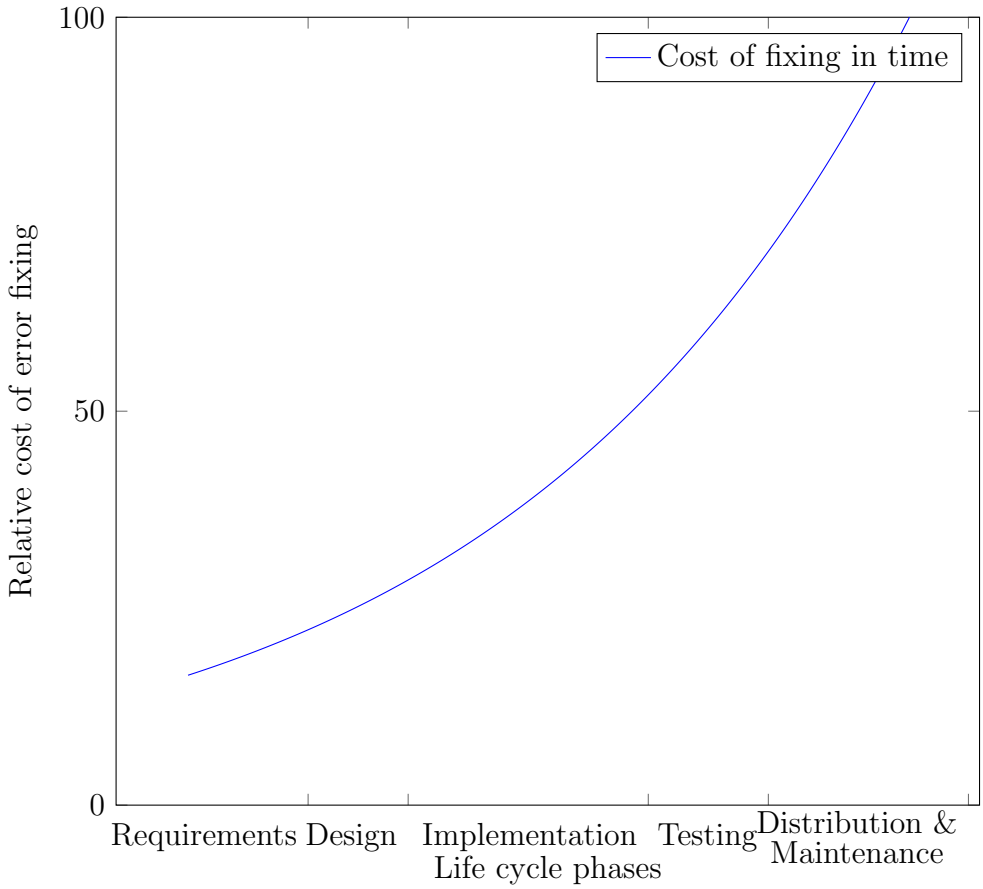

Table 2.1. Cost of fixing errors increases in later phases of the life cycle [12]

- **Human Safety Security:** Automotive embedded software must meet stringent reliability and security requirements, while delivering performance and a reasonable memory footprint. To develop automotive embedded software, you need the right tools that meet safety and security standards to evaluate, prototype and test your software.

At this point, Several valuable lessons can be learned from studying the barriers that apply to the vehicle life cycle. Historically, the vehicle lifecycle has been characterised by the simultaneous production and deployment of tightly integrated hardware and software. Once the vehicle was in the hands of the consumer, its characteristics remained largely unchanged until the end of its life. However, the SDV paradigm introduces the possibility of decoupling hardware and software release dates, a prerequisite for adopting a digital-first approach. This approach brings the design and virtual validation of the digital vehicle experience to the forefront of the lifecycle. It also requires the application of the digital-first concept, which means that new ideas for the vehicle experience are first explored in virtual environments to ensure early user feedback, long before any custom hardware needs to be developed or a physical test vehicle is available. Digital first is the application of design thinking and lean startup principles, originally rooted in internet culture, to the tangible realm of automotive development.

2.2 Introduction to Software Defined Vehicle

The SDV represents the new frontier of automotive manufacturing and is poised to completely change the paradigm of automotive production.

Let's try to imagine to bringing a feature update to one of today's vehicles. It will most likely take anywhere from one to seven years from the idea to when that feature is actually perceptible in the production vehicle; this takes so long because the vehicles produced up to this point have not been designed with frequent updates in mind [21]. Traditionally focused on physical functionality, the automotive industry has evolved from early electronic features such as airbags, vehicle stabilisation and braking systems to modern driver assistance and even automated driving. The current shift towards a digital experience is possible thanks to vehicle design that includes software integration as a fundamental part. Software should no longer be seen as an accessory to the vehicle, but as an integral part of the vehicle itself.

The simultaneous efforts of major automotive companies such as *Bosch*, *Renault* and *Stellantis*, in collaboration with leading computer developers such as *Arm*, *BlackBerry* and *AWS*, have given rise to the SDV concept, which they define as "any vehicle that manages its own operations, adds functionality and enables new features primarily or entirely through software" [15].

The SDV solution is nowadays being considered by several companies as the manifesto of a new era of vehicle development. An example is given by the *Renault Group*, which in an overview of its products describes: "Today, it is already possible to make remote updates of some vehicles via the Firmware-Over-The-Air (FOTA) system. This keeps the vehicle safe by making it easier and faster to improve the

on-board system and apply patches. Tomorrow, the SDV's flexible and scalable architecture will enable the faster development and integration of new features throughout the vehicle lifecycle, directly into the cloud, that is, in secure online servers accessible from anywhere and anytime" [22].

It is evident that SDVs represent the future of the automotive industry, promising an enriched and sustainable user experience as vehicle technologies evolve. This section further clarifies the current state of the industry, highlighting the key enablers that are allowing the development of the SDV paradigm and the benefits of this innovation.

2.2.1 Enablers

There are mainly four fundamental technologies that contribute to the realisation of the SDV: standardized hardware, cloud, Over-The-Air (OTA) updates via OTA servers, and Message Queuing Telemetry Transport (MQTT) communication. All these enablers are developed by leading companies in the computing industry. In this section, each technology will be analysed with reference to concrete examples from the current market.

Standardized hardware

One of the most important aspects of SDV is the separation of software from hardware. To achieve this, it is essential to move away from the approach of using dedicated hardware for each vehicle component system, and instead favour an approach based on general purpose processors that are as centralised as possible. This transition not only promotes ease of software development and scalability, but also offers the opportunity to create parity between the virtual development and test environment and the real execution environment.

Several players in the semiconductor industry have stepped up to the challenge of realising this vision, including *Arm*. Through the development of energy-efficient processors, *Arm* is present in every part of the vehicle, from high-performance systems in Advanced Driver Assistance Systems (ADAS), Automated Driving (AD), In-Vehicle Infotainment (IVI) and digital cockpits, to gateway, body and microcontroller endpoints [23]. The aim is to create Arm-based MicroController Unit (MCU)s that enable implementation of a common architecture, scalability between applications to meet processing requirements, software reuse and reduced development costs.

Another major player is *Qualcomm*, which is being adopted by the *Renault Group* through its *Snapdragon Digital Chassis* vehicle architecture, a set of cloud-connected platforms for telematics and connectivity, digital cockpits, assistance and driver autonomy.

Cloud

Using a cloud platform that offers scalable and secure solutions for real-time application updates, increased connectivity and efficient data management is essential for SDV.

Well-known companies such as *AWS* and *Google Cloud* are already present in the automotive industry as partners of partner of many automotive companies. The AWS services and technologies will be in depth described in the futher chapters.

Over-The-Air updates

An OTA update is the remote and wireless transfer of applications, services, firmware and configurations from a server to a target device. This process takes place over an available network, preferably the Internet. The main purposes of OTA are to remotely update software or firmware, provide power-safe procedures to ensure that the device will boot even if power is lost during the update process, maintain a robust implementation, ensure data protection and reduce overall maintenance costs [24].

In the context of the thesis, it is crucial to acknowledge that the implementation of OTA updates may increase the vulnerability of automotive systems to hacking and other cyber attacks. These vulnerabilities could potentially be exploited by hackers to gain unauthorised access to private information, take remote control of the vehicle or even cause it to malfunction. Another significant issue is the leakage of information about updates and their sources. This can enable malicious actors to introduce viruses and malware, further exacerbating the security risks associated with OTA updates [25].

To perform an OTA update, both a client on the vehicle, responsible for waiting and checking for incoming updates, and a server, facilitating the availability of the update broadcast to all connected devices, are essential. In this context, Autosar can be considered, as it represents a standard and open source architecture for intelligent mobility [26], which includes a dedicated platform for client and server management of OTA updates. Another notable example is Hawkbit, which serves as a backend framework for deploying software updates to edge devices and is being developed by the Eclipse Foundation; this tool will be discussed in more detail in later chapters as it will be used to create a proof of concept. The final tool of note is AWS Greengrass, an edge agent manager for managing software updates in edge IoT devices, provided by AWS; this tool will also be discussed in later chapters as an alternative solution to the client manager.

MQTT communication

The MQTT is a standardized protocol, specified by *ISO/IEC 20922:2016* and developed by the *Oaesis* organization. It enables the exchange of *Application Messages (AM)* over a network connection, providing an ordered, lossless stream of bytes from the *Client* to *Server* and *Server* to *Client* without the need to support of a specific transport protocol.

In an MQTT transport, an AM carries payload data, a Quality of Service (QoS), a collection of *Properties*, and a *Topic Name (TN)*. *Clients*, which can be programs or devices, perform various actions such as opening and closing network connections, publishing *AM*, subscribing to requested *AM*, and managing subscriptions [27].

On the *Server* side, it acts as an intermediary between publishing and subscribing *Clients*. The *Server* accepts network connections, processes *Subscribe* and *Unsubscribe* requests, and forwards *AM* matching *Client Subscriptions*. The *Server*, also known as the *Broker*, essentially coordinates messages among various *Clients*. Its responsibilities extend to authorizing and authenticating *MQTT Clients*, transmitting messages to other systems for further analysis, and managing tasks such as handling missed messages and *Client* sessions [28].

Sessions, representing stateful interactions between Clients and Servers, can last for the duration of a Network Connection or span multiple consecutive connections.

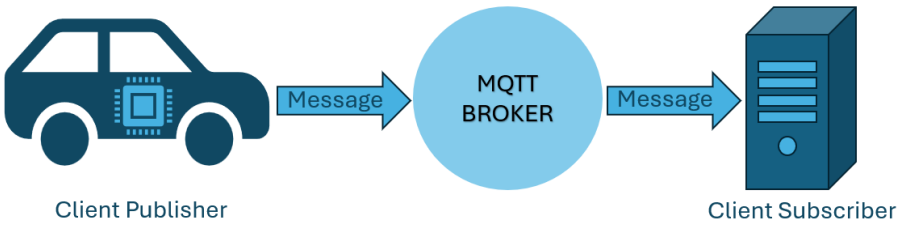

Figure 2.1. A simple representation of communication using the *MQTT* protocol

The MQTT protocol can be used in SDV, both for sending data produced by the vehicle to the cloud servers and for sending updates from the servers to the vehicle. This is because the MQTT protocol allows asynchronous and misaligned communication even in the presence of poor connectivity, a situation that cannot be underestimated in the automotive field.

The collaborative efforts of this technologies contribute to advancement of SDV for makeing vehicles not only defined by their physical attributes but also as dynamic entities that can be continuously updated through software.

2.2.2 Benefits

The SDV, as introduced in the previous chapters, brings several benefits to both automotive companies and the end-user experience. These innovations are made possible by the fact that the vehicle becomes a device that can be constantly monitored and updated in real time via the cloud throughout its entire lifecycle. Let us now look at the key benefits.

From the point of view of this project, the main innovation brought by this technology is the security of the device software. Since, as mentioned above [16],

vehicles are considered as safety elements critical to human life, the safety benefits can be analysed from two perspectives:

- **Human Safety Critical Security:** The ability of SDV to receive real-time data from the vehicle allows in-depth monitoring of all its components. Taking the influence of tyres as an example, it has been found that most road accidents are caused by tyre wear and lack of regular maintenance. It is therefore necessary to assess the health of tyres through continuous monitoring of physical parameters such as tyre thickness, temperature and pressure, as well as regular maintenance. This helps to eliminate or minimise the possibility of tyre bursts and subsequent accidents. It also improves the safety of people and vehicles [29]. These factors can be monitored either manually or automatically: manual predictive maintenance requires human intervention and can lead to some errors; automatic predictive maintenance using artificial intelligence can be more efficient [30]. *Renault* defines this work as "predictive maintenance" [22], stressing the importance of collecting and analysing data in a centralised system to anticipate and prevent potential failures, ensure the safety of people, reduce maintenance costs and improve the performance of the vehicle.

- **Intrinsic Software Security:** In the presence of bugs and vulnerabilities in the vehicle's software, SDV makes it possible to intervene promptly to resolve each problem and reduce the window of exposure.

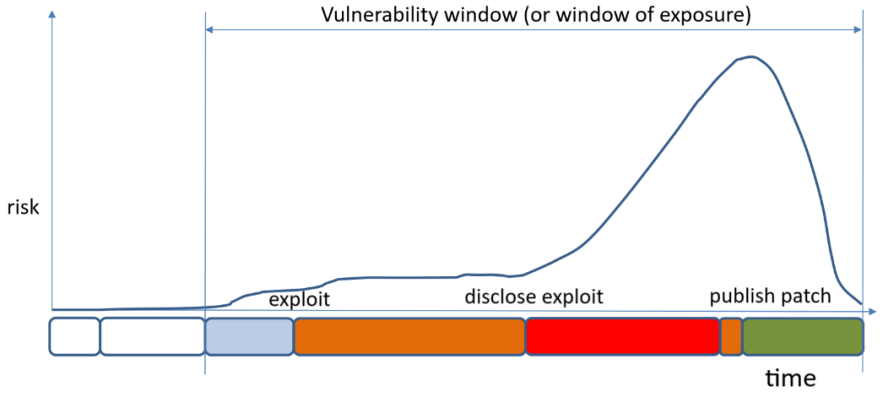

Table 2.2. Risks and time relationship in the various phases of a vulnerability lifecycle

A crucial aspect of vehicle software security is the robustness of the algorithms, especially in the context of autonomous driving. In this context, a predictive algorithm responsible for vehicle safety decisions can be continuously improved and optimised. The SDV also introduces the concept of the *Digital Twin*, that is a platform that virtually replicates the functionality and behaviour of the vehicle in a computing environment. Thanks to this technology, predictive algorithms used in autonomous driving can be effectively tested on the cloud platform and, when ready, integrated directly into the vehicle.

From a user experience point of view, two other significant benefits can be identified: an increase in the value of the vehicle, which can be continuously upgraded over time, and the ability to enable additional vehicle functions via software. For example, the user can decide to activate a feature for a certain period of time and then deactivate it (paying only for the time it is used), or activate a new feature that was not available at the time of purchase. In essence, the vehicle becomes a dynamic platform that is constantly evolving and fully customisable through the software.

For automotive companies, the benefits mentioned so far can bring direct advantages to the entire industry. In support of this, *Stellantis* reports that: "the team in Poland will contribute to the global software creation network that is key to Stellantis' work in creating SDV that offer customer-focused features throughout the vehicle's life span, including updates and features that will be added years after the vehicle is manufactured. Creating an infrastructure inside our vehicles that easily and seamlessly adapts to meet driver expectations is a key element of Stellantis' global drive to deliver cutting edge mobility. Stellantis' software-driven strategy deploys next-generation tech platforms, building on existing connected vehicle capabilities to transform how customers interact with their vehicles and to generate €20 billion in incremental annual revenues by 2030" [31].

In addition, the SDV paradigm brings an advantage from a software production pipeline perspective. In today's software production scenario, there can be two development mechanisms:

- A more traditional mode in which software is created directly on the system, hence on the processor itself. This is undoubtedly the most inconvenient solution, as it would require unnecessary overuse of processors, wasting resources, money and time.
- Alternatively, developers rely on cumbersome operating system emulation tools on the host machine and the cross-compilation process, which uses a dedicated compiler to produce executable code for the target system. Once the code is on the development system, a final integration and validation test can be performed, but scalability is limited to the number of physical hardware platforms.

Typical workflows for the development, integration and validation of embedded systems are as follows:

By using the SDV, such as operating systems that rely on general porpouse architectures to provide parity between cloud and edge systems, it is possible to reduce the embedded developer's workflow to remove many of the steps that are now no longer required, as shown in the diagram below 2.3. More specifically, the development and integration workflow eliminates the build and test phases. Instead, a validation function is added to verify the product in a cloud environment, where the digital twin concept can be used directly. All tests can be conducted in a virtual cloud environment where a digital copy of the actual vehicle is available for distributing the product software or related updates. This reduces software production times, costs, and waste of physical resources.

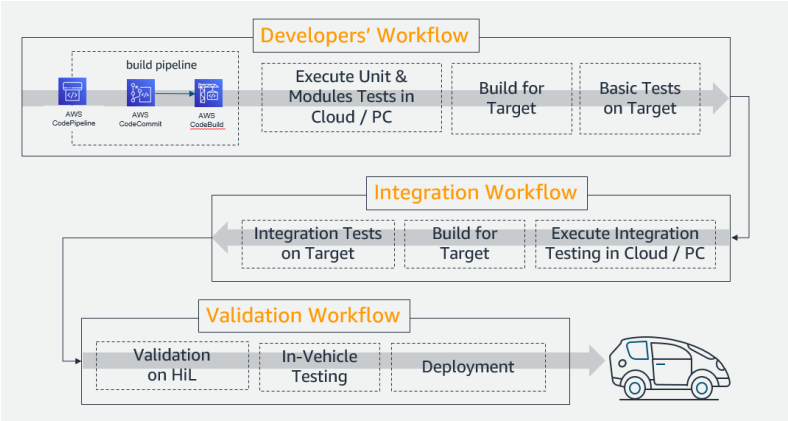

Figure 2.2. today development, integration, and validation workflows for embedded systems [3].

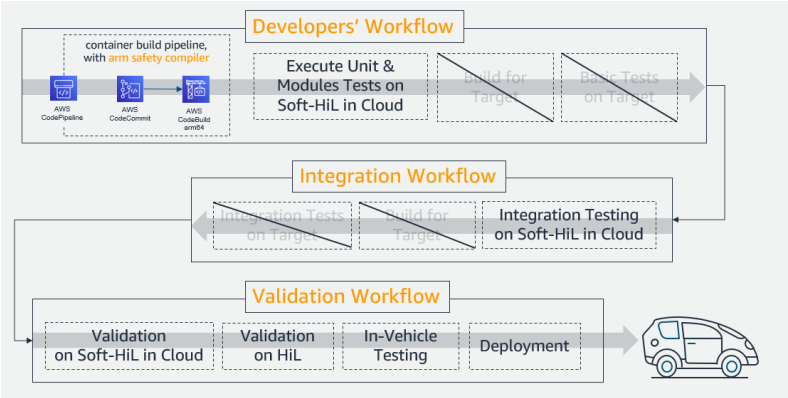

Figure 2.3. future development, integration, and validation workflows for embedded systems [3].

2.2.3 initiatives: SOAFEE

In 2021 *Arm*, *AWS*, and other founding members announced the *SOAFEE Special Interest Group*, which brings together automakers, semiconductor, and cloud technology leaders to define a new open-standards based architecture to implement the lowest levels of a software-defined vehicle stack [3].

SOAFEE is created to achive SDV objective, and for doing that four-pillar principle are used [32]:

1. **Standards:** standardization ensures interoperability and compatibility among various software components, fostering a cohesive ecosystem for SDVs.

2. **New software architecture and methodologies:** this involves transitioning from traditional monolithic architectures to more modular and scalable designs; the incorporation of agile development practices and *DevOps* methodologies ensures efficient and continuous software evolution.

3. **Industry collaboration:** Fostering partnerships, knowledge sharing and

collaboration among key stakeholders, including automakers, technology companies and regulators, is essential.

4. **Vehicle simulation:** simulated environments allow in-depth testing and refinement of software functionality to ensure optimal performance and security under a variety of conditions.

SOAFEE aims to adopt and enhance current standards used in today's cloud-native world to help manage the software and hardware complexity of the automotive SDV architecture.

The core principles of safety, security, and real time are inherent in each pillar. It is fully expected that the SOAFEE architecture will support use-cases that execute safety-critical services alongside non-safety-critical ones. It is fully expected that the SOAFEE architecture will support use cases that execute safety-critical services alongside non-safety-critical services. As it is not reasonable to develop the whole platform according to one safety standard, the strategy is to develop only safety-critical elements according to *ISO 26262* and to isolate them from the non-safety-critical elements in order to ensure spatial, temporal and communication isolation. All implementations pass security checks and follow a set of best practices [33].

The SOAFEE paradigm is based on a very sophisticated architecture becouse it should work in the same way in the vehicle and in the cloud and follow cloud native technologies while considering the automotive specific needs for safety and limited resource footprints [4].

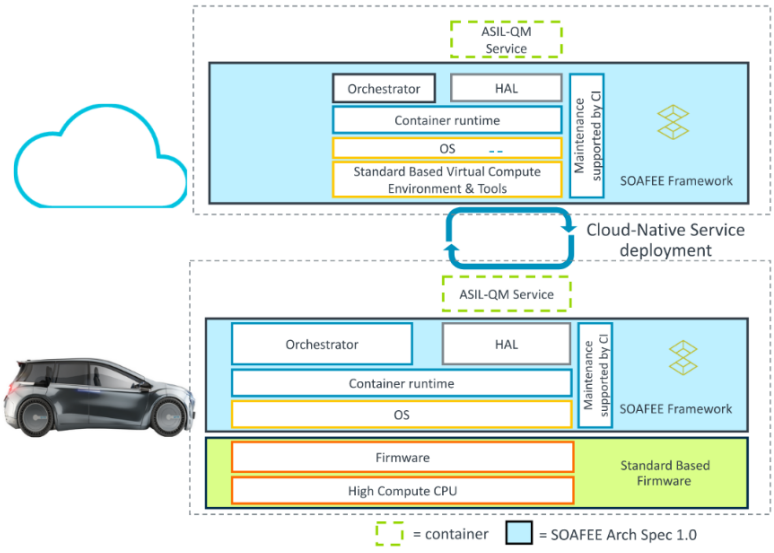

Figure 2.4. *SOAFEE* Architecture v1.0 [4]

Chapter 3

Cloud Computing and Amazon Web Services

As the analysis in previous chapters has shown, the SDV is a pivotal advancement in the evolution of the entire automotive industry toward a safer, more efficient, and more sustainable future. Cloud computing is a crucial resource for SDV development due to its facilitation of development through its features and benefits. In the following section, cloud computing technologies will be analyzed in detail, focusing on one of the most important providers, AWS.

3.1 Cloud Computing

The NIST provides the most comprehensive definition of cloud computing such as: "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models" [34]. This allows for a thorough analysis of the features of cloud computing in relation to AWS services, starting with the five essential characteristics.

- **On-demand self-service:** Consumers can access and allocate computing resources autonomously, such as server time and network storage, without direct involvement with service providers. AWS offers a vast cloud infrastructure with over 200 fully-featured services that consumers can easily access and use from their AWS account.
- **Broad network access:** Resources can be accessed over the network through standard mechanisms, making them usable across various client platforms. In AWS services, this is translated as an on-demand delivery of IT resources over the *Internet* with "pay-as-you-go" pricing.
- **Resource pooling:** Providers pool computing resources in a multi-tenant model, dynamically assigning them based on consumer demand. As said before AWS services are allocable e pagabili in base alle necessità del momento.

The customer has limited control over the exact resource location but can specify a higher-level abstraction as country, state, or datacenter. In AWS, clients can select the geographic location of their services through regions. textitAWS Regions provide access to AWS services that are physically located in a specific geographic area. AWS provides the option to view the availability of a particular service in a specific region, in addition to selecting different regions [35]. Resources include storage, processing, memory, and network bandwidth. It also provides services for the IoT, machine learning, data lakes, and analytics.

- **Rapid elasticity:** Resources can be easily adjusted to match fluctuations in demand, either automatically or manually. AWS provides various automated resource allocation systems, including the AWS Cloud Development Kit (CDK) framework, which will be discussed later. The available capabilities are perceived as virtually limitless, and consumers can acquire them in any quantity at any time, always with a "pay-per-use" system.

- **Measured service:** Cloud systems efficiently manage resources through automated control and optimization, utilizing metering capabilities tailored to specific services such as storage, processing, bandwidth, and user accounts. For instance, AWS has infrastructure worldwide, allowing for easy deployment of applications in multiple physical locations. The proximity to end-users reduces latency and enhances their experience. This feature allows for clear and objective monitoring, control, and reporting of resource usage by both providers and consumers.

The three primary types of cloud computing are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These options provide different levels of control, flexibility, and management, allowing users to configure services to meet their specific requirements.

- **IaaS:** Consumers are able to utilize and deploy fundamental computing resources, including processing, storage, and networks. However, they only have control over operating systems, storage, and applications, as the cloud infrastructure is managed by the provider. Consumer control over some networking components is limited. IaaS provides a high level of flexibility and management control over IT resources. It is similar in practice to existing IT resources that many IT departments and developers are already familiar with.

- **PaaS:** Consumers can deploy their applications on the cloud infrastructure using the programming languages, libraries, services, and tools supported by the provider. The provider manages the underlying cloud infrastructure, including network, servers, operating systems, and storage, while consumers maintain control over their applications and configuration settings. This approach improves efficiency by eliminating the need to manage resource procurement, capacity allocation, software maintenance, patching, or any other tasks involved in running your application.

- **SaaS:** Consumers can use the provider's applications on the cloud infrastructure, which are accessible from different client devices through interfaces such as web browsers or programs. However, consumers do not have control over the underlying cloud infrastructure, including the network, servers, operating systems, and storage, except for limited user-specific application configuration settings. With a SaaS offering, users do not need to worry about maintaining the service or managing the underlying infrastructure. The focus should be on how to use the software effectively.

The analysis thus far has focused on cloud computing, specifically the essential characteristics that a cloud service must possess to be considered a true cloud service, as well as the service models that can be offered. Now, let's analyze in more detail the part related to cloud computing service deployment models and explore which models are most suitable for which workloads using AWS [36]. Note that in this case, there are slight differences between the NIST and AWS definitions of the various deployment modes.

- **public cloud:** According to NIST, a public cloud is defined as cloud infrastructure that is publicly accessible and owned, managed, and operated by businesses, academic institutions, government entities, or a combination thereof. In contrast, AWS defines a public cloud as infrastructure and services that are accessible over the public internet and hosted in a specific *AWS Region*.
- **private cloud:** Both NIST and AWS define private cloud as a cloud infrastructure exclusively provisioned for a single organization, which may own, manage, and operate it independently or in collaboration with a third party. However, there is a difference in the location of the infrastructure. According to NIST, the infrastructure can be located on or off premises, while in AWS documentation, the infrastructure is provisioned on premises using a virtualization layer.
- **hybrid cloud:** The hybrid cloud is a combination of two or more separate cloud infrastructures, private or public, connected by technology to facilitate data and application portability. It allows organizations to leverage the cloud for its efficiency and cost savings while also maintaining on-site security, privacy, and control.

Exploring the many benefits of cloud computing, the focus now shifts to a comprehensive analysis of the main value patterns of cloud computing, with some charts and graphs to help clarify the outlook.

Cloud computing reduces costs by aggregating resources needed by different companies in a transparent way to consumers. This means that a company will no longer need to spend excessive amounts of money on building its IT system to meet overestimated demands or satisfy underestimations of necessary resources as it is shown in 3.1. In addition to shortening the time to market and increasing earnings, cloud computing allows for access to resources anytime and anywhere, optimizing resource management with lower latency and a better experience as it is shown in 3.2.

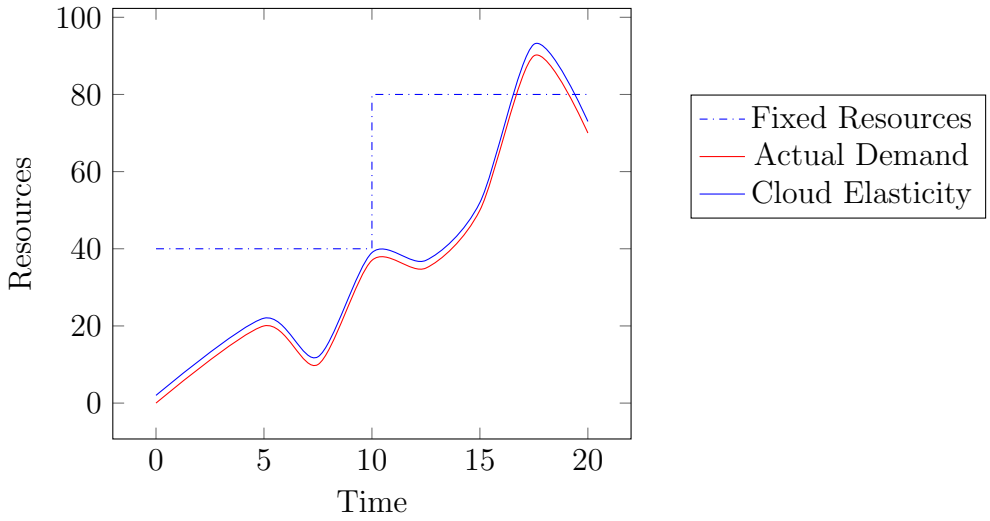

Table 3.1. Operating expenditure value model [13]

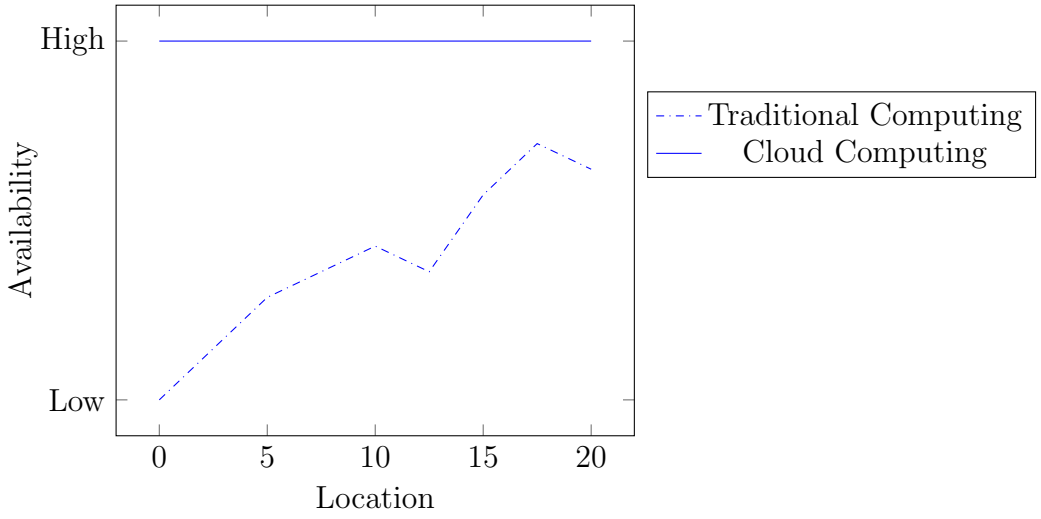

Table 3.2. Location flexibility value model [13]

In conclusion, cloud computing, exemplified by platforms like AWS, enables organizations to access IT resources on-demand via the Internet. This is facilitated by pay-as-you-go pricing models, which liberate organizations from the burdens of procuring, owning, and maintaining physical infrastructure. Cloud computing has a wide range of applications across industries, including the automotive sector. One of the main benefits of cloud computing is its dynamic scalability, which improves operational efficiency and reduces costs by utilizing resources more cost-effectively. This is due to the economies of scale inherent in cloud services, resulting in significantly lower variable expenses compared to self-managed infrastructure [37]. The characteristics of AWS are analyzed in depth below.

3.2 Amazon Web Services

AWS is a widely adopted cloud solution with over 200 fully featured services available globally across multiple data centers. It is used by millions of customers, from emerging startups to industry giants and government agencies, as the cloud platform of choice to reduce costs, increase agility, and accelerate innovation [38].

AWS stands out by providing a broad set of services, including infrastructure technologies as well as cutting-edge capabilities such as machine learning, artificial intelligence, data lakes, analytics, and the IoT. This extensive service portfolio facilitates the fast, easy and cost-effective migration of existing applications to the cloud and the creation of diverse digital solutions. AWS provides purpose-built databases for various application types, allowing users to choose the most suitable tool for optimal cost and performance. The depth of AWS services is unmatched, providing customers with a comprehensive toolkit for diverse computing needs.

Beyond its vast offerings, AWS has a large and dynamic global community with millions of active customers and tens of thousands of partners. This inclusive ecosystem spans industries and business sizes, with startups, enterprises, and public sector entities leveraging AWS for a myriad of use cases. The AWS Partner Network (APN) solidifies this network with thousands of system integrators and independent software vendors who adapt their technology to work on AWS.

AWS demonstrates its commitment to innovation through continuous technological advancements. In 2014, AWS launched *AWS Lambda*, which pioneered serverless computing. This allows developers to run their code without the need to provision or manage servers. Another example is *Amazon SageMaker*, a fully managed machine learning service that empowers developers to use machine learning without any previous experience.

Rooted in more than 17 years of operational experience, AWS offers unmatched reliability, security, and performance [39]. Since its establishment in 2006, AWS has become a globally trusted platform, revolutionizing IT infrastructure services by providing a highly reliable, scalable, and cost-effective cloud solution for businesses worldwide in the form of web services with "pay-as-you-go" pricing [40]. One of the main advantages of cloud computing is the ability to replace a company's initial capital expenditures required for infrastructure with low costs that vary as needed and can scale with the business. AWS places great emphasis on the security of its systems and services, which is a fundamental pillar of their platform. In this thesis, we will analyze this feature in more detail.

3.2.1 Security

AWS is known for its flexible and secure cloud computing environment, designed to meet the strict security requirements of military, global banks, and high-sensitivity organizations. The infrastructure includes over 300 security, compliance, and governance services, supporting 143 security standards and compliance certifications. This architecture ensures scalability, reliability, and rapid deployment of applications and data while adhering to the highest security standards. Strong security

at the core of an organization enables digital transformation and innovation. AWS utilizes redundant controls, continuous testing, and automation to maintain monitoring and protection forever non-stop continuously. Unlike customers' IT departments, which often operate on limited budgets, AWS prioritizes security as a core business aspect and allocates significant resources to safeguard the cloud and assist customers in ensuring robust cloud security [41].

AWS empowers customers to confidently advance their businesses by providing a secure and innovative cloud infrastructure, a comprehensive suite of security services, and strategic partnerships. The AWS cloud infrastructure, combined with a comprehensive suite of security services and strategic partnerships, provides a solid foundation for secure innovation. Security is integrated and automated at every level of the organization, ensuring a swift and secure development process while reducing human errors. AWS offers a wide range of security services and partner solutions to help organizations effectively navigate evolving threats and compliance challenges. These expert-built capabilities equip organizations with the tools they need to stay secure and compliant [42].

The AWS global infrastructure follows rigorous security best practices and compliance standards, ensuring that users have access to one of the most secure computing environments in the world. It is designed and managed in alignment with a range of IT security standards, providing assurance to customers, including those in the life sciences industry, that their web architectures are built on exceptionally secure computing infrastructure. The main security standards obtained from infrastructure will now be explored through AWS documentation [43].

- **System and Organization Controls (SOC) 1, 2, 3:** AWS SOC Reports are third-party examination reports that demonstrate AWS's alignment with key compliance controls and objectives. SOC 1 focuses on controls relevant to a financial audit, covering security organization, access, data handling, change management, and more. SOC 2 expands to *American Institute of Certified Public Accountants (AICPA) Trust Services Principles*, evaluating controls related to security, availability, processing integrity, confidentiality, and privacy. The SOC 3 report is a publicly available summary of SOC 2. It includes an external auditor's assessment, AWS management's assertion, and an overview of AWS Infrastructure and Services. The report provides transparency and demonstrates AWS's commitment to security, compliance, and protection of customer data [44].

- **Federal Risk and Authorization Management Program (FedRAMP):** FedRAMP is a United States government program that ensures a standardized approach to security assessment, authorization, and continuous monitoring for cloud products and services. It is aligned with NIST SP 800 series. The program mandates that cloud service providers undergo an independent security assessment by a Third-Party Assessment Organization (3PAO) to verify compliance with the Federal Information Security Management Act (FISMA) [45].

Figure 3.1. *AWS SOC* Logo

- **ISO 9001:** "ISO 9001 is a globally recognized standard for quality management. It helps organizations of all sizes and sectors to improve their performance, meet customer expectations and demonstrate their commitment to quality. Its requirements define how to establish, implement, maintain, and continually improve a Quality management System (QMS)" [46]. AWS ISO 9001:2015 certification directly supports customers developing, migrating, and operating their quality-controlled IT systems in the AWS cloud. They can use AWS compliance reports as evidence for their own ISO 9001:2015 programs and industry-specific quality programs [47].

- **ISO/International Electrotechnical Commission (IEC) 27001:** ISO/IEC 27001 is a global security standard that outlines requirements for the systematic management of corporate and customer information. AWS has achieved ISO 27001 certification, demonstrating a comprehensive approach to assessing, managing, and mitigating information security risks. The certification covers AWS infrastructure, data centers, and services, ensuring ongoing compliance with international security standards [48].

- **ISO/IEC 27017:** "ISO/IEC 27017:2015 gives guidelines for information security controls applicable to the provision and use of cloud services by providing: additional implementation guidance for relevant controls specified in ISO/IEC 27002; additional controls with implementation guidance that specifically relate to cloud services. This Recommendation — International Standard provides controls and implementation guidance for both cloud service providers and cloud service customers" [49]. This certification ensures the implementation of precise, cloud-specific controls and validates AWS commitment to robust security measures in cloud services [50].

- **ISO/IEC 27018:** ISO 27018 is a global code of practice for safeguarding personal data in the cloud. It builds upon ISO 27002 and offers guidance on implementing controls for Personally Identifiable Information (PII) in public clouds. AWS's ISO 27018 certification affirms its dedication to internationally recognized standards, emphasizing privacy and content protection through the use of this certification[51].

- **Health Information Trust Alliance Common Security Framework (HITRUST CSF):** The HITRUST CSF integrates global standards such as General Data Protection Regulation (GDPR), ISO, NIST, Payment Card Industry (PCI), and Health Insurance Portability and Accountability Act (HIPAA) to establish a comprehensive framework for security and privacy controls. Some AWS services have been assessed under the HITRUST CSF CSF Assurance Program by an approved *HITRUST CSF Assessor* and have been found to meet the *HITRUST CSF Certification Criteria*. Customers can inherit AWS certification for controls relevant to their cloud architectures established under the *HITRUST CSF Shared Responsibility Matrix (SRM)*. The certification is valid for two years, describes the AWS services that have been validated, and can be publicly accessed [52].

- **Security, Trust, Assurance, and Risk (STAR):** The Cloud Security Alliance (CSA) introduced the STAR to promote transparency in cloud provider security practices. STAR is a publicly accessible registry that documents the security controls of cloud computing offerings. AWS has joined the *CSA STAR Self-Assessment*, aligning with CSA best practices. The completed *CSA Consensus Assessments Initiative Questionnaire (CAIQ)* reports for AWS are publicly available [53].

Chapter 4

Cloud Infrastructure

As discussed in previous chapters, the development of SDV technology requires a cloud infrastructure to handle server-side operations. AWS is a leading player in the cloud world, and therefore an ideal alternative for the advancement of SDV, as well as an active partner in the implementation of technologies that contribute to the creation of a publicly available SDV for all. The following discussion introduces and analyzes, via AWS documentation the key tools for successful Proof of Concept (PoC) implementation which will be explored in more detail later.

4.1 AWS Used Services

Among the hundreds of services offered by AWS, here are the ones that are used to build a cloud infrastructure that is useful for work purposes. The following services have been sorted in order of importance. In particular, services dedicated to development support, services related to IoT devices, services related to execution, services related to data management, and services related to virtual machine instances are analyzed.

AWS Command Line Interface (AWS CLI)

The AWS CLI is an essential tool for developing with AWS services. It allows interaction with AWS services from the command line of a local Personal Computer (PC), enabling the creation of infrastructure and management of properties from the command line.

AWS Boto

Boto is an AWS Software Development Kit (SDK) made for *Python*. A SDK, more generally, is a set of creation tools specifically for developing and running software in a single platform. It includes resources such as documentation, examples, and APIs to facilitate faster application development. Boto basically works as an interface for applications that need to interact with and take advantage of the services provided by AWS. The AWS SDK for *JavaScript v3* is another example of an SDK for *JavaScript* that works basically in the same way.

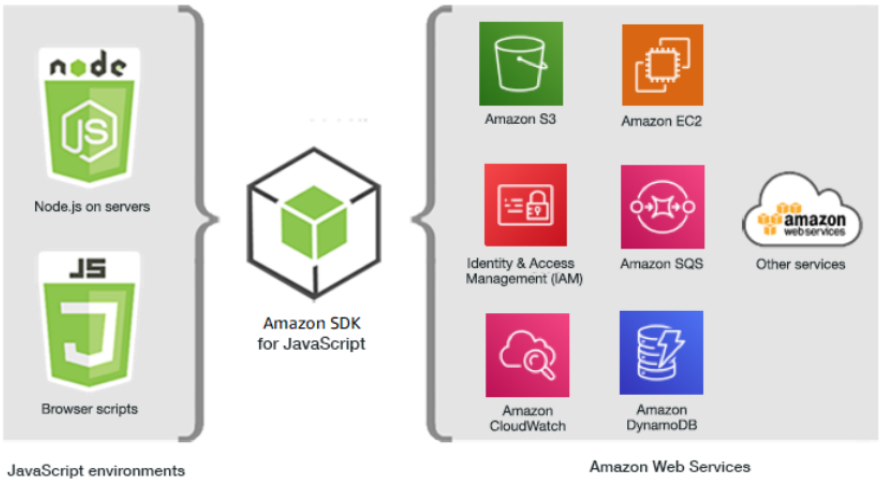

Figure 4.1. The high level rappresentation of the *AWS SDK* for *JavaScript v3* [5]

Cloud Development Kit (AWS CDK)

The AWS CDK "is an open-source software development framework for defining cloud infrastructure in code and provisioning it through *AWS CloudFormation*" [54]. It is compatible with both *JavaScript* and *Python* languages and it support the automatic creation of several services with just one execution command. This code-tool was used in the final phase of the PoC design to automate the creation of the stack comprising all the services used.

AWS IoT Core

AWS IoT Core provides the ability to connect IoT devices to AWS cloud services. *AWS IoT Core* enables the connection of IoT devices to AWS cloud services. It simplifies the integration of IoT devices with other AWS services. This is especially relevant in the automotive industry, where vehicle system Electronic Control Unit (ECU)s can be viewed as multiple IoT devices. Communication between the device and AWS services can occur in several modes, with the MQTT protocol being the most important for this project. The device can be connected by developing applications that utilize the SDK libraries. Once the data is transmitted, it can be utilized for various purposes such as testing, validation, and analysis. The AWS IoT services, including the *AWS IoT Core* service, allow for the creation of digital twins of physical IoT devices, known as *Thing*, and monitoring of traffic on selected MQTT channels. These elements will be explored in greater detail later in the PoC analysis were it is use as the connection point between the IoT device an the cloud infrastructure.

AWS IoT Greengrass

"*AWS IoT Greengrass* is an open source IoT edge runtime and cloud service that helps you build, deploy and manage IoT applications on devices" [55]. It is designed to work with intermittent connections and can manage fleets of devices in the field, locally or remotely, using MQTT or other protocols. Once installed, this service can be accessed through the command line. It

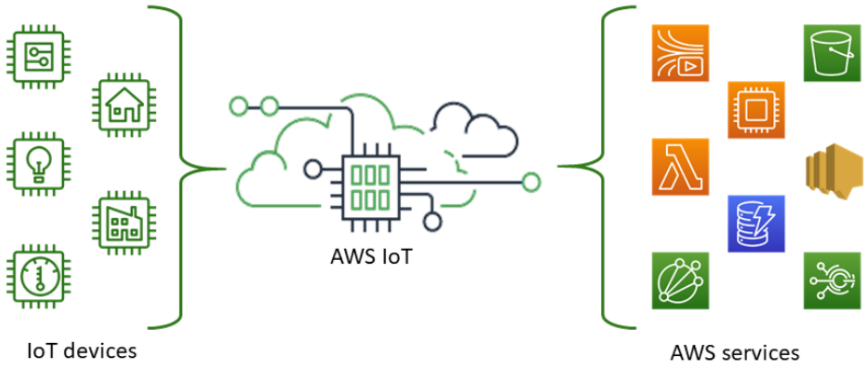

Figure 4.2. *AWS IoT Core* connection system between *IoT* device and *AWS* service [6]

was utilized in the early stages of project development as an agent to handle updates on the vehicle simulator side. However, this solution will be replaced by a custom solution as explained later.

AWS Identity and Access Management (AWS IAM)

"AWS IAM is a web service for securely controlling access to AWS services [...] such as access keys, and permissions that control which AWS resources users and applications can access" [56]. AWS IAM is a service that provides a powerful access management mechanism. However, for the purpose of this thesis, only the relevant functionality to the project will be analyzed, specifically AWS IAM's role management capabilities.An IAM role is an identity within AWS that can be assigned specific permissions via permission policies to determine what actions can and cannot be taken. Roles can be assumed by users, applications, or services that do not normally have access to the specific AWS resources. The AWS IAM service also provides another important concept, that of policy, which is an AWS object that, when attached to an identity (including roles) or a resource, enables the creation of permissions and access control to other resources. For example, as explained below, a policy can be attached to the cloud representation of an *AWS IoT Core* device to enable the connection of the physical dual IoT device or to grant Subscriber or *Publisher* permissions in a communication via MQTT protocol.

AWS Lambda

AWS Lambda is a computing service that provides the ability to run code without servers. It runs code on a high-performance computing infrastructure and handles administrative tasks related to computing resources autonomously, such as server and operating system management, capacity provisioning, automatic scaling, and logging. It is possible to run code for potentially any type of backend application or service [57]. Code can be written directly in *Lambda* console or imported from the local environment, and it supports several languages, including *Python* and *JavaScript*. The *Lambda* service can also manipulate data from other AWS services or manage tasks

with services outside AWS as will be analyzed below.

Amazon Cloudwatch

Amazon CloudWatch is a system to monitor the AWS resources and the applications running on the infrastructure in real time. With the use of *Amazon CloudWatch* it is possible to collect and track metrics from other AWS services such as *Lambda*, which are numeric variables that can be measured and analyzed for resources applications [58]. Practically this service represents the center for viewing and analyzing logs from the various AWS services in use.

AWS CodePipeline

AWS CodePipeline is a fully managed continuous delivery service that automates release pipelines for software updates. It enables fast and reliable updates to applications and infrastructure, facilitating the rapid release of new features, iterative development based on feedback, and bug detection through testing every code change. The software release process can be modeled and configured quickly via the *Stages* execution. A *Stage* is a logical unit that creates an isolated environment and allows for the execution of a limited number of concurrent software changes. Each stage contains actions that are executed on application artifacts, such as source code from *AWS CodeCommit*. For instance, as shown in the image 4.3, it is feasible to establish a software development pipeline that incorporates a CodeCommit repository as its source stage. This way, a CodeCommit-related event triggers the pipeline execution which then proceeds to the software build stage. An execution is defined as a series of modifications released from a pipeline. Each execution represents a set of modifications, such as a merged commit or a manual release of the last commit. Subsequently, the pipeline moves on to the test stage where the desired tests can be launched via *AWS CodeBuild*, and finally delivers the application for production.

Figure 4.3. An example of a *AWS CodePipeline* in which some stages are reported [7]

AWS CodeBuild

AWS CodeBuild is a fully managed build service in the cloud that provides source code compilation, unit testing, and production of executable programs ready for distribution [59]. *AWS CodeBuild* provides "out-of-the-box" configuration of compilation environments for popular programming languages,

such as *Python*. It is also possible to create build platforms for programming languages for which there is no preconfiguration, but in this case it is necessary to leverage multiple AWS services. It is also possible to use CodeBuild to run tests on application code using for example the *Pytest* tool that allows you to test *Python* code.

AWS CodeCommit

"*AWS CodeCommit* is a version control service that enables you to privately store and manage *Git* repositories in the AWS cloud" [60]. This service becomes particularly interesting in the context of multiple services working together, including *Lambda*, *CodePipeline*, and *CodeBuild*, because it allows the repository's *Git* and all its associated events (such as commit and push) to be used to trigger events that can automate various operations, such as triggering a pipeline in CodePipeline. As a result, *CodePipelines* typically use a *CodeCommit* repository as their input source stage that contains the code necessary for the subsequent stages.

Amazon Simple Storage Service (Amazon S3)

"Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance" [61]. The data saved in the storage is physically placed in multiple locations to ensure the durability of the data even if there is tampering with an item due to the presence of these copies; optionally, it can also be chosen to store the data in a single location to reduce the cost of the service. Amazon S3 can be used for data collection, aggregation, and analysis in many contexts and scenarios, but in the scope of this project, this service is used to store data that is transferred from one stage of the *CodePipeline* to another. *AWS CodePipeline* service automatically implements this method of output use. However, data stored in Amazon S3 from one stage to another can be manipulated through integration with other AWS services, such as *Lambda*. This service is not explicitly mentioned in the description of the project as it plays a secondary role in the implementation, but it is present, even if not always visible to the user.

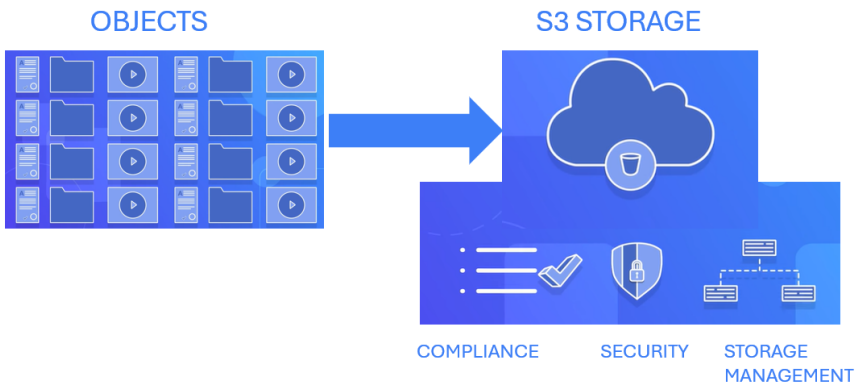

Figure 4.4. *Amazon S3* high level storing rappresentation

Amazon Kinesis Data Streams

Amazon Kinesis Data Streams is used to collect and process large streams of data records in real time, and eventually route them through other AWS services to various data collection and analysis applications, such as Amazon S3 as it is shown in the image 4.5. "The delay between the time a record is put into the stream and the time it can be retrieved (put-to-get delay) is typically less than 1 second. In other words, a *Kinesis Data Streams* application can start consuming the data from the stream almost immediately after the data is added" [62]. The *Kinesis Data Stream* service allows for the selection of specific data based on characteristics through an integrated query system. Additionally, this service can serve as input for *Lambda* functions or to populate databases.

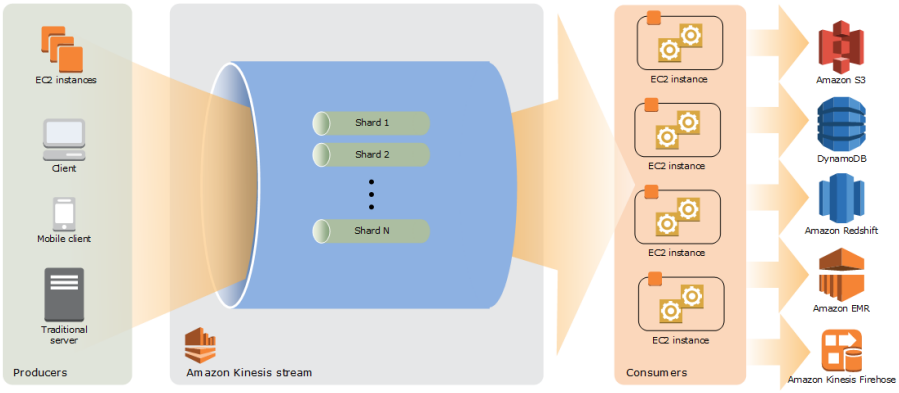

Figure 4.5. Illustration of the high-level architecture of *Kinesis Data Streams* with some examples of services that use the output of the stream. [8]

Amazon Timestream

Amazon Timestream is a time-series database that allows you to store and easily analyze large amounts of data stored at regular intervals, ensuring that the time-series data is always encrypted, whether at rest or in transit. This service simplifies the complex process of managing the lifecycle of data by providing storage tiering with an in-memory store for current data and a magnetic store for historical data using Amazon S3 space. The transition of data between these two storage types is enabled through the use of user-configurable policies. The data lifecycle management mechanism makes *Amazon Timestream* ideal for handling telemetry data from IoT devices, for example. The service also provides a built-in interface for accessing data through a query engine [63]. The *Amazion Timestrem* service also provides an interface for *Grafana* to view and analyze stored data, which will be explored later.

DinamoDB

Amazon DynamoDB is a full-featured NoSQL database service that provides high perfomances both speed and scalability. *Amazon DynamoDB* removes the administrative complexity of running and scaling your distributed database, so there's no need to manage provisioning, hardware setup and

configuration, replication, software patching, or cluster sizing. *Amazon DynamoDB* also provides encryption at rest, eliminating the operational costs associated with protecting sensitive data. *Amazon DynamoDB* provides the ability to change the allocation of resources needed to store data in real time to use only the resources required. Additionally, *Amazon DynamoDB* offers on-demand backup functionality for long-term retention and archival purposes, as well as "point-in-time" recovery to safeguard against accidental write or delete operations. This feature enables users to restore a table to any point within the last 35 days [64]. Note that this service was not utilized in the final version of the project, but was considered during development as an alternative for data storage and as a case study for understanding the data storage mechanisms used by AWS services.

AWS System Manager

AWS Systems Manager is a service that provides visibility and control of the infrastructure on AWS. It allows users to view operational data from multiple AWS services and manage the automation of operational tasks across different AWS resources [65]. The *AWS System Manager* service is particularly relevant to the project due to its application management capability, namely the *Parameter Store*. *Parameter Store* is used to securely store configuration data and secrets, such as passwords, connection strings, and Amazon Machine Images (AMI) identifiers. Values are stored hierarchically by assigning hierarchical names to stored values using the "/" character, while maintaining the uniqueness of the name. For example, names such as "Parameters/Parameter1", "Parameters/Parameter2" can be used. In addition, it is possible to choose whether to store the data as plain text or encrypted data. Stored data can be retrieved directly from other services, for example, by interacting with Lambdas and SDK code functions.

Amazon Elastic Container Registry (Amazon ECR)

"Amazon ECR is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM. This is so that specified users or instances can access [...] container repositories and images" [66]. Basically, as shown in the figure 4.6, once the software has been produced and packaged, for example through the use of the *CodeBuild* service, it can be uploaded to Amazon ECR. The Amazon ECR takes care of encrypting the image and controlling access to it, and then automatically manages the entire lifecycle of the image. Once the image is on Amazon ECR, it can be used either as an image for local download or through other AWS services.

Amazon Elastic Container Cloud (Amazon EC2)

Amazon EC2 provides scalable, on-demand computing capacity in the AWS cloud. With Amazon EC2, users can create and use virtual machines in the cloud, instantiating resources as needed to perform compute operations. Amazon EC2 is a common choice for rapidly deploying applications because it provides an excellent computing resource at a low cost [67], and it is possible to manage networks of different instances of Amazon EC2 virtual machines

Figure 4.6. Example of how *Amazon ECR* works in production and for pulling images [9]

through Amazon Virtual Private Cloud (Amazon VPC) and set their relative security, either on a per-instance basis or on an overall network basis. Additionally, it is possible to increase the capacity (scale up) of the instance after creation to handle computationally heavy tasks, such as spikes in website traffic. Conversely, if utilization decreases, capacity can be reduced (scale down). Amazon EC2 instances can be launched with AMIs, which are preconfigured templates containing the necessary components to use the server, including the operating system and additional software. AWS provides prebuilt AMIs, but it is also possible to create your own AMIs using containers. Furthermore, it is possible to connect to an Amazon EC2 instance through various communication systems, such as using Secure Socket Shell (SSH) keys provided at the time of instance creation.

4.2 Infrastructure Schema

All of the previously listed services have been useful, both as an active part in the project's realization and as potential options for the project's implementation, which will be analyzed below.

After reviewing the various services theoretically, it is now possible to understand a high-level look at how the various services interact with each other. The interaction system is intricate and consists of two main circuits. The image 4.7 shows the management of data from the TCU edge device in the first part. The data is sent to the cloud via the *IoT Core Thing*, inserted into a *Kinesis* channel, and then sent to a *Timestream* database in relevant tables.

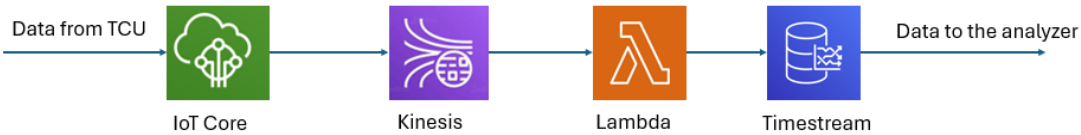

Figure 4.7. The high level rappresentation of the *AWS* services for the data managing

During the second part of the service interaction shown in the image 4.8, the system consists of a *CodePipeline* that can be triggered by a *CodeCommit* event

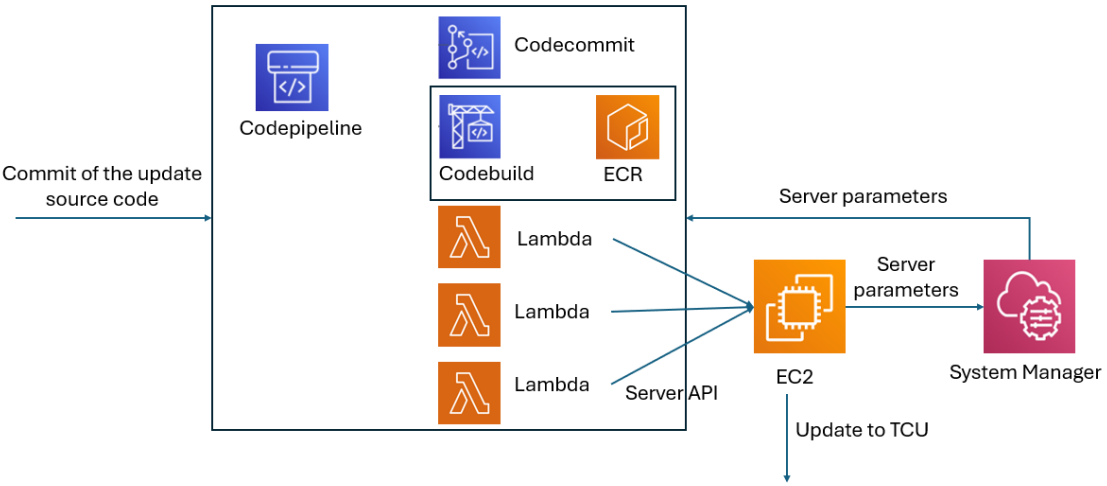

Figure 4.8. The high level rappresentation of the *AWS* services for the update managing

acting as a source. The pipeline includes a build phase via *CodeBuild*, which can utilize images from Amazon ECR registries, followed by three phases consisting of *Lambda* functions. The pipeline interacts with a server located on an Amazon EC2 instance, which stores its key information in the *Parameter Stor* of the *System Manager* service.

The components of the cloud structure may appear separate, indeed there is no actual interaction between data management services and update management services. However, the TCU device serves as the point of connection. It continuously sends data to the cloud, which can be analyzed, and receives updates. This process can continue indefinitely.

The following section provides a detailed analysis of the PoC structure, covering the edge device, cloud infrastructure, their connection, and data analysis system.

Chapter 5

Proof Of Concept

This chapter explores the PoC phase within the context of the thesis, aiming to validate the feasibility and efficacy of implementing SDV technologies. The main objective of this chapter is to translate theoretical concepts into concrete results, demonstrating the practical application of SDV in real-world situations. Through the PoC, we aim to confirm the fundamental principles and features of SDV, including its potential impact on vehicle performance, user experience, and overall safety.

The multifaceted nature of SDV requires a structured approach to its implementation, taking into account factors such as standardized hardware, cloud integration, and OTA updates. To achieve this, a PoC was designed to address these components individually and holistically, ensuring a seamless integration that aligns with the envisioned paradigm shift in automotive manufacturing. Furthermore, this chapter aims to demonstrate the collaborative efforts with industry-leading technologies and platforms, highlighting the strategic partnerships forged with key players in the automotive and software development sectors. By aligning with renowned entities, the PoC aims to leverage their expertise, technologies, and frameworks, thereby enhancing the robustness and scalability of the SDV ecosystem.

The PoC exploration utilizes the description of services and technologies provided by AWS in IoT, data management, and automotive industries, which are essential for the project's implementation.

Now, let's look at the components of the PoC in detail through a high-level architectural representation of the previously introduced elements and a more detailed analysis of the code that compose them.

5.1 Architectural design

This section delves into the heart of the project implementation, starting with a high-level view of the various systems that make up its realization and transitioning to a visualization of the interactions between the various elements.

The study and analysis of the SDV case study revealed that three fundamental elements were essential to create a concrete example of SDV implementation:

- **TCU simulator:** The TCU simulator is a system that replicates the basic functions of a telematic control unit (TCU). The system has the capability to send data packets and receive updates from a cloud server structure.
- **Cloud infrastructure:** The cloud infrastructure must be capable of managing both the data from the TCU and the update function.
- **Data viewer:** The data viewer is a platform that enables the visualization of manipulated and processed data in a manner that clearly displays changes in data behavior resulting from variations in the data and updates to the TCU.

With these three elements, a practical implementation made it possible to achieve what should happen in an SDV: having a vehicle system capable of updating itself via OTA updates. To support this implementation, it was necessary to use the services previously mentioned for creating the cloud infrastructure through the services made available by AWS.

5.1.1 TCU Simulator

This project considers a TCU to be a hardware system capable of generating data from one or more subsystems of a potential vehicle, collecting it, and then preparing it for transmission outside the vehicle. To familiarize with this structure and to design the sample project, it was necessary to simulate a TCU in a way that was as close as possible to a real system, without having the availability of a real vehicle. Therefore, to simplify the concept, a *Raspberry Pi* board was chosen as the TCU endpoint capable of generating telemetric data.

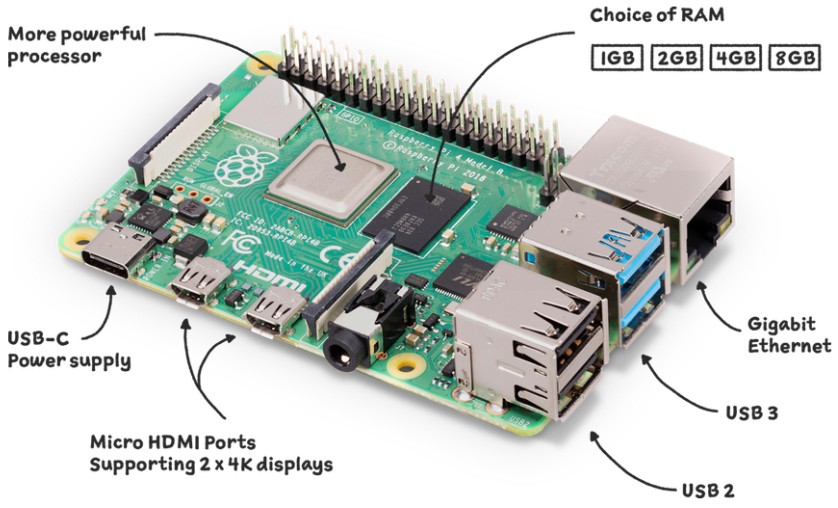

Figure 5.1. Illustration of a *Raspberry Pi* board with its periferics [10]

As shown in the figure above 5.1 a *Raspberry Pi* is a board containing everything necessary to function as an independent system to which various peripherals can be attached, making it the ideal abstraction of a general-purpose TCU needed

to implement the SDV case study. Before working with the raspberry, it was necessary to implement the project on a virtual machine, which greatly facilitated the development and testing of the written code on a PC with a different operating system and architecture than the *Raspberry Pi* board.

The simulator was thought to consist of four different software components, each with its own functionality, during its various stages of development:

- A component is responsible for connecting to the cloud server to send telematics data;
- A component for establishing a connection to the cloud server in order to update the telematics unit;
- A component for generating telematics data;
- A component for recognizing and managing local unit updates.

An external script manages the startup of the simulation unit via command line using the four items.

Analyzing these four elements in detail, the first element in chronological order was the component responsible for connecting the system to the cloud infrastructure via the MQTT protocol. This component required the presence of an active cloud-side IoT Core service, which, as we will discuss later, generates a certificate with an attached public and private key pair to ensure the device's identity. The consideration of the cloud infrastructure will be analyzed later when the cloud infrastructure is discussed.

In order to establish a connection to the cloud platform, the following code snippet from listing 5.1 is analyzed. The Python script launches and establishes a connection to the *AWS IoT Core* server using the function specified on line 1 from the designated library. This enables the data to be sent to the server. The code on line 12 creates an MQTT client. On line 13, the code configures the client's host name and port number. Finally, on line 14, the code is configured with the authentication certificates. It is important to note that these files are stored locally on the device and must be accessed by the linking script. The MQTT connection will be established on the topic specified on line 19. This will enable the AWS service, which will be connected to the topic through appropriate policies (as we will see later), to receive the transmitted data.

Listing 5.1. *MQTT* connection to the *AWS IoT Core AWS* service

```
1 from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
2 certificate_path="./Permanent/Certificates/"
3 def telemetry_handler():
4     global mqttc
5     global connection_event
6     VIN = "HawkbitDevice001" ##This is the Thing name
7     ENDPOINT = "**********.amazonaws.com" #This field contains
          the aws region
8     CERT_FILEPATH = f"{certificate_path}{VIN}.cert.pem"
```

```
9     PRIVATE_KEY_FILEPATH = f"{certificate_path}{VIN}.private.key"
10    ROOT_CA_FILEPATH = f"{certificate_path}root-CA.crt"
11    mqttc = AWSIoTMQTTClient(VIN)
12    mqttc.configureEndpoint(ENDPOINT, 8883)
13    mqttc.configureCredentials(ROOT_CA_FILEPATH,
          PRIVATE_KEY_FILEPATH, CERT_FILEPATH)
14    if mqttc.connect():
15        print("Connected␣to␣IoT␣core.␣Now␣the␣device␣sends␣its␣
              telemetry␣every␣1␣seconds")
16        connection_event.set() #Send connected signal to the main
17        publish_topic = f"device/{VIN}/telemetry"
```

The second item of analysis used in the realization of the simulator is the component that allows connecting to the OTA server to detect and download OTA updates. This was made possible by utilizing the 'Device Simulator' provided by *Hawkbit*. The simulator is a *Java* script that takes advantage of the *Hawkbit Server* interface to connect to the server and listen for updates specifically targeted to the device.

The code automatically establishes the connection when started through the simulation properties. For experimental design purposes, the device name is set statically as shown on line 2 of code fragment 5.2, while the server's Internet Protocol (IP) address to connect to is taken as input in the provided code 5.3, as shown on line 6.

Listing 5.2. Simulation properties of the *Hawkbit Device Simulator*

```
1  public static class Autostart {
2      private String name = "HawkbitDevice001";
3      private int amount = 1;
4      @NotEmpty
5      private String tenant = "DEFAULT";
6      private Protocol api = Protocol.DMF_AMQP;
7      private String endpoint = "http://localhost:8080";
8      private int pollDelay = (int) TimeUnit.MINUTES.toSeconds(30);
9      private String gatewayToken = "";
10     ...
11 }
```

Listing 5.3. Input arguments to set the ip of the *OTA* server to contact

```
1  public static void main(final String[] args) {
2      //take endpoint rabbit server as input
3      if (args.length > 0) {
4          String newHost = args[0];
5          if (!newHost.isEmpty()) {
6              System.setProperty("spring.rabbitmq.host", newHost);
7          }
8      }
9      SpringApplication.run(DeviceSimulator.class, args);
10 }
```

As demonstrated in the relevant sections of source code 5.4, when the device receives a download signal, it performs a series of security checks before starting to download the received data into the folder specified in line 4 of the code, using a stream that takes the incoming data from the link and places it in the selected folder, with the filename obtained from the download Uniform Resource Locator (URL).

Listing 5.4. Downloading files from the *OTA* server to the specific device simulator folder

```
1  private static long getOverallRead(final CloseableHttpResponse
       response, final MessageDigest md, final String url) throws
       IOException {
2      long overallread = 0L;
3      String[] urlParts = url.split("/");
4      File downloadFolder = new File("./TCU/downloads");
5      if (!downloadFolder.exists()) { //If "Download" folder
           doesn't exist
6          boolean created = downloadFolder.mkdirs();
7          if (!created) {
8              System.err.println("Error␣in␣the␣directory␣creation!");
9          }
10     }
11     File downloadFile = new File("./TCU/downloads/"+urlParts[10]);
12     try (FileOutputStream outputStream = new
           FileOutputStream(downloadFile);
13     final BufferedOutputStream bos = new BufferedOutputStream(new
           DigestOutputStream(outputStream, md))) {
14         try (BufferedInputStream bis = new BufferedInputStream(
               response.getEntity().getContent())) {
15             byte[] buffer = new byte[8192]; //byte dimension from
                   createBuffer of ByteStream.class
16             int bytesRead;
17             while ((bytesRead = bis.read(buffer)) != -1) {
18                 bos.write(buffer, 0, bytesRead); //Here only for
                       the md hash correctness.
19                 overallread += bytesRead;
20             }
21         }
22     }
23     return overallread;
24 }
```

During the TCU software update phase, at the time the server sends the update, the device in charge of listening for the update receives the update signal and the update download, the log file is written. The file is created in a specially created location each time the system is booted, or it is overwritten if it already exists. It will print a confirmation if the download was successful 5.2, and the cause of the error if the download was unsuccessful.

Figure 5.2. Update log file of the *Device Simulator*

The third component of the simulator is the heart of the TCU, which is the system that can generate the simulated vehicle data that changes in a progressive manner over time. This component has undergone several modifications during the course of development, in particular it was initially designed as a command line interface device written in *Python* language.

More specifically, as shown in Figure 5.3, in the first version of the simulator, once started, based on commands given by the user via the command line, it was able to increase or decrease its simulated speed to a limit until it stopped. The speed information, which varied over time, was sent every second to the component that manages communication with the *AWS IoT Core* server and then sent to the server.

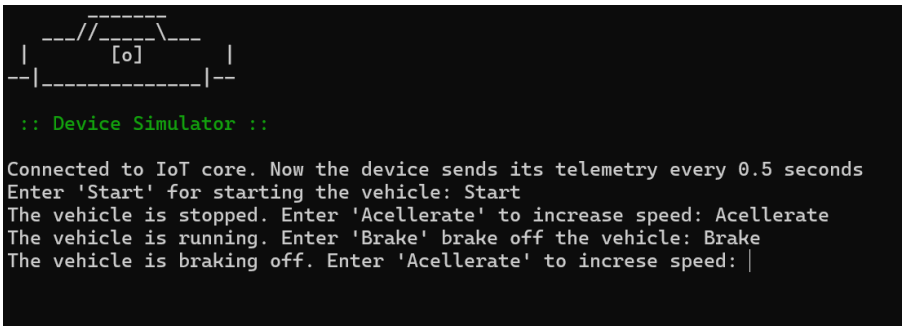

Figure 5.3. A snapshot of the first version of the *TCU* interface

In this case, there were two different versions of the speed handler code: the first one represented the initial state of the simulated vehicle, while the second one represented the vehicle after the update. In this way, it was clear that the update of the vehicle had occurred.

At a later stage, for the creation of the final *Python* TCU, it was decided to increase the number of telemetry data provided in such a way as to be able to collect and analyze a more realistic number of values in the cloud, but at the expense of a predefined data generation not decided by the user. This solution solved the fact that a real TCU would not present a graphical interface, but would simply collect the data generated by the subsystems present in the vehicle as a function of time. In this case, it was possible to build five subsystems capable of generating data every second in a way that can be considered as close as possible to a real system,

then this data is collected by an orchestrating script 5.5 that aggregates it to make it ready to be sent to the listening cloud service.

Listing 5.5. *TCU* orchestrator that collects data from other subsystems

```
1 for sub in subsystems:
2     values[sub.get_name()] = sub.get_info(t)
3     values["Timestamp"] = datetime.isoformat(datetime.utcnow())
4     values["DeviceID"] = f"{VIN}"
5     print(values)
6     messageFinal=json.dumps(values)
7     mqttc.publish(publish_topic, messageFinal, 0)
```

As shown in code 5.5, an iteration is performed on each subsystem present in the simulator, the data is collected in *Json* format and sent to the *AWS IoT Core* service through the previously seen object in script 5.1 to establish the connection with the service itself. This system allows you to have maximum modularity of the subsystems, therefore possibly being able to add new ones with future updates, the only constraint is to specify the list of present subsystems in the *Python* init 5.6 file and then add any new subsystems present.

Listing 5.6. *TCU* init file for the import of the *TCU* subsystems

```
1 from .airConditioning import AirConditioning
2 from .airbag import Airbag
3 from .heatedSeats import HeatedSeats
4 from .abs import ABS
5 from .engine import Engine
6 from .battery import Battery
7 subsystems = [Engine(), Battery(), AirConditioning(), Airbag(),
      HeatedSeats(), ABS()]
```

Now, we will examine one of the sample subsystems developed for the project and assess its primary functions. A simulator has been implemented to generate data from a hypothetical vehicle battery in the case of hybrid or fully electric vehicles. This subsystem reports three main pieces of information: the energy added to the battery through regenerative braking, the total energy stored in the battery at a given moment, and the battery temperature at a given instant. The vehicle is considered a system where an electric motor provides energy during acceleration, causing a decrease in the battery's energy. During deceleration, part of the energy is stored in the battery through regenerative braking. The battery's temperature varies over time due to these operations.

The functions listed in code block 5.7 are the most important parts of the class composing the subsystem simulator. These functions practically identify the common parts of each subsystem present in the project. The telemetry data is first aggregated in a *Json* format and then returned by these functions. Additionally, a function is required to return an identifying name of the subsystem. This name is useful to the orchestrator shown previously in 5.5 at line 2 for constructing the data packet to be sent to the server correctly.

Listing 5.7. Battery subsystem return code

```
1 def get_info(self, time):
2     return {
3         "StateOfCharge" : self.stateOfCharge,
4         "BatteryTemperature": self.batteryTemperature,
5         "EnergyAdded" : self.energyAdded,
6     }
7 def get_name(self):
8     return "Battery"
```

During the updating phase, this data 5.4, as well as all other subsystems, is created using a different algorithm. This ensures that when the telemetry data is analyzed, the download event is clearly visible. It is important to note that each *Python* subsystem has a related set of tests that can be utilized by the cloud structure, as analyzed further below.

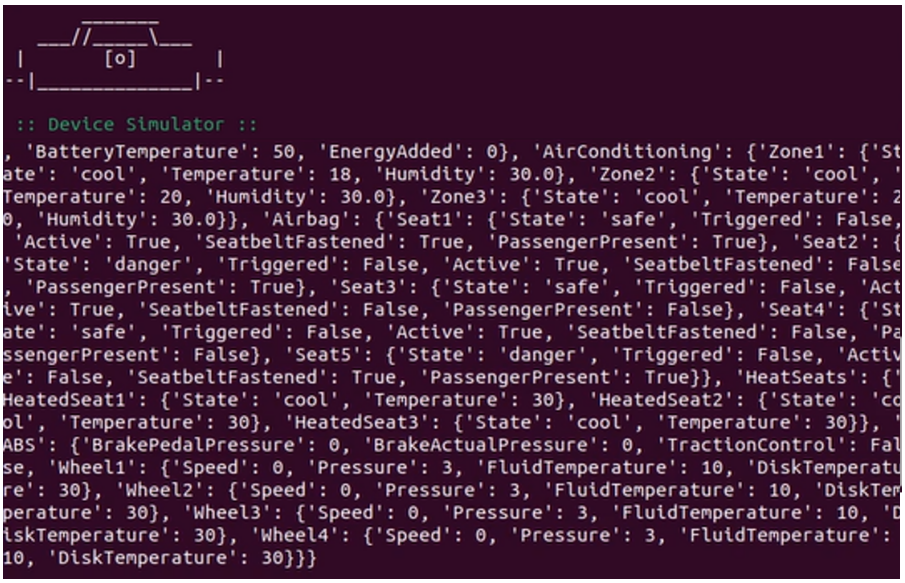

Figure 5.4. A snapshot of the *TCU* simulator on the virtual machine

In the final phase of the project, it was decided to modify the simulator to use a compiled script. This change makes the system more similar to a real world situation, since in a real environment there would be systems where it is not possible to have scripts based on interpreters such as the *Python* language, for reasons of size and resources, but where compiled programs are needed, which are more optimized. In order to do so and to give a concrete demonstration of the changes made to the infrastructure as analyzed in the following, it was decided to create a much simpler TCU simulator in *C* language (*C* simulator code).

In order to be compatible with both the first and second versions of the simulator (for backward compatibility), it was also necessary to adapt the orchestrator element 5.5 to be capable of recognizing the type of simulator to which it would be interfaced with. To make this possible, a specification file indicating the type of simulator used 5.8 was added to each TCU simulator.

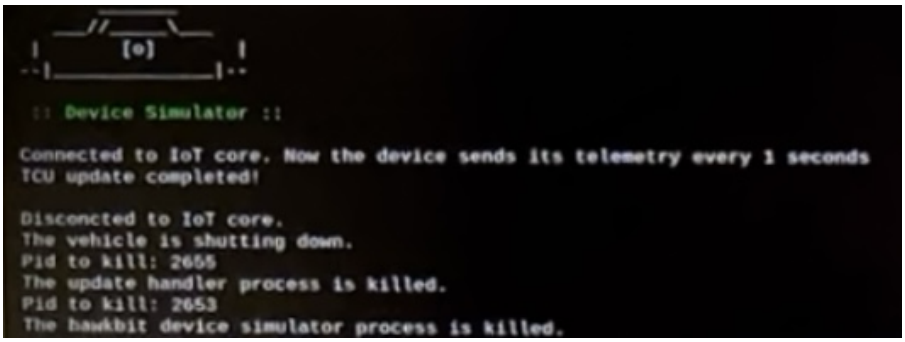

Figure 5.5. A snapshot of the *TCU* compiled simulator on the *Raspberry Pi* interface

Listing 5.8. Manifest example of compiled *TCU* simulator

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project>
    <name>C Project</name>
    <language>C</language>
    <compiler>gcc</compiler>
    <build_command>gcc main.c -o main.exe</build_command>
    <run_command>./main.exe</run_command>
</project>
```

The TCU simulator's download recognition system is the last element to be explored in detail. It is capable of activating downloads and ensuring that new elements are fully functional within the TCU system. Essentially, it functions as an edge agent, providing some of the tasks that the *AWS IoT Greengrass* service would have performed in parallel if *AWS IoT Greengrass* had been used on the edge device.

Listing 5.9. Main function of the update recognition system

```
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
folder_to_watch = './TCU' # Define the folder to monitor
def main():
    global observer
    while True:
        observer = Observer() # Create an observer
        event_handler = DownloadHandler()
        observer.schedule(event_handler, folder_to_watch,
            recursive=True) # Attach the event handler to the
            observer
        observer.daemon = True
        observer.start() # Start the observer in a separate
            thread
        observer.join() # Wait for the observer to finish
            operations

class DownloadHandler(FileSystemEventHandler):
```

```
15      def on_created(self, event):
16          update_file(event)
17
18      def on_modified(self, event):
19          update_file(event)
```

The *Python* script remains in an infinite loop, waiting for changes to the folder defined in line 3, using the libraries shown in lines 1 and 2 of code 5.9. If a folder is created or modified, and updates from the server infrastructure are downloaded (which will be analyzed in a later subsection), this program aims to detect the processes involved in the execution of the TCU simulator (as done in lines 3 and 12 of code 5.10) and kill the identified processes. Meanwhile, the update file will be identified by a compressed folder, as specified by the update pipeline (which will be analyzed later). The update files will then be extracted according to line 19 of the code. Finally, this component will reboot the entire system so that the update can be installed and take effect.

Listing 5.10. Code for performing actions when the designated download folder is changed

```
1  def update_file(event):
2      global observer
3      process = subprocess.Popen(["pgrep", "-f", "OS.py"],
           stdout=subprocess.PIPE, text=True)
4      output, _ = process.communicate()
5      pid_to_kill_list = output.splitlines()
6      if not event.is_directory and ('downloads' in
           event.src_path): #If the new item is a directory
7          print(f"src_path␣type:␣{type(event.src_path)}")
8          print(f"New␣file␣downloaded:␣{event.src_path}")
9          for pid_to_kill in pid_to_kill_list:
10             print(f"Pid␣to␣kill:␣{pid_to_kill}")
11             os.kill(int(pid_to_kill), signal.SIGTERM)
12         process = subprocess.Popen(["pgrep", "-f", "start.sh"],
               stdout=subprocess.PIPE, text=True)
13         output, _ = process.communicate()
14         pid_to_kill_list = output.splitlines()
15         for pid_to_kill in pid_to_kill_list:
16             print(f"Pid␣to␣kill:␣{pid_to_kill}")
17             os.kill(int(pid_to_kill), signal.SIGTERM) #Kill the
                   process
18         if '.zip' in event.src_path:
19             with zipfile.ZipFile(event.src_path, 'r') as zip_ref:
20                 zip_ref.extractall(folder_to_watch)
21         else:
22             shutil.copy(event.src_path, folder_to_watch)
23         print("File␣updated!")
24         observer.stop()
```

Figure 5.6. A snapshot of the *TCU* recognition module logs

The TCU simulator is composed of the three elements analyzed so far, and their evolution during the project made it possible to create a system compatible with systems based on *Arm* architectures, that is the *Raspberry Pi*, capable of generating telemetry data and sending them to the cloud server connected through *AWS IoT Core*, receiving updates, and rebooting the system to make the updates effective and active. Simulations were performed on virtual machines with "x86 architecture processors". The project will proceed to real test phases on actual systems once all necessary information has been acquired.

5.1.2 Cloud Infrastructure

Now the core of the project will be analyzed in detail, which is the cloud structure that allows, through the services offered by AWS previously introduced, both the connection of the vehicle to send, analyze and manipulate data telemetry, and the cloud structure that allows the execution of the *Hawkbit* server for the deployment of the update. This part of the project, as well as the previous one of the TCU simulator, experienced several changes during the implementation of the project, starting from a more experimental part of analysis and study of the various services, passing through a part of use more concretely linked to the development of the project, always relying on the console visualization tools, to arrive at a phase of creation of the entire structure through CDK, by executing the various stacks using *Python* script. To avoid unnecessary repetition of operations, to simplify things, it was decided to directly analyze the structure of the stack built with CDK. In the following sections, each component of the structure will be examined in detail, assuming that there is already a general and theoretical knowledge of the various services used.

The initial stack analyzed is related to the construction of *AWS IoT Core* services for device connectivity, that is, the connection of the TCU simulator to the cloud service for collecting telemetry data. In order to utilize the *AWS IoT Core* service, an *AWS IoT Core Thing* had to be defined, which can be described as the cloud-based version of the device's digital twin. After creating the IoT device assigning it a name (as shown in line 1 of code 5.11), policies were added to enable connection with the TCU simulator and exchange of telemetry data via the MQTT protocol (line 4).

Listing 5.11. Code for the creation of *AWS IoT Core Thing* and related policies

```
1  cfn_thing=iot.CfnThing(self, "HawkbitDevice",
2      thing_name="HawkbitDevice001"
3  )
4  cfn_policy = iot.CfnPolicy(self, "CfnPolicy", # Create a policy
       of the certificate
5      policy_document={
6      "Version":"2012-10-17",
7      "Statement":[{
8          "Effect":"Allow",
9          "Action":["iot:Connect"],
10         "Resource":[f"arn:aws:iot:"+region+":"+account+":client/"
               + cfn_thing.thing_name]
11     },
12     {
13         "Effect": "Allow",
14         "Action": ["iot:Publish"],
15         "Resource":[f"arn:aws:iot:"+region+":"+account+":topic/*"]
16     }]},
17     policy_name=f"{cfn_thing.thing_name}IoTCertPolicy",
18 )
```

Note that, in this and all other stacks, the region and account Identification (ID) are taken directly from the CDK functionality. The process of creating certificates was then implemented using *Boto3* functions to save resources. Code 5.12 certificates and their related keys are generated using a string seed and saved in a local directory for use in the physical device. The certificates are then returned to the CDK for use by the stack in saving to the *AWS IoT Core Thing*. The two-second waiting period at line 23 is a precautionary measure to ensure the completion of the certificate creation operation.

Listing 5.12. Code for the creation of *AWS IoT Core Thing* certificates and keys

```
1  import boto3
2  import os
3  SECRET_NAME = "****"
4  iot = boto3.client('iot', region_name='*****')
5  def on_create(thing_name):
6      response = iot.create_keys_and_certificate(setAsActive=True)
7      certificate_id = response['certificateId']
8      certificate_pem = response['certificatePem']
9      key_pair = response['keyPair']
10     directory_path="./certificates"
11     if not os.path.exists(directory_path):
12         os.makedirs(directory_path)
13     file_path = os.path.join(directory_path,
           f"{thing_name}.private.key")
14     with open(file_path, 'w') as file:
15         file.write(key_pair['PrivateKey'])
```

```
16      file_path = os.path.join(directory_path,
            f"{thing_name}.public.key")
17      with open(file_path, 'w') as file:
18          file.write(key_pair['PublicKey'])
19      file_path = os.path.join(directory_path,
            f"{thing_name}.cert.pem")
20      with open(file_path, 'w') as file:
21          file.write(certificate_pem)
22      time.sleep(2)
23      return {
24          'PhysicalResourceId': certificate_id,
25          'Data': {
26              'certificateId': certificate_id
27      }}
```

To enable the creation of security files via *Boto3*, separate from the CDK stack, a status variable was introduced to indicate the deploy or destroy status of the system 5.13. This was necessary to synchronize the deployment of the certificate and keys. Contrary to the rest of the mechanisms built into the CDK, the *Boto3* functions are independent and not affected by the CDK commands. If this state variable had not been used, there would have been a misalignment between CDK stack for the *AWS IoT Core Thing* creation and *Boto3* certificates. CDK functions can be used to destroy the certificate and keys as in line 5. Finally, the created certificate is linked to the relevant policy and to the *AWS IoT Core Thing* for proper usage.

Listing 5.13. Code for the creation and destruction of *AWS IoT Core Thing* certificates and keys from the *CDK* stack

```
1  if (status=="deploy"): # Creation of certificate with Boto3
2      certificate=cert_handler(cfn_thing.thing_name)
3  else:
4      certificate={"Data": {"certificateId": "Null"}}
5  print(f"{certificate['Data']['certificateId']}")
6  deactive_certificate_resource = cr.AwsCustomResource(self,
       "DeactiveCertificateResource",
7      on_delete=cr.AwsSdkCall(
8          service="Iot",
9          action="UpdateCertificate",
10         parameters={
11             "certificateId":
                   f"{certificate['Data']['certificateId']}",
12             "newStatus": "INACTIVE",
13     },),
14     policy=cr.AwsCustomResourcePolicy.from_sdk_calls(
15         resources=cr.AwsCustomResourcePolicy.ANY_RESOURCE
16 ))
17 delete_certificate_resource = cr.AwsCustomResource(self,
       "DeleteCertificateResource", # Destruction of certificates
```

```
18          service="Iot",
19          action="DeleteCertificate",
20          parameters={
21              "certificateId":
                    f"{certificate['Data']['certificateId']}",
22              "forceDelete": f"{True}"
23      },),
24      policy=cr.AwsCustomResourcePolicy.from_sdk_calls(
25          resources=cr.AwsCustomResourcePolicy.ANY_RESOURCE
26  ))
27  deactive_certificate_resource.node.add_dependency(
        delete_certificate_resource )
```

The next step in building the cloud infrastructure involves creating the necessary environment for the *Hawkbit* server. The idea is to use a basic Amazon EC2 machine to build the server on, which, as discussed later, will be contacted by the *AWS Codepipeline* via the server's Application Programming Interface (API) interfaces to deploy the updates. To create the Amazon EC2 instance, as shown in code 5.14, requires a Amazon VPC network in which to insert the instance itself as in line 1, an AMI that contains everything necessary for the computer to run properly (line 2), and a role that can provide the correct policies for the computer to function properly (line 3).

Listing 5.14. Code for the creation the *EC2 Hawkbit* server istance

```
1  vpc = ec2.Vpc(self, "VPC",
2      nat_gateways=0,
3      subnet_configuration=[ ec2.SubnetConfiguration(
           name="public",subnet_type=ec2.SubnetType.PUBLIC ) ]
4  )
5  generic_linux = ec2.MachineImage.generic_linux({ # AMI
6      'eu-west-1': 'ami-0905a3c97561e0b69',
7  })
8  role = iam.Role(self, "InstanceSSM",
       assumed_by=iam.ServicePrincipal("ec2.amazonaws.com"))
9  role.add_managed_policy(
       iam.ManagedPolicy.from_aws_managed_policy_name(
       "AmazonSSMManagedInstanceCore" ) )
10 instance = ec2.Instance(self, "Instance", # Instance
11     instance_type=ec2.InstanceType("t2.small"),
12     machine_image=generic_linux,
13     vpc = vpc,
14     role = role,
15 )
```

After creating the Amazon EC2 machine instance, it is necessary to modify its properties to ensure correct operation of the *Hawkbit* server. Specifically, two ports must be opened for Hypertext Transfer Protocol (HTTP) connections 5.15. This allows the server to be accessible to both the *CodePipeline* for deploying updates

from the *CodePipeline* to the server, and the TCU simulator device for downloading updates from the server to the edge device [68].

Listing 5.15. Code for opening the server doors

```
1 instance.connections.connections.allow_from_any_ipv4(
      ec2.Port.tcp(8080), "Allow␣inbound␣HTTP␣traffic" )
2 instance.connections.connections.allow_from_any_ipv4(
      ec2.Port.tcp(5672), "Allow␣inbound␣HTTP␣traffic" )
```

At this step, it is possible to insert commands directly into the created machine, which are interpreted as command-line inputs necessary to start the server via the *Docker* image 5.16. Specifically, a *Docker-compose* file is used to instantiate all the necessary server elements, including a database to record various information and a queue manager to manage external connections 5.17. Note that the properties mentioned in the code on line 36 are present in a local file. These properties are necessary to set the correct IP address in the server's *Docker* image interface. The *Docker-compose* file was created based on information from the *Hawkbit* guide.

Listing 5.16. Code to run commands on the machine

```
1  file_path = "./files/docker-compose.yml"
2  with open(file_path, 'r') as file:
3      docker_compose = file.read()
4  instance.user_data.add_commands(
5      'sudo␣apt-get␣update␣-y',
6      'sudo␣apt-get␣install␣-y␣docker-compose',
7      f'sudo␣echo␣-e␣"{docker_compose}"␣>␣
           /home/ubuntu/docker-compose.yml',
8      'sudo␣sed␣-i␣"s/\[server_ip_address\]/$(sudo␣curl␣-s␣
           http://169.254.169.254/latest/meta-data/public-ipv4)/g"␣
           /home/ubuntu/docker-compose.yml',
9      'sudo␣docker-compose␣-f␣/home/ubuntu/docker-compose.yml␣up␣
           -d',
10 )
```

Listing 5.17. *Hawkbit* server *Docker* compose

```
1  version: '3'
2  services:
3      rabbitmq:
4      image: 'rabbitmq:3-management'
5      restart: always
6      ports:
7          - '15672:15672'
8          - '5672:5672'
9      labels:
10         NAME: 'rabbitmq'
11     mysql:
12     image: 'mysql:8.0'
13     environment:
```

```
14          MYSQL_DATABASE: 'hawkbit'
15          # MYSQL_USER: 'root' is created by default in the
                container for mysql 8.0+
16          MYSQL_ALLOW_EMPTY_PASSWORD: 'true'
17      restart: always
18      ports:
19          - '3306:3306'
20      labels:
21          NAME: 'mysql'
22      hawkbit:
23      image: 'hawkbit/hawkbit-update-server:latest-mysql'
24      environment:
25          - 'SPRING_DATASOURCE_URL=␣
                jdbc:mariadb://mysql:3306/hawkbit'
26          - 'SPRING_RABBITMQ_HOST=rabbitmq'
27          - 'SPRING_RABBITMQ_USERNAME=guest'
28          - 'SPRING_RABBITMQ_PASSWORD=guest'
29          - 'SPRING_DATASOURCE_USERNAME=root'
30          - 'HAWKBIT_ARTIFACT_URL_PROTOCOLS_DOWNLOAD-HTTP_HOSTNAME=␣
                [server_ip_address]'
31          - 'HAWKBIT_ARTIFACT_URL_PROTOCOLS_DOWNLOAD-HTTP_IP=␣
                [server_ip_address]'
32      restart: always
33      ports:
34          - '8080:8080'
35      volumes:
36          - ./application.properties:
                /opt/hawkbit/application.properties
37      labels:
38          NAME: 'hawkbit'
```

The final step in the *Hawkbit* Amazon EC2 server stack involves saving the necessary parameters in the *Parameter Store* of the *System Manager* to connect the *AWS CodePipeline* with the *Hawkbit* server via its APIs. For example, let's consider the saving of one of the parameters, specifically the IP address of the Amazon EC2 machine in code 5.18. The interaction with the *System Manager* was performed using custom components of the CDK. The identifying name of the parameter, the value, and the type of the variable to be saved are indicated from line 5. In this case, the variable will simply be saved as a *String* since it does not need to be obscured, this is different from the user's password, which is saved as *SecureString*. The address is retrieved from the newly created instance using CDK functions in the line 7 of the code. The *Lambda* functions utilized in the *CodePipeline* will then access the saved parameters as will be examined later.

Listing 5.18. Code for the Parameter Store parameters creation

```
1   create_param3 = cr.AwsCustomResource(self, "CreateParam3",
2   on_create=cr.AwsSdkCall(
3       service="SSM",
```

```
4         action="PutParameter",
5         parameters={
6             "Name": "/hawkbitServer/ip_address",
7             "Value": f"{instance.instance_public_ip}",
8             "Type": "String"
9         },
10        physical_resource_id=
              cr.PhysicalResourceId.of("create_param3")
11    ),
12    on_delete=cr.AwsSdkCall(
13        service="SSM",
14        action="DeleteParameter",
15        parameters={
16            "Name": "/hawkbitServer/ip_address",
17        },
18        physical_resource_id=
              cr.PhysicalResourceId.of("delete_param3")
19    ),
20    policy=cr.AwsCustomResourcePolicy.from_sdk_calls(
21        resources=cr.AwsCustomResourcePolicy.ANY_RESOURCE
22 ))
```

Let's now analyze the construction of the pipeline for managing updates via *CodePipeline*. Two pipeline variants were elaborated during the development phase: one for managing updates via *Python* scripts, and one for managing compiled updates, written in the *C* language. To prevent description redundancy of the code, the two *CodePipeline*s will be analyzed step by step. Common stages will be explained only once, while the specifics of the reference *CodePipeline* will be discussed for the discordant parts. The *Stack* for updates in *C* language will serve as a reference for the common parts of *CodePipeline*. To begin, create a Codecommit repository for the pipeline source 5.19. The *CodeCommit* repository will contain the code and functionality to update the TCU simulator unit seen previously. The update will be created from a local *ZIP* file as shown in line 4. For the C-compiled pipeline, use the source code provided in the zip file for compilation. The *Python* script for the *CodePipeline Python* does not require compilation and will be ready to run in theory.

Listing 5.19. *CDK* Code for the creation of the *TCU* simulator *CodeCommit* repository

```
1 code_repo = codecommit.Repository(
2     self, "HawkbitDeviceC",
3     repository_name="HawkbitDeviceC",
4     code = codecommit.Code.from_zip_file(
          "./repos/Lorenzo_DeviceC.zip", "master" ) # Copies files
          from app directory to the repo as the initial commit
5 )
```

The *CodeCommit* repository is placed in the first stage of the pipeline, as shown in 5.20. Each *CodePipeline* stage can use an artifact as input and output. Since this

is the first stage, only an output artifact is set, as shown in line 2. The pipelines require a default artifact bucket, which is an Amazon S3 bucket or folder, as shown in line 1, and can be used by the pipeline if needed as will be shown in the actual pipeline creation phase later.

Listing 5.20. *CDK* code for the *CodeCommit* source stage set up

```
1  artifact_bucket = s3.Bucket.from_bucket_arn(self,
       f"codepipeline-{region}-****",
       f"arn:aws:s3:::codepipeline-{region}-****") #default
       codepipeline bucket
2  source_artifact = pipeline.Artifact("SourceArtifact")
3  source_stage = pipeline.StageProps(
4      stage_name="Source",
5      actions=[
6          pipelineactions.CodeCommitSourceAction(
7              action_name="CodeCommit",
8              branch="master",
9              output=source_artifact,
10             repository=code_repo,
11             variables_namespace="SourceVariables"
12 )])
```

The next stage requires separate and specific descriptions for both the *Python* and *C* pipelines. This stage concerns the build process. Specifically, for the build stage of the compiled update in *C*, it is essential to establish a dedicated *CodeBuild* stage for compiling *C* scripts since it is not directly supported by the *CodeBuild* service through the use of an Amazon ECR registry. A second supporting *CodePipeline* is created by following the steps shown in code 5.21. The first step involves creating a *CodeCommit* repository (at line 4) that contains the necessary information for building the new *CodeBuild* module to compile the script. This includes a *Dockerfile* for configuring the image with the necessary command to be created and a configuration file for the second pipeline build phase. The second stage of this pipeline involves the *Codebuild* stage (line 20), which is the actual creation of the image, placed into a special Amazon ECR (line 15). This Amazon ECR is then used in the update pipeline. It is important to note that during the creation of the second pipeline different roles are created with the related policies for the interaction between the different components.

Listing 5.21. *CDK* code for the *Codepipeline* for the *C* compiled file build creation

```
1  C_custom_code_repo = codecommit.Repository(
2      self, "HawkbitCCustomBuildImageRepo",
3      repository_name="HawkbitCCustomBuildImage",
4      code= codecommit.Code.from_zip_file(
           "./repos/HawkbitCCustomBuildImage.zip", "main" )
5  )
6  source_stage_c = pipeline.StageProps(
7      stage_name="Source",
8      actions=[
9          pipelineactions.CodeCommitSourceAction(
```

```
            action_name="CodeCommit",
            branch="main",
            output=source_artifact_c,
            repository=C_custom_code_repo,
    )])
ecr_repository = ecr.Repository(self, "HawkbitCBlogRepo",
    repository_name="hawkbit-C-blog",
    removal_policy=RemovalPolicy.DESTROY,
    auto_delete_images=True,
)
C_custom_build = codebuild.Project(
    self, "HawkbitCCustomBuildImageBuild",
    build_spec=codebuild.BuildSpec.from_source_filename(
        "buildspec.yml"),
    source=codebuild.Source.code_commit(
        repository=C_custom_code_repo,
        branch_or_ref="master"),
    environment=codebuild.BuildEnvironment(
        build_image=
           codebuild.LinuxBuildImage.AMAZON_LINUX_2_ARM_2,
        privileged=True),
    role=codebuild_role,
    project_name="HawkbitCCustomBuildImage",
    environment_variables={
        'ecr': codebuild.BuildEnvironmentVariable(
            value=ecr_repository.repository_uri),
        'tag': codebuild.BuildEnvironmentVariable(
            value='v1'),
    },
    timeout=Duration.minutes(60)
)
C_custom_pipeline = pipeline.Pipeline(
    self, "hawkbit-device-c",
    pipeline_name="hawkbit-device-c",
    artifact_bucket=artifact_bucket,
    stages=[source_stage_c, build_stage_c]
)
```

In the initial *CodePipeline*, a machine is built during the stage to compile the *C* script from the source stage using the image saved in the previously analyzed Amazon ECR. At this point, the build stage can compile and build the necessary *C* script for the update. The final result of this stage 5.22 is an executable in the pipeline that contains the functionality produced by the *C* script in the Source stage.

Listing 5.22. *CDK* code for the *Codecommit* build stage set up

```
hawkbit_build = codebuild.Project(
    self, "HawkbitDeviceBuildC",
```

```
3     build_spec=codebuild.BuildSpec.from_source_filename(
4         "buildspec.yml"),
5     source=codebuild.Source.code_commit(
6         repository=code_repo,
7         branch_or_ref="master"
8     ),
9     environment=codebuild.BuildEnvironment(
10        privileged=True,
11        build_image=
              codebuild.LinuxArmBuildImage.from_ecr_repository(
              ecr_repository, "v1")
12    ),
13    project_name="HawkbitDeviceBuildC"
14 )
15
16 build_stage = pipeline.StageProps(
17    stage_name="Build",
18    actions=[
19        pipelineactions.CodeBuildAction(
20            action_name="Build",
21            input=pipeline.Artifact("SourceArtifact"),
22            project=hawkbit_build,
23            outputs=[build_artifact]
24    )])
```

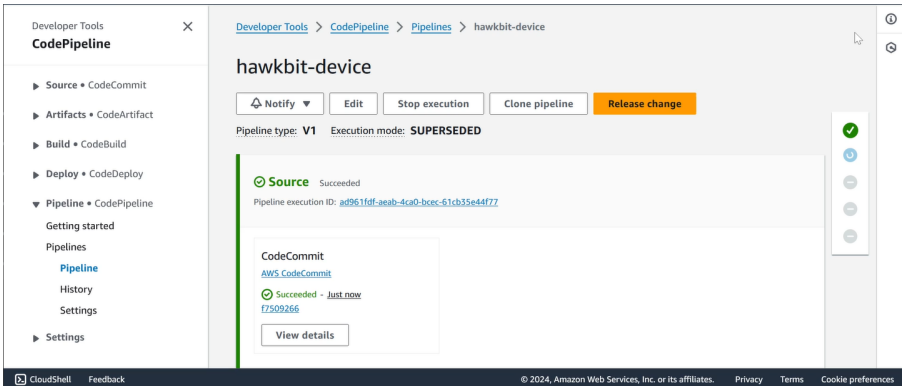

Figure 5.7. A snapshot of the *CodePipeline* source stage

Regarding the *Python* pipeline, the build stage is utilized as a test stage to launch tests built specifically for the repository code. This is due to the fact that *Python* scripts do not require actual code compilation. Using the pytest tool, it is possible to initiate the battery of tests present in the code repository. This information is located in the build configuration file within the repository.

The next stage of both pipelines contains *Lambda* functions that interact with the *Hawkbit* server through its API. To ensure greater modularity, there are three *Lambda* functions distributed over three different stages. As a first step, a dedicated role is created for each *Lambda* function. This role contains the necessary policies

for the function to perform its tasks once associated with the *Lambda*. Starting from the deployment stage of the software update on the *Hawkbit* server, it is evident in code 5.23 that the function is extracted from a local *ZIP* file and uploaded to the *Lambda* service, and this approach is followed for all subsequent *Lambda* stages.

Listing 5.23. *CDK* code for the deploy software on *Hawkbit* server *Lambda* creation

```
lambda_function = _lambda.Function(
    self,"hawkbitDeploySoftwareOnHawkbitServer",
    function_name="hawkbitDeploySoftwareOnHawkbitServer",
    runtime=_lambda.Runtime.PYTHON_3_11,
    code=_lambda.Code.from_asset(
        "./lambda/hawkbitDeploySoftwareOnHawkbitServer.zip"),
    handler="hawkbitDeploySoftwareOnHawkbitServer.lambda_handler",
    role=lambda_role,
    log_retention=logs.RetentionDays.ONE_DAY,
    timeout=Duration.seconds(60)
)
hawkbitDeploySoftwareOnHawkbitServer_stage = pipeline.StageProps(
    stage_name="hawkbitDeploySoftwareOnHawkbitServer",
    actions=[hawkbitDeploySoftwareOnHawkbitServer_invoke],
)
```

When analyzing the Lambda function, it can be seen that the calls to the *Hawkbit Server API*, after obtaining all the parameters necessary for execution, basically follow a fairly precise pattern. This is demonstrated in code 5.24 for building the software module, which is the component of the *Hawkbit* server that contains the update file. It starts by specifying the URL of the server, then specifies the information to send in the request payload, then specifies the headers containing the authentication information, and finally sends the HTTP request.

Listing 5.24. *Lambda* code for the software module creation

```
url = f"http://{server_ip}:8080/rest/v1/softwaremodules"
payload = json.dumps([{
    "name": project_name,
    "version": project_version,
    "type": "Application",
    "description": "Hawkbit␣device␣simulator␣module␣from␣
        codecommit",
    "vendor": "Reply",
    "encrypted": False
}])
headers = {
    'Content-Type': 'application/json',
    'Authorization': auth_header
}
response = requests.request("POST", url, headers=headers,
    data=payload)
if response.status_code != 201:
```

```
16      print(f"Error␣in␣the␣Software␣Module␣creation!␣␣Error:␣
            {response.status_code}")
17      traceback.print_exc()
18      put_job_failure(job_id, 'Function␣exception:␣')
19      return
20  print('Software␣Module␣creation␣completed')
```

After creating the software module, the update file is retrieved from the Amazon S3 bucket where it was saved in the source stage and loaded into the module. Then the distribution set is created following the same pattern as the previous software module. In this experimental project, the distribution set only contains one software module. It is important to note that any errors in these steps, like in any *Lambda* stage, will cause the pipeline to abort and report the error.

After analyzing the *Lambda* stage, the *Hawkbit* server now has a distribution set that includes a software module. This module contains the update file that needs to be downloaded to the TCU simulator device. The *CodePipeline* proceeds by creating two *Lambda* stages that perform similar operations: deploying the distribution set to the device connected to the OTA server. Specifically, in the first case reported in the code 5.25 a roll-out is performed, creating a fleet of devices to which the update is scheduled. The pool of devices on which to roll out the update is selected by reviewing the devices currently connected to the OTA server, so given the nature of the project, the pool will consist of only one edge device. In the second case, however, the update is directly assigned to the device. Both stages essentially perform the same operation, but they use different APIs of the *Hawkbit* server. Executing one stage excludes the execution of the other. The stages are designed to provide the user with the maximum API usability.

Listing 5.25. *Lambda* code for the roll out creation and execution

```
1  url = f"http://{server_ip}:8080/rest/v1/rollouts"
2  payload = json.dumps({
3      "createdBy": "Reply",
4      "createdAt": int(datetime.now().timestamp()),
5      "lastModifiedBy": "Reply",
6      "lastModifiedAt": int(datetime.now().timestamp()),
7      "name":
           f"{distribution_set['name']}{distribution_set['version']}",
8      "description": "Rollout␣on␣HawkbitDevice",
9      "targetFilterQuery": f"id=={target_names}*",
10     "distributionSetId": distribution_set['id'],
11     "amountGroups": 1,
12     "type": "forced"
13 })
14 headers = {
15     'Content-Type': 'application/json',
16     'Authorization': auth_header
17 }
18 response = requests.request("POST", url, headers=headers,
       data=payload)
```

```
19 url =
       f"http://{server_ip}:8080/rest/v1/rollouts/{rollout_id}/start"
20 payload = ""
21 headers = {
22     'Content-Type': 'application/json',
23     'Authorization': auth_header
24 }
25 response = requests.request("POST", url, headers=headers,
       data=payload)
```

As a last point in both analyzed CDK implementations the pipeline is declared with all the necessary stages so that it is built correctly. In practice with these stages it is possible to build or a test the source code as needed, and perform a deployment of the updates directly to the TCU simulator device through the use of the *Hawkbit* server and its exposed API. The final stack of the cloud infrastructure created by

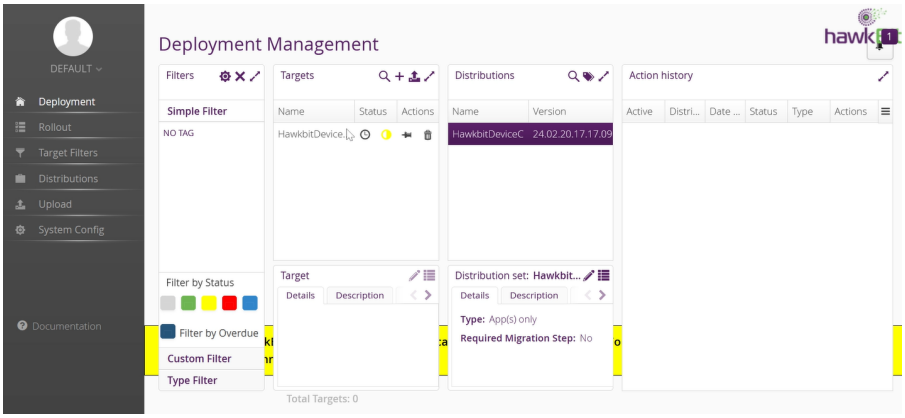

Figure 5.8. A snapshot of the *Hawkbit* server during the deployment of the update on the device

CDK is the actual data collection stack via *AWS Timestream*. In this stack, a *Kinesis Data Stream* is first created. The stream takes the data published to the channel created by the device via an Structured Query Language (SQL) query and routes it to a new destination. Next, the database and tables needed to store the telemetry data are created. Then, the *Kinesis* stream is used as input, triggering the start of a *Lambda* function that takes the output data from the stream as input, decompresses and formats it so that it can be handled by *Timestream*, and finally sends it to the corresponding tables, which at this point already exist because they were created previously. Upon analyzing code 5.26 in detail, it becomes clear that once the Kinesis Stram has been created, it is assigned a role for reading data from the channel (line 1) and then, via a rule to be executed, is given the very instruction to read data from the channel (line 14).

Listing 5.26. *CDK* code for the creation of the *Kinesis* stream with its role and rule

```
1 kinesis_stream = kinesis.Stream(self, "hawkbitDeviceData",
2     stream_mode=kinesis.StreamMode.ON_DEMAND,
3     stream_name="hawkbitDeviceData"
4 )
```

```
device_to_kinesis_role = iam.Role(self,
    "hawkbitDeviceToKinesis",
    assumed_by=iam.ServicePrincipal("iot.amazonaws.com"),
    role_name="hawkbitDeviceToKinesis")
device_to_kinesis_role.add_to_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["kinesis:*"],
    resources=[kinesis_stream.stream_arn],
))
device_to_kinesis_rule = iot.CfnTopicRule(self,
    "fromDeviceToKinesis",
    rule_name="HawkbitDeviceDataToKinesis",
    topic_rule_payload=iot.CfnTopicRule.TopicRulePayloadProperty(
        sql="SELECT␣*␣FROM␣'device/HawkbitDevice001/telemetry'",
        actions=[iot.CfnTopicRule.ActionProperty(
            kinesis=iot.CfnTopicRule.KinesisActionProperty(
                role_arn=device_to_kinesis_role.role_arn,
                stream_name=kinesis_stream.stream_name,
                partition_key="${DeviceID}"
),)]))
```

After creating the stream, the database is established. First, the general configuration settings are provided, including the data retention period and the identifying name. Then, the necessary tables shown in example code 5.27 are created one by one. In this case, there are 5 tables since the TCU simulator device has 5 subsystems.

Listing 5.27. *CDK* code for the creation of the battery table of the *Timestream* database

```
Battery_table = timestream.CfnTable(self, "Battery",
    database_name=database.database_name,
    schema=timestream.CfnTable.SchemaProperty(
        composite_partition_key=
            [timestream.CfnTable.PartitionKeyProperty(
            type="DIMENSION",
            enforcement_in_record="REQUIRED",
            name="DeviceID"
    )]),
    retention_properties=retention,
    table_name="Battery",
)
Battery_table.add_dependency(database)
```

The *Lambda* function is created at the end, with its policies attached to the role that was specifically created for it. As demonstrated in code 5.28, the *Lambda* is retrieved from a local file and imported into the AWS service. In this case, the *Lambda* function processes the received data by formatting the *Json* format into a flat one, so that there are no table sub-levels because they are not supported by the

Timestream service, and organizes the data into the appropriate tables by making an association between the data name and the table name.

Listing 5.28. *CDK* code for the creation of *Lambda* function that takes data from *Kinesis* stream and sends it to the *Timestream* tables

```
lambda_function = _lambda.Function(
    self,"hawkbitFromKinesisToTimestream",
    function_name="hawkbitFromKinesisToTimestream",
    runtime=_lambda.Runtime.PYTHON_3_11,
    code=_lambda.Code.from_asset(
        "./lambda/hawkbitFromKinesisToTimestream.zip"),
    handler="hawkbitFromKinesisToTimestream.lambda_handler",
    role=lambda_role,
    timeout=Duration.seconds(60),
)
lambda_function.add_event_source(eventSources.KinesisEventSource(
    kinesis_stream,
    batch_size=100, # default
    starting_position=_lambda.StartingPosition.LATEST
))
```

Now that the entire cloud infrastructure system has been established, it is possible to gain a better understanding of the interactions between its various components. The system begins with four basic elements: the *IoT Core Thing*, the *CodePipeline* pipeline, the *Hawkbit* server, and the *Timestrem* database. After creating the *IoT Core* device, it can retrieve data that will be stored in the *Timestrem* database. Updates can be deployed to the physical TCU simulator device through the Codepipeline pipeline and the *Hawkbit* server. After installation, the data received by *IoT Core* and stored in the database will undergo a change that can be detected during analysis. The process can be repeated indefinitely for any updates made to the code in the development repository.

5.1.3 Data Viewer: Grafana

Grafana was selected to visualize the data in the *Timestream* database. It natively supports linking to *Timestream* service as a data source by providing account credentials. To use *Grafana*, start a *Docker* container containing the server to construct graphs using data from the chosen source, in this case, the *Timestream* project.

To utilize the *Grafana* server, a separate virtual machine was created to connect to the *Timestream* database using AWS credentials and retrieve the stored data. By querying the *Timestream* tables, real-time graphs can be generated as the database is continuously updated. It is important to maintain a consistent update cadence between the *Timestream* database and the *Grafana* server. These queries produce a dataset that is plotted and interpreted differently depending on the type of graph used.

Real dashboards can be created by combining multiple graphs, even if they display the same data, to provide different perspectives. For the purposes of this

project, it was decided to create three dashboards for three different taballe to represent data from the three most relevant subsystems of the TCU simulator.

With the use of these dashboards as shown in Figure 5.9, it was possible to clearly represent the successful update of the TCU simulator device. For instance, this example shows the simulation of a vehicle's battery system. The dashboard displays two progressive data sets over time and one instantaneous data set. Specifically, the bottom left corner of the dashboard shows the battery temperature over time. From this information, it is evident that the temperature exhibits greater oscillations following the update, while remaining around a precise value. The total amount of energy present in the battery is also displayed at the bottom right, although it is not possible to detect the presence of the update due to the instantaneous nature of the data. Above, the graph shows the progression of energy added to the battery over time. It is evident that after the update, the addition of energy to the battery increased significantly after one second. This observation is based on the detected data. With this update, the goal is basically to simulate an improvement in performance related to the recovery of energy from regenerative braking, and this is shown in the graph both with the fact that after a certain moment the energy present in the battery is higher, and with the fact that the total net energy accumulated by regenerative braking is higher after the update.

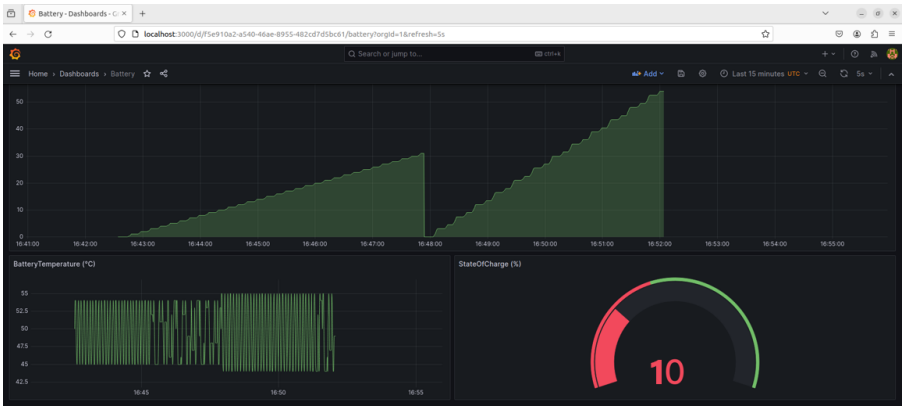

Figure 5.9. A snapshot of the *ABS* graph on the *Grafana* server

The dashboards for the Anti-lock Braking System (ABS) system of the TCU simulator are produced in *Graphana*. This allows for easy detection of updates through changes in the data. The data shows improved performance of regenerative braking, resulting in less energy needed for the physical brakes to apply to the discs. As a result, the simulated temperatures are lower. The dashboard for the vehicle's acceleration system is presented last.

To provide a complete summary of all the PoC elements, the system interacts as described below. Firstly, launch the CDK script to activate the supporting cloud infrastructure immediately. This will also generate the necessary certificates for device authentication. Secondly, after correctly positioning the certificates, it will be necessary to start the TCU simulator device by providing the correct IP address of the *Hawkbit* server to connect to for any future updates. Once a sufficient amount of data has been produced and analyzed in the *Grafafana* dashboard, updates can

be made. The *AWS CodeCommit* repository can be used to produce a specific code that represents the source of the update. After a commit is made on the master branch, the pipeline will deploy the update on the *Hawkbit* server and the device. Once the update is received, the TCU simulator will position the functions correctly and restart the system for the update to work effectively. At this point, there are two ways to evaluate the update: either through the log files produced directly on the device (if access to the physical device is available), or by analyzing the data produced by the TCU simulator after the update, which at this point will be different from the initial data. The execution of this complex system was confirmed during the testing phase with the demo on the real *Raspberry Pi* device, as will be seen later.

Chapter 6

Conclding Remarks

In this final chapter, first of all, the results obtained with the test demo of the project developed during the course are presented, then an analysis is made in relation to the objectives proposed by the thesis, evaluating what has been achieved and the future work that can be related to the project.

This chapter aims to be a conclusion for the thesis work done, both from the descriptive side by providing a summary of the work done, and from the practical side of project development, but also with the purpose of providing open options for furthering the project. The SDV technologies are very young and there are many development ideas. In addition, as an open project with the objective of creating a standard for the entire automotive industry, there are many opportunities for collaboration.

Let's start the description with the conclusions gathered with the final test demo of the project on *Raspberry Pi*.

6.1 Test and Validation on Raspberry Pi

To create the test demo, a *Raspberry Pi 4* board was used, with an architecture based on *Cortex-A72*, a processor from the *Arm* family. This device was found to be the best solution for simulating a modern automotive TCU physical system.

First of all, it was necessary to instantiate the cloud stacks of AWS services so that the supporting cloud structure was operational and ready to interact with the board. Once the necessary certificates for communication with the cloud platform had been obtained, it was possible to insert the communication scripts on the *Raspberry Pi*.

Once the script was started, the communication was successful and the generated data arrived at the cloud server ready to be analyzed, as shown in the image 6.1. The only difference compared to the simulations performed on a local virtual machine was the time it took to receive the data. In particular, in the case of the *Raspberry Pi*, there was a delay in sending data, although it was almost imperceptible and acceptable for the purposes of the test.

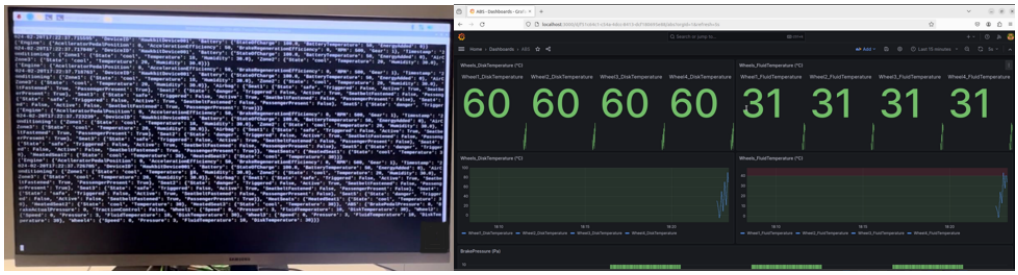

Figure 6.1. Comunication between the *Raspberry Pi* board and the *Grafana* server via the *AWS* cloud services

At this point, it was possible to start the update using the *AWS CodePipeline* service. In particular, once the update process was started on the pipeline, it was able to correctly contact the *Hawkbit* server where the TCU simulator was already present to start the update. Also in this case, it was necessary to wait a few more seconds for the update to be received on the device, but once this happened, the *Raspberry Pi* was able to interrupt the flow, receive the update, and restart the system with the updated features. As shown in the figure 6.2, the update was also detected by the data analysis server. Specifically, both the *Python* updates, which showed a drastic change in the data sent, and the update compiled in *C*, which caused an interruption in the sending of data, as expected by the update itself were launched on the demo board.

The presentation of test demos on the *Raspberry Pi* board was useful from several points of view and showed the following characteristics

- The ability of the system to communicate through the MQTT protocol across different connection networks.
- The capability of the software simulator to be used on different platforms, including *Arm*.
- The flexibility to adapt the cloud infrastructure to different conditions, whether it is a virtual machine or a physical board.
- Correct implementation of the pipeline capable of computing update projects written in compiled languages.

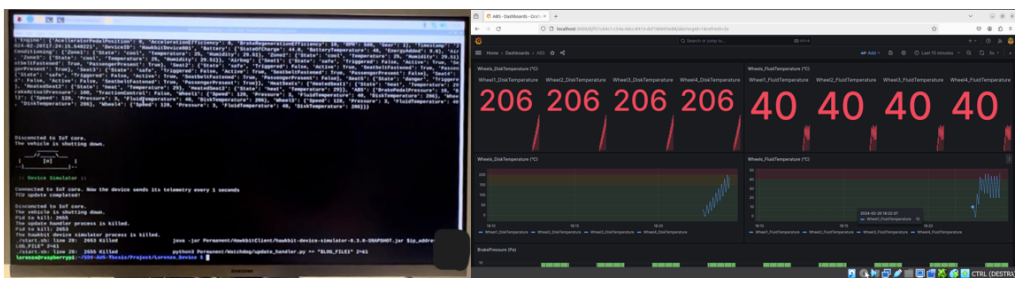

Figure 6.2. Update to the *Raspberry Pi* board and *Grafana* data after the update

6.2 Contribution Recaps

The aim of the thesis was to show the innovations introduced by the SDV technology, which is capable of completely revolutionizing the automotive industry. This result was achieved thanks to the support of the partner company, which provided the necessary resources to study, understand and analyze the cloud services offered by AWS, but also to put them into practice in the implementation of the project, which included technologies currently used by several companies in the automotive sector.

The SDV represents a true innovation in the automotive field, as it would allow the industry to move forward in creating more efficient and safer vehicles. In addition, it would open the door to a different and innovative development and production method that could significantly reduce production costs and times, thereby reducing the waste of necessary resources.

The thesis has certainly achieved its objectives, but let's now evaluate whether the objectives of the project, which was created to give a practical demonstration of the cooperation between the various elements involved in the SDV, have been achieved.

6.2.1 Are the PoC goals being met?

One of the main objectives of the proof of concept was to demonstrate how the cloud infrastructure composed of services provided by AWS could support the development of the SDV by using its resources. Certainly, this objective was fully achieved with the use of more than one pipeline responsible for managing updates. In particular, both the updates in *Python* language, easily portable from one platform to another, and the updates in compiled languages, more targeted and specific to each platform, but also more optimized and closer to the real use case, were successfully carried out.

Another important goal of the work was to be able to bring together different technologies to support the creation of the infrastructure. Specifically, the project succeeded in making AWS cloud services work with the *Hawkbit* system (already under development in several automotive companies) for managing the deployment of updates, and with *Grafana* services for data analysis. Achieving this goal made it possible to make the most of the technologies offered at the lowest possible cost to the company, obtaining a true ecosystem that is as open as possible.

As a final goal of the project, we can identify the need to introduce general purpose systems to support the development of the SDV. Also in this case, as seen before, the demo on the *Raspberry Pi* board showed that it can interact masterfully with different general purpose platforms and make the best use of the available resources.

In conclusion, the results obtained at the beginning of the implementation can be considered as achieved. However, it is important to emphasize that the PoC was a demonstration example, still in its early stages as far as actual production is concerned. The remaining points, deliberately left open due to limited resources, will be addressed below.

6.3 Future Works

Now the open aspects for the future development of the project are analyzed. In consideration of the nature of the work, some aspects were deliberately neglected in order to achieve a complete and functional result. If there had been an ambition to create a fully functional product on a real vehicle ready for use, not even a small part of the project could have been achieved with the available resources. Now let's talk about future work.

6.3.1 Transform the PoC in a product

To transform the proof of concept into a real product, usable in an automotive system capable of producing vehicles, it is necessary to cover three different steps: interfacing with a real vehicle to study the operational dynamics of a real TCU with all the subsystems that compose it, delving into the detailed analysis of the free software used such as *Hawkbit*, and managing additional elements of the vehicle such as machine learning or cockpit applications. A brief overview of each of these future works is given below.

1. Making the system usable with a real vehicle is essential to creating a market-ready commercial product. A real vehicle is made up of dozens, if not hundreds, of subsystems interconnected at various levels, and the management of all the data produced may be slightly different from what is seen in the PoC. The telemetry systems of real vehicles contain security systems that prevent direct connection to each individual subsystem, for example through firewalls; this is another aspect to consider when building a real product.

2. Another fundamental aspect for the creation of a concrete product is the in-depth analysis of the operational dynamics of the free software used in the development of the infrastructure. In particular, with regard to the *Hawkbit* server, in the PoC, after a not too detailed analysis of the various components, it was decided to use a pre-built *Docker* image. This made it easier to manage the elements, since everything was ready, but limited the freedom of customization. To fully exploit the potential of these technologies, it will be essential to fully customize the software used, as happens in real companies.

3. Last, but not least, is the adaptability of the SDV system to additional elements of the vehicle, such as machine learning systems for decision making or cockpit systems. For the PoC to become a usable product in reality, it is essential that it is fully integrated into the vehicle. Therefore, it is inconceivable that there are systems in the vehicle itself that do not interface with the SDV system, especially if they are other systems strictly related to IT, such as those mentioned above.

In conclusion, the full integration of the PoC with a real vehicle represents an essential phase in the development of the technologies covered in the thesis. This would allow vehicles to be transformed into fully upgradeable devices, thus contributing to the improvement of efficiency and safety, fundamental elements in the automotive sector.

Bibliography

[1] (2021) Sdv: Software defined vehicles. IEEE. Accessed: 01 03, 2024. [Online]. Available: https://cmte.ieee.org/futuredirections/2022/11/01/sdv-software-defined-vehicles/

[2] J. Scheibmeir, S. Sicular, A. Batchu, M. Fang, V. Baker, and F. O'Connor, "Magic quadrant for cloud ai developer services," *Gartner*, 2023. [Online]. Available: https://pages.awscloud.com/Gartner-Magic-Quadrant-for-Cloud-AI-Developer-Services.html?trk=d59e704f-4f30-4d43-8902-eb63c3692af4&sc_channel=el

[3] (2022) Building an automotive embedded linux image for edge and cloud using arm-based graviton instances, yocto project, and soafee. Luke Harvey Marcio da Ros Gomes and Sebastian Goscik. Accessed: 01 03, 2024. [Online]. Available: https://aws.amazon.com/blogs/industries/

[4] (2023) Architecture. SOAFEE project and Matt Spencer. Accessed: 01 03, 2024. [Online]. Available: https://architecture.docs.soafee.io/en/latest/contents/architecture.html

[5] (2024) What's the aws sdk for javascript? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/welcome.html

[6] (2024) What is aws iot? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[7] (2024) Aws codepipeline. AWS. Accessed: 06 03, 2024. [Online]. Available: https://aws.amazon.com/codepipeline/

[8] (2024) Amazon kinesis data streams terminology and concepts. AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html

[9] (2024) Amazon elastic container registry. AWS. Accessed: 06 03, 2024. [Online]. Available: https://aws.amazon.com/it/ecr/

[10] (2024) Raspberry pi 4. Raspberry Pi. Accessed: 07 03, 2024. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[11] (2022) Production statistics. OICA. Accessed: 01 03, 2024. [Online]. Available: https://www.oica.net/category/production-statistics/

[12] M. KARLSSON and L. SCHÖNBECK, "Department of technology management and economics," *CHALMERS UNIVERSITY OF TECHNOLOGY*, p. 11, 2018. [Online]. Available: https://odr.chalmers.se/server/api/core/bitstreams/077c3440-c033-418e-92ed-eda5dd638c5f/content

[13] N. B. Ruparelia, "Cloud computing," *The MIT Press Essential Knowledge Series*, p. 278, 2016. [Online]. Available: https://ebookcentral.proquest.com/

lib/polito-ebooks/reader.action?docID=4527741&query=cloud+computing+

[14] (2015) Elon musk: Model s not a car but a 'sophisticated computer on wheels'. Los Angeles Times. Accessed: 01 03, 2024. [Online]. Available: https://www.latimes.com/business/autos/la-fi-hy-musk-computer-on-wheels-20150319-story.html

[15] B. QNX, "What is a software-defined vehicle?" *Software-Defined Vehicles*, 2024. [Online]. Available: https://blackberry.qnx.com/en/ultimate-guides/software-defined-vehicle

[16] I. Society, "Functional safety," *ISO 26262-1:2018 Road vehicles*, no. 2, 2018. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:68383:en

[17] C. D. D. O. E. N. Fong and P. E. Black, "Impact of code complexity on software analysis," *NIST IR*, vol. 8165, no. upd1, pp. 1–12, 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8165.pdf

[18] (2024) Who we are. Storm Reply. Accessed: 01 03, 2024. [Online]. Available: https://www.reply.com/storm-reply/en/

[19] AWS, "Aws global infrastructure," *About-aws*, 2024. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/

[20] D. Slama, A. Nonnenmacher, and T. Irawan, "The software-defined vehicle," *A Digital-First Approach to Creating Next-Generation Experiences*, pp. 1–6, 2023. [Online]. Available: https://www.bosch-mobility.com/media/global/mobility-topics/mobility-topics/software-defined-vehicle/

[21] (2024) What is a software-defined vehicle in your opinion? Achim Nonnenmacher and Lead Product. Accessed: 01 03, 2024. [Online]. Available: https://www.bosch-mobility.com/en/mobility-topics/software-defined-vehicle/?gad_source=1

[22] (2024) What is software defined vehicle? Renault Group. Accessed: 01 03, 2024. [Online]. Available: https://www.renaultgroup.com/en/news-on-air/news/all-about-software-defined-vehicle/

[23] (2024) Do you have the right tools? Arm. Accessed: 01 03, 2024. [Online]. Available: https://www.arm.com/developer-hub/embedded-systems/automotive-tools

[24] A. K. Srivastava, K. CS, D. Lilaramani, R. R, and K. Sree, "An open-source swupdate and hawkbit framework for ota updates of risc-v based resource constrained devices," *2nd International Conference on Communication, Computing and Industry 4.0*, p. 1, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9689433

[25] M. Helmy and M. Mahmoud, "Enhanced multi-level secure over-the-air update system using adaptive autosar," *International Conference on Computer and Applications*, p. 1, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10401797

[26] (2024) Vision. Autosar. Accessed: 01 03, 2024. [Online]. Available: https://www.autosar.org/about

[27] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "Mqtt version 5.0," *OASIS Standard*, pp. 10 – 13, 2020. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf

[28] (2024) What are mqtt components? AWS. Accessed: 01 03, 2024. [Online]. Available: https://aws.amazon.com/it/what-is/mqtt/

[29] B. J. R. G and R. P, "Iot based system to predict the defects of tires in heavy vehicle," *International Conference on Sustainable Communication Networks and Application (ICSCNA)*, p. 1, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10370342

[30] K. T. Selvi, N. Praveena, K. Pratheksha, S. Ragunanthan, and R. Thamilselvan, "Air pressure system failure prediction and classification in scania trucks using machine learning," *Second International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, p. 1, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9742716

[31] (2023) Stellantis grows software development network with new hub in poland. Stellantis. Accessed: 05 03, 2024. [Online]. Available: https://www.stellantis.com/en/news/press-releases/2023/february/stellantis-grows-software-development-network-with-new-hub-in-poland

[32] (2023) Achieving software-defined vehicles. SOAFEE project. Accessed: 01 03, 2024. [Online]. Available: https://www.soafee.io/

[33] (2023) Architecture overview. SOAFEE project and Matt Spencer. Accessed: 01 03, 2024. [Online]. Available: https://architecture.docs.soafee.io/en/latest/contents/overview.html

[34] A. S. Rebecca M. Blank, "The nist definition of cloud computing," *National Institute of Standards and Technology Special Publication 800-145*, p. 1, 2012. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[35] (2024) Use aws regions. AWS. Accessed: 05 03, 2024. [Online]. Available: https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/region-selection.html

[36] (2024) Selecting the right cloud for workloads-differences between public, private, and hybrid. AWS. Accessed: 05 03, 2024. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/public-sector-cloud-transformation/

[37] (2024) What is cloud computing? AWS. Accessed: 01 03, 2024. [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/

[38] (2024) Amazon web services. AWS. Accessed: 01 03, 2024. [Online]. Available: https://www.aboutamazon.eu/what-we-do/amazon-web-services

[39] (2024) Cloud computing with aws. AWS. Accessed: 01 03, 2024. [Online]. Available: https://aws.amazon.com/what-is-aws/

[40] (2024) About aws. AWS. Accessed: 01 03, 2024. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/gxp-systems-on-aws/about-aws.html

[41] (2024) Aws cloud security. AWS. Accessed: 01 03, 2024. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/gxp-systems-on-aws/aws-cloud-security.html

[42] (2024) Aws cloud security. AWS. Accessed: 01 03, 2024. [Online]. Available: https://aws.amazon.com/security/

[43] (2024) Aws certifications and attestations. AWS. Accessed: 01 03, 2024. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/gxp-systems-on-aws/aws-certifications-and-attestations.html

[44] I. o. i. a. Amazon.com, "System and organization controls 3 (soc 3) report," *Report on the Amazon Web Services System Relevant to*

Security, Availability, Confidentiality, and Privacy For the Period October 1, 2022 - September 30, 2023, pp. 1–53, 2023. [Online]. Available: https://d1.awsstatic.com/whitepapers/compliance/AWS_SOC3.pdf

[45] (2024) Fedramp. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/fedramp/

[46] (2015) Iso 9001:2015 quality management systems requirements. ISO. Accessed: 05 03, 2024. [Online]. Available: https://www.iso.org/standard/62085.html

[47] (2024) Iso 9001:2015 compliance. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/iso-9001-faqs/

[48] (2024) Iso/iec 27001:2022. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/iso-27001-faqs/

[49] (2015) Iso/iec 27017:2015 information technology security techniques. ISO. Accessed: 05 03, 2024. [Online]. Available: https://www.iso.org/standard/43757.html

[50] (2024) Iso/iec 27017:2015 compliance. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/iso-27017-faqs/

[51] (2024) Iso/iec 27018:2019 compliance. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/iso-27018-faqs/

[52] (2024) Hitrust csf. AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/hitrust/

[53] (2024) Cloud security alliance (csa). AWS. Accessed: 05 03, 2024. [Online]. Available: https://aws.amazon.com/compliance/csa/

[54] (2024) What is the aws cdk? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/cdk/v2/guide/home.html

[55] (2024) What is aws iot greengrass? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html

[56] (2024) Aws identity and access management documentation. AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/iam/

[57] (2024) Aws lambda documentation. AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/

[58] (2024) What is amazon cloudwatch? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

[59] (2024) What is aws codebuild? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html

[60] (2024) Aws codecommit documentation. AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/codecommit/

[61] (2024) What is amazon s3? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

[62] (2024) What is amazon kinesis data streams? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/streams/latest/dev/introduction.html

[63] (2024) What is amazon timestream? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/timestream/latest/developerguide/what-is-timestream.html

[64] (2024) What is amazon dynamodb? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[65] (2024) Aws systems manager documentation. AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/systems-manager/

[66] (2024) What is amazon elastic container registry? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html

[67] (2024) What is amazon ec2? AWS. Accessed: 06 03, 2024. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

[68] (2024) From docker image. eclipse. Accessed: 10 03, 2024. [Online]. Available: https://eclipse.dev/hawkbit/gettingstarted/