



**Leonor Rodrigues Galhoz Vieira Grosso**

Bachelor's Degree in Electrical and Computer Engineering Sciences

**Cloud Rule-based System  
for Analysis of IoT Data  
in a Big Data Context**

Dissertation submitted in fulfilment  
of the requirements for the degree of

Master of Science in  
**Electrical and Computer Engineering**

Adviser: João Paulo Pimentão, Associate Professor,  
NOVA University of Lisbon

Co-adviser: Sérgio Miguel da Silva Barata Onofre, Project  
Manager, Holos S.A.



September 2021



### **Analysis of IoT data in a Big Data context**

Copyright © Leonor Rodrigues Galhoz Vieira Grosso, Faculty of Sciences and Technology, NOVA University Lisbon. The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.



*To my little peanut.  
May technology keep making the world a better and safer place  
for you and all the little peanuts who will come after you.*



## Acknowledgements

Firstly, I would like to thank Sérgio Onofre, Ph.D., without him I would have never followed the path that got me here today. Since I started this degree, almost six years ago, he helped me whenever I needed with his knowledge and words of support and for that I am beyond thankful. Secondly, I would like to thank Pedro Lima Monteiro, M.Sc., for always being ready to help me throughout this thesis with his great input, ideas and knowledge. This project would not be finished without his precious advice.

I also want to thank my adviser and Holos' CTO João Paulo Pimentão, Ph.D., for enabling me to work on this subject with him; NOVA School of Science and Technology for being a second home for the last years and all its professors who shared with me the knowledge I needed to become a Master of Science.

To my mother and my boyfriend, a big thank you. They are my support system. I could never thank them enough for their patience, motivation, understanding and nurture. To my father, thank you for the discipline. To my brothers, thank you for your interest in my work, your curiosity made me want to learn even more about this. Last but not least, I want to thank my grandma, although she doesn't understand anything about my work, her care was fundamental, even when it was as simple as bringing me peeled fruit to eat.

Finally, I want to thank my friends and colleagues, especially, my friend Miguel Vaqueiro, for being my rock and always being there for me, and my co-worker André Jardim, who went through this insane adventure with me.

Lastly, an honourable mention goes to Jamie xx whose music was the fuel to my brain on the long nights writing this thesis.





## Abstract

---

Nowadays, enormous amounts of information are produced, on a daily basis, by sensors. Information which, after being analysed, is transformed from simple data into knowledge which, in itself, can be an asset to those who can take advantage of that knowledge.

An example of this situation is the data being generated by sensors installed on trains, that can be analysed to different ends, one of which, the condition-based maintenance of trains.

Condition-based maintenance takes advantage of data to understand the current state of mechanical equipment, avoiding unnecessary replacements or preventing accidents consequent of late maintenance.

In this dissertation, it is presented an architecture which integrates a rule-based system functioning over cloud applications that analyses all the data that's being acquired by the trains' sensors in a way that, whenever a specific set of conditions is met alerts are activated, so the train operators, the mechanics in charge and all their staff know how to proceed.

This architecture is to be created on a cloud environment since, with this vast amount of data being generated, these highly scalable environments assure that data processing performance isn't compromised and that all this data is analysed in a timely manner, taking advantage of all its computational components.

The process of creating this architecture is demonstrated step by step and the test results are presented and analysed.

**Keywords:** Big Data; Internet of Things; Cloud Technologies; Industry 4.0; Google Cloud Platform; Rule-based System; Condition-based Maintenance; Railway.

---



## Resumo

---

Nos dias de hoje são produzidas, diariamente, enormíssimas quantidades de informação por parte de sensores, informação essa que após analisada se transforma de simples dados em conhecimento que, por si, é uma mais-valia para quem pode fazer uso desse conhecimento.

Um exemplo destes casos são os dados produzidos pelos sensores instalados nos comboios, que podem ser analisados com variadas finalidades, uma delas, a manutenção baseada na condição.

A manutenção baseada na condição tira proveito dos dados para compreender o estado dos equipamentos, evitando substituições desnecessárias ou prevenindo acidentes consequentes de manutenções tardias.

Nesta dissertação é apresentada uma arquitetura para o funcionamento de um sistema de regras na cloud, que analise todos estes dados que estão a ser adquiridos nos sensores dos comboios de forma que, quando certas condições são cumpridas, alertas sejam ativados e os operadores dos comboios, os mecânicos responsáveis e toda a equipa envolvente saibam como atuar.

Esta arquitetura quer-se criada na cloud pois, com uma quantidade enorme de dados a ser gerada, estes ambientes altamente escaláveis garantem que o desempenho no processamento dos dados não fique comprometido. Garantem também que os intervalos de tempo necessários para analisar todos estes dados sejam muito pequenos, tirando partido do poder computacional disponível em tais ambientes.

O processo de criação desta arquitetura é demonstrado passo a passo e os resultados obtidos são apresentados e analisados.

**Palavras-chave:** Big Data; Internet of Things; Tecnologias Cloud; Indústria 4.0; Google Cloud Platform; Sistema de Regras; Manutenção Baseada na Condição; Ferrovia.

---



# Contents

List of Figures	xv
List of Tables	xvii
List of Listings	xix
Glossary	xxi
Acronyms	xxiii
Introduction	1
1.1 Context and Motivation .....	1
1.2 The Problem.....	3
1.3 Proposed Solution .....	4
1.4 Outline of the Document.....	5
State of the Art	7
2.1 Big Data Analytics: a need .....	7
2.2 Big Data Architecture .....	8
2.3 Big Data Analysis Technologies.....	9
2.3.1 Hadoop	9
2.4 Big Data Processing.....	10
2.4.1 Batch Processing	10
2.4.2 Real-time Processing	11
2.5 ETL: Extract, Transform and Load.....	11
2.6 Cloud Computing: as a Service .....	12
2.6.1 Infrastructure as a Service	12
2.7 Google Cloud Platform Solutions.....	13
2.7.1 Google BigQuery	13
2.7.2 Google Cloud Logging	15
2.7.3 Google Cloud Pub/Sub	16
2.7.4 Google Cloud Functions	17
2.7.5 Google Cloud Scheduler	17
2.7.6 Google Cloud Firestore	18
2.8 Amazon Web Services Solutions.....	19
2.8.1 Amazon Redshift	19
2.8.2 Amazon CloudWatch	20
2.8.3 Amazon Kinesis	21

2.8.4	AWS Lambda	22
2.9	Google vs Amazon: who wins? .....	23
2.10	Rule-Based Systems .....	25
System Architecture		27
3.1	Data Acquisition .....	27
3.2	Data Filtration .....	29
3.3	Implemented Architecture .....	32
3.3.1	Other Architectures Considered	34
3.4	Rule System .....	37
Implementation and Results		41
4.1	Implementation of Rule 1 .....	42
4.1.1	Trials of Rule 1	44
4.2	Implementation of Rule 2 .....	46
4.2.1	Trials of Rule 2	48
4.3	Results and Impracticalities .....	50
Conclusions and Future Work		51
Bibliography		55

## List of Figures

Figure 1 – Representation of the growth of the digital universe throughout the years. ....	2
Figure 2 - Components of a Big data architecture. [14].....	8
Figure 3 - HDFS architecture. [16] .....	10
Figure 4 - Extract, transform and load. [20].....	12
Figure 5 - Relationship between Dremel, Jupiter, Colossus and Borg. [25] .....	13
Figure 6 - Dremel's columnar storage. [26].....	14
Figure 7 - Dremel's tree architecture. [26] .....	14
Figure 8 – Pub/Sub publisher-subscriber relationships. [32] .....	16
Figure 9 – Unix-cron schedule fields. [39].....	17
Figure 10 – Firestore ‘add a field’ prompt. [41].....	18
Figure 11 – AWS Redshift simplified architecture. [50] .....	20
Figure 12 - AWS Kinesis Data Analytics process. [56].....	21
Figure 13 - Scenario using Kinesis Data Streams and Kinesis Data Analytics. Adapted from [56] .....	22
Figure 14 - Hypothetical scenario to be applied on the trains' project. Adapted from: [56], [59] .....	23
Figure 15 - Revision of the possible Amazon scenario. Adapted from: [56], [59] .....	24
Figure 16 – Train composition and device distribution. ....	27
Figure 17 – Smart train monitor. [63] .....	28
Figure 18 – Data acquisition process from source to cloud. ....	28
Figure 19 – GCP Architecture: triggering the rule-based system. ....	32
Figure 20 – GCP Architecture: cycling the rule system.....	33
Figure 21 – Architecture schema for hypothesis 1.....	35
Figure 22 – Architecture schema for hypothesis 2.....	35
Figure 23 – Architecture schema designed to provide a solution for hypothesis 1.....	36
Figure 24 – Architecture schema designed to provide a solution for hypothesis 2.....	37
Figure 25 – Rule 1 flowchart.....	43
Figure 26 – Rule 1 insert log.....	44
Figure 27 – Rule 1 first verification function logs. ....	45
Figure 28 – Rule 1 second verification function logs. ....	45
Figure 29 – Rule 1 fact table data inserted.....	45
Figure 30 – Rule 2 flowchart.....	47
Figure 31 – Rule 2 insert logs. ....	48
Figure 32 – Rule 2 first verification function logs. ....	49
Figure 33 – Rule 2 second verification function logs. ....	49
Figure 34 – Rule 2 fact table data inserted.....	49
Figure 35 – Architecture diagram for designing a connected vehicle platform. [67] .....	53





## List of Tables

Table 1: Original BigQuery table schema.....	29
Table 2: Device source, event type and value correlation.....	30
Table 3: Example of the value field on a data entry from device -40 and -3030. ....	31
Table 4: Unique schematic fields of tables -40 and -3030 to -3033.....	31
Table 5: Fact table schema. ....	38
Table 6: Flowcharts' blocks guide. ....	41



## List of Listings

Listing 1: Query to check if the train is less than 1 km from the next station.....	42
Listing 2: Query to check if the train's speed is higher than 80 km/h.....	43
Listing 3: Query to check if the train's speed is 0 km/h.....	46
Listing 4: Query to check if the train had a 'Break Alert' in the last four minutes.....	47



# Glossary

**attribute**

A characteristic associated, in the scope of this thesis, with data.

**cloud**

General term used to describe a global network of computing resources available through the internet and managed by its providers.

**data**

Information in digital form that can be transmitted or processed; includes both useful and irrelevant information and must be processed to be meaningful.

**log**

A record of events.



## Acronyms

AGATE	Advanced Generic Alstom Transport Electronics
ANSI	American National Standards Institute
API	Application Programming Interface
AWS	Amazon Web Services
BST	British Summer Time
CBM	Condition-Based Maintenance
CD	Compact Disk
CPU	Central Process Unit
EaaS	Everything as a Service
ETL	Extract, Transform and Load
ELT	Extract, Load and Transform
FaaS	Function as a Service
GB	Gigabyte
GCP	Google Cloud Platform
GPS	Global Positioning System
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
ML	Machine Learning
NA	Not Applicable
PaaS	Platform as a Service
RIOM	Register Input Output Module
SaaS	Software as a Service
SQL	Structured Query Language

TB	Terabyte
TBD	To Be Defined
TOC	Train Operating Company
UTC	Coordinated Universal Time
UTS	Unité de Traitement Sono
XaaS	Anything as a Service





# Introduction

## 1.1 Context and Motivation

In 2011, the term ‘Industry 4.0’ emerged at the Hannover Fair, in Germany [1]. Since then, the i4.0 era, also known as, the fourth industrial revolution has officially began and, consequently, the world became even more digital, especially due to the evolution of Internet of Things (IoT) [2] and cloud computing [3]. All around the globe, industries have become more digitized and, because of that, more and more digital data is being produced, data that needs to be sorted, analysed and interpreted. IoT sensors are one of the main sources of digital data today and, each second, so much data is being produced that the traditional software and hardware solutions are too weak to be able to analyse it all and keep up with it. Instead of just having smartphones, computers and tablets connected to the internet, thanks to IoT, objects that go from household appliances, cameras, watches, vehicles, books, medical prosthetics or even clothes can be connected to the internet [4]. The data collected from all these new kinds of sensors and gets uploaded online, every second of every day. Companies everywhere are understanding the potential of analysing all this data and deciding to take advantage of that.

When the i4.0 era started, the digital universe (term given to the collection of all digital data that exists), had around 1.2 zettabytes<sup>1</sup> of data [5]. By 2018, according to the International Data Corporation, it was 33 zettabytes. To put that into perspective, that is, approximately, as much as 47 trillion 80 minutes CDs. The digital universe got 27.5 times bigger in less than a decade, with most of its growth being the consequence of IoT evolution, and the estimate for 2025 is for it to grow up to 175 zettabytes [6], or 250 trillion CDs, proving that IoT is just going to become a bigger reality, not only in people’s private lives but, also, in enterprises who try to keep up with this current evolution and make use of IoT to evolve and transform their companies’ products.

---

<sup>1</sup> One zettabyte is equivalent to  $10^{21}$  bytes.

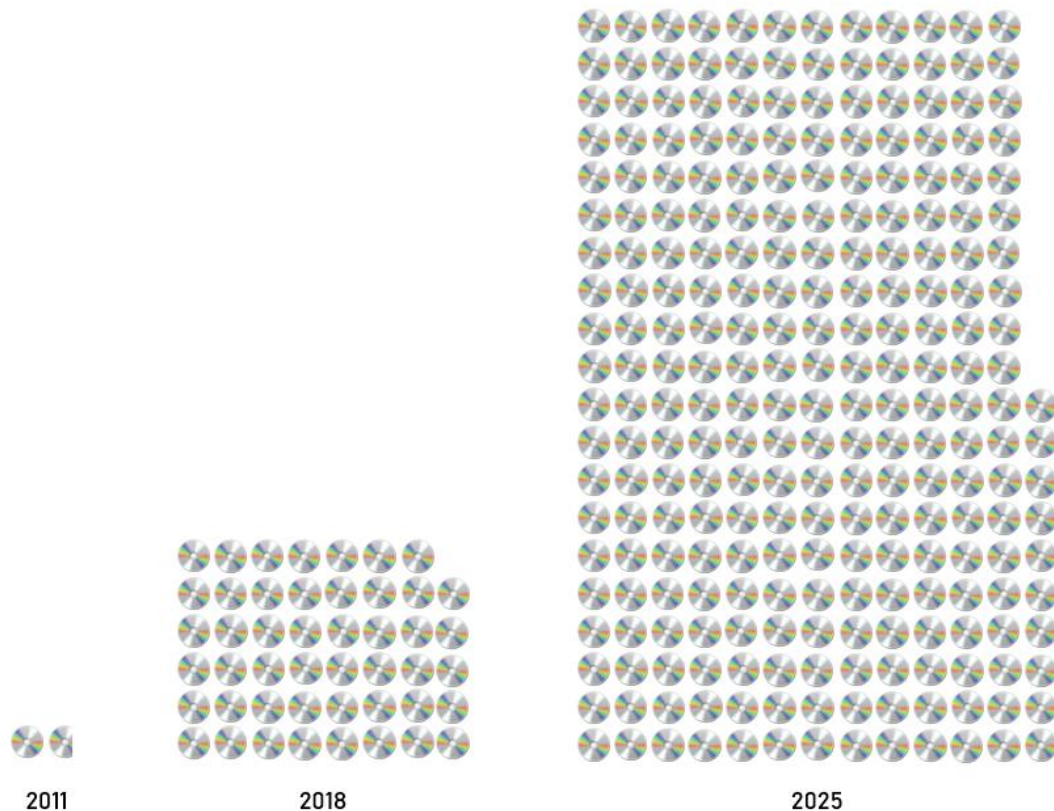


Figure 1 – Representation of the growth of the digital universe throughout the years.

Consequently, companies who decide to embrace IoT and analyse the data it's producing have to be conscious that they might face a Big data [7] issue caused by the quantity of information the IoT sensors might collect.

However, Big data isn't just about the amount of data being enormous, sometimes even inconceivable amounts, but so much more defines what in reality makes data big. Big data was initially defined by three main keywords: volume, velocity and variety. Volume defines the amount of data there is in the dataset, i.e., the group of data. Velocity is related to the frequency and periodicity in which the data is being received, it can be in real time, or in batches. The sensors and other devices can be collecting data in intervals of milliseconds, minutes or any other time interval defined as the appropriate one. Variety can relate to the variety of sources and the types of data, since data can come from the same or different source and it can come in a structured way (with tags that make it easier to sort), unstructured (when it is random), and semi-structured [7]. Nowadays there are much more Vs, the keywords that define Big data, with some of the most relevant being Veracity, if the source is reliable, and Value, what information can be obtained from the data in order to profit on it [8]. Validity is also relevant to this specific situation and it's when one tries to assess if the data in the set is precise and correct for the analysis in place.

Well, all this data must be stored somewhere and that's where cloud computing comes to shine. Cloud computing is here to transform the way people store data and work on the internet. Instead of having programs and data locally installed and stored in their computers, cloud computing made it possible to have it stored on data centres that can be spread out around the globe [9]. This evolution brings advantages such as easier ways to share documents as to improve

collaboration between parties, no need for IT staff and scale the storage needs at will (rent more if more storage is needed or let go of storage capacity to reduce costs) [10]. Corporations are giving up on having servers of their own and are transitioning to on-demand services, so someone somewhere still needs to have the infrastructure for corporations to rent. Big companies like Microsoft, Google or Amazon are providing this type of rental services to other corporations [9]. Common applications are supplied online over the cloud (like text processors and presentation tools, for example) and are accessed through a web service or a web browser. The software and data are stored on “servers” out of the user’s control so the providers are the ones in charge of the security measures (a disadvantage for some since the security guarantees of a cloud service might be insufficient), keeping software updated and hardware maintenance. [10]

Considering all these innovations, and in order to contextualize the work being done in this thesis, Holos S.A. developed an innovative product called ‘Life 4.0 Trains’ which, in its essence, has the goal of technologically evolve the trains circulating in the Single European Railway Area without having to replace the trains entirely, making this evolution cheaper and efficient. A train operating company (which will be referred to as TOC throughout this document) bought this product because they felt that their trains needed to be upgraded to keep up and take advantage of all the technological evolution mentioned before. Due to this deal, sensor information of more than 15 trains will be available and a maintenance problem will need to be solved, which will be explained in detail in the next section.

## **1.2 The Problem**

Maintenance is defined in the Cambridge Dictionary as “the activity of keeping a building, vehicle, road, etc. in good condition by checking it regularly and repairing it when necessary” [11]. On Bevilacqua and Braglia’s paper about maintenance selection [12] they sort the maintenance types into five different groups: corrective, preventive, opportunistic, condition based and predictive.

The TOC’s current ways of performing maintenance on their trains are mainly corrective and preventive. Corrective maintenance happens after something breaks or fails [12], that’s why it’s called corrective because the workers are correcting what has already failed. Corrective maintenance should be avoided, because it can cause downtime if the workers are not available when the failure happens, and it can also affect brand value because consumers may stop trusting the service provided. In the particular case of trains, security is also a big factor and since a failure can put passengers at risk, most of the maintenance done by TOC is preventive, which as the name states, is the maintenance done with the objective of preventing failure [12]. In the realm of preventive maintenance one can do time-based maintenance, which happens according to a schedule, normally recommended by the manufacturer (whose subtype is age-based maintenance, done when an object gets to a certain age) or maintenance according to the distance travelled (since this project is about maintenance in vehicles, this type of maintenance can be applied).

These types of maintenance imply that all workers from all the different areas be ready to check on the train, that materials which may still have some lifetime to spare be replaced, making the company be less efficient and end up losing more money replacing parts that could have not been replaced so soon. Not only needing the workers present more often and the amount of materials spent is a downside in the perspective of the company, but if it is considered the environmental perspective of these types of maintenance, much less materials are being wasted and thrown away by only replacing what actually needs replacement, which is better for the environment, causing less pollution due to less manufacturing and causing less waste. Of the other three maintenance methods mentioned before (opportunistic, condition based and predictive), the proposed solution will focus on condition-based maintenance and its advantages, which will be explained in the next section.

For condition-based maintenance to be done, large volumes of data need to be acquired to keep track of the state of the machinery. This implies that there's also a Big data issue related to the need for condition based maintenance. The Big data Vs that impact this work the most are volume, velocity and validity. Since the sensors installed on a single train can constantly collect information, one of the first problems is understanding how to process the volume of data that is coming from the sensors. The current volume received and software solutions to visualize and analyse available are incompatible, so a solution that can process the data and analyse it according to this project needs must be found, a solution that provides feedback and can act according to a rule-based system. Next, there's a need to understand at which velocity is it worth to collect the information and figure out the most effective velocities for different types of sensors because in some sensors the information can vary quickly but in others that might not be the case, so collecting frequency is key if one doesn't want to have much unnecessary data. Finally, after having all the data at hand, it has to be filtered in order to comprehend what is valid information and what is information that will not be included in the assessment. This can be done according to the software's feedback or by hand (using the knowledge the workers have collected from their experience).

The problem this thesis hopes to solve is the ability to perform condition-based maintenance on this TOC's trains, using a rule-based system that's capable of handling the Big data necessities that come with this project.

### **1.3 Proposed Solution**

As it was said in section 1.2, the solution to be tried to put in place is a different way of doing maintenance referred to as condition-based maintenance (CBM). CBM is a service that 'Life 4.0 Trains' is trying to provide to its customers. CBM is only possible due to all the technological evolution explained in section 1.1. It combines the analysis resulting from measurements obtained from different sensors and sets of rules. The sensors monitor the machine at all times making it easier to detect if a parameter enters an abnormal state (it's out of a defined interval, for example) and if something like that happens, the workers know exactly where to act to solve the abnormal situation, saving time and resources [12]. This solution not only offers the information related to

the components but the information it offers is georeferenced because the system of sensors is collecting GPS data, making it possible to assess if the failure is caused only because of the train's usage or if something on the tracks is contributing to it.

For this continuous analysis to happen, the sensors must be collecting data continuously so finding a software infrastructure that can deal with all this amount of data, analyse, provide feedback on it, deal with rules and possibly automatically infer rules of its own is what will be studied throughout this document.

The main objective of this thesis is to provide a possible solution for a rule-based system capable of handling the Big data characteristics. This system will be orchestrated on a cloud environment since these environments commonly have the capacity of dealing with the necessities of Big data problems.

## 1.4 Outline of the Document

Apart from this introductory chapter, this dissertation has four more chapters which are:

- Chapter 2 | State of the Art: this chapter explains the theory behind this dissertation. From Big data's origins, its evolution, to the cloud and how its applications work, all is explained in this chapter so the reader has a brief understanding of how things work and how the applications, that are the foundations of the system, work.
- Chapter 3 | System Architecture: this chapter explains how the data is being acquired and sorted, how the rule-based system is created, which cloud applications are used and demonstrates other architectures that didn't work or had some issues.
- Chapter 4 | Implementation and Results: here the rule implementation and some results of the tests made on the rule-based system are presented and analysed.
- Chapter 5 | Conclusions and Future Work: a brief reflexion on what should be done in the future to improve the work done so far.





## State of the Art

### 2.1 Big Data Analytics: a need

Big data, raw and untouched, isn't worth much. From raw data to actual knowledge, there's a long process to go through. Data must be stored, processed and transformed before rules can be applied, knowledge can be gathered from the datasets and, eventually, value can be acquired from that knowledge. This process can be divided into two main stages: data management and data analytics. [13] Data management involves the acquisition and storage of data so that, later on, data can be processed and analysed.

Data can be acquired in different ways. In this project, the acquisition of data is done by the sensors on the train. The sensors, spread throughout the four train carriages, collect data related to hundreds of variables. The information is being collected at a 300 ms rate which means that, in just an hour, around 12 thousand data entries will have been gathered for just one variable, one out of hundreds. At the end of the day, tens of millions of data entries will easily be created in a single train. All this data must be stored in order to be processed, which leads to the second main stage of data management.

Databases and data warehouses are two types of data storages. A database has the structured information organized in tables in order to simplify querying. It's usually incorporated in a database management system (the interface between the database and its end user). [13] Databases store current information and are often used for fast querying making them the choice for daily transactions. [14] A data warehouse collects data from various potentially heterogeneous sources, providing correlation between information coming from different sources. In the data warehouse, data isn't usually loaded in real time, so they're not used for immediate queries like databases, instead their primary focus is to provide reports created by applying complex queries to the stored data. [15]

The collected data is being inserted into relational databases. A relational database is a simple and straightforward way of showcasing data in tables in which each row has a key or ID and the columns have the data attributes, i.e., data's characteristics, making it easy to relate data points with each other. [16] In the relational table, each row can be defined by the timestamp, which is different for every collection of data, and the columns have the information related to the different variables for each data entry. There's a relational database per monitor and two monitors per train. These databases are working just as temporary storage because the objective is to load those data entries into a data infrastructure. Choosing which data infrastructure best fits the purpose of this project out of the infrastructures available nowadays is the next step. There's a positive point worth mentioning about the temporary databases which is the fact that they end up doing a redundant job and having that redundant information helps prevent losses of information.

## 2.2 Big Data Architecture

Since the size and complexity of data can be a problem traditional machines face, a Big data architecture is designed to handle that type of data. The velocity in which data is collected varies, with some data arriving fast and in small parts and with other data arriving slowly and in chunks [14]. Big data architectures are considered when one needs to store and process data in volumes and varieties larger than a traditional database can handle, transform data for analysis and analyse data on the spot or with low latency [14].

The majority of Big data solutions include some part or every component of the following components represented in the next figure:

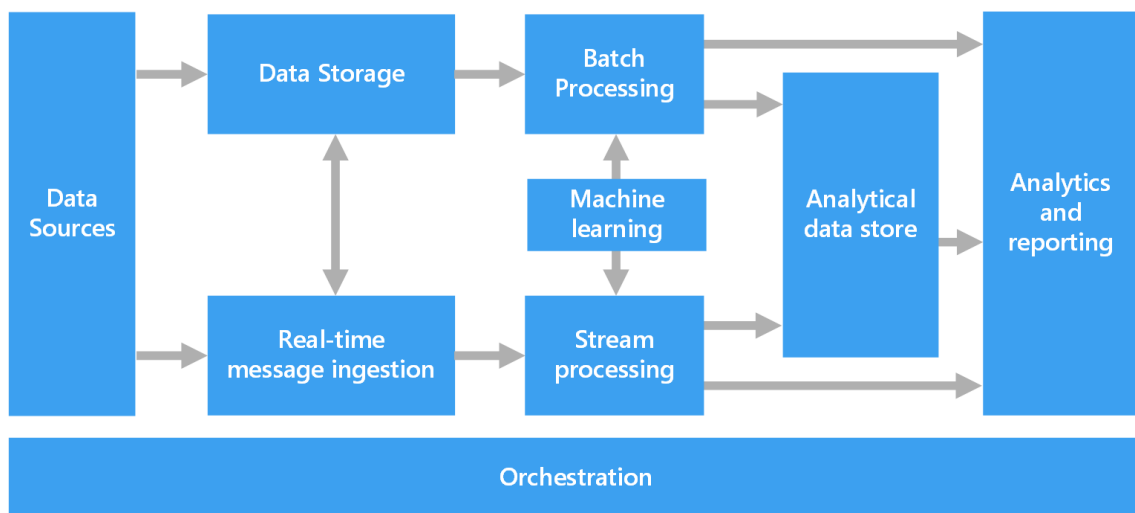


Figure 2 - Components of a Big data architecture. [14]

**Data sources:** data can come from multiple sources and each source can send a different type of data. The sources in the project are the train sensors but in other situations data can come from IoT devices like appliances or even clothes (see chapter 1), statistics from applications such as social media or video streaming platforms or data storages.



**Data storage:** traditional databases can't handle these quantities of data and all the different formats it might embody so, before being processed, it's stored on data storages tailored for Big data.

**Real-time message ingestion:** the solution in place might have to manage, capture and store real-time alerts and messages so it must be equipped with a component capable of handling real-time streams of data.

**Batch processing:** datasets can be huge hence there's often a solution to process data files using batch jobs.

**Stream processing:** this happens when one's processing real-time acquired data.

**Machine learning:** after being processed, data can be fed into machine learning models which use mathematical algorithms to make predictions and decisions based on patterns found within the data.

**Analytical data store:** when the data is prepared for analysis it's often in a structured format that can be queried and is stored in a component that can run these queries using analytical tools.

**Analysis and reporting:** the objective of most Big data solutions is to acquire value from the data collected and that is done by analysing it and generating reports that gather all the findings and insights one can eventually put to use for improvement of one's project or business.

**Orchestration:** the process of automating these solutions.

## 2.3 Big Data Analysis Technologies

For one to make use of all the insight and knowledge Big data might provide, the data itself must be analysed, hence, tools that can deal with major amounts of data had to be developed. One important tool worth mentioning in this thesis is Hadoop.

### 2.3.1 Hadoop

Hadoop is an open-source software developed for reliable and scalable computing. Created by Apache™, it can process large data sets across clusters of computers using simple algorithms [15]. It is a Java based infrastructure capable of detecting and handling failures. In its essence, Hadoop is a combination of a distributed file system (HDFS) and MapReduce, a program developed by Google [16].

HDFS is a file system built with great redundancy, making it tolerant to hardware failure by preventing data loss. Its architecture, as illustrated in figure 2, includes NameNodes and DataNodes. The DataNodes store small file blocks and perform read, write, creation, deletion and replication of the blocks depending on what the NameNode requests. Every time a new DataNode is added to the cluster the HDFS balances its load automatically. The NameNode is the master of the DataNodes. There's only one NameNode per cluster making the system much simpler and it saves all the metadata (names, replicas, permissions, file blocks locations), speeding up the

metadata access process. There are multiple redundant systems in place, allowing the recovery of metadata if the NameNode crashes [17].

MapReduce is a program used to model applications able to process large amounts of data in parallel. Its behaviour is divided in three stages.

The first stage is mapping, i.e., the data elements are organized into <key, value> pairs. The key corresponds to a keyword that identifies the element and the value is the amount of times that specific element with that specific keyword shows up.

After, in the second stage, the program sorts pairs with the same key value into groups.

In the third stage, the values of pairs with the same key (grouped pairs) are summed. The output of this process is pairs of <key, value> that have different keys with the amount of times that key showed up in the dataset specified in the 'value' argument [16].

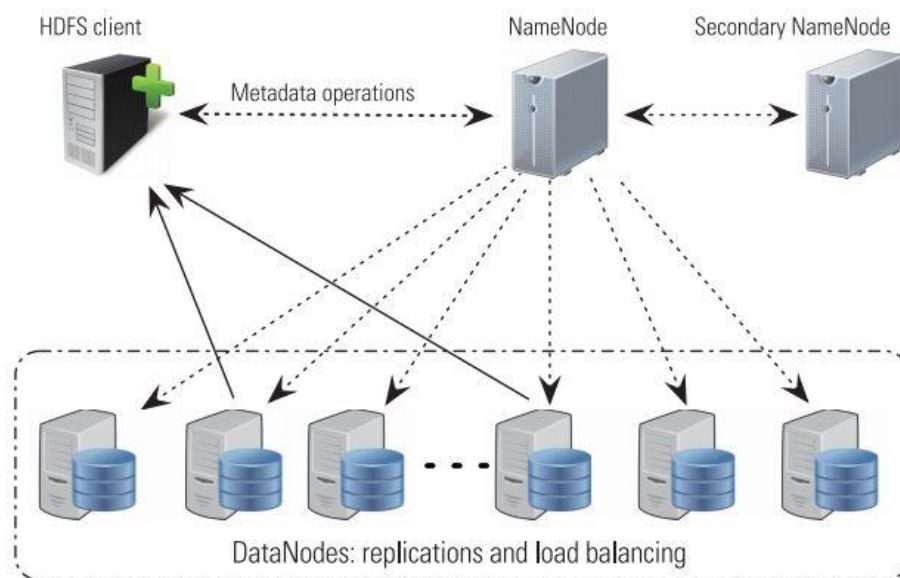


Figure 3 - HDFS architecture. [16]

## 2.4 Big Data Processing

Within the Big data architecture discussed in the previous section, two main processing types were mentioned: batch processing and real-time processing.

### 2.4.1 Batch Processing

Batch processing is done in data at rest. Looking at the Big data architecture presented previously (see figure 1), notice that the data is loaded from the data sources into a data storage and it's on that stored data that the processing will occur [18].

Batch processing is used in many occasions which can be simple data transformations or complete ETL (extract, transform and load – see section 2.5) situations. The data processed through batch processing is normally used for further exploration and analysis [18].

An application of batch processing is, for example, on cases in which there is a large set of semi-structured data and it needs to be transformed into a structured format, ready for queries [18].

## **2.4.2 Stream Processing**

Stream processing deals with data captured in the moment, i.e., in real-time or near-real-time, in order to generate immediate reports or responses such as alerts [19].

In this case, that is the process intended to occur on the trains' system. For example, if the breaks detection system detects an error an alarm must be set on immediately (in real-time).

The process of real-time processing meets the same requirements of batch processing but with shorter times to achieve real-time processing. After processing the data goes, just like batch processed data, into a storage for analysis and visualization [19].

The main challenge faced by real-time processing is the capability of processing large amounts of data in real-time in order to not congest the input stream [19].

## **2.5 ETL: Extract, Transform and Load**

The ETL process is the most common process used in data analytics. When starting a project, the person or group responsible for the data engineering must firstly understand the origin of the data: which are the sources, what type of data files are being received, are they all structured or not, and some other aspects related to the beginning of the collection process. This process is commonly known as the 'Extract' part. After having this understanding, the data engineers can plan how to process the data, what kind of transformations need to be done in order for it to be clean, organized and ready to be exported onto the next part. This chapter of data filtering, cleansing and validating is referred to as the 'Transform' part. Lastly, the data goes to a data warehouse or some other kind of data storage, in what it is commonly known as the data 'Load' [20].

To save time, the three operations can run simultaneously if there's already data received from the extraction process and data already prepared for loading, but an alternative to the ETL process is the ELT (extract, load and transform) whereas instead of waiting for the data transformation process to occur before loading, which can take a long time, the data is stored in its original format, which might be an advantage for some projects or among areas of development that may have different requirements regarding the data transformation [21].

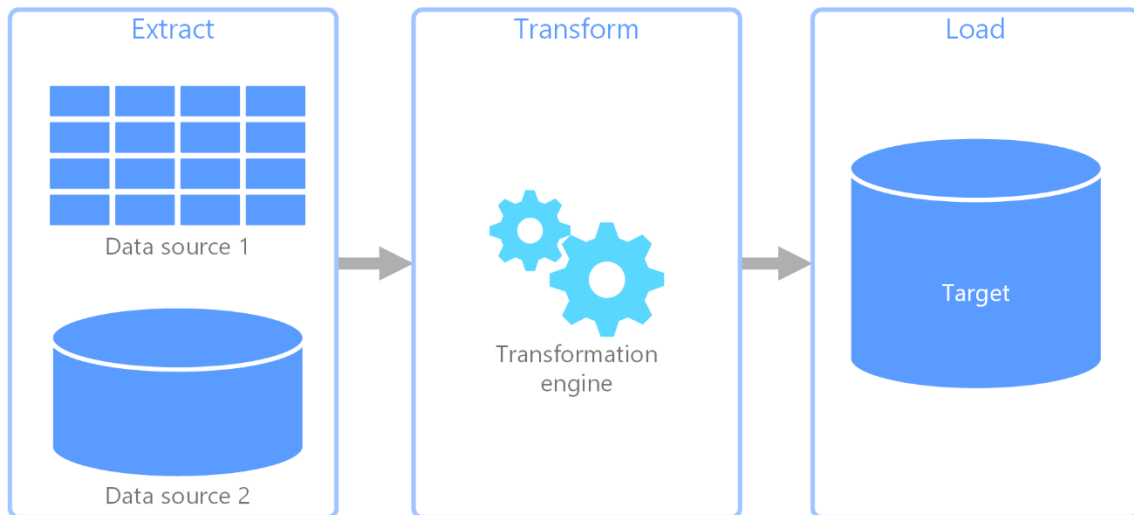


Figure 4 - Extract, transform and load. [20]

## 2.6 Cloud Computing: as a Service

Cloud computing made it possible to share resources over the internet. In its stack, the cloud can be divided into three layers: software, platform and infrastructure. As these are provided over the cloud, they're referred to as Software as a Service, Platform as a Service and Infrastructure as a Service, or SaaS, PaaS and IaaS, respectively [10]. Some even refer to the cloud as a model that supports Everything as a Service (using EaaS or XaaS for short) [22].

SaaS includes the applications and software that provide direct service to the user. The user itself might have to configure a bit of the software but no programming is needed. The providers of SaaS either have their own data center or use another provider's PaaS or IaaS to host their service [10], [22].

PaaS offers components and services (hardware) and a programming environment for developers to create and deploy their applications, making the building and executing of web applications cheaper to its developer [10], [22].

IaaS combines the supply of hardware from PaaS and the associated software solutions from SaaS. Since the cloud service this project will be set on is IaaS, an explanation for it will be presented in the next section.

### 2.6.1 Infrastructure as a Service

Like other cloud services solutions, IaaS has the advantage of allowing its users to use its resources on demand, with pay-per-use and the possibility of short-term commitments with different rental options. IaaS is a way of hosting including computing services, storage and network access. The provider does not do much more than providing the service needed to run and store applications and maintaining the hardware running properly, it's the user that manages the software like if it was working on its own servers instead of the cloud. [10] Scalability is a benefit that users of these services get, like in most cloud services. There are some differences to be considered between the providers which are reliability, performance and pricing. In terms of

pricing IaaS is very flexible, so companies who don't have much money up front still manage to take advantage of these services, paying per use instead of signing long term contracts [10]. Reliability is a key element when choosing an IaaS provider since security threats and downtimes are main factors that affect the consumers reliability itself.

Even though there are different IaaS solutions available, such as Amazon's Web Services, Microsoft Azure and Google Cloud Platform, the choice between one of them eventually comes to company or client preference, the project's current needs and what both parties envision the evolution of the project to be. This project will be developed on Google Cloud Platform since the company responsible already stores their data on BigQuery.

Next, different services provided by Google Cloud Platform that can be adopted in the project will be explored.

## 2.7 Google Cloud Platform Solutions

### 2.7.1 Google BigQuery

Google BigQuery was designed to be scalable and make analysts more productive, working with SQL to quickly analyze up to petabytes<sup>2</sup> of data [23]. Backed up by multiple data centers, Google BigQuery runs queries on tables with about 7 terabytes of size in less than 30 seconds [24] because each data center has hundreds of thousands of cores, dozens of petabytes of storage and terabytes in network capacity [25].

#### 2.7.1.1 Architecture

In its architecture, Google BigQuery works with technologies like Dremel, Jupiter, Colossus and Borg [25].

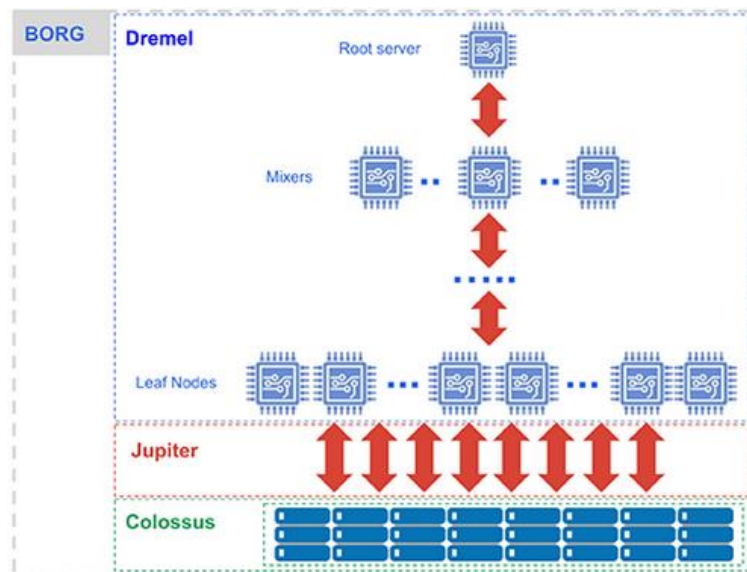


Figure 5 - Relationship between Dremel, Jupiter, Colossus and Borg. [25]

<sup>2</sup> One petabyte is equivalent to  $10^{15}$  bytes.

Dremel is a technology developed by Google to scan Big data at astonishing speeds. It can scan 35 billion rows in tens of seconds [24]. It is this fast because Dremel's architecture has two core technologies: columnar storage and tree architecture. The columnar storage, figure 5, allows for high compression ratio and traffic minimization. The idea of columnar storage is that all the values of a common field are stored adjacently [25].

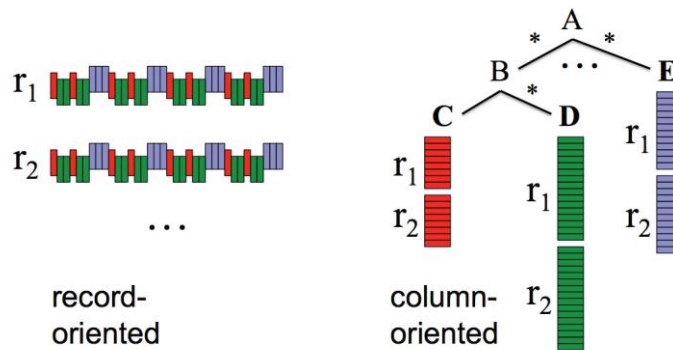


Figure 6 - Dremel's columnar storage. [26]

Tree architecture is a multi-level serving tree, next figure, to execute queries. A query is pushed to the tree and the results are aggregated at incredibly fast speeds. The root server receives the query and routes it to the next level on the tree. The leaves communicate with the storage layer or local storage to access the data [27]. In BigQuery, Dremel accesses Colossus, a distributed storage [25].

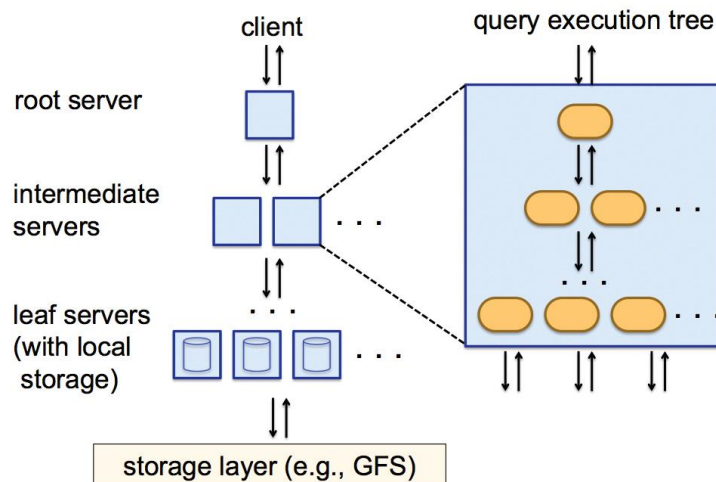


Figure 7 - Dremel's tree architecture. [26]

This service is used by Google on YouTube, Gmail, Search and other services, guaranteeing its continuous evolution and improvement.

Jupiter, the network, quickly distributes large workloads due to its capacity of delivering 1 Petabit/sec of total bisection bandwidth (the bandwidth available between two partitions) [25]. Its bandwidth is big enough to allow one thousand machines to talk to other machines at 1Gbps.

Colossus, the distributed storage on which Dremel runs its queries, is present at each Google datacenter. Each cluster at each datacenter has so many disks that they can provide each user with thousands of disks at a time [25]. If a disk crashes, Colossus is capable of replication, recovery and distributed management (if a disk crashes or other problem happens there's no point of failure). The data in Colossus is also stored in a columnar fashion since it's an optimal way of reading large amounts of structured data [25].

Borg is a cluster management system able to provide the user with thousands of CPU cores. Borg is so immensely huge that even if a query used 3300 CPUs to run that would only be a fraction of the capacity Borg has reserved for BigQuery. It assigns the servers resources needed for a Dremel cluster. Because of Borg, the user never notices failures in the datacenter since Borg routes around the failure [25].

All this infrastructure is part of BigQuery's architecture, paired with APIs, services and high-level technologies. Just like Google made Gmail an email service for everyone, it is democratizing Big data by sharing these types of powerful resources with everyone in just one service – BigQuery [24].

### **2.7.1.2 Features**

Besides the serverless feature mentioned before and the reliability of the service when it comes to failures in the system, there are many more features that make BigQuery great. Having an API that streams data into BigQuery at high speeds makes real-time analytics available for enterprises since the latest data is immediately available for analysis. Due to its ANSI SQL compatibility, time is saved with less need for code rewrites. Its automatic backup feature allows the user to recover work up to seven days. Having in mind the objectives planned to achieve, BigQuery's geospatial feature may be very handy since it allows the user to visualize location-based data in new ways and it simplifies geospatial analysis. Since BigQuery uses Google's storages, it provides the users with such scalable environments that they can store and analyse up to exabytes of data with ease. Encrypted data at every moment and identity and access management assures strong security.

## **2.7.2 Google Cloud Logging**

Google Cloud Logging is a tool within the Google Cloud Platform, GCP, particularly on the Google Cloud Operations Suite (formerly known as Stackdriver) that allows the user to manage, store, control and analyse log data from all the other tools being used in a project, all in one place and in real time [28]. Within the Cloud Logging console, you can query logs to have a better understanding of a particular information. A log entry has at least the following fields:

- A timestamp indicating when the event occurred or when it was received by Cloud Logging.
- The resource where the log was created.
- A payload, i.e. a message, in JSON format.
- The name of the log to which it belongs [29].

One of the fields described in a log entry that can be very useful to quickly filter through the log results and find issues in this system is the severity field. It can be set as ‘default’ which means the log has no severity level, but there are other definitions that can be set as the severity of a log, such as ‘debug’, ‘info’, ‘notice’, ‘warning’, ‘error’, ‘critical’, ‘alert’ and ‘emergency’ [30].

Logs can be exported to sinks as they arrive. A sink contains a destination and a filter that specifies which logs to export. The sink’s destinations can be in Cloud Storage, BigQuery or Pub/Sub [29]. Routing logs to these destinations can later enable the creation of alerts and since logs have a defined retention period, i.e., they’re stored in Cloud Logging for a limited period of time (that can be the default period or be customized by the user) before being deleted [29], routing them to sinks allows you to save log information for more than the retention period.

### 2.7.3 Google Cloud Pub/Sub

Google Cloud Pub/Sub is a subscription-based messaging system for event-driven and streaming analytics [31]. Not only can Pub/Sub be used for messaging dissemination, but it can also be used as an event ingestion and delivery tool to create analytics pipelines.

In Pub/Sub there are two main agents: the publisher and the subscriber. The publisher (an application) creates a message and sends it to a topic to be delivered to the topic subscribers. Publisher-subscriber communication can be one-to-many, many-to-one and many-to-many as shown in the following figure [32].

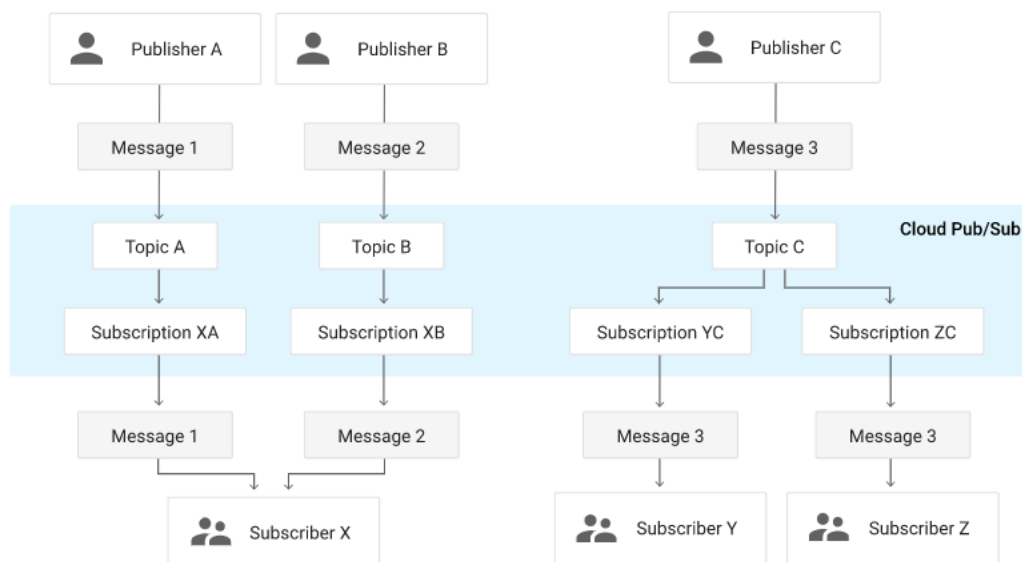


Figure 8 – Pub/Sub publisher-subscriber relationships. [32]

Pub/Sub can be integrated with many Google Cloud components. In this case, it would serve as a Cloud Functions (see next section) trigger and as a sink for logs coming from Cloud Logging.



## 2.7.4 Google Cloud Functions

Google Cloud Functions provides Functions as a Service, or FaaS for short, meaning that it is possible to code and build that code in the cloud, without worrying with the infrastructure necessary to run that code since it all runs on Google Cloud's servers.

Just like BigQuery, Cloud Functions is serverless, automatically scales functions and it's pay per use, being free of charge when functions are idle [33]. Developers also have the possibility to require authentication when accessing functions, making function access restricted, i.e., security is provided through authentication.

One of the features that's the most interesting is the possibility of running Cloud Functions in response to events. Multiple events happening in the cloud can trigger a function. Those events can come from Cloud Storage, Cloud Pub/Sub, Cloud Firestore, Firebase and they can also come from third-party applications through HTTP requests [34]. Indirectly, Cloud Logging can also trigger Cloud Functions by creating a sink into Cloud Pub/Sub.

There are different runtimes available to developers, i.e., Cloud Functions isn't restricted to a single programming language. Developers can code in Node.js, Python, Java, Go, Ruby and .NET [35].

The combination between Cloud Functions and Cloud Pub/Sub is advertised as the use case for IoT scenarios. The suggestion for this use case is having Pub/Sub receive data and triggering Cloud Functions to process, transform and store that data [36].

## 2.7.5 Google Cloud Scheduler

Google Cloud Scheduler allows the user to schedule and automate anything, from cloud operations to failure retries, to reduce manual interaction [37]. The scheduled units of work are known as cron jobs. Each cron job is sent to a target according to a specific schedule. The target can be an HTTP endpoint, a Pub/Sub topic or an App Engine application [38].

When configuring a cron job, the schedule must be defined using unix-cron format. A unix-cron string (visually explained in the next figure for clarity purposes) has five fields ( \* \* \* \* \* ) each representing the minute (from 0 to 59), hour (from 0 to 23), day of the month (from 1 to 31), month (from 1 to 12) and day of the week (from 0 to 6 meaning Sunday to Saturday), respectively [39].

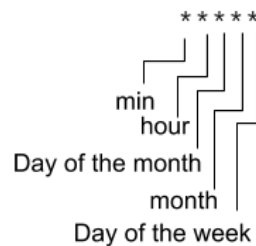


Figure 9 – Unix-cron schedule fields. [39]

For example, a cron job set as \* \* \* \* \* means it happens every minute and a cron-job set 30 10 1 \* \* means it happens on the first day of every month at 10:30. Having to schedule jobs

using unix-cron means the minimum period between a job is one minute which might be a problem to some projects that may need to trigger jobs within a shorter period.

## 2.7.6 Google Cloud Firestore

Google Cloud Firebase is a serverless NoSQL document database. Instead of tables and rows, Firestore organizes data in documents which later are organized into collections. All documents have to belong to a collection so, if a user assigns data to a document that doesn't exist it will be automatically created by Firestore as well as its collection [40].

Documents in Firestore are basically like JSON files with minor differences (limited document size to 1 MB, for example). So, here's an example of what a document in Firestore looks like:

```
Document – person1
name : "John"
surname : "Green"
born : 1977
green-eyes : false
```

Collections are basically folders for your documents. They can't contain other collections neither raw data inside, just documents, and the documents within a collection must have unique names [40]. Although collections can't have other collections inside, documents can have collections associated to them called subcollections. The organization hierarchy goes as follows:

- Collection
  - o Document
    - Subcollection
      - Document
      - o ...

This organizational sequence can go up to 100 levels deep, but a document can never be referenced in a document as well as a collection can't be referenced in a collection, as mentioned before.

There are specific data types that can be stored in Firestore documents. When adding a field to a document, the field types available are string, number, boolean, map, array, null, timestamp, geopoint and reference.

The image shows a 'Add a field' dialog box. At the top, it says 'Add a field'. Below this, there are three input fields: 'Field name' (empty), 'Field type' (set to 'string'), and 'Field value' (empty). At the bottom right, there are two buttons: 'CANCEL' and 'SAVE FIELD'.

Figure 10 – Firestore 'add a field' prompt. [41]

As mentioned in the Google Cloud Functions section, Firestore can relate to Cloud Functions, triggering functions when a document is created, updated, deleted or when any of those actions happen (i.e., a specific trigger that doesn't make a distinction between create, update and delete) [42].

## 2.8 Amazon Web Services Solutions

Amazon Web Services (AWS) offers many IaaS solutions. Since the core of this project is IoT and data analytics, it will be explored which AWS solutions fit the purpose best. There's a need for a solution that can deal with the Big data Vs mentioned in chapter one which were volume, velocity and validity.

### 2.8.1 Amazon Redshift

Redshift, cited online by many to be the AWS BigQuery alternative [43]–[46], is a data warehouse, such as GCP BigQuery, where queries can be run using standard SQL over up to exabytes<sup>3</sup> of data [47]. It is a fully managed data warehouse that scales up automatically to meet the user's needs [48], eliminating the setting up and managing needs that come with having a data warehouse.

#### 2.8.1.1 Architecture

Amazon Redshift infrastructure is running upon sets of nodes, called clusters, which vary in size depending on the amount of data the user has and needs to query. Each cluster has a leader node and compute nodes [49]. Following the figure bellow, it can be visualized how the leader node is the one who makes the connection between the applications and the compute nodes, it decides what steps will be taken to fulfill the operations, i.e., the queries. Since data can be stored either on the leader nodes or the compute nodes, the location where queries are executed depends on where the table is stored [50].

Compute nodes, just like computers, have their own CPU, memory and disk storage. These attributes vary with node type and computation can be increased by adding more nodes or upgrading the node type [50]. Such nodes execute their tasks immediately and send the results back to the leader node to be compiled together. Diving deeper, the compute nodes are divided into slices which process, in parallel, parts of the task ordered by the leader node [50]. The number of slices per node varies but although the number of slices per node might change if the user resizes the cluster, the number of total slices in the cluster stays the same [51].

---

<sup>3</sup> One exabyte is equivalent to  $10^{18}$  bytes.

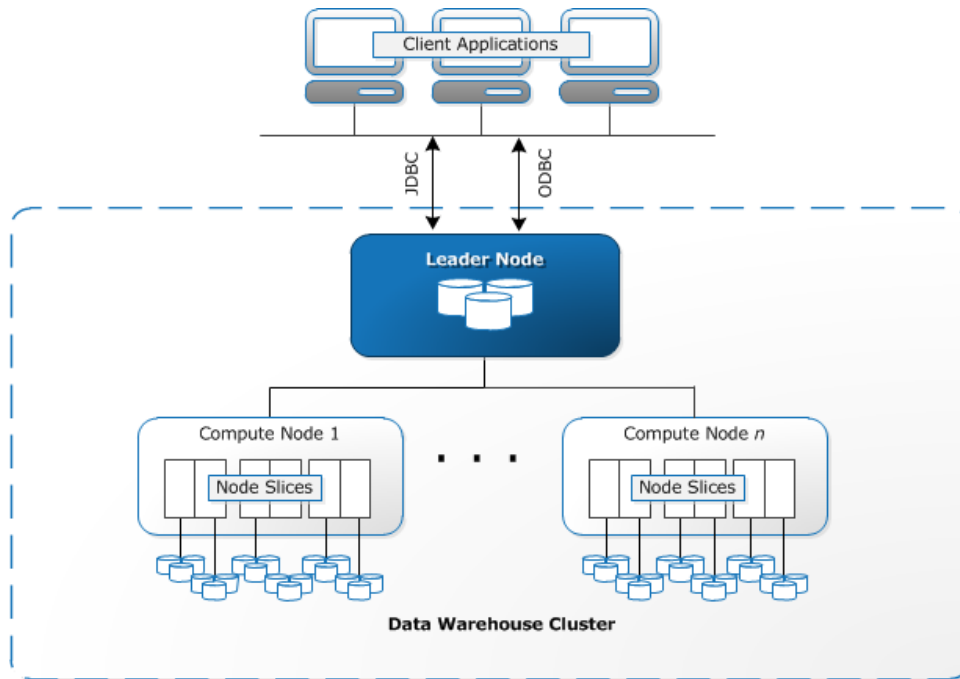


Figure 11 – AWS Redshift simplified architecture. [50]

### 2.8.1.2 Features

Apart from the automatic scaling up or down to guarantee that the users' need are met, Redshift features integration with ETL, data mining, analytics and Business Intelligence tools [50]. Additionally, SQL statements can be used to create and train machine learning models, bringing analytics to the next level.

Speaking of analytics, Redshift can connect with other AWS applications such as Amazon Kinesis Data Firehose (see section 2.8.3) to receive data in real-time and other applications that allow users to monitor, create reports, connect with a data lake and more things related to providing and end-to-end analytics system [52].

An extra feature that improves its speed is having results of queries cached temporarily providing repetitive queries with immediate results.

Furthermore, Redshift also features automated backups and infrastructure management, high security capabilities and integration with third-party certified tools [52].

## 2.8.2 Amazon CloudWatch

Amazon CloudWatch is an application that monitors the other AWS applications, looking at its logs, events and metrics to provide the user with a wide look over its cloud system and how everything is running [53].

Similarly to Cloud Logging, CloudWatch enables the user to set alarms and act in response to cloud events. Large volumes of data are being generated by all the microservices running and that data can be visualized, analysed and used to resolve issues, trigger events, schedule operations and more [53].

CloudWatch features the possibility to perform mathematical analysis on metrics and retain data for up to fifteen months. It also includes a correlation feature between logs and metrics helping developers find issues and understand their root cause quickly, consequently reducing the Mean Time to Resolution[54]. With custom metric analysis, machine learning algorithms can be applied to metrics in order to find unexpected anomalies in the behaviour of applications [54].

While GCP has a separate application to schedule functions (i.e. Cloud Scheduler), CloudWatch works with AWS Lambda (see section 2.8.4) to trigger its functions within specific time periods [55].

### 2.8.3 Amazon Kinesis

Having a need for storing high volumes of data and analysing it in real-time or near-real-time, a quick search on AWS website leads to Amazon Kinesis, a solution designed for IoT real-time analytics.

Kinesis simplifies the collection, processing and analysing of real-time data without any scale limits. Data processing starts as soon as the data is collected in order to act immediately depending on the information gathered from the data [56].

Kinesis offers four main services: Kinesis Video Streams (to capture, process and store video streams), Kinesis Data Streams (to capture, process and store data streams), Kinesis Data Firehose (to load data streams into an AWS data store) and Kinesis Data Analytics (to analyse data streams with SQL or Java – figure 11) [56].

The latter is the service to be studied because the goal is to create alerts and respond in real-time to the information coming from the trains’ sensors.

Kinesis Data Analytics allows the user to query the incoming data using SQL and Java developers to build streaming applications using open-source Java libraries. Kinesis scales automatically in order to match the volume and throughput of the incoming data [57].

Just like the Google BigQuery solution mentioned before, Kinesis Data Analytics is serverless hence the user doesn’t have to manage any infrastructure.



Figure 12 - AWS Kinesis Data Analytics process. [56]

For things to work as planned, Kinesis Data Analytics, which just analyses streams of data in real-time, as to be paired with other AWS services that transform the data, in real-time as well, and tools that can create alerts and act on the data’s information.

AWS Kinesis Data Streams collects data from multiple sources like IoT sensors and makes it available in milliseconds to assure real-time analytics [58]. As part of a possible solution, Kinesis Data Streams would ingest the data generated by the trains' sensors and store it for processing. The data would be streamed by Data Stream into Kinesis Data Analytics to be processed and analysed in real-time.



Figure 13 - Scenario using Kinesis Data Streams and Kinesis Data Analytics. *Adapted from [56]*

## 2.8.4 AWS Lambda

Since the objective is to create a rule-based system, there's a need for something capable of activating alarms when a threshold is surpassed or a value deviates from a certain interval, for example. Amazon's suggestion for this type of scenario is using AWS Lambda, an Amazon service that lets the user run its own code and enables the setup of triggers from other AWS services [59].

Basically, a possible solution for this project, working just with Amazon services, would consist of Kinesis Data Streams receiving data from the sensors and loading it into Kinesis Data Analytics for processing and querying and finally AWS Lambda to trigger alarms.

Through coding in AWS Lambda not only alarms can be triggered but also more complex rules as, for example, 'if in the area delimited by the rectangle defined by X and Y the speed exceeds 150 km/h then set alarm Z on'.



Figure 14 - Hypothetical scenario to be applied on the trains' project. *Adapted from: [56], [59]*

## 2.9 Google vs Amazon: who wins?

Between the big sharks of IaaS, the choice is not clear. At first sight, Google's solution and Amazon's solution seem to both do the job needed for this project. They both store enormous quantities of data and they both manage real-time processing.

Either of them has analytics solutions in place even though it seems that with Amazon Lambda might be easier to create alerts and set rules.

There's also another very important factor to keep in mind when choosing a service that hasn't been mentioned before: pricing.

First, when searching for pricing, it was discovered that Amazon Kinesis Data Streams stores data only for 24 hours (or up to 7 days if one chooses to extend the storage period), which is not what is intended in the first place. So, after a revision of Amazon's offers, the conclusion was that the solution made with just Amazon's products has to include Amazon's Redshift plus Amazon's Kinesis Firehose instead of Amazon's Kinesis Data streams hence figure 13 would change into this:

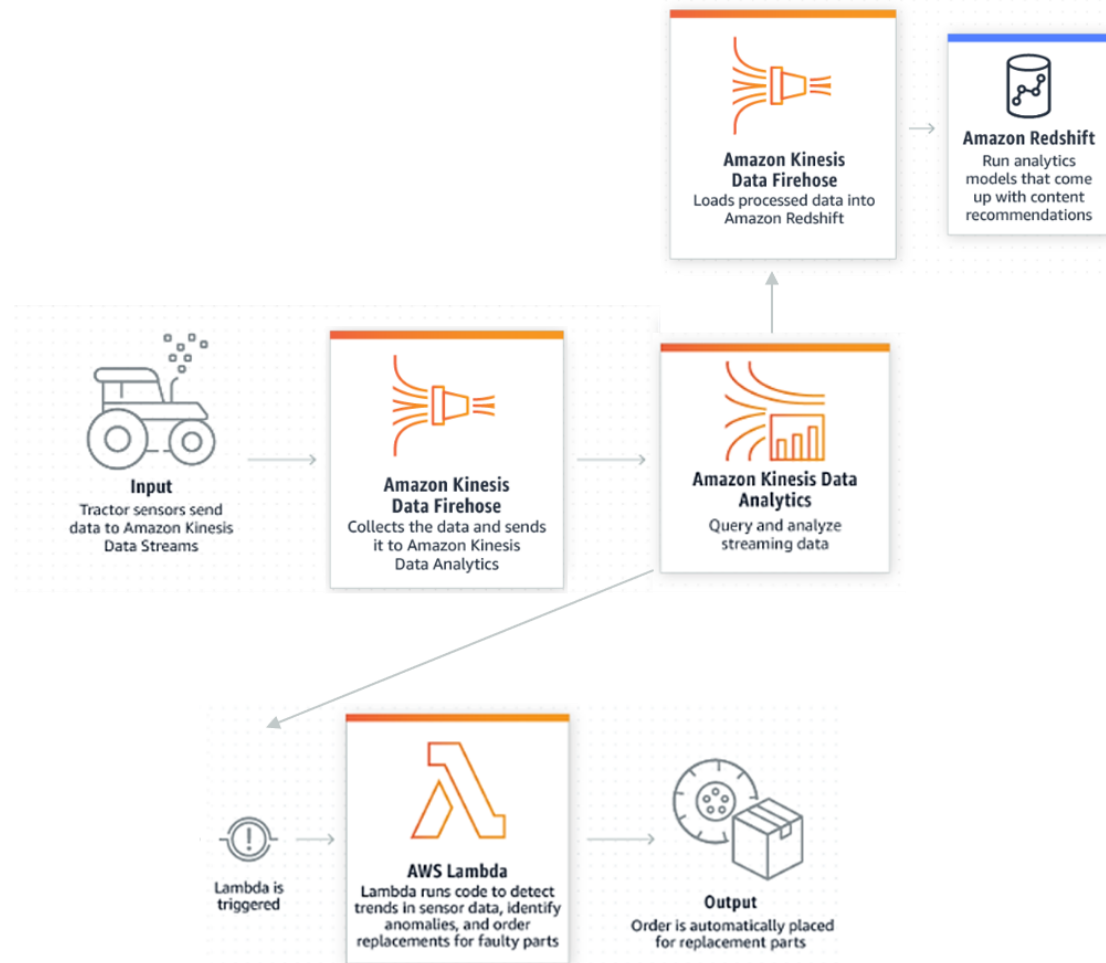


Figure 15 - Revision of the possible Amazon scenario. Adapted from: [56], [59]

As it was mentioned before, latency is to be considered so the prices to be looked at are of the locations where the price to distance between servers ratio is the cheapest. So with that being said, Amazon's prices are of services located in Ireland and Google's prices are of services located in London.

To store, BigQuery costs \$0,023 per GB (offering the first 10GB each month) and Amazon Redshift costs \$0,0256 per GB per month. By the looks of it, BigQuery seems cheaper but that's just for storage purposes, the charges per query also need to be considered.

According to some guides comparing BigQuery with Redshift [43], [46], BigQuery ends up costing more if the number of queries done a month is high because it charges by query while Redshift charges by node use.

Eventually the choice comes down to company preference and since the company who oversees this project has their data stored on GCP's BigQuery, that will be the IaaS used to try to develop a rule-based system with its applications.



## 2.10 Rule-Based Systems

Whenever one needs a program capable of making knowledgeable expert-level decisions and provide solutions to problems the solution is to resort to an expert system [60]. The growth of artificial intelligence has been crucial to the development of these systems. A particular family within the expert systems are the rule-based systems.

Rule-based systems evaluate if different facts match certain conditions, called rules, and do that using the *modus ponens* [61] inference method as its base method. A simple rule looks like this: if A is true and A implies B, then B is true [60].

However, this simplified version of rule-based systems has had different influences and modifications. One of those influences comes from production systems.

Production systems have three major components: a fact table (that acts like a database of facts), a rule base and a rule interpreter [60]. The rule assessment is conditioned by the data instead of following a prespecified order and the rules still follow the *modus ponens* logical reasoning of ‘if *condition* then *action*’ where the action can add or replace facts onto the fact table or even trigger other rules.

This specific type of rule-based system is the one that will be tried to be implemented over the Google Cloud Platform using its multiple products, mentioned in section 2.7.





## System Architecture

Looking back at the purpose of this thesis which is to find a possible solution for the creation of a rule-based system capable of handling data in large volumes, high speeds and great varieties, means that there's a need for great quantities of data available for testing.

Using the data provided by the Life 4.0 Trains project, developed by Holos S.A., it will be possible to test the proposed architecture, integrating it with this product, allowing for a solution for CBM to be applied over trains using the data that the sensors are gathering.

In this chapter it will be presented how the data is being gathered in this project and how that influences the rule system to be created. It will also be explained the treatment and filtration done to the data for it to become usable in the system. Lastly, the architecture implemented over the cloud will be described and explained.

### 3.1 Data Acquisition

Since this thesis is intertwined with the train's project, the data needed to test the architecture of the GCP rule-based system hypothesis will come from the train operating company's trains, around eighteen trains give or take, involved in the project. These trains are made up by four carriages each, in which the edges are motor carriages and the middle ones are trailer carriages.

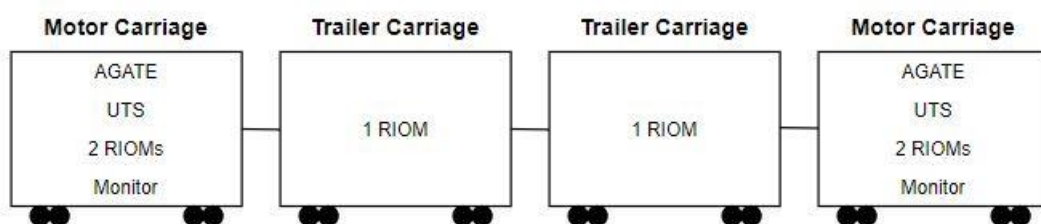


Figure 16 – Train composition and device distribution.

All these trains are equipped with two monitors (see figure 16 below), two UTSs (*Unité de Traitement Sono*, i.e., a centralized unit of information and sonorization), two AGATEs (Advanced Generic Alstom Transport Electronics [62]) and six RIOMs (Register Input Output Module) distributed throughout the train as shown in figure 15.



Figure 17 – Smart train monitor. [63]

The RIOMs receive sensory information from the train, such as door’s movements (opening and closing) or the dead man’s switch (a pedal which the train drivers must press from time to time signalling they are alive, hence the name, making the train come to a full stop if no signal is detected), and send that information to the train’s network.

The AGATEs acquire information coming from the train’s motors. At each end of the train there are two motors and one AGATE handling each pair of motors.

The UTSs not only provide information related to the train’s sound system but they help manage the network as well, working with the train’s network communication protocol – Factory Instrumentation Protocol (FIP) – deciding whose turn it is to send or read information from the network. The FIP exchanges information required for monitoring and maintenance between sensors, actuators and programmable controllers. It also features a distributed real time data base model to be applied in automated systems called the PDC – Producer, Distributor, Consumer – and designed to meet the requirements of data consistency and data exchange periods. [64]

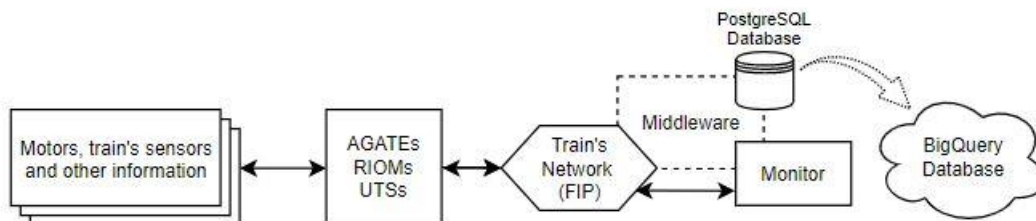


Figure 18 – Data acquisition process from source to cloud.

All this information running in the network, plus information from the monitors, is going to a PostgreSQL database through a middleware and from that database is being uploaded in batches into BigQuery.

This data is not being uploaded in real time yet but the solution to be implemented has that near real time factor in consideration to facilitate possible future implementations of the rule-based system in the project.

## 3.2 Data Filtration

The data available is currently stored in GCP BigQuery tables and, to be able to apply functions to the data and have the system working properly, this data must be filtered, organized and understandable so that the data scientist that comes to analyse can operate it. The transformation of this specific data from binary to structured and categorized data is explained in detail on Pedro Trabuco's thesis on train breakdown's prediction using Knowledge Discovery processes [65] which was a base point for the understanding of the data available in this project.

Although not in binary, the data stored in BigQuery was stored all in one table regardless of which device sent the data. All the common fields were separated into columns and the fields specific to particular devices were gathered in a *value* field. This organization, or lack thereof, didn't allow for rule analysis to be implemented so a data filtering process had to be done first.

The first task was to understand the schema in which the data was stored and then understand what to filter for the rule system.

Table 1: Original BigQuery table schema.

Field name	Description
value	Data specific to the information the device collects
date_sys	Date when the data was inserted in the system
date_train	Date when the data was collected in the train
coord_lat	Geographic latitude coordinates where the train was when the data was collected
coord_lon	Geographic longitude coordinates where the train was when the data was collected
carriage_reader	Indicates from which train the information is
monitor_reader	The number of the monitor related to providing the information
carriage_source	Indicates from which carriage of the train the information is
device_source	The number of the device which provided the information
event_type	The type of event related to the data (ex: error, notification)
mission	A code related to the direction the train is going as well as the start and end stations of the trip
error_type	Specifies the type of error
error_sub_type	Specifies the sub type of error

When looking, not at the schema but, at the data itself a correlation between the fields value, *device\_source* and *event\_type* was noticeable. Specific *event\_type* values were related to specific devices and lots of entries had their value field still in binary. All the entries in binary were discarded because it required extra data treatment and the non-binary entries were enough to test a rule-based system.

Table 2: Device source, event type and value correlation.

<b>Device_source</b>	<b>Event_type</b>	<b>Type of value</b>
0	V (equipment variable)	Binary
1	V (equipment variable)	Binary
3000	V (equipment variable)	Binary
3002	V (equipment variable)	Binary
3003	V (equipment variable)	Binary
3004	V (equipment variable)	Binary
3005	V (equipment variable)	Binary
3006	V (equipment variable)	Binary
3007	V (equipment variable)	Binary
3021	V (equipment variable)	Binary
3022	V (equipment variable)	Binary
3030	V (equipment variable)	Binary
3031	V (equipment variable)	Binary
3032	V (equipment variable)	Binary
3033	V (equipment variable)	Binary
4021	V (equipment variable)	Binary
4022	V (equipment variable)	Binary
-1	E (error)	Decoded
-2	E (error)	Decoded
-3	N (notification)	Decoded
-40	I (system information)	Decoded
-3021	I (system information)	Binary
-3022	I (system information)	Binary
-3030	I (system information)	Decoded
-3031	I (system information)	Decoded
-3032	I (system information)	Decoded
-3033	I (system information)	Decoded

At this point, the devices that had data that didn't need much treatment are known although the data from some of those devices still needed to be sorted from the *value* field into different columns.

The data from devices -1, -2 and -3, which indicate errors and notifications, have the error/notification described on the *value* field and do not need any other treatment. On the other hand, the *value* field on data entries from device -40 and devices -3030 to -3033 has data

condensed into comma separated values, with devices -3030, -3031, -3032 and -3033 having the same format.

Table 3: Example of the value field on a data entry from device -40 and -3030.

Device_source	Value
-40	ABC;01;1;140 km/h;BLUE;0;0;LONDON;2;1;150;5;1;0
-3030 / -3031 -3032 / -3033	239;0;0;0;0;0;0;0;0;24540

Notice in the table above that, although the data is not in binary anymore, what it represents is still not known. One might guess that some fields correspond to velocity and location but on the information from devices -3030 to -3033 there is no hint. Cue, the *var\_types* table.

The *var\_types* table describes what each value in the *value* field of a device means giving a small description and the offset of that value within the *value* field. With this information, new tables were created from the original BigQuery table for the purpose of testing the rule-system. One table for the information coming from device -40 and another for the information coming from devices -3030 to -3033 since they had the same schema. The schemas of these new tables had in common all the fields represented in Table 1, except for the fields *date\_train* (which was a duplicate of the field *date\_sys*), *event\_type* (which never changed within the same device), *error\_type* and *error\_sub\_type* (for simplification purposes). Apart from these common fields, the schemas were as follows.

Table 4: Unique schematic fields of tables -40 and -3030 to -3033.

Devices	-40	-3030 to -3033
Schema	Company	Moderate_electrical_current
	Direction	Reference_speed
	Carriage_occupation	Axis1_speed
	Max_speed	Axis2_speed
	Color_max_speed	Axis3_speed
	Isolated_motor_block	Axis4_speed
	Isolated_bogies	Effort_commanded
	Next_station	Real_effort
	Next_station_id	Converter_current
	Next_station_distance_km	Cantenary_tension
	Next_station_distance_m	Sliding
	Train_composition	Isolated_BM
	Total_effort_percentage	CI_tension
Train_scheme		

Now that the data was all sorted and it is known what almost every field means (almost and not every field because fields like *isolated\_bogies* or *effort\_commanded* have information very specific to experts in trains and for the purpose of making a basic rule-based system and

testing it out, the concrete meaning of the values on this fields was never understood), it was time to fabricate some fictional rules and figure out how to implement them.

### 3.3 Implemented Architecture

After having a broad understanding of how the data is being stored, what Google provides and what can actually be conceived (concepted architectures that cannot be implemented will be further explained in the next sub-section), a proposition for an architecture correlating BigQuery with a rule-based system of some sort created within their cloud system was defined.

Knowing that there's a need to get the data arriving on BigQuery and run queries over it to see if it complies with rules almost immediately, the first minimum requisite of this architecture was to receive an alert whenever new rows of data had been uploaded onto the BigQuery table to trigger the operations needed to check if the new data matches any rules. Since BigQuery only "speaks SQL" (meaning that it's needed to run SQL queries to access, filter, gather and correlate data), all the rules of the rule system need to have an SQL background for at least accessing the data and inserting it into variables in which operations can be worked over. To automate this process, rules were regarded as functions one could code. The first approach was to create a function that was alerted every time new data was inserted in BigQuery but since BigQuery cannot trigger Cloud Functions, it had to be figured out another way to find out that data had been inserted into BigQuery and then trigger a function.

With this information it was discovered that logs could be filtered on the Cloud Logging application to only see logs related to BigQuery queries (in particular, successful queries), gathering in one place all the 'successful insert alerts' needed.

Over on the Cloud Logging application, in order to spot this particular event in the middle of all that's happening in the cloud system, the log metadata had to be filtered by specifying:

- what application was the log from (BigQuery);
- that the job, in this case the query, had to be successful (i.e., job completed);
- the query should be an INSERT query;
- in what table did the INSERT took place.

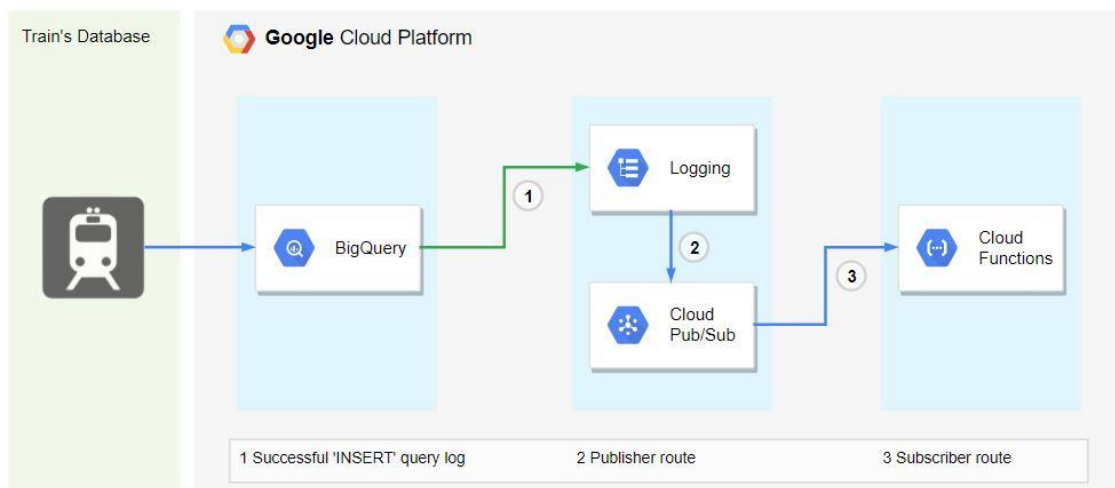


Figure 19 – GCP Architecture: triggering the rule-based system.



As it was said, BigQuery can't trigger Cloud Functions, but Cloud Logging can, indirectly, through Cloud Pub/Sub. Cloud Logging will act as a Cloud Pub/Sub Publisher and its messages are the 'successful insert' logs. The messages are arriving at a Pub/Sub sink and a function within Cloud Functions is defined as a subscriber of that sink so whenever a new insert in BigQuery happens, a message is posted and that subscriber function is triggered.

With this architecture one could say that BigQuery is indirectly triggering Cloud Functions but that would be a bit of a stretch since it is stated that Cloud Logging is already the indirect Functions' trigger.

In the meantime, the specifications of the rule-based system need to be laid out. Facts need to be separated from rules but have they have to correlate with each other and with the new data arriving from the trains. The data arriving from the trains is already stored in a BigQuery table. The facts need to be stored as well (in section 2.10 it was mentioned the need of some type of fact database - a fact table). Following my advisers' suggestion, this fact table will be stored as a BigQuery table, reducing the number of different cloud applications used.

Every time a rule is met, a new fact is generated and stored on the fact table for further analysis and comparison. Each fact table row will include the data that came from the data table, which triggered the rule to be true (generating a fact), as well as a tag and an informational field to help the end user understand what was triggered.

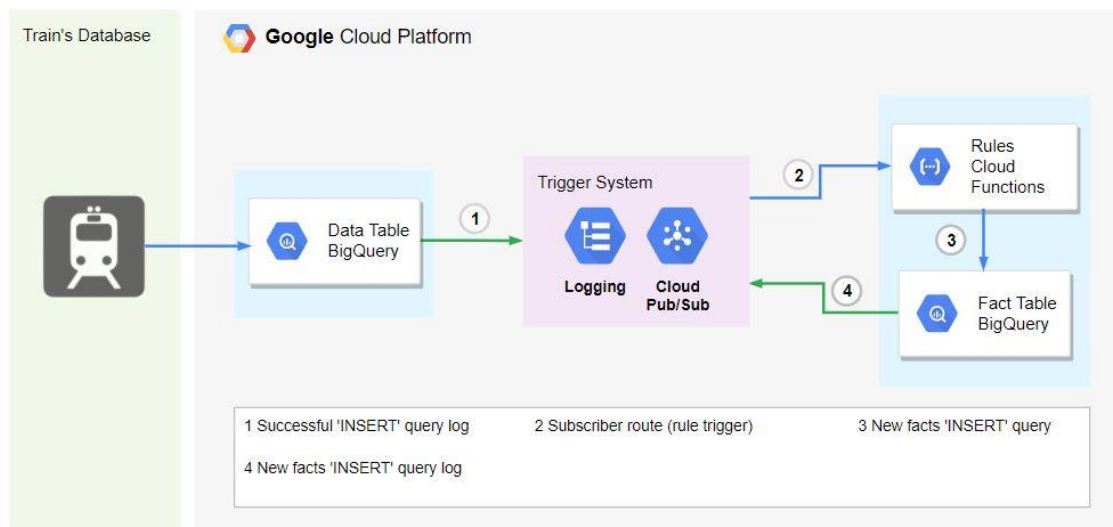


Figure 20 – GCP Architecture: cycling the rule system.

Whenever a new fact is inserted into the fact table, that INSERT query also triggers the rule system again (figure 19 – line number 4), making the system cycle itself until no more rules are triggered and, consequently, no more facts are generated.

Over on Cloud Functions, from all the different languages available to code, Python was chosen to code the rules because Python features multiple libraries that are designed for data scientists and data engineers, simplifying machine learning algorithms, data analysis and data visualization, all of which can be useful in the continuous evolution of the bigger goal of this project – the condition-based maintenance of trains.

### 3.3.1 Other Architectures Considered

Before coming up with this specific architecture for triggering and having a functional rule-based system, some trial-and-error testing happened.

The first architectures considered always involved having Firestore as the primary storage for the fact table. Remembering what was said in chapter 2 about Firestore, it has an hierarchical organization and the idea was to have the fact table not really as a table but as a collection of different documents associated to every rule. For example, a document could be structured like this:

```
Document – rule1
variable A : 90
variable B : 30
outcome : true
timestamp : 2000-01-01
```

If rule1 was met and the outcome changed from false to true, that change could trigger Cloud Functions directly because Firestore is a direct trigger of Cloud Functions probably saving some fractions of second in the meantime, which might not seem much but are relevant when data is being generated with that kind of periodicity. The option of having Firestore as the storage of facts was not studied in depth because all the hypothesis involving it were disregarded, either because of impossibility, impracticability or because they seemed to be less profitable, i.e., needed more resources working at the same time. Nonetheless, here are the hypothesis considered and the problems associated with each.

#### **Hypothesis 1:**

This was the first scenario to be considered, what was thought to be ideal. Keep in mind figure 18 which explains how BigQuery can trigger Cloud Functions whenever new data is inserted into a table and triggering Cloud Functions means triggering the rule system. All hypothesis considered have that trigger schema in consideration since it's the only way found to have BigQuery trigger Cloud Functions.

Looking at Cloud Functions triggers, trying to save time and resources by not having BigQuery as the storage system for the fact table, Firestore seemed like a great option, particularly because it is a direct Cloud Functions trigger.



Figure 21 – Architecture schema for hypothesis 1.

**Problem 1:**

After some more investigation on Cloud Functions properties and the requirements needed to trigger one function using two different cloud applications, it was discovered that Cloud Functions can't handle multiple triggers. Only a single trigger can trigger a Cloud Function so this scenario that sounded good in theory cannot be executed.

**Hypothesis 2:**

In an attempt to understand how people with bigger GCP experience would try to come up with an architecture to trigger Cloud Functions using Firestore, so that the fact table could be stored in Firestore, an online GCP community was contacted to see how they thought this connection could be made. Their idea was to have Firestore (the 'changes in Firestore' alert) post on a Pub/Sub topic, just like how Cloud Logging does with the BigQuery logs. With this, Pub/Sub would be the Cloud Functions active trigger, with Cloud Functions being a subscriber to a Pub/Sub topic where both Firestore and Cloud Logging would post their alerts.

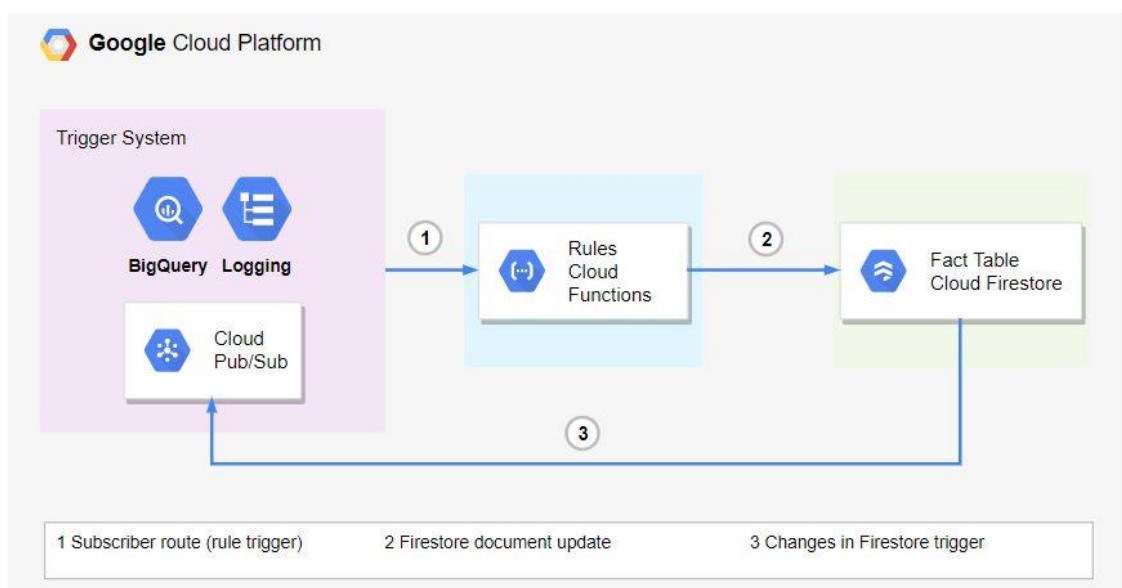


Figure 22 – Architecture schema for hypothesis 2.

### Problem 2:

After extensive research over GCP documents on Pub/Sub and Firestore, no way was found to transform Firestore into a publisher of Pub/Sub so, even though in theory this architecture would work, the features GCP has at the moment makes it impossible to orchestrate this solution.

### Hypothesis 1.1 (upgrade of hypothesis 1):

Unlike hypothesis 1, which is impossible to architect, this option is doable, but it doesn't seem as profitable because it implies having two functions doing the same thing. This wasn't tested, nonetheless it will be presented here.

In hypothesis 1 the issue was that Cloud Functions couldn't have two simultaneous triggers working with one single function. In this solution the idea was to have two functions (duplicates of each other) working simultaneously. One of those functions would receive triggers from Pub/Sub, the other would receive triggers from Firestore and both would write on the same fact table in Firestore.

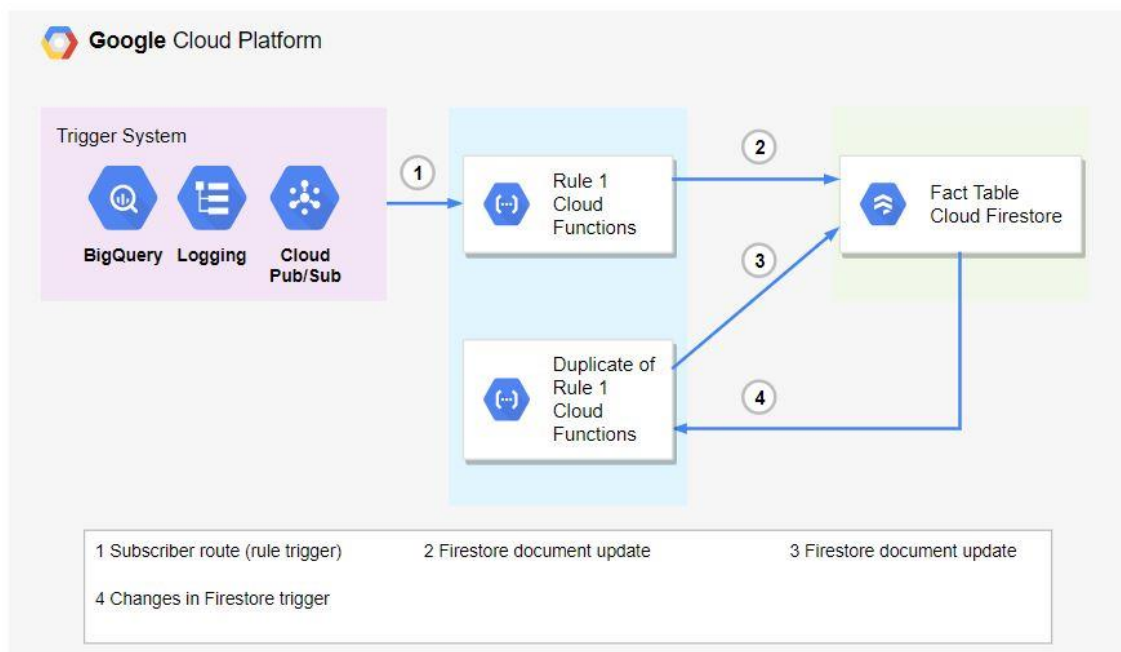


Figure 23 – Architecture schema designed to provide a solution for hypothesis 1.

### Problem 1.1:

As mentioned, having two functions running the exact same code not only can make the system prone to maintenance error but also it is not profitable since more resources are needed to keep the functions active, although this is not the biggest concern with this hypothesis. What is more concerning is having two functions writing on the same Firestore documents and there's a likelihood that this could happen at the same time, overwriting each other and making the rule system lose important information in the process. This wasn't tested either so, as there's a possibility this overwrite might happen there's also a possibility that things may run smoothly. It is not an impossible architecture but a riskier one for sure.

### Hypothesis 2.1 (upgrade of hypothesis 2):

Following the same logic of hypothesis 1.1, this hypothesis was considered to provide a solution to hypothesis 2. Reviewing hypothesis 2, its main focus was to condense all the changes (either new BigQuery inserts or Firestore alterations) in a Pub/Sub topic and having that topic trigger a single function. In terms of resources, this architecture would only need an extra Cloud Function compared to architecture 2 and nothing else. This extra function, which will be called ‘Firestore changes’, would be triggered by Firestore whenever alterations happened and then publish a message on the Pub/Sub topic, triggering the rule system all over again.

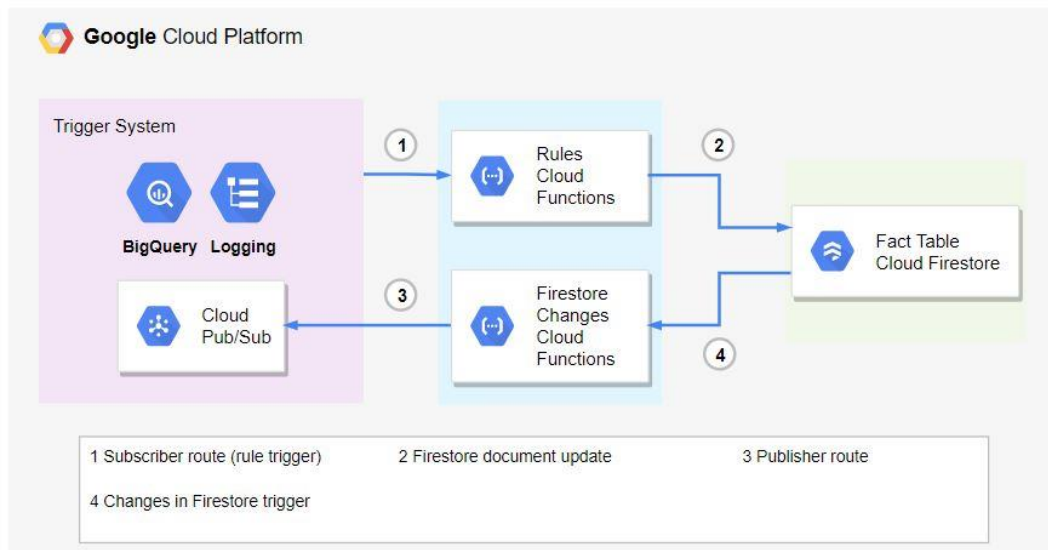


Figure 24 – Architecture schema designed to provide a solution for hypothesis 2.

### Problem 2.1:

The only problem related to this architecture when compared with the architecture implemented was the extra Cloud Function needed to run the system. The issue with the risk of overwriting data which was present in hypothesis 1.1 is no longer present and everything seems to run smoothly. Once again, all these assumptions are just theoretical since this solution wasn't tested.

Concluding, if one would be interested in using Firestore as the fact table storage method, hypothesis 2.1 would be the solution recommended for anyone to test and put to practice.

## 3.4 Rule System

The architecture of the rule system (as can be seen in figure 19) contemplates the creation of a fact table to store the events triggered by the rules. This fact table will store general train information common to all devices and information about what is the fact related to that data entry. Its schema will be as such:

Table 5: Fact table schema.

Field name	Description
date_sys	Values common to the other tables ('device40' and 'device3030to3033')
coord_lat	
coord_lon	
carriage_reader	
monitor_reader	
carriage_source	
device_source	
mission	
fact	
fact_info	

Among the attributes exclusive to the fact table, *fact* is defined as an integer and is the id of the type of fact triggered, *fact\_info* is a string describing what the fact is so the human user can perceive it quicker.

This particular schema was chosen because all the rules to be applied during the testing phase are not known and with this schema the data stored as a fact can come either from device -40 or devices -3030 to -3033. Later, if the user needs more information about the variables that triggered a particular fact, a simple intersection query between the fact table and devices tables can be executed to gather more information about the event.

Evaluating the attributes of the data at hand, hypothetical rules had to be created to test the rule system. Not having worked with trains and not knowing the specifics of what the train engineers want to create alerts for, these rules are completely hypothetical and its variables might not be the most accurate but they do what is needed of them to, which is to test the architecture for the rule system.

The following are some rule ideas considered, keeping in mind the attributes of tables 'device40' and 'device3030to3033' (names given to the tables containing data from device -40 and devices -3030 to -3033, respectively):

**Rule 1:**

*If the current distance to the next station is less than 1 km and the speed is over 80 km/h, then create a 'Break Alert'.*

Looking at the attributes in each table this rule translates to:

IF *next\_station\_distance\_km* = 0 and *reference\_speed* >= 80

THEN *fact* = 1 and *fact\_info* = 'Break Alert'

**Rule 2:**

*If the train is stopped and the 'Break Alert' event is activated, then turn off the 'Break Alert'.*

Again, looking at the attributes in each table, this rule translates to:

IF *reference\_speed* = 0 and *fact* = 1

THEN *fact* = 0 and *fact\_info* = 'NA'

**Rule 3:**

*If the train is in the maintenance area, then activate 'Maintenance' event.*

Which translates to:

IF *coord\_lat* and *coord\_lon* are within a polygon with coordinates A, B, C and D as limits

THEN *fact* = 2 and *fact\_info* = 'Maintenance'

**Rule 4:**

*If the train is outside the maintenance area and the 'Maintenance' event is activated, then deactivate 'Maintenance' event.*

Translating to:

IF *coord\_lat* and *coord\_lon* are NOT within a polygon with coordinates A, B, C and D as limits

THEN *fact* = 0 and *fact\_info* = 'NA'

Other rules more directly related to the subject of CBM could be affected if the sensors provide information for that (for example, sensors measuring thickness of the wheels), or if different calculations are made, like how many km has the train travelled since the last maintenance.

To be able to create these rules, the correlation between trains on table 'device40' and table 'device3030to3033' had to be figured out, i.e., which variables could tell that the data in analysis was from the exact same train at the exact same time. After the data analysis and filtration process briefly explained in chapter 3.2, the two variables used to correlate trains between the different tables were *date\_sys* and *carriage\_reader*. The combination of these two variables gives information about the exact train from where the gathered data is coming and the exact time it was processed, correlating different devices within the same train and allowing for information to be crossed like speed and location, which are measured by different devices (devices -3030 to -3033 and device -40, respectively, for this specific example).

Over on the next chapter, the explanation of how these rules were applied as well as tests and some remarks about how everything worked out will be discussed. These explanations will be accompanied by flowcharts to help the reader have a better understanding of the logical process as well as all to keep up with the steps that occur in the cloud, from data insertion to fact storage.





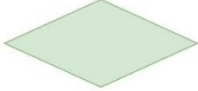



## Implementation and Results

Following good development practices, which say that – if you want to build a car that works, you start by building a bicycle and test it first, add two more wheels and test again – the first step will be to create a single rule, test it and then go from it.

The flowcharts that will accompany the implementation process through this chapter have a colour code that correlates parts of the rule with parts of the architecture explained in chapter 3.3. Every verification block, green and diamond shaped, also corresponds to an active Cloud Function. The blue and curvy rectangles mean data insertions, either on tables ‘device40’ and ‘device3030to3033’ or the fact table with more specific information detailed on the flowcharts.

Table 6: Flowcharts’ blocks guide.

Shape	Description
	Verification block. Represents a single Cloud Function.
	Data storage block. Represents a data INSERT.

These verifications (the Cloud Functions) are running on Python 3.9 and make use of BigQuery Python libraries (google-cloud-bigquery 2.16.0 and google-cloud-bigquery-storage 2.4.0) to communicate with BigQuery. Even with these libraries, one always needs to transmit information to BigQuery via SQL queries.

## 4.1 Implementation of Rule 1

Starting with the creation of rule 1 – “If the current distance to the next station is less than 1 km and the speed is over 80 km/h, then create a ‘Break Alert’” – the flowchart in figure 24 helps to understand the process since new data enters the system until that data is analysed to check if it matches the rule. In this flowchart, the first verification is if the distance to the next station is less than 1 km. For this verification, the data stored in table ‘device40’ which contains the information about location needs to be consulted.

Therefore, to query table ‘device40’ and notice if there were new data entries that placed the trains in less than 1 km of the next station, this is what BigQuery was asked to infer:

Listing 1: Query to check if the train is less than 1 km from the next station.

```
1 SELECT * FROM 'device40'  
2 WHERE next_station_distance_km = 0  
3 AND date_sys >= timestamp_sub(current_timestamp(), INTERVAL 1 MINUTE)  
4 AND date_sys < current_timestamp()
```

In layman’s terms, besides questioning if the distance is less than 1 km, it is needed to only query data that was generated in the last minute, otherwise it would list every single data entry since it has records that matches with the condition. The reason behind the one-minute interval is because since a data flow inserting data in real time or batches onto BigQuery tables doesn’t exist, this insertion must be done by hand, making the minute mark the smallest time interval that can be tested.

Following the flowchart below, if the first verification is true, i.e., the query comes back with results, the second verification will be triggered. This trigger happens via an HTTP request in which it is sent the *date\_sys* and the *carriage\_reader* of each result with the POST method so that the next function can use that data to query a different table, this means that if the query comes back with multiple results, the next verification will run multiple times, as many as the number of results.

The second verification of rule 1 checks if the speed of the train when it reaches the “less than 1 km to the next station” mark is equal or over 80 km/h, creating an alert that the train operator needs to break.

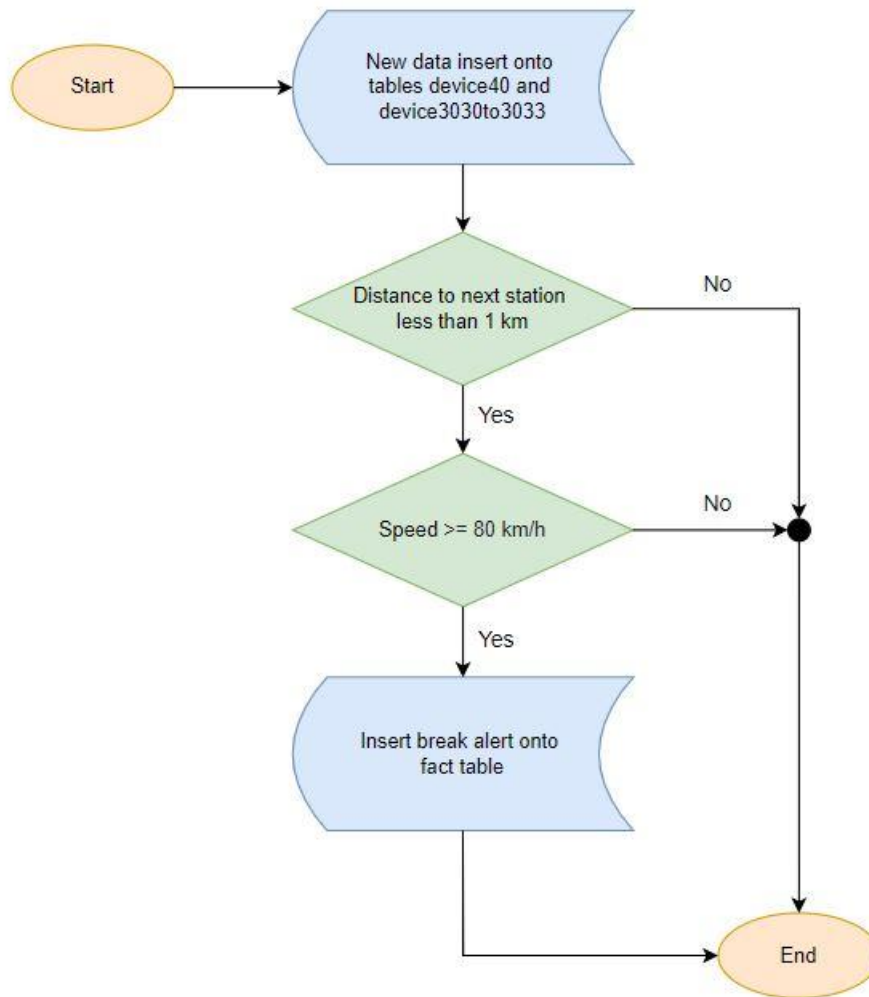


Figure 25 – Rule 1 flowchart.

The next listing illustrates the query sent to BigQuery using data sorted from table ‘device40’ to query the table ‘device3030to3033’, looking up data from the exact same trains.

Listing 2: Query to check if the train’s speed is higher than 80 km/h.

```

1 SELECT * FROM 'device3030to3033'
2 WHERE date_sys = '{}'
3 --the fields '{}' will be matched with the fields coming in the HTTP request
4 AND carriage_reader = '{}'
5 AND reference_speed >= 80
  
```

If this query also produces results, an INSERT query will be executed to store on the fact table the data associated with the train in question and append to that data the ‘Break Alert’ fact. The data regarding at what speed the train was traveling (if 85 or 90 km/h, for example) will not be added to the fact table but one can always figure out at what speed the train was traveling when the alert was generated just by cross referencing the *date\_sys* and *carriage\_reader* variables

between the fact table and table ‘device3030to3033’. This works to figure out speed or any other variable registered at the time of the alert.

### 4.1.1 Trials of Rule 1

To start testing rule 1 it will be inserted, at virtually the same time, data entries in tables ‘device40’ and ‘device3030to3033’ that have the *next\_station\_distance\_km* and the *reference\_speed* variables defined as 0 and 80, respectively. These data entries have their *date\_sys* field rounded up to the minute because different entries could not be matched on the same second by hand. These data entries with those conditions, should generate a new entry in the fact table.

To follow the data and events in the cloud step by step, the different logs from the different applications included in the system, mentioned in chapter 3.3 figure 19, need to be checked.

In chapter 3.3 it is mentioned that, to create a Pub/Sub sink from specific logs, it is needed to filter the logs according to a specific set of conditions, especially because only successful insert logs on a specific table are to be filtered. Over on the GCP it was consulted the Logs Explorer, within the Logging application, and the logs with the exact same filter it is used to route logs to a Pub/Sub topic were filtered. Although some information is omitted from the image, in figure 25 the result of filtering the logs needed to trigger the first verification of rule 1 can be checked.



Figure 26 – Rule 1 insert log.

Since only one data entry was inserted into table ‘device40’, only one result is shown in the figure above, after the application of the filter. This single result is what will go to the Pub/Sub topic to be read by its subscriber, the function ‘tab40-km0-pubsub’.

Once the function ‘tab40-km0-pubsub’ notices a new message on the topic, it starts running. In figure 26 it is highlighted the start and end times of the first verification. This particular one took approximately 17.164 seconds, but execution time is not always the same since, from running multiple tests, it was noticed execution times as low as 7.771 seconds and as high as 17.214 seconds.

Although some information was omitted again, the ‘check 3030 function’ log was left out. This log helps to know straight away that this first verification was met, i.e., the new data entry was less than 1 km from the next station, and that the next verification function was called.

Logs Showing 96 log entries Severity: Default Filter Filter logs

▶	2021-09-05 12:22:09.338 BST	tab40-km0-pubsub	vkcea20by0ds	Function execution started
▶	2021-09-05 12:22:11.281 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:17.592 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:17.592 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:17.592 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:17.592 BST	tab40-km0-pubsub	vkcea20by0ds	check 3030 function
▶	2021-09-05 12:22:17.594 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:26.500 BST	tab40-km0-pubsub	vkcea20by0ds	
▶	2021-09-05 12:22:26.502 BST	tab40-km0-pubsub	vkcea20by0ds	Function execution took 17166 ms, finished with status: 'ok'

Figure 27 – Rule 1 first verification function logs.

As shown in the next logs, figure 27, the second verification, function ‘tab3030-highspeed’, starts running before the first function comes to an end, shortening the total run time since a new data entry is inserted on the tables ‘device40’ and ‘device3030to3033’ and a new fact is inserted into the fact table. By looking at the logs of the second function to see if it was called and observe other tracking information, not represented in figure 27, such as the data transmitted in the POST request, for example. All this to make clear that anything the user needs to print out from the function is show here, helping the user with the debugging process.

Logs Showing 185 log entries Severity: Default Filter Filter logs

▶	2021-09-05 12:22:18.206 BST	tab3030-highspeed	9ea0a9urvjcb	Function execution started
---	-----------------------------	-------------------	--------------	----------------------------

Figure 28 – Rule 1 second verification function logs.

Figure 28 shows that the new fact was created, as expected, and inserted into the fact table (note that the timestamp says 11:22:00 instead of 12:22:00 because logs are in BST and *date\_sys* is in UTC), having rule 1 come to an end and proving the system works at this small scale of analysing data entries as they come to the tables and creating alerts conditioned by the information on those entries.

fact\_table QUERY SHARE CO

SCHEMA	DETAILS	PREVIEW
2535	2021-09-05 11:22:00 UTC	M1301   22   M1301   -3031   13   1   Break Alert

Figure 29 – Rule 1 fact table data inserted.

Even though these entries were added by hand, the filter on the Logging application can be adapted to filter other forms of data insertions once the system is receiving information directly from the trains. The rest of the system works as it worked in this trial.

Other tests were made with different combinations of data inputs, such as combinations that would match only with the first verification, only with the second verification (that would not run even though it had data that matched with it since it didn’t match with the first verification)

or with neither, to consider all scenarios and the flowchart pathways were always met in these tests.

## 4.2 Implementation of Rule 2

Following rule 1, rule 2 will be created so that there is a rule that infers on data generated by this system, and not only on data that came from the train's sensors.

Remembering rule 2, it stated that – *“If the train is stopped and the ‘Break Alert’ event is activated, then turn off the ‘Break Alert’”*. In this case, turning off the ‘Break Alert’ means filling the *fact* and *fact\_info* fields with null values, which are defined as *fact* = 0 and *fact\_info* = ‘NA’.

Looking at the previous flowchart, it is noticeable that the first verification that occurs within this rule is if the train is stopped, in other words, if its speed is 0 km/h. This verification needs to check the table ‘device3030to3033’ whenever there’s new data inputs in the system, asking BigQuery the following query.

Listing 3: Query to check if the train’s speed is 0 km/h.

```
1 SELECT * FROM 'device3030to3033'  
2 WHERE reference_speed = 0  
3 AND date_sys >= timestamp_sub(current_timestamp(), INTERVAL 1 MINUTE)  
4 AND date_sys < current_timestamp()
```

This query is almost identical to the query of the first verification of rule 1. Both queries need to check if the data entry was generated in the last minute, so the verifications don’t overlap older data that was already checked. This prevents duplicate values in case previous data matched this verification, since it would be matching this verification two or more times if the time interval was bigger than 1 minute.

If the verification comes back with results, it will trigger and send data to the next function the same way the first rule did it, through an HTTP POST request. The second verification will query the fact table to see if there is any data indicating that this train had a ‘Break Alert’.

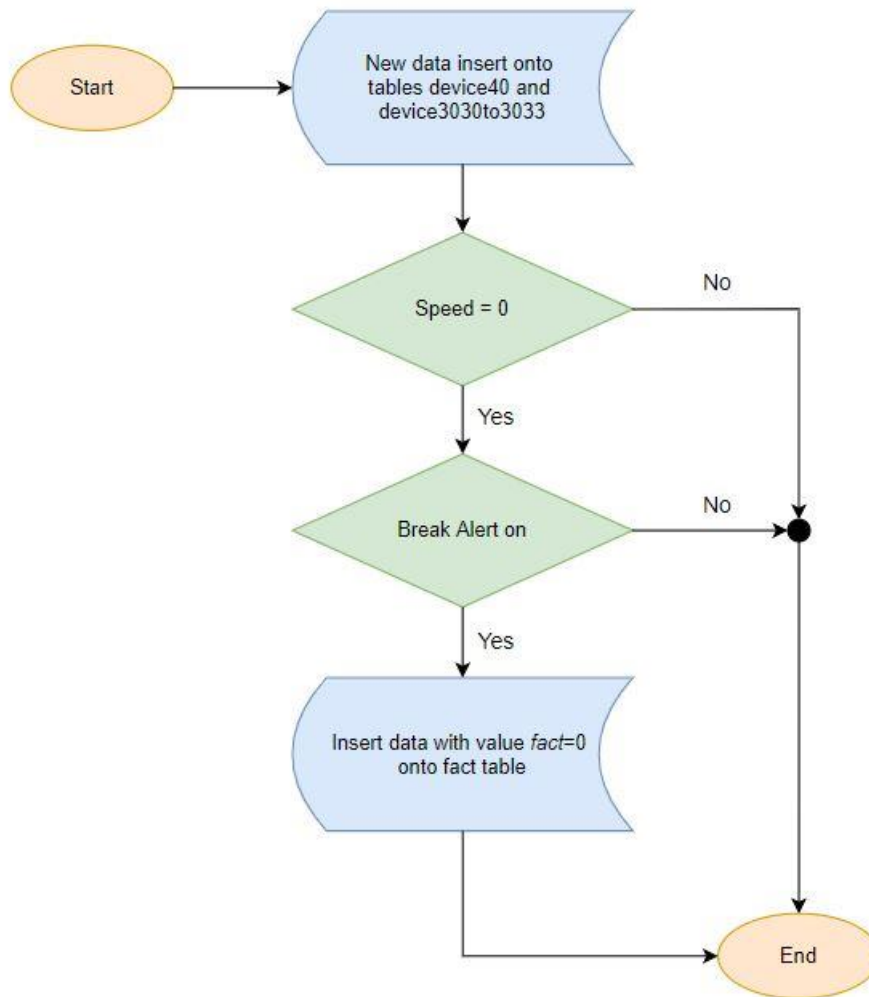


Figure 30 – Rule 2 flowchart.

In this query, the time interval, instead of a one-minute interval like the other queries, was set at four minutes, because a train might not come to a full stop in the minute after rule 1 is triggered. So older data stored in the fact table needs to be checked as well (this variable can be adapted to scan as much time as the train experts feel like the train needs to come to a full stop after a ‘Break Alert’. It doesn’t need to be measured in minutes. The smallest time unit available in these time functions is the microsecond<sup>4</sup>).

Listing 4: Query to check if the train had a ‘Break Alert’ in the last four minutes.

```

1 SELECT * FROM 'fact_table'
2 WHERE date_sys >= timestamp_sub('{}', INTERVAL 4 MINUTE) AND date_sys < '{}'
3 --the fields '{}' will be matched with the fields coming in the HTTP request
4 AND carriage_reader = '{}'
5 AND fact = 1
  
```

<sup>4</sup> One microsecond equals 0.000001 seconds

Once again, this rule comes to an end with a new data INSERT on the fact table, which feels like an update to the data that existed in the fact table but, since the previous data record that stated the train had a ‘Break Alert’ is not altered nor removed, in lieu, a new data entry that states that the train no longer has a ‘Break Alert’ is created, means that it isn’t actually an SQL data UPDATE, instead it is a new INSERT.

### 4.2.1 Trials of Rule 2

To test rule 2, three different data entries were created in three minutes: two with speeds above 80 km/h and with the distance to the next station in km equalling 0 so they would match rule 1 and create ‘Break Alert’ facts, and then one with the speed being 0 km/h to test if not only the rule was working but also if it wouldn’t duplicate data entries on the fact table since, in the last four minutes, two break alerts for the same train would have been generated.

Notice that the filter for the logs is identical to the filter from rule 1, with the exception that this time it is scanning data entries in table ‘device3030to3033’ which are the ones that will trigger the first function through the Pub/Sub topic it subscribed. All entries were created within three minutes, each in a minute.

The screenshot displays a log viewer interface. At the top, there are navigation tabs for 'Query', 'Recent (4)', 'Saved (3)', and 'Suggested (0)'. Below these are filters for 'Resource', 'Log name', and 'Severity'. A code block shows a log entry with the following content:

```

1 resource.type="bigquery_resource"
2 protoPayload.methodName="jobservice.jobcompleted"
3 protoPayload.serviceData.jobCompletedEvent.job.jobConfiguration.query.statementType="INSERT"
4 protoPayload.serviceData.jobCompletedEvent.job.jobStatistics.referencedTables.tableId="device3030to3033"

```

The 'Query results' section shows a table with columns for SEVERITY, TIMESTAMP, and SUMMARY. It displays three log entries:

SEVERITY	TIMESTAMP	SUMMARY
Info	2021-09-05 21:38:06.434 BST	bigquery.googleapis.com jobservice.jobcompleted /jobs/bquxjob_11ee9bc4_17bb
Info	2021-09-05 21:39:04.433 BST	bigquery.googleapis.com jobservice.jobcompleted /jobs/bquxjob_4bee5831_17bb
Info	2021-09-05 21:40:05.355 BST	bigquery.googleapis.com jobservice.jobcompleted /jobs/bquxjob_1c9b4f89_17bb

Figure 31 – Rule 2 insert logs.

With everything working according to plan, the function ‘tab3030-stopped’ verified that the last data entry (created at 21:40 BST), met the requirements of having the *reference\_speed* equalling zero. Since the train was indeed stopped, the second verification was triggered. Again, it happened through an HTTP request where the *date\_sys* and *carriage\_reader* variables were sent by the POST method.



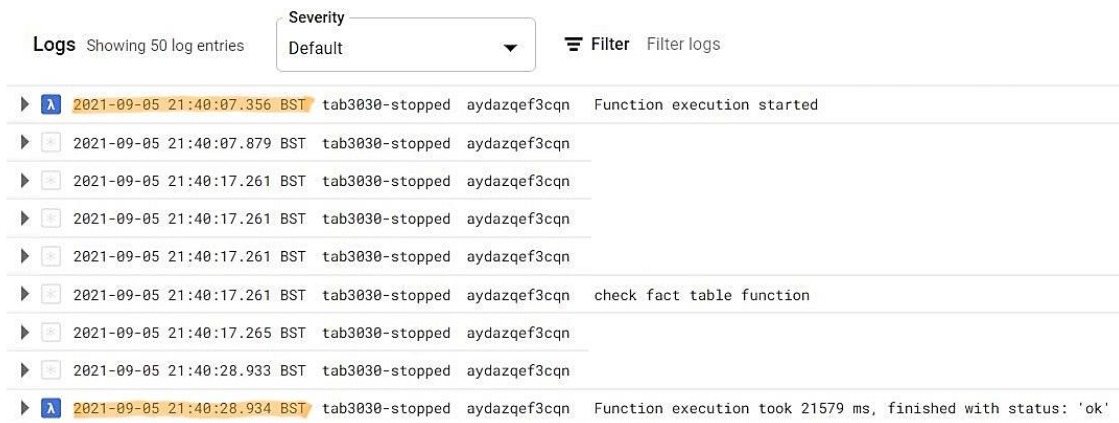


Figure 32 – Rule 2 first verification function logs.

In this particular case, the function’s total execution time was 21.578 seconds. Again, the execution time is not always the same, because cloud resources allocated to run a function are not always the same (more on this subject can be read on this Figiela et al. article about cloud functions performance in different cloud providers’ systems [66]).

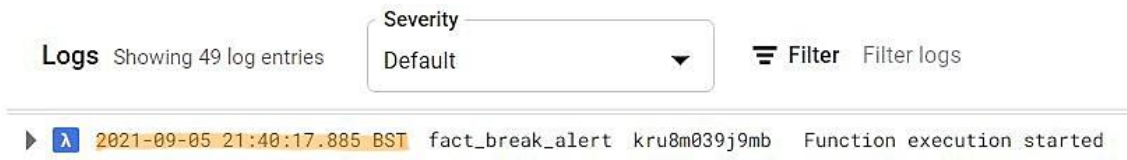


Figure 33 – Rule 2 second verification function logs.

In figure 33, although there were two data entries referencing a ‘Break Alert’ in a four-minute time period, only one data entry disregarding the ‘Break Alert’ was generated, meaning that the portion of code added on the ‘fact\_break\_alert’ function to prevent the creation of duplicate entries was working as well.

fact\_table

QUERY SHARE COPY

	SCHEMA	DETAILS	PREVIEW						
2645	2021-09-05 20:38:00 UTC			M1301	22	M1301	-3031	13	1 Break Alert
2646	2021-09-05 20:39:00 UTC			M1301	22	M1301	-3031	13	1 Break Alert
2647	2021-09-05 20:40:00 UTC			M1301	22	M1301	-3031	13	0 NA

Figure 34 – Rule 2 fact table data inserted.

With these results, it is verified that rule 2 also follows the flowchart designed and, in the end, generates the programmed fact, which in this case was the nonexistence of a particular fact, therefore the *fact\_info* attribute being ‘NA’.

### 4.3 Results and Impracticalities

In chapter two (section 2.1) it was stated that, “the information is being collected at a 300 ms rate”. Because of that, from all the rules presented earlier, only the rules where time was a determining factor were tested. Although it might not happen with every sensor because some may have a smaller reading rate as they might be measuring something that doesn’t need to be measured every 300 milliseconds, this particular system needs to be capable of handling that data as well and here is where this system, the way it is now, is not great for this project.

According to Cloud Functions metrics, the fastest execution time was averaged around 7.99 seconds (in function ‘tab3030-highspeed’, the second verification of the first rule) and the slowest execution time averaged around 23.36 seconds (in function ‘tab3030-stopped’, the first verification of the second rule).

In the trial for the first rule presented in section 4.1.1, the data inserted that will generate a new fact is timestamped at 12:22:06.371 BST (see figure 25). By filtering the logs generated whenever a new fact is inserted in the fact table, the corresponding fact was inserted at 12:22:27.895 BST, therefore the system needed approximately 21.524 seconds to generate the fact. Applying the same logic to the trial for the second rule presented in section 4.2.1, the system needed approximately 24.969 seconds.

From the point of view of an alarm system, having an handicap of only being able of evaluating data every 30 seconds (almost) makes the system flawed. In 30 seconds, major problems can happen with the train and preventing accidents, one of the requirements of this project, might not be doable.

Since the aim of this thesis, besides this particular project, is providing a rule-based system for the application of CBM, this architecture works fine. Rules 3 and 4, presented earlier, are examples of rules tailored for CBM of trains. These rules don’t have the urgency factor like rules 1 and 2, which are alarms. So, for the purpose of CBM in which most alerts are not as time sensitive as the alerts of an alarm system, the architecture implemented can be used. Also, since the data is kept on the fact table for long time storage, this can also be used for performing audits on the train system later in time.

In the next chapter, some future work suggestions involving pipeline analysis of data, in real time, will be presented as well as a project development plan that needs to be implemented before any of this rule-system evaluations can happen.

The rule-system works. The cloud environment in which it was designed has the capacity of handling great amounts of data, making it a solution for when handling large volumes of data is required.

In this particular use case, it can’t handle the velocity the project requires, but for projects with lower real-time data acquisition rates or even using batch processing instead of real time, where the time factor isn’t a deal breaker, this architecture can be implemented.



## Conclusions and Future Work

As stated by Buchanan in his chapter about the principles of rule-based expert systems, “coding a new system from scratch, however, does not allow concentrating primarily on the *knowledge* required for high performance. Rather, one tends to spend more time on debugging the procedures that access and manipulate the knowledge.” [60] This statement relates to this project because it felt like something was being developed by starting in the middle and not in the beginning. A lot of time was spent treating data that should’ve been treated before considering analysing it with a rule-based system, which was the intention of this thesis. Also, to meet up with the big objective of this project which is the condition-based maintenance of trains combined with the alarm system it incorporates, the real time factor is very important to give alerts in a timely manner.

With the future of this particular project in mind, the suggestion would be to use Google’s applications designed to make a direct connection of IoT sensors and the cloud, in particular Google IoT Core.

IoT Core establishes a connection between the devices and the cloud. On Google Cloud’s website, a use case using IoT to manage a vehicle network is presented and the architecture suggested for that use case can be seen in the next figure.

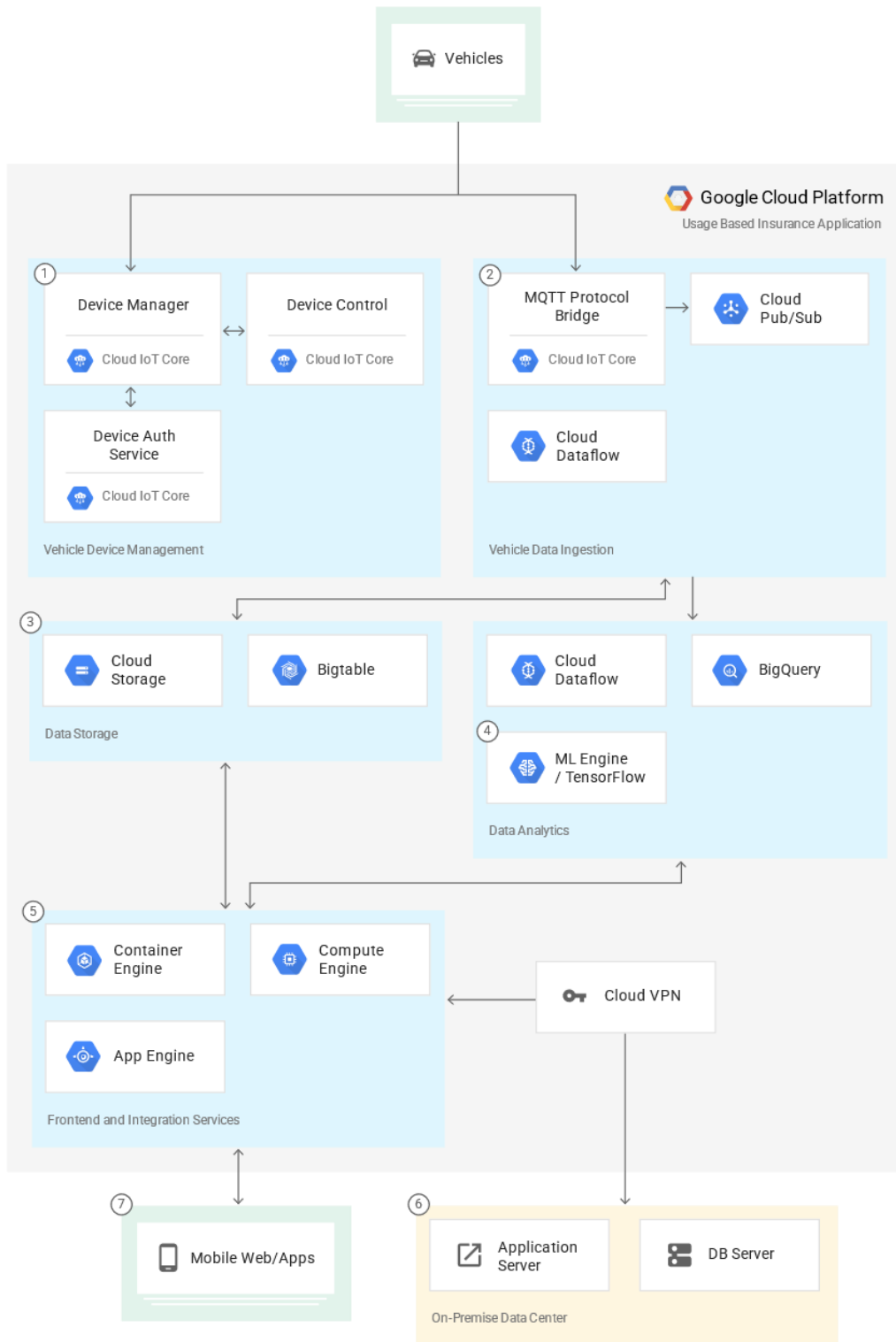
In short, IoT Core makes the connection between the IoT devices and the cloud, granting security. Then, the information – messages – go to Pub/Sub where various cloud applications can retrieve them and work with them. Cloud Dataflow (also pictured in the next figure – figure 34 – on square number 2) can aggregate messages in real time and treat data in streaming, as well as do some calculations while processing data (things like average speed or fuel consumption). After, treated messages are posted in Pub/Sub to be acquired by BigQuery and, only then, go to Cloud Functions and through the rule-based system. In this scenario, time sensitive alerts would go off during the data processing, and the rule-system would work with data that is not as urgent, still generating facts and creating an history of the train.

Regarding the rule-based system, the architecture for the cloud system works when the data acquisition period is around the 20 second mark. It might not be what this project is looking for but might be useful for a project with different conditions. Having in mind the amount of data the system is analysing, this solution solves an issue present in other systems which is the capacity of handling large quantities of data, one of Big data key factors.

For future work, as mentioned in chapter 3, tests using Firestore as the fact table storage can also be run if one is curious about its performance compared to the solution presented.

For the long run, taking advantage of Google Cloud ML services and combining them with this system can bring new knowledge to the table. In this project, it will also improve the maintenance features, enabling predictive maintenance to occur due to the predictive capabilities ML models have.

With all that said, this thesis showed that it is possible to develop a rule-based system like architecture on Google Cloud Platform, although a key Big data aspect, velocity, is not met making the system not ideal for some projects.



- ① **Device Management**

  - Mutual authentication
  - Device authorization
  - 2-way communication
- ② **Vehicle Data Stream**

  - Mutual authentication
  - Device authorization
  - 2-way communication
- ③ **Data Storage**

  - Time-series telemetry data
  - Historical accident data
  - Select Customer data
- ④ **Machine Learning**

  - Driver behavior analytics
  - Risk modeling
- ⑤ **Application**

  - Frontend apps
  - Backend services
  - Integration services
  - Mobile APIs
- ⑥ **Integrations**

  - Integration with on-premise apps and databases
  - Customer, vehicle, billing, services, policy data
- ⑦ **Web/Mobile Apps**

  - Consumer website
  - Consumer apps
  - Field adjustor apps

Figure 35 – Architecture diagram for designing a connected vehicle platform. [67]  
53



## Bibliography

- [1] K. Schwab, ‘The Fourth Industrial Revolution’, p. 172.
- [2] ‘Definition of INTERNET OF THINGS’. <https://www.merriam-webster.com/dictionary/Internet+of+Things> (accessed Jan. 07, 2020).
- [3] ‘Definition of CLOUD COMPUTING’. <https://www.merriam-webster.com/dictionary/cloud+computing> (accessed Jan. 07, 2020).
- [4] L. V. Langenhove and C. Hertleer, ‘Smart clothing: a new life’, p. 11.
- [5] J. Gantz and D. Reinsel, ‘2010 Digital Universe Study’, *IDC Digit. Universe Study*, p. 18, May 2010.
- [6] D. Reinsel, J. Gantz, and J. Rydning, ‘The Digitization of the World from Edge to Core’, p. 28, 2018.
- [7] S. Sagiroglu and D. Sinanc, ‘Big Data - A Review’, San Diego, CA, USA, May 2013, p. 6. doi: 10.1109/CTS.2013.6567202.
- [8] Y. Demchenko, C. de Laat, and P. Membrey, ‘Defining architecture components of the Big Data Ecosystem’, in *2014 International Conference on Collaboration Technologies and Systems (CTS)*, Minneapolis, MN, USA, May 2014, pp. 104–112. doi: 10.1109/CTS.2014.6867550.
- [9] B. Hayes, ‘Cloud computing’, *Commun. ACM*, vol. 51, no. 7, p. 9, Jul. 2008, doi: 10.1145/1364782.1364786.
- [10] S. Bhardwaj, L. Jain, and S. Jain, ‘CLOUD COMPUTING: A STUDY OF INFRASTRUCTURE AS A SERVICE (IAAS)’, p. 4, 2010.
- [11] ‘MAINTENANCE | Significado, definição em Dicionário Inglês’. <https://dictionary.cambridge.org/pt/dicionario/ingles/maintenance> (accessed Jan. 06, 2020).
- [12] M. Bevilacqua and M. Braglia, ‘The analytic hierarchy process applied to maintenance strategy selection’, *Reliab. Eng. Syst. Saf.*, vol. 70, no. 1, pp. 71–83, Oct. 2000, doi: 10.1016/S0951-8320(00)00047-8.
- [13] A. Gandomi and M. Haider, ‘Beyond the hype: Big data concepts, methods, and analytics’, *Int. J. Inf. Manag.*, vol. 35, no. 2, pp. 137–144, Apr. 2015, doi: 10.1016/j.ijinfomgt.2014.10.007.
- [14] ‘Big data architectures’. <https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/> (accessed Feb. 02, 2020).
- [15] ‘Apache Hadoop’. <http://hadoop.apache.org/> (accessed Jan. 15, 2020).
- [16] J. Fan, F. Han, and H. Liu, ‘Challenges of Big Data analysis’, *Natl. Sci. Rev.*, vol. 1, no. 2, pp. 293–314, Jun. 2014, doi: 10.1093/nsr/nwt032.
- [17] ‘HDFS Architecture Guide’. [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html) (accessed Jan. 16, 2020).
- [18] ‘Batch processing’. <https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/batch-processing> (accessed Feb. 03, 2020).

- [19] ‘Real time processing’. <https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/real-time-processing> (accessed Feb. 03, 2020).
- [20] ‘Extract, transform, and load (ETL)’. <https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/etl> (accessed Feb. 01, 2020).
- [21] P. M. Marín-Ortega, V. Dmitriyev, M. Abilov, and J. M. Gómez, ‘ELTA: New Approach in Designing Business Intelligence Solutions in Era of Big Data’, *Procedia Technol.*, vol. 16, pp. 667–674, 2014, doi: 10.1016/j.protcy.2014.10.015.
- [22] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, ‘What’s inside the Cloud? An architectural map of the Cloud landscape’, in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, BC, Canada, 2009, pp. 23–31. doi: 10.1109/CLOUD.2009.5071529.
- [23] ‘BigQuery: Cloud Data Warehouse’, *Google Cloud*. <https://cloud.google.com/bigquery> (accessed Jan. 26, 2020).
- [24] ‘Anatomy of a Query: How Fast is BigQuery?’, *Google Cloud Blog*. <https://cloud.google.com/blog/products/gcp/anatomy-of-a-bigquery-query/> (accessed Jan. 26, 2020).
- [25] ‘BigQuery under the hood’, *Google Cloud Blog*. <https://cloud.google.com/blog/products/gcp/bigquery-under-the-hood/> (accessed Jan. 26, 2020).
- [26] K. Sato, ‘An Inside Look at Google BigQuery’. Google, 2012.
- [27] S. Melnik *et al.*, ‘Dremel: Interactive Analysis of Web-Scale Datasets’, p. 10.
- [28] ‘Cloud Logging’, *Google Cloud*. <https://cloud.google.com/logging> (accessed Feb. 10, 2021).
- [29] ‘Basic concepts | Cloud Logging | Google Cloud’. <https://cloud.google.com/logging/docs/basic-concepts> (accessed Feb. 14, 2021).
- [30] ‘LogEntry | Cloud Logging’, *Google Cloud*. <https://cloud.google.com/logging/docs/reference/v2/rest/v2/LogEntry> (accessed Feb. 14, 2021).
- [31] ‘Cloud Pub/Sub’, *Google Cloud*. <https://cloud.google.com/pubsub> (accessed Feb. 14, 2021).
- [32] ‘What Is Pub/Sub? | Cloud Pub/Sub Documentation | Google Cloud’. <https://cloud.google.com/pubsub/docs/overview> (accessed Feb. 14, 2021).
- [33] ‘Cloud Functions’, *Google Developers*. <https://developers.google.com/learn/topics/functions> (accessed Dec. 23, 2020).
- [34] ‘Events and Triggers | Cloud Functions Documentation’, *Google Cloud*. <https://cloud.google.com/functions/docs/concepts/events-triggers> (accessed Feb. 11, 2021).
- [35] ‘Writing Cloud Functions | Cloud Functions Documentation | Google Cloud’. <https://cloud.google.com/functions/docs/writing> (accessed Feb. 11, 2021).
- [36] ‘Cloud Functions Overview | Cloud Functions Documentation’. <https://cloud.google.com/functions/docs/concepts/overview> (accessed Feb. 14, 2021).
- [37] ‘Cloud Scheduler’, *Google Cloud*. <https://cloud.google.com/scheduler> (accessed Feb. 14, 2021).
- [38] ‘Cloud Scheduler overview | Cloud Scheduler Documentation’. <https://cloud.google.com/scheduler/docs> (accessed Feb. 14, 2021).
- [39] ‘Configuring cron job schedules | Cloud Scheduler Documentation’. <https://cloud.google.com/scheduler/docs/configuring/cron-job-schedules> (accessed Feb. 14, 2021).



- [40] ‘Data model | Firestore | Google Cloud’. <https://cloud.google.com/firestore/docs/data-model> (accessed Feb. 17, 2021).
- [41] ‘Firestore Console – Google Cloud Platform’. <https://console.cloud.google.com/firestore> (accessed Feb. 18, 2021).
- [42] ‘Extend Cloud Firestore with Cloud Functions’, *Firebase*. <https://firebase.google.com/docs/firestore/extend-with-functions> (accessed Feb. 18, 2021).
- [43] ‘Redshift vs. BigQuery: The Full Comparison’. <https://blog.panoply.io/a-full-comparison-of-redshift-and-bigquery> (accessed Feb. 08, 2020).
- [44] ‘Amazon Redshift vs. Google BigQuery: a comparison | Stitch resource’. <https://www.stitchdata.com/resources/redshift-vs-bigquery/> (accessed Feb. 07, 2020).
- [45] ‘What are some alternatives to Google BigQuery?’, *Stackshare*. <https://stackshare.io/google-bigquery/alternatives> (accessed Jun. 29, 2021).
- [46] D. Tobin, ‘Redshift vs. BigQuery - Comprehensive Guide’, *Xplenty*. <https://www.xplenty.com/blog/redshift-vs-bigquery-comprehensive-guide/> (accessed Feb. 08, 2020).
- [47] ‘Amazon Redshift - Cloud Data Warehouse - Amazon Web Services’, *Amazon Web Services, Inc.* <https://aws.amazon.com/redshift/> (accessed Jun. 30, 2021).
- [48] ‘Getting started with Amazon Redshift - Amazon Redshift’. <https://docs.aws.amazon.com/redshift/latest/gsg/getting-started.html> (accessed Jul. 04, 2021).
- [49] ‘Amazon Redshift management overview - Amazon Redshift’. <https://docs.aws.amazon.com/redshift/latest/mgmt/overview.html> (accessed Jul. 05, 2021).
- [50] ‘Data warehouse system architecture - Amazon Redshift’. [https://docs.aws.amazon.com/redshift/latest/dg/c\\_high\\_level\\_system\\_architecture.html](https://docs.aws.amazon.com/redshift/latest/dg/c_high_level_system_architecture.html) (accessed Jul. 04, 2021).
- [51] ‘Amazon Redshift clusters - Amazon Redshift’. <https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-clusters.html#rs-about-clusters-and-nodes> (accessed Jul. 05, 2021).
- [52] ‘Amazon Redshift Features - Cloud Data Warehouse - Amazon Web Services’, *Amazon Web Services, Inc.* <https://aws.amazon.com/redshift/features/> (accessed Jul. 04, 2021).
- [53] ‘Amazon CloudWatch - Application and Infrastructure Monitoring’, *Amazon Web Services, Inc.* <https://aws.amazon.com/cloudwatch/> (accessed Jul. 10, 2021).
- [54] ‘Amazon CloudWatch Product Features - Amazon Web Services (AWS)’, *Amazon Web Services, Inc.* <https://aws.amazon.com/cloudwatch/features/> (accessed Jul. 10, 2021).
- [55] ‘Tutorial: Schedule AWS Lambda Functions Using CloudWatch Events - Amazon CloudWatch Events’. <https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/RunLambdaSchedule.html> (accessed Jul. 04, 2021).
- [56] ‘Amazon Kinesis’, *Amazon Web Services, Inc.* <https://aws.amazon.com/kinesis/> (accessed Feb. 05, 2020).
- [57] ‘Amazon Kinesis Data Analytics - Amazon Web Services (AWS)’, *Amazon Web Services, Inc.* <https://aws.amazon.com/kinesis/data-analytics/> (accessed Feb. 05, 2020).
- [58] ‘Amazon Kinesis Data Streams - AWS’. [https://aws.amazon.com/kinesis/data-streams/?nc1=h\\_ls](https://aws.amazon.com/kinesis/data-streams/?nc1=h_ls) (accessed Feb. 06, 2020).

- [59] ‘AWS Lambda – Serverless Compute - Amazon Web Services’. [https://aws.amazon.com/lambda/?nc1=h\\_ls](https://aws.amazon.com/lambda/?nc1=h_ls) (accessed Feb. 06, 2020).
- [60] B. G. Buchanan and R. O. Duda, ‘Principles of Rule-Based Expert Systems’, in *Advances in Computers*, vol. 22, Elsevier, 1983, pp. 163–216. doi: 10.1016/S0065-2458(08)60129-1.
- [61] ‘Definition of MODUS PONENS’. <https://www.merriam-webster.com/dictionary/modus+ponens> (accessed May 31, 2021).
- [62] C. H. Loch, L. Van der Heyden, L. N. Van Wassenhove, A. Huchzermeier, and C. Escalle, ‘Alstom Transport Equipment Electronic Systems (EES): Supplier Integration Excellence’, in *Industrial Excellence*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 63–83. doi: 10.1007/978-3-540-24758-6\_4.
- [63] ‘Life 4.0 Trains’, *holos*. <https://holos.pt/en/life-40-trains/> (accessed May 24, 2021).
- [64] ALSTOM, ‘FIP Network General Introduction’. Feb. 2000.
- [65] P. Trabuco, ‘Using Knowledge Discovery Processes For Comprehension and Prediction of Breakdowns In Trains’. Sep. 2019.
- [66] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, ‘Performance evaluation of heterogeneous cloud functions: Performance Evaluation of Heterogeneous Cloud Functions’, *Concurr. Comput. Pract. Exp.*, vol. 30, no. 23, p. e4792, Dec. 2018, doi: 10.1002/cpe.4792.
- [67] ‘Designing a Connected Vehicle Platform on Cloud IoT Core’, *Google Cloud*. <https://cloud.google.com/architecture/designing-connected-vehicle-platform> (accessed Aug. 02, 2021).