Data ingestion in Smart Cities
Francisco Gonçalves Palma Barão Santos
1 2 9 0 UNIVERSIDADE D COIMBRA

1 2 9 0

UNIVERSIDADE D
COIMBRA

Francisco Gonçalves Palma Barão Santos

Data ingestion in Smart Cities

**Dissertation in the context of the Master in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Fernando Barros
and Engineer André Duarte and presented to
Faculty of Sciences and Technology / Department of Informatics Engineering.**

June 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Data ingestion in Smart Cities

Francisco Gonçalves Palma Barão Santos

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Fernando Barros and Eng. André Duarte and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2021

1 2 9 0

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

To my Supervisors,
To Ubiwhere and DEI,
To my Mum, Aunt, Brother and Mariana,
To my Girlfriend,
To my Family and Friends,
Thank you

This page is intentionally left blank.

Abstract

The swift expansion of urban areas and the rapid advances of technology, led to the technological entanglement of cities, making them dependent in a variety of sectors on the systems that control them. With the different sectors in a city becoming controlled by machines, the need to monitor them has increased giving rise to an opportunity to have the information centralized so that city leaders can take more coherent and logical decisions. These so-called Smart Cities have been in constant evolution bringing about new Internet of Things devices and new data sources generating more and more information as the years go by. The main issue with this exponential evolution is that legacy ingestion systems cannot cope with the rise in the amount of data that is generated.
This work had in view Ubiwhere´s Urban Platform ingestion issue and aimed to propose a system that allows the gathering of data from the different sources that Ubiwhere has and may have in the future whilst providing near real-time processing and delivery.
The following document proposes an architecture to solve Ubiwhere´s problem in collecting and processing data from different Smart City sources with emphasis on the analysis of the different options to solve the problem of data ingestion. Additionally, this thesis describes the Rule mechanism developed for allowing users to provide their own custom rules so that real-time data comparisons can be made given different data sets for more preemptive and efficient decision making based on the result of the user-defined thresholds. Last but not least, we demonstrate our solution using a real-world Smart Cities use case namely in the sectors of Traffic.

Keywords

Smart City, Internet of Things, Data Ingestion, Data Processing

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

API Application Programming Interface.

CDC Change Data Capture.

CEP Complex Event Processing.

DAG Directed Acyclic Graph.

ELT Extract, Load, Transform.

EPL Event Processing Language.

ETL Extract, Transform, Load.

GDPR DGeneral Data Protection Regulation.

GUI Graphic User Interface.

IMDG In-Memory Data Grid.

IoT Internet of Things.

REST Representational State Transfer.

Tps Transactions per second.

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

The following report documents the work done in 2020/2021 during the internship in order to conclude the Master´s degree in Informatics Engineering specifically in the branch of Software Engineering. The internship was supervised by André Duarte, Head of Tech, at Ubiwhere in conjunction with PhD, Fernando Barros, professor at the University of Coimbra and MSc. The internship took place remotely at Ubiwhere which is a Portuguese research and development software company created in 2007, headquartered in the city of Aveiro with offices in Coimbra, that specializes primarily in Sectors like Smart Cities, Telecommunications and Future Internet.
The internship is focused on the development of a Proof of Concept System to solve a current problem at Ubiwhere. This Proof of Concept consists of a base system that aims to collect information from several data sources, in sectors like traffic and environment, to cleanse that data and to process it in real-time making the data available for use on other Ubiwhere projects, and also consists of an additional Rule Mechanism built externally so as to not only allowing users to make real-time comparisons, given the existing data sets but also allowing them to define certain custom thresholds that if surpassed will trigger an alarm to notify them.
The following chapter introduces the scope of the thesis, the motivation behind it and the context of the problem. Afterwards, the main objectives and a brief description of the structure of the document will be presented.

1.1 Context

"Today 78% of European citizens live in cities, and 85% of the EU's GDP is generated in cities" [1]
With an ever-increasing world population, large expansions of the urban areas have become more common creating numerous bigger cities. In addition to this, technology has exponentially increased in the last years crawling deeper and deeper into our lives leading us to become ever more dependent on every little gadget we own and every piece of technology that comes with it. With this approach of a once thought far fetched future where every step we give we stumble upon man-made machines, the liveability within these fast-growing cities has become dependent on technology to solve issues in numerous sectors such as traffic congestion, pollution, health, infrastructure and waste management [2]. As a consequence, Smart Cities have emerged whilst being in constant development and evolution with sources of information becoming more productive and available bringing about the exponential increase in data generation which requires analysis and special attention

from the city council so they can make better and more coherent decisions.
The main issues behind this major augmentation in data are the non-existence of an integration system that allows different data sources to be collected, cleansed, stored and analysed and the inability of the already existing legacy integration systems to cope with the collecting and processing of all the information from the many sources like the IoT devices in time which ultimately promotes an incomplete analysis of the available data and therefore prompting fraudulent analytic conclusions with misleading reports based on clouded judgement. As we can see from [3], if data is incomplete or is inadequately prepared, the chances of this generating additional problems and sequentially increasing the costs are remarkably high.
To further demonstrate how this is in fact a constant problem in Smart Cities, we can take a look at Santander, where in [4] we can see that the developed system which involves over 20 000 installed devices still does not have an Integration system to monitor the various sectors in the city. Despite mentioning the clear benefits that this system would provide, it is stated throughout the document that this Smart City platform is not only the next step for Santander but also one of their major challenges.
Alternatively If we look at Barcelona´s Smart City planning, they implemented a lot of IoT devices and were inspired to create a cross-platform, later called Sentilo, for sharing information between systems and to easily integrate legacy systems [5]. In the development of Sentilo, the amount of data and its heterogeneous nature was taken into account to be able to expand as the Smart City evolved.

1.2 Motivation and Problem

On top of the previously mentioned examples, Ubiwhere that is working in several different sectors of Smart Cities decided to create a Smart City Platform called Urban platform that aggregates data from different sources and domains and that allows cities responsible personnel to view the systems in a smart city in a holistic manner and also provides them with a way to manage more easily different areas such as environment, energy, traffic, parking, waste and others. This platform is essential for cities since it allows their departments to have access to near real-time information which allows them to take more precise and objective measures. For example, in emergencies, real-time analysis allows the police, firefighters or ambulances to arrive earlier where a few minutes may make the difference in a person's life. This ever-increasing volume of data, it's highly irregular nature and the need to have information available as the sources continually generate it, constitutes a challenge to Ubiwhere.
There are two problems with the Urban Platform at this moment, where the first is the ingestion of the data from the sources to the platform itself. The lack of a proper data processing infrastructure is creating a bottleneck that is limiting the platform by either not allowing the system to collect from all the different sources they could or by not delivering results to the Urban Platform in near real-time results. In addition, this hindrance is mostly a restriction due to the fact that smart cities are in constant evolution with new IoT devices always appearing which in the long run means that the amount of data will just keep rising.
The second problem is related to the need to have real-time cross-analysis between the available data sources due to the non-discardable latency that comes from the transporting, analysing and decision-making stages which are required for the Urban Platform to reach any conclusions. This is an issue since the platform integrates multiple data sources and domains which cross-analysed can create the most diversified scenarios which in some cases may be urgent to reach a certain conclusion.

The main purpose of this internship is, therefore, to provide a Proof of Concept system that will enable the ingestion, processing and integration of third party's data into Ubiwhere's platform in a Smart City environment while focusing on the ability of this system to scale given the increasing difficulties provided from the challenges of Big Data that are in conformity with the ones in our system. Additionally, given that the system heavily relies on the idea of integrating data from multiple sources and domains with the need to allow users to make real-time comparisons between those data sets, this thesis will, therefore, make the most out of the system in order to develop a rule-based mechanism in which users will be able to provide their own rules so that the data can be assessed in real-time allowing for quicker and more efficient preemptive decision making. This mechanism will allow the system to assess and evaluate, user-defined rules with their specified thresholds while not only enabling the cross-analysis of the different data sets but also generating alarms when the previously mentioned thresholds are surpassed.

1.3 Main Contributions

The value proposition of this thesis is based on the objectives and main contributions it brings to not only the market but also the academic community. The main contributions are:

- Propose a Data Ingestion architecture that allows multiple data sources to be connected to the existing platform.
- Scalable Event-based system that allows not only data gathering, cleansing and processing with near-real-time results as output but also cross-analysis through user-defined rules.
- Development of a Rule Mechanism that allows system users to include or remove rules, thresholds and alarms for analysing the information from the already integrated data sets.
- Development of an Alarm API that is used for warning the users that the alarms were triggered in the main system.
- Documentation in the best possible way to make the system as easy as possible to integrate and to make the integration of more data sets possible.

1.4 Document Structure

In the following subsection, the content of each chapter will be explained:

- **Chapter 2 -** The results of the research done are displayed, contextualized and explained. The topics present in this chapter are Smart Cities, the IoT, Big Data, Data Ingestion and Data Processing and lastly there is a subsection about the existing tools for the ingestion, processing and rule mechanism layer.
- **Chapter 3 -** The requirements specification are presented and explained, namely the scope, stakeholders, constraints and the functional and non-functional requirements.
- **Chapter 4 -** The thought process behind the technologies chosen and the proposed architecture is presented after analysing the requirements.

- **Chapter 5 -** An analysis of the choices that had to be taken during the Implementation phase is presented, together with the problems that arose as well as the respective solutions.
- **Chapter 6 -** The tests performed on the system are presented together with the results and analysis.
- **Chapter 7 -** The methodology and planning of the internship is described. The main tasks, the success criteria, the risk management and the tools used are introduced.
- **Chapter 8 -** The process behind the internship and an analysis of what happened during the two semesters is explained.

This page is intentionally left blank.

Chapter 2

State of the Art

The area of study regarding this thesis is not new and the information available is boundless. In the following section, the outcome of the substantial study and analysis on the concepts, technologies and studies of this ecosystem will be presented. This section begins by contextualizing Smart Cities, IoT and Big data down to the challenges they arise. Consecutively, we will review the concepts of Data Ingestion and Data Processing. Last but not least we will go over the existing tools focusing on the more modern ones that were taken into consideration in the scope of this thesis.
It stands to reason that the presented analysis will be high-level whilst trying to convey as much information possible.

2.1 Smart Cities

As can be seen in [1], nowadays most people live in cities which have become large settlements that together with the technological advances, tend to work towards increasing the quality of life of its citizens. As expressed in [2] the opportunity to live in these ever-increasing cities depends on the ability of City authorities to deal with the issues of our everyday lives like traffic congestion, pollution, health, infrastructure, and waste management.
This rapid growth in technology and in the urbanisation of cities has led to the appearance of the concept of Smart City [6]. Despite the fact that the Future we once thought to be a far fetched idea, has been exponentially becoming the reality in which we live in, there is still no clear theoretical definition for what a Smart City really is [7].
This concept has been an issue that has been tackled in many different ways from several perspectives and therefore, many have been the people who have tried to define it giving rise to many variants that are often inconsistent and unclear which according to [8], ultimately leads to the lack of a unified template of a smart city and the lack of a definition that appropriately manages all the variants.
While some believe that to be Smart, a city must be able to connect the Physical, IT, Social and Business infrastructures to grasp all the information of a city [8], others support that a city must be focused on the use of high-tech devices with the latest technologies in order to connect people and information with the final objective being the creation of a more economically focused and innovative city with special attention to the quality of life of its inhabitants and to the pollution that the city produces [8]. We can see from [8] the sheer amount of different definitions that there are.
More modern approaches have classified a Smart City as a System of Systems (City of Sys-

tems) that consists of different sectors containing systems that collaborate with one another promoting communication and interoperability to provide relevantly, well-fundamented information in real-time [9].
This attitude towards the concept in question not only allows people to move away from the misleading idea that in order for a city to be smart it needs to have the most advanced technology but also grants them the opportunity to focus on the ability of systems integrating with each other creating a web that, as stated in [8], will allow machines to be more productive minimizing the need to have people working with them which in end removes the most faulty part of the equation.
Smart cities are considered to be extremely dynamic environments that are in constant development and therefore, it is important to outline the challenges that these kinds of cities bring. Due to the ever-growing cities and the constant evolution of technology [10], Smart Cities have a tendency to become even more intelligent as new advances are made in the field. This means that not only more city sectors will come into contact with technology but also the ones already existing will be upgraded which in the long run this leads to the exponential growth in data. This increase in data not only consists of the sheer volume of information but also the speed in which it is generated and the diversity provided from new sources that will arise [11].

2.1.1 Smart Cities Examples

As previously mentioned, the concept of Smart City is not new and therefore, more and more cities have taken steps in order to become more efficient, more intelligent and above all, more comfortable for those who live in them. When looking for Smart Cities, quite a few names appeared [8], however, there are a few that are arguably often considered pioneers in the area not only because they are in constant expansion with innovative projects but also because they helped to set the ground on good practices and to this day serve the purpose of being testing grounds for new technologies and devices [12].
Cities like Amsterdam [13], which was not only one of the first European cities to start a Smart City program in 2009 but also one of the cities that introduced the benefits of open-sourcing data. Amsterdam started by creating a database that was not only open but also contained 12 000 data sets from all 32 districts primarily in the sectors of healthcare, traffic and education. Nowadays, Amsterdam has created an integration platform for managing the different Smart City projects, currently more than 70 projects, and to share data which contains projects in seven different areas, namely, Infrastructure and Technology, sustainable energy sources, transportation, governance and education, Smart City academy, Citizen participation and Circular City [14].
While Singapore [15] is not one of the first Smart Cities or Smart Nation as they call themselves, having started in 2014, it is a name that stands out when it comes to innovative Cities with its citizens in the core and not technology. Singapore has become one of the most advanced cities when it comes to three sectors. First and foremost, security with the deployment of over 52 000 surveillance cameras that are constantly obtaining information from public spaces. Secondly, there is transportation, where government policies in conjunction with technological alternatives to cars have encouraged the use of public transports ultimately turning the city into a testing ground for newer and more environmentalist approaches. Last but not least, administrative services with an application called Mobile Government, allows citizens to have access to government services and information anywhere, anytime [16].
One other name that constantly emerges when talking about Smart Cities is Barcelona [17]. Having started in 2011, the city heavily invested in infrastructure which includes a vast IoT sensor network in the sectors of transportation, energy and air quality. In order

to improve the use of this network, a cross-platform named Sentilo [18] was developed in order to integrate the applications and to better monitor the data. As we can see from [19], a study with the aim of estimating the amount of data generated in the many sensors throughout Barcelona that are concentrated in Sentilo, the sheer quantity of information that is generated in a smart city is currently immense and with the advances in technology, it will only increase in the near future. If Sentilo is not prepared for this exponential growth, it will soon start decaying and become yet another legacy system that will not be able to cope with information in real-time.
Taking a look at Santander [4], the Smart project was funded in 2010 and was highly based on the idea of machine-to-machine communication focusing on the interactions between sensors and devices. It is often referenced as a living urban laboratory due to the 20 000 devices that have been installed on the most various sectors. As stated in [19], an integration platform to analyse the different data sets as a whole still doesn't exist as it stands as one of their biggest challenges due to the vast amount of data provided by all those devices.

2.2 Internet of Things (IoT)

According to [20] the Internet of Things is the connection between the physical world and the Internet by connecting real-world devices like sensors that provide data and machines that interact in conformity with the results provided from the sensors. As Kevin Ashton, who came up with the term IoT, specifies "In the twentieth century, computers were brains without senses - they only knew what we told them. In the twenty-first century, because of the IoT, computers can sense things for themselves" [21]
As claimed by [22], one way to look at the IoT concept is to divide it into three perspectives which as a whole define it. The three perspectives are:

- **Things-Oriented -** Focuses on the physical components that allow the connection between the physical world and the digital world. Some examples of this are the wireless sensors and actuators, RFID, NFC;
- **Internet-Oriented -** Plays an important role in how the Internet Protocol is used to connect the Things mentioned above. This is imperative so that the physical components can communicate with each other and so that they can be integrated into a platform;
- **Semantic-Oriented -** Is mainly related to the prediction that in the future, the number of devices will grow exponentially. This perspective is related to the issues that come with this growth of devices, namely how we will connect all the devices, how we will store them and how we will search, organize and analyse all this information.

The Internet of Things is considered to be one of the most noticeable technological trends that have emerged lately which can highly impact the whole business spectrum. These impacts are often beneficial as they allow the connection between these devices, their systems and services which go beyond machine-to-machine scenarios [23] allowing the use of these interconnections to analyse data in a way never thought possible which has opened up new doors especially in Smart Cities sectors like traffic congestion, waste management, healthcare, security, emergency services, logistics, retails, industrial control, and health care.
Taking [24] into account, the IoT market in Smart Cities is already one of the core fundamentals on how data is produced and is expected to increase at an alarming rate with

expectations to more than double in 2025 moving from 113.1 billion to 250 billion USD. If we look into the challenges previously mentioned that Smart Cities have due to the amount of information produced, the variety of its nature and the speed with which it is produced, this is prone to become a much more challenging problem as the IoT evolves.

2.3 Big Data

In spite of not existing a clear universal definition, within the literature there are many who have tried to define the concept of Big Data. According to [25], the term was initially introduced to describe the great amounts of data generated from newer technologies like smartphones and IoT Sensors. In [26] Big Data is referred to as huge amounts of unstructured data that is produced by applications considered to be high-performance.
Alternatively, IBM in [27] defines Big Data as information that comes from everywhere like for example, sensors used to gather climate information, social media, digital pictures and videos, transaction records, cell phone GPS and many other examples.
As stated by [28], Big Data is considered to be an indicator that the data has gotten too big, too fast or too hard for the existing tools to process. By "Too Big" the writer conveys that companies must deal with great amounts of data provided from the most diverse of sources like streams, transaction histories or sensors. As to "Too Fast" the writer denotes that not only do companies have to deal with huge amounts of data, they also have to do it quickly. An example of this is analytic platforms in Smart Cities that must have near real-time information in order to make the most precise and correct decisions. Last but not least, the writer with "Too Hard" means that information may not be appropriate for how the existing processing or analysis tool is configured.
Another common definition of Big data, is defining it through the challenges it presents to data. Gartner in [29] defines Big Data as information provided from high-volume, high-velocity and high-variety sources that need innovative ways of processing data in a cost-effective way that allows data to be used for better decision making and to prevent wrong conclusions and misleading reports. Alternatively, Bernard Marr in [30] connects the concept of Big Data to two phenomena, namely the speed with which we generate data and the ability we have to store, process and analyse that information. In sequence, he defines Big Data similarly to how Gartner does but proceeds to add 2 more V's to the definition, namely Veracity and Value.
While previously, Gartner and Marr stated that Big data consists of 3 and 5 V's respectively, over the years others started to consider Big data to have more V's reaching numbers such as 19 V's which can be seen in [31].
Despite this and in order to simplify and better connect Big Data to the problem in this Thesis, we will focus on the main Big Data's challenges that consists of 4 of the V's mentioned in [32] and that are seen in Figure 1, namely:

- **Volume -** Symbolizes the large amounts of data that in Smart Cities are constantly provided from the many sources;
- **Velocity -** Focuses on the speed in which data is generated, updated and made ready for collection or delivered;
- **Variety -** Consists of the data diversity provided from the ever-increasing number of IoT sources;
- **Veracity -** Stands for not only the accuracy and quality of the data but also how trustworthy the information and the source really is.

Figure 1: Big Data 4V's [33]

2.4 Data Ingestion | Data processing

According to [34], Data Processing Cycle is the term given to the sequence of events behind getting data from somewhere, transforming it into information with meaning and making it available for later use. This cycle can be granularly divided into the following stages:

- **Collection -** First step in the cycle that involves getting the data from sources and getting them into our system. In a high-level clarification, this can either be done synchronously by polling requests from a source every previously set measure of time or by having them asynchronously being pushed into our system whenever they are available;
- **Preparation -** Considered to be the most abstract and adaptable stage due to the number of operations that can be done to the data and due to the fact that they can take place on different layers. These operations often consist of formatting the data into the correct format, sorting, filtering, removing unused data and other Data Cleansing [35] Methods;
- **Input -** Delivers the data to the Processing unit and therefore needs to make sure that it is without errors because as said in [34], the quality of the Output data is defined by the quality of the Input. This step is commonly done in conjunction with the Preparation stage;
- **Processing -** Procedure where data is computed in order to turn raw data into useful and structured information. This is the most lengthy and heavy-duty based step that depends on the processing power of the machine, on the data complexity and on the volume of the data. It is worth mentioning that all the previous help fastening this process;
- **Output -** Step that after having the data ready, is in charge of transferring that data into other systems that use this data. Alternatively, instead of transferring the data directly, it is also possible to update the information in a queue or other storage system so that it can be requested at a later date;
- **Storage -** Last and mostly optional stage as it depends on the system resolves around safely storing the metadata previously created for later use.

In a more abstract point of view, Ingestion is considered to be the act of taking something in or absorbing, commonly related to the human body [36]. When related to Data, Ingestion

is often described as the process of collecting information from one or more sources to a destination storage system as efficiently and correctly as possible [37]. When comparing with the data processing cycle, Data Ingestion is the term given that englobes the first 3 steps joining Collecting, Preparing and Inputting data allowing it to become available for further modifications in the Processing stage.
According to the European Commission on Article 4 of the GDPR [38], Processing is a term that covers many data operations that can either be manual or automated. The mentioned Data processing Cycle steps are just some of the operations mentioned in [39]. Additionally, [40] defines Data processing as the collection and manipulation of information to produce new data or alternatively, just any changes to data. However, in the scope of this thesis, we will focus on Data Processing purely as the Processing stage in the data cycle.
The problem of Ingesting and processing data became significantly more relevant when Smart Cities emerged due to the Big Data challenges they present especially with the rise of IoT. The challenges of ingesting and processing data go mostly hand in hand with the challenges of Big Data, namely, the need to have future proof scalability since new IoT sources are in constant appearance and evolution, the need to handle the vast diversity and voluminous nature of data and last but not least, the demanding need of speed which becomes difficult to maintain due to the fact that the process of pipeline developing and keeping up with the more complex data becomes harder and more time consuming as Smart Cities and its sources evolve [41]. One other challenge that has arisen in these concepts is the obligation of pipelines having to comply with the Legal and Compliance requirements that have emerged throughout the world. Some examples of this are the GDPR [42] and the US Health Insurance Portability and Accountability Act (HIPAA) [43], that affects healthcare data.
Another topic worth mentioning is the types of data ingestion and processing. These types are also the most commonly used in this kind of system. In the literature there are many different nomenclatures for the data types, however, in order to simplify the process, the main types will be presented and explained:

- **Batch type -** Considered to be the most widely used and consists in grouping large sets of information in a block unit called Batch before sending them [44]. From Figure 2, we can see an example of how batch works. The several data sources send information to the system which creates Batches, that are pieces of information grouped together that when reaching a certain amount of data, is transferred somewhere so that it can be analysed. This transfer process can take minutes to days which creates a delay in having the available information updated. A perfect example of this would be when a company registers the whole log information only at the end of the day during the night because the system is under less stress and there aren't as many users;

- **Micro-Batch -** Subset of Batch processing and according to [45] is an adaptation of the traditional processing type to a more digital world that constantly generates great amounts of data and purely consists of generating smaller batches of information and processing them more frequently;

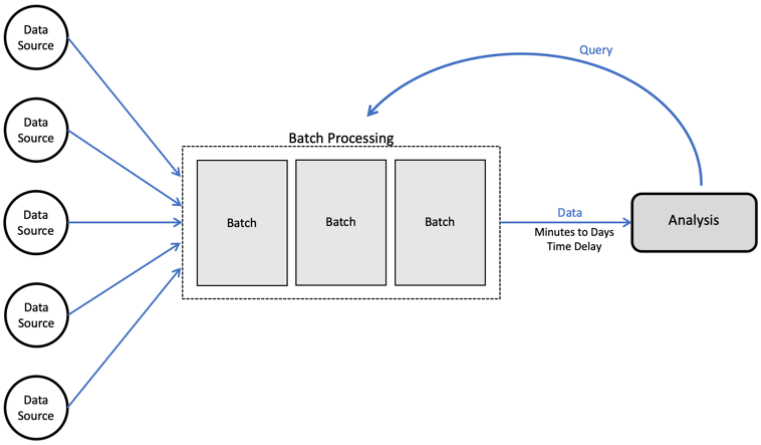


Figure 2: Batch Processing [46]

- **Stream -** As stated in [47], it is a type that focuses on dealing with information as soon as it is created by making it available in a publisher/subscriber channel [48]. From Figure 3, we can see an example of how the Stream type works. The several data sources send the information to the system which by using specialized pipelines, is continuously sending and constantly processing information which is allowing data to become available for analysis in near real-time results. The transfer process is near real-time and happens as soon as the information arrives and is processed;

- **Event Stream -** Type that focuses on the Event Sourcing pattern described by Martin Fowler in [49]. This type of Stream processing focuses on the idea that all the changes in a system are stored as a sequence of events, which are objects stored in the sequence they were applied to instead of single raw information [50].

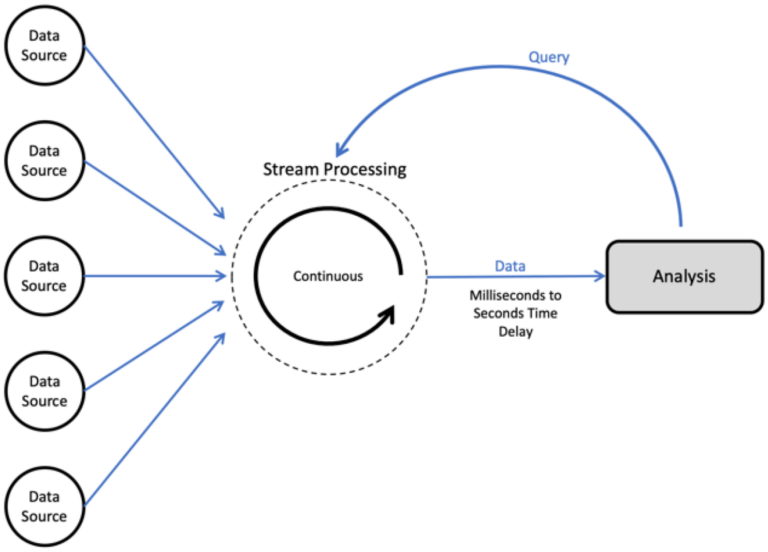


Figure 3: Stream Processing [46]

2.4.1 ETL vs ELT

In this subsection, we will present two paradigms of gathering data from multiple sources and integrating them into one storage system [51].
The first one is the ETL process, represented in Figure 4 below, which is translated to the three different stages in data integration [52]. The phases are as follows:

- **Extract -** Process of collecting data from one or more sources and then preparing it for the next phase;
- **Transform -** Most critical phase as it is where you change raw data into meaningful data ready to be injected and analysed;
- **Load -** Revolves around getting the data provided from the transformation phase and saving it into some kind of storage system like a database or a data warehouse.

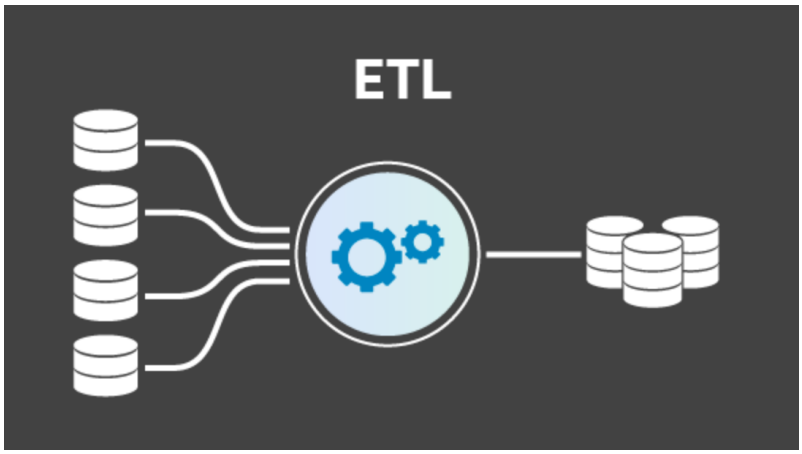


Figure 4: ETL Paradigm [53]

In [54], Alooma states that while ETL is a powerful tool, it is not without challenges. These are in conformity with the previously mentioned challenges of data processing, namely Scaling where for example if your system requires to reload data if you add new sources, with time, it is bound to decay and eventually become infeasible to add more sources. The second challenge is Transforming data which according to Alooma requires careful planning and testing due to the number of issues that can arise from data manipulation. Last but not least, the third challenge is dealing with the diversity of data sources coming from the many IoT devices, streaming sources, databases, CSV files and many others. Common use cases for ETL include scenarios where you want to perform complex computations as it's more efficient than on a data warehouse, where you need to perform extensive data cleansing or enrichment or whenever you are working exclusively with structured data.

Secondly is the ELT process which stands for Extraction, Load, Transform which involves the same stages as ETL with some important differences [55]. Whereas in ETL you transform the data before loading the information into the data storage, in ELT you load the unstructured data immediately into the data storage system to be later transformed when it is needed as can be seen in Figure 5. This is mostly a question of efficiency coming from the ability of data storages using pure computer power to perform transformations on big volumes of data. The benefits of ELT are efficiency and flexibility on the data sets since

you are saving the data as you collect it, not restricting the data schema. Common use cases for ELT include scenarios where data is massive but at the same time considerably simple and when data is unstructured whilst not requiring many initial transformations.

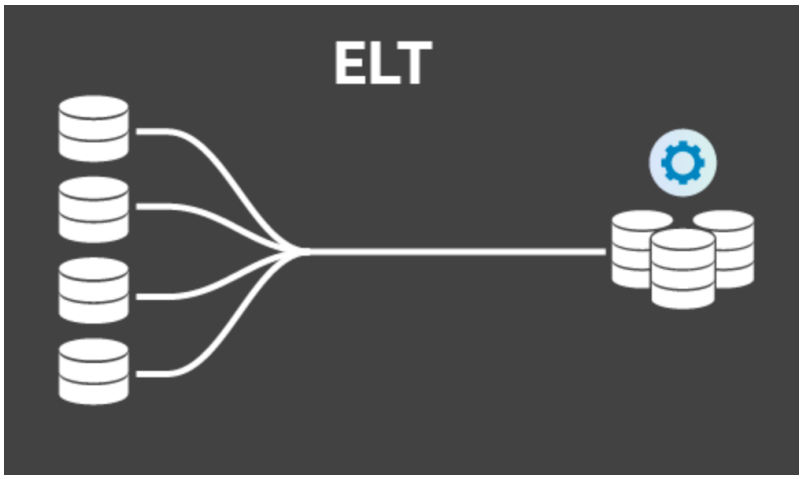


Figure 5: ELT Paradigm [53]

2.4.2 Traditional vs Modern

In order to understand what was used and how it evolved, we must first understand the notion of Data at Rest and Data in Motion. According to [56] data at rest means that the information is only analysed separately after it has been collected from several sources and stored in a storage system. For this, batch processing is the most common method as the focus is not having real-time data but instead is having the flexibility to support vast and often unstructured data sets. In contrast, data in motion while having the same procedure for collecting information differs from data at rest in where it analyses the data. In this method, the analysis occurs in real-time as soon as the information is collected and the event is triggered. This method which usually revolves around Stream Processing, allows for less stored information as data is cleansed and treated before storage and last but not least allows for more analysis with better and faster decision making.
From [57] we can see that in the 1990s traditional bare-metal ETL approaches emerged and companies would adapt to this convention. However as time went by and the amount of data grew, many were the problems with this approach. First of all, in order to scale vertically to increase computing power, companies had to make expensive investments in order to acquire better and more powerful hardware. In addition to this, given that databases were relational and not prepared for unstructured data, every time data sources appeared they had to readjust the system which indicated that the systems were highly inflexible and unscalable. Lastly, Traditional ETL was mostly based on Batch Processing which is not ideal for delivering information as quickly as possible removing this approach from the picture if we need real-time results.
With the increasing attention towards big data and in order to overcome the problems in more traditional approaches, more powerful and reliant alternatives were developed. This is where the ELT paradigm comes in, with its main focus being the use of data storages to provide better performance and faster data analysis. In [58], [59] and [60] we can see examples of data ingestion and analysis using Hadoop [61] which is open-source software that provides a framework for processing big data using the MapReduce programming model [62] and [63]. This ELT approach using Hadoop like frameworks whilst amazing for some use cases, still presented issues when used in a context of integrating a Smart City

where having access to information in real-time is imperative. An example of this is that when the data becomes voluminous enough with increasingly different sources, the use of Batch ingestion with analysis post storage starts decaying the performance and increasing the delay between the occurrence of an event and the analysis of the same. This use of data at rest concept to allow powerful data storages process the information as a bulk allows for extremely fast processing over stale information that is not time-critical, for example in [64] we can see five examples of Hadoop use cases such as financial companies using it for risk assessment or the energy industry using it for predictive maintenance.
Given the latest advances in IoT and the ever-increasing demand for scalable and fast-performance analytic tools, from [65] we can see that researchers started looking at Batch processing and realizing that it could not keep up with the demands of Big Data due to latency problems provided from the need to collect and organize data into batches before being processed. With this in mind and looking at data at rest, they started working on how they could accelerate the process and this led them to analyse the information while it was being transferred through some kind of smart pipeline. This would later change the paradigm of waiting for information to fill the batch and just process it immediately as it was being streamed and thus Stream Processing was born [66].
Ververica in [67] defines Stream Processing as being the act of processing information as soon as it is produced or received (Data in Motion). It is stated in [68] that the main objective of Stream Processing is to solve the problem of performance in processing continuous, infinite streams of data from live or historical sources. Also according to [67] and [69], Stream Processing addresses some of the issues and challenges that real-time analysts have, namely, the almost complete reduction of latency between an event happening, it being collected and our system reacting to it while the information is still new and meaningful. In addition to this, given that Stream Processing filters and transforms the data before it is stored and on the go, it ends up decoupling the infrastructure while reducing the amount of useless and meaningless information on the database resulting in more precise and precious data in a smaller and less expensive storage system. Also, due to the fact that it decouples the architectural components, stream processing is also more adaptable and advantageous in a microservices architecture. In [70] we can see some examples of scenarios that benefit from the use of Stream Processing includes real-time analytics like fraud detection systems for secure transactions and IoT edge analytics in Smart Cities. Last but not least, [65] describes Stream Processing as being a new technology that has become the go-to choice when it comes to IoT data processing due to how well thought it is and how well it works.

2.4.3 Serverless

In [71], serverless is described as an innovative platform for application execution similar to IaaS, which stands for Infrastructure as a Service [72], with the only main difference being that while IaaS, you are responsible for the application configurations, in Serverless, the provider handles it as an abstraction from the infrastructure to which we have no control over.
Moreover, Microsoft in [73] presents Serverless as one of 4 types of Cloud computing, which stands for the delivery of computing services in which the user pays for the services he uses which according to the source allows for lower costs, easy scalability and efficient infrastructure management. The other 3 types of cloud computing are IaaS, Platform as a Service (PaaS) and Software as a Service (SaaS).
According to [74], this term is a reference to the computing model that allows us to use a provider, that is in charge of making the resources transparently available and in charge of scaling up or down depending on the resources demanded, in order to execute certain

functions or methods without us not needing to have any care for the infrastructure.
Also in this source, Serverless Architectures are presented as a software design pattern that incorporates Backend as a Service (BaaS) third-party services and/or Functions as a Service (FaaS) platforms. They are described below, based on [75], as:

- **Backend as a Service -** Cloud computing model which focuses on automation and management of the backend side of a web or mobile application development;
- **Functions as a Service -** Cloud computing service is a platform that allows you to run functions in the cloud.

As stated in [76] and [77], the main identified benefits from a Serverless approach are the reduced operating cost, the lowering of complexity and time of development and DevOps and lastly the improved capability of scaling a system.
The operating costs in more traditional solutions where implementing redundant servers, in order to increase availability or fault-tolerance, and buying hardware for scalability needs, increases exponentially as in comparison with Serverless approaches where the costs directly relate to demands on scaling.
As to lowering the complexity of development and DevOps, Serverless approaches enable a development pipeline purely focused on the individual functions of the code instead of having to worry about Virtual Machines, Containers, web server management and other time-consuming annoyances.
Last but not least, the problem of scaling becomes transparent and insignificant, except on costs, to the developers as it is all managed by the cloud computing provider.
As most things in the Academic world, nothing is perfect and therefore serverless presents some drawbacks. In [78] and [79] we can see the limitations that Serverless Computing has, which are presented below:

- Not suitable for all types of applications since if the system is composed of long-running processes instead of event-based functions then it would be more expensive to go for a Serverless Approach;
- In some cases, latency can happen, for example as referenced in [78] if there are two functions on different nodes and the second one runs first, the notification will not be immediately resulting in lost waiting time;
- Vendor-lock is a risk due to the fact that we become dependent on services provided which also removes any possible open-source support from the equation. While this risk can be mitigated by working around it, it still requires previous planning and certain abstractions in the architectural design;
- Cold start issues on certain services which refers to resuming a previous state of a function when serving an invocation request. This kind of issue, affects newly instantiated runtimes that with frequent occurrences may greatly increase latency;
- Last but not least, based on [80], [81] and [82], Kinesis which is a cloud data streaming service has less performance than Kafka which indicates that when it comes to real-time processing, local solutions might take the lead.

2.4.4 Architectures

Among the vast literature on processing information while more architectures exist, as we can see from [83], two common names often come up. One of those is the **Lambda Ar-**

chitecture coined by Nathan Marz [84], [85], and is an architectural approach to describe fault-tolerance, scalability and general data processing. This hybrid deployment model combines the traditional batch processing with a high performance streaming pipeline in order to create a robust system against not only hardware failures but also the common human errors.

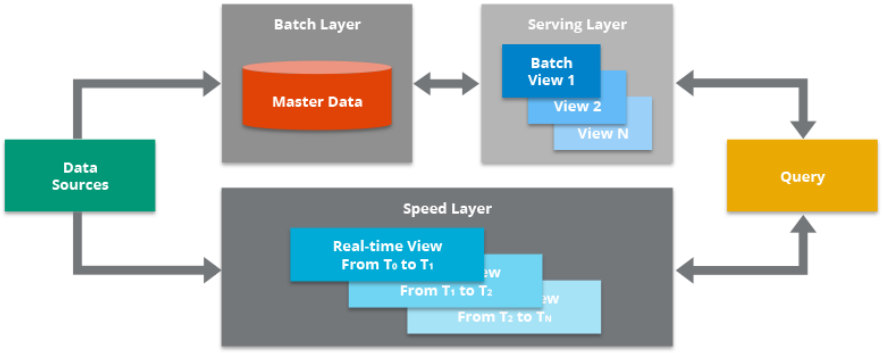


Figure 6: Lambda Architecture [86]

According to [87], the architecture, as shown in Figure 6, above, is divided into 5 components which we will describe in detail:

- **Data Sources -** Components made of the different sources that deliver information to the Batch and Speed layers. While it is not imperative, it is advised to implement in this step an intermediary streaming store in order to facilitate source connections since all data is sent to both the Batch and the Speed layers in order to enable parallel indexing;
- **Batch Layer -** In charge of two functions namely, managing the master dataset and pre-computing the batch views. The former consists of saving all raw data in an immutable and append-only format in order to have a trustworthy historical record of all data that enters the system. It is often used a CDC [88] design pattern which mainly consists of saving the data changes made to the database instead of all data in order to make the system more scalable and faster. The latter consists of turning the newly arrived information into batch views in order to prepare them for indexing in the Serving layer;
- **Serving Layer -** Layer used to index the latest batch views in order to make them queryable for the end-users. It is imperative that this is done parallelly so as to minimize the indexing times since while one batch is being indexed, new batches will be generated and queued for indexing;
- **Speed Layer -** Layer that uses Stream Processing in order to complement the serving layer by making the most freshly added data available for querying since there is a time lag interval between the data arriving, being converted to a batch and being indexed in the serving layer. This layer is used to allow all newly added information to be available and therefore diminish the time interval of which data is not available for querying;
- **Query -** Component in charge of receiving end-users queries and delivering them to both the serving and the speed layer to receive the results and present a complete

overview of all the near real-time data requested.

As to the main advantages of this architectural approach, one of the main assets is the resilience and fault-tolerance capabilities based on the fact that the batch layer, if correctly implemented, does not allow information to be updated or deleted and therefore reduces the risk of human or hardware errors corrupting the data. In addition to this, despite being susceptible to errors and there being a chance for data corruption in the Speed layer, even if the layer itself fails since information is being replicated to the Batch layer, which cannot be corrupted, the results will eventually synchronise. This mechanism means that the information will always at some point be consistent and therefore the Lambda architecture enables to some extent, all the three characteristics in the CAP theorem [89], namely Consistency, Availability and Partition Tolerance. It stands to notice that since the Speed layer is prone to errors, we must choose between increasing the consistency or making this layer more available [90].
Additionally, when it comes to latency, due to the Stream processing software used in the Speed layer, recent data becomes immediately available while the Serving layer is indexing the older batches of information which results in reduced latency and ultimately enables all information to be available near real-time.
Alternatively, as mentioned in [91] by Jay Kreps, the main drawback of this architectural approach consists of having to reproduce all the results in two complex distributed systems in the Batch and Speed layer which makes the process extremely complicated. Such complexity comes from having two different pipelines which use completely different technologies leading to problems not only on the synchronization between the layers but also on supporting and maintaining distinct distributed layers with completely different. Last but not least, there is the fact that this architecture re-processes every information in the batch and speed layer which in some scenarios may not be beneficial.

The other Architecture that often appeared during the research for this thesis was the **Kappa Architecture** which according to [92] is a simpler alternative to the Lambda architecture created by Jay Kreps in 2014. In [93], it is stated that Batch oriented systems cannot stand to the continuous and limitless nature of data and therefore, Kappa's architectural approach revolves around the idea of focusing entirely on the Stream Processing system and dropping the Batch layer as seen in Figure 7. Hazelcast in [94] states that this design model focuses on achieving near real-time results by reading the data and transforming it immediately after it is on the pipeline which enables recent data to be quickly accessible. This architecture also supports historical analytics but instead of using a database, the data store is an append-only permanent.

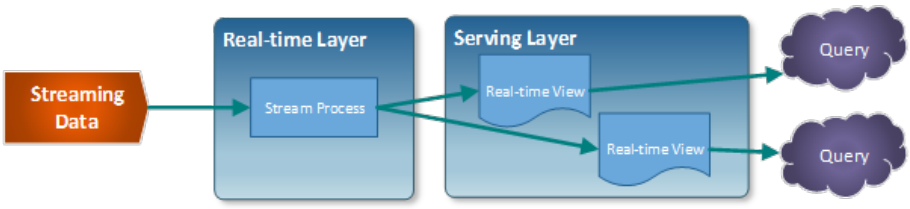


Figure 7: Kappa Architecture [95]

The main difference between Kappa and Lambda architectures is that in Kappa all the information is treated as a stream so there is only the stream processing engine which extremely simplifies the complexity of only having to deal with a Stream Processing engine. As to the layers of Kappa, the Streaming Data component coincides with the data sources

from the Lambda architecture, the real-time layer is often called Speed Layer as they serve the same purpose and last but not least, the serving layer while similar to the one on the Lambda architecture, has some differences since it usually consists of a database used to index the transformed data from the real-time layer and making it available to query.
The main benefits of Kappa are that with a Stream Processing that is good enough when it comes to performance, you may not need a specialized Batch Processing alternative which means you can just focus all the resources to one technology which tremendously simplifies not only the development process but also the maintenance and support on the technologies and also removes the need to have synchronization which adds to the previously mentioned advantages. Additionally, assuming good performance on the processor, you can use the parallelism capabilities in order to read streaming data while it is being processed allowing to replicate the batch processing layer in a faster and more efficient manner. Last but not least, since we drop the batch layer, there is no more re-processing every information but instead only becomes a requirement when the code changes which in the long run indicates that on normal behaviour we cut the processing weight in half.
As to the drawbacks of the Kappa Architecture, since we cut down heavily on the complexity of the architecture, we also reduce on the fault tolerance mainly due to the absence of the batch layer, we no longer have guaranteed consistency after a certain period of time and therefore errors may occur during the data processing or while updating the database which when all is said and done indicates we require an exception manager to reprocess or reconcile the information when needed.

2.5 Existing Tools

In the following section, the most relevant and worth mentioning technologies for data Ingestion, data Processing and data CEP will be presented. It stands to note that since this subject is not new, and therefore there exist many technologies, especially on the Processing layer as we can see in [69]. Given that there are too many technologies to include, only the most adequate, most popular and open-source Stream Processing frameworks, will be taken into account due to the scope of this thesis. Last but not least, the main purpose of this chapter is to provide insight into the most appropriate and used technologies in the market.

2.5.1 Ingestion Layer

Apache Nifi

According to [96] and [97], Nifi is open-source software that deals with automation and management of distributed data flow among systems. Nifi's origins trace back to the National Security Agency (NSA) which developed the software called "Niagara Files" that became the base for Nifi after a transfer program in 2014. Nifi, offers a web-based user interface that allows users to design, control, manage and monitor the different flows of directed graphs of data routing, transformation and system mediation logic.
In Nifi, once data arrives, it is represented as a Flowfile which is meta-data that allows processing quite a number of different types like CSV, Database Records, Audio, Video and others. Additionally, in [98] states that a Processor is a component that allows to perform some kind of work in a Flowfile, be it as data as a whole, or just its attributes or contents. It is defined as the basic building block that may perform the most various functions like reading, writing, updating, ingesting, routing, extracting or modifying Flowfiles.
Processors are connected through Connectors which graphically can be described as just

arrows, however, every arrow contains a queue with back pressure which can be configured. These processors, instead of being individually configured, a Flow Controller is used in order to provide the Processors with the information they need.
As to the advantages of Nifi, one of its main features is that to change a processor's settings you only need to stop the processors directly connected instead of having to stop the whole data flow. Additionally, Nifi contains a well-thought implementation of flow-based programming concept, explained in [99], allowing to manage data flows easily while offering real-time control with a GUI to monitor the data movement.
Moreover, Nifi contains a mechanism called Data Provenance as seen in [100], which is a service in charge of recording everything happening in the dataflows as a backlog which can come as very handy since it includes how the dataflows perform, the saved contents of the Flowfiles and other information. Last but not least, Nifi provides many tools and extensions that allow it to take advantage of the many existing Java libraries which easily helps integrate Nifi.
As to the disadvantages of Nifi, one of the main complaints, as can be seen in [101], is the often claimed bad user interface. Additionally, in [97] we can see that Nifi has a big learning curve which requires a user to understand well how the underlying system works. Lastly, one feature that many claims are missing is the ability to live monitor and debug features while being able to see each Flowfile statistics.

Apache Streamsets

According to [102] and [103], Streamsets is a collection of DataOps products designed to help control data drift, which as stated in [104], is the sum of unexpected or undocumented changes to data that result in corrupting it while possibly breaking processes. Streamsets provides two main products, the Data Collector and the Transformer. The Data Collector is the only open-source product of Streamsets and allows users to build optimized and flexible pipelines for continuous ingestion with little latency. The transformer allows to control and monitor the different dataflows using a user interface
In Streamsets every information that is ingested is automatically converted into the standard format, the Record, which all processors can handle as a stream of records. In Streamsets there are four types of processors, namely Origin processor which is the processor in charge of extracting data from the external sources, Processors which represent a stage where there is some kind of data transformation, Destination which have the function of saving information to some external storage and Executors which process events, generated by other Processors.
As to the advantages of Streamsets, it enables live monitoring and live debugging features with the aid of the User interface, that is considered to be quite good, using visual per-record statistics in each and every processor which allow to better manage and understand what is going on in the dataflow.
Last but not least, the disadvantages of Streamsets include it not being completely open-source as the only available component is the Data Collector. Additionally, in order to re-configure a Processor, the whole data flow must be stopped.

Nussknacker

According to [105] and TouK in [106], Nussnaker appeared as a response to the need for real-time processing and started as an Apache Flink process authoring tool. Nowadays, Nussknacker is open-source and allows to design, deploy and monitor processes through a GUI, as shown in Figure 8, in which you draw a diagram and the new processes are running in a Flink Cluster.
Some of the features of Nussknacker include a sandbox environment to allow users to deploy,

test and debug the processes they created using the GUI. Additionally, as stated by TouK, other features include having subprocesses, versioning, generating PDF documentation and lastly enabling the migration between environments.

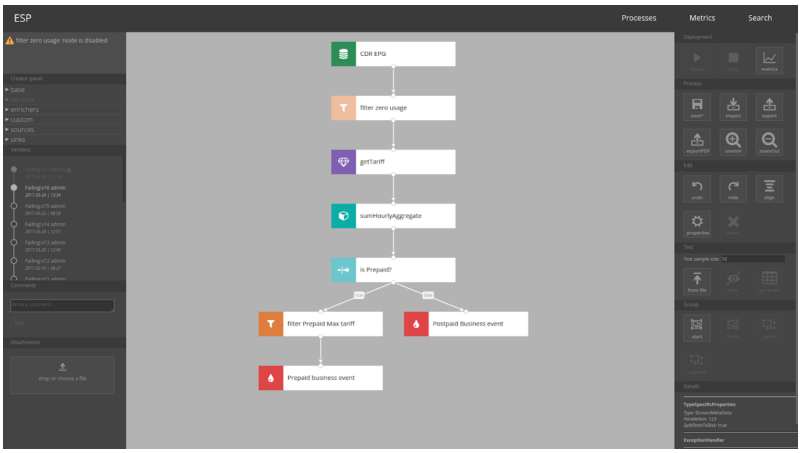


Figure 8: NussKnacker User Interface [107]

2.5.2 Processing Layer

Apache Storm

In [108], Storm is described as a distributed real-time processing framework that became open-source when it was acquired by Twitter.
According to [109] and [110], Storm is represented by a topology, similar to a MapReduce but instead of eventually finishing, it just runs forever. This topology forms a DAG, further described in [111], and consists of spouts, which are sources of streams, and bolts, that are processing or sink operators in which some kind of operation occurs that may lead to the emission of newly created tuples. Generally, a spout will read a tuple from one or many external sources and release it into the topology, which means that the bolts subscribed to the spout will receive the input. The example in Figure 9 below, taken from [69] which is a simple word count topology, better exemplifies this. Storm actually uses micro-batching which according to [112], makes the tool flexible for several different use cases.
Storm contains built-in support for sliding and tumbling windows in which any process in the topology sends an acknowledgement to the executor for a processed tuple that if results in failure, Storm re-sends the tuple.
In [113], which compares Spark Streaming and Structured Streaming to Kafka, Flink, Hazelcast and Storm, states that with low data generation and simple stateless pipelines, Storm is a considerably good option if stability, community support and resource consumption are must-haves.

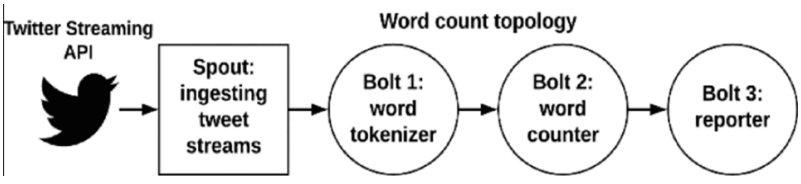


Figure 9: Storm word count topology [69]

As far as disadvantages are concerned, also on [109], since the acknowledgements can only be accepted once a window operator completely removes all the tuples from the window,

this can be an issue for lengthy windows with small slides. It is also stated that despite having a backpressure thread, it is centralized and highly complex which in some cases can block or interrupt the system.
Also on [112] states that Storm can have its reliability compromised due to the fact that the ordering of messages is not guaranteed by the system.
Lastly in [114], it is stated that Twitter used Storm for many years but eventually developed and changed to Apache Heron, due to the increasing amount of issues related to scalability, ability to debug, manageability and efficiency in sharing cluster resources that ultimately led to the loss of performance and decaying of the processor.

Apache Spark

Spark is considered in [115] a unified analytics engine for large-scale data processing containing several high-level tools for different processing scenarios. At the moment of this thesis, in [116] we can see that Spark provides two ways of dealing with streaming data, namely Spark Streaming and Structured Streaming. While the former focuses more on Batch processing while using old data structures, the latter is more inclined to real-time processing and is considered to be the future of Spark Streaming. While they have nearly the same architecture, Structured Streaming no longer depicts the batch concept and instead, the data is appended to the continuously flowing data stream in which there is a result table that is updated every time a row of the data stream is processed, as can be seen in Figure 10 below. This means that, as stated in [117], Structured streaming, provides two execution modes, a micro-batch and a continuous processing mode for event-driven processing.

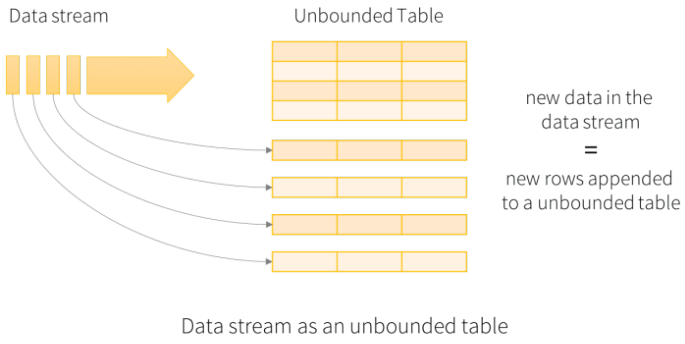


Figure 10: Structured Streaming's unbounded table [118]

As to data structures, on the literature, there are many comparisons between the RDDs and DataFrames, like [119], but as it is also stated in [116], those comparisons usually have the same output which is that the latter structure is not only more optimized for processing but also provides more data transformation options.
In the same source, it is also asserted that with the addition of event-time handling, Spark can now handle late data to get more accurate results. Besides, with the inclusion of restricted sinks, Spark always provides end-to-end, exactly-once semantics which means that in spite of any possible errors, each event only affects the final result once which protects the system against duplicate or unprocessed data.
In [120], Spark Streaming is compared to Storm and Flink and while it is concluded that there is no clear winner, it is stated that for huge incoming volume in high-throughput environments, where the information is massively and quickly generated, and where latency

is not of the priority, Spark works best. It stands to notice that for lower volumes, the results are not the same.
The authors in [121], compare Spark, Storm and Flink in a number of different experiments and conclude that if a stream contains skewed data [122] then Spark is considered the best out of the three.
In line with [123], Structured Streaming is compared with Spark Streaming, Storm, Flink, Kafka Stream, and Google Dataflow. They concluded that this processor makes integration with larger applications easier and that it is a good choice for Batch jobs due to the prefix integrity guarantee and its incremental batch queries.
Last but not least, in the previously mentioned [113], the authors conclude that Spark Streaming is a good choice if throughput is more important than latency and that Structured Streaming can be a favourable choice if usability is the focus since it provides SQL-like interface to build the processing pipelines.

Apache Kafka

According to [124], Kafka is considered to be an open-source distributed streaming platform that was originally developed by LinkedIn in 2011.
In [125], Kafka is described as a publisher-subscriber system composed of producers that publish information to a topic, which is a queue, and consumers that read the information from a topic. Each topic is split into one or more partitions which enables Kafka to store, transport or replicate data easily. In Kafka Streams, the Java library specifically for handling streams of information, a topology is a graph of stream processors (Nodes) that are connected by streams (Edges). A stream processor represents some kind of processing transformation applied to data and a stream represents a number of unlimited and immutable data records.

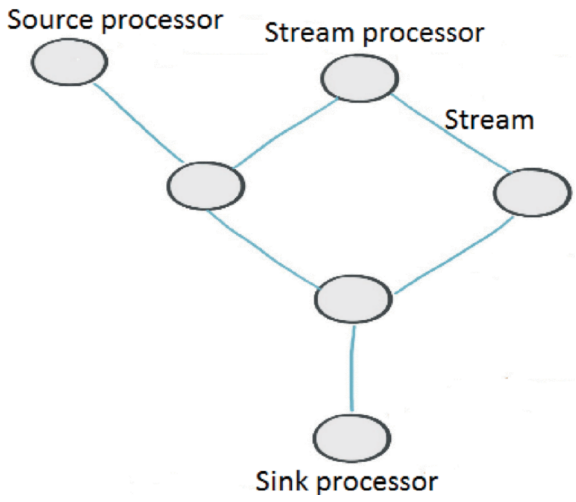


Figure 11: Kafka Streams topology [69]

In [69] and in Figure 11, we can see that there is a source processor which extracts the data records from one or more topics and forwards them to the connected stream processors that will perform some data processing function. Consequently, there is a sink processor which stores in another Kafka topic, the information that was delivered to it from the previous nodes.
In [126], we can see that Kafka guarantees exactly-once semantics, provides high-availability, SQL support and functionality and supports stateful operations. Additionally, Kafka Streams can be easily integrated and deployed since it works with the existing Kafka

application through an API.
Also according to [113], Kafka Streams is said to be the easiest system to deploy and troubleshoot with the support of not only good documentation but also much support in the internet community. In addition to this, it is also stated that if the low latency goal can be somewhat sacrificed, then Kafka is still a good option.
Lastly, on [117] it is stated that for systems already residing on a Kafka cluster with no requirement for very high-throughput and have reasonable latency requirements, that Kafka Streams offer a possible alternative to Flink with the advantage of being more easily integrated and installed.

Apache Flink

According to [127], Flink is a distributed processing engine for stateful computations over data streams, that was released in 2011.
In [69], it is stated that Flink is made of streams and transformation operators that are depicted in streaming dataflows. A stream is described as a continuous flow of data records, which can be sourced from message queues, file sources or socket streams, and an operator is a process that receives streams, carries out some kind of transformation function and produces a processed stream.
As we can see from Figure 12, a dataflow that resembles a DAG [111], starts with a source operator which connects to one or more sources to extract information and finishes with a sink operator which writes data to some storage or other operators.

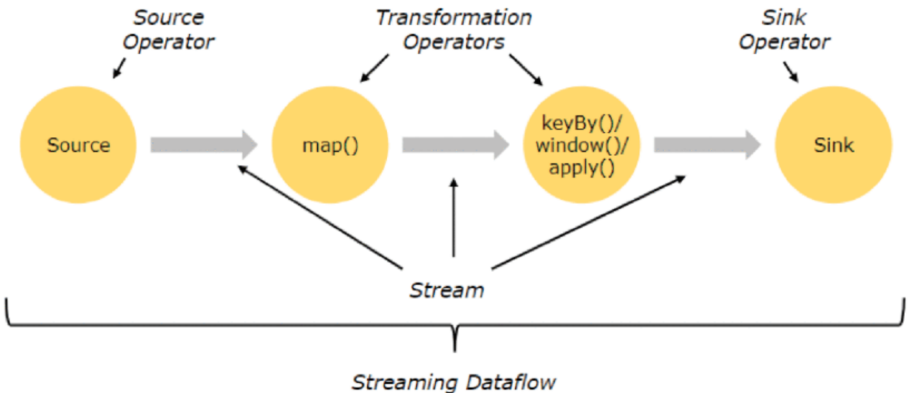


Figure 12: Flink's programming model [69]

Also in [127], it is stated Flink can process unbounded and bounded data where the former include streams which start but have no defined end which means that they are continuously processed and the latter include streams which start and end which means they can be fully ingested before being processed. Additionally, this source states that Flink due to its parallelism in executing thousands of tasks concurrently in a cluster, it is designed to run stateful streaming on any scale. Also on this, it is described that Flinks asynchronous and incremental checkpointing algorithm guarantees exactly-once semantics with high-performance and low latency.
In the previously mentioned [117], the authors consider Flink to be highly recommended if latency is a critical requirement even if it's only in certain periods of time. Additionally, Flink is also recommended if the needed requirements include a balance between latency and throughput. Also to support this, the authors in [113] report that when the primary tiebreaker is latency, then Flink is the best choice. Furthermore, in [121], Flink is described as the go-to stream processor in cases where latency and performance is the main factor, and it is also described to work better than Spark and Storm in data fluctuations, on use

cases containing large windows and is even described to have better overall throughput in aggregations and join queries.

Hazelcast

According to [128], Hazelcast is open-source storage and computing at in-memory speeds. Hazelcast is divided into two subproducts which include Hazelcast IMDG and Hazelcast Jet. The former is described as a system that allows storing data in an IMDG which is described in [129] as a network or cluster of computers that allow the sharing of their RAM and the processing power parallelly to provide higher application performance. As to the latter, Hazelcast Jet was introduced in 2018 as a distributed processing system built on top of IMDG which makes it the most recent processor in this list. According to [113], Jet, similarly to previously mentioned stream processors, makes use of DAG where the nodes relate to some kind of processing operation and the edges represent the data flow and act as a buffer for data produced by previous nodes that let more advanced nodes pull the information in it. This means that queues are running at the same time between processors making these more available and increasing the performance whilst reducing their pressure. Additionally, Jet has an engine that makes the decisions on the number of tasks or threads that are running in order to make the most out of the Jet tools so as to achieve high performance.
As described in the same source, Jet has the lowest latency until a certain data rate (110kTps) where it starts growing exponentially which indicates that for certain volumes of data, Jet achieves great latency. Lastly, while Jet is said to be easy to deploy and configure, it stands to notice that since it runs on its resource manager, it may become hard to integrate into bigger processing environments.

2.5.3 CEP Layer

Esper

According to [130], Esper from EsperTech is a popular open-source language compiler and runtime for CEP and streaming analytics that is available for Java and .NET that offers a language similar to SQL called EPL for complex event detection. Also on that source, the Esper compiler and runtime, instead of storing data and executing queries against it like a database, allows applications to store queries, or as they are called in Esper, EPL Statements and run data streams through those statements allowing real-time response when the conditions from those statements are met, which results in a continuous execution model instead of a querying one. As stated in [**?**], Esper introduces two methods of creating statements, namely event stream queries and event patterns. The former is similar to SQL queries and are more related to the event stream analytics component since they provide aggregation, filtering, joining and other functionalities. As to the latter, the underlying system is a simple implementation of a state machine leading to statements that trigger when one or more events occur in a way that matches the pattern's definition. These patterns consist of atoms, which are the most simple and atomic filter expressions and operators which combine the atoms either logically or temporally. As to the Esper Engine, according to the documentation in [130] the Esper compiler, compiles EPL code into Java Virtual Machine (JVM) bytecode so that the resulting executable code can run in the Esper runtime environment which runs on top of a JVM. The user interacts with the engine by compiling and deploying modules that ultimately contain statements. As to how to get data from the engine, Esper offers two methods, namely, listeners and subscribers wherein both, the user has to first define them using Java language to implement what will happen when the statement's conditions are triggered and lastly, the user has to attach

them to the queries. One last thing worth mentioning about Esper is that the user, to create and add new statements, does not have to compile the whole application again which means that rules can be injected dynamically during runtime.

Drools

According to [131], Drools is an open-source Business Rules Management System released under Apache License 2.0. Drools currently encapsulate several projects like a Business Rule Engine, Web authoring, Rules Management Application, and runtime support for Decision Model and Notation models. As to the CEP component, Drools Engine is the rules engine in the Drools project, in charge of storing, processing and evaluating data to execute the rules previously defined. In Drools, incoming data is referenced as facts and the engine's objective is to match those facts to the rule's conditions to determine when they are executed.
As we can see from Figure 13, taken from the official documentation from the previous source, the Drools Engine is composed of the following components:

- **Rules -** Business rules or Decision Model Notations containing at least one condition;
- **Facts -** Data entering or changing in the engine which is matched to the rule's conditions;
- **Production Memory -** Location where rules are stored;
- **Working Memory -** Location where facts are stored;
- **Agenda -** Location where activated rules are registered and sorted.

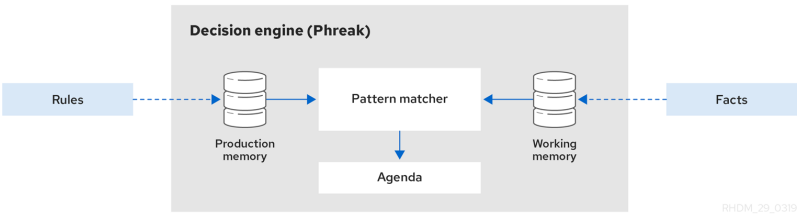


Figure 13: Drools engine composition [131]

As the author states in [**?**], In the engine, the matching of data against rules, or facts, is called pattern matching and it can result in firing rules when the agenda schedules them, which, as a consequence, leads to the firing of rules that can match against other rules that in a chain reaction that also fires those rules. This mechanism is referred to as forward chaining. Also on the same source, the author references that the reason behind Drools rule engine's speed and scalability is the Rete-OO algorithm, which is an enhanced version of the Rete Algorithm for object-oriented systems, in charge of matching the facts against the rules controlling which rules activate.
Last but not least, Drools Fusion is the module that gives the rule engine the capability to model events. In other words, it gives the engine the ability to define facts as events, manage the event cycles and manage the temporal/sliding windows.

This page is intentionally left blank.

Chapter 3

Requirements

The following chapter aims to tackle two important subjects. The first one is the description and specification of all the Requirements that appeared throughout the process of this internship and during the development of the proposed system. The second topic is the presentation and explanation of the proposed architecture used to overcome the challenges posed by the advances in data in a Smart Cities environment as previously mentioned in the State of the Art.

3.1 Scope

The scope of this internship revolves around a data ingestion system that allows for the integration and process of data streams into Ubiwhere's analytics project, the Urban Platform. In addition to this, the system will include a Rule Mechanism allowing end-users to cross-analyse in real-time the streams that are constantly being transformed and outputted from the processing tool. This application must allow the system users to create, add, remove and monitor the so-called rules, which are pre-established comparisons between the available data set fields with certain user-defined thresholds. Lastly, users must be able to add, remove or edit these thresholds as they are the triggers for the Alarms.

3.2 Stakeholders

Given the scope of the project, the main stakeholders that will interact with the system can be divided into three groups. The first group are the source systems which consist of third party operators like Here Technologies which generates data that is either pushed or polled into our platform. The second group is composed of the Urban platform which is the system that is given the output data after it is verified, validated and processed by our platform. And last but not least, the third group includes all the system users that may directly interact with the Rule Mechanism.

3.3 Constraints

There are two types of constraints that will have an impact on the requirements contemplated, namely Business constraints and Technical constraints. The former constraints are

associated with the availability of resources in order to fulfil the later detailed requirements and given the nature of this internship, the limiting resources are mainly connected to the available time for the development of the system. As to the latter, it stands to notice that the technologies used must be open-source and that the developed system must integrate with the existing APIs in Ubiwhere, more specifically, the ones from Urban Platform. The restrictions are described below using C#ID tag:

- **C#1: Development Time -** The proposed system must be developed until the end of the internship, more precisely, for 5 months by one person;
- **C#2: Integrate with Urban Platform -** The developed system must integrate at least two datasets with Ubiwhere's system, the Urban Platform;
- **C#3: Open Source -** The technologies used must be Open Source.

3.4 Functional Requirements

Given that our system can be roughly divided into two subsystems with all the user interactions being focused on the Rule Mechanism, we decided to divide the functional requirements into two subsets. On the one hand, the chosen method for specifying the Rule Mechanism interactions with the user was using User Stories, mostly due to the fact that the adopted methodology was based on Scrum which is an agile framework and, therefore, puts users at the centre of the conversation. User Stories can be described as informal, generic descriptions of a software functionality from an end-user perspective with the purpose of clarifying how the feature will have an impact. Additionally, to set the boundaries of the story and to make the user stories more measurable and easy to understand, the acceptance criteria will be presented for each User Story following the Gherkin language due to its human-readable nature for describing the system conduct [132]. On the other hand, as to the requirements for the System which includes the ingestion, processing and decision making using data, these will be specified using Use Cases mostly due to the fact that since there are no user interactions, most use cases relate to the actions that the system will take, and therefore require a higher degree of formality and granularity which ultimately offer a finer description of how the system acts.
In addition to this, the requirements will be prioritized according to the MoSCoW method [133], which, as stated in the source, can be described in the following scale:

- **Must have -** Mandatory requirements that have to be present in the proposed system;
- **Should have -** Important requirements that while not obligatory, would add relevant value to the system;
- **Could have -** Nice to have requirements that while they could improve the system, they are considered to be unimportant and can be overlooked;
- **Will not have -** Requirements that are not considered to be important for the time being.

As to the User Stories, the requirements are presented in the following format:

US#ID
Story:
As a <role>,
I want <feature>
so that <reason>.
Acceptance Criteria:
Scenario: Some determinable business situation
Given some precondition
When some action by the actor
Then some testable outcome is achieved
Dependencies: US#ID
Priority: Must Have, Should have, Nice to Have

Last but not least, as far as the User Cases are concerned, the requirements are presented in the following format:

UC#ID
Title: <Title>
Actor: <Actor>
Scope: <Scope>
Priority: <Priority>
Frequency: <Frequency>
Trigger: <Trigger>
Preconditions:
Preconditions
Main Success Scenario:

1. Step 1.
2. Step 2.

Alternative Paths:

1. Step 1.
   (a) Step 2.

Postconditions:
Postconditions

3.4.1 User Stories

In the following Subsection, all the user stories are presented:

US#1
Story:
As a system user,
I want I want to be able to create new Rules
so that so that the system may compare the data sets according to the rules in real-time.
Acceptance Criteria:

Scenario: User creates a valid rule
Given that I am in the application
And I am in the Create Rule menu
When I correctly choose the data sets, the fields to compare and the threshold of comparison
And I choose the "Create Rule" option
Then I should see a confirmation that the Rule has been created

Scenario: User attempts to create an invalid rule
Given that I am in the application
And I am in the Create Rule menu
When I incorrectly choose either the data sets, the fields to compare, the threshold of comparison or the Alarms
And I choose the "Create Rule" option
Then I should be presented an error message stating that the Rule has not been created and explaining what is wrong with it
Dependencies: None
Priority: Must Have

US#2
Story:
As a system user,
I want I want to be able to view all the existing Rules
so that I know what Rules are being used.
Acceptance Criteria:
Scenario: User views all Rules
Given that I am in the application
When I choose the "View Rules" option
Then I should be able to see all the existing rules
Dependencies: None
Priority: Must Have

US#3
Story:
As a system user,
I want to be able to filter what Rules I see
so that I can find the rules I want more easily.
Acceptance Criteria:
Scenario: User filters the Rules
Given that I am seeing all the Rules
When I fill the Search by Name, Data Set, Threshold or Alarms filter
And choose the "Filter Views" option
Then I should be able to see all existing rules with the restrictions I chose.
Dependencies: US#2
Priority: Could Have

US#4
Story:
As a system user,
I want to be able to see the details about a specific Rule
so that I can check what the Rule is applied to and what it is doing.
Acceptance Criteria:
Scenario: User views the details about a specific Rule
Given that I am seeing all the Rules
When I click on a Rule
Then I should be able to see all the details about the Rule.
Dependencies: US#2
Priority: Must Have

US#5
Story:
As a system user,
I want to be able to edit existing Rules
so that the existing Rules can be updated in accordance to my needs.
Acceptance Criteria:
Scenario: User correctly edits a rule
Given that I am viewing a specific Rule's details
When I click on "Edit Rule"
And I correctly fill the fields I want to change
Then I should see a confirmation that the Rule has been updated

Scenario: User incorrectly edits a rule
Given that I am viewing a specific Rule's details
When I click on "Edit Rule"
And I incorrectly fill the fields I want to change
Then I should view an error message stating that the Rule change wasn't applied and explaining what fields were incorrect.
Dependencies: US#4
Priority: Should Have

US#6
Story:
As a system user,
I want to be notified when a certain Rule threshold has been surpassed
so that I can know when and why it happened.
Acceptance Criteria:
Scenario: A threshold has been surpassed and a user is notified
Given that the Rule defined included a threshold of having a maximum of 10 cars in a certain road.
When the number of vehicles in that road surpasses the threshold defined
Then I should receive a notification of the event.
Dependencies: US#1
Priority: Should Have

US#7
Story:
As a system user,
I want to be able to define different levels of Alarms
so that I can differentiate when a threshold being surpassed is critical or not.
Acceptance Criteria:
Scenario: An informational threshold has been surpassed and a user is notified
Given that the Rule defined included an informational threshold of having a maximum of 10 cars in a certain road
When the number of vehicles in that road surpasses the threshold defined
Then I should receive a system notification about the event.
Scenario: A warning threshold has been surpassed and a user is notified
Given that the Rule defined included a warning threshold of having a maximum of 10 cars in a certain road.
When the number of vehicles in that road surpasses the threshold defined
Then I should receive a system notification and an email notification about the event.
Scenario: A critical threshold has been surpassed and a user is notified
Given that the Rule defined included a critical threshold of having a maximum of 10 cars in a certain road
When the number of vehicles in that road surpasses the threshold defined
Then I should receive a system notification while all the contacts defined should receive an email notification about the event.
Dependencies: US#6
Priority: Could Have

US#8
Story:
As a system user,
I want to be able to add emails to a rule
so that those emails can be notified if a rule is triggered.
Acceptance Criteria:
Scenario: Add emails to a rule
Given That a rule with an alarm level above informational is previously created.
When I input a number of emails and submit the request
Then I should receive a message with the added emails.
Scenario: Try to Add emails to a rule that can't have emails
Given That a rule with an alarm level informational is previously created.
When I try to find where to input the emails
Then I should not be able to find the option.
Dependencies: US#1
Priority: Should Have

US#9
Story:
As a system user,
I want to be able to remove emails from a rule
so that I can change which emails are notified if a rule is triggered.
Acceptance Criteria:
Scenario: Remove email from a rule

Given That a rule with an alarm level above informational is previously created and at least an email has been added.

When I choose the rule and input the existing email I want to delete

Then I should receive a message notifying that the email has been removed from the rule.

Scenario: Try to remove an email from a rule

Given That a rule with an alarm level above informational is previously created and at least an email has been added.

When I choose the rule and input an email that has not been added.

Then I should receive a message notifying that the email has not been found and therefore, was not deleted.

Dependencies: US#1
Priority: Should Have

3.4.2 Use Cases

In the following Subsection, all the use cases are presented:

UC#1
Title: Data Ingestion
Actor: Nifi
Scope: Nifi polls data from Here technologies
Priority: Must
Frequency: Every 60 seconds
Trigger: Time
Preconditions:

A Component must be previously configured with a certificate and the URL with the location, API key and other optional metrics.

Main Success Scenario:

1. Nifi performs a Get Request to Here technologies using the predefined URL;
2. Nifi saves some data before splitting;
3. Nifi splits the data into events;
4. Nifi merges the events with the data previously stored;
5. Nifi remodels the events into a better format;
6. Nifi sends the events to a Kafka topic.

Alternative Paths:

2. Nifi does not store data because it is empty;
   (a) Nifi adds empty fields to the event.
3. Nifi fails to split events due to malformed data.
   (a) Nifi redirects the events to a failure queue where they can be later analysed.
   (b) The use case ends with those events not being sent to a Kafka Topic.

Postconditions:

The data has been split into events and sent to a Kafka Topic.

UC#2
Title: Data Processing
Actor: Flink
Scope: Flink consumes and processes an event and sends it to a Kafka Topic
Priority: Must
Frequency: Every time a new event is sunk in the input Kafka Topic
Trigger: Arrival of data in an input Kafka Topic
Preconditions:
Flink Job has been executed in the server cluster
Main Success Scenario:

1. Flink consumes the data into a stream;
2. Flink converts the data in the stream into an input event;
3. Flink transforms the event into the correct output data standard event;
4. Flink converts the stream event into a JSON message;
5. Flink sinks the event into another Kafka Topic.

Alternative Paths:

2. Flink fails to convert the data because the input event is malformed;
   (a) Flink logs the occurrence and discards the event.
3. Flink fails to transform the event because some fields are incorrect or missing
   (a) Flink logs the occurrence and discards the event.

Postconditions:
The data has been converted into the correct standard and sent to a Kafka Topic.

UC#3
Title: Rule Engine data consumption
Actor: Rule Engine
Scope: Rule Engine consumes and sends an event to the engine
Priority: Must
Frequency: Every time a new event is sunk in the input Kafka Topic
Trigger: Arrival of an event in an input Kafka Topic
Preconditions:
The Rule Engine and Kafka must be running and the necessary configurations for these events must have been made.
Main Success Scenario:

1. Rule Engine consumes the data from the Kafka Topic;
2. Rule Engine converts the Json message into the correct data model;
3. Rule Engine sends the event to the Esper engine;
4. Rule Engine logs the occurrence.

Alternative Paths:

2. Rule Engine fails to convert the message due to a malformed event.
   (a) Rule Engine logs the occurrence and discards the event

Postconditions:
The Esper engine received the event.

UC#4
Title: Rule Trigger Check
Actor: Rule Engine
Scope: Rule Engine checks for triggered rules.
Priority: Must
Frequency: Every time a new event is sent to the Esper engine
Trigger: Arrival of event in Esper Engine
Preconditions:
The Rule Engine must be running and **UC#3** must have been triggered
Main Success Scenario:

1. Rule Engine analyses the event against a rule;
2. Rule Engine matches the rule's pattern;
3. Rule Engine triggers the callback method;
4. Rule Engine logs the occurrence;
5. Rule Engine repeats the previous processes for each rule.

Alternative Paths:

2. The event doesn't match the rule's pattern;
   (a) Rule Engine doesn't trigger the callback method.
5. Rule Engine has verified each rule.
   (a) Use case ends.

Postconditions:
An event matches a rule's pattern.

UC#5
Title: Rule Trigger
Actor: Rule Engine
Scope: A rule is triggered.
Priority: Must
Frequency: Every time an event matches a rule's pattern.
Trigger: An event matched a rule's pattern in the Esper Engine.
Preconditions:
The **UC#4** must have been triggered
Main Success Scenario:

1. Rule Engine retrieves the events that matched the pattern;
2. Rule Engine retrieves the necessary emails;
3. Rule Engine prepares the alarm and converts it to JSON;
4. Rule Engine logs the occurrence;
5. Rule Engine sinks the event into another Kafka Topic.

Alternative Paths:

2. The rule didn't have attached emails or had an Alarm level which didn't include them;
   (a) Rule Engine doesn't include the emails in the alarm message.

Postconditions:
An alarm is sunk into a Kafka Topic.

UC#6
Title: Alarm Received
Actor: Rule Alarm
Scope: Rule Alarm consumes an alarm and triggers notifications.
Priority: Should
Frequency: Every time an alarm is put in a Kafka Topic.
Trigger: An alarm was sent to a Kafka Topic
Preconditions:
The Rule Alarm microservice must be running and the **UC#5** must have been triggered.
Main Success Scenario:

1. Rule Alarm consumes the data from the Kafka Topic;
2. Rule Alarm sends a notification to the Rule API;
3. Rule Alarm sends an email to each of the attached emails.

Alternative Paths:

3. The alarm didn't have attached emails or had an Alarm level which didn't include them;
   (a) Rule Alarm doesn't send emails.

Postconditions:
A notification and emails are sent.

3.5 Non-Functional Requirements

Non-functional requirements specify the quality attributes that restrict and impact the system design due to their architectural significance. Due to the fact that the developed platform must meet a variety of quality standards, it is crucial that we describe and survey how the system should behave and, therefore, a number of attributes will be presented:

- **Performance -** The system should quickly ingest and process the information arriving from the different data sources due to its low-latency needs. This attribute is partially validated in Chapter 6. Testing where we measured latency during Load Testing;
- **Scalability -** The system should be able to add more, different, data sources and support the already existing ones to increase the amount or rate at which they produce data without the decaying of the system performance. This attribute is met in the Proposed Architecture and was also validated in Chapter 6. Testing where we measured throughput during Load Testing;

- **Interoperability -** The system should be able to collect information from several data sources and process them while enabling the processed data to be available to be requested by the Urban Platform. This requirement is met in Chapter 4. Proposed Architecture by choosing to sink all resulting events in Kafka Topics so that the Urban Platform can access them;
- **Availability -** The system should be operational and have normal behaviour and performance when required. Additionally, the system should be able to recover from malfunctioning circumstances and in case of a fault turning into a system crash, the system administrators should be notified with the detailed cause. This requirement is met through the choices made in the architecture due to a microservice oriented pattern where the tools are not only isolated from one another by always communicating through the use of Kafka topics, but also have high availability. Additionally, all tools are running as Docker containers which can be horizontally scaled;
- **Modifiability -** The system should be easily replicable and expandable to cope with eventual changes in the existing data sets and with the addition of more data sources. This requirement is met by dividing the architecture into several microservices and it is also validated in chapter 5. Implementation where we present the necessary steps to add new data sets to the system.

Last but not least, for each quality attribute, a scenario depicting it's most relevant aspects will be presented in the following format:

Source of stimulus - An entity (E.g a human, a computer system, or other actuators) that produced the stimulus;
Stimulus - A condition that needs to be taken into consideration by the Artefact;
Environment - The conditions in which the stimulus occurs;
Artefact - A stimulated entity which can be the whole system or some parts of it;
Response - An action tackled by the artefact in reaction to the arrival of the stimulus;
Response measure - The criteria used to measure the quality of the action.
Result - A description of whether the attribute was tested and if the requirement was met.

Scenario 1

Quality Attribute	**Scalability**
Source of Stimulus	Data Source
Stimulus	Increase in the amount of information that consumes the system's processing resources.
Environment Conditions	Permanent or temporary increase in demand of system resources.
Artefacts	System
Response	The system should perform the necessary tasks without degradation of other quality attributes.
Response Measurement	Proof of Concept system should be able to process 100 events per second.
Result	Proof of Concept system should be able to process 100 events per second.

Table 1: Scenario 1 - Scalability

Scenario 2

Quality Attribute	**Performance**
Source of Stimulus	Data Sources
Stimulus	New information arrives at the system
Environment Conditions	Normal conditions
Artefacts	System
Response	The system processes the information.
Response Measurement	The system should be able to process an event in under 1 second 99.9% of the time.

Table 2: Scenario 2 - Performance

Scenario 3

Quality Attribute	**Availability**
Source of Stimulus	System
Stimulus	One or more faults occurs in the system
Environment Conditions	Faults have occured
Artefacts	System
Response	The system should continue to process data, should log the occurrence and notify the people responsible.
Response Measurement	The system should be available 99.9% of the time.

Table 3: Scenario 3 - Availability

Scenario 4

Quality Attribute	**Modifiability**
Source of Stimulus	Developers
Stimulus	Want to modify or add a data set.
Environment Conditions	Development
Artefacts	System
Response	Modification or addition of a data set has no impact on other data sets and has no side effects on the system.
Response Measurement	The data set should be modified or added to the system within a week of labor

Table 4: Scenario 4 - Modifiability

Scenario 5

Quality Attribute	**Interoperability**
Source of Stimulus	Urban Platform tries to get data that has been processed by our system.
Stimulus	The request to exchange information between our system and the Urban Platform
Environment Conditions	Systems wishing to exchange data are known prior to run time
Artefacts	System
Response	The request is accepted and the available data is exchanged.
Response Measurement	Our processed data is correctly made available for the Urban Platform 99.9% of the time.

Table 5: Scenario 5 - Interoperability

This page is intentionally left blank.

Chapter 4

Proposed Architecture

The following chapter introduces our proposition for an Architecture for a Smart Cities Data Ingestion platform. The tool analysis on the subject given the previously mentioned requirements, is provided, followed by the description of the Architecture and last but not least, a practical use case is delivered so that the proposed solution can be better understood.

4.1 Tool Analysis

Data ingestion platforms in Smart Cities environments require different technologies to fully cope with the challenges that arise from integrating Big Data like velocity, variety, volume and veracity, and therefore, the following subsection aims to go over the choices made about the presented technologies in the State of the Art, in order to fulfil the previously stated requirements. Given the architecture presented in section 4.2, the remaining major choices are related to the technologies used in the Ingestion, Processing and Rule Mechanism layer and therefore those will be previously discussed in this chapter. First of all, we will decide on the processing tool since it is the core of our Data Ingestion system which dictates the architectural needs and limitations and after that, we will establish the best Ingestion tool to support the processor and last but not least, we will go over our options regarding the Rule Mechanism tools in order to choose the best one.

4.1.1 Processing

To better understand the choice made, a table was presented, namely, Table 6, in which the possible tools are displayed as well as the major factors behind the decision. Those major factors will now be introduced and discussed.
First and foremost is **Latency**, which according to [117] is the time that the processing engine requires to generate an output given one or more inputs, that was displayed in the table through ratings from low to very high through the analysis and interpretation of [113], which reports and analyses the latency of the proposed technologies through a benchmark.
Additionally, **Guarantees** refers to the Fault Tolerance mechanism that indicates whether the messages being handled in a distributed system can be lost or duplicated in certain scenarios. As to the differences between at-least-once and exactly-once guarantees, the former indicates that each message may be multiply delivered and therefore while it can't

be lost, it may be duplicated. Alternatively, the later specifies that the message is only delivered once and therefore can't be lost or duplicated at the expense of added latency due to the increase in the message overhead.
As to **State-fullness** [134], this field represents the ability of the processor to handle data in a stateful manner. Whereas in Stateless stream, each event is independently handled from the others which means that the processor will treat every event in the exact same way, in Stateful Stream, there is a shared state between events which means that the anterior can influence how the posterior will be treated changing how the processor deals with it. While stateless is more easily scaled due to its simplicity, it cannot do real-time comparisons like aggregated counts for example to know how many errors that are in a certain time window.
Lastly, when it comes to **Community Support**, two fields were taken into account, namely **Git Pull Requests** and **Stack Overflow Questions**. Since both metrics are always changing according to how big the community is, the results displayed in the table that will be the target of analysis are about the values obtained on the 5th of May. The former is a Git mechanism for a developer to notify its team that certain changes have been pushed to a separate branch providing others with the ability to review and discuss the added features. This field, which is divided into open and closed requests, helps understand not only how often changes are made, but also how big the community is. One thing worth mentioning is that the pull requests can be divided into open which means that they are still being reviewed and discussed, and closed which means that the request was either accepted or refused. As to the latter, Stack Overflow is a question and answer site that is broadly used by all the programming community. By analysing the number of questions, we can have a better understanding of the size and the support behind each tool's user pool.

Processors Comparisons	**Flink**	**Kafka**	**Storm**	**Spark**	**Structure Streaming**	**Hazelcast**
Latency	Very High	High	Low	Low	Medium	Very High
Guarantees	Exactly-Once	Exactly-Once	At-Least-Once	Exactly-Once	Exactly-Once	Exactly-Once
State-fullness	Yes	Yes	Yes	Yes	Yes	Yes
Processing Model	Real-Time	Real-Time	Real-Time	Micro-Batch	Real-Time	Real-Time
Git Pull Requests	553 / 15273	866 / 9764	46 / 3350	227 / 32190	227 / 32190	832 / 5805
Stack Overflow Questions	5308	3047	2549	5225	1740	2449

Table 6: Processors Comparison

When it comes to **Latency**, in the benchmark mentioned above, Flink and Hazelcast showed the best results with Kafka being right behind them. Supporting Flink's results were other benchmarks, mentioned in Section 2.5.2, which didn't include Hazelcast, that often ranked Flink as the go-to processor when it comes down to low latency. Additionally, Flink is the only stream processing engine that was able to maintain results of under 2.5 seconds even when the throughput was increased to 150K Tps (Transactions per second). When it comes to Hazelcast, the most recent of the presented processors it was showing promising, competitive results until it started decaying exponentially after surpassing a certain number of Tps, more specifically 120k. While not alarming, this could pose a threat in busier, more high-throughput scenarios. As to Kafka, this processor showed the best results after the other two only decaying after 140kTps.
Moving on to fault-tolerance **Guarantees**, most stream processing engines provide Exactly-Once semantics except for Storm which only provides at-least-once. It stands to notice that some of the Stream Processing engines like Hazelcast offer the ability to choose what kind of guarantee each job adapts to offer more dynamic approaches.

Continuing to **State-fullness**, all stream processing engines provide the ability to have stateful computations, however, despite having this in common according to [134] there are conflicting views on the best implementation. An example of this is that while Spark Streaming only enables stateful computations through DStream (Discretized Streams) which is the basic abstraction of continuous streams of data in Spark, Flink includes stateful operators that makes stateful application development much easier.
When it comes to **Community Support**, as far as **Git Pull Requests** are concerned, there was no Git directly for Spark Streaming or Structured Streaming as the only Git belonged to Apache Spark. As these two tools are extensions of the core Spark API, there was no way of correctly knowing the number of git pull requests specifically related to them and therefore are considered outliers in this analysis. Apart from this, Flink showed the best results as it showed approximately 16k pull requests, followed by Kafka with 10.5k, Hazelcast with 6.6k and lastly Storm with 3.3k. As to the field of **Stack Overflow questions**, Flink takes the lead with 5.3k, immediately followed by Spark that holds 5.2k questions, with the remaining tools having below 3k questions each. This goes to show that Flink is the tool that appears to have the biggest community with the most support.
Alternatively, one other factor that was taken into account that wasn't put in the table was **Throughput**, which according to [117] is the number of messages that a streaming engine can process per second. The reason behind it not being on the table is that not only did different sources have different conclusions which made it difficult to generalise the ranking of different frameworks but also Hazelcast wasn't in any of the sources that compared this parameter. In [121], a series of different experiments were conducted analysing Flink, Spark and Storm. It stands to notice that Storm drops some of the connections when tested with high throughput with backpressure disabled. Additionally, it is concluded that Flink has better overall throughput for aggregation and join queries as well as for scenarios with large windows. Lastly, it is also concluded that Flink's throughput is not only higher but still maintains fewer fluctuations, followed by Spark and lastly Storm. Moving on to [117], the author compares Flink, Kafka, Spark Streaming and Structured Streaming in different performance-wise scenarios. On a sustainable throughput environment, which is the highest load of traffic that a system can handle with no signs of prolonged backpressure, Structured streaming reached the highest values in unconstrained latency conditions followed by Spark Streaming. Alternatively, on constrained latency conditions, Flink showed the best results followed by Kafka. On a single burst on startup workload, the author notes that micro-batch frameworks take longer to catch up in comparison to event-driven ones, and therefore, Structured streaming is the fastest streaming engine to process the burst of information followed by Flink, Spark Streaming and Kafka. Lastly, on a workload with periodic bursts, when the pipeline gets more complex, Flink is the framework who suffers the least from periodic bursts.
As a consequence of all the previous factors, we filtered the options down to Flink and Kafka. Ultimately, given that Flink showed the best latency results, was a name that stood out on the sources analysed, as a good option for high-throughput on an environment where latency is not discardable and also appeared to have the biggest community, the chosen stream processing engine for the proposed system was Flink.

4.1.2 Ingestion

Having reached a decision on the Processor, the next step is to take into account the possible ingestion technologies to deal with the collection, preparation and routing of data. Having the possible tools and the major deciding factors, Table 7 is presented. First of all, the technology used must be **Open-source** due to the constraint **C#3** and when it comes to that, StreamSets is the only technology pondered that is only partially open-source.

This is because StreamSets contains several products but only the Data Collector and the Data Collector Edge are available for free and whilst there are no limitations on these products, the fact that it has other limitations excludes it from the competition. Moving on to **Graphic Use Interface**, all available technologies considered in the State of the Art have a GUI, however, there was always the option of disregarding the GUI in order to develop a coded SDK that would need to be developed from scratch. Despite the advantage of this approach in having more control over the pipelines by casting off the limitations imposed by the GUI, this procedure would unnecessarily consume the most important resource in the internship, time. Also on this topic, while Nifi presents some obvious advantages over Nussknacker when it comes to community support, available tools and extensions that make use of Java libraries, and overall quality due to time it has been around and the upgrades it has received, it is often claimed to have a bad and not user-friendly UI.

Ingestion Tools Comparisons	**Nussknacker**	**Streamsets**	**Nifi**	**Coded SDK**
Learning Curve	High	High	High	High
GUI	Yes	Yes	Yes	No
Open-Source	Yes	Partially	Yes	Yes
Language	Scala	Java	Java	Any

Table 7: Ingestion Tools Comparison

As far as Nussknacker's advantages are concerned, the fact that it was developed exclusively for Flink, allows it to make use of its overwhelming performance and reliability to make streams faster and easier to develop. On top of that, the fact that Nussknacker uses Flink means that the learning curve of the proposed system as a whole reduces, making the system simpler, easier to develop and most importantly, less painful to maintain and troubleshoot. Lastly, the simplicity of Nussknacker compared to Nifi means that the learning curve that in Nifi is considered to be high due to problems in the GUI, is lower resulting in more time to focus on the proposed system.
As to other disadvantages, upon further analysis, Nussknacker's lack of documentation and examples, turn what would appear to be a reasonable learning curve into an extensive period of comprehensive code analysis to understand how to do simple things. In addition to this, Nussnacker only provides Kafka connectors as out-of-the-box sources and sinks quickly increasing the complexity by having to code any other connectors that may be necessary. To make things worse, whereas Nifi has data generator components that easily create simulated environments for testing purposes, in Nussknacker you have to manually implement those which roughly translates to additional work. Lastly, most of Nussknacker's documentation and examples are only available in Scala which, despite its similarities to Java, still adds unnecessary complexity in learning a new language since all the chosen technologies were in Java.
Due to all the reasons mentioned above, even if Nussknacker has some performance advantages with its simplicity over Nifi, because of Flink, it still loses in almost every aspect and therefore the chosen technology for the Ingestion layer in the Proposed System was Nifi.

4.1.3 Rule Mechanism

The next step is to take into account the possible technologies to create the Rule Mechanism. One type of data processing that is worth mentioning in the spectrum of the IoT is **CEP**. According to the author in [135], CEP can be defined as a mechanism that analyses

not only the data but also its context in order to trigger events. An example of how CEP can be used while analysing streams of traffic and environment, is to understand how the amount of traffic in a certain zone can relate to the different types of events in the Environment, for example, CO2 emissions. These event comparisons can be described as the Rule Mechanism that is to be developed in our system. Given that some stream processors offer the ability to develop CEP natively, we are given two options:

1. Choose a Stream Processor that has a CEP library to create the rule mechanism;

2. Use another technology to provide CEP functionality.

When it comes to the first option, of the processing tools analysed, Storm has an Esper integration library which roughly translates to using two of the aforementioned technologies where one must be Storm, Flink has a native library, namely FlinkCEP that has an API that allows the developer to define complex pattern sequences that may be extracted from the streams without having to create them from nothing [136], and other external libraries like Flink-Siddhi [137] and Flink-Esper [138] and lastly, Kafka which has an external library called Kafka-Streams-CEP [139]. When it comes to performance, FlinkCEP showed several advantages as it was natively present in Flink ultimately reducing the number of technologies to learn, develop and maintain and also making use of Flink's low latency and high throughput to easily scale and achieve better real-time results. Additionally, the fact that we could have a single technology for processing and evaluating rules, would mean that data wouldn't have to be carried between so many systems, ultimately reducing the latency generated from transporting data. Nevertheless, during the learning period, it came to the conclusion that both FlinkCEP and FlinkSQL don't allow dynamic runtime injection of patterns(or queries) [140] which ultimately implies that the rules would have to be present during compilation and therefore, the system would have to be stopped in order to add newer rules. In addition, Flink-Siddhi only runs in older versions of Flink 1.7 (currently at 1.12) which would create future version problems and Flink-Esper not only is undocumented but also hasn't been supported for over three years. Similarly to FlinkCEP, Kafka-Streams-CEP also doesn't support dynamic runtime injection of rules. Ultimately, these factors led to discarding the previously mentioned processor libraries to create the rule mechanism.
As to the other option, there are two tools that often come up as far as CEP is concerned, namely Esper and Drools. As far as dynamic runtime injection is concerned, both technologies allow to compile, deploy and undeploy rules or, as they are called in Esper, Statements while running in a multithread-safe environment. When it comes to performance, according to [141] Drools was concluded to have a better performance compared to Esper achieving overall better results though not by a lot. Despite its superior performance results, Drools also showed memory management issues leading to some unfinished experiments. Alternatively and on the same source, Esper was concluded to be a robust and acceptable compromise between performance, configuration flexibility and easiness of setup. As to syntax, Drools makes use of its own syntax called Drools Rule Language (DRL) which according to [142], is verbose and overall less attractive compared to Esper's EPL which is a SQL-like syntax that supposedly makes it easier to learn and develop.
In conclusion, both Drools and Esper can be integrated with Java Spring and both are viable options, however, given that Esper appears to have a less steep learning curve with a more simple, yet powerful syntax while also being flexible and easy to set up, it will be the chosen technology for the CEP layer.

4.2 Architecture

Taking into account all the previously mentioned information, we have come up with an architectural proposition which will depict a way to collect data from the many different data sources, process it and make the data available for the Urban Platform to consume it.
It stands to notice that the architecture chosen is not serverless due to the fact that not only is performance one of the main priorities, but also one of the technical constraints states that the technologies used should be open source.
One of the crucial points is that our proposed system was inspired by the Kappa Architecture since the research done showed that this approach was not only more appropriate for real-time streaming environments but also discarded the increasing difficulty of learning, developing and managing a Lambda Architecture. This approach solved some important issues like for example managing out-of-order events which in Flink is transparently handled and also allowed us to set our focus on the performance of the streaming platform which is one of our most important requirements due to the Big Data challenges that Smart Cities sources impose. In order to make things simpler, we can divide the proposed system into two main phases.
First and foremost, as we can see in Figure 14, is the base of the platform, or the called Data Ingestion Platform, which consists of a three-layered system that is divided into ingestion, processing and queueing. As the name suggests, the first layer of this subsystem is in charge of collecting, preparing and routing the information to the correct processors which will in turn perform any needed operations or transformations to the data. Lastly, the queueing system is responsible for handling the pressure of holding the data for Flink and for external systems, like the Urban Platform, to enable a Back Pressure mechanism [143] so as not to create a bottleneck in the entrance of these two systems and therefore create a more scalable system.
The division of this system into a layered architecture allows us to have a more flexible and modifiable system where changes to a certain aspect of a layer don't have direct repercussions on the other layers. Additionally, this allows us to simplify the problem while making it more manageable and ultimately more robust since the emergence of faults in a layer is contained in that environment. One other advantage that is worth mentioning is that by having an ingestion layer, we can always transform data before routing it to the processors by always modifying it into the same formats which ultimately not only simplifies the processes of changing and using the available information but also makes data more efficient and less prone to human error.

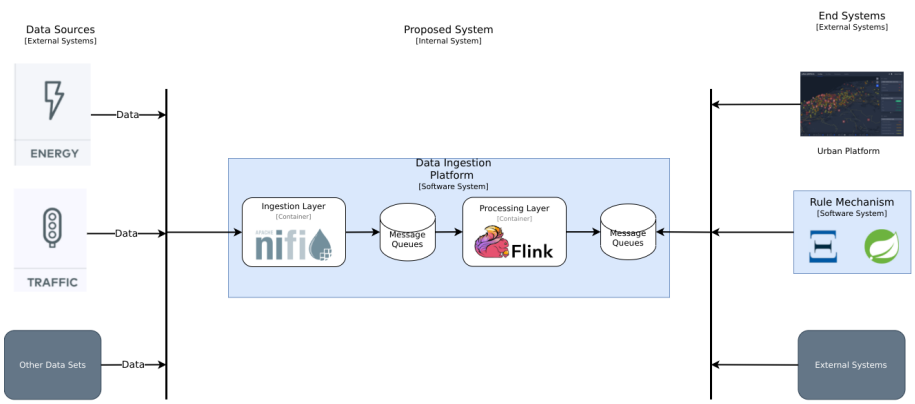


Figure 14: Proposed Architecture

Given this, the base of the proposed system consists of the following layers:

- **Ingestion Layer -** This layer is in charge of collecting data and ingesting it into the processor and as previously mentioned in subsection 4.1.2, the chosen technology was Nussknacker due to its use of Flink´s performance, its lower learning curve and its GUI. It stands to notice that this layer allows us to unify the formats that enter the processing layer, maximizing the fault tolerance of our system and its robustness towards format problems;
- **Processing Layer -** This layer is responsible for filtering and validating the data, checking the user-defined rules and applying any needed transformations on the data to be later sent to the message queue. The technology chosen was Flink since compared to the considered open-source solutions, it showed the best latency results and was considered a good choice for high-throughput scenarios;
- **Message System -** This layer is accountable for removing the pressure from Flink and the Urban Platform's entrances, so as not to push more data than these systems can process. Additionally, as to the message queues for the external systems, this layer can be thought of as intermediary storage that allows the Urban Platform, the Rule Mechanism and eventually other external systems to acquire the data that is being processed by our proposed system. For this, Kafka Topics will be used to store data so that it can be later fetched.

Lastly is the Rule Mechanism which will be developed using Spring Java. It stands to notice that since FlinkCEP was not an option due to the lack of dynamic runtime injection of rules, a new tool, Esper, had to be added and will be running parallelly to the Urban Platform. To further explain this, when Flink is done processing data and it is ready for the Urban Platform to consume, it will be stored in a Kafka Topic which will be then polled parallelly from both services preventing the pipeline from being slowed down because of the Rule Mechanism and also enabling the data to be immediately ready to the Urban Platform while at the same time, allowing the rule mechanism to perform real-time analytics resulting in faster results.

This mechanism will allow users to submit custom rules with custom thresholds so that the information flowing through the pipelines can be in constant comparison with the ultimate goal of setting off a user-defined alarm when those thresholds are surpassed. An example of why this rule mechanism is important can be seen in merging traffic and environment data. For example, we may want to be alerted as soon as a certain road is highly congested, so that we can check if the CO2 emission levels are dangerous or not. In certain scenarios which are more time-bound, the irreducible latency between collecting, processing, analyzing and comparing the data in the Urban platform may make the difference between a useful warning and a useless one.

4.3 Use Case - Traffic Flow

Taking into account all the different data sets that smart cities platforms ingest and analyse, there are many possible scenarios for our proposed architecture. To better demonstrate what these scenarios consist of, an example will be provided so that the process can be better understood and explained. For the different data sources, we can either have push-based systems that asynchronously send a message to a proxy which is in charge of not only determining the data type but also choosing and redirecting the information to the

right processor, or we can have a poll like systems that synchronously request information every previously set period of time The chosen use case, for demonstration purposes, was a poll data source and one of Ubiwhere's sources, more specifically Traffic Flow from Here Technologies [144]. As can be seen in Figure 15, a simplified illustration of the described example scenario, the data source is constantly making streams of data available for our system to collect, process and ultimately deliver to Ubiwhere's analytic platform, Urban Platform.

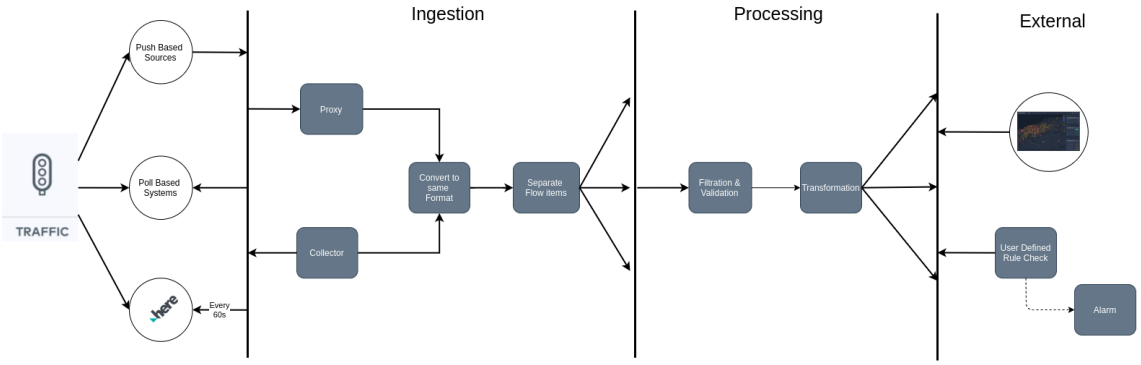


Figure 15: Use Case

The data acquisition process occurs as frequently as the source updates itself, more specifically, every 60 seconds through a REST API request, via polling. Given the unknown specification of the data format at the time of the request, the first step which also reduces flow redundancy is to convert different data formats to one common format, for example, JSON. In view of the micro batch nature of this source where a request can have one or multiple flow items, the last step of the Ingestion layer is to separate the information in the response into several different objects that are immediately sent to the Processing layer to be validated and transformed into the data set that will become the output of our system. Moving on to the Processing Layer, the first step is filtering and validating the information that was redirected, more specifically, this is where we assure the quality of the data by ensuring that each and every field that is imperative for the output data set is accounted for. In addition to this, this is also where we perform any coherence evaluations on the data in order to ensure that no data set consisting of inaccurate sources or wrong information is allowed to pass freely through the system. An example of when this is extremely useful is the common mistake of data sources resolved around locations, wrongfully switching the latitude and longitude information which results in the production of incorrect locations.
Having the data validated, the next and last step in the Processing Layer is to create the final data set and fill it with the available information. Taking into account that the available data might not be in the correct format, it is necessary to extract each and every possible useful piece of information and go over them in order to convert them to the format that is accepted by the Urban Platform. Some examples of these transformations are dates that may not be in the needed ISO 8601 format or location points that in Here Technologies data sources come as shapefiles and need to be converted to geoJSON.
Additionally, after processing the data until it is ready, it is then sent to a Kafka Topic which will store the data until it is polled by the Urban Platform and by the Rule Mechanism. Last but not least, the Rule Mechanism polls the processed and ready data from the Kafka Topic and sequentially sends it to the Esper engine which in turn cross-checks it against the user-defined rules for it to be evaluated and compared against other data sets with a certain customized threshold that works as a limit previously imposed that once surpassed, triggers an alarm which notifies the user. These rules can be added or deleted at any time during runtime using an API and the user can define any number of

rules. Lastly, since the alarm triggers are completely dependent on the result of the rule threshold having been surpassed, it is therefore a circumstantial occurrence.
All things considered, this is just an example for one data set and all these steps are repeated for each location which may include several flow items. If we take into consideration data sets other than Traffic Flow, in order to replicate the same procedure, some steps must be taken. First of all, in the Ingestion Layer, the proxy or the collector must be adjusted to be able to cope with the new sources, the format needed must be established and the conversions to that format must be taken into account and lastly, if needed, items should be separated. As to the Processing Layer, in spite of the steps remaining the same, the logic behind each particular validation or transformation is most likely different and therefore needs to be rethought and prepared.
Last but not least, if we extend this for several locations and If we look at the many different sectors that the Urban Platform analyses, the amount of data, the speed in which it is generated, the variety of the data and the need to have the data as accurate as possible easily scales manifesting the challenges of our system which are in conformity with the challenges of Big Data.

This page is intentionally left blank.

Chapter 5

Implementation

The following chapter presents the more challenging topics, that arose during the development period of this thesis and puts forward the solutions and the thought process behind them. First of all, we will go over the Development Environment and then we will go into detail about the Rule Mechanism. Sequentially, we will describe the Interface, the Grammar and we will provide information about Nifi and Flink. Last but not least we will provide an explanation of what needs to be done to add another data set to the system.

5.1 Environment

The following subchapter intends to describe the development environment while giving insight into the decisions made and the overall tools used.

5.1.1 Overview

At the start of a new project, the selected tools and technologies need to be installed and set up in an environment that In this internship, was the intern's computer. Given the complexity of the architecture and the number of microservices intended to be developed, a challenge arose due to the need to integrate multiple services. In view of the fact that installing all the architecture's components directly on the computer would turn out to be not only a time-consuming task but it would also lead to having to replicate the whole lengthy process in other computers which would most likely result in a quite a few numbers of dependencies, versions or packages errors. To mitigate this problem, the intern looked into the virtualization tool, Docker due to its advantages over Virtual Machines when it came to performance and disk space and also because it is a widely used tool in the company. Docker is a tool that makes creating, deploying and running applications easier through the use of containers which are units of software that package not only code but also all of its dependencies to the goal of making applications quicker and easier to set up in new environments. A container only becomes one when a container image, a standalone executable package of software that has everything that the application needs to run built-in, runs on the Docker Engine. Additionally, given the complexity of the architecture and the number of services developed, another tool, Docker-Compose, was used to simplify the process of defining and running multiple Docker containers. In Docker-Compose, you issue a single command to start all the services through the use of a configuration YAML file. Last but not least, it stands to reason that every component in our architecture is a Docker

Container with the exception of the interface since it requires command-line input.

5.1.2 Setup

With regards to the **Dockerfiles**, given that the language used not only to develop the Rule Mechanism components but also in the Flink Job's, was Java and since the Java and Maven images are quite spacious, we decided to use an optimized Dockerfile, depicted on Figure 16, that makes use of a smaller java image, namely *openjdk:8-jdk-alpine*, which ultimately only installs the required dependencies and packages allowing to create an image that is only 344Mb. This docker image is used in every Java component in our architecture and has allowed us to save not only disk space but improve the startup performance of the architecture. As to the remaining containers, the default docker images of the technologies were used.

```
# Dockerfile for Java/Maven Image
FROM openjdk:8-jdk-alpine

# Install Maven
# Remove cache and tmp files (Fixed an issue on running apk add)
RUN rm -rf /var/cache/apk/* && \
    rm -rf /tmp/*
RUN apk update && apk add --no-cache bash tar curl
ARG MAVEN_VERSION=3.6.3
ARG USER_HOME_DIR="/root"
RUN mkdir -p /usr/share/maven && \
curl -fsSL http://apache.osuosl.org/maven/maven-3/$MAVEN_VERSION/binaries/ \
apache-maven-$MAVEN_VERSION-bin.tar.gz | tar -xzC /usr/share/maven --strip-components=1 && \
ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
ENV MAVEN_HOME /usr/share/maven
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"
# speed up Maven JVM a bit
ENV MAVEN_OPTS="-XX:+TieredCompilation -XX:TieredStopAtLevel=1"
ENTRYPOINT ["/usr/bin/mvn"]
# Install project dependencies and keep sources
# make source folder
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
# install maven dependency packages (keep in image)
COPY pom.xml /usr/src/app
RUN mvn clean install && rm -rf target
# copy other source files (keep in izmage)
COPY src /usr/src/app/src
```

Figure 16: Java and Maven Dockerfile

When it comes to **storage**, volumes are the preferred mechanism for data persistence in containers and Docker allows the use of two kinds, namely Bind Mounts and Named Volumes. According to Docker's documentation in [145], binds are dependent on the host machine's operating system and directory file structure whilst volumes are completely administered by Docker. Also on the same source, named volumes are said to offer several advantages and are considered the better choice. Because of this, we decided to use named volumes for almost all of the system container's storage with two exceptions. First and foremost is the certificates directory used in Nifi's storage since they are unique to the host and need therefore to be re-created in every new environment. Secondly and arguably the biggest are the bind mounts used to store the Nifi-Registry data required to transfer Nifi's templates and settings between environments using Github.
As far as how the components contact each other, docker enables **networking** which allows a user to link containers to as many networks as required. In this case, all system services in the docker-compose file have the same network which means that all containers operate in the same network being able to contact each other.
Last but not least, we will provide an example of a **docker-compose** file where the Rule-Engine and the Database services are. As we can see in Figure 17, we instantiate

two services, where in the first one, the Rule-Engine, we specify the image, previously mentioned, and build on the directory where the YAML file is which allow us to create and build the Spring Java component. Additionally, we enable the port where the service runs in the network and create a dependency with the database so that they are connected. Sequentially, we also provide some environment variables which in this case are all related to the database and to the connection made using Hibernate and JPA. This service ends with a command that runs the spring boot server. In the database service, we can see it has a Named Volume by the name *rule-engine-db-data* which stores the data. Last but not least, while we could create the volumes automatically which would save us some trouble when changing environments, it also means that the volume's names would be prefixed by the directory of where they were created, which is why we can see that both the volume and the networks are defined as external allowing us to keep a simpler and more human-readable nomenclature to the expense of having them previously created.

```yaml
version: "3"
services:
  rule-engine:
    container_name: rule-engine
    image: mvn-builder
    build: .
    ports:
      - "8084:8084"
    depends_on:
      - rule-engine-db
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://rule-engine-db:5432/postgres
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=postgres
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
      - SPRING_JPA_SHOW_SQL=true
    command: "spring-boot:run"


  rule-engine-db:
    container_name: rule-engine-db
    image: 'postgres:13.2-alpine'
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    volumes:
      - rule-engine-db-data:/var/lib/postgresql/data

volumes:
  rule-engine-db-data:
    external: true

networks:
  default:
    external:
      name: network
```

Figure 17: Rule Engine and Database Docker-compose YAML file

5.1.3 How to change environments?

The following subsection intends to describe what needs to be done in order to change environments. First and foremost, the host device must have Docker, Docker-compose and to make things easier, Git installed. Following this, the user should clone the project using Git and run the createVolumes script which will create every Named Volume as well as the required network. Provided this, the user could already start the system, however, given that Nifi's files are not persisted in Github, the template would appear empty. To solve

this, the user will have to go through a couple of steps:

1. **Add Nifi-Registry -** First and foremost, the user must add a Registry Client using Nifi Registry URL in the Nifi Settings.
2. **Import Nifi Template -** The user should drag a Process Group into the home template and import a template from Nifi Registry which was previously added.
3. **Generating Certificates -** The user should create all the required certificates which in this case, relate to generating a PK12 SSL certificate so that one of Nifi's components can pull information from Here Technologies. To do this, a script is provided in the directory Nifi/certificates, which generates all the required files. It stands to reason that if the user wants to change the certificate's information, he should change the script accordingly;
4. **Enable Context Services -** The user must enable all the Context Services since they are automatically turned off when pulled from the Registry.

Given that NIfi is up and running, the user should be able to just run the system by executing the run.sh script provided in the main folder which not only runs the docker-compose but also executes the Flink Job in the Flink Server. It stands to reason that the Flink job is automatically compiled and the packaged Jar is shared with the Flink Server through a Docker Volume, however, in order for this Jar to be applied, it needs to be executed using a docker command.

5.2 Rule Mechanism

This subchapter goes over the Rule Mechanism while going over some of the important decisions that had to be made. The overall architecture of the Rule will be presented, followed by a description of the endpoints and lastly, a simple use case of the system flow will be outlined. First of all, it was decided that the most suitable technology for the rule mechanism was CEP, and for this, given that the Flink library couldn't inject rules dynamically during runtime, the chosen tool for this mechanism was the java-based Esper.

5.2.1 Architecture

For the Rule Mechanism, a microservice approach was adopted dividing the system into the different components that we can see in Figure 18. It stands to reason that every component except for the interface, are all Docker Containers. Before explaining each component, we will tackle a few points. First of all, the **division of the Rule Engine into a Rule Engine and a Rule API** is disputable since they could be merged and that would eliminate the messaging overhead, however, the main argument in favour is to separate the rule translation mechanism from the engine to create a more scalable and fool-proof system. Taking into consideration the scenario where the engine could have many rules and therefore need to scale by launching another engine, the rule translation mechanism might not need to be replicated and vice-versa. Additionally, in a faulty environment where a component crashes or the rules are just being incorrectly created, the Rule API can be taken down and worked on without having to take down the Rule Engine disabling only the ability to create more rules, allowing the existing rules to continue to evaluate the data and to trigger in case a certain threshold is surpassed.

As to the reasons behind **separating the rule alarm API and storing the alarms in a Kafka Topic**, it is arguable that this component didn't have to be separated from the engine which would completely extinguish the messaging overhead between components, however, on the one hand, it allows for a more scalable and fool-proof system that allows not only to horizontally scale the alarm mechanism but also enables the developers to effortlessly add more alarm services. Taking into account that the current alarm mechanism only triggers a system notification and sends an email to the previously added emails. Other notification services could be added like an automatic Slack or phone message and by storing the alarms in a Kafka topic, we can add more microservices that read from those topics completely separating the logic between different notification services and making sure that if some component is malfunctioning, we can easily pinpoint the problem and simultaneously be sure that t it does not impact the remaining notification services. On the other hand, the Kafka topic also provides the alarm API with a backpressure mechanism that allows it to better manage alarms when many rules are triggered and the queues fill up.

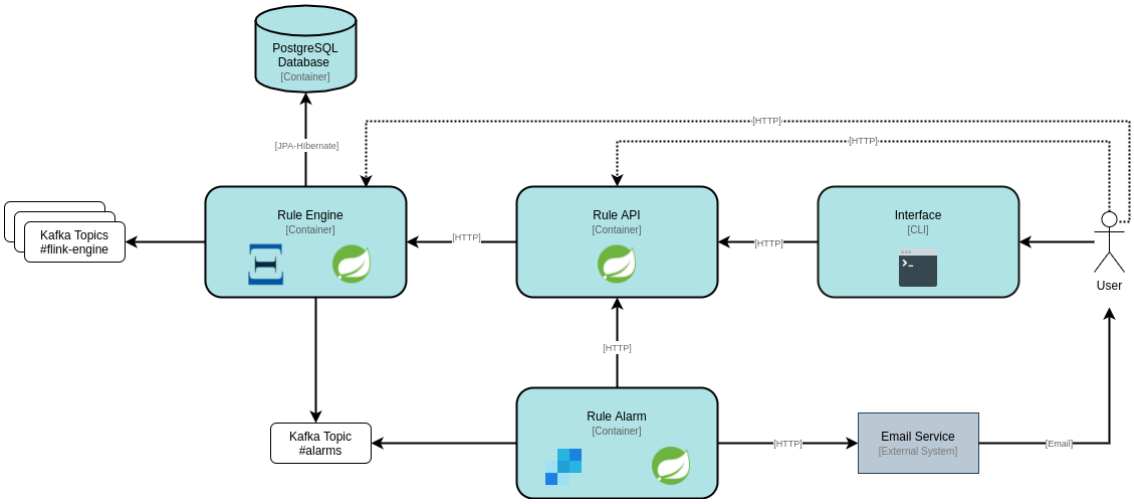


Figure 18: Rule Mechanism Architecture

To provide a deeper and more technical view of the architecture, we will now go over each component:

- **Kafka Topics -** These components are intended to queue the incoming data and the resulting alarms generated at the Rule Engine, to provide scalability and fault tolerance to the system. All information on a Kafka Topic is saved as a JSON message;
- **Rule Engine -** This component is the most complex in this architecture and it is where the Esper engine runs. It was developed in Java and mainly consists of three modules. The first one is made up of all engine-related logic, like reading data from a Kafka Topic and sending it to the engine, creating and adding a statement, adding a generic listener to the statement that when triggered sends an alarm to another Kafka topic whenever a certain condition threshold is surpassed and finding and deleting a statement. The second module is mainly responsible for handling all of the API calls and exposing the REST endpoints to receive, route and return data to the user. The third and last module is in charge of storing, querying and removing information from the database through the use of Hibernate, an object-relational mapping tool and Java Persistence API (JPA);

- **Rule API -** This component is responsible for bridging the requests that the user has to the Rule Engine whilst transforming the new rule requests into a meaningful and working EPL Statement. In other words, this is the API that a more beginner user should interact with either directly through the local swagger user interface or through the developed command-line interface, which in turn will make API Calls to the Rule-Engine. While most of the REST endpoints end up being the same as the Rule Engine, the main difference is the new rule endpoint which turns this API into the rule translation mechanism that makes use of the developed grammar in order to make the system simpler and more beginner-friendly;

- **Rule Alarm -** This component is responsible for retrieving alarm notifications from a Kafka Topic and triggering the corresponding notification services. The triggered rule has one out of three alarm levels, namely Informational, Warning and Critical which depending on the level triggers more notification services. In case it is an informational alarm, an API call is made to the Rule API to notify the user. If it's a higher level and the user has previously submitted emails, an email is sent to those submitted. The emails are sent through the use of SendGrid, an email delivery service that provides a Java library to effortlessly send up to 100 emails a day through their web API. There are no completely free email delivery services and while SendGrid isn't the one that offers the most monthly emails, it seemed to provide the easiest integration through a simple API. On a different subject, while the critical level is currently the same as the warning level, the main objective of having these different levels is that when more notification services are added, the alarm environments would change and tweak accordingly. Lastly, this component could arguably be divided into several microservices, one for each notification service, however, due to the time restrictions and since there were only two planned notification services, it was kept as a single one;

- **Interface -** This component is a command-line interface developed so that users that are not completely familiar with the available data models or that have little to no experience with Esper's EPL language can more easily create rules that evaluate the different data sets. The interface presents the user with multiple choices that the user must choose from in order to ultimately interact with the Rule API through its REST endpoints. This component will be more thoroughly analysed in the next implementation subchapter;

- **Database -** This component is responsible for storing all of the Rules as well as all the emails that should be notified if a certain rule is triggered. For this, We have chosen to use PostgreSQL for its reliability, and ease of development since the intern was already familiar with it.

Figures 19 and 20 below, show the lifecycle for an event that reaches the Rule Engine, triggers a rule and sends an alarm. In the former, The flow starts by sending an event to the Esper Engine after immediately consuming a JSON message from a Kafka Topic and converting it to the event type, for example, in this case, we will assume that the event is a traffic flow event. The flow ends with either sending one, two or no alarms to the Kafka Topic. As to the latter, the flow starts when a JSON message arrives and is immediately converted to an alarm. For this specific use case, we have decided to assume that two existing rules intend to trigger an alarm if: (1) *An average vehicle speed is above 50 km/h which will result in an informational alarm;* (2) *There are two congested events in a five minute period which will trigger a warning alarm.* Note that these rules are assumed to be already introduced in the system in an EPL format that is specific to the Esper Engine.

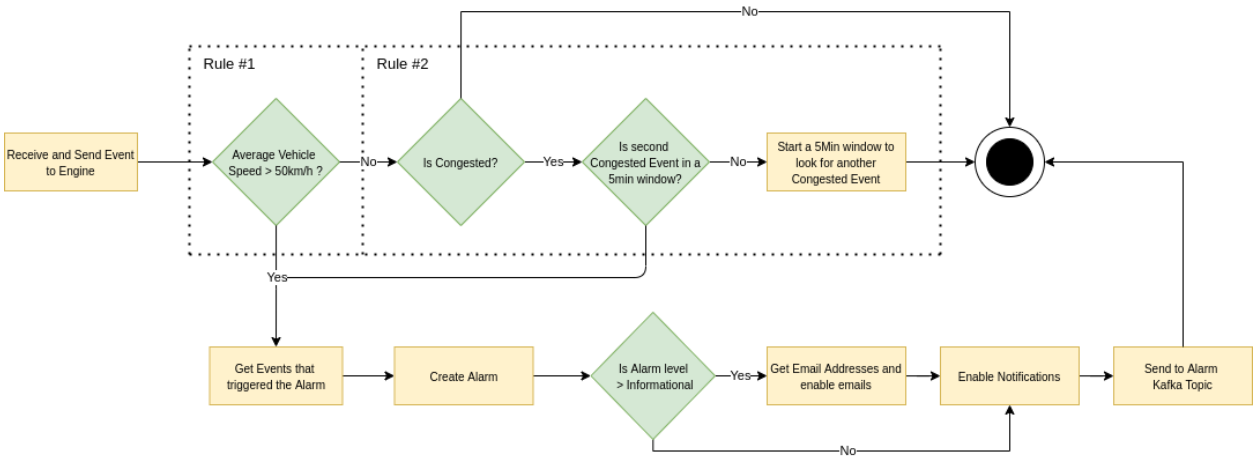


Figure 19: Rule Engine event lifecycle with two rules flow diagram

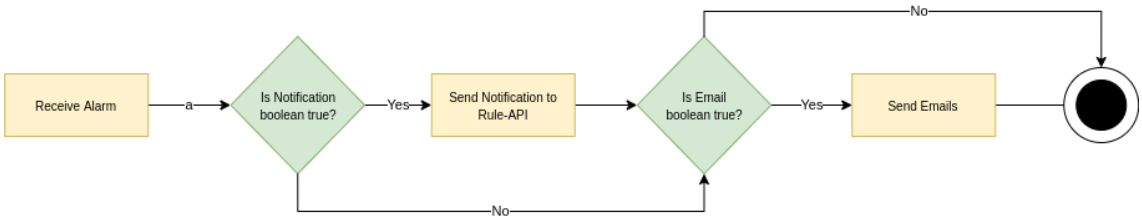


Figure 20: Alarm lifecycle flow diagram

5.2.2 API Endpoints

An endpoint is one end of a communication channel or in other words, an entry point usually available on REST APIs. This subchapter intends to demonstrate which endpoints are available on the Rule-API. It stands to notice that since the Rule-API is a bridge between the user and the Rule Engine, most endpoints are similar except for the add rule endpoint which is why both will be included to explain their difference. The list of endpoints is as follows:

- **POST /rule -** This endpoint is used to receive a new rule and transform the fields received into an adequate EPL statement which in turn will be sent to the Rule Engine. The endpoint receives a rather complex JSON that contains all fields of a rule and either responds with the successfully added rule or an error message;
- **GET /rule/ruleName -** This endpoint is used to retrieve a certain rule given that one exists with the provided name which is unique to each rule. Otherwise, an error message is returned;
- **DELETE /rule/ruleName -** This endpoint receives a name, and if a rule exists, it will be undeployed from the engine and deleted from the database. Despite the result, a message explaining the outcome is returned;
- **GET /rule/all -** This endpoint allows the application to retrieve all the existing rules. If there are no rules, an empty list will be returned;
- **DELETE /rule/all -** This endpoint ensures that all the existing rules are undeployed from the engine and deleted from the database. Despite the result, a message will be returned;

- **POST /rule/ruleName/email -** This endpoint is used to add an email to an existing rule. It receives a JSON with an email string as well as the rule name as an HTTP Parameter and if that rule exists and that email is not already associated, it is added;
- **GET /rule/ruleName/email -** This endpoint allows the application to retrieve all the existing emails of a provided rule name given that it exists. The endpoint will then respond with a list of all the existing emails;
- **DELETE /rule/ruleName/email -** This endpoint ensures that all the emails of an existing rule are deleted;
- **GET /rule/ruleName/email/batch -** This endpoint is used to register a batch of emails to an existing rule. It receives a JSON with a list of emails as well as the rule name as an HTTP Parameter and if that rule exists, each email will be added if not already assigned. The endpoint returns a list of the emails that were successfully added;
- **DELETE /rule/ruleName/email/emailId -** This endpoint receives an email Id in order to delete that email;
- **POST /notification -** This endpoint is used when a rule's threshold is surpassed and an alarm is triggered which in turn asynchronously triggers a notification to the Rule-API. This endpoint receives a message which should contain what rule was triggered;
- **GET /event/eventType -** This endpoint receives an event type as an HTTP parameter and if it is registered, using JJSchema which is a JSON schema generator, the endpoint returns a JSON with all the information about the event fields;
- **GET /event/all -** This endpoint has the purpose of returning all the event types registered in the Rule Engine so that the user has a way of knowing which events are available.

5.2.3 How to add a Rule?

This subchapter's objective is to go over the flow of a user adding a rule to the Esper Engine to give a better understanding of how the architecture works, how different types of users can interact with the system and what problems had to be tackled. First of all, we established two types of users that could want to interact with the system:

1. Users who have experience with Esper and know exactly what data sets exist and what their fields are, on the Engine.
2. Users who have little experience with Esper and may not know what data sets or fields exist on the Engine.

In order to provide a way for these two users to interact with the system, we decided to present two different entry points that the user can connect to. On the one hand, for the first type of users, we decided to use an OpenAPI specification tool, namely Swagger API which provides an interactive user interface that lists all the available endpoints whilst describing each one and allowing users to actively use the system in a less restrictive and more efficient way. It stands to notice that both the Rule Engine and the Rule API have

this tool, however, while the add rule endpoint in the Rule API receives a complex JSON that will construct the rule using the developed grammar, the Rule Engine only receives a string with the rule, and is, therefore, less restrictive and less protected. This ultimately also leads to having two different user input methods with different levels of freedom for the first type of users. On the other hand, for the less experienced users, since there was no time to develop a proper graphical user interface we decided to instead develop a command-line interface that intends to help users to create rules interactively. In other words, we prompt different options that the user can choose from, similar to how the Swagger API does for each endpoint, however, the big difference is on the add rule functionality in which, instead of giving full control to the user, we provide the user with the available options that he can choose from throughout the whole process. This interface and its developed grammar will be further explained in the next section of the Implementation chapter.

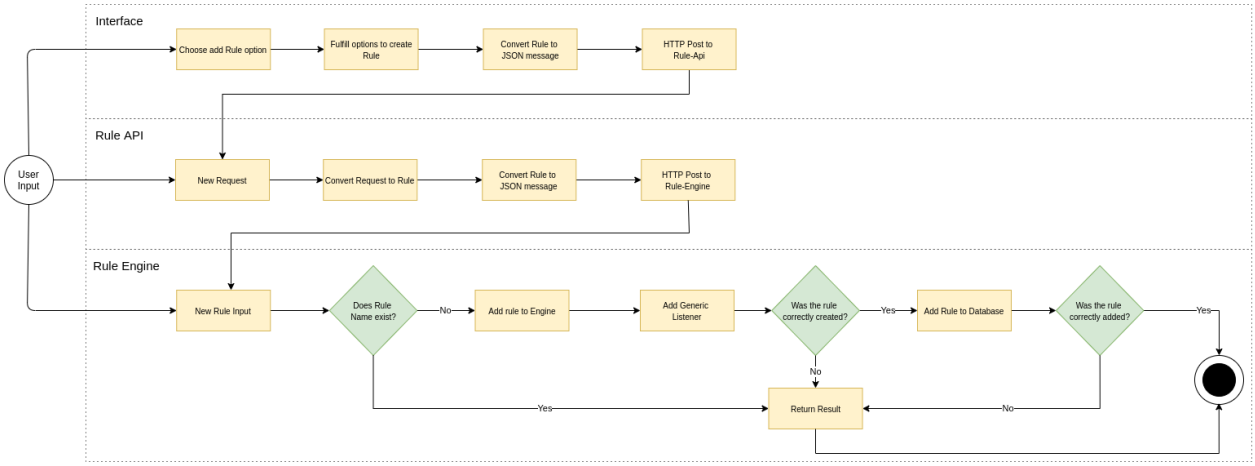


Figure 21: Add Rule flow diagram

Having explained the two entry points of our system, we will now describe the flow behind adding a new rule to the Engine, as depicted in Figure 21:

1. **User input -** As previously explained, can be directed to one out of three entry points. This input can either be through the choices made in the interface, through the restricted rule manually introduced into the Rule API endpoint or through the rule engine where there are no restrictions. The remaining steps of the flow will consider that the input choice was the interface to include all the steps;

2. **Interface -** After the user chooses the add rule option, he proceeds to incrementally construct the rule through the many options presented. After having finished, a JSON message will be constructed and an HTTP Post call will be made to the Rule API;

3. **Rule API -** After converting the JSON body into a request object, that same request will be transformed into a rule containing the name, the alarm level and a string, that is the rule itself, which in turn will be converted into a JSON message that will be sent through an HTTP Post call to the Rule Engine;

4. **Rule Engine -** First of all, the rule name existence will be checked to avoid repetition since it's a unique field. It stands to reason that if the interface was used, this verification was already performed when the user chose a name. Afterwards, the system is going to insert the rule in the engine and if there are no errors, the generic

listener will be attached to the statement. This generic listener, which main purpose was to remove the need to have one listener for each event type, started by being developed specifically for rules that were similar to SQL, however, after testing and moving towards the EPL statements for Patterns, this listener became unusable and a new one had to be developed. It stands to reason that this was not an easy task as Esper doesn't have a generic method and the solution to this problem went through a lot of trial and error debugging to understand how to get the events from the complex event beans objects. One note worth mentioning about this listener is that for the events not to appear empty, they must be anteceded by a tag name. Whilst in most scenarios, this method works and the events which triggered the rule are correctly returned, there may be some faults and it may not work one hundred per cent of the time as it would need more testing and tweaking. After adding the rule to the engine, if everything was successful, the rule will then be added to the database. In the end, the user either receives the rule or an error message as a response.

5.3 Grammar

The following subchapter goes over the developed grammar while describing some of the important decisions that had to be made. An overview of what a rule and a pattern are will be presented followed by an explanation of the grammar structure. Last but not least, a couple of example use cases will be provided along with their explanations to better demonstrate what the grammar is capable of. One detail worth bringing up is that Esper's event processing language, or EPL, is quite complex in the sense that there is a lot of freedom when it comes to developing a statement which means that little details can easily distinguish two rules apart. Because of this, the only way found to provide an inexperienced user with a helpful mechanism that allows them to more easily understand and create rules is to restrict a fraction of the vast EPL language and create a translation grammar. The developed grammar intends to include as many use cases as possible, however, it will always restrict certain rule scenarios. Additionally, for all the examples and use cases below, to make things easier, we will mention an Event that is strictly for demonstration purposes and could be thought of as being composed by only an id and a name.

5.3.1 Overview

A rule is no more than a set of conditions that analyse cause-and-effect relationships among events in real-time in the Esper Engine. In other words, a rule can be as simple as wanting to know if there are any events with a certain value above a predetermined threshold or they can become extremely complex depending on what the user wishes to analyse. Alternatively, an event can be thought of as a single type of data reading that arrived in the engine after being processed by the architecture (E.g a Traffic Flow event contains information of traffic in a certain location and time).

During the analysis and testing of Esper's language, we found it easier to divide the rules into three different levels. The first one could be described as the **basic rules** which include only looking for one event that could be restricted by one or more conditions. This type of rules are similar to SQL queries with some of the same syntax being used (E.g *select * from Event where id > 3 and name='A'*). These rules, while being simple, can quickly become complex through the addition of more conditions however they would always be referred to as a single event. In other words, basic rules don't have the ability to look for

different events that may or not be related. For this problem, Esper provides the second level of rules which we called **complex rules** that use the Event Pattern syntax which revolves around defining patterns that match when one or more events occur in a specific way according to the pattern's definition. A pattern must appear in the *From* clause of a statement and while it can be used together with other SQL syntax clauses, it also exponentially increases the complexity of the rules and the grammar and therefore the main focus of the grammar will be the Pattern syntax.

According to [146], a pattern expression is composed of two things, namely atoms and operators. The former is the base of the patterns and while they can be filter expressions, plug-in custom observers or time schedulers, to further simplify and remove what seemed unnecessary, we will assume that pattern atoms can only be filter expressions. These are nothing more than the previously mentioned basic rules with a slightly different, more event directed, syntax (E.g *Event(id>3 and name='A')*). As to the latter, these combine the atoms either logically or temporarily and in the developed grammar they include the Tags, Operators and Timers. Last but not least, the third set of rules, or **timed rules** are the result of restricting the complex rules temporarily through the addition of Timers. Since events are asynchronous and can arrive out of order, that means that we cannot create a rule that looks for two consecutive events but instead we can define a temporal window that restricts how long we wait for another event after finding the first one which ultimately creates the same effect.

5.3.2 Structure

As to details worth mentioning in the Grammar structure, as previously mentioned, we decided to discard the SQL clauses that could be added and built on top of the Pattern syntax to create even more complex rules and decided to focus on the main functionalities of patterns. Given this, all rules have the same basic building blocks, starting by having a name (*@name('string')*) tag which is unique to each rule, a description (*@description('string')*) tag which allows users to better describe rules so that they can later be understood and the basic SQL syntax that allows the use of patterns (*select * from pattern [rule]*).
As to the rules themselves, they are going to be composed of one or more atoms that can be logically connected through operators or temporarily through timers. It stands to notice that there are two scenarios where a rule can be temporarily restricted. First and foremost, the rule must have at least two atoms whereas in the second scenario, as we can see in the Atom clause of the grammar, a temporal restriction can be applied if a repeat operator is present which means that while it allows the sole atom to be present, there must be more than one event of the same type. We can think of this exception as being equivalent to having one atom followed by the same atom (A -> A) which ultimately reinforces the first scenario where two atoms must be present for a timer to be applicable.
Moving on to the structure itself, portrayed below, shows the whole grammar developed from a formal standpoint where the red words and symbols represent their literal values, the blue words represent the variables and the black words represent either dynamic variables or user input from a certain type. It stands to notice that the Property and Type clauses, which are portrayed in black, dynamically represent one data set model and their properties out of every available data model in the Rule Engine. In other words, the Type clause content will vary depending on what data models are registered in the engine while the Property variable options will also vary depending on the properties of the chosen data model. To provide an example, if the only data models available in the system are traffic

and environment, those will be the options for the Type clause whereas if we chose traffic, on the Property clause will be all traffic properties like average vehicle speed, location, and so on.
Additionally, the remaining black words represent user input from the specified data type. One last thing worth mentioning about the structure of the grammar is the **tag#count** which, as previously mentioned, exists so that the events can be later fetched when the rule is triggered to know which ones did it. While it is represented in red, only the **tag** component is always the same, since the count resets in every rule, the #**count** will always start at 1 and increment every time an atom is added. Despite changing, this is always pre-defined and always takes the same values and therefore, is not user-dependent which is why we portrayed it in red.

Base -> @name(‘String’) @description(‘String’) select * from pattern[Rule]
Rule –> Atom
Rule –> Atom Operator Rule
Rule –> (Rule Operator Rule Timer)
Atom –> (Tag AtomBody)
Atom –> ([Integer] AtomBody Timer)
Atom –> (every ([Integer] AtomBody Timer))
AtomBody –> tag#count = Type(Condition)
Condition –> InnerCondition ConditionList
ConditionList –> Logical InnerCondition ConditionList
ConditionList –>
InnerCondition –> (Property Relational Value)
Timer –> where time:within (Integer Time)
Time –> sec
Time –> min
Relational –> <
Relational –> >
Relational –> =
Relational –> !=
Relational –> <=
Relational –> =>
Logical –> and
Logical –> or
Operator –> and
Operator –> or
Operator –> ->
Tag –> not
Tag –> [Integer]
Tag –> every
Tag –>
Value –> Integer
Value –> String
Value –> Property
Value –> Boolean
Value –> Float

5.3.3 Examples

Having explained the grammar structure, we will now present a few grammar examples to better demonstrate how the grammar works and what it can do. To achieve that, we've divided the examples into two tables, where we will firstly showcase rules that are mostly conditioned to help show what the tags do and how the rules are built and secondly a group with the different timers to better exemplify how the grammar incorporates them into the rules.
Each table will have the grammar notification, an explanation, and the created Esper rule. To simplify things, the esper rule presented will be minified and therefore, discard the name, description and SQL syntax which is always the same, to make them more readable. Additionally, for demonstration purposes, the rules will be portrayed using a test Event that has two properties, namely id and name. When A and B are used in the grammar, they refer to an event with the name A or B in order to make it more readable.
For the tag 'every', we decided to include the examples on the Esper documentation used to differentiate the impact of the tag depending on how you apply it. It stands to reason that we only included three out of four examples since one of them cannot be achieved using the developed grammar. As to these examples, given the sequence of events (A1 B1 C1 B2 A2 D1 A3 B3 E1 A4 F1 B4), we will include the pair of events that match and trigger the pattern.

Grammar	**Explanation**	**Esper Rule**
event(name=A) and event(Id=3)	If an event with an Id =3 appears, then it will wait for another event with a name=A	[tag1=Event(id = 3) and tag2=Event(name = 'A')]
event(name=A and Id=3)	Looks for one event that has id=3 and name=A	[tag1=Event((id = 3) and (name='A'))]
(A) ->(B) and not (C)	Looks for an A, followed by a B with no C before or in between A and B.	[(tag1= Event(name='A') -> tag2=Event(name='B')) and not tag3=Event(name='C')]
[3] (A)	After finding an event with name=A it will look for two more events with the same name.	[[3] (tag1=Event((name='A')))]
(A) ->(B)	If an A is found, looks for a B (Only triggered the first A and first B occurrence). Matches on B2 {A1, B2}.	[(tag1=Event(name='A')) -> (tag2=Event(name='B'))]
every (A) ->(B)	For every A found, looks for next B. (Triggers once for every A) Matches on: B1 {A1, B1} B3 {A2,B3} {A3,B3} B4 {A4, B4}	[every (tag1=Event(name='A')) -> (tag2=Event(name='B'))]
(A) ->every (B)	For the first A found, looks for all next Bs. (Triggers once for every B) Matches on: B1 {A1, B1} B1 {A1, B2} B3 {A1, B3} B4 {A1, B4}	[every (tag1=Event(name='A')) -> every (tag2=Event(name='B'))]
every (A) ->every (B)	Pattern matches on every combination of A followed by B. Matches on: B1 {A1, B1} B1 {A1, B2} B3 {A1, B3}, {A2, B3}, {A3, B3} B4 {A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}	[every (tag1=Event(name='A')) -> every (tag2=Event(name='B'))]

Table 8: Grammar Examples

Grammar	Explanation	Esper Rule
(A) ->(B) Timer	The first time it catches an A, it waits 10 secs for a B. If a B never arrives during those 10 seconds, the rule is never triggered. (Can only happen once)	[(tag1=Event(name='A')) -> (tag2=Event(name='B')) where timer:within(10 sec)]
every (A) ->(B) Timer	Every time an A appears, it waits 10 seconds for a B to appear. If it doesn't, it just starts waiting for another A. (E.g A A B B, just triggers once)	[every (tag1=Event(name='A')) -> (tag2=Event(name='B')) where timer:within(10 sec)]
[3] (A) Timer (Can only happen once)	If a B is found, it waits 10 seconds for another 2 B's [[3] (tag1=Event(name='A')) where timer:within(10 sec)]	
every ([2] (A) Timer) ->(B) Timer	For each (AA) that occurs within 3 seconds it waits for another B. The whole (AAB) must occur within 10 seconds	[every ([2] (tag1=Event(name='A') where timer:within(3 sec)) -> (tag2=Event(name='B'))) where timer:within(10 sec)]
A ->((B ->C) Timer)	If an A is found, it waits 'forever' for a B. If a B is found, it waits 10 seconds until it finds a C.	[(tag1=Event(name='A')) -> ((tag2=Event(name='B')) -> (tag2=Event(name='C')) where timer:within(10 sec))]
every ((A ->B) Timer) ->C	For each A found, it waits 10 seconds for a B. If (A B) is found on time, then waits 'forever' for a C	[every (((tag1=Event(name='A')) -> (tag2=Event(name='B'))) where timer:within(10 sec)) -> (tag2=Event(name='C'))]
(A ->B ->C Timer)	Must find A followed by B followed by C within 10 seconds. (Can only happen once)	[((tag1 = Event((name = 'A'))) -> (tag2 = Event((name = 'B'))) -> (tag3 = Event((name = 'C'))) where timer:within(10 sec))]
every (A) ->every (B)	Pattern matches on every combination of A followed by B. Matches on: B1 {A1, B1} B1 {A1, B2} B3 {A1, B3}, {A2, B3}, {A3, B3} B4 {A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}	[[every (tag1=Event(name='A')) -> every (tag2=Event(name='B'))]

Table 9: Grammar Timer Examples

5.4 Interface

The following subchapter reviews the developed interface and the important decisions that had to be made. An overview of what is possible to do with the interface will be provided, followed by a deeper discussion about the Add Rule functionality since it is the most important aspect of the interface.

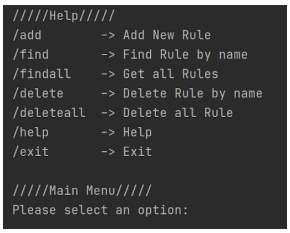
5.4.1 Overview and Functionalities

First of all, it stands to reason that since there wasn't enough time for the development of a graphical user interface, we opted to develop a command-line interface that intends to facilitate the use of the system to users that don't either have much experience in EPL or are unfamiliar with the data models present in the system. The main focus of the interface is to provide these users with a fool-proof, more restricted yet easier alternative to adding rules that makes use of developed Grammar previously explained whilst allowing them to make use of all the other functionalities of the Rule Engine. We will now go over all the interface's menus and functionalities except for the Add Rule which will be explained in the next subsection.
First and foremost, when the user starts the interface, he is presented with the Main Menu,

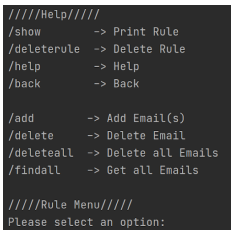
as we can see from figure 22, where he can choose a couple of options such as:

- /**add -** The user goes through a couple of options to create a new rule. This Will be further explained in subsequent subsections;
- /**find -** The user will be asked to input an alphanumeric word which is sent to the RuleAPI which verifies if there is a rule registered with that name. If the result is positive and therefore holds a rule, the user will be redirected to the Rule Menu where he will be presented with rule related options;
- /**findall -** If the user selects this option, he will be presented with all the registered rules in the Engine;
- /**delete -** Similarly to the find functionality, the user will be asked for an alphanumeric word that indicates which rule he wants to delete. After requesting the RuleAPI to delete that rule, he will be presented with the result, either informing him that the rule was deleted or not found;
- /**deleteall -** If the user selects this option, a request to delete all the registered rules will be issued to the RuleAPI. The user will be informed of the result;
- /**help -** This option allows the user to view the Main Menu options again by reprinting them on the console;
- /**exit -** This option allows the user to shut down the interface and exit the program.

It stands to reason that in all user input scenarios, he can opt to select the /**back** option which returns the user to the latest menu discarding all the information he had inputted.



(a) Interface Main Menu



(b) Interface Rule Menu

Figure 22: Interface Menus

Given that the user finds a rule by inputting a name in the /**find** option, he is then prompted with the rule he found and with the Rule Menu, which as depicted in Figure 22, allows the user to choose from the following options:

- /**show -** This option allows the user to view the rule and its details by reprinting them on the console;
- /**deleterule -** This option is a shortcut that allows the user to delete the rule without having to input the name again and if the user chose this option, he will be presented with the result and redirected back to the Main Menu;
- /**help -** This option allows the user to view the Rule Menu options again by reprinting them on the console;
- /**back -** This option allows the user to go back to the Main Menu.

If the rule searched for has an Alarm level of Warning or Critical, the user will also be presented with the email options related to the rule. Otherwise, these options will not appear.

- /**add -** The user will be asked to submit any number of email addresses which will be validated using the Email Validator class provided by the Apache Commons-validator library. Each input, if valid, adds an email address to a list which will be sent to the RuleAPI to be added when the user inputs the /**submit** option. Additionally, the user can at all times reset the list by using the /**clear** option.
- /**delete -** The user will be requested to input an email Id to delete an email from the rule. The user may use the /**findall** option to see the registered email addresses and their corresponding identification numbers.
- /**deleteall -** If the user selects this option, a request to delete all the emails registered to this rule will be issued to the RuleAPI. The user will be informed of the result;
- /**findall -** If the user selects this option, he will be presented with all the emails registered to this rule in the Engine.

5.4.2 Create Rule

The main purpose of this subsection is to review and analyse the Add Rule functionality in the Interface to give a better understanding of how the user interacts with it. This functionality has three levels, the first one being the base of the rule decisions, the second relates to the construction of one atomic condition and the third and last refer to building the inner basic conditions. We will now go over these three steps.
The user starts the process by choosing the /**add** option in the Main Menu where he will be redirected to the first input, choosing the rule name. As we can see in Figure 23, When the user inputs a name, a request will be made to the Rule-API to make sure that it is available. If it is, the user will then pick a description and an alarm level. From there, a request is made to the Rule-API to retrieve the event types that currently exist in the Engine (E.g TrafficFlow and Environment). After interpreting the response, the user will be asked to add an **Atomic Condition** which we will explain later. Having finished one condition, the user will be asked if he wants to add any more, and if he does, he will also be asked to select an operator that will connect the atomic conditions logically.

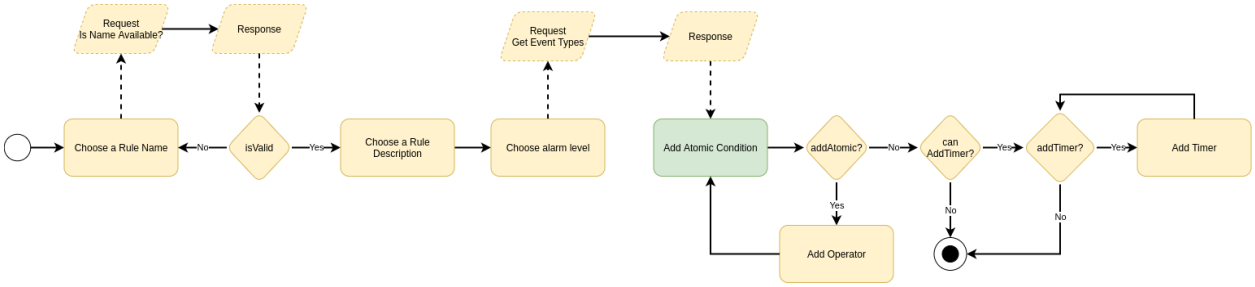


Figure 23: Create Rule Flow Diagram

After the user is finished, if he has added more than one condition, the rule is eligible to add a timer to restrict it and therefore, the user will be prompted to connect the rules temporarily. If the user chooses to do so, he will be presented with the chosen conditions and will be asked to select a start and an end position followed by a time window. These

start and end positions will dictate in which condition the timer starts and where it ends to dynamically append the timer to the rule to provide the user with the freedom of restricting any set of conditions. To further exemplify this, given the rule (A -> B -> C) the user may choose to add a timer to restrict A and B ((A->B Timer) -> C), B and C (A -> (B->C Timer)) or the whole rule ((A->B->C)Timer). It stands to reason, that while the user can add N-1 timers, being N the number of atomic conditions in the rule, the user cannot however add two timers that start and end in the same positions.
As to the amount of time to restrict the conditions, the user is provided with a few options such as (5 sec, 10sec, 1min, 10min) instead of being allowed to input any desirable window since long windows can allocate an enormous amount of resources and therefore decline the platforms processing capability. Last but not least, the rule will be converted to a JSON and be sent to the Rule-API to be added to the Engine.

As to adding an **Atomic Condition**, the user starts by choosing a tag and if the choice was a repeat ([Integer]), the user has to also provide the number of times he wants that condition to repeat. Additionally, given the same tag, the user is also presented with the option of adding a timer to that condition as previously explained. No matter what the user decides on, after the tag, the user will be presented with all the available event types in order to select one. Given the user choice, a request will be made to the Rule API to retrieve all of those event properties which in turn will be used to add Conditions which we will explain next. If the user wants to attach any additional number of conditions he will also have to choose logical operators to connect the conditions.

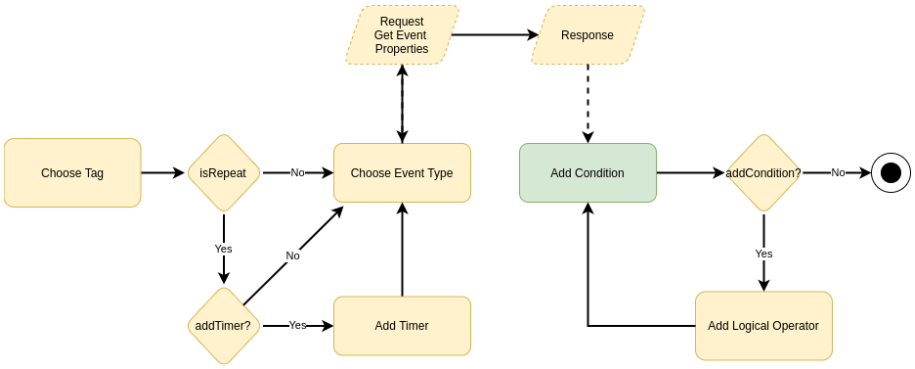


Figure 24: Atomic Condition Flow Diagramt

Last but not least, regarding the **Inner Conditions**, the user will be presented with all the available properties of the events, which were retrieved using JJSchema, previously configured, in which he has to choose one. The next choice is going to be the relational operator which is going to be different depending on the type of property. For example, if the selected property is a Boolean, we can only pick an equality or inequality operator whereas if it is an integer we can pick from a wide variety of operators. Lastly, the user will have the ability to select a value that can be one out of two things. First, it can be another event property that can be comparable, for example, if the user selected a property that is a boolean, he may compare it to another boolean property and secondly, the user may select an atomic value of the same type, for example, given the boolean choice, the user may compare it to true or false.

Figure 25: Inner Condition Flow Diagram

5.5 Data Pipeline

This subchapter reviews the developed pipeline while describing some of the important decisions that had to be made. An in-depth view of the Nifi template will be given, followed by a description of the developed Flink Job. It stands to reason that the following steps are related to the Traffic Flow data model and therefore are highly specific to the previously presented Use case scenario.

5.5.1 Nifi

To better understand how the ingestion layer works, Nifi provides basic building blocks called processors that perform some kind of operation on a flowfile which is the abstraction of a single piece of data. To simplify data flows and better organize them, Nifi supplies Templates which are aggregations of processors in order to design a larger block of data flow logic. It stands to reason that these templates should be as generic as possible and therefore include as many different data sources as possible to prepare for future deployments. Despite this, given the shortage of time and the sole source provided to the intern, the developed template is specifically designed for the Traffic Flow data model provided by Here Technologies API.
The developed template can be seen in Figure 26 and 27 below and starts with a GetHTTP processor which performs a GET request every 60 seconds to the predefined URL. It stands to reason that this request gets information from a certain bounding box. If the user wanted to retrieve information from a bigger perspective that contains more streets, he should enlarge the bounding box range and if he wanted to retrieve data from several different locations, he should replicate the processor using different URLs. Lastly, on this component, a certificate is needed since the connection is secure.

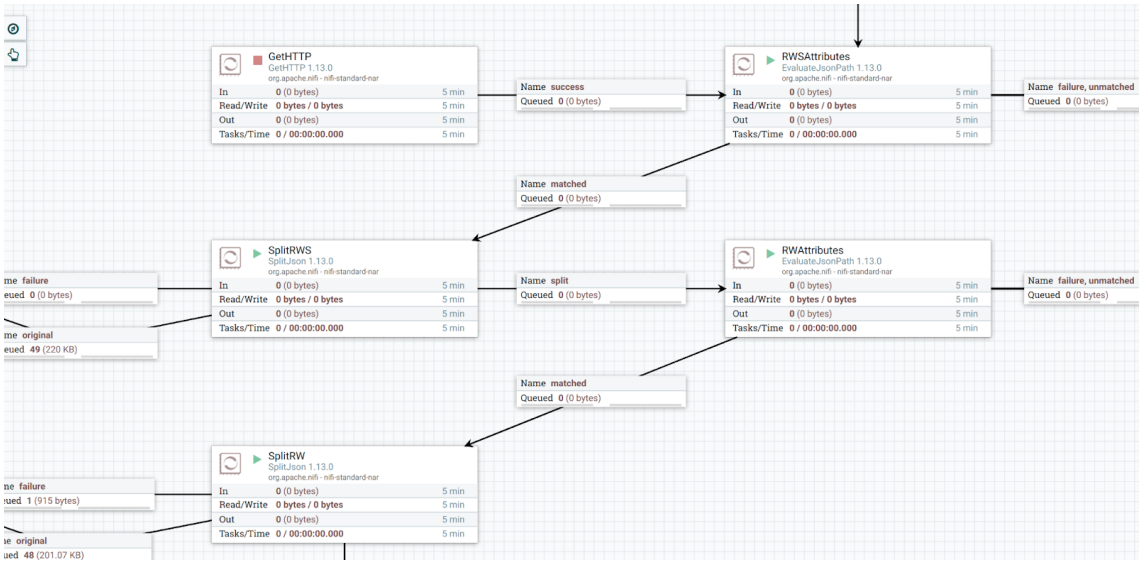


Figure 26: Nifi Template 1

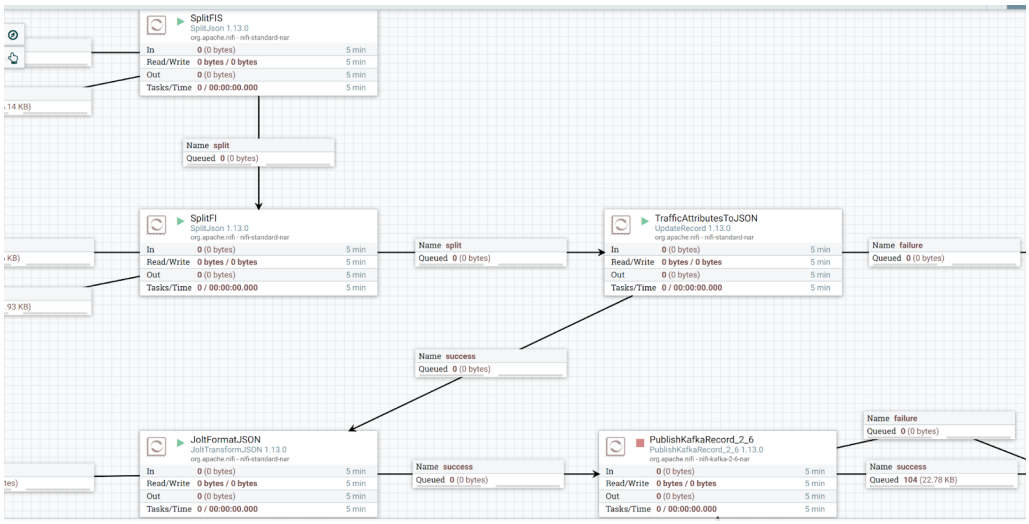


Figure 27: Nifi Template 2

As to the data model, Traffic Flow can be thought of as a hierarchical tree containing several ramifications where the leaves are the Flowfiles or Events that we need to analyse and therefore we need to go deeper inside the tree whilst splitting the JSON ramifications to obtain all the events. Additionally, given that there is important data that only the parent nodes have to reduce unnecessary duplication of data, whilst going through the tree there are some fields we need to store to later merge them with the resulting Flowfiles to create more informational final events. Examples of these attributes include the Created Timestamp and the Units fields.
We can see that we have EvaluateJsonPath processors (E.g RWSAttributes) that obtain specific JSON fields and create attributes that are connected to the Flowfile throughout the template. In addition to this, we have SplitJson Processors that split the JSON Flowfiles according to an expression that holds the key of the list (E.g SplitRWS has the expression $.RWS). By continuously splitting the JSON Flowfiles we end up having several Flow Items (FI) which are the basic events of the Traffic Flow request, that will then be merged with the attributes previously stored in the TrafficAttributesToJSON processor. Last but not least, given that the FIs end up with an unnecessarily complicated JSON, we decided to make use of a JoltTransformJSON processor, as depicted in Figure 28, to simplify and better identify the Events to make them easier to convert to POJOs in Flink.

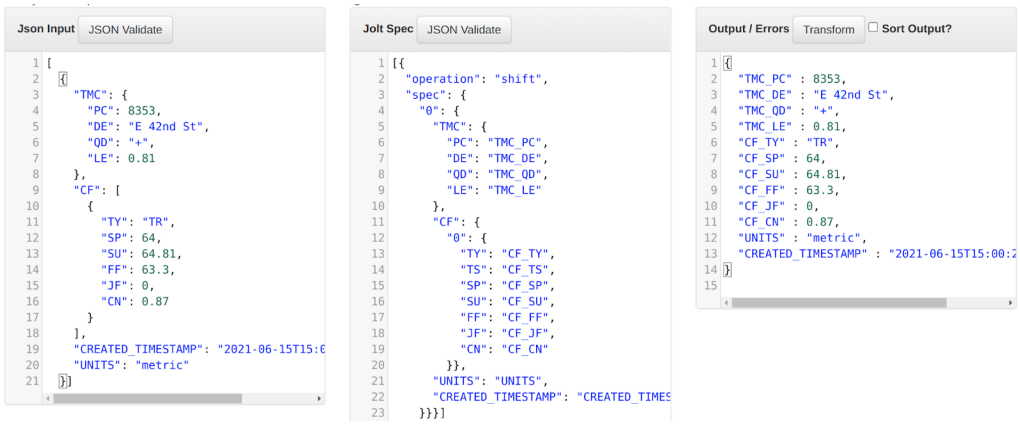


Figure 28: Jolt Transformation Example

In the example provided, there is one detail worth going over which is the absence of the SHP location field. In reality, it is present in the JSON on the left and was removed for demonstration purposes, however, given that it is not referenced in the Jolt specification, it also doesn't appear in the final event. The reason behind not including the location field is that the final Traffic Flow event that results from Flink after being converted into the NGSI-LD Standard, must have the Location field in a GeoJson format instead of Shapefile which presented itself as a problem since the only method that the intern found was by converting an external shapefile file into a GeoJson file using a library called GeoTool. Given the remaining time for the deadline and the complexity behind analysing not only how these formats work but also how the library operates to implement this feature, the intern decided not to carry out this task after discussing and validating it with the supervisors.
Subsequently, when the final events are ready, they will be sent to a Kafka topic using a PublishKafkaRecord processor which needs to be configured to identify the Kafka broker and topic. Additionally, this processor in addition to the previously mentioned TrafficAttributesToJSON, require a Record Reader and Writer which in these cases are JSON readers and writers that require a schema. It stands to reason that there are two basic options when it comes to providing a schema, the first one being to use the infer-schema option which automatically generates the schema based on the Flowfile and the second one being to use a manually defined schema. The advantages of the former are that you don't have to spend time analysing and creating a complex schema since it is automatic, however it is also heavier performance-wise. The chosen method, due to time-restrictions was to use the infer-schema option.
Last but not least, we can see that the data flow only advances when processors successfully match or split the data with the failures, unmatched and also the original Flowfiles being stored in queues so that they can be analysed. Arguably, given more time, all these should also be logged in a way that they could easily be detected.

5.5.2 Flink

When it comes to the main processing tool of our system, Flink offers numerous options to deal with data. For the Traffic Flow scenario, we decided to use the DataStream API which provides detailed control over state and time that allows for easier and better event-driven systems. The developed Flink Job or in other words, the Flink's Java program, can be divided into the following steps:

1. **Obtaining the stream execution environment -** Allows to execute the job in the previously started server cluster;
2. **Create the Kafka Consumer -** Allows to configure the Kafka properties that allow the data source to work;
3. **Add the Data Source -** Creates an input stream using the source with the previously configured Kafka Consumer;
4. **Convert the JSON messages to the Traffic Flow POJO -** Allows the input stream to be mapped into a stream containing Traffic Flow events using Gson;
5. **Convert Traffic Flow Event to Standard NGSI-LD Format -** This is the transformation process in our pipeline in which each field of the input event will be analysed and mapped or transformed, if necessary, into the correct output format. It stands to reason that Here Technologies traffic flow data set fields are mostly

provided in the correct formats which simplified this process. This step can arguably be the most complex depending on the necessary changes. As to what is done in this scenario, we first start by creating the DateObserved field since it is the timestamp for the beginning of the transformation. Following this, we add a unique UUID to the event and set the event type to 'TrafficFlowObserved'. Additionally, we set the reversed lane and the congested lane booleans according to the values in the input event. If these fields are not present, the booleans are set to false. Sequentially, we check for the existence of the lane direction and if it exists, we translate it into the correct format (E.g The character '+' is translated into the string 'forward'). In the next place, given that the date format is already in ISO-8601 (YYYY-MM-DDTHH:mm:ss. sssZ (UTC)), there was no need for any transformation. Lastly, as to the average speed, we check the field existence and if positive, we will either use the provided value given a metric unit system or we will convert the value from miles to kilometres. It should be noted that this transformation between events, due to the time restrictions imposed on the internship, was not as protected as it should be which means that one of the future steps would include improving the Flink's error tolerance and response;

6. **Create output stream -** Allows the new Traffic Flow standard event stream to be converted into a stream with JSON messages using Gson;

7. **Create the Kafka Producer -** Allows to configure the Kafka properties that allow the data sink to work;

8. **Add the Data Sink -** Adds the configured sink using the Kafka Producer to the output stream previously created in step 6;

9. **Executing the environment -** Executes the defined job with the previously designed steps.

5.6 How to add another data set

The following subchapter aims to underline and describe the changes that must be made to the system to add a new data set. We will go over each system component to provide a deeper and clearer explanation of what needs to be included.

Nifi
First of all, when it comes to data ingestion since the added data sets can be completely independent of one another, these must be treated accordingly. Therefore, to include a new data set, the developers must first understand and clarify whether the data being introduced in the system is new or not. If a data set of the same type already exists, there is the possibility of them being processed similarly and even if there are differences, there might be the opportunity to re-use some of the logic. In spite of this, the developers must also inspect to see if the information must be split into several events or if the data being sent into Nifi is already split and ready to be consumed by Flink. This means that for every data set of a different type, the comprehension, analysis and development of new data ingestion components must be completely renewed.

Flink
Similarly, when it comes to the data processing that is done in Flink, the **input models** that Nifi generates, will most certainly be different. The same might happen with the **output models**, if we introduce a new data set of traffic flow into the system since we

have already created the output standard model, it can be reused for the new data set, however, if the data sets contain completely different properties, then we need to add a new output model. As to the transformation stage, when a new data set is added, we need to create a new **Map Function** that transforms the input data into the output data. While some components can be reused, most likely, the whole transformation process will have to be designed. Despite this, the abstract process of adding a new data set in Flink is always the same:

1. Create a new Kafka Consumer;
2. Create the input stream that reads data from the consumer;
3. Convert the input stream into a stream of the input model using Gson;
4. Transform the input data into the correct output data format using a Map Function;
5. Convert the output data into a JSON data stream using Gson;
6. Create a new Kafka Producer;
7. Sink the information to the Kafka topic.

Rule Engine
As to what needs to be added in the Rule Engine, in comparison to Nifi and Flink, most changes are simple and straightforward. First of all, the developer must, only if needed, add a new **POJO model** with the addition of identifying each property using JJSchema attributes so that they can be later retrieved when the user wants to know which properties are in that data set.
Additionally, the user must include a new **Kafka Consumer** using the @KafkaListener tag, which retrieves events from a previously created Kafka topic. In this Kafka consumer, the user must convert the JSON string into the correct format, namely the previously created POJO model. To better organize things, the user should add a **method in the format service interface** that receives a string and returns an instance of the model using Gson. Alternatively, the user must add a new **Send Event method** that receives the new model and sends it to the Engine. To simplify, this method consists of only one line of code that needs to receive the event and sends it to the engine.
For this method to work, we also need to include the new **Event type to the Engine Configuration**, which in the end, means adding a line of code to the addEventTypes method that fills the engine configuration with the available event types.
Last but not least, the developer should also include the **event type in the getEventTypes method and the getEventProperties method**. The former returns a list with the names of the event types in the engine so that the user can see which are available and the latter returns the event properties specified in the model using JJSchema.

Kafka Topics
Last but not least, given that the Kafka Topics are the queues used to store data between Nifi, Flink and the Rule Engine and given that we want a queue for each different data set to allow a better division and overall a more organized system ready to scale, we need to create two new topics which will store data between Nifi and Flink and between Flink external systems like the Urban Platform or the Rule Engine. As to how we create the new Kafka topics we can either use the docker exec command presented below, as the container is running or we can add it by changing the docker-compose file before running the service.

```
docker exec kafka /opt/kafka_2.11-0.10.1.0/bin/kafka-topics.sh
```

```
--create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic [TOPIC]
```

Rule API and Interface

It stands to reason that in these components, since they get event types and their properties from the Rule Engine, they don't directly have access to the data models and therefore don't require any changes when data sets are added.

Chapter 6

Testing

The following chapter presents the formal tests performed on the system and is mainly divided into three sections. First and foremost we will go over the setup, the monitoring tools used and the metrics and then will move on to the test scenarios and the results regarding Load Testing and Failure Injection.

6.1 Setup and Environment

In order to test the developed system, the intern used his personal desktop which has the following **specifications**:

- **CPU -** Intel® Core™ i5-4460 CPU @ 3.20GHz × 4
- **GPU -** NVIDIA GeForce GTX 970
- **RAM -** 8gb
- **Disk -** HDD 1TB

First of all, it is important to note that the intern's testing environment was nowhere near as recommended due to the number of heavy performance-wise tools running as Docker containers like Nifi and Flink, and also due to the desktop's lack of performance. It is visible during the tests that the computer was already in a stressful state just by running most of the architecture, with the Memory usage being above 65%. These conditions led the tests to quickly reach their practical limit. To conclude, the intern was not provided with a suitable testing environment and did not have one at his disposal.

As to how **monitoring** was conducted, the intern looked into the tools Prometheus and Grafana to easily collect information and design a proper analytical dashboard. **Prometheus** is an open-source monitoring software that collects metrics, or small descriptions of an event so that they can be analysed. The Prometheus server actively pulls metrics from applications it monitors, through an exposed HTTP endpoint that makes the metrics available, at a certain defined interval. Last but not least, Prometheus provides the user with a functional query language called PromQL (Prometheus Query Language) that enables the selection and aggregation of time series data in real-time. As to the other tool, **Grafana** is an open-source visualization and analytics software that allows the user to add Prometheus as a source and, therefore, easily add visual and analytical capabilities to

the metrics captured by Prometheus. In addition to these tools, a Prometheus exporter, a software component that can collect statistics from another non-Prometheus system, was used to collect hardware and OS metrics, namely **Node-Exporter**.

The following Figure 29 describes the **testing environment** which is composed of the base system and most of the Rule Mechanism components. It stands to reason that the Interface and the Nifi Registry were not part of the following setup mostly because they weren't used by any component and therefore, there was no need to further stress a system that was already holding more performance requiring tools than it should. As to the data types, due to the problems that arose during the second semester, we were only able to include Traffic Flow and therefore, all the tests will reflect the impact of this data type. It stands to reason that while the system is ready to add other data types, there was no time to fully analyse and include all the processing and ingestion steps of another complex data set.
As to the data flow, the data will be simulated at Nifi and sunk into a Kafka topic that Flink will consume, transform into the correct format and in turn, output the events into another topic. The events will, last but not least, be sent to the rule engine where they will be cross-referenced against two previously created rules that if triggered will send an alarm notification to a Kafka topic that the Rule alarm microservice will consume and not only send a notification to the Rule-API but also an email to the intern. As to the topics, there is a Kafka container that manages and orchestrates them. Last but not least, Prometheus will receive metrics from the Node Exporter, regarding hardware and OS, and from Flink which will, in turn, be analysed using PromQL queries that will be visualized using Grafana.

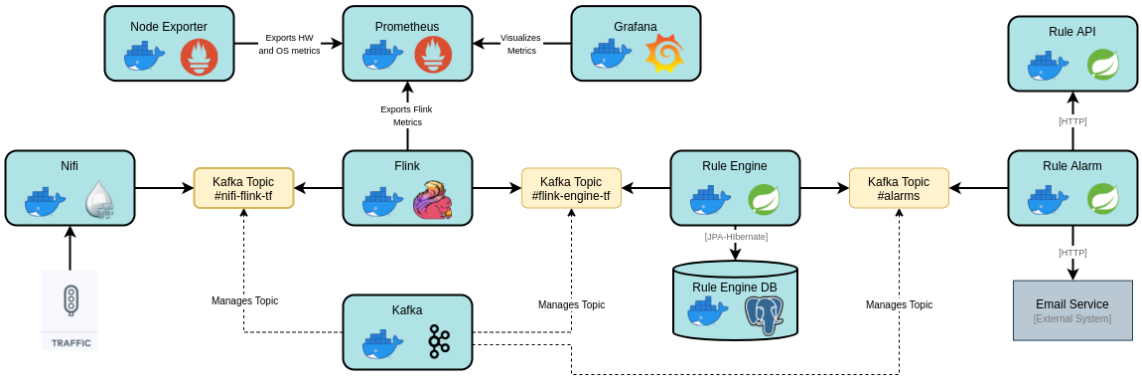


Figure 29: Testing Environment

As to how **data was simulated**, we decided to make use of a Nifi component to generate the events that were ingested into our system. As previously mentioned, there was no time to include more data sets, and therefore, the tests will consist of Traffic Flow events. The component *generateFlowFile* receives a String parameter that replicates into each and every Flowfile and also includes a configurable parameter that alters the rate at which messages are generated. Additionally, by replicating a real-life message from Here Technologies right into the start of the designed Nifi template we were able to provide a more realistic scenario while at the same time including Nifi in the testing scenarios.

Last but not least, as to the analysed **metrics**, we decided to use Node-Exporter to obtain hardware information such as CPU and Memory percentage usage. We also used Flink manual metrics to export the number of events being processed, which we used to display the system throughput and latency which depicts the duration that the event was in Nifi and Flink. Additionally, we also used Flink's default metrics to evaluate the number

of Kafka records being consumed and to check whether Flink had enabled backpressure mechanisms.

6.2 Load Testing

In the following subsection, we will present the configurations for the load test we performed whilst going over the test acceptance criteria and results.

6.2.1 Overview

First and foremost, as to how the system was tested, we decided to divide the load tests into two scenarios:

- **Stress Scenario -** The simulated events are incrementally simulated to better understand the maximal load that the system can handle and how the system recovers;
- **Spike Scenario -** The simulated events are released in sudden large bursts or spikes to analyse how the system handles these situations.

Alternatively, we decided to include the Rule Mechanism with two previously defined rules in order to provide a better whole system testing scenario. This allowed us to simultaneously display that these rules were triggered and that the email service was working since an email was previously added to the first rule. We added the following rules:

- **Rule #1 -** When the third Traffic Flow event with an average speed superior to 50 km/h arrives at the Esper engine, the rule is triggered and an email is sent to the intern;
- **Rule #2 -** When a Traffic Flow congested event arrives after an event with an average speed superior to 50 km/h in a 5-minute window, the rule is triggered.

6.2.2 Stress Scenario

In order to incrementally test the system, we decided to set a certain frequency to the Nifi component and more or less every 5 minutes, we would stop the component, quickly change the parameter to a higher frequency and would just turn the component back on. This means that before every change of throughput, there is a slight interruption. Additionally, in Table 10, we display the frequencies used as well as the number of requests and events that would generate and lastly the timestamps of when we changed the frequencies.

Frequency in seconds	Number of Requests	Number of Events	Time of Increase
1	60	240	20:05
0.1	600	2400	20:10
0.02	3 000	12 000	20:15
0.01	6 000	24 000	20:20
0.005	12 000	48 000	20:25
0.0025	24 000	96 000	20:30

Table 10: Stress loads

Figure 30: Stress Grafana Dashboard 1

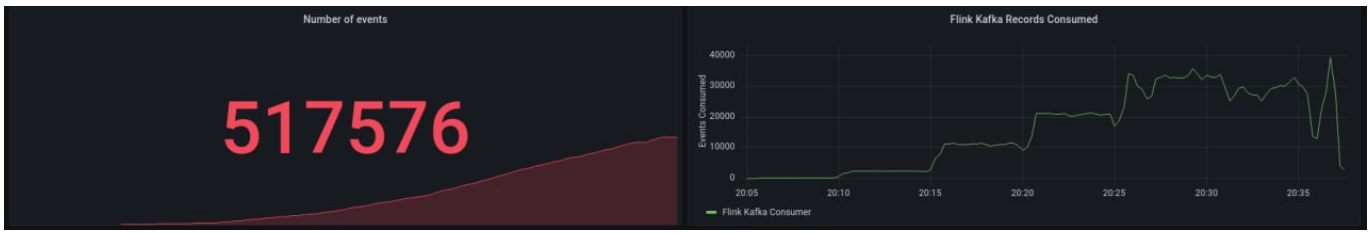


Figure 31: Stress Grafana Dashboard 2

Analysis

First and foremost, we can see that the system's CPU increases gradually with the throughput and surpasses 80% usage at around 28k to 30k events which indicates that the system starts to reach its limit at this range.

Alternatively, with regards to throughput, we can see that until 20:25 where the number of events per minute increased to 48k, the system easily scaled, showing little to no fluctuations. However, from that point, while Nifi didn't queue any event as they were all dispatched into Kafka, Flink's throughput variations fluctuated between 26k and 33k which led us to speculate two possible scenarios. On the one hand, it was possible that with the system reaching its practical limit having little to no free resources left, Flink could have activated backpressure mechanisms which would lead to its inability to cope with all the data being generated. To further back this up, we can see from the Flink Kafka consumer rate graphic that while there appeared to be 48k events being sunk into the Kafka Topic per minute, only around a third of those records were being consumed by Flink. On the other hand, when we looked at the latency values, while they showed an increase, if more than 10k events were being queued per minute due to Flink's backpressure mechanism, there would be some significant and easily noticeable increase in latency and not just 30 milliseconds. This led us to perform some tests on Nifi where we found out that despite being configured to produce 48k events per minute, it was not keeping up with demand. We can also notice this reduction in production in the 24k events per minute where Flink only consumes around 21k with no latency decay. These tests mainly consisted of changing the rate at which information should be generated while analysing the actual amount of data being outputted, using or two similar sources to see if the problem derived from using increasingly smaller units of time. After reaching the conclusion that the lower units of time had nothing to do with the problem, we concluded that this was most likely due to the computer having little to no resources left but it could have something to do with

some misconfigured setting that the intern may not have noticed. To further back this up, after looking through Flink's metrics being exported to Prometheus, we found one that indicated when the TaskManager activated the backpressure mechanism and there was no sign of it ever happening which led us to discard the first possible scenario.
When looking at latency, we can see a spike to the 100s milliseconds which was after we increased the output rate to 96k events per minute which eventually froze the computer that only came back to life after 20:35 where the data being generated decreased.
Moving on to the Memory usage graphic, we can see that it quickly reaches values above 85% at only 2.4k events per minute and reaches its peak of 90% at 20k events. After that, it appears to be practically stable.
We can conclude that with the setup previously mentioned, the system stably handles around 21k events per minute, or 350 events per second. After this, while the system, especially Flink, appears to easily handle more events, the computer has practically reached its limit and Nifi starts having trouble maintaining a stable generation of data.
Last but not least, we can see in the Figure **??** below, that Rule1 was triggered and that the system sent an email to the intern.

Figure 32: Email Alarm Notification

6.2.3 Spike Scenario

In order to test the system with bursts of data, we decided to set a certain limit on the queue between the component that generates data and the beginning of Nifi's template, let it fill up and then let the queue evacuate the data into the system. When the system finishes processing all the events, we would redefine the limit and repeat the process. In the following Table 11, we display the number of requests generated as well as the number of events.

Number of Requests	Number of Events
1000	4 000
2000	8 000
3000	12 000
4000	16 000
5000	20 000
7000	28 000
10000	40 000
15000	60 000

Table 11: Spike loads

Figure 33: Spike Grafana Dashboard 1

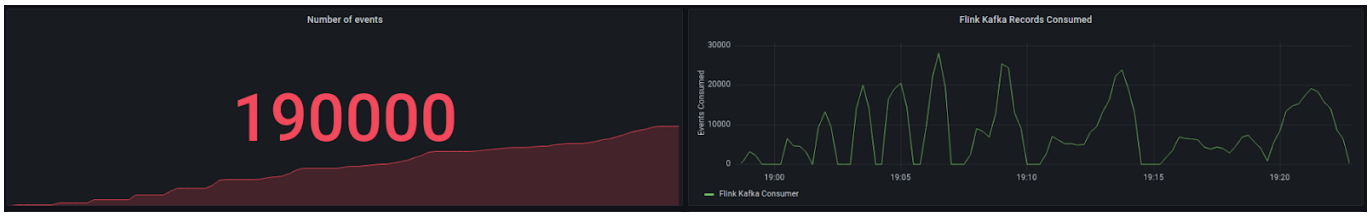


Figure 34: Spike Grafana Dashboard 2

Analysis

First and foremost, we can see that when it comes to CPU usage, the system easily reaches above 80% usage with a 28k burst of events which is in accordance with the previous stress test, however, by further increasing the number of events, the main difference in this metric is how long the system takes to process all the events.

As to Memory usage, we can notice that there is a stable increase of around 25% while the bursts of data get bigger.

Alternatively, we can see that after the 6th spike of events, the throughput begins to show a less steep ascension with the appearance of smaller increases. At first glance, we could argue that Flink's backpressure had something to do with this, however, after looking at Nifi during the more substantial 40k burst of events, we can see on Figure **??**, that a specific queue was backing up the pipeline.

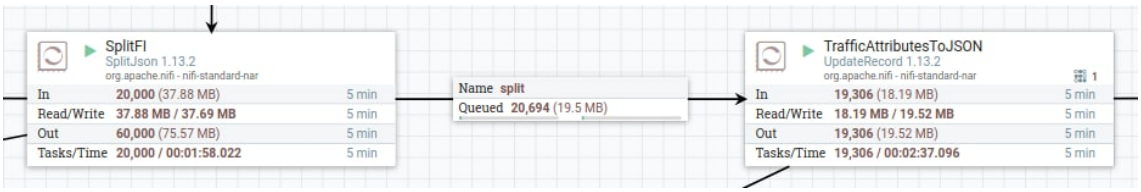


Figure 35: Nifi Queue

After further analysis, it was quite logically concluded that the automatic schema creation for transforming the events into JSON was not a good performance-wise option despite making things easier. The solution for this bottleneck would be to create manual schemas that did not have to be automatically generated for each and every event.

Moving on to latency, we decided to generate the simulated events with a timestamp which would be later used, after transforming them in Flink, to obtain the values that reflect the life of an event inside Nifi and Flink together. This method, in conjunction with how

bursts of events were injected into the system, led to the appearance of a human error that contemplates stopping the data generator and initiating the template. Alternatively, we are able to see that the latency values more than doubled after the 6th burst of data when Nifi started backing up the previously mentioned queue.
Despite this, we can see that with the current configurations, the system stably handles up to 20k events in a burst environment with the ability to reach even better results if the aforementioned problem is dealt with.

6.3 Failure Injection

To perform failure tests on the system, we decided to focus on three injection points, namely Nifi, Flink and Rule Engine since they are the core of our system. In addition to this, we decided to perform two scenarios of failure injection tests, malformed JSON inputs which can include missing fields, unknown fields or even malformed fields and lastly wrong data types. In the following table 12, we will present the failure tests performed, in what tool they were performed and lastly the result. To better display which tests failed, they will be displayed in red while the tests that pass will be green. Last but not least we will present an overview of the tests that fail.
It stands to reason that we will be injecting malformed events based on the Traffic Flow data set.

Analysis
From the performed tests, we can see that **Flink** is currently the least protected system component with the most vulnerabilities. As to the specific results, it is evident that Flink's input Json has to be in accordance with the input model as well as the data types or it will most likely crash. In addition to this, if Flink receives an empty event with just two empty fields, he will create an output event that contains null values which opens up possible future problems. It is without a doubt that Flink is the component that requires the most attention to create a more protected system.
Moving on to **Nifi**, we can see that it is mostly protected against malformed Json and just requires some tweaking on what to do if the event fields are empty. Nevertheless, the developed Nifi template does not currently have any protection against malformed data types which poses a problem. The solution for this would be to include an intermediary component that would verify the data type of each possible field to validate the event.
As to the **Rule Engine**, since the development time of this component was not affected by the problems that arose during the initial of the second semester, we can see that it is the most secured component of all passing all of the Failure Injection tests performed.
Last but not least, it should be noted that losing about three weeks in refactoring the architecture, as was previously mentioned, mainly impacted Flink's and Nifi's development. These weeks were essential to further make the components more fool-proof and also to add more data sets.

ID	Failure Scenario	Injection Point	Result
FS#1	Json containing unknown fields during a split	Nifi	Given that Nifi splits the events based on a field name and that field is unknown, Nifi sets the event aside to a Failure Queue not allowing it to continue in the template.
FS#2	Json containing unknown fields as a leaf (after the events are all split)	Nifi	Given that in the Jolt component Nifi reformats the input event into a specific format, all unknown fields are simply ignored and don't appear in the final event.
FS#3	Json containing no known leaf fields and no Units and Created Timestamps fields	Nifi	Nifi ignores every unknown field and the event is created with Units and Created timestamp being empty due to how they are added to the event.
FS#4	Malformed data types (String where Integer should be)	Nifi	Nifi doesn't detect the malformed types and normally deals with the events.
FS#5	Json with only 2 empty fields. Namely Unit and Created Timestamp.	Flink	Flink transforms the event with default values and processes it normally. It stands to reason that the final event ends up having two null values (dateCreated and laneDirection)
FS#6	Json containing a Json array where a field should be	Flink	Flink crashes - Json Syntax Exception
FS#7	Malformed data types (E.g String where Integer should be)	Flink	Flink crashes - Number Format Exception
FS#8	Adding 2 rules with the same name	Rule Engine	The Rule Engine catches the error and returns that the rule already exists
FS#9	Adding a wrongly constructed Rule	Rule Engine	The Rule Engine catches the error and returns a warning message
FS#10	Adding a rule with malformed data types (E.g String where Integer should be)	Rule Engine	The Rule Engine catches the error and returns a warning message
FS#11	Adding a rule using a Malformed Json with missing fields	Rule Engine	The Rule Engine catches the error and returns a warning message
FS#12	Adding a rule with the correct Json but with more unknown fields	Rule Engine	The Rule Engine ignores the added fields and correctly adds the rule
FS#13	Attempt to add, delete or find an email from a non-existing rule	Rule Engine	The Rule Engine catches the error and returns that the rule was not found
FS#14	Attempt to add a malformed email to an existing rule	Rule Engine	The Rule Engine catches the error and returns a warning message
FS#15	Attempt to find or delete a rule that doesn't exist	Rule Engine	The Rule Engine catches the error and returns that the rule was not found

Table 12: Failure Injection tests

Chapter 7

Planning and Methodology

In this chapter, the methodology adopted throughout the internship is described as well as all the planning performed, the established success criteria, the used tools and the risk management performed in order to mitigate possible problems.

7.1 Success Criteria

The main goals of this project are to develop a Master's thesis report and to develop the proposed proof of concept system in order to solve the challenge presented by Ubiwhere to the Intern. As to the Threshold of Success (ToS), in order to have a successful internship, a group of success criteria were defined:

- The project's proposed architecture and the requirements must be reviewed and approved by Ubiwhere;
- The developed architecture and project must ensure all the quality attributes requirements defined in this document;
- The proposed requirements, classified as "Must Have" in accordance with the MoSCoW methodology, must be successfully implemented;
- The internship and all the previously defined goals must be completed within the time planned.

7.2 Process Management

Agile has become one of the most popular and most commonly used project management methodologies. One of the main principles behind Agile is that it is human-centred, meaning that continuous delivery with periodical meetings between teams and the client proposes a more open environment where requirements are open to change even in late development. Additionally, Agile methodologies are iterative, which means that they divide the workload between several time periods called sprints, preferably shorter ones, where working software is frequently analysed, changed, improved and delivered promoting more disciplined management that has recurrent inspections and adaptations that better encourage teamwork [147], [148].
Throughout the duration of this internship, an adaptation of Scrum, which is an Agile

framework, was used. In Scrum [149], software development starts with the analysis of the product backlog which is a list of features that are wanted in the project and how long they are estimated to take. Additionally, Scrum relies on the Agile concept of Sprint by dividing the workload between periodical deliveries that may take from one week to two months to complete an item of the product backlog. In each sprint, the team has a meeting to the extent of forecasting the work needed in order to fulfil the backlog item which is then divided into smaller tasks to be developed, tested and integrated. During the sprint, daily meetings, which are brief and descriptive, are advised to make sure that every team member is aware of the project state and also to analyse possible changes and improvements. At the end of the sprint, the current state of the product is presented to the clients so as to promote a better product that matches the client's needs and to prioritize the most important requirements. Sequentially, the team meets and discusses the sprint's result in order to plan for the future sprint, repeating the cycle mentioned above.
Provided that the development is the responsibility of solely the intern, there is no team, and therefore, the framework proposed must adapt to this specific case. First and foremost, the clients, in this case are the intern supervisors and should therefore be present on the sprint planning and retrospective meeting. Additionally, it stands to notice that while daily meetings are not needed, communication is still crucial and therefore the intern should maintain open channels of communication with the supervisors so that the proposed work does not get delayed.

7.3 Tools

Through the duration of the Internship, the following tools were and are expected to be used:

- **Clockify -** Clockify is a time tracking tool used by the company in order to track the hours spent by each employee on all tasks;
- **BambooHR -** Bamboo is a human resources management software as a service used by the employees to track time, request time-off and perform self and hetero-assessments;
- **Slack -** Slack is an instant messaging platform used as the primary communication tool inside Ubiwhere that allows for a better, more organized reporting through the use of not only direct messaging, but also groups and chat threads;
- **Google Meets -** Meets is a video-communication service developed by Google and was the primary tool used for video conferencing between the intern and Ubiwhere;
- **Google Drive -** Google Drive is a file storage service developed by Google that was used to share files between Ubiwhere and the Intern;
- **Gitlab -** Gitlab is an open-source Git-repository manager used internally in Ubiwhere with plenty of features. Some of these features include Continuous Integration/Continuous Deployment (CI/CD) pipelines, issue-tracking and Wiki;
- **Docker -** Docker is a set of open-source Platform-as-a-Service products that simplify the process of creating, deploying and running applications through the use of containers. These containers allow bundling the application software with the addition of the needed libraries and other dependencies into one package ensuring that the application will execute on any given machine.

7.4 Planning

In the following section, the planning for the first and second semester is explained and the planned Gantt for the internship is displayed.

7.4.1 First Semester

During the first semester of the internship, the following stages were expected:

- **Problem Context and Motivation -** The first stage in the internship was defining and narrowing down the scope of the project in order to better understand Ubiwhere's problem and how we would tackle it. This is available in chapter 1;
- **Research -** This task included the continuous research on the several concepts such as Smart Cities, IoT, Big Data, Data ingestion and Data processing as well as the analysis of the tools and techniques available on the market. This can be translated into chapter 2;
- **Requirements Specification -** Given the problem statement, the next step was gathering, analysing and specifying the scope, stakeholders, constraints and lastly the requirements. Subsequently, the defined requirements were discussed and reviewed by Ubiwhere. These requirements can be found in chapter 3;
- **Propose Architecture -** For this task, having gathered the requirements and having researched the topics, the next step was coming up with an architecture that fulfils the previously stated requirements whilst analysing the available technologies. This is available in chapter 4;
- **Planning and Methodologies -** For this phase, the process and tools used were described. Additionally, planning and risk assessment were presented. This can be seen in chapter 5;
- **Intermediate Report -** The intermediate report, which was done in parallel to the other phases, aims to present all the work done during the first semester and is expected to include all of the above topics which went through several iterations after receiving feedback from both advisors.

7.4.2 Second Semester

During the second semester of the internship, the following stages were expected:

- **Training & Setup -** Before starting the development phase, the intern had to become accustomed to the technologies chosen since it is crucial for the subsequent tasks. Additionally, this is the phase where the intern configures the projects environment using Docker;
- **Base System Development -** This is one of the two main phases of the semester since this is where all the previous research, planning and preparation came to use. This task consists of the development of the Ingestion layer, Processing layer and Message Queue which ingest data into the Urban Platform by making it available on a Kafka Topic;

- **Rule Mechanism Development -** This is the other main phase of the semester. This task consists of the development of the Rule Mechanism which allows the users to submit custom rules with custom thresholds and alarms;

- **Testing Base System -** This task was performed during the base system development phase, inside each sprint;

- **Testing Rule Mechanism -** This task was performed during the rule mechanism development phase, inside each sprint;

- **Testing the Whole System -** This task was performed after integrating the rule mechanism with the base system and focuses on making sure the system works as a whole accordingly and on finding possible problems;

- **Final Report -** The report was done parallelly to the rest of the tasks as it should address all the decisions made and all the challenges, solutions and thought-process behind them. Additionally, this report was revised incrementally by both Ubiwhere and the University supervisor.

7.4.3 Planning Overview

In the following subsection, a Gantt chart, namely Figure 36, is displayed in order to provide a better understanding of the project planning over the duration of the intern.

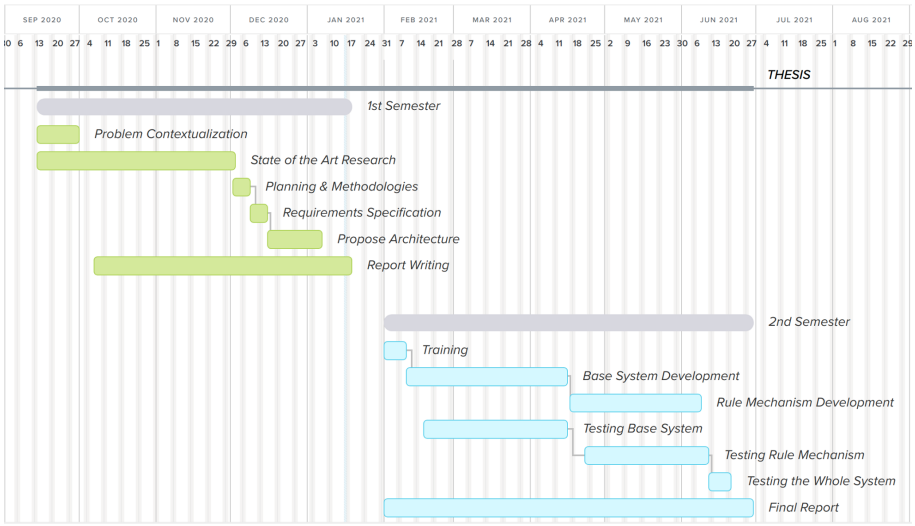


Figure 36: Planned Gantt

7.4.4 Planning Analysis

In the following subchapter, a second Gantt diagram of exclusively the second semester, namely Figure 37, is presented to show the real duration of each task in comparison with the expected diagram presented in the first semester.

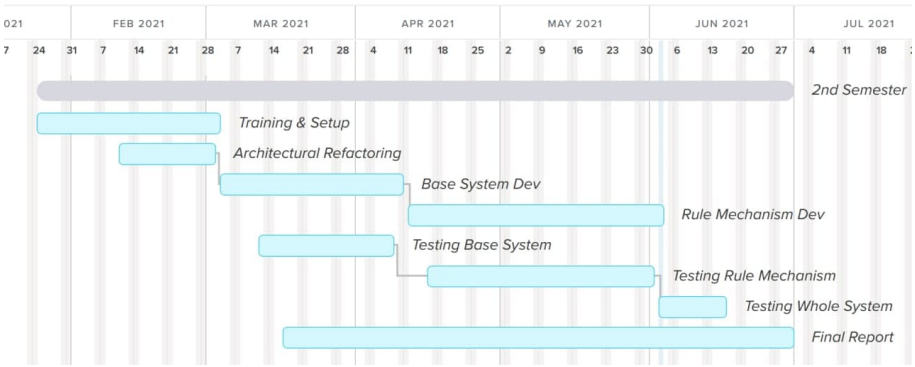


Figure 37: Second Semester Gantt

When comparing them, we can immediately see that there is a newly added task called Architecture Refactoring which is directly related to the problem that arose. During the first phase of the second semester, development ended up deviating from the planning due to two setbacks. On the one hand, while going through Nussknacker's documentation and installation tutorial, the intern discovered that not only did the tool only have one source and one sink components but also to develop new ones, it had to be done in Scala with close to no documentation or examples. This led to analysing Nifi's components and ultimately changing what tool to use in the Ingestion layer. Additionally, during the same process with Flink, after experimenting with FlinkCEP library, it came to our attention that there was no mechanism that allowed injecting rules during runtime which in the end led to discarding FlinkCEP for Esper. This setback was longer than expected since in addition to all the time spent experimenting with FlinkCEP, there was also the time spent in researching solutions, restructuring the architecture and learning the new tool. Ultimately, not only did this problem impact the training duration but these issues also set the intern back around three weeks, that while didn't compromise the project by delaying the delivery, still withdrew development time from other tasks.

Alternatively, we can see that training was renamed to training and setup and took around four times the planned duration. As to the previous planning, It is arguable that the planned time was naive and optimistic especially since there were many tools that the intern was unfamiliar with which ultimately led to a longer training period. One of these tools, Docker, which the intern had to learn to configure the many technologies, is the main reason behind why the task was renamed. It stands to reason, that this task was the one that took the longest during training, since setting up the environment and making it not only effortlessly easier to setup and run but also lighter, since the intern's computer was nowhere near as good as it should performance-wise, was a harder challenge than anticipated.

In the end, both these changes led to not having as much time to make the system more fool-proof and to include more datasets.

7.5 Risk Management

Risk management is a crucial and continuous task that allows for project managers to minimize the potential problems that may arise throughout the life cycle of the project by identifying, evaluating and creating mitigation approaches to certain possible situations that may compromise the process and possibly lead to a project failure. This activity is therefore essential due to the fact that if we plan for possible problems, we can preemptively

act and evade catastrophic outcomes that affect the thresholds of success. The first step is risk identification with the subsequent activity being the risk impact and prioritization analysis followed by the creation of a mitigation plan so that the risk may be continuously monitored so as to minimize the damages it may cause in case it activates.
In order to classify and better understand the risk, the impact and the probability of the risk were presented. While the impact of a risk is connected to the damage it would create if it was not dealt with, the probability of an identified risk is related to the odds of a risk happening. Both scales were identified, in Table 13, as:

Rating	**Impact**	**Probability**
High	The success criteria is not achievable	The risk is likely to happen.
Medium	The success criteria can be achieved. Requires great effort and cost	The risk might happen
Low	The success criteria can be achieved. Doesn't require much effort and cost	It is unlikely that the risk happens

Table 13: Rating Impact and Probability

7.5.1 Risk Analysis

Each of the following tables identifies and analyses a risk, identified by an Id of the format **R#Number**, while categorizing it in accordance with the previously established scales.

Id	**R#1**
Condition	The intern has little to no experience in the technologies and tools chosen
Consequence	The development of the Proof of Concept system may take longer than expected
Impact	Low
Probability	High

Table 14: Risk - R#1

Id	**R#2**
Condition	Given the current pandemic state we are living in, the intern may become infected with the disease
Consequence	The development of the system may have to come to a stop for the duration of the symptoms
Impact	Medium
Probability	High

Table 15: Risk - R#2

Id	**R#3**
Condition	Given the current pandemic state, we are living in, the intern's direct supervisors may become infected with the disease
Consequence	There will be a delay in the feedback and on the instructions needed to continue the development of the system.
Impact	Low
Probability	High

Table 16: Risk - R#3

Id	**R#4**
Condition	The internship's scope is not clear
Consequence	The complexity of the system increases, leading to changes in the Requirements and the Proposed Architecture
Impact	High
Probability	Low

Table 17: Risk - R#4

Id	**R#5**
Condition	Time spent on a task surpasses the expected.
Consequence	Next tasks will be delayed which will impact the time left for the deadline
Impact	Medium
Probability	Medium

Table 18: Risk - R#5

Id	**R#6**
Condition	The chosen tools have to be replaced
Consequence	The complexity of the system may increase, leading to changes in the Requirements and the Proposed Architecture and an overall development delay.
Impact	Medium
Probability	Medium

Table 19: Risk - R#6

7.5.2 Risk Exposure Matrix

In the previous subsection, the different risks were identified and in the sequence, they were labelled in accordance with their respective Probability and Impact on the Internship's success. Following that, in Figure 38, an exposure matrix where these two criteria are represented is shown. The main goal of this matrix is to graphically represent the severity of each risk that needs to be mitigated. It stands to notice that each square represents a rating previously stated as Low, Medium or High.

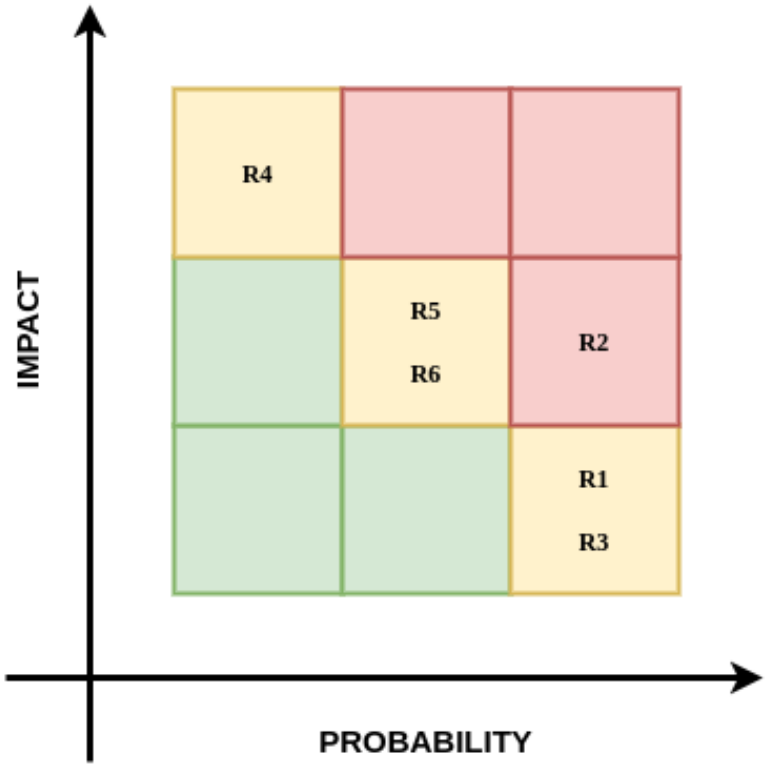


Figure 38: Risk Exposure Matrix

7.5.3 Risk Mitigation

Having, classified and analysed each risk, the remaining step is to create a mitigation plan for each one in order to not only minimize the probability of them happening but also to develop a way to diminish the consequences of them triggering during the internship.
From the previous matrix we can see only one of the risks is on the high-risk "critical zone" while the remaining are all on the medium-risk zone. The following measures were defined in order to mitigate each risk:

- **R#1**
  Allocate some time to learn the technologies and if needed, delay the start of the project's development while the intern is not comfortable.

- **R#2**
  After checking with Ubiwhere, the internship will mostly take place remotely to minimize the risk of infection.

- **R#3**
  To minimize this risk, Ubiwhere has several Supervisors which may help other interns if the need arises.

- **R#4**
  The interns must not only be explicit on the requirements and on the proposed architecture but must also validate these with Ubiwhere and with the University supervisor so that no doubts emerge in later stages.

- **R#5**
  The event should be reported to the supervisor in the weekly reviews to analyse the situation and re-evaluate the importance of the task while bearing in mind the remaining tasks so as to firstly focus on the most relevant and crucial tasks.

- **R#6**
  Assign some time to experiment with the technologies and if required, exchange the necessary tools before starting development. All changes must be evaluated with the Supervisors.

7.5.4 Risk Analysis

Risk **R#6**, which had not been formally analysed in the first semester, was realised leading to changes in not only the tools previously selected but also changes in the architecture in the Rule Mechanism. The first change was switching Nussnacker with Nifi due to problems identified during the testing phase where the intern experimented with the tool. The other, more significant change, was the removal of FlinkCEP as the CEP tool since it could not have dynamic runtime injection of rules. This ultimately led to restructuring the architecture, moving the Rule Mechanism away from the main pipeline and turning it into an external system that parallelly consumes the data with the Urban Platform. Despite not having predicted the risk in the first semester, its mitigation process was the same as the Risk **R#1** and therefore, while it was triggered, its consequences ended up being diminished. As to the exact consequences of triggering this risk, there was a loss of 3 weeks since all the work spent on researching FlinkCEP and Nussnacker was discarded and also because the architecture had to be remade and new tools had to be analysed, compared and learned.

This page is intentionally left blank.

Chapter 8

Conclusion

The following chapter presents the conclusion of not only the internship but also the work done throughout its duration. Additionally, some remarks will be made concerning future work to improve the proof of concept.
As to the **work done**, this document describes the work done by the intern during the internship at Ubiwhere. Taking into account the previously defined success criteria, even if there were some setbacks during the internship, we can consider the overall results a success since:

- Both the requirements and the architecture were reviewed and approved by Ubiwhere previous to starting development.
- All the quality attributes were detailed in the document, either met in chapter 4.Proposed Architecture or tested in chapter 6.Testing.
- Of the proposed functional requirements, only 2, which were not 'Must Have', were not developed:
  - **US#3 -** This Could Have requirement, implied filtering the rules and the main reason behind not implementing this requirement was a shortage of time.
  - **US#5 -** This Should Have requirement stated that users should be able to edit existing rules and it was not implemented because Esper does not allow to update running statements.
- Of all the previously defined goals, only 1 was not completed within the planned time. This goal was the constraint R02 where the intern had to implement 2 data sets. As previously discussed in chapter 7. Planning and Methodologies, the changes in the architecture led to not being able to include a second data set.

Moving on to the **challenges and lessons learned**, firstly, the intern had never been involved in a project of this magnitude both in size and complexity which ended up being a great learning experience due to not only all the work done but also all the problems that the intern had to overcome. On top of that, the intern had no experience in a company which proved to be quite an opportunity to improve his critical thinking and problem-solving capabilities while gaining some knowledge of how working in a company is like. Additionally, the development of this proposed system included quite a few setbacks. First of all, the proposed system was more complex than anticipated which involved adding more unexpected technologies while maintaining the time constraints ultimately resulting

in prioritizing some aspects of the system. Additionally, It stands to reason that the intern had close to no experience with most of the technologies chosen which resulted in a longer training period than expected. One other aspect that complicated the process was the project setup which revolved around learning to use Docker and Docker-Compose while at the same time having to prevail over the DevOps problems that arose. This was one of the areas where the intern had the least experience and also one of the most rewarding. Alternatively, one of the most noteworthy learning experiences throughout this internship was that software development is far from straightforward where problems appear and have to be solved which was clearly seen in the architectural refactoring stage during the second semester.

As to **future work**, even though the proof of concept system is nowhere near ready for production, there are still some notable improvements that could enhance the results which would help in the decision-making process of improving not only the Urban Platform but also any other system currently facing a data ingestion bottleneck. First and foremost, the current system uses the exact deployed solution that has to manage its resources ultimately leading to problems in horizontally scaling. To solve this problem, an orchestration system that handles deployment, scaling and management of the containers like Kubernetes could be used. However, due to the considerable increase in complexity that this would imply, this option had to be discarded. One other improvement worth noticing would be to improve Flink's resilience by protecting it against malformed inputs. In addition to this, Nifi could also use some improvements by not only making it more fool-proof but also switching from automatically generated schemas to previously created manual schemas to prevent a bottleneck in backed-up queues and ultimately boosting performance. Additionally, the developed Grammar can always be improved to broaden the spectre of rules it allows and it can also be further tested so as to more efficiently outline its capabilities and restrictions. Alternatively, a more adequate, user-friendly interface could be developed using some web framework to not only make the system more accessible to use by the more inexperienced users but also improve the overall usability of the developed grammar. To further improve the current data set, the transformation of the location field from shapefile to geoJson using GeoTools should also be added to the Traffic Flow events. Last but not least, more tests should be done using additional data sets in a better, more performant environment to provide more realistic testing scenarios and also better results.

References

[1] "Eu-smartcities.eu." https://eu-smartcities.eu/page/european-context, 2021. [Online; accessed -17-October-2020].

[2] H. Chourabi, T. Nam, S. Walker, J. R. Gil-Garcia, S. Mellouli, K. Nahon, T. A. Pardo, and H. J. Scholl, "Understanding smart cities: An integrative framework," in *2012 45th Hawaii International Conference on System Sciences*, pp. 2289–2297, 2012.

[3] F. Ridzuan and W. Zainon, "A review on data cleansing methods for big data," *Procedia Computer Science*, vol. 161, pp. 731–738, 01 2019.

[4] "Smartsantander.eu." https://www.smartsantander.eu/, 2021. [Online; accessed -12-November-2020].

[5] M. Bain, "Joinup.ec.europa.eu." https://joinup.ec.europa.eu/sites/default/files/document/2014-06/SENTILO%20case_joinup_v_1%202.pdf, 2021. [Online; accessed -17-October-2020].

[6] V. Moustaka, A. Vakali, and L. Anthopoulos, "A systematic review for smart city data analytics," *ACM Computing Surveys*, vol. 51, pp. 1–41, 12 2018.

[7] G. Piro, I. Cianci, L. Grieco, G. Boggia, and P. Camarda, "Information centric services in smart cities," *Journal of Systems and Software*, vol. 88, 01 2013.

[8] V. Albino, U. Berardi, and R. Dangelico, "Smart cities: Definitions, dimensions, performance, and initiatives," *Journal of Urban Technology*, vol. 22, pp. 3–21, 02 2015.

[9] V. Javidroozi, H. Shah, and G. Feldman, "Urban computing and smart cities: Towards changing city processes by applying enterprise systems integration practices," *IEEE Access*, vol. 7, pp. 108023–108034, 2019.

[10] "Knowledge4policy.ec.europa.eu." https://knowledge4policy.ec.europa.eu/foresight/topic/continuing-urbanisation/developments-and-forecasts-on-continuing-urbanisation_en, 2021. [Online; accessed -17-October-2020].

[11] E. Nuaimi, H. Neyadi, N. Mohamed, and J. Al-Jaroodi, "Applications of big data to smart cities," *Journal of Internet Services and Applications*, vol. 6, 08 2015.

[12] "Europarl.europa.eu." https://www.europarl.europa.eu/RegData/etudes/etudes/join/2014/507480/IPOL-ITRE_ET(2014)507480_EN.pdf, 2021. [Online; accessed -17-October-2020].

[13] A. City, "Home - amsterdam smart city." https://amsterdamsmartcity.com/?lang=en, 2021. [Online; accessed -12-November-2020].

[14] A. City, "Mobility amsterdam smart city." `https://mobility.here.com/learn/smart-city-initiatives/amsterdams-smart-city-ambitious-goals-collaborative-innovation`, 2021. [Online; accessed -12-November-2020].

[15] "Default." `https://www.smartnation.gov.sg/`, 2021. [Online; accessed -12-November-2020].

[16] "Mobility singapore smart city." `https://mobility.here.com/learn/smart-city-initiatives/singapore-smart-city-holistic-transformation`, 2021. [Online; accessed -12-November-2020].

[17] "Here." `https://mobility.here.com/learn/smart-city-initiatives/barcelona-smart-city-people-people`, 2021. [Online; accessed -12-November-2020].

[18] "Sentilo." `https://www.sentilo.io/wordpress/`, 2021. [Online; accessed -12-November-2020].

[19] A. Sinaeepourfard, J. Garcia Almiñana, X. Masip, E. Marin-Tordera, J. Cirera, G. Grau, and F. Casaus, "Estimating smart city sensors data generation current and future data in the city of barcelona," 06 2016.

[20] M. Milenkovic, *Internet of Things: Concepts and System Design*. 05 2020.

[21] K. Ashton, "Kevin ashton describes "the internet of things"." `https://www.smithsonianmag.com/innovation/kevin-ashton-describes-the-internet-of-things-180953749/`, 2015. [Online; accessed -14-November-2020].

[22] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[23] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, "Chapter 1 - introduction and book structure," in *From Machine-To-Machine to the Internet of Things* (J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, eds.), pp. 3 – 8, Oxford: Academic Press, 2014.

[24] I. Market, "Iot in smart cities market by solution service - 2025 | covid-19 impact analysis | marketsandmarkets." `https://www.marketsandmarkets.com/Market-Reports/iot-smart-cities-market-215714954.html#:~:text=What%20is%20IoT%20in%20smart,control%20between%20citizens%20and%20governments`, 2020. [Online; accessed -14-November-2020].

[25] M. Gupta and J. George, "Toward the development of a big data analytics capability," *Information Management*, vol. 53, 07 2016.

[26] A. Cuzzocrea, I.-Y. Song, and K. Davis, "Analytics over large-scale multidimensional data," pp. 101–104, 10 2011.

[27] P. Michalik, J. Štofa, and I. Zolotová, "Concept definition for big data architecture in the education system," in *2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pp. 331–334, 2014.

[28] S. Madden, "From databases to big data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 4–6, 2012.

[29] "gartner's big data definition consists of three parts, not to be confused with three "v"s." https://blogs.gartner.com/svetlana-sicular/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/, 2013. [Online; accessed -14-November-2020].

[30] B. Marr, "why only one of the 5 vs of big data really matters." https://www.ibmbigdatahub.com/blog/why-only-one-5-vs-big-data-really-matters, 2015. [Online; accessed -14-November-2020].

[31] C. Cartledge, "How many vs are there in big data?." http://clc-ent.com/TBDE/Docs/vs.pdf, 2019. [Online; accessed -14-November-2020].

[32] S. Fosso Wamba, S. Akter, A. Edwards, G. Chopin, and D. Gnanzou, "How 'big data' can make big impact: Findings from a systematic review and a longitudinal case study," *International Journal of Production Economics*, vol. 165, pp. 234 – 246, 2015.

[33] "What are the 4v's of big data." https://whataftercollege.com/big-data/what-are-the-4vs-of-big-data/, 2020. [Online; accessed -15-November-2020].

[34] https://planningtank.com/computer-applications/data-processing-cycle, 2020. [Online; accessed -16-November-2020].

[35] "What is data cleansing?." https://www.alooma.com/blog/what-is-data-cleansing, 2018. [Online; accessed -16-November-2020].

[36] L. Dictionaries, "definition of ingestion by oxford dictionary on lexico.com." https://www.lexico.com/definition/ingestion, 2021. [Online; accessed -16-November-2020].

[37] "Data ingestion for the connected world." http://cidrdb.org/cidr2017/papers/p124-meehan-cidr17.pdf. [Online; accessed -2-December-2020].

[38] N. Vollmer, "Article 4 eu general data protection regulation (eu-gdpr)." https://www.privacy-regulation.eu/en/article-4-definitions-GDPR.html, 2021. [Online; accessed -16-November-2020].

[39] "What constitutes data processing?." https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-constitutes-data-processing_en, 2021. [Online; accessed -16-November-2020].

[40] "What is data processing?." https://www.itpro.co.uk/business-operations/31681/what-is-data-processing, 2021. [Online; accessed -16-November-2020].

[41] "What is data processing?." https://www.stitchdata.com/resources/data-ingestion/#:~:text=Data%20ingestion%20is%20the%20transportation,database%2C%20or%20a%20document%20store, 2021. [Online; accessed -16-November-2020].

[42] "Gdpr regulation." https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679, 2021. [Online; accessed -16-November-2020].

[43] "Hipaa regulation." https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html, 2021. [Online; accessed -16-November-2020].

[44] "Batch vs stream processing." https://www.precisely.com/blog/big-data/big-data-101-batch-stream-processing, 2021. [Online; accessed -18-November-2020].

[45] "Micro-batch processing." https://hazelcast.com/glossary/micro-batch-processing/, 2021. [Online; accessed -18-November-2020].

[46] "Upsolver." https://www.upsolver.com/blog/batch-stream-a-cheat-sheet, 2019. [Online; accessed -16-November-2020].

[47] "What is real-time stream processing?." https://hazelcast.com/glossary/real-time-stream-processing/, 2020. [Online; accessed -18-November-2020].

[48] "Publisher subscriber." https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber, 2018. [Online; accessed -18-November-2020].

[49] M. Fowler, "Event sourcing." https://www.martinfowler.com/eaaDev/EventSourcing.html, 2005. [Online; accessed -18-November-2020].

[50] "Events and stream processing." https://www.oreilly.com/library/view/making-sense-of/9781492042563/ch01.html, 2020. [Online; accessed -18-November-2020].

[51] "An overview of etl and elt architecture." https://www.sqlshack.com/an-overview-of-etl-and-elt-architecture/, 2020. [Online; accessed -18-November-2020].

[52] "What do we do? - etl." https://blog.bismart.com/en/what-do-we-do-etl, 2020. [Online; accessed -18-November-2020].

[53] G. Alley, "Etl vs elt differences." https://www.alooma.com/blog/etl-vs-elt-differences, 2018. [Online; accessed -17-November-2020].

[54] "What is etl." https://www.alooma.com/blog/what-is-etl, 2018. [Online; accessed -18-November-2020].

[55] "What is elt." https://www.alooma.com/blog/what-is-elt, 2018. [Online; accessed -18-November-2020].

[56] "Big data infrastructure decisions: Data in motion vs. data at rest." https://virtualizationreview.com/articles/2014/11/05/data-in-motion-and-at-rest.aspx, 2014. [Online; accessed -17-November-2020].

[57] "Modern vs traditional etl: What's the difference?." https://www.matillion.com/resources/blog/modern-vs-traditional-etl-what-s-the-difference, 2020. [Online; accessed -17-November-2020].

[58] L. Qiao, Y. Li, S. Takiar, Z. Liu, N. Veeramreddy, M. Tu, Y. Dai, I. Buenrostro, K. Surlaker, S. Das, and C. Botev, *Gobblin: Unifying data ingestion for hadoop*, pp. 1764–1760. 01 2015.

[59] P. Merla and Y. Liang, "Data analysis using hadoop mapreduce environment," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 4783–4785, 2017.

[60] S. G. Manikandan and S. Ravi, "Big data analysis using apache hadoop," in *2014 International Conference on IT Convergence and Security (ICITCS)*, pp. 1–4, 2014.

[61] "Apache hadoop." https://hadoop.apache.org/, 2020. [Online; accessed -17-November-2020].

[62] "What is map reduce." https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm, 2020. [Online; accessed -17-November-2020].

[63] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, p. 107–113, Jan. 2008.

[64] "Hadoop examples." https://www.bmc.com/blogs/hadoop-examples/. [Online; accessed -17-November-2020].

[65] Vikash, L. Mishra, and S. Varma, "Performance evaluation of real-time stream processing systems for internet of things applications," *Future Generation Computer Systems*, vol. 113, pp. 207 – 217, 2020.

[66] S. Kamburugamuve, G. Fox, J. Qiu, and D. Leake, "Survey of distributed stream processing for large stream sources," 12 2014.

[67] "Ververica what is stream processing." https://www.ververica.com/what-is-stream-processing#:~:text=Stream%20processing%20is%20the%20processing,series%20of%20events%20over%20time, 2020. [Online; accessed -19-November-2020].

[68] M. A. Talhaoui, "Real-time data stream processing - challenges and perspectives," *International Journal of Computer Science Issues*, vol. 14, 08 2018.

[69] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 1–1, 10 2019.

[70] "What is stream processing?." https://hazelcast.com/glossary/stream-processing/, 2020. [Online; accessed -19-November-2020].

[71] J. Enes, R. R. Expósito, and J. Touriño, "Real-time resource scaling platform for big data workloads on serverless environments," *Future Generation Computer Systems*, vol. 105, pp. 361 – 379, 2020.

[72] "What is iaas." https://azure.microsoft.com/en-us/overview/what-is-iaas/#:~:text=Infrastructure%20as%20a%20service%20(IaaS,(PaaS)%2C%20and%20serverless, 2020. [Online; accessed -19-November-2020].

[73] "What is cloud computing." https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#cloud-deployment-types, 2020. [Online; accessed -19-November-2020].

[74] S. Trilles, A. González-Pérez, and J. Huerta, "An iot platform based on microservices and serverless paradigms for smart farming purposes," *Sensors*, vol. 20, p. 2418, Apr 2020.

[75] "Baas vs faas." https://www.hitechnectar.com/blogs/baas-vs-faas-explaining-the-two-serverless-architectures/#:~:text=So%2C%20Serverless%20Architectures%20are%20execution,(Function%20as%20a%20Service), 2020. [Online; accessed -20-November-2020].

[76] A. Christidis, S. Moschoyiannis, C. Hsu, and R. Davies, "Enabling serverless deployment of large-scale ai workloads," *IEEE Access*, vol. 8, pp. 70150–70161, 2020.

[77] "Serverless architecture." https://www.twilio.com/docs/glossary/what-is-serverless-architecture#:~:text=Serverless%20architecture%20(also%20known%20as,hardware%20management%20by%20the%20developer, 2020. [Online; accessed -20-November-2020].

[78] S. Kazimov, "Limitations of serverless computing," 06 2019.

[79] "Things to consider before building a serverless data warehouse." https://www.serverless.com/blog/things-consider-building-serverless-data-warehouse, 2019. [Online; accessed -20-November-2020].

[80] "Kinesis vs kafka." https://medium.com/flo-engineering/kinesis-vs-kafka-6709c968813#:~:text=Kafka%20beats%20Kinesis%20in%20all,So%20the%20winner%20is%20Kafka, 2020. [Online; accessed -20-November-2020].

[81] "Kinesis vs kafka comparison." https://www.softkraft.co/aws-kinesis-vs-kafka-comparison/, 2020. [Online; accessed -20-November-2020].

[82] D. Nguyen, A. Luckow, E. Duffy, K. Kennedy, and A. Apon, "Evaluation of highly available cloud streaming systems for performance and price," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 360–363, 2018.

[83] K. Singh, R. Behera, and J. Mantri, *Big Data Ecosystem: Review on Architectural Evolution: Proceedings of IEMIS 2018, Volume 2*, pp. 335–345. 01 2019.

[84] N. Marz, "Lambda architecture." http://lambda-architecture.net/, 2020. [Online; accessed -21-November-2020].

[85] Y. Kumar, *Lambda Architecture - Realtime Data Processing*. PhD thesis, 01 2020.

[86] "Hazelcast - lambda architecture." https://hazelcast.com/glossary/lambda-architecture/. [Online; accessed -21-November-2020].

[87] "What is lambda architecture?." https://hazelcast.com/glossary/lambda-architecture/, 2020. [Online; accessed -21-November-2020].

[88] "How to use change data capture (cdc) for database replication." https://www.flydata.com/blog/what-change-data-capture-cdc-is-and-why-its-important/, 2020. [Online; accessed -21-November-2020].

[89] "Cap theorem." https://www.ibm.com/cloud/learn/cap-theorem, 2020. [Online; accessed -21-November-2020].

[90] "Lambda architecture: Design simpler, resilient, maintainable and scalable big data solutions." https://www.infoq.com/articles/lambda-architecture-scalable-big-data-solutions/, 2014. [Online; accessed -21-November-2020].

[91] J. Kreps, "Questioning the lambda architecture." https://www.oreilly.com/radar/questioning-the-lambda-architecture/, 2014. [Online; accessed -21-November-2020].

[92] S. Ounacer, M. A. Talhaoui, S. Ardchir, A. Daif, and M. Azouazi, "A new architecture for real time data stream processing," *International Journal of Advanced Computer Science and Applications*, vol. 8, 01 2017.

[93] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, "Real-time stream processing for big data," *it - Information Technology*, vol. 58, 01 2016.

[94] "What is the kappa architecture?." `https://hazelcast.com/glossary/kappa-architecture/`, 2020. [Online; accessed -21-November-2020].

[95] "Evolution of big data architecture." `https://en.paradigmadigital.com/dev/from-lambda-to-kappa-evolution-of-big-data-architectures/`, 2018. [Online; accessed -21-November-2020].

[96] "Apache nifi." `https://nifi.apache.org/`, 2020. [Online; accessed -22-November-2020].

[97] "Apache nifi tutorial: What is, architecture installation." `https://www.guru99.com/apache-nifi-tutorial.html`, 2020. [Online; accessed -22-November-2020].

[98] "Nifi developer's guide." `https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#components`, 2020. [Online; accessed -22-November-2020].

[99] "Flow-based programming." `https://jpaulm.github.io/fbp/`, 2020. [Online; accessed -22-November-2020].

[100] "Open source etl: Apache nifi vs streamsets." `https://cube.dev/blog/open-source-etl/`, 2018. [Online; accessed -22-November-2020].

[101] "What really grinds my gears: Apache nifi." `https://touk.pl/blog/2018/07/19/what-really-grinds-my-gears-apache-nifi/`. [Online; accessed -22-November-2020].

[102] "Streamsets." `https://streamsets.com/`, 2020. [Online; accessed -23-November-2020].

[103] "What is streamsets?." `https://www.alooma.com/answers/what-is-streamsets`, 2017. [Online; accessed -23-November-2020].

[104] "What is data drift?." `https://streamsets.com/why-dataops/what-is-data-drift/`, 2020. [Online; accessed -23-November-2020].

[105] "Nussknacker." `https://nussknacker.io/`, 2020. [Online; accessed -23-November-2020].

[106] "Touk nussknacker – using apache flink made easier for analysts and business." `https://touk.pl/blog/2017/09/04/touk-nussknacker-using-apache-flink-made-easier-for-analysts-and-business/`, 2020. [Online; accessed -23-November-2020].

[107] "Nussknacker.io." `https://nussknacker.io/`. [Online; accessed -23-November-2020].

[108] "Apache storm." `https://storm.apache.org/`, 2020. [Online; accessed -23-November-2020].

[109] "Ad-hoc stream query processing." `https://d-nb.info/1204995648/34`, 2020. [Online; accessed -24-November-2020].

[110] "An overview of etl and elt architecture." `https://storm.apache.org/releases/current/Concepts.html`, 2020. [Online; accessed -24-November-2020].

[111] "Directed acyclic graph (dag)." `https://hazelcast.com/glossary/directed-acyclic-graph/`. [Online; accessed -24-November-2020].

[112] "7 popular stream processing frameworks compared." `https://www.upsolver.com/blog/popular-stream-processing-frameworks-compared`. [Online; accessed -24-November-2020].

[113] E. Shahverdi, A. Awad, and S. Sakr, "Big stream processing systems: An experimental evaluation," pp. 53–60, 04 2019.

[114] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, May 2015.

[115] "Spark overview." `https://spark.apache.org/docs/latest/`, 2020. [Online; accessed -24-November-2020].

[116] "Spark streaming vs. structured streaming." `https://dzone.com/articles/spark-streaming-vs-structured-streaming`, 2019. [Online; accessed -24-November-2020].

[117] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[118] "Spark streaming vs structured streaming." `https://dzone.com/articles/spark-streaming-vs-structured-streaming`, 2019. [Online; accessed -23-November-2020].

[119] "Spark: Rdd vs dataframes." `https://blog.knoldus.com/spark-rdd-vs-dataframes/`, 2019. [Online; accessed -24-November-2020].

[120] D. Gorasiya, "Comparison of open-source data stream processing engines: Spark streaming, flink and storm," 09 2019.

[121] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," pp. 1507–1518, 04 2018.

[122] "What is skewness?." `https://www.investopedia.com/terms/s/skewness.asp#:~:text=Skewness%20refers%20to%20distortion%20or,varies%20from%20a%20normal%20distribution`, 2021. [Online; accessed -24-November-2020].

[123] "Spark structured streaming." `https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html`, 2016. [Online; accessed -24-November-2020].

[124] "Apache kafka." `https://kafka.apache.org/`, 2020. [Online; accessed -25-November-2020].

[125] "Kafka streams." `https://docs.confluent.io/platform/current/streams/concepts.html#`, 2020. [Online; accessed -25-November-2020].

[126] "Flinkathon: What makes flink better than kafka streams?." `https://blog.knoldus.com/flinkathon-what-makes-flink-better-than-kafka-streams/`, 2019. [Online; accessed -25-November-2020].

[127] "Apache flink." https://flink.apache.org/flink-architecture.html, 2020. [Online; accessed -25-November-2020].

[128] "Hazelcast." https://hazelcast.org/, 2020. [Online; accessed -25-November-2020].

[129] "What is an in-memory data grid?." https://hazelcast.com/glossary/in-memory-data-grid/, 2020. [Online; accessed -25-November-2020].

[130] "Esper." https://www.espertech.com/esper/, 2021. [Online; accessed -26-May-2021].

[131]

[132] "Gherkin reference." https://cucumber.io/docs/gherkin/reference/, 2020. [Online; accessed -6-December-2020].

[133] "Moscow prioritization." https://www.productplan.com/glossary/moscow-prioritization/#:~:text=The%20method%20is%20commonly%20used,not%20have%20at%20this%20time, 2020. [Online; accessed -6-December-2020].

[134] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, 12 2018.

[135] C. Y. Chen, J. H. Fu, T. Sung, P. Wang, E. Jou, and M. Feng, "Complex event processing for the internet of things and its applications," in *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 1144–1149, 2014.

[136] "Flinkcep - complex event processing for flink." https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html, 2020. [Online; accessed -12-December-2020].

[137] "Flink siddhi." https://github.com/haoch/flink-siddhi, 2021. [Online; accessed -28-May-2021].

[138] "Flink esper." https://github.com/phil3k3/flink-esper, 2021. [Online; accessed -28-May-2021].

[139] "Kafka streams cep." https://github.com/fhussonnois/kafkastreams-cep, 2021. [Online; accessed -28-May-2021].

[140] "Flink 7129." https://issues.apache.org/jira/browse/FLINK-7129, 2021. [Online; accessed -28-May-2021].

[141] L. Rosa, P. Alves, T. Cruz, P. Simoes, and E. Monteiro, "A comparative study of correlation engines for security event management," 04 2015.

[142] "Esper vs drools." https://www.zymr.com/using-drools-vs-esper-rules/, 2021. [Online; accessed -28-May-2021].

[143] "How apache flink™ handles backpressure." https://www.ververica.com/blog/how-flink-handles-backpressure. [Online; accessed -12-December-2020].

[144] "Filtering traffic flow data." https://developer.here.com/documentation/traffic/dev_guide/topics_v6.1/flow.html, 2020. [Online; accessed -15-December-2020].

[145] "Docker volumes." https://docs.docker.com/storage/volumes/, 2021. [Online; accessed -28-May-2021].

[146] "Esper pattern documentation." http://esper.espertech.com/release-8.7.0/reference-esper/html/event_patterns.html, 2021. [Online; accessed -28-May-2021].

[147] "Principles behind the agile manifesto." http://agilemanifesto.org/iso/en/principles.html, 2020. [Online; accessed -2-December-2020].

[148] "Agile 101." https://www.agilealliance.org/agile101/. [Online; accessed -2-December-2020].

[149] "What is scrum? how it works, best practices, and more." https://dzone.com/articles/what-is-scrum-software-development-how-it-works-be, 2020. [Online; accessed -25-November-2020].