POLITECNICO DI TORINO

Master of Science Degree in Computer Engineering

Politecnico
di Torino

MASTER'S THESIS

In collaboration with
California State University Los Angeles

Cloud-native Kubernetes application to efficiently and securely stream and collect real-time data

Supervisors
Prof. Giovanni MALNATI
Prof. Marina MONDIN

Candidate
Alessio SANTANGELO

April 2023

Abstract

Properly collecting and distributing data has become more and more important in recent years. The vast majority of companies, universities, research groups, and other similar entities need to manage some kind of data. For this reason, it is really important to have a system capable of suitably performing this task. The objective of this thesis is to design, develop and test such a system. More specifically, the intent is to build a product that can be easily adapted to most of the applications in which is important to gather and distribute big amounts of data in real-time. Nevertheless, in order to study a real use case with actual requirements, it has been developed a system capable of efficiently and securely collecting and streaming real-time data representing electromagnetic field (EMF) measurements, which allows human users to study, both utilizing real-time and historical data, the cellular coverage in California. To tackle this problem, a Microservices architecture has been chosen. This architecture was selected because it is well suited for the goal of having a scalable, flexible, and easily extensible distributed system. Kubernetes has been utilized as the containers orchestrator and the entirety of the cluster has been hosted on Google Cloud. The data is streamed and stored using Apache Kafka and MongoDB, respectively, while the security of the system is managed with the use of Anthos Service Mesh. The final product is able to properly and flawlessly process the necessary requests to suitably analyze the cellular coverage of the whole of California. In addition, it does so without overpaying for unused resources that have been carefully allocated to specifically meet the requirements.

To my mom

Table of Contents

List of Tables

List of Figures

Acronyms

A

AI	Artificial Inteligence
API	Application Programming Interface
AWS	Amazon Web Service

C

CA	Certificate Authority
CPU	Central Processing Unit
CSP	Cloud Services Provider

D

DBMS	Database Management System

E

EMF	Electric and Magnetic Fields

G

GCP	Google Cloud Platform
GKE	Google Kubernetes Engine

H

HTTP	HyperText Transfer Protocol

I

IaaS	Infrastructure as a service
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol

IT	Information Technology
J	
JSON	JavaScript Object Notation
JWT	JSON Web Token
K	
K8s	Kubernetes
M	
MAC	Message authentication code
mTLS	mutual TLS
O	
OOP	Object Oriented Programming
OS	Operating System
P	
PaaS	Platform as a service
PC	Personal Computer
R	
RAM	Random Access Memory
S	
SaaS	Software as a service
SQL	Structured Query Language
SSL	Secure Sockets Layer
T	
TCP	Transmission Control Protocol
TLS	Transport layer security
V	
VM	Virtual Machine

Chapter 1

Introduction

The term data is nowadays used in every possible context. At its most basic, data is a mere collection of different information like measurements, findings, and numbers, that is formatted in a way compatible with what handles it. As simple as it looks, this concept has drastically changed how we live our lives and how every company and organization around the world works. Any business produces tons of data every day and, at the same time, it knows, if it wants to be competitive and grow, it is vital to collect and analyze all the possible information related to its activities. Modern businesses make use of data in a variety of ways. They discover faults in their system by analyzing logs, improve their services by gathering information obtained from customer surveys, and of course data, in form of results, is what is needed to understand what has already been done well. The list can go on and on, but, at this point, it is evident how important this concept is.
Every company, university, research group, and whatsoever type of entity studies and analyzes data in the way that is more suitable for what is their objective. However, what they all have in common and what they all need is a way for collecting it. For this reason, it is crucial to have a system capable of properly performing this task.

Over the years thousands of solutions have been developed to suitably collect data. The most straightforward way is obviously to manually write it on a piece of paper or on a text file. However, it is equally obvious as this is not the most efficient, flexible, and *up-to-date* solution. Instead, what is commonly used nowadays is a database, a set of data stored in a computer that is set up for easy access, management, and updating. Every database could be local, meaning that it is possible to access it only from the machine in which it is held. Nevertheless, a local solution would not suitably work in most cases since every data source would have to be directly connected to the computer hosting the database. To solve this problem, an architecture that allows to access the data from different locations is

needed. It is possible to build a piece of software capable of accessing the database and exposing it to the Internet in order to be accessed by other devices around the world. This is one of the most basic distributed systems architectures in which we have a single server, accessing and exposing the database, and different clients that retrieve the data by contacting the server through the Internet. However, one of the main drawbacks of this type of architecture is that we actually need a server machine capable of performing this task. In some cases, this could be expensive and hard to set up. For this reason, Cloud providers, like Google or Amazon, offer their own servers where developers can run their software. This drastically cuts costs and eases the configuration process. Distributed applications developed using this idea are named Cloud-native applications.

This thesis work is exactly about the study, design, development, and testing of the last described type of architecture to solve the aforementioned data collection problem. In particular, the project consists of the development of a Cloud-native application capable of collecting and distributing data in a reliable and secure way.

1.1 Case study

Even if the system developed can be easily applied to every application in which is important to gather and stream in real-time big amounts of data; it has been chosen to tackle a specific case study in order to deal with a real-world problem with actual requirements. In particular, the objective has been to develop an application capable of streaming in real-time and, at the same time, storing for future access electromagnetic field (EMF) measurements to study the cellular coverage in California.

1.2 Thesis outline

The thesis is organized into an introduction, 6 main chapters, and conclusions. The content of the main chapters is briefly described hereinafter:

- **Distributed systems** It explains some of the most important theoretical topics required to properly understand what is covered in the following.
- **Applicable technologies** This chapter briefly describes the main technologies utilized in the development of the system studied in this thesis.
- **Design** It starts with an analysis of the requirements to later move to what design choices have been made in order to meet those requirements. In particular, the developed architecture, including its data format and flow, is presented.

- **System components** It includes a detailed description of every component of the system from its configuration to what benefits it provides to the overall application.
- **Security** This chapter explains how the system has been protected and secured from possible attacks. In particular, the chapter is divided into two different sections that separately discuss inner and outer cluster (i.e. the set of servers where the application was hosted) security.
- **Tests and Results** Here the testing tools utilized and results obtained are presented. Specifically, a series of graphs and tables show how the developed application adequately works when subject to real use cases.

Chapter 2

Distributed Systems

Most of the software built today works in a distributed environment. This means software components are not produced as an isolated system anymore but instead they are designed to communicate with other likewise entities through a network (the Internet in most cases). The vast majority of modern applications are conceived as different distributed components from the ground up. This technique allows completing the job more efficiently and, often, with benefits unreachable by a single local machine. One of the most significant examples is preventing the single point of failure problem. Having different distributed components in the system means that when a single one of them stops working, all the others can keep functioning. This characteristic is of fundamental importance, especially for those applications that are defined as *mission critical* where even a brief downtime is likely to have negative consequences.

There are 4 different types of distributed systems [1]:

1. **Client-server** With this technique several clients send requests to a single central server that returns an output response to the clients. It is considered the most simple and traditional type of distributed system.

2. **Peer-to-peer** Here a decentralized architecture is followed. Each component can operate as both the client and server. Every process has the same privileges and capabilities as all the other processes in the system.

3. **Three-tier** This type of architecture uses different servers for different activities. Most specifically, it includes a presentation layer to manage the user interface, an application layer to access the data and provide it to the presentation layer, and a data tier that hosts the database. With respect to the simple client-server model, this technique allows independent deployment and more scalability

4. **N-tier** It is an extension of the three-tier model where several layers, each one for a single physical server, perform different functions in the network. A typical example of n-tier architecture is the Microservices-based architecture where each service is responsible for its own data and communicates with the other services in order to compute the final output to send to the clients.

2.1 Key characteristics

Distributed systems have several features and characteristics that make them so widely used and important for IT and computer science. Some of the most important ones are [2]:

- **Scalability** The ability to grow when the size of the workload increases. This can be achieved with vertical or horizontal scaling. With the first technique, scalability is obtained by adding more power (e.g. CPU and/or RAM) to an existing machine, whereas with horizontal scaling more machines are added to the pool of resources.
- **Concurrency** The different components of the system can run simultaneously and independently.
- **Availability/fault tolerance** If a node fails, the others can keep working without, in most cases, big issues.
- **Transparency** A distributed system is usually exposed to its users as a single computation unit. An external programmer or end user does not need to deal with the complexity of the different internal components.
- **Heterogeneity** The nodes composing the system have the possibility to use different hardware and software since they represent different physical components.
- **Replication** Distributed systems enable a server to be replicated, meaning that two or more servers menages the same redundant resource to provide more reliability and scalability

It is already been partially described as those characteristics come with different vantages. The overall system is more reliable and better performing (easier to scale) and, at the same time, it enables greater flexibility with its heterogeneity and transparency. Nevertheless, it is essential to notice that distributed systems also come with some disadvantages with respect to traditional computing environments. They are more complex to design, manage and understand. In addition, properly

synchronizing the processes can be really challenging and requires careful programming. Finally yet importantly, a distributed system includes different components each of which introduces the possibility of security breaches and issues.

However, it is important to understand and specify the relationship between advantages and disadvantages and how this applies differently depending on the specific distributed system considered. Usually, if properly designed, the more an architecture is distributed (higher number of components in different physical locations), the more of its benefits are achieved, but, at the same time, a more complex and insecure system is obtained. When designing a distributed system, there are so many variables and technologies to study and consider. That is why every final product is different from all the others and none of them share the exact same properties.

2.2 Microservices architecture

As already briefly mentioned at the beginning of this chapter, the Microservices architecture is a type of n-tier distributed system. This kind of architecture has been conceived to deal with the complex and huge codebases most modern businesses work with nowadays. In this context, different requirements need to be satisfied [3]:

- several teams need to be able to work on the same application at the same time
- new members must be able to quickly understand the architecture, most specifically the components they will be working on, in order to promptly become productive
- all the different elements composing the application must be easy to deploy, modify, and replicate
- the company needs to be able to smoothly and quickly integrate new technologies in order to keep being competitive

The solution to address these requirements is to design an architecture that structures the application as a set of small, loosely coupled, and collaborating services, each one for a specific function. In this way, all the different components are independent and thus easily understandable, deployable, and testable. Every team can simultaneously work on a single service without impacting the work of other teams and vice versa allowing faster productivity and lower communication overhead. Furthermore, having isolated services with separate functionalities improves fault isolation. If there is a problem in a service and thus in a single functionality, only

that component will be affected while the others can continue to work.

Every service manages its own database in order to be further decoupled from the other services. Whenever a component necessitates some information from another element in the system, it needs to request it from the other party since it cannot directly access other databases that are not its own.

Two communication methods are mainly used in Microservices architectures: synchronous and asynchronous. The first one needs the entity asking for information to wait until it receives a response and, at the same time, the other entity needs to be up and running in order to receive the request; a typical example is HTTP. On the other hand, what happens with asynchronous protocols is that the two communicating elements send messages to a third party that works as a broker. In this way, neither of the two services needs to be up and listening to the other simultaneously. Instead, they can retrieve the message from the message broker whenever they deem it necessary. Typical examples of technologies based on a message broker are RabbitMQ and Apache Kafka.

2.2.1 Example

A typical example of a Microservices application is shown below in figure 2.1. In particular, a simplified view of an E-Commerce App is depicted.

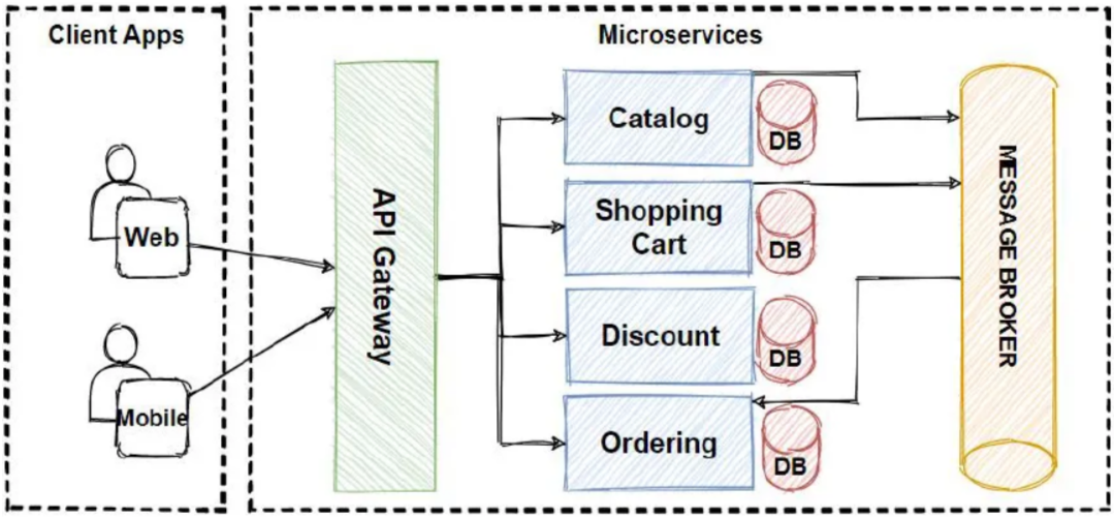

Figure 2.1: Microservices application example [4].

The figure shows 4 different microservices (Catalog, Shopping Cart, Discount, and Ordering) with their own private databases. They communicate with each other

using a message broker, thus asynchronous intra-service communication is used. Furthermore, an API Gateway is introduced. This component is of fundamental importance for Microservices architecture, it is the proxy that accepts requests from client applications, in this case, Web and Mobile clients, and routes them to the appropriate service. It is through this element that the transparency property is obtained.

2.2.2 When to use the microservices architecture?

This type of architecture is designed to deeply exploit most of the benefits a distributed system has to offer, plus some extra advantages such as faster development and deployment thanks to the small and highly isolated elements of the system. However, this also means dealing with the often challenging intra-service interactions and, generally, with the extra complexity and security issues introduced.

It can be really hard, especially for a small company or a startup, to design, deploy and maintain a microservices architecture and, furthermore, it might not be a good idea to use it if the objective is to obtain an application that will not likely need to scale and/or evolve in the future. Nevertheless, if the goal is to rapidly evolve the business model and corresponding application, introducing new functionalities and technologies and, simultaneously, being able to satisfy the hopefully increasing number of customers, adopting a Microservices architecture could be a valuable and smart solution.

2.3 Containerization

Typically, whenever it is needed to run a software program on a specific machine, the properly compatible version of that software for the used hardware and operating system needs to be employed. However, even if this technique can work for desktop or mobile apps, it is not the best solution for server applications where developers often need to be able to run the same piece of software on different physical machines without big changes. In addition, in an environment in which a single server needs to be able to run several software elements at the same time, lightweight and performing applications are necessary.

In the early 2000s, virtualization was mainly used to tackle this problem. A Virtual Machine (VM) is a digital copy of the host machine's physical hardware, with its own operating system, used to run programs and deploy apps. A single physical device can run several VMs at the same time. In this context, every instance shares the physical CPUs and memory with all the other VM instances. Virtualization allows an application to be portable to all the systems capable of

running VMs, not a big constraint since there are several free and multi-platform applications capable of running Virtual Machines.

However, this method is not the most efficient solution because an application just needs a small set of all the functionalities offered by an OS. For this reason, another technique, called containerization, exploded in popularity in 2013. According to AWS: "containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure" [5]. It is a concept similar to virtualization, but here applications are provided with the exact resources they need.

2.3.1 How does it work?

Software developers build the so-called *images*. These represent the written code plus all the dependencies and configuration properties needed to run that code in a containerized environment. To obtain this last element, a containerization architecture is needed. As illustrated in Figure 2.2, it is composed of 4 different layers [5]:

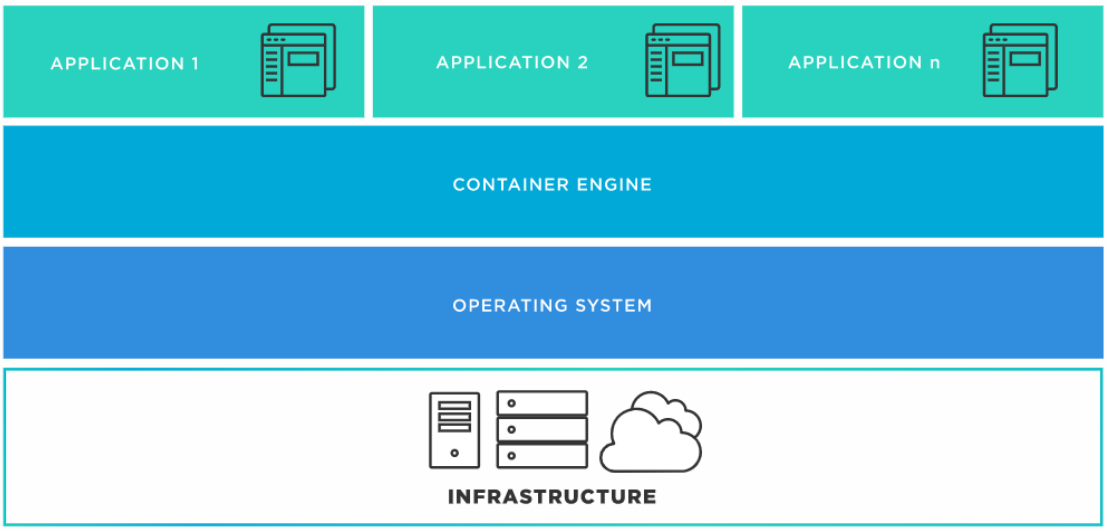

Figure 2.2: Containerization architecture [6].

- **Infrastructure** Starting off with the lowest one, the infrastructure represents the hardware of the physical machine that runs the containerized applications. It can be a bare-metal server, a Cloud infrastructure (better described in the next section), or even a common PC.

- **Operating System** The second layer simply represents the operating system. Its specific type depends on the infrastructure in use.
- **Container Engine** This is the broker between the container (software package containing code and all the related resources to run it) and the OS. It is the entity that allocates resources to the different containerized applications.
- **Applications and dependencies** The topmost layer contains the different running containers. In addition to the already mentioned application code, configuration files, and dependencies. Every element in this layer might also contain a lightweight OS that gets installed over the host operating system.

2.3.2 Benefits

Containerization is so popular today because of the following advantages [5]:

- **Portability** An application written using containers, can be run in every system capable of operating a container engine
- **Scalability** Containers are composed of the essential elements for running a specific application. Thus, they only consume the minimum physical resources they need, leaving space for other containers to run on the same machine.
- **Efficency** The lightweight application does not usually need to boot an operating system or any other unnecessary components and is, thereby, faster.
- **Fault tolerance** Every container operates in an isolated user space. For this reason, a single faulty container does not affect the others.
- **Agility** Since applications run in an isolated environment, developers do not need to worry about the OS, hardware, or any other element when they want to change or troubleshoot the application.

2.4 Cloud computing

IBM describes Cloud computing as "*on-demand access, via the Internet, to computing resources . . . hosted at a remote data center managed by a cloud services provider (or CSP). The CSP makes these resources available for a monthly subscription fee or bills them according to usage*" [7].

Using this type of service has become more and more common in recent years. This is due to the numerous benefits it brings. First of all, Cloud computing greatly cuts costs. There is no more need for a proprietary rack of servers that requires

adequate software, setup, and a group of IT experts that manages it.
Second of all, it is much faster to start working on a new product. It only takes a few minutes to create a cluster on a cloud platform, whereas it is much more time-consuming to purchase and configure the hardware and software required to start a even simple server machine. Similarly, when, later on, the already developed and working system requires scaling, it is simple and fast to perform such a task in a cloud environment. In some cases, it is also possible to allocate resources dynamically depending on the current traffic.
Another benefit regards the possibility to have a reliable and fast system everywhere in the world since, usually, a CSP has servers spread in a lot of different geographical locations.

Cloud technologies are nowadays utilized everywhere. Most of the modern online services take advantage of some sort of cloud service behind the scene. The possible uses are uncountable. However, some of the 2 most representing examples regard storing, backing up, recovering, and analyzing data and creating, testing, and building *cloud-native* (built from the group up on a cloud environment) applications. The just described use cases are often extended and integrated with several other additional tools that the Cloud Platform provides such as load balancers, data analytics, machine learning, technologies to strengthen the developed product's security, and more.

2.4.1 Types of cloud computing

There are three different types of cloud computing architectures [8]:

1. **Public cloud** It is operated and owned by a third-party CSP that offers its services over the Internet. With this type of architecture, the developers do not directly own and manage the hardware and software utilized, the CSP does. Usually, a single public cloud is utilized by several clients that belong to different businesses or organizations. Some examples of public clouds are Google Cloud Platform (GCP), Amazon Web Service (AWS), and Microsoft Azure.

2. **Private cloud** In this case, the cloud computing resources are allocated specifically for a single organization. For this reason, all the related hardware is often situated on the company's on-site datacenter and uses a private network. In this way, the business that owns it has access to all the advantages of a cloud platform (e.g. scalability and easy deployment) plus enhanced security and access control provided by the on-premises infrastructure. However, all of this comes with, obviously, much higher costs with respect to the first type of architecture.

3. **Hybrid cloud** It combines public and private clouds. Specifically, the business's private cloud services are combined with some of the public cloud services to create a flexible infrastructure. This is because the organization can decide the optimal cloud for each application, or even change it on the fly, thus, it can meet its technical and business objectives more effectively and cost-efficiently than it could with public or private cloud alone.

2.4.2 The cloud computing stack

There are different types of services that a CSP can provide to its clients. These are commonly categorized using the cloud computing stack (figure 2.3).

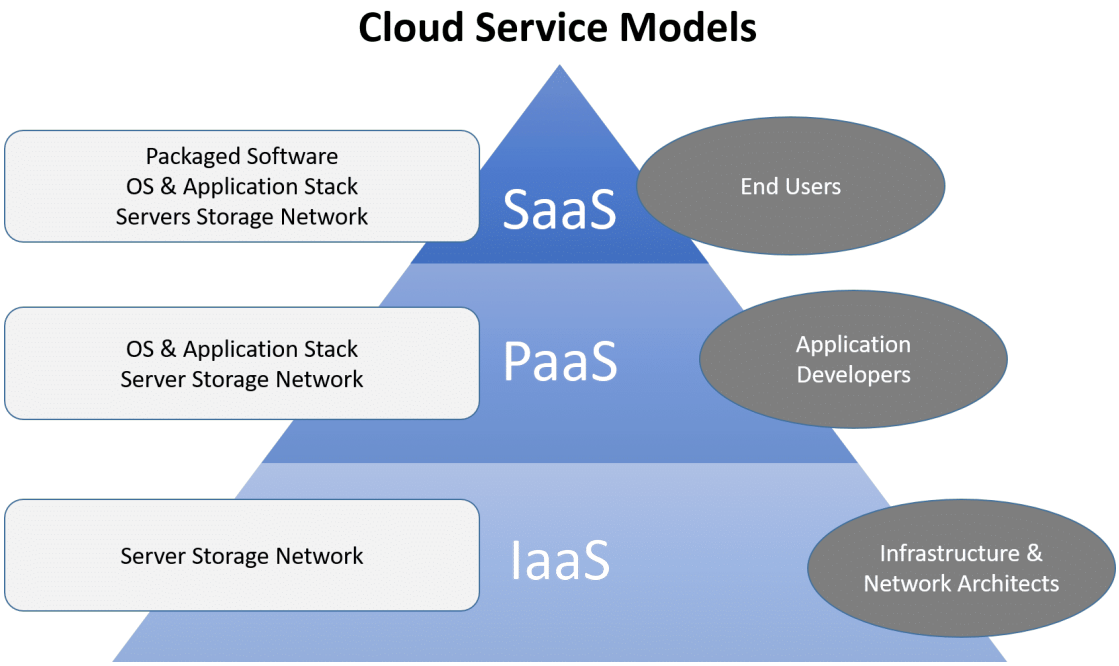

Figure 2.3: Cloud computing stack [9].

As it is shown in the picture, there are 3 different categories built on top of each other. This means that all the services belonging to a certain layer also belong to all the lower ones.

- **Infrastructure as a service (IaaS)** Starting off with the lowermost layer, the developers simply "rent" physical resources (e.g. virtual machines, storage, operating system) from a cloud provider and use them as needed to create their own applications and set up the system to run it. In most cases, this is a pay-as-you-go service.

- **Platform as a service (PaaS)** The CSP offers an environment that makes it easier for developers to quickly create, test and deploy software applications. Specifically, developers only need to care about the application development without worrying about setting up and managing the hardware and software underneath. In this case, the client pays for the actual resource used, as it would do with IaaS alone, plus an extra cost for the development platform provided.

- **Software as a service (SaaS)** This is basically a method for delivering software applications over the Internet. The CSP manages both the application and its underlying infrastructure while the client only needs to access the service through the Internet, commonly utilizing a web browser. Clients usually pay for a monthly subscription to get access to the service

2.5 Security

Security in computer science is a really wide topic. It can be described, at its most basic, as the study of all the IT systems' vulnerabilities and what can be done to avoid people unsolicitedly exploiting them for their own interests. Some examples of possible unwanted actions that can be performed by nefarious users are: reading private data, masquerading as another person or entity, and destroying, altering, or providing false information. Specifically, security is critical when talking about distributed systems, where data is usually transferred on public networks and servers can be accessed by whoever has a valid address.

In this context, it is crucial to provide the following [10]:

- **Confidentiality** It represents the state in which information is hidden from unauthorized individuals. It is usually addressed by encrypting the data. In such circumstances, whether or not this data is accessible to a user depends on their identity.

- **Integrity** It deals with the trustworthiness of data. It allows identifying if some information has been unsolicitedly changed. Other than verifying the correctness of data, integrity is also useful to authenticate users and entities.

- **Availability** It is about having access to computing resources by users. In particular, it refers to how accessible and properly functioning a system is. Attackers can perform some harmful actions that make the IT system stop working properly, impairing its availability,

2.5.1 Cryptography

Cryptography essentially represents the study of techniques to secure communications in front of unwanted and unauthorized external actions. It is used to provide encryption and authentication, but it can also offer integrity and nonrepudiation (assuring an entity cannot deny the creation of a message).
One of the most fundamental applications of cryptography is encryption, a technique used to convert human-readable information (often called plaintext) into incomprehensible data (also known as ciphertext). One or more keys are usually used to encrypt (plaintext to ciphertext) and decrypt (ciphertext to plaintext) the message.
In particular, there are two different types of encryption:

- **Symmetric encryption** The same key is used to both encrypt and decrypt the message. This is a fast technique but needs the two parties to share the same key, this can be a complication since an attacker could read that key while it is distributed. This type of encryption is widely used to provide confidentiality.

- **Asymmetric encryption** Here two different keys are used, private and public keys. Whenever one of the two is used to encrypt the message, the other must be used to decrypt it. Usually, a public key is accessible by everybody while a private key is only known by a single entity.
  This type of algorithm is slower than the symmetric version but it is usually more secure since no key distribution is needed. Asymmetric encryption is widely used to digitally sign messages (described below) and distribute symmetric keys. This last operation is performed by encrypting the key that needs to be distributed with a public key. In this way, it is certain that only who owns the associated private key can decrypt and read the encrypted key.

Another important method studied in cryptography is cryptographic hashing, A hash function is similar to encryption because it generates incomprehensible data from human-readable information. However, it differs from encryption because the output of a hash function has always the same amount of bits.
This technique can be used to provide integrity, whenever a message is sent on a public network, its hash code (the output of a hash function) is sent with it. Doing so, who receives the message can easily check the message was not modified by an attacker simply recomputing its hash code and comparing it with the one sent by the other party. However, it is easy for an attacker to recompute the hash and modify that as well. For this reason, message authentication codes have been introduced (MAC). With this technique, the hash function is combined with asymmetric encryption in order to allow only those who know the encryption key to properly compute and verify a MAC. This technique provides authentication

and integrity but does not provide nonrepudiation. It is not possible to tell for sure which one of the two parties involved in the conversation has actually generated the data.

A digital signature is similar to a MAC but it also provides nonrepudiation. In this case, the hash code is encrypted using the sender's private key. Only one of the two parties has access to that key and, for this reason, it is unquestionably clear who generated the message.

2.5.2 Transport Layer Security (TLS)

TLS is a cryptographic protocol that provides end-to-end security to data sent over the Internet. It is most known for being used to establish security when web browsing, but it can also be spent for providing security to other applications such as e-mail, file transfer, and voice-over-IP. TLS evolved from Secure Socket Layers (SSL) which was originally developed to secure web sessions. In particular, TLS is based on SSL 3.0.

TLS and SSL are not neatly positioned in one of the 4 layers of the TCP/IP model. Instead, they can be considered placed in between the application and transport layers. This is because, by definition, TLS and SSL run on top of a reliable transport layer to provide services to an application layer.
Their main functionality is to encrypt data sent over the Internet to avoid unauthorized users to read it. As already mentioned, TLS is really used when accessing web pages, this is because, usually, a lot of sensitive information, such as login credentials and credit card details, is being sent between the web browser and the backend (server-side software). Nevertheless, it can be utilized in all the contexts in which is needed to protect sensitive data sent over a public network. In addition, TLS can be also used to authenticate one or both the entities involved in the communication. In the case of web browsing, only the server is commonly authenticated. This is due to the fact that in most of the web applications it is the client that sends sensitive information to the server and thus, he is the one who must verify to whom or what this data is being sent.
When only one of the two entities is authenticated, TLS is commonly called *simple TLS*, whereas it is named mutual TLS or mTLS when both parties are authenticated.

TLS uses symmetry encryption to protect the data sent. However, before doing that, the two parties need to exchange a key to perform encryption and decryption. To do so, asymmetric encryption is used. In the typical example of a web browser communicating with a web server, a digital certificate associates a public key to the server. A digital certificate is a file that ties the identity of

a node in a network to an asymmetric key pair. A certificate basically assures that the certified entity owns and knows the private key associated with the public key indicated on the certificate. This document, in order to be trusted, needs a certificate authority (CA) that digitally signs it. A CA is simply an entity that issues certificates. Most modern browsers already include a set of trusted CAs when installed.
As mentioned, simple TLS is used in most cases and, in this context, it is the server that is authenticated and needs a digital certificate. However, whenever mTLS is utilized, also the client requires a certificate to share with the server.

It is important to point out that setting up a TLS session is surely a time-consuming operation. This must be taken into consideration when designing a distributed system. However, it still remains the choice of a lot of engineers since, in the majority of cases, it is a price worth paying to suitably protect the application.

Chapter 3

Applicable technologies

Several technologies have been used during the development of the application studied in this thesis. Why every one of them has been specifically chosen is comprehensively described in the next chapter, while it has been decided to list and define all the tools used in the current chapter. This approach provides a concise overview of the stack utilized and might help in better understating why some choices have been made in the design of the developed system.

3.1 Docker

It has already been described in the previous chapter what containers are and Docker is nothing less than the most famous and used container engine.
It is defined on its official website as: "an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly." [11].
Specifically, it does so, differently from the traditional development environment, by creating a system that understands, through proper user setup, what are the requirements to run a specific software application independently of the infrastructure (i.e. mainly hardware components and OS).
In other words, being a container engine, Docker is used to develop and test an application (with its dependencies) in an isolated and lightweight environment (container) and allows to easily and efficiently ship, modify, and scale such an application to production. All of this, without worrying about properly setting up the various systems to run a specific application since it is enough to install Docker in those systems to be able to run every type of Docker container.

Docker is written using the Go programming language and exploits some features of Linux to work as a container engine. In particular, it uses Linux namespaces to

provide isolated containers. In particular, every characteristic of each container runs in a specific namespace and cannot be accessed from elsewhere.

Docker uses a client-server application where the server is represented by the *daemon*, the component that actually manages the containers. Instead, the client is simply a piece of software that communicates with the daemon and asks it to perform the operations needed to run applications.

Another important functionality provided by Docker is *Docker registry*. It represents a place where to store different Docker images. One of the most important registries is Docker Hub which is managed by Docker itself and is public, this means everyone can host their own Docker registry on it. In addition, it provides free access to most of the commonly used pieces of software such as operating systems, databases, and development environments. A lot of public images publish there often represent the starting point for new custom images.

3.2 Kubernetes

"Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications" [12].
It has already been described why containers are an efficient way of developing, testing, and deploying an application. However, in modern software products, it is really common to have different containers that run together in a cluster of servers in order to provide a powerful, scalable, and fast system. In this context, ensuring that all the components run smoothly without downtimes and/or errors could be really hard to achieve. For this reason, a container orchestrator, like Kubernetes, is needed.
More specifically, the main features of Kubernetes are [12]:

- **Service discovery and load balancing** Kubernetes is able to expose a container to the other containers in the cluster or directly to the public Internet providing an IP address and network port. In addition, whenever there is high traffic, Kubernetes can load balance requests to the different instances of the same container.
- **Storage orchestration** K8s allows to automatically mount a storage system to the containers it manages. It supports different types of storage, including the "virtual" ones offered by cloud platforms.
- **Automated rollouts and rollbacks** Every time the system manager wants to change the state of a deployment (i.e. a set of containers that run the same image to offer scalability and availability), Kubernetes allows doing it

smoothly without any downtime. For example, when the system requires to increase or decrease the number of replicas of a deployment, it is enough to slightly modify a configuration file and K8s performs the update while keeping the deployment up and available.

- **Automatic bin packing** It is possible to specify how many resources to allocate for each container in the system and Kubernetes, considering the current state of the cluster, automatically allocates such resources in the most efficient way.

- **Self-healing** Every time a container of the system stops working, K8s either restarts it or kills and substitutes it with another working one.

- **Secret and configuration management** Kubernetes stores and manages all the configuration information needed for the cluster to work. Whenever a configuration change occurs, it automatically detects it and performs the due actions to update the system. It also permits storing sensitive information such as passwords and cryptographic keys without exposing them to unauthorized users.

3.2.1 Architecture

The Kubernetes architecture is complex and composed of several elements, deeply describing it is out of the scope of this thesis. However, it is important to briefly explain its main components, what abstractions provides, and how engineers can interact with it.
A Kubernetes cluster mainly consists of a set of working machines, called nodes, and a control panel that manages them. The system manager can use a group of APIs to talk with the control panel and configure the cluster (i.e. change the state of the nodes). Every worker node hosts one or more *Pods* that in turn host one or more containers. A Pod is the smallest manageable unit in Kubernetes. This means that does not exist a smaller component that can be created, updated, or deleted by K8s.

Engineers do not usually create Pods directly but instead, they create *Delpoyments*. A Deployment is one of the so-called Kubernetes Objects, elements used to represent the state of the cluster. Specifically, they can describe what containers are running, what resources are allocated, how storage is managed, how network policies are applied and so much more. To create Objects, the Kubernetes APIs need to be called. A common technique to perform such a task is to declare Objects through specific *.yaml* files and then use the official command-line interface, called *kubectl*, to directly contact the control plane.
Some examples of Kubernetes Objects, other than Deployments, are *Services* (used

to expose a network application that is running on a Pod), *PersistentVolumes* (define pieces of storage in the cluster), and *HorizontalPodAutoscalers* (automatically scales a workload resource to match the current load).

For its characteristics, Kubernetes is the perfect solution for orchestrating microservices applications. In such systems, it is very common to deploy different technologies that follow different principles and need specific management. In this context, it can be really difficult for a human user to properly create and handle all the necessary default Kubernetes resources to run this diversity of techniques. Having to manually deal with such complexity every time a new component is added to Kubernetes can be a huge waste of time and human resources. For this reason, two K8s features have been introduced: *custom resources* and *Operators*. The first one allows extending the default Kubernetes APIs in order to enable the creation of new resources represented by a set of specific objects. In this way, an engineer only needs to refer to a small group of functions added to the default APIs to create all the objects necessary to deploy a specific technology. For example, a newly added custom resource, called *myDB*, could allow, with the use of a single and simple *.yaml* file, creating a set of Deployments, Services, and configuration files to set up and run a specific database.
Side to side with custom resources there are Operators. Their objective is to automate all the repetitive processes regarding the management of custom resources. For instance, an operator could automatically handle updates, take backups of a resource's state, and choose a leader for a distributed component.
Both custom resources and Operators are widely used for the development of a lot of Kubernetes applications since they deeply help in the proper deployment of some well-known technologies such as Apache Kafka and MongoDB.

3.3 Google Cloud Platform

Google Cloud Platform or GCP [13] is a collection of cloud computing services offered by Google. The huge clusters of servers and backbone that allows Google to provide these cloud services to its client are also the same that Google uses to develop its own products such as Gmail and YouTube.
Google Cloud has been used by different important companies around the world including Twitter, PayPal, and Carrefour
It offers more than 150 cutting-edge products, most of them available with a pay-as-you-go service. It gives the possibility to create and run virtual machines, store and retrieve huge amounts of data, deploy fully managed databases, use AI tools, and much more.

3.3.1 Google Kubernetes Engine

Google Kubernetes Engine (GKE) [14] is one of the most powerful tools provided by GCP. It gives the possibility to host an entire Kubernetes cluster entirely using the infrastructure owned by Google. GKE provides both the control plane and a set of worker nodes, where the user can decide to request as much computational power (in terms of worker nodes, CPUs and memory) as they need. In addition to providing the resources to run a Kubernetes cluster, GKE is also integrated with some other Google Cloud Platform features, such as logging, monitoring, network, and security solutions in order to help developers to build a more reliable and forefront system.

GKE services have a cost that depends on how many resources (nodes, CPUs, and memory) have been allocated to the cluster, plus the cost of all the other integration tools that have been used to enhance the system. In this context, it is really common for clients to over-allocate resources that are eventually never used. For this reason, Google Cloud Platform offers two different modes of operating a GKE cluster:

- **Autopilot** The cluster is already configured by Google to host production-ready workloads (containers). Every time the developers deploy a new workload, GKE automatically optimizes the cluster to allocate the necessary resources to do that. Thus, users will not have to overpay for allocated resources that were never needed.
- **Standard** The engineers who are using the service have full control of the cluster and its nodes' infrastructure. They will pay for the nodes and resources they have explicitly requested.

3.3.2 Anthos Service Mesh

A service mesh is a dedicated infrastructure layer that can be "attached" to a distributed system to transparently add capabilities such as security and observability. It is mainly used in systems with several components, such as microservices applications, since most of the features it provides usually regard inter-cluster communication (i.e. communication between the components in the system). Generally, whenever a distributed system has a service mesh installed and configured, it is enough to add a new component to the architecture to automatically and transparently let the service mesh provide its features to this new component.

Anthos Service Mesh [15] is the Google implementation of the famous Istio open-source project, a highly configurable and powerful open-source service mesh platform.

Anthos Service Mesh, together with most of the other service meshes, consists of a control plan and a data plan:

- **Control plan** It configures the communications between components in the system. Anthos Service Mesh offers two different types of control plan. The first one, called *managed control plane*, is fully managed by Google, meaning that the system administrators only need to configure it and then Google will do all the work to make it work properly. The second type of control plane is called *in-cluster control plane*. In this case, the system engineers are the ones responsible for administrating the control plane.
- **Data plan** This is the part of the service mesh that actually handles the communications in the system. It works through proxies deployed as sidecars (i.e. containers that run alongside the main containers to provide extra features). This means that every workload in the service mesh is supported by a proxy that performs all the actions useful to provide the extra features that the service mesh offers. A simple example is encrypting and decrypting the traffic sent and received.

Anthos Service Mesh offers a set of features that helps manage, observe, and secure system components in a simple and effective way. For instance, it can provide fine-grained control over intra-cluster communication, auto-gather and log a lot of information regarding how the cluster is working, set up mutual TLS between elements in the network, furnish API gateway, and more.

3.4 Apache Kafka

"Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications" [16].

To properly understand what just cited is fundamental to know what event streaming is. It is the practice of capturing data from different sources, such as web applications, databases, and IoT sensors and storing it for later retrieval by other software entities. A lot of people think of an event streaming platform as a digital equivalent of the human body's central nervous system. It is a system that receives data from some units and reroutes it to other units. Such a system is essential for most modern software applications where there is a continuous flow of information that needs to be at the right place and at the right time.

As already mentioned, Apache Kafka works as an event streaming platform. In particular, it provides three capabilities to its users:

1. It gives the possibility to publish (write) and subscribe (read) events (data processed by Kafka) in a real-time fashion.

2. All the events can be stored for as long as needed. Even if some data is read, it is not deleted and can be read again.

3. Events can be consumed both when they occur or even later on when consumers actually need them.

All of this is provided in a secure, scalable, and reliable way. Additionally, Kafka can be deployed as a classical application on bare metal hardware, but also on virtual machines and containers, and on-premises as well as in the cloud.

3.4.1 Architecture

Kafka is run as a cluster composed of several distributed servers. The brokers represent Kafka's core, they are the servers responsible for storing and distributing the events (data produced and consumed). Usually, there are different brokers to increase availability and performance. Specifically, every event can be replicated and partitioned to different brokers. This means that various copies of the same event can be sent to different servers, avoiding loss of data when a broker goes down and thus leading to increased availability. At the same time, the same event is divided into several servers allowing a consumer to process the same event in parallel and obtain better performance.
Other types of servers run alongside the brokers. One or more of them usually work as "coordinators". A typical example is *ZooKeeper*, this piece of software is responsible for storing and handling information regarding the Kafka cluster and details of the consumer clients. This component is also in charge of managing partitions, topic creations, and failures.
Another type of software component often present in a Kafka cluster is Kafka Connect. It allows continuously producing and consuming data from/by an existing system such as a database or even another Kafka cluster.

Image 3.1 shows a simplified view of a Kafka ecosystem. It is possible to see how several consumers and producers can interact with the cluster and how different brokers in the cluster manage different topic partitions. In particular, a single broker usually handles some topics which are *leaders* and some others which are *replicas*. All the consumers that want to read a specific partition of a particular topic, will do it from the leader partition. Replicas, instead, are only used when the broker managing the master partition crushes.

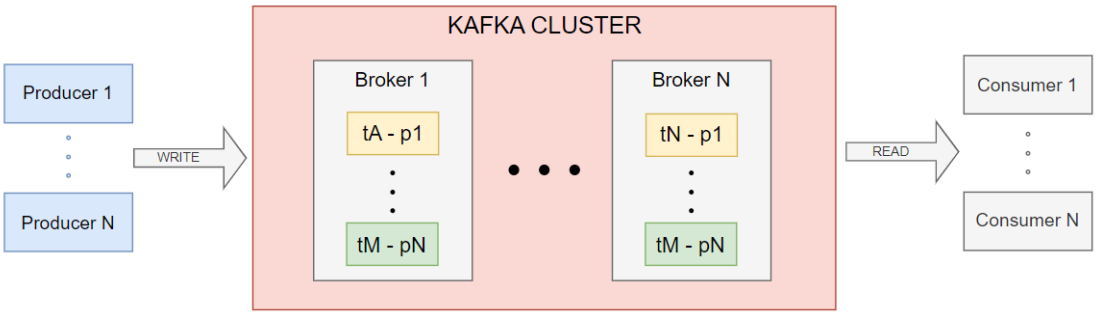

Figure 3.1: Simplified view of the Apache Kafka architecture.

It has already been mentioned how Apache Kafka can be deployed as a container and also on a cloud platform. Thus, it is an ideal candidate for Kubernetes and GCP. Public images of the various Kafka components are available for free on Docker Hub and it is, thus, possible to deploy Kafka on Kubernetes a piece at a time. However, since properly setting up the several elements of a Kafka cluster could be complex, various Kubernetes Operators and customer resources have been formulated to easily perform such a task. A popular and open-source tool that simplifies the process of deploying, configuring, and managing Kafka clusters on Kubernetes is Strimzi. Other than its main purpose (i.e. operating Apache Kafka on K8s), Strimzi provides extra suites to enhance the security, operativity, and observability of the cluster.

3.5 MongoDB

NoSQL stands for *not only SQL* and refers to all the database technologies that do not strictly follow the constraints imposed by SQL. This fundamentally means this type of database does not need to store data in form of tables with rows and columns and, as a result, a system with a flexible data model is obtained. However, it is important to point out that *NoSQL* does not mean that none of the SQL principles can be used but, on the contrary, it is its developers that can choose if to apply all, some or none of those principles.
There are different types of NoSQL databases such as document databases, key-value stores, wide-column databases, and graph databases. All of them come with some pros and some cons and whether to choose one or another depends on the specific application. Describing every data model is out of the scope of this thesis. However, it is important to remark that usually most NoSQL databases have in common, independently from the specific type, to be built from the ground up to efficiently process big amounts of data.

One of the most known NoSQL databases is MongoDB [17]. Specifically, it is a document-oriented NoSQL database, meaning that it is designed to store documents, which consist of key-value pairs that are grouped into collections that are in turn grouped into databases. In particular, its syntax follows the JSON format that is based on the object notation of the famous JavaScript programming language.

Some of the most important properties of MongoDB are:

- The size and number of fields of each document can be different from the others in the same collection.
- The JSON format is in line with what usually deal with since it highly recalls how objects are represented in most OOP languages.
- Every document can contain other smaller documents forming a hierarchical structure. This often avoids performing join operations that are usually quite expensive.
- It is not necessary to define a schema for a collection. Even if there are ways to do that, it is usually an operation that is not performed since, after doing that, part of the flexibility the database provides is lost.
- MongoDB offers the possibility to replicate the data it stores to different servers in order to increase availability and performance.
- Other than replication, MongoDB also provides *sharding*, a technique that consists in splitting the data in different machines to support very large datasets and offer higher throughput.
- It is possible to deploy MongoDB both in an on-promise server and cloud environment. In addition, a lot of distributions and versions are available, most of them also accessible as container images.

Considering all its properties, MongoDB should be definitely taken into consideration for all those applications in which is important to rapidly and reliably store and retrieve unstructured information.

As it happens with Kafka, deploying MongoDB could result in a complex distributed system with numerous components that is hard to be appropriately configured and managed. Therefore, MongoDB itself provides Kubernetes Operators and custom resources to automate and make its deployment as smooth as possible for the developers. In particular, two different official MongoDB Kubernetes Operators are available. The first one is community-driven and free but has some limitations (e.g. sharding is not available) while the other version, called enterprise, needs a subscription to be used but offers much more functionalities.

3.6 Spring

Spring [18] is a Java and Kotlin framework that mainly helps in the development of web applications. However, its core features and some others can be used by any Java/Kotlin application. It is the most known and used Java framework. Its popularity is due to the fact that it makes Java programming quicker, easier, and safer. It saves developers from writing a lot of boilerplate code (i.e. code that is often repeated with only little modifications) and provides many out-of-the-box solutions that are helpful for a vast variety of use cases. Moreover, it is supported by a huge worldwide community which makes finding support and resources related to every aspect of the framework really straightforward.

Spring was born with the idea of simplifying the often complex relationships that happen in OOP. In particular, it wants to facilitate the configuration phase (object instantiation and bindings) to allow developers to mainly focus on the logic of the code. Spring abstracts the concept of objects and provides some mechanisms to create a, even really complex, app by simply defining its components and declaring their configuration. In this way, it is the framework that will add later all the other elements needed to form a complete and working Java application.
In particular, the Spring core is based on three principles:

- **Inversion of Control** Instead of having the developer that uses, in their custom code, functions of the framework to solve general tasks, it is the custom code written by the developer that is called by the framework.
- **Dependency Injection** Whenever an object or function is developed, the dependencies it depends on are injected by the framework. This highly separates the concerns of using objects. Developers do not need to focus on how to connect the various objects but on what external functions the objects need and they will be provided by the framework. As a result, loosely coupled programs are obtained.
- **Aspect Oriented Programming** There are often some pieces of code that are needed in different parts of the program. For instance, a function to perform logging could need to be called in most of the objects of the application developed. For this reason, in order to avoid all these repetitions, Aspect Oriented Programming allows specifying the code to perform an often repeated action in a single place and then, through proper configuration, the framework spreads this piece of code where it is needed.

These properties represent only the principles on which Spring is built and they point out how, as already mentioned, this framework highly helps developers by providing

an easy, flexible, and fast way of writing Java and Kotlin applications. Nevertheless, there are much more other features it provides that are worth mentioning. Spring has really high performance; it is fast to start up, execute and shut down. All of this, while taking high care of security concerns. Vulnerabilities and issues are always immediately fixed. Moreover, many modules, both proprietary and third-party owned, are offered and some new ones come out very often. Spring is well integrated to work with most of the new techniques and technologies. It provides modules to deal with cloud computing, serverless applications, reactive programming, microservices, and much more.

Chapter 4

Design

This chapter starts with describing what is the problem at hand, firstly outlining what the general idea is and then imposing some specific requirements in order to deal with a real use case scenario. In particular, other than listing what are the actual functionalities and performance requested, it will be described how the series of constraints imposed leads to a problem that needs particular attention to be properly solved. At the same time, it will be briefly mentioned what are the last trends in the industry and will be specified that the objective of this thesis is not to only build a working system that meets the requirements; the goal is to do so with the most advanced techniques and cutting-edge technologies that a real competitive company would use.

The second section of this chapter will describe what are the design choices made, both in terms of techniques and technologies utilized. Especially, it will take into consideration why they have been chosen, what benefits they can provide, and what problems they can solve for the studied use case. After characterizing each component on its own, the complete and final architecture of the developed system is shown and analyzed. Finally, the last part of this chapter is dedicated to specifying the format and flow of the data managed by the application.

4.1 Requirements

As already briefly mentioned in the first chapter, the objective of this thesis is to build an application capable of reliably, efficiently, and securely streaming in real-time and storing data coming from EMF sensors. All of this with the goal of allowing human users to access such data to study the cellular coverage of the whole of California. In particular, the human users need to have the possibility to retrieve in real-time all the measurements just performed by all the sensors,

and, to conduct deeper studies, they must also be able to retrieve data related to measurements of the past.

However, the objective is not only to develop an application that can be utilized for this specific use case, on the contrary, the goal is to obtain a system as flexible, modular, and extensible as possible. In particular, the system should be easily adapted, with small changes (e.g. regarding the data format, the resources allocated, and/or some endpoints), to every application in which is requested to reliably and securely stream and store big amounts of data.

Another important requirement of the studied application is to be "industry ready". First of all, this means that the developed system must be accessible from the public internet (i.e. a valid and public IP address is needed). Second of all, the system needs to be secure.
Distributed system security is a really wide topic and a lot of different requirements can be requested for an application depending on the different types of attacks it deems to be protected from. Specifically, for the system studied in this thesis, it is, firstly, required for the app to only be accessible through HTTPS (HTTP on top of TLS) and by authenticated users. Secondly, all the communications inside the cluster (i.e. between components of the system) are required to be authenticated and encrypted.

General requirements have been discussed so far, but, as mentioned above, in order to meet actual performance requirements, a specific application needs to be taken into consideration. The use case of choice requires gathering EMF measurements that cover the whole of California. To do that, considering that California has an area of 423,970 km^2 and that every EMF sensor covers approximately 12 km^2, about 35.000 sensors are necessary. In addition, in order to properly study cellular coverage, measurements need to be collected from every sensor every 30 seconds roughly. This leads to a requirement of about 1200 requests per second that the application needs to be able to process. It is important to notice that human users' requests are not considered since it would be a much lower number with respect to the ones sent by the sensors.

A final requirement for the application is to obtain a product that would be affordable even for small businesses. Since a reliable and secure "industry ready" application is requested, actual costly resources need to be utilized to meet all the requirements and, in this context, it is easy to mistakenly over-allocate resources (i.e. use more CPUs, memory, and/or machines than the ones needed). The objective of this thesis is to perform numerous tests in order to allocate just the required resources needed to meet the requirements. To this end, some margins will

be considered in order to avoid anomalous crushes in the event that more requests than the ones predicted are received by the system.

4.2 Challenges and solutions applied

The list of requirements described in the previous section leads to a product that needs meticulous study in order to work properly. This is due to the fact that, when building such complex systems, it is common to obtain some properties at the expense of others. Thus, the main challenge has been to carefully choose every technique and technology utilized with the objective to get the benefits it brings without being too affected by the disadvantages its use could imply (e.g. some security protocols could drastically slow the system down).

To obtain the required adaptability, modularity, and extensibility, it has been chosen to use a Microservices architecture. Other than all its benefits, such as fast deployment and fault isolation, this type of distributed system makes adding and removing functionalities quite simple since it is possible to smoothly add and remove microservices. This property permits the system developed to be highly versatile. Most of the specific features needed by every single application can be added by simply adding services. The reason for this is that the core features, such as data gathering and storing, authentication, and security, are usually needed by all the applications belonging to the type addressed (i.e. securely and reliably store and stream data). At the same time, specific functionalities can be typically included modularly through single microservices.
In addition, this type of architecture is really useful to meet the high availability requested. Another benefit of a Microservices architecture is, as already mentioned, its modularity. This, other than helping in including and/or removing functionalities, eases adding replicas of single components. Such a technique is widely used to increase the availability of an element in a distributed system. Other than that, it increases scalability and performance. For these reasons, replicas have been widely utilized in the development of the application examined.

Nonetheless, even if the architecture chosen helps, it could still be hard to configure the system to add replicas. Moreover, other than different functions, various applications have different requirements in terms of performance (e.g. number of requests per second to be processed and level of availability required). Therefore, some sort of additional help needs to be added in order to ease configuring the system to meet the various performance requirements considered.
In this context, Kubernetes is the technology that perfectly matches the needs. Other than offering a lot of features that are really helpful for a Microservices

distributed system (this type of architecture and Kubernetes are often used together in modern complex applications), it really helps in configuring the different components of the system to meet the specific requirements the various applications request. With K8s it is as easy as modifying single lines in some .yaml files to change the replicas, CPUs and memory allocated to each element in the system.

The design choices discussed so far only regard software components. However, since an *industry-ready* system needs to be developed, an actual hardware infrastructure is required to run all the elements of the system. Specifically, supporting a microservices application that runs "on top" of Kubernetes requires a complex cluster of servers which can be really expensive and complex to set up and configure, especially for a small company that could not have the necessary workforce and capital to build and manage such a system. In addition, since the objective is also to create a product that can be adaptable to various applications and that, thus, needs to easily acquire and release resources depending on the requirements, it is required an infrastructure that can afford and dynamically manage extra machines in case they are needed. For these reasons, it has been decided to exploit the infrastructure already offered by a Cloud provider and thus a Cloud-native application has been developed. In particular, Google Cloud Platform was the provider of choice. Other than being one of the most renamed cloud providers that offers a huge infrastructure where is possible to essentially deploy every type of application, GCP has been chosen for its comprehensive Kubernetes integration and for its additional security and observability tools that are of big help for the correct development and testing of the system.

Specifically, for what concern the security requirements, Google Cloud Service offers its own service mesh that can be installed on top of a Kubernetes cluster. With proper configuration of this powerful GCP's tool, it is possible to meet all the security concerns mentioned in the previous section. Precisely, it provides support for mTLS between every component of the mesh and it allows to deploy an API gateway that is able to work as TLS endpoint and authentication system.

On the other hand, the extensive observability tools offered by Google are critical for correctly allocating the resources required by the system for meeting the performance requirements and for avoiding overpaying for allocated resources that were not actually needed.

4.2.1 Architecture

All the techniques and technologies presented up to this point regard only what is the high-level design of the product. The specific elements composing the system

have not been mentioned yet. Nevertheless, important design choices regarding every component of the system are essential to building a product that satisfies all the needs.

This subsection shows through a schema what is an overview of the architecture of the system. Successively, every element will be briefly described in order to give an outline of how the overall system works. A thorough description of every component will be provided in the next chapter.

Figure 4.1 illustrates the schema just mentioned. The first thing important to notice is that the cluster that hosts the application is everything contained inside the block "Google Cloud Platform" while everything outside of it represents the external actors (i.e. who or what accesses the application from outside the cluster). In particular, in this category, two different types of entities are depicted. The first one is constituted by the human users that are who access the measurements, both real-time and historical, in order to study the cellular coverage. The second group corresponds to all the sensors that continuously send EMF measurements to the cluster. In this context, the human users could be considered as who always "consumes" data and the sensor as what always "produces" it.

Moving now to the actual architecture, as already described, being a Cloud-native application, everything is deployed inside GCP and, in particular, inside a specific service that allows the deployment of a Kubernetes application called Google Kubernetes Engine (more details on its set up and configuration are outlined in the next chapter). On top of that, there is the service mesh that mainly supports the application with security features (this component is meticulously described in the sixth chapter that is completely dedicated to the security of the system).

For what regards the single components depicted in the schema 4.1:

- **Istio Gateway** It represents the entry point for all the external users (sensors and human users) that want to communicate with the application to receive or send data. This type of gateway is provided by Anthos Service Mesh and, other than working as API gateway, it is the main load balancer of the system. In addition, it offers several security features. All the communications between outside and the gateway use simple TLS, In this context, the Istio gateway works as the TLS termination proxy. Moreover, the gateway provides support for authenticating external users. Specifically, among all the possible authentication methods it offers, it has been chosen to use JSON Web Tokens (JWTs) that are issued by the Authentication Service.

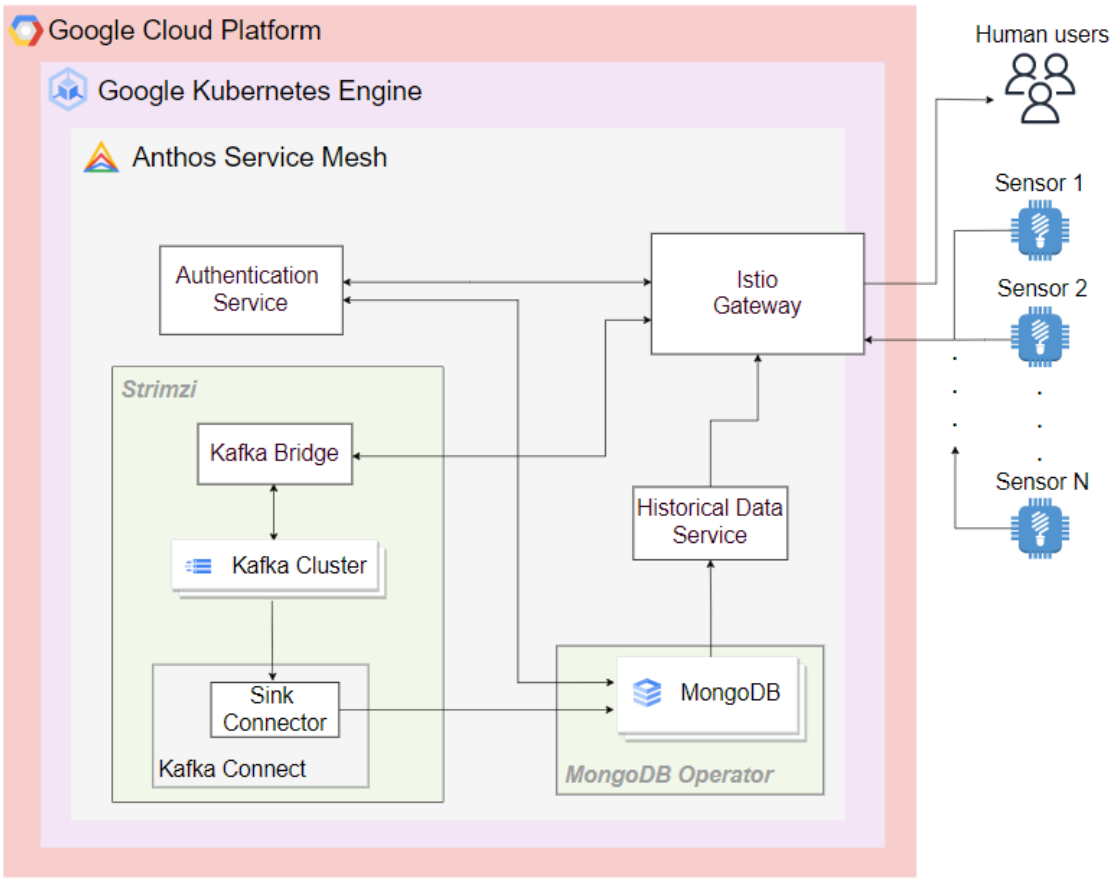

Figure 4.1: System architecture overview

- **Authentication Service** Spring Boot application that manages sign-up and log-in of external users. For every log-in request, it verifies if matching credentials exist and, if so, returns an according JWT.
- **Kafka** It works as the central component of the system. All the measurements are processed by Kafka. It has been chosen for its high performance (in terms of speed, scalability, and availability), vast user community, and integration with several databases and cloud platforms. It is deployed using Strimzi, one of the most famous open-source Kubernetes operators for Kafka.
  Kafka Cluster (the actual message broker) is composed of 3 brokers managed by Zookeeper. Kafka Bridge is used to allow to interact with Kafka using an HTTP-based interface. Finally, Kafka Connect is integrated with a MongoDB plug-in in order to send all received measurements to MongoDB for future access.

- **MongoDB** The database of choice is MongoDB. It has been selected for its flexibility, performance, scalability, simplicity, and cloud support. It is deployed using the MongoDB Community Kubernetes Operator. Specifically, a replica set composed of 3 different instances is used in this architecture.

- **Historical Data Service** Spring Boot application that exposes a HTTP interface to retrieve old sensor data from MongoDB.

4.2.2 Data format and flow

The data format schema of the measurements send and retrieved is indicated below (other data collections are used in the application and are described in the data storage section of the next chapter. They are not cited here as well because they are not as important and only represent "accessory" information):

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://my-exemple-id.com/measurement.schema.json",
  "title": "Measurement",
  "description": "A measurement from an EMF sensor",
  "type": "object",
  "properties": {
    "value": {
    "description": "The EMF value measured",
    "type": "number"
    },
    "timestamp": {
      "description": "The time in which the measurement has been performed",
      "type": "string"
    },
    "sensorId": {
      "description": "id of the sensor where the measurement has been performed",
      "type": "integer"
    }
  },
  "required": [ "value", "timestamp", "zoneId" ]
}
```

As is shown by the schema, no more information than the measurement value, sensor id, and timestamp is needed. This is because all the other data related

to the zone and/or the peculiar type of sensor can be retrieved, through the sensorId, from another collection that contains sensor information. In this case, it is necessary to perform a join operation. However, it is uncommon to read sensor details if compared to how often measurements (without such specific details) are read. Thus, performing a slow join operation from time to time is not a big issue.

Referring to figure 4.1, the direction of the arrows indicates the flow of data. In particular, as already mentioned, all the requests go through the Istio gateway. Specifically, every sensor sends its measurements to the gateway approximately every 30 seconds while human users request them from the system whenever needed. In both cases, all the requests must have a valid JWT in order to be authenticated by the gateway. For this reason, before sending/retrieving measurements, a JWT must be obtained from the Authentication service.

After going through the gateway and being authenticated, all the incoming sensor data is forwarded to Kafka Bridge which in turn forwards it to Kafka Cluster. The measurements, at this point, are "inside" Kafka's partitions, and from there they are shipped to two different entities. They are sent to MongoDB through the Kafka Connector, and, at the same time, they are forwarded to all the human users who are requesting them in real-time. In particular, human users can request real-time data by HTTP requests to the Kafka bridge that of course will have to pass through the gateway as well to arrive there.

For what instead regards the data flow when referring to measurements retrieval, other than the real-time one just described, the human users interested in obtaining historical data contact an HTTP interface provided by the Historical Data Service that in turn talks with MongoDB and retrieves the asked measurements. Also in this case, all the requests go through the gateway that takes care of all the security concerns and forwarding rules.

Chapter 5

Components

This chapter is dedicated to a complete description of the developed system's components. Particular focus will regard how every element has been set up and configured and what specific functionalities it offers. Specifically, the objective of this chapter is not to be a tutorial on how to build a cloud-native application. Instead, it wants to point out how every component has been configured and linked to the others to obtain a complete product capable of efficiently satisfying the requirements.

Before dealing specifically with every single component, it is necessary a description of how the Google Cloud Platform and Google Kubernetes Engine have been utilized and set up in order to host the application with all its components.
GCP offers a complete environment for developers that contains a series of interfaces dedicated to all the possible services provided. Other than this, it gives the possibility to access a powerful IDE from where developers can code and run commands to deploy pieces of software to Google Cloud. Moreover, its integrated bash offers native support for a lot of command-line tools useful to directly contact GCP's services.
In particular, after having created a project and having activated the required APIs to use GKE, the command used to deploy the Kubernetes cluster that will host the studied application is:

```
1  $ gcloud container clusters create-auto thesis-project --region=us-
     west2 sh_id=proj-$PROJECT_NUMBER"
```

This command creates a Kubernetes cluster of name *thesis-project* in the region *us-west2* and with the autopilot mode of operation. The region simply specifies where the cluster's resources are located. The western United States region has been chosen to be as close as possible to where the EMF sensors are situated. In

this regard, it is important to point out that the application can be contacted from everywhere in the world regardless, independently of the specific region selected. The only difference is that the further from the west coast of the United States, the higher the delay.

It is possible to use a multi-cloud environment where resources are located in different regions, but this highly complicates the system and only gives advantages in terms of how fast human users can access the measurements, a requirement that is not so strict. It is enough that human users can access the data with a delay of a few hundred seconds, something that is obtained with the cluster located in a single region.

For what regards the mode of operation, the autopilot mode has been selected in order to let GKE automatically optimize the cluster's resources allocation. This does not mean developers do not need to care about that. It only means that the level of abstraction is different. Developers only need to manage resources at the Kubernetes level and not also at the Google Cloud level (i.e. they do not have to contact GCP to request resources, Google allocates them automatically depending on the resources requested through K8s).

After having created the cluster, the service mesh needs to be installed. This process will be described in the next chapter dedicated to the security of the system. The important thing to understand now is that every component that will be deployed on the system comes with a proxy that provide some security features to it.

Before starting to describe every component of the system, Figure 5.1 shows what is the workload (i.e. applications running on K8s pods) of the completely deployed and ready-to-use product. In particular, the picture is a snapshot of the web page section dedicated to GKE to check the status of all the workload components. From this view, among other information like the type of the workload and namespace it belongs, it is possible to immediately see, from the *Pods* column, how many replicas are associated with each element and, in addition, by clicking on one of them, it is provided access to all the details related to every single element such as logs, errors, containers, associated configuration files, and more. This page has been of fundamental importance for the development of the system since it provides a really straightway and complete way for debugging the application, allowing an immediate understanding of where and when something is not working the way it should.

On the other hand, figure 5.2 displays the services overview. It shows a list of all the Kubernetes services with their status and endpoints. By clicking on every one

Name ↑	Status	Type	Pods	Namespace	Cluster
authentication	OK	Deployment	1/1	default	thesis-project
historicaldata	OK	Deployment	1/1	default	thesis-project
istio-ingressgateway	OK	Deployment	3/3	default	thesis-project
mongodb	OK	Stateful Set	3/3	default	thesis-project
mongodb-arb	OK	Stateful Set	0/0	default	thesis-project
mongodb-kubernetes-operator	OK	Deployment	1/1	default	thesis-project
my-bridge-bridge	OK	Deployment	4/4	default	thesis-project
my-connect-cluster-connect	OK	Deployment	2/2	default	thesis-project
my-kafka-entity-operator	OK	Deployment	1/1	default	thesis-project
my-kafka-kafka-0	Running	Pod	1/1	default	thesis-project
my-kafka-kafka-1	Running	Pod	1/1	default	thesis-project
my-kafka-kafka-2	Running	Pod	1/1	default	thesis-project
my-kafka-zookeeper-0	Running	Pod	1/1	default	thesis-project
strimzi-cluster-operator	OK	Deployment	1/1	default	thesis-project

Figure 5.1: GKE's workload overview page

of them, all the details related to that specific service are displayed.
Even if a lot of services have an IP address associated, this does not mean they can be accessed from everywhere. In particular, the only publicly accessible service is the Istio gateway (named *istio-ingressgateway* in the picture) that exposes three different endpoints out of which only the one with port 443, associated with the TLS connection, is the one that actually allows contacting the application. The other two ports are used only for testing and health-checking purposes.
The IP associated with all the other services are only accessible from within the cluster and are, thus, not public.

5.1 Data streaming

As already mentioned, the data streaming platform of choice for the developed application is Apache Kafka. All the Kafka components have been deployed thanks to the help of the Kafka Kubernetes operators and custom resources called Strimzi. Setting it up is as easy as running a simple command that installs all the required elements to use Strimzi in the cluster. After having done that, it is possible to deploy the custom resources related to Kafka. In particular, the main component necessary to make everything work is the group of brokers and ZooKeeper. To avoid confusion, this group of elements will be called "Kafka Cluster" in the following.

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
authentication	OK	Cluster IP	10.16.130.49	1/1	default	thesis-project
historicaldata	OK	Cluster IP	10.16.128.142	1/1	default	thesis-project
istio-ingressgateway	OK	External load balancer	34.145.93.180:15021 34.145.93.180:80 34.145.93.180:443	3/3	default	thesis-project
kubernetes	OK	Cluster IP	10.16.128.1	0/0	default	thesis-project
mongodb-svc	OK	Cluster IP	None	3/3	default	thesis-project
my-bridge-bridge-service	OK	Cluster IP	10.16.129.44	4/4	default	thesis-project
my-connect-cluster-connect-api	OK	Cluster IP	10.16.131.187	2/2	default	thesis-project
my-kafka-kafka-bootstrap	OK	Cluster IP	10.16.128.28	3/3	default	thesis-project
my-kafka-kafka-brokers	OK	Cluster IP	None	3/3	default	thesis-project
my-kafka-zookeeper-client	OK	Cluster IP	10.16.131.35	1/1	default	thesis-project
my-kafka-zookeeper-nodes	OK	Cluster IP	None	1/1	default	thesis-project

Figure 5.2: GKE's services overview page

Instead, whenever the term "cluster" is used without being preceded by "Kafka", the entirety of the application hosted on the Google Cloud Platform is meant.
Other two components offered by Strimzi that have been widely used are Kafka Connect and Kafka Bridge, their setup and use are described later in this chapter. Even if Strimzi is completely integrated with Kafka Connect (i.e. it offers the possibility to deploy it through K8s custom resources), it is a feature naively offered by Apache Kafka. On the other hand, Kafka Bridge is provided by Strimzi which extends what is natively provided by Kafka.

5.1.1 Kafka Cluster

The Kafka Cluster is deployable through a custom resource called "Kafka". Specifically, it does not only create a Kafka Cluster with as many brokers as requested, but it also deploys ZooKeeper and two other operators that are useful to manage Kafka users and topics in a declarative way. The "Kafka" YAML file allows requesting specific resources for all the elements (i.e. Kafka Cluster, ZooKepers, users operator, and topic operator) separately.
The level of customization offered by Strimzi is impressive. It is possible to configure a lot of aspects of the Kafka Cluster. For example, other than the number of brokers and some characteristics related to ZooKeepers, Strimzi provides a way to specify the exact version of Kafka, the authorization and authentication techniques used, the type of storage attached, and so much more.

Specifically, some of the properties important for the thesis regard the type of listeners specified (i.e. how Kafka is exposed and accessible from the outside). In this context, it has been sufficient to expose the message broker internally

to the Kubernetes cluster and without any form of encryption. That is because such security properties are provided by the service mesh that deploys its proxies alongside every component in the cluster.

Another important property is obviously the number of brokers. It has been chosen to deploy three brokers. This number is really common in production environments since it is the minimum number that makes really unlikely to have all of the replicas down at the same time.

Other than the properties related to the actual deployment of the Kafka ecosystem, it is obviously possible to configure internal characteristics of the even streaming platform, such as log and message format, timeouts, consumer and producer-specific properties, and so on. In particular, a fundamental configuration property for this thesis project is the log retention that has been set to one minute. It is so important because a lot of data is produced and it would be easy to quickly fill Kafka's memory. Deleting the measurements from Kafka is not a problem because they are stored in the database. The retention factor could have been lower but a minute has been selected to be sure all the produced measurements are written in the database and to allow every human user that is requesting the data in real-time to receive them.

Another major characterization of a Kafka ecosystem surely regards its topics. The Strimzi topic operator allows to define them declaratively through YAML files. The corresponding resources created can be modified on the fly both by updating the YAML files associated and by directly contacting Kafka's APIs. The most important topic used in the studied application is the *signals* topic. In particular, it has been configured to have 3 different partitions and a replication factor of 3. This decision permits the topic to be as available and reliable as possible considering the cluster setup.

5.1.2 Kafka Connect

Kafka Connect is an additional component that can be integrated with the Kafka Cluster in order to provide two functionalities: writing data coming from an external source to one or many Kafka topics and reading data from one or many topics and writing it to an external system. How to interact with the different external entities depends on the external entity itself. However, Kafka Connect was born with the idea of providing a uniform and standard interface to send and retrieve data, independently of the source/destination. For this reason, every time the developers want to integrate a new system with Kafka Connect, they need to use a plugin (called connector) that specifically orchestrates how the data can be

moved in and out of Kafka to go or come from/to the other system. Fortunately, a lot of modern and famous databases, message brokers, and big data platforms already provide their own plugins. In this case, the developers only need to install and configure the required plugin for their system and everything is ready to work. It has already been mentioned how the data can go through Kafka Connect from a topic to an external source or vice versa. In the first case, the connector is called *source connector* whereas in the second case is called *sink connector*.

Kafka Connect is useful in the application studied to retrieve the measurements from the signals topic and write them on a MongoDB collection. Hence, what the application requires is a sink connector. Luckily, MongoDB provides both sink and source connectors free of charge.
After a "KafkaConnect" resource with the proper plugins installed and two replicas for improved performance has been deployed on GKE. Another custom resource needs to be defined in order to make the connector work. Such resource is called "KafkaConnector" and the specific YAML file describing its configuration for the deployed application is shown below.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-mongo-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: com.mongodb.kafka.connect.MongoSinkConnector
  tasksMax: 2
  config:
    topics : signals
    connection.uri: mongodb://my-user:my-user@mongodb-0....
    database: myDb
    namespace.mapper : com.mongodb.kafkaconnect.sink.namespace.
   mapping.FieldPathNamespaceMapper
    namespace.mapper.key.database.field: "db"
    namespace.mapper.key.collection.field : "topic"
    key.converter.schemas.enable: false
    value.converter.schemas.enable: false
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
```

What it is important to notice in this file is, first of all, the number of tasks that can be run in parallel (it has been chosen equal to the number of replicas the Kafka Connect has) and, second of all, everything inside the "config" section. In

particular, it specifies:

- the topic from where to retrieve data
- the address of the MongoDB database management system
- on which specific database the data needs to be written (this property is considered only if the database is not specified with the mapper as described below)
- how to map some fields' values of the key of the messages written in the topics considered to the name of the database and collection to use to write the data on MongoDB. In particular, in the YAML file considered, it is specified that the fields of the key named "db" and "topic" indicate where to search for the database and topic name, respectively
- how to convert the data from the topic to the MongoDB collection. In the studied case, a simple JSON converter without referring to any particular schema is used

5.1.3 Kafka Bridge

In order to produce and consume messages to/from Kafka it is necessary to use a specific protocol that is provided by Kafka itself. However, since the objective of this application is to be as adaptable and flexible as possible, having clients that need to follow a so specific protocol to perform what is one of the most important functions of the system (i.e. producing and consuming data) would be not ideal. For this reason, it has been decided to take advantage of one of the additional functionalities that Strimzi offers to Apache Kafka that is called Kafka Bridge, This component provides an HTTP interface, on top of the proprietary Kafka protocol, to contact the Kafka Cluster. In this way, clients do not need to interpret the Kafka-specific protocol anymore, it is enough to simply send HTTP requests to Kafka Bridge that will interpret and transform them to be compatible with Kafka.

On the other hand, as will be described in more detail in the testing chapter, this component represents one of the main bottlenecks of the system. Consequently, 4 different replicas are deployed to avoid killing the performance. However, whenever the use case analyzed requires higher speed, it could be considered to remove this component and "lose" the interoperability obtained with the HTTP interface to obtain a faster system and also save some hardware resources.

5.2 Data storage

As described in the design chapter, the database management system of choice is MongoDB. It has been deployed with the use of the open-source MongoDB Kubernetes Community Operator. In particular, a replica set composed of three different replicas has been deployed to obtain higher availability and redundancy. The selected data storage system contains all the measurement data produced and some additional information related to the EMF sensors and external users of the system. In particular, two different databases are hosted on MongoDB.

The first one is called "authentication" and contains a collection that stores all the data regarding the authentication process. More specifically, the collection contains the usernames, type (i.e. human user or sensor), and passwords of all the registered users. It is possible to register new users only through one of the system administrators that have to insert the user's credentials in the collection manually. This is required since it should not be allowed for everybody (it is enough to know the public IP of the application to access it) to create an account and then access the measurements. Only the specific engineers that are hired to study the cellular coverage of California and the sensors will acquire their credentials in the system. Having the system administrator register all the external users "by hand" is not a big problem because the sensors almost never change and only a few people will access the data thus not a lot of management for them is required. The collection is accessed by the Authentication service that, by checking the users' credentials, creates conforming JWTs that will be later utilized to authenticate the external users.

The second database, called "measurements", contains two collections. One that stores information related to the characteristics of each sensor, such as sensor id, position, and model. This collection is called "sensors". The second one, instead, is the "signals" collection and stores all the measurements produced. The schema of data it contains has already been presented in the previous chapter. The signals collection is definitely the most important one in the system and will store much more data with respect to the other collections.
Both collections in the measurements database can be accessed by external users through the HistoricalData service which is described in the next section.

A central decision that has been made in the design of MongoDB is to divide the data into two different databases. This is really important to be compliant with one of the major principles in Microservices architectures that is: every microservice owns its data. This means that two different components of the system cannot access the data of the other one without going through the component that actually

owns the data. Such a principle lowers what is the coupling of the system. Every service needs to be as independent as possible from the others in order to be more scalable, fault isolated, and flexible.
Even if in a lot of Microservices applications is common to have two completely different data storage systems (e.g. a SQL and a NoSQL database), it has been decided to go for MongoDB as the only data storage system because, firstly, the Microservices principle is not violated. The two databases are independent of each other even if hosted in the same MongoDB replica set, the services that access them cannot access any other database other than their own, and MongoDB has been deployed as a replica set and thus it is difficult for it to crash and lose all the data. Secondly, it has been thought that having two distinct database management systems for the two different databases was not a good idea for the following reason: the authentication database contains only little data and a single collection, it would have been overkill to have an entire DBMS only for such a small mole of information.

5.3 Historical Data Service

The Historical Data Service offers an HTTP interface for human users contacting the application from outside to access the measurements and sensors data. Specifically, it provides different endpoints that allow retrieval of information from both the "signals" and "sensors" collections stored in the "measurements" database.

The service has been developed using the Spring framework for the Java programming language. In particular, the only two modules utilized are Spring Boot and Spring MongoDB. Any other module was not required because most of the needed features are already provided by external components such as GKE, the service mesh, and the API gateway. The only task for the Historical Data Service is to connect to MongoDB and retrieve all the data that the users request when contacting the service. In particular, the *Entity*, *Repository*, and *RestController* functionalities of Spring have been used to access the database, retrieve the data, and expose the endpoints, respectively.

The deployment of the service has been performed with the help of Docker. Particularly, after having tested the system locally, a Docker image has been defined and pushed to a private Docker Hub repository. In this way, to have the service up and running in the Google Kubernetes Engine cluster, it is enough to create a K8s Deployment that works with the aforementioned image and expose it to the other entities in the cluster (it is the API gateway that exposes the service to the public Internet) through a Service Kubernetes resource.

5.4 Authentication Service

The Authentication Service provides an HTTP interface to obtain JWTs in order to be authenticated by the API gateway and get access to the main functionalities of the application. Specifically, the main endpoint exposed by the Authentication Service, called */login*, expects to receive in every request a username and a password. After having verified this information is actually present, it tries to match such credentials to a document in the "users" collection of the "authentication" database. If such a document exists and both username and password match, the user is provided with a valid JWT. In particular, a possible example of a decoded JWT is shown below:

```
{
    "header": {
        {
          "alg": "RS256",
          "kid", "myKeyId"
          "typ": "JWT"
        },
    "payload": {
        {
          "sub": "myUsername",
          "category": "human user",
          "issuer": "thesis@secure.com",
          "iat": 1516239022
          "exp": 1516326115
        }
    }
}
```

As it is possible to see, other than some general information such as the algorithm used and the "typ" (type) field, the token indicates the username of the authenticated entity ("sub" field), if its a human user or a sensor ("category"), the issue and the expiration dates ("iat" and "exp", respectively), and some fields specifically dedicate to the signer of the token that are named "kid" and "issuer". These fields are used by the API gateway to verify the token is valid (how this is performed is described in the next section dedicated to the API gateway). In this context, it is important to point out that it is possible to use JWTs in the developed application only because all communications are properly encrypted by means of TLS. Hence, the exchanged JSON Web Tokens cannot be read by unauthorized entities.

As happens with the Historical Data Service, the Authentication Service is developed with the use of the Spring framework for Java. Also in this case the Spring Boot and MongoDB modules, and the *Entity*, *Repository*, and *RestController* functionalities are used. Moreover, similarly to the component described in the previous section, the Authentication Service is deployed on Google Kubernetes Engine using a Docker image pushed on Docker Hub and pulled by a Deployment that is exposed through a Service.

5.5 API Gateway

The API gateway represents the only component in the system from where external users can access the application. In particular, its primary role is to expose the application on the public Internet and, after receiving a valid request, forward it to the right microservice. Nonetheless, the API gateway offers some other important functionalities. First of all, it is the main load balancer of the whole cluster. As already specified, all the requests go through this component and are thus load balanced by it. Secondly, it manages different security mechanisms. It handles the TLS connections with the external users and also authenticates them through validating the JWT present in every request (but the one to obtain the JWT itself).

The API gateway deployment has been possible thanks to the support of the Anthos Service Mesh. which is based on the Istio Service Mesh (for this reason, the gateway is often called "Istio gateway"), which provides capabilities to make use of this powerful service. In particular, the gateway has been deployed on the studied cluster with the default configuration provided by Istio. This includes, other than a Deployment and a Service, some additional Kubernetes resources that enhance the overall system such as a *HorizontalPodAutoscaler* that scales the 3 default replicas of the gateway up to 5 when the average utilization is above 80%.

5.5.1 Endpoints

After the gateway has been deployed, it is necessary to further configure it to set up the routing rules and security features specifically for the developed system. More details on how the security features provided by the gateway have been configured are specified in the next chapter. Instead, for what regards the first task which is the configuration of the routing rules, it is enough to define a single Kubernetes custom resource provided by Istio that is called "VirtualService". Specifically, the part of the used YAML file that specifies which ones are the requests to forward to the Historical Data Service is shown below:

```
....
....

http:
  - name: "historicaldata-routes"
    match:
    - uri:
        prefix: /historical/
    rewrite:
      uri: /
    route:
    - destination:
        port:
          number: 8080
        host: historicaldata

....
....
```

Every entry in the "http" section of the file specifies a particular routing rule. This means that other entries are present in the complete file that indicate all the rules for the other services. However, it is not necessary to show and describe all of them because they are all similar to the Historical Data one, what changes is only related to the port and service names.
Each rule has a name associated with it ("historicaldata-routes" in the illustrated case) and, after that, it indicates what uri prefix such a rule refers to and where to forward the matched requests (indicated in the destination field). Moreover, following what is written in the "rewrite" field, the gateway modifies every request to remove the first part ("historical/" in the shown example) because, in the designed system, that section of the uri is only useful for the gateway itself to understand where to route the request. The service which will receive the request does not understand that prefix that is, therefore, removed.

Chapter 6

Security

This chapter describes what are the security features that the application possesses and how they have been set up to work with the developed system hosted in the Google Kubernetes Engine cluster. In particular, the technology that has been used is a service mesh (i.e. an infrastructure that manages the communication between services). Every packet that goes from one service to another passes first through a proxy, present in each service of the cluster, that manipulates such a packet to introduce some security features. Specifically, it has been taken advantage of the Google Cloud service called *Anthos* that offers a service mesh specifically for GKE that is based on a powerful and famous service mesh for Kubernetes called Istio.

Before describing how the service mesh has been installed in the developed application, it is important to understand some details about the Istio and Anthos Service Mesh architecture. It is composed of two elements: the data plan and the control plane. The first component represents the set of all the proxies installed on each service of the system whereas the second element is a sort of coordinator that manages all the proxies.
Google provides two ways of deploying its Anthos Service Mesh. The first one is called *Managed* and its peculiarity is that Google handles a lot of work for the developers. It manages the updates, security concerns, and scaling for them. In this way, developers can focus more on the actual application development and let Google handle most of the configuration and management required by Anthos. On the other hand, the second type of deployment for the Anthos Service Mesh is the so-called *In-cluster control plane* where the users have full control of the installation, scaling, and upgrade of the mesh.

The deployment mode of choice for the studied system is the Managed mode. This type of deployment provides several features that are enabled by default. Some of them regard traffic logs, cloud monitoring and tracing, and several security

features. In particular, what is really important for the developed application is that all the data are authenticated and encrypted. In this context, it is necessary to distinguish inner and outer mesh communication. How this has specifically been managed is described in the two sections of this chapter.

Before starting a detailed description of the system's traffic security, briefly specifying how the Managed Anthos Service Mesh can be installed in a GKE cluster and how it works is important to understand the overall mechanism behind everything that is provided by Anthos and Istio.
Enabling the Managed service mesh requires registering the Kubernetes cluster to a *fleet* (i.e. a group of clusters. It is useful for multi-cluster deployments and so it will not be relevant for the studied system) where the service mesh has already been enabled. Then, it is required to specifically activate the automatic management of each individual cluster and, after that, the mesh is up and running. In particular, every Kubernetes component that is deployed to a namespace (i.e. a way for grouping K8s resources) labeled with *istio-injection=enabled* will be injected with a *istio-proxy* and thus will receive all the features the mesh provides.

6.1 Inner-mesh communication

The inner-mesh communication regards all the communications happening between two or more microservices of the application. Differently from what takes place in a simple application deployed in a small private cluster, where usually only a few entities communicate with each other through a private network, in a Cloud-native application, the components are deployed on a third-party cluster and can communicate through the public Internet. In this context, other than a cluster in which developers do not have full control, an infrastructure where data ca be exchanged through a public network is highly prone to attacks. For this reason, it is fundamental to protect every communication of the application, even if between components inside the cluster.

Fortunately, the Managed Anthos Service Mesh provides by default mTLS for all the communications happening inside the GKE cluster. As already specified in the second chapter, mTLS offers encryption of all the data exchanged, and, moreover, both parties involved in the connection are authenticated through digital certificates. The service mesh control plane automatically manages the distribution of the certificates among the various elements inside the cluster.
Even if mTLS is enabled by default, all the services in the mesh still accept both plain-text and encrypted traffic. For this reason, it has been imposed, through the *PeerAuthentication* Kubernetes policy, that only encrypted traffic is acceptable.

6.2 Outer-mesh communication

It has already been mentioned that the Istio gateway is the only way external users can access the system. In addition to its functionalities as load balancer and requests "router", the gateway manages all the security features that regard the communication between the users and the system (i.e. the outer-mesh communication). Specifically, what has been used in the developed system is the Istio-provided ingress (for traffic entering the cluster) gateway. This element is composed of a standalone *istio-proxy* which is the component that runs on every service and represents the data plane of the service mesh and that, by itself, can work as a gateway. It supports a lot of security features and it has been used in the studied system to provide encryption, authentication, and access control for the outer-mesh communication.

6.2.1 Encryption

In the previous chapter, it has been described how the API gateway has been configured to follow some specific routing rules. Similarly, it is necessary to configure the gateway to expose it and allow users to access the application. This can be performed with the help of a Kubernetes-specific YAML file. The main part of the one used in the developed application is depicted below:

```
....

  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: https-credential

 ....
```

The file configures the gateway to be accessible through the HTTPS (HTTP on top of TLS) protocol on port 443. In addition, it specifies some peculiar properties for the protocol utilized. The *SIMPLE* in the *mode* field solely represents that the simple mode of operation of TLS is employed. This means that only the gateway will be authenticated by the protocol. In particular, the digital certificate and associated private key needed are specified in a Kubernetes secret that must be indicated in the *credentialName* field of the YAML file.

6.2.2 Authentication and access control

For what are the requirements of the application, it is not sufficient to have a simple TLS connection for outer-mesh communications. No client authentication is supplied. For this reason, it is necessary to provide some sort of authentication to place side by side with the simple TLS protocol. Fortunately, the Istio gateway also offers help for what regards the authentication process. In particular, it is compatible with JSON Web Tokens. Hence, it is possible to combine the Authentication Service (that provides JWTs to signed-up users) with the gateway in order to obtain a complete authentication system.

More specifically, every external user needs to obtain a JWT from the Authentication service before having complete access to the system's functionalities. However, every request must go through the Istio gateway to reach such a service and, at the same time, the gateway needs a valid JWT to allow requests inside the system. Hence, some access control mechanism must be imposed to deny access to any requests without a valid token but the ones for the Authentication service. It is possible to define an *AuthorizationPolicy* YAML file to impose such a constraint. In particular, the one utilized in the studied application simply specifies that all the requests directed to a URI that does not start with "/auth" (the ones for the Authentication service) need to contain a valid JWT.

The only remaining configuration to perform in order to have a working and complete authentication system regards how to validate the JWTs. Again, as happens for most of the components in the system, it is possible to define a YAML file to provide the needed configuration. Particularly, to properly instruct the Istio gateway to validate the tokens, a *RequestAuthentication* Kubernetes resource needs to be defined. It specifies, firstly, that a JWT authentication is utilized and, secondly, who is the issuer of the tokens and what is the public key to verify the signature associated with each token. When the Authentication service is created, it is given a private key to sign every issued token. Such a key, as with every private key, comes with an associated public key that is exactly the one that must be indicated in the *RequestAuthentication* configuration file.

Chapter 7

Tests and Results

This chapter is divided into two sections. The first one regards a brief description of what are the services provided by the Google Cloud Platform to help developers monitor and test their applications. Moreover, it illustrates, through one of the dashboards that GCP offers, what is an overview of some of the resources utilized by the cluster that has hosted the studied application.
On the other hand, the second section of this chapter shows and analyzes what are the results of the tests that have been performed on the system. Particular focus is placed on how the developed application performs when it processes around 1200 requests per second which is the typical load in a real-case scenario. In this context, different graphs, mainly regarding resource usage, are presented and analyzed with the objective to demonstrate that only the right amount of resources needed for the application to work properly have been allocated. This section also reports the cost needed to run the final product on GCP.

7.1 Google Cloud Platform monitoring

Google offers the users of its Cloud services a whole section dedicated to monitoring. In order to provide such a functionality, most of the services are always "observed" by GCP that keeps track of a lot of information regarding the operations that such services perform. Specifically, all this data can be explored from the Google's monitoring page which offers a huge amount of metrics regarding a lot of aspects of the cloud functionalities used. Some examples are: bytes received by the deployed system, resource consumption, errors, and costs. In addition, Google gives the possibility to create dashboards (i.e. a set of metrics shown, in form of graphs or tables, in a single place) that are really useful to have a global view of what has been happening in the system. GCP also suggests some already-created dashboards that refer only to what are the services that the users have actually been utilizing.

For instance, a dashboard that was suggested by Google while the studied application was running is the one depicted in figure 7.1.

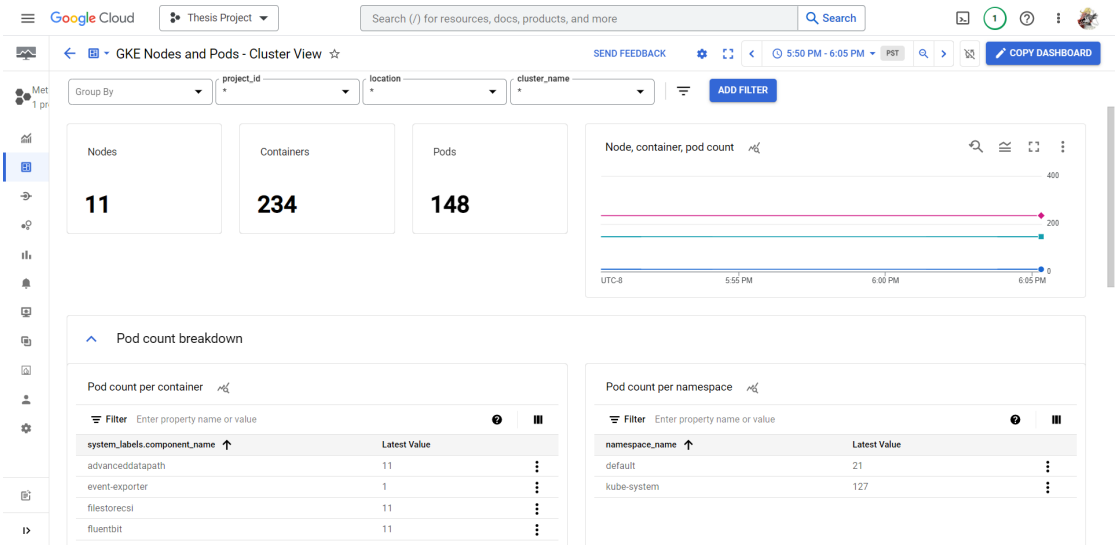

Figure 7.1: GKE Nodes, Containers, and Pods

The figure shows an overview of the deployed Google Kubernetes Engine cluster that focuses on the number of nodes, containers, and Pods running. The displayed numbers are so high because, other than the fact that every deployed component comes with the proxy sidecar and thus doubles the count, the dashboard also considers the resources that are automatically deployed by Google to make the Kubernetes cluster and the Google Cloud services work. Examples of the services that run without direct deployment by the developers are: some monitoring, logging, and security tools, some Anthos components, K8s control plane, and all the services needed to manage and interact with the Google Cloud Platform in general.

7.2 Performance analysis and bechmarking

Before presenting and describing the graphs representing the state of the system when dealing with a typical real-life load, it is important to mention what have been the tools used to actually test the application. In particular, the studied system has been tested throughout the entire development, at first, to understand if the application was working properly and later, when the complete system was deployed, to actually examine if the final product could meet the imposed requirements. In the first phase, *cURL*[19] (a really popular command line tool that allows sending HTTP(S) requests) has been mainly used to test the system every time a new functionality was added or an existing one was modified. Later, when the

application was finished, *loadtest*[20] (another command line tool to send HTTP(S) but more specific to test high and real-world loads) has been utilized to properly scale the system to be able to process all the requests necessary for the application to function appropriately. To this end, it is important to point out that, firstly, even if the system needs to process that specific load, it has not been forgotten, during testing, that it needs to do so while caring about the other requirements in terms of speed, resource allocation, security, and reliability. Secondly, when the term "scale" is used, it is meant for both vertical and horizontal scaling. Hence, the application has gone through a process that considers both increasing the memory and CPUs of single components and deploying more replicas of the same component.

It has already been specified different times how particular attention has been put to not over-allocate resources. In order to do so, a gradual testing process has been performed on the final system. At first, every component of the architecture was given only the minimum quantity of resources and few req/sec (around 400) was sent to the system that was able to process all of them without crashing. The load was then increased to 800. In this context, the system started to have some problems and, even if not immediately, the it stopped working. For this reason, thanks to the metrics provided by GCP, more resources (in terms of CPUs and memory) were allocated to the components of the system that crashed or that were about to. This process continued until the required 1200 req/sec where reached. However, reaching the goal with this type of approach has the risk of over-allocating resources in the "last step" of the incremental procedure. Therefore, other tests were performed to see if some of the CPUs and memory allocated to some components could be decreased without impairing the application.

It is of fundamental importance to point out, considering that in real-case scenarios it could happen that more than the usual load is witnessed, that further additional test cases where the requests per second were higher than the usual value have been considered, and the system, even if with some delays, was still able to work as desired. In particular, the application was able to work with around 100 req/sec more than the normal load.

7.2.1 Bytes processed

To give more context to what is the actual load that the system has to deal with when all the sensors send their measurements every 30 seconds and thus produce the renowned 1200 requests per second, figure 7.2 shows what are the bytes associated with such a number of HTTP requests.
As it is possible to see from the picture, more than 10 Mega Bytes per second are processed by the application when it gathers data from all the sensors. Requests

from human users are not considered because really few if compared with the ones sent by the sensors.

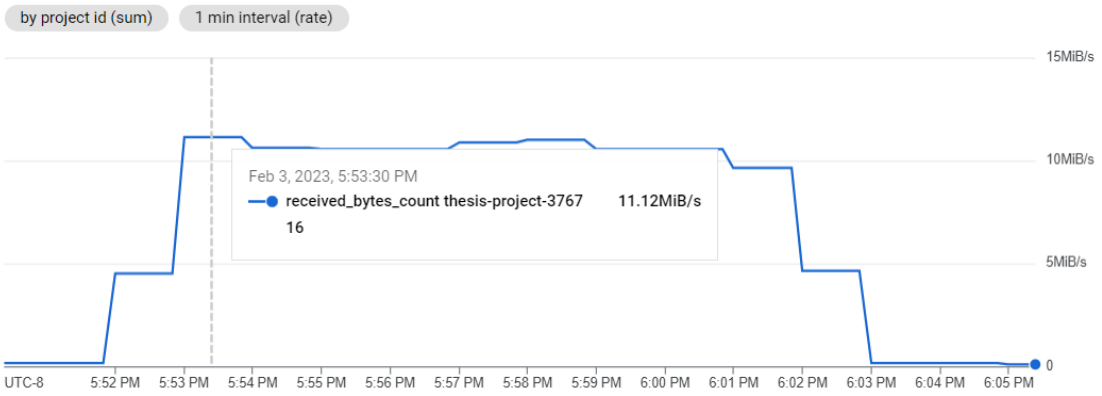

Figure 7.2: Bytes associated with 1200 req/sec.

The bytes that can be processed are only to be considered as a piece of information to relate to the total cost of the application and not as its maximum capacity. It is possible to have a system capable of processing more data if more resources are strategically allocated, but this comes with higher costs and would not make sense for the chosen use case.

7.2.2 Cost

The total cost of the Google Cloud services used, which mainly include, as not free components, Google Kubernetes Engine and Anthos Service Mesh, is around $120 per day. One of the requirements of the application was to be affordable, even for small companies. An average cost of $3600 per month to rent an entire cluster and host such a complex application is something that most companies should be willing to pay and afford.

7.2.3 CPU utilization

Figure 7.3 represents the CPU utilization of the various architecture components when the application processes the usual 1200 requests per second. As happens with the previous graph, only a time frame of 10 minutes is shown for visualization purposes. Nevertheless, several other tests have been run where longer time frames, and more req/sec, were considered and the application was still able to work satisfactorily.
Specifically, the picture shows for every component of the architecture what is the percentage of the CPU allocated that is actually being used (e.g. 1 means that

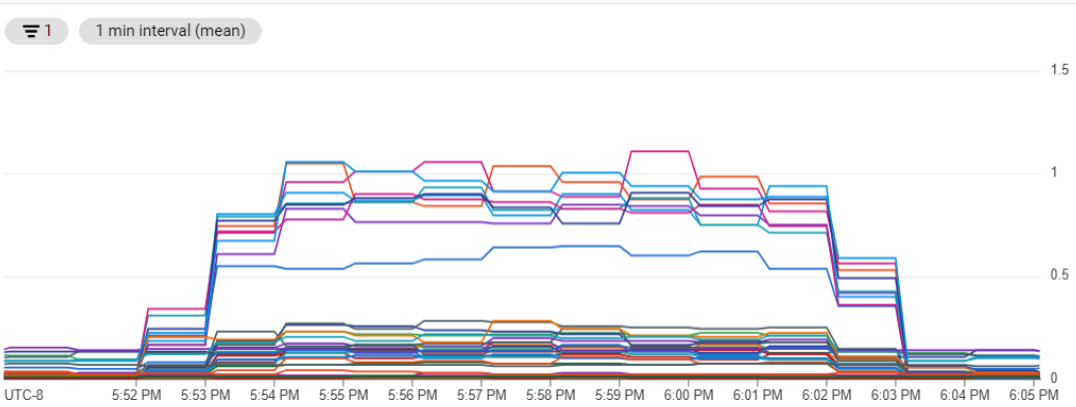

Figure 7.3: CPU utilization with 1200 req/sec.

all the requested CPU for that component is being used at that moment). In this context, it is important to notice from the graph that for some elements the limit of 100% (1) is exceeded. This is because Kubernetes allows for some short period of time to use more CPU that the one allocated. However, if this happens for too long and with a value that goes too much over the threshold (that depends on the specific architecture deployed) the component will crash. As will be described in the next subsection, the same cannot happen for memory allocation. In this case, every component cannot get more than 100% of the requested RAM.

As was expected and as is also illustrated in the graph, only some elements of the system witness high load while others only use a small portion of the CPU allocated for them. This is totally normal since the heavy operations that the application performs are streaming and storing measurements. Such activities only regard some of the components of the system while others are only used to provide "accessories" functionalities. For instance, the Authentication and HistoricalData Services only work when the application needs to provide some JWTs and historical data, respectively. Both requests are performed rarely if compared with the constant and fast flow of measurements that all the sensors send to the cluster. The minimum resources that GKE with the Autopilot mode of operation allows allocating for each Pod are 0.25 vCPU (stands for virtual CPU and represents the portion of the real CPU that is assigned to a particular virtual machine or container) and 0.5 GiB. Even if these values have been assigned to the "low-load" components, they are still a lot for the few simple operations that such components perform. Hence, some resources had to be assigned and only partially used as it is possible to see from those flat lines present in figure 7.3. In this context, it is important to specify that two different lines are present for both the element's container and the isto-proxy.

For illustration purposes, a specific time in the graph has been randomly chosen (the precise moment chosen does not matter as far as it is in the tested time frame since the behavior is somehow constant) and its values have been reported in table 7.1. In particular, only the 9 highest values at that moment are reported and what they represent is not the specific CPU utilization for those specific components at that exact time but the average value in the whole minute considered.

Table 7.1: CPU utilization represented in figure 7.3 at 6 pm.

Pod name	Container name	Value
my-kafka-0	kafka	1.003
mongodb-1	mongodb	0.957
my-bridge.2	istio-proxy	0.898
mongodb-0	mongodb	0.889
mongodb-2	mongodb	0.851
my-bridge-0	istio-proxy	0.829
my-bridge-1	istio-proxy	0.825
my-kafka-0	isto-proxy	0.754
my-bridge-3	istio-proxy	0.643

As expected, all the components with high CPU utilization are the ones that have to deal with data gathering and streaming.
First on the list is one of the Kafka brokers that slightly exceed the 100% usage. However, it does so only by 0.3%, and, as already mentioned, this is something that Kubernetes allows. In a normal context, it would be weird to have only one of the three Kafka brokers appears in the list since it would be imposed on every sensor to only publish messages having the sensor id both in the key and value of the message itself. Hence, different partitions would be assigned to different sensors. However, for testing purposes, the same message was published over and over again and thus only one of the brokers actually had a partition where measurements were published (this without obviously considering replicas that instead concern all the brokers).

Lower in the list, there are all the MongoDB replica set instances, this was predictable since all the measurements are written and replicated in the database instances and, moreover, such operations are quite resource-demanding. It can be seen that only the Kafka and MongoDB Pods are the ones that have more CPU utilization for the main container instead of the istio-proxy. This is probably due to the fact that, taking into consideration how expensive some TLS

operations are, only those elements that have to perform complex operations (i.e. Kafka and MongoDB in the studied case) utilize more resources than the isto-proxy.

Close to the MongoDB replicas, there are the Kafka Bridge replicas. In particular, the sidecar proxies of those Pods. Again, this can be explained by thinking about how expensive the TLS protocol is with respect to what the Kafka Bridge main container does (i.e. translating HTTP requests to Kafka messages and vice versa). Properly allocating resources for this component was challenging. It often crushed and it was necessary to perform a lot of tests to make it suitably work. Specifically, it was required to deploy 4 Kafka Bridge replicas in order to meet the requirements.
Even if the operation performed by this component is simple, it demands a lot of resources. As it is possible to notice from the table, all 4 replicas are present among the 9 highest resource-demanding Pods containers in the system. All of this with the only objective of providing more flexibility (through HTTP access to Kafka) to the application. As it will be also mentioned in the concluding chapter, if the use case considered requires higher performance and the developers are willing to deal with the Kafka protocol every time a message needs to be produced or consumed, then a good idea could be to remove the Kafka Bridge component from the system and have external users directly communicating, after the API gateway, with the Kafka cluster.

The only component in the table that has not been mentioned yet is the istio-proxy instance of the *my-kafka-0* Pod. Even if, as just specified, TLS related operations are most likely not as expensive as most of the operations performed by the Kafka cluster, it still processes all the measurements that go through the system and is therefore in the table as well.

7.2.4 Memory utilization

Figure 7.4 represents the memory utilization of the various architecture components when the application processes 1200 requests per second. The same time frame as the previous graph is depicted and so these values are to consider in relation to the ones presented in the previous subsection.
Differently from what happens with CPU limits, memory limits cannot be exceeded in Kubernetes. In case more than the maximum allocated memory is occupied, the component simply crashes. Hence, it is good practice to keep a good margin with respect to the limit in order to avoid inconvenient situations. Specifically, considering the system studied, a margin of roughly 20% is kept when 1200 req/sec are processed. As already mentioned, the system has been tested with more requests than the usual ones and in those cases, no more than 90% of memory was reached.

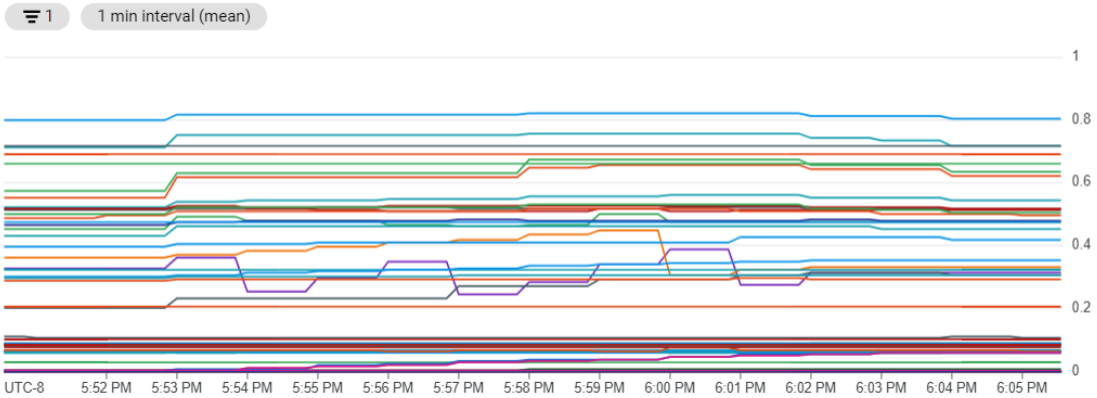

Figure 7.4: Memory utilization with 1200 req/sec.

Another difference between this graph and the CPU utilization one is that a lot of components seem to utilize most of the memory available at the same time. This is probably due to the way the operating system used by the nodes on the Google Cloud Platform frees memory. Most likely, the difference in the two graphs can be explained by the fact that data are kept in memory for some time while CPU highly depends on the specific operations that are performed at the moment analyzed. In this context, it is important to cite one of the configuration choices that has been made when Kafka has been deployed. It regards how the Kafka Cluster manages the retention of the messages produced. In order to avoid keeping a lot of messages at the same time, which could easily fill the allocated memory, a retention of 1 minute has been set up. As is noticeable from the 7.2, even if, considering the testing framework, only one of the three brokers is what receives all the messages, it is not present among the highest 9 memory-demanding components. What would have probably happened with a higher retention value would have been the Kafka broker crashing abruptly.

As happens in the table of the previous subsection and as has been just mentioned, table 7.2 indicates the 9 components that requested more memory in the minute considered (6:00 pm). Similarly to CPU utilization, the istio-proxy that is present as a sidecar in all components consumes a lot of resources also in terms of memory. In particular, again similarly to the previous table, the sidecars associated with Kafka Bridge use up a lot of memory. This is easily noticeable from the table that contains all 4 replicas of Kafka Bridge istio-proxy. During the testing phase, one or more of the Kafka Bridge replicas crashed also for low memory allocation other than low CPU. This proves even more how it should be considered to remove such a component in applications where it is important to save resources and/or have higher processing speed and, at the same time, it is not so fundamental

to access Kafka through HTTP. Moreover, this time differently from the CPU consumption table, an entry is also dedicated to the Kafka Bridge "main" container that, even if should perform quite simple operations, utilizes a lot of RAM.

Table 7.2: Memory utilization represented in figure 7.4 at 6 pm.

Pod name	Container name	Value
my-bridge-1	istio-proxy	0.820
my-kafka-0	istio-proxy	0.756
my-connect-0	my-connect	0.689
my-bridge-3	istio-proxy	0.672
my-connect-2	my-connect	0.659
my-bridge-0	istio-proxy	0.649
my-bridge-2	istio-proxy	0.554
my-kafka-2	isto-proxy	0.530
my-bridge-0	my-bridge	0.524

The other two entries containing the istio-proxy in the "Container name" column are both Kafka brokers. Being the central data streaming unit of the system, it is not surprising to find them in the table. However, what is instead surprising is that, in this case, it is the sidecar that consumes more resources than the main container. Such behavior is the opposite of the one witnessed in the previous subsection.

The last element in the list yet to consider is Kafka Connect. Specifically, it is present two times. This means that, despite being not so demanding CPU wise, requires a lot of memory. In particular, its main container is the one that appears to be one of the most memory-intensive elements of the system.

Chapter 8

Conclusions

The objective of this work has been to develop a distributed system capable of streaming and storing big amounts of data in a secure, reliable, and efficient way. All of this with the use of cutting-edge technologies and the most up-to-date techniques in the industry. Moreover, the system is required to be as flexible and cheap as possible in order to be adaptable to a lot of applications and usable by even small companies. Specifically, this last aspect represents what is the biggest contribution of this thesis. The whole study required knowledge of all the aspects of what are backend design, development, and testing. The entirety of the project demanded an understanding of several topics from basic decentralized schemes and principles to advanced security concerns and complex distributed architectures.

The main challenge has been properly designing a system capable to meet all the imposed requirements at the same time. In order to do so, the coordination of different technologies and techniques has been required. First of all, to obtain a reliable, scalable, and flexible architecture, a Microservices approach has been chosen. However, this type of architecture can be expensive to deploy and hard to manage and coordinate. The latter issue has been solved thanks to the use of the famous container orchestrator Kubernetes while the first problem has been tackled with the help of Google Cloud services. In particular, its capability of hosting Kubernetes applications on its clusters was the solution needed for the problem. Instead, for what regards security, a proper technique was required to protect such a complex system. Fortunately, an ad hoc solution for this type of problem is available. It allows building a dedicated network over the traditional network in which the application is deployed. It imposes specific rules on the components of the cluster and the data exchanged within and outside of it. Such a technique is called *service mesh* and several implementations are provided by open-source projects and private companies. In particular, Google offers its own implementation named Anthos Service Mesh which is based on the open-source Istio project.

After having decided what techniques and technologies to build, manage and deploy the application infrastructure, it has been necessary to choose and configure the single elements composing the Microservices architecture. This phase also needed a thorough study to properly opt for the right components for the cloud environment chosen and, moreover, that could be introduced in the architecture without impairing the normal function of the system that must meet all requirements. In this context, the efficient and well-known distributed event streaming platform offered by Apache Kafka has been used as the central data distribution technology while a MongoDB replica set, thanks to its flexibility and reliability, has been deployed as data storage. Two other components of the architecture, that have been developed with the use of the Spring framework for Java, manage what is the retrieval of data from the database and part of the authentication process. The last element of the system is the API gateway that load-balances and routes to the right microservice all the requests reaching the cluster. In addition, it also performs a fundamental role concerning data encryption and end-users authentication.

Up to this point, it has not been mentioned if a specific use case has been considered. That is to further highlight what is the most important result of this thesis which is the adaptability of the system developed. The described architecture can be utilized in most situations where efficient streaming and collection of big amounts of data is required. However, in order to test the system within a real case scenario a specific use case was needed. In particular, an application capable of gathering data from EMF sensors to allow human users to study the cellular coverage of a specific area has been examined. In this context, the developed application was successfully able to satisfy the expectations. Specifically, cluster resources have been accurately allocated in order to obtain a system that could gather, store, and stream all the sensors data in the most resource-efficient way. This has led to an application that costs $50 dollars per day which is acceptable and affordable considering the complexity and functionalities of the product.

8.1 Future works

One of the main characteristics of the developed application, thanks to its Microservices nature, is to be extensible. For this reason, some of the future works that could be performed on the system in the future could be the integration with other functionalities with the objective to extend or enhance the current product. For instance, all the information stored in MongoDB may be enhanced with some data analytics and machine learning tools. Moreover, adding a frontend application to provide a graphical user interface to the end users would surely benefit the overall system.

In addition, more testing could be performed to further improve the performance of the application. For example, the Kafka Bridge component, which needs a lot of resources to work, could be removed to analyze how much faster the system can get without it, or the *sharding* functionality of MongoDB may be introduced in use cases where a high number of requests of historical data is witnessed.
Another aspect that may be further explored in future works is security. Even though the security features implemented in the system presented in this thesis are all correctly working, some improvements may be introduced for what regards certificate management and the authentication technique. Especially, a highly tested and secure external authentication provider may substitute the internal one utilized in the developed application.

Bibliography

[1] Indeed Editorial Team. *4 Distributed Systems Types (Plus Pros and Cons).* [Online; accessed 09-February-2023]. Aug. 2022. URL: https://www.indeed.com/career-advice/career-development/distributed-systems-types (cit. on p. 4).

[2] Splunk. *What Are Distributed Systems?* [Online; accessed 09-February-2023]. Feb. 2021. URL: https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html (cit. on p. 5).

[3] Chris Richardson. *Pattern: Microservice Architecture.* [Online; accessed 10-February-2023]. 2023. URL: https://microservices.io/patterns/microservices.html (cit. on p. 6).

[4] Mehmet Ozkaya. *Microservices Architecture.* [Online; accessed 10-February-2023]. Sept. 2023. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-2bec9da7d42a (cit. on p. 7).

[5] AWS. *What Is Containerization?* [Online; accessed 11-February-2023]. URL: https://aws.amazon.com/what-is/containerization/#:~:text=Containerization%20is%20a%20software%20deployment,matched%20your%20machine's%20operating%20system. (cit. on pp. 9, 10).

[6] TIBC. *What is Containerization?* [Online; accessed 11-February-2023]. URL: https://www.tibco.com/reference-center/what-is-containerization (cit. on p. 9).

[7] IBM. *What is cloud computing?* [Online; accessed 17-February-2023]. URL: https://www.ibm.com/topics/cloud-computing (cit. on p. 10).

[8] Microsoft Azure. *What is cloud computing?* [Online; accessed 24-February-2023]. URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing (cit. on p. 11).

[9] Arron Fu, CTO UniPrint.net. *7 Different Types of Cloud Computing Structures.* [Online; accessed 24-February-2023]. URL: https://www.uniprint.net/en/7-types-cloud-computing-structures/ (cit. on p. 12).

[10] Paul Krzyzanowski. *Distributed Systems Security*. [Online; accessed 24-February-2023]. Apr. 2021. URL: https://people.cs.rutgers.edu/~pxk/417/notes/crypto.html#:~:text=Security (cit. on p. 13).

[11] Docker. *Docker overview*. [Online; accessed 25-February-2023]. URL: https://docs.docker.com/get-started/overview/ (cit. on p. 17).

[12] Kubernetes. *Kuberetes*. [Online; accessed 26-February-2023]. URL: https://kubernetes.io/ (cit. on p. 18).

[13] Google. *Google Cloud Platform*. [Online; accessed 28-February-2023]. URL: https://cloud.google.com/ (cit. on p. 20).

[14] Google. *Google Kubernetes Engine*. [Online; accessed 28-February-2023]. URL: https://cloud.google.com/kubernetes-engine (cit. on p. 21).

[15] Google. *Anthos Service Mesh*. [Online; accessed 28-February-2023]. URL: https://cloud.google.com/anthos/service-mesh (cit. on p. 21).

[16] Apache. *Apache Kafka*. [Online; accessed 01-March-2023]. URL: https://kafka.apache.org/ (cit. on p. 22).

[17] MongoDB. *MongoDB*. [Online; accessed 02-March-2023]. URL: https://www.mongodb.com (cit. on p. 25).

[18] Spring. *Spring framework*. [Online; accessed 03-March-2023]. URL: https://spring.io/ (cit. on p. 26).

[19] Daniel Stenberg. *cURL*. [Online; accessed 15-March-2023]. URL: https://curl.se/ (cit. on p. 53).

[20] Alex Fernández and contributors. *loadtest*. [Online; accessed 15-March-2023]. URL: https://github.com/alexfernandez/loadtest (cit. on p. 54).