Mariana Luísa Lança Miguel e Fernandes Marques

A Real Application of Multi-tenancy in an Alarm System Software

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. João R. Campos and Carlos G. Araújo and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

Department of Informatics Engineering

Mariana Luísa Lança Miguel e Fernandes Marques

A Real Application of Multi-tenancy in an Alarm System Software

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. João R. Campos and Carlos G. Araújo and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

Abstract

Nowadays, alarm systems are widely used for multiple reasons, from intruders to fire detection, or even environmental hazards and medical services. Throughout the years, this type of systems has gone through a lot of improvements, becoming more and more appealing to their customers and expanding in the global market. Hence, the demand for Alarm Systems Management Software has grown throughout the years in a more competitive market. The **Alarm Manager** is a tool offered by Altice Labs that fits in this Software category, allowing for real-time detection of problems in equipment and offering predictive mechanisms for fault tolerance. However, the system has problems related to high expenses and a substantial ecological footprint, which can be addressed by optimizing resources through the implementation of **multi-tenancy**.

Multi-tenancy is a paradigm of resource sharing between multiple clients, also called tenants. It is a widely used approach because it allows companies to reduce the need for infrastructure while maximizing the usage of the available resources. However, it also has drawbacks, such as complexity, resource competition, and lack of isolation.

The purpose of this project is to conduct a thorough study of the Alarm Manager's current architecture and host services and devise, implement, and test a multi-tenant approach on the most relevant layers, taking into account aspects such as performance and security to the user. Finally, develop a dashboard to assist tenant resource monitoring.

Keywords

Alarm System, Optimization, Resource usage, Resource monitoring, Multi-tenancy

Contents

Acronyms

ACL	Access Control List.
AM	Alarm Manager.
CEP	Complex Event Processing.
FaaS	Function-as-a-Service.
IaaS	Infrastructure-as-a-service.
IAM	Identity and Access Manager.
JAAS	Java Authentication and Authorization Service.
JPA	Persistence API.
NFS	Network File System.
PaaS	Platform-as-a-service.
RLS	Row-Level Security.
SaaS	Software-as-a-service.
TTK	Trouble Ticket.
VM	Virtual Machine.

List of Figures

List of Tables

Chapter 1

Introduction

This report is part of a curricular internship in Software Engineering for the Master's Degree in Informatics Engineering, from the Faculty of Sciences and Technology, at the University of Coimbra, in the curricular year 2022/2023. It was oriented by professor João R. Campos, from the Department Informatics Engineering and Carlos G. Araújo, from Altice Labs.

The following sections present the motivation and context for this internship, as well as its objectives and contributions. Finally, the structure of the document is presented.

1.1 Motivation

Alarm devices are largely used nowadays for a multitude of purposes since these types of equipment are a very efficient and automated way of detecting an immense amount of potential threats. These threats range from intruders to a fire, or even environmental hazards and medical services [45]. Throughout the years, alarm systems have gone through a lot of improvements, becoming more and more appealing to their customers and expanding in the global market. Between the years 2021 and 2022, the global alarm systems and equipment market grew at a compound annual growth rate (CAGR) 8.2% (from $5.11 billion to $5.53 billion)[21], meaning that, for a given year interval, the mean of annual growth was of 8.2% [29].

When having an alarm system installed, it's important to keep a software system for management, known as **Alarm Systems Management Software**. From power and CPU usage to circumstances in which the equipment might fail, this software must let the owner have an insight into the state of the equipment. Furthermore, this type of information must be delivered in real-time, while still maintaining maximum efficiency since these types of equipment generate significant amount of notifications that affect fault detection. This way, these software systems must know how to filter the notifications so as to optimize fault detection and management. In response to this problem, and allied to a growing market, there is an increasing interest in offering better and more competitive monitoring tools for real-time alarm management.

1.2 Context

The Alarm Manager (AM) is an Alarm Systems Management Software from Altice Labs. These types of software allow for either single users or companies to detect equipment issues in real-time, as well as increase service availability. This is done by detecting when a piece of equipment is down and allowing for a professional to resolve the problem and execute a report on the situation. Besides these advantages, the Alarm Manager also offers a predictive fault-tolerant mechanism alongside auto-healing and automated fault resolution before it impacts customer quality. The predictive mechanism can anticipate potential problems affecting the service by detecting the root cause of the problem.

In order to fulfill its purpose, the AM must offer multiple functionalities and be able to store large amounts of data. Consequently, the system requires a large infrastructure so as to respond efficiently to its clients, resulting in it being expensive due to hosting service costs and having a high ecological footprint. However, this is a problem that can be mitigated by optimizing the way these resources are used, i.e, minimizing the number of resources necessary to respond to the same number of clients. **Multi-tenancy** is a paradigm that helps address this matter in an efficient way.

1.3 Objectives

The scope of this curricular internship is to optimize resource usage in the Alarm Manager system. For this, multi-tenancy is thought to be a good solution, at both application and database levels. However, it raises problems regarding the changes to implement in the software and how to mitigate drawbacks associated with the approach.

The objective of this project is to conduct a thorough study of the Alarm Manager's current architecture and used host services and technologies, providing an insight into a solution to implement and in which layers since some offer better optimization than others. After this study, the planned multi-tenant approach is to be implemented and tested, as well as a dashboard required for the clients' management that shall allow an administrator to monitor tenant's resource usage. The approach must take into consideration important aspects such as performance and security to the client.

1.4 Contributions

With the work carried out during the internship, the main contributions for the Alarm Manager are:

- A study on possible multi-tenant solutions that apply to Alarm Manager system and how could these be implemented in the product;
- A comparison of the various solutions and proposal of one of them;
- A practical implementation of the solution proposed and evaluation of the enhancements and setbacks;
- A proposal of a monitoring dashboard for a multi-tenant solution.

1.5 Document Structure

This report is divided into the following chapters:

- **Chapter 2** provides a background to the project, describing important concepts, such as Multi-tenancy, and models to support it. Details on the technologies associated with for the project are also presented;
- **Chapter 3** introduces Alarm Manager's current architecture and its layers' operation process;
- **Chapter 4** presents and describes the functional and non-functional requirements and a risk analysis for the project development;
- **Chapter 5** analyses and discusses how multi-tenancy can be implemented in the proposed layers of the Alarm Manager.
- **Chapter 6** details the development plan, including the project lifecycle;
- **Chapter 7** presents the work developed in this internship, focusing on the multi-tenant application and the Kafka solution separately;
- **Chapter 8** reports and reflects on the test results done to verify the viability of the solution;
- **Chapter 9** concludes this document, presenting a brief description of what was achieved and the future work.

Chapter 2

Background Concepts and Related Work

Multi-tenancy is a software architecture in which users' provisioned resources are shared. Although it seems a simple idea, put like this, it is a much more complex concept that takes into account multiple aspects, from its implementation to problems such as isolation or performance.

In this chapter, the required concepts to understand single and multi-tenancy are introduced.

2.1 Cloud Computing

Cloud computing can be acknowledged as a paradigm for hosting and delivering services over the Internet [68]. The Cloud itself refers to the physical servers, their software, and databases, that, being located in **data centers**, provide users and companies with the necessary assistance to host software systems that are accessible remotely. These data centers are facilities containing many networked computers that manage the data in a cost-saving, productive, efficient, and secure way [23].

A Cloud is, therefore, a pool of virtualized computer resources, which supply the necessary server provisions and configurations, dynamically [20], and that are hosted somewhere out of the scope of the Software owner. Thus, the need to manage physical servers is eliminated while granting the ability to **deliver software on demand**, i.e, only the cloud resources delivered and used by the users will be paid. This payment is the responsibility of the Software owner.

There are four **models of cloud computing** [17]:

- **Infrastructure-as-a-service (IaaS)** - Refers to a virtualized computer environment or physical infrastructure delivered as a service. It includes servers and storage, network equipment, and software.

- **Platform-as-a-service (PaaS)** - It's the delivery of a computing platform (operating system and its services) as a service over the Internet, such as middleware, development tools, database management, and more.

- **Software-as-a-service (SaaS)** - It's an application hosted on a server and delivered over the internet.

- **Function-as-a-Service (FaaS) or Serverless** - Like SaaS, it's an application delivered over the Internet. In this case, the server or cluster management is done by the Cloud provider and the user writes and uploads individual functions that are executed in response to certain events [12].

Note that there is a hierarchical relationship between these models: FaaS runs on top of SaaS, that in turn runs on top of PaaS, and PaaS on top of IaaS [67].

Some widely known service providers are Google Cloud, Amazon Web Services (AWS), Microsoft Azure, and IBM Cloud, among others. Each provider has its advantages and disadvantages, being the Software owner's responsibility to choose what best fits the project.

In Cloud computing, when users access a Web application, they are retrieving data housed in servers located in data centers, instead of locally [23]. These **virtualized** servers contain information to host multiple applications and their databases, but also implement data isolation and security, the necessary precaution and recovery measures in case of disaster, and data backup, among other procedures to keep data available, secure, and always up-to-date.

2.1.1 Virtualization technologies

Virtualization, in the Cloud, comes in two different ways: **Virtual Machines (VMs)** and **Software containers**.

Virtual Machines, or VMs, are the result of an abstraction layer that splits the hardware elements of a single computer into multiple virtual computers. These behave as if they were physical machines with their own hardware and operating systems. So a virtual machine contains several files such as configuration files, storage, and some snapshots that preserve its state at a point in time[35]. For each host infrastructure containing multiple VMs, there is also a Hypervisor that creates and executes each machine[57]. The **left** side of the figure 2.1 shows the architecture of a Host Machine containing two Virtual Machines.

One of the main advantages of VMs, is that they address scheduling, packaging, and security problems, by keeping a clear division between Operating Systems and filesystems[57], and the snapshots which allow the VM to be deleted and recreated in case something goes wrong[35]. However, this technology has disadvantages regarding storage matters. VM images are usually large and thus take considerable space in the host machine They also take long to boot, from 1 to 10 minutes [57].

It is also due to virtualization that is possible to deliver software on demand, by managing the provisioning of resources of each virtual machine, and scaling them as the workloads grow.

On the other hand, *Containerization* is a lighter way, in terms of memory footprint, to deliver Software utilizing OS virtualization principles[57]. These can run directly on physical servers or be hosted in Virtual Machines [30], sharing OS, and contain packages and ready-to-deploy parts of an application, such as the source code, and they can still host the middleware and business logic[57]. This means that the container only stores what it needs to execute its operations. The **right** side of figure 2.1 shows the architecture of containers hosted on a VM.

Contrary to VMs, Containers can boot in a matter of seconds and don't require much storage, as multiple containers share OS, and offers performance benefits, while still maintaining the level of isolation and security of a virtual machine[27].

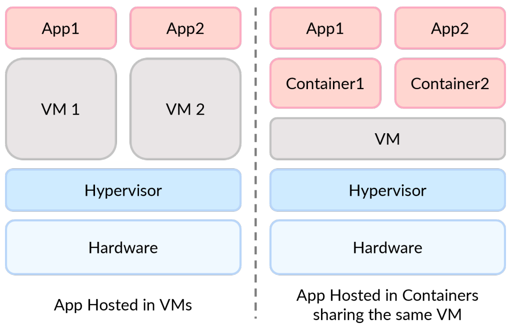


Figure 2.1: VM and Container Architectures, from [57]

2.1.2 Software-as-a-Service

As discussed previously, there are four cloud models. However, a closer look at SaaS is required since, nowadays, a lot of applications are delivered to their users as SaaS applications. Remember that, in this **delivery model**, the **service provider** hosts the application, and handles all of the infrastructure, application logic, deployments, Software upgrades, bug fixes, and other maintenance tasks, while the **Software owner**, who is required to pay a **fee** for the usage of the product[64], manages some application-specific parameters and users. On account of this, the SaaS provider has total control over the management, performance, security, scalability, and privacy of the Software[44].

The users can belong to different **tenants**, being a tenant a group of software users that may have different roles and that share common access to the software instance, such as the application code and data. For example, a company with multiple users sharing the same environment is a case of a tenant[16]. How the resources are shared among these tenants, is defined by two models: a **Single** and a **Multi-tenant**. In the following sections, these two architectural models are explained in depth.

2.2 Single-Tenancy

In the first and most basic Software architecture, **Single-Tenancy**, each tenant is provisioned with a single instance[55], that can be, for example, a Web application or a database. This means that each customer is provided by a dedicated set of server infrastructures, guarantying isolation from other tenants and reducing the risk of accidental data leakages [43]. In this case, the tenant has full control over its resource usage and customization, while still being aided by the hosting provider for software management. However, this architecture has its **drawback**, such as the low cost-efficiency, since each infrastructure is dedicated to each tenant.

Figure 2.3 illustrates how Single-tenancy works, where each of the A, B, and C tenants have their own deployment with dedicated Web servers and database instances.

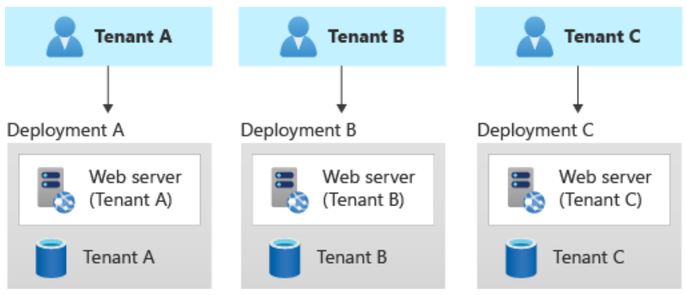


Figure 2.2: Single tenancy, from [43]

Advantages

Some of the advantages of this architecture are the following[43]:

1. **Data isolation** - Data is isolated from other tenant's data, and a tenant's activities cannot influence one another's;
2. **More customization** - Also due to the tenant's separation, it is possible to get data customization;
3. **Easier restore** - Isolated backups are easier to restore in case of disaster.

Disadvantages

On the other hand, regarding the disadvantages of Single-tenancy[43]:

1. **Expensive** - Keeping a whole infrastructure for each tenant is more expensive in terms of fees paid to the Cloud Provider;
2. **Set up and migration** - Setting up a tenant, upgrading or migrating its software version is a slower process and, for migration, it must be done for each tenant;

3. **Resources Usage** - Resources, in a less optimized system, may not be fully exhausted.

2.3 Multi-Tenancy

Aside from Single-tenancy, there is *Multi-tenancy*, in which a single instance of an application and/or database serves multiple customers/tenants. In Cloud computing, this means that multiple tenants share the same cloud resources, such as CPU, networking, storage/database, or application stack, therefore reducing the cost of the resources necessary to serve each tenant and decreasing infrastructure utilization. Multi-tenancy is often used by **Software-as-a-Service** vendors since these environments are cheaper to scale and maintain.

The following figure shows, at a high level, how a multi-tenant SaaS responds to tenants' requests. It owns a shared pool of resources, accessed and used by multiple tenants, depending on their needs.

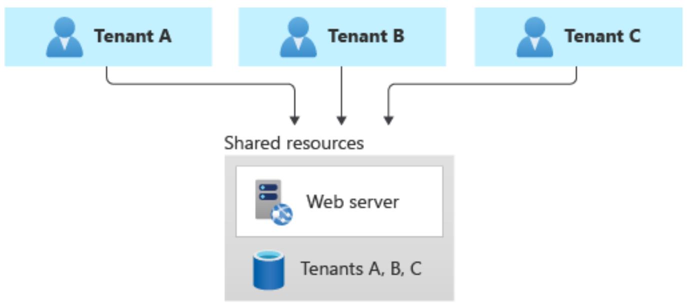


Figure 2.3: Multi-tenancy, from [43]

Although different tenants share the same resources, at a user experience level, it must be the same as if the tenant owns a full instance of the application, providing users with isolation, availability, scalability, and customizability. **Isolation**, since it must appear to the tenant as though they have exclusive access to the application, meaning that their activities do not affect the use of the application by other tenants, and vice-versa, and they cannot access each other's data. **Availability** of the application can not be affected by the actions of other tenants. **Scalability**, for the application must behave independently of the number of tenants, i.e, it must scale to meet the needs of its tenants. And, finally, **Customizability**, because a tenant must have the ability to customize the application, depending on their needs.

Advantages

Multi-tenancy has its advantages when compared with single-tenancy [43][56]:

1. **Lower costs per tenant** - Services can be sold to many customers at lower costs than if each one needed their own dedicated infrastructure, with shared

resources;

2. **Better resource usage** - Sharing the available resources among tenants allows for a maximization of their usage, such as memory and CPU;
3. **Easier maintenance and update** - Since there is only one codebase and data structure to be dealt with, the implementation of upgrades includes all tenants, allowing for these upgrades and maintenance operations to be done at once;
4. **Ease to add new tenants** - There is no need to configure an infrastructure.

Disadvantages However, there are also some drawbacks of this architecture and cases in which a single-tenant architecture can be a better option [43][56]:

1. **Greater security risks** - Since all tenants share the same physical environment, there is a need for strict authentication and access control to prevent clients from reading, writing, or changing each other's data. Also, there is the risk of corrupted data propagation through all clients. An example of a security problem in multi-tenancy is the noisy neighbor, in which a tenant's performance decays due to the activities of another tenant[19];
2. **Generalized problems** - A problem at the provider's end can lead to issues for all users. This may apply to up time, system upgrades and other global processes;
3. **Vulnerability** - Multi-tenant environments allow multiple access points that can be explored, constituting a threat to the application's security. Also, shared services can become a single point of failure if not well-architected;
4. **Less customization** - Due to a common codebase, assuring total customization for each tenant is impossible;
5. **Possibility of competing for resources** - Tenants share resources in a resource pool, and as the number of tenants grows, more resources must be added to that pool. However, there are protocols that deal with this problem, by managing resources usage;
6. **Complexity** - Having a single codebase/database that must be able to serve multiple tenants, adds complexity to its development and maintenance;
7. **Backup and Restoration** - In a shared database, backup and restore data per tenant is a more complex operation.

2.3.1 Considerations when choosing Multi-tenancy

Choosing whether to construct a multi-tenant application or not depends on the trade-offs between the advantages and disadvantages, and so, some aspects must be considered, such as:[56]

- **Tenant requirements** - Such as:
  - **Scalability** - The number of tenants the system will support, the storage per each one and its workload affect the application architecture;
    - For a lot of users that make a low usage of resources, sharing as many components as possible is advantageous;
    - For important tenants that make a large volume of operations and manage a lot of data that may be sensitive, isolated resources might be the best option.
  - **Tenant isolation and security** - The importance of data and workload isolation, for each of the tenants, how it can be done and the influence a tenant may have in others' workload is an important factor to consider when deciding whether to share resources among customers or not;
  - **Customizability** - How important it is, for a tenant, to have customized software that has influence when choosing shared or isolated infrastructures.
- **Per-tenant cost** - How expensive it is to maintain all resources needed for a growing number of customers;
- **Development complexity** - Depending on the system's architecture, what changes are needed to be done in the application and its database, such as schemas and queries, and how complex these tasks are;
- **Operational complexity** - Includes monitoring and managing the software performance, and disaster recovery, among many others.

Ultimately, multi-tenancy is useful for supporting multiple customers in a cloud environment, keeping a good ratio of cost-quality, or simply to reduce infrastructure footprint. However, it is also possible to provide tenants with different architectures, since they have different needs, being that some make extensive use of the resources while others do not. This implies that some would share their resources with other tenants, while others would have their dedicated infrastructure, in a single-tenant architecture. This is referred to as a **Hybrid architecture** [44].

2.3.2 Multi-Tenancy in Databases

In a *database*, multi-tenancy can improve the cost of provisioning and operating databases, handling high-traffic volumes of data at low-cost [40]. This is done by sharing the resources of a single database server with multiple tenants. However, this approach has its downsides, such as a degradation in the performance that arises due to tenant workload competition for critical resources such as CPU, I/O, and memory at the server[53]. Sharing resources can be done in three different ways [66], as seen in figure 2.4: **a.** *Isolated database*, **b.** *isolated schema*, and **c.** *shared table*.

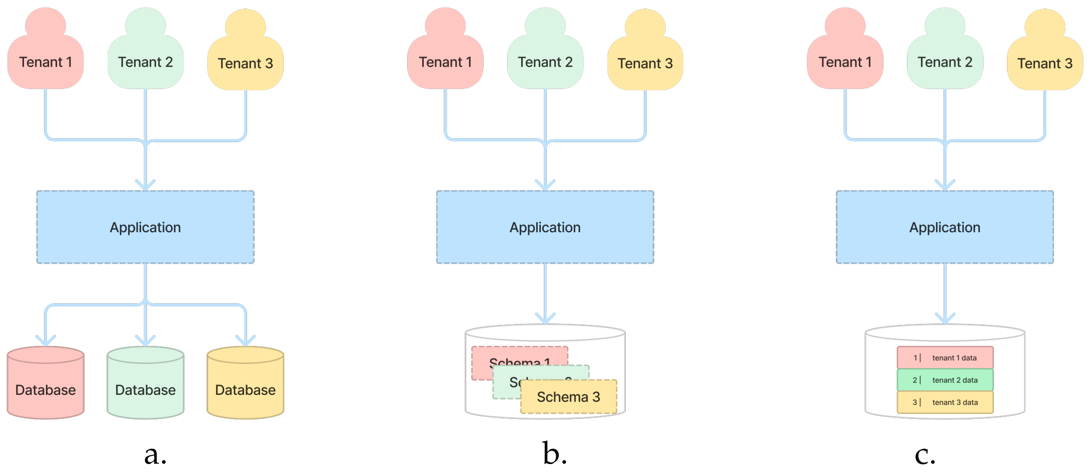


Figure 2.4: Database Multi-tenancy models overview, based on [66]

Isolated database

In this *database model*, each tenant's instance is hosted in a separate database, either within a shared database server or in multiple servers. Due to this database separation, this model allows for greater customization and data and workload isolation, which means that resource usage by a tenant will not affect others [18].

Figure 2.4 a. represents this model, where each tenant accesses a different database through the same application.

This model offers the following **advantages** [15][18]:

- Performance;
- Security isolation;
- Might eliminate the noisy-neighbor problem.

The last point depends whether the databases are hosted in the same or in multiple servers. If in the same server, tenants might still compete for resources, such as bandwidth, and generate bottlenecks. In case the databases are located in different servers, this problem is mitigated.

This model also has its **drawbacks**, such as [15][18]:

- It is not a cost-effective option;
- Managing independently operated workloads increase operational overhead;
- Costs in terms of time to scale, since the ever-increasing number of tenant-specific instances will demand more operational time to administer;

- The complexity due to the necessity to keep a map of tenants to their associated instances.

Isolated Schema

In the *isolated schema model*, each tenant's information is hosted in separated schemas that live in a database. This way, the application needs to connect to the right schema to access data, and hence, having the need to keep a mapping of these associations, just like in the database multi-tenant model [66].

Contrary to the previous model, in an isolated schema model, it is important to have the noisy neighbor problem into account, thus requiring effective monitoring to respond quickly to tenant performance concerns[13].

As shown in figure 2.4 b., a single instance of the application, that serves multiple tenants, accesses different schemas, depending on the tenant.

As usual, this model has its advantages, such as [13][31]:

- Offers data isolation;
- Optimizations can be done at an individual tenant's schema level.

As well as its disadvantages [13][31]:

- Operational and provisioning overhead due to multiple schemas provision.

Shared Table

On top of having isolated databases or schemas, it is possible for tenants to share tables, where every row is associated with a tenant by an identifying value. Furthermore, the application doesn't have to know which schema or database is dedicated to the tenant since the business logic maintains the tenancy logic[66].

However, one of the major problems with this solution is regarding both data and workload isolation, thus being important to ensure that queries never expose data from more than one tenant, and by monitoring data usage to ensure acceptable performance, avoiding cases such as **noisy neighbors**. One solution is **Row-Level Security**, or RLS [58]. The purpose of this mechanism is to filter rows based on a per-user basis, by assigning permissions to different roles, and substituting the WHERE query to request the right tenant information since it is a flaw-prone alternative[58]. RLS works by being applied to tables, meaning that it's not necessary to modify each query to have access restrictions, and, each time a tenant tries to connect to the database, the user's tenant id is sent in the connection and authenticated. When the user fetches storage data, the RLS policies are automatically applied, instead of filtering when querying. This allows to separate what is application logic with multi-tenancy related filters. When the database connection is closed, the tenant id is removed from connection.

One of the advantages of this solution is that if a policy is defined and it is too restricted, no data is retrieved. Contrarily, when using WHERE clauses to filter tenant specific information, if something is missing, data will be leaked, compromising tenant security and isolation. However, this solution has some **performance issues** when escalating the number of tenants or adding complexity to the queries.

In conclusion, some of the advantages of a shared table model are [14][31]:

- A single database schema to maintain, making the rollout process simpler;
- A single database to connect;
- Easier to add a new tenant;
- Lower costs;

On the other hand, some of the disadvantages are [14][31]:

- Lack of tenant isolation;
- Hard to restore a single tenant's data;
- Potential query complexity

2.4 Technologies

Several technologies exist to support cloud and multi-tenancy solutions. This section overviews some of the most relevant for this work.

2.4.1 Docker

Docker[1] is an open-source solution that utilizes containers to **run Software separately from its infrastructure**, hence, allowing for quicker shipping, testing, and deploying of code [26][65].

A Docker container is a loosely isolated environment with the purpose of distributing and testing the application [26]. It contains all the necessary components, such as the host, runtime, code, operating system, tools, and libraries [65].

Architecture

Docker follows a client-server architecture, being that the user communicates with the Docker *daemon*, also known as **dockerd**, either through a REST API, UNIX socket, or a network interface.

As follows, the client, through console commands that are sent to the dockerd through the Docker API, can manage and create **Docker objects**, such as images or containers. **Images** are a lightweight and small template for creating a Docker container, built from a *Dockerfile*, and that can also be downloaded from a Docker Registry. **Containers** are isolated instances of an image, containing its configuration options that are provided when created or started[26]. In order to respond to the client, Docker has the following architecture:

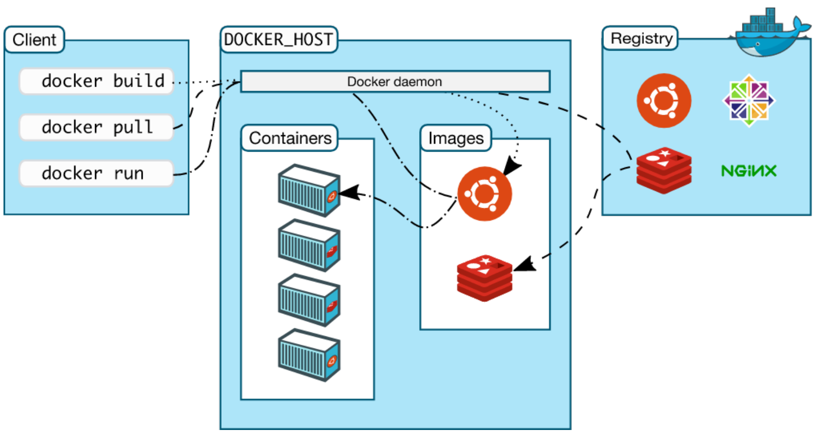


Figure 2.5: Docker architecture, from [26]

[1]https://www.docker.com

As seen in figure 2.5, the docker architecture consists of the following parts [22][26]:

- **Docker engine** - Acts as a client-server application with the purpose of managing the Docker objects. It supports the Docker client CLI, the API and the docker daemon.
- **Docker Client** (docker) - It's the console used to interact with one or more Docker Daemons, through the Docker API.
- **Docker Daemon** (dockerd) - Listens for the API requests and manages the objects, such as images, containers, networks and volumes.
- **Docker Host** - It's a virtual or physical machine running the dockerd and the built containers.
- **Docker Images** - It's a type of a Docker object that contains the instructions for creating a Docker container. To build an image, a **Dockerfile**, which is a text document containing all the commands to assemble the image, is used [1].
- **Docker Container** - It's another type of a Docker object that represents a runnable instance of an image, being provided with the configuration options when created or started.
- **Docker API** - REST API for interaction with the Docker daemon.
- **Docker Registry** - Docker repository for sharing images.

In conclusion, Docker is a method to virtualize an operating system along with the tools to run the software application, in a lightweight and fast way.

2.4.2 Kubernetes

Kubernetes[2] is an open-source platform for management of containerized applications [46]. It helps orchestrate tasks such as networking, provisioning and deploying in containerized workloads, i.e, define an execution wokflow for the existing tasks [48], as well as the mechanisms for failure situations, failover and scalability.

A Kubernetes Cluster is a set of nodes responsible for running the application [47]. It also provides, among others, a load balancer that distributes the network traffic throughout the various containers, orchestrates both workflow and storage, and provides a self healing mechanism. This is capable of restarting, replacing and killing containers if needed.

[2]https://kubernetes.io

Cluster architecture

A Kubernetes Cluster is divided in two parts: a control plane and a data plane. The *data plane* contains the worker nodes that run the application and execute the workload. These worker nodes host the **pods**, i.e, a set of running containers. The **control plane** makes the global decisions regarding the cluster, such as worker nodes and pods, it exposes the API and interface to manage the containers[48].

Figure 2.6 shows a Kubernetes Cluster, composed of a control plane, with three working nodes, and a cloud provider API, which allows to embed cloud-specific control logic.

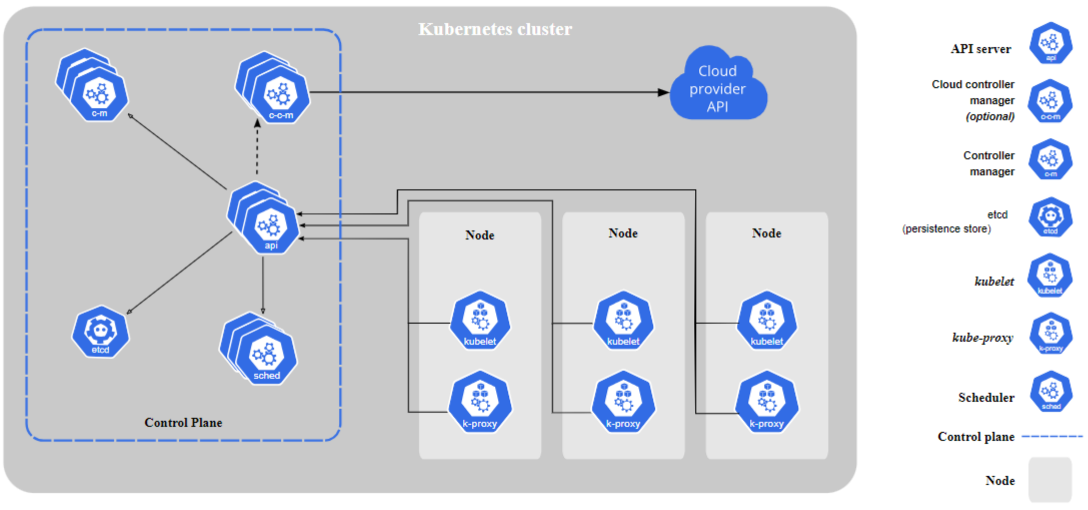


Figure 2.6: Kubernetes cluster, from [48]

As seen, the control plane contains multiple components, such as a **kube-apiserver** that serves as frontend for the control plane, the **etcd**, which is the backing store of the cluster data, the **kube-scheduler**, and the **cloud-controller-manager**. Inside each of the nodes, there is a **kubelet** that makes sure that the containers are running healthily inside the pods, and a **kube-proxy**, which is a network proxy that enables the communication inside or outside the cluster with the pod [47].

2.4.3 Kafka

Apache Kafka[3] is an open-source messaging system for end-to-end event streaming [8]. For this communication, Kafka provides an API that grants applications the ability to consume and produce events from/to the streams in real time, also known as topics, via a TCP network protocol. These events can also be stored for later retrieval, as well as manipulated and processed [8].

[3]https://kafka.apache.org/

In Kafka, there are two types of **clients**: **consumers** and **producers**. A producer can publish messages to a specific stream, the topic, that can be consumed by a consumer, or multiple consumers simultaneously, that have subscribed to the topic, by pulling data from the **server** [39]. These two types of clients are fully **decoupled** from each other. For example, producers do not need to wait for consumers [54].

Consumers are inserted in **consumer groups** that work together to consume from topics, and ensuring that each partition is not consumed by more than one consumer, by balancing and rebalancing in case some member goes missing [54].

Architecture

Kafka is a distributed system that runs in a set of clusters, in multiple data centers. Inside these clusters, multiple nodes, also known as **Kafka brokers**, are divided into a number of partitions that are part of a topic and are either a topic leader or replica. The partition is, in fact, just a single log, where messages are appended to and read in order [54].

The **leader** is one of the topics inside the cluster that is responsible for updating the replicas with the new data as well as coordinating the reads/writes [54]. This coordination consists of receiving messages from producers and furthermore assigning offsets, and committing them to the disk, and, on the other hand, fetch consumers requests and respond with the messages committed to the disk. Since messages are ordered, a consumer can define the offset from which on they want to start reading the messages [8].

The data **replication** along multiple servers works as a fault-tolerance mechanism [8].

Figure 2.7 shows the case of a single Kafka cluster with two brokers responding to a producer and a consumer, as well as replicating the data as this is received.

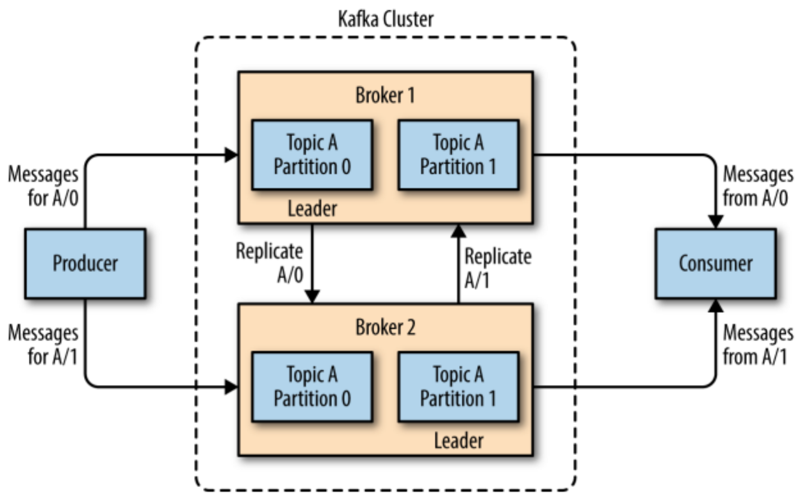


Figure 2.7: Kafka clients interacting with a Kafka Cluster, from [54]

Topics are generally partitioned into multiple brokers. This distribution is therefore what allows multiple clients to both read and write from/to topics simultaneously. Each of events is stored and organized in filesystems, consisting of a key, value, timestamp and optional metadata headers. These are **retained** in the broker for as long as configured in the topic [8].

Configurations, as all as metadata regarding the Kafka clusters and consumer client details is stored in ZooKeeper. **ZooKeeper**, such as Kafka, is an open source Apache project, for management of distributed systems over large clusters [33].

Kafka supports **authentication** as well as **authorization**. Kafka uses **()** and it can be enabled between brokers, between clients and brokers or between brokers and ZooKeeper. There are four protocols supported, next explained:

- **PLAIN** - Is a password based mechanism that uses plain text, i.e, the client proves to the server its identity by sending its credentials not encrypted [5];

- **SCRAM** - also known as Salted Challenge Response Authentication Mechanism, is a password authentication protocol. However, it uses a hashing technique that, using the password, generates a secure value, that is used to authenticate the client. Thus, the password itself is not transmitted. It is used agains eavesdropping [6];

- **OAUTH** - Short for Open Authorization, is a token-based mechanism acting as an intermediary between the end user and a third-party. With this mechanism, the user authenticates inside its application, and a token is provided to the third-party service, acting as a token that authorizes certain information to be shared [50];

- **GSSAPI** - Also known as Generic Security Services Application Programming Interface, is not an authentication mechanism itself. Instead, it provides a generic API for another application, such as Kerberos [41].

The access credentials are stored in a Java Authentication and Authorization Service file, allowing a decoupling of the authentication method.

This authentication mechanism is necessary for the **authorization** as well, which can be done resorting to a JAAS, as well, or with Access Control List (ACL). ACL, as in the name, is a list containing all the operations different users can or cannot execute in the topics. It is stored in the Zookeeper and, when a Kafka broker starts up, the Access Control List is loaded into a cache and maintained for authorization whenever a request comes through [8]. Figure 2.8 shows the workflow of the authorization, from the moment the broker starts and loads its ACLs, to the client request.

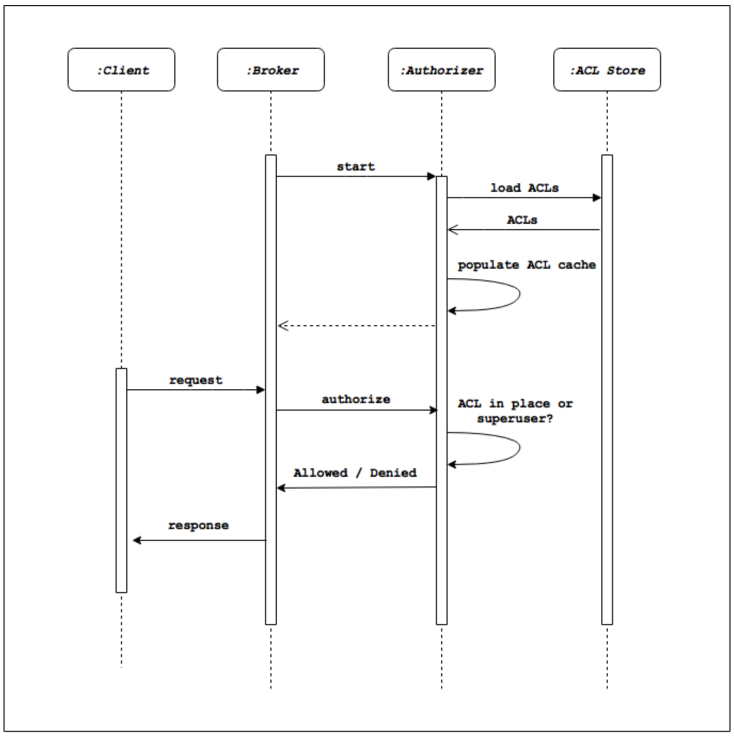


Figure 2.8: Kafka authorization workflow, from [34]

2.4.4 Prometheus and Grafana

Both *Prometheus*[4] and *Grafana*[5] are open-source solutions to scrape and display metrics in a user-friendly way. These two technologies are used to gather and display a multitude of metrics about the application, from resource usage, to the number of active connections in a database or the duration of an alarm's processing. **Metrics** are numeric measurements that are presented as time series.

Prometheus is thus responsible for **acquiring and storing metrics** as time series data, i.e, with a timestamp alongside the metric values[4]. It is composed of multiple components, such as [4]:

- **Prometheus server** - collects and stores the data with an HTTP pull model [62];
- **Client libraries** - to define and expose metrics that are defined inside an application via an HTTP endpoint [61];
- **Push gateway** - so as to expose the metrics from jobs that may not exist long enough to be scraped, such as batch jobs [63];
- **Alertmanager** - to handles alerts sent from the client application [60].

[4]https://prometheus.io/
[5]https://grafana.com/oss/grafana/

This project is supported by Grafana, which is a Web application able to access the stored metrics and to **query, visualize, and create alerts** in personalized dashboards [7].

Chapter 3

Alarm Manager

The **Alarm Manager (AM)** is a system which main purpose is to ensure fault management by surveying, detecting, treating, and monitoring alarms from a network. It is a **passive** approach, which means it does not know of the existence of a fault point, i.e, an alarm device, until the first occurrence of a networking event. This event may be a notification or a critical problem in the device. It should be noted that being a passive system is different from saying that it is predictive. The first one is related to the system discovering new pieces of equipment, while the second with predicting, in certain circumstances, if a problem would happen in an already known fault point.

Given that the Alarm Manager is a complex system, it is extremely important to deeply comprehend its purpose and architecture. So, in this chapter, the system actors and the architecture of the AM is explained, focusing on its layers' purpose, method of operation, and change and resource cost.

3.1 Actors

The system has four **primary actors**, each one with different roles and goals to achieve with the system, and thus, use cases have either the objective of satisfying or being initiated by the actor[24]. The system has four primary actors, next described:

- **Alarm Manager Admin** - This administrator is part of the AM team and has the purpose of managing the system. Hence, he/she can perform actions such as configuring tenant-wide resources and tools;
- **Tenant Admin** - The tenant administrator represents an employee working in a company that uses the Alarm Manager System, and that has the authorization to execute management actions such as alarm and user configuration. It also can perform basic tasks such as monitoring alarms;
- **User** - The user represents a common employee working in a company that uses the Alarm Manager System. He/She has the purpose of monitoring

alarms;

- **Network Operations Center (NOC)** - The NOC is responsible for Monitoring the System Status and its alarms, as well as accessing alarm statistics.

3.2 Alarm Manager's Architecture

The system is hosted in a Cloud Service, **Azure Cloud Services**, where it is deployed in two different ways, depending on the client's choice: in a **Virtual Machine (VM)** or in **Kubernetes**. In the case of Kubernetes, in each Cluster, there are a group of worker nodes, each containing **three different namespaces**, running different tasks in **Docker** containers in every pod:

- One for the authentication system, known as **Identity and Access Manager (IAM)**, that is responsible for generating a bearer token when the user authenticates. This token is used when needed, throughout the system, to identify the user and restrict his/hers actions if not authorized;
- Another for the **AM layers** that processes and executes the logic operations upon the alarm information that is received;
- And a third one for the **Kafka** operations, which streams information between the layers.

The system follows a three-tier architecture, which, as the name indicates, is composed of three different tiers, with different purposes [32]:

1. **Presentation Tier** - Consists of web servers and aims to display information to the user and allow him/her to interact with the server-side;
2. **Application Tier** - This tier is composed of the application servers where the business logic is performed. It has the purpose of communicating with the client side, interacting with the data tier, and processing information;
3. **Data Tier** - Consists of the set of databases that stores and retrieves the necessary data to process and display to the client.

This architecture is widely used nowadays by most Web Applications designed as multi-tier systems, due to its advantages, such as flexibility. [32]

The operations in the AM are possible due to a group of layers that work sequentially to process the data and generate an outcome. These are the **Protocol Adapter**, **Processing**, and the **Web application**, as illustrated in figure 3.1 and detailed ahead:

- **Protocol Adapter** - This adapter processes events received from both internal or external sources. In the second case, these events are received in Gateways. It processes SNMP, HTTP, and Syslog events, and events generated by the AM;
- **Processing** - This layer is responsible for performing actions in alarms as well as in events, such as event processing, suppression, duplication, alarm group, and inhibition;
- **Web Application** - Main GUI to access variety of services offered, such as the Active Alarm Window, Monitoring, Management, Reports and Statistics.

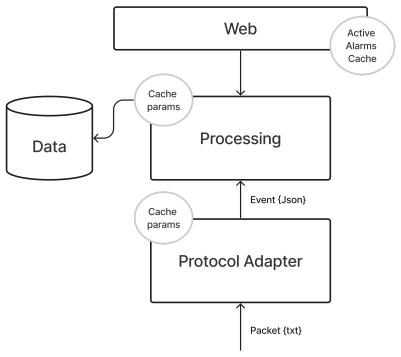


Figure 3.1: AM layers

Also, the system provides another tools so as to allow the client to customize the alarm processing and its rules. These are called **toolkits**, and can be of two different types:

- The **correlation toolkit**, in the AM, allows the end user to manage and create rules for the **Correlation Engines**, which analyse and identify the root causes of problems, their impact, and relates with all the alarms derived from that cause;
- The **SNMP toolkit** allows the end user to adapt the format of SNMP events, in order to let the system know how to parse the custom parameters of these events.

These layers function separately from each other, i.e, are **decoupled**, **communicating data** through **Kafka** pipelines or **Network File System (NFS)**. The pipelines are controlled by the **Zookeeper** that manages all the metadata regarding its producers, brokers, and consumers [38]. In Kafka, the data is published in topics and sent through a stream to the right consumers, who, later on, consume it, whereas, in the case of the NFS, the data is stored in files on a network, allowing the system to access those files and directories remotely [36]. For communication with

external systems or with both the Correlation and Simple Network Management Protocol (SNMP) Toolkit, an API is used.

Data, such as the active alarms information, event configurations and parameterizations, and rule sets, is stored directly in either a **PostgreSQL** or an **Oracle** database. However, in order to minimize access to the database, and hence maximize performance, data that is normally displayed and edited, and therefore needs to be rapidly and commonly accessed, is also stored in a **cache**. As consequence, when a client enters the web application, the displayed information is retrieved from a cache, or, if needed, from the database to be stored in the cache for later use. The technology used for the database depends on the client's preferences.

The system is coded in Java language.

3.3 Alarm Procedures

The **procedure** starts with the arrival of a text packet, such as SNMP traps, HTTP packets, or Syslog messages, in one of the gateways. Depending on the gateway it lands on, the packet will have a different identifier that will correspond to different event formats, known by the system. This translation is done in the protocol adapter. In case the packet comes as a trap, i.e, as a SNMP packet, customizations, defined by the client in the SNMP toolkit, can be applied. This means that different traps can follow different formats. After this translation is done, what is now an event and no longer a packet is sent to the processing layer to have client's attributes and parameterizations, that are stored in cache, applied. At this time, the event becomes an alarm and is ready to be displayed in the Web layer.

Other processes happen in parallel, such as event correlation, meaning that a set of correlation engines are applied to this alarm with the purpose of analysing its root cause and identifying what generated the problem and its impact.

Associated with these alarms and their parameterizations, some timers may activate a trigger that is responsible for executing periodic action on the alarms. An example of this is the incremental priority option, that defines, under certain circumstances, how the alarm priority increments.

In the web application, the user is able to define the alarm parameterizations as well as new rules for the correlation engines.

3.4 Alarm Manager Layers

Since the system has a **modular architecture**, the layers are **decoupled** from each other, communicating through the Kafka pipelines or NFS. This means that whatever method is used to process information inside a layer, the other layers will not be affected. Only the output must maintain its format, so it can be recognized by the other layers.

3.4.1 Protocol Adapter

This layer has the purpose of receiving the packets caught by the Gateway and translating them from the original text format to a uniform JSON format that the system has the ability of processing.

The system, in order to parse and translate the packet, needs to know its structure, which is known through a gateway identifier that comes with the packet. Depending on this identifier, the Protocol adapter manages to deconstruct the text packet, and gather all the information that is needed to create an event, in a JSON format.

However, in case it is an SNMP event, custom parsing can be done, since the client, through the SNMP toolkit, can add new rules to this process, also called **mib**.

Finally, the event is sent through a **Kafka Topic** or a REST API to the Processing Layer and to the Correlation Engines, in case the client has acquired the service.

3.4.2 Processing

The Processing layer, as the name says, processes the events and execute the required actions for the system to work properly. It functions as the core of the whole system, having **three main** tasks:

1. **Process the received events** - Events arrive at the Processing layer, from the Protocol Adapter and through the correct Kafka topic with the right format. Next, it is necessary to verify whether the alarm is suppressed and if the event's required fields are validated. The event, itself, is then processed, i.e, parametrizations are calculated and applied and, in case the process has been successsful, it's registered and both the fault point and the alarm registration are created and updated in the database. At last, the alarm is processed and flush asynchronously to the Web layer, through a Kafka topic, to the web cache.

2. **Respond to client's requests** - Client requests arrive at the Processing via REST API requests. Depending on the request, different courses of action will be taken, such as recognize, archive or update an alarm parametrizations. These changes are flushed to the database and active alarms cache, in the web layer. Finally, a success or failure message is sent back to the client.

3. **Handle firing timers** - The Processing is constantly listening to automatic timers, such as waiting, guard, and archive timers. For example, an alarm that has a 5 minute archive timer, if during that interval it does not receive a new event, it is archived automatically.

The Processing layer, as the name says, processes the JSON events that are sent from the Protocol Adapter, with the expected format, by customizing, applying parameterizations, such as applying timers, setting alarm priorities, and adding thresholds, and creating the alarm itself. Since these parameterizations rely on the client and the type of event, the Processing looks up this information in the cache, or, if not possible, from the database. Finally, these newly built alarms are flushed to the Active Alarms Cache, in order to display the new information in the Web application.

It thus interact with multiple components:

- **Nginx** - It is responsible for load balancing, redirecting the requests to available processing nodes, using Round-Robin. This means that, when an event arrives from the Protocol adapter or from the Web layer, Nginx has to delegate one of the nodes in order to execute the required task;
- **Hibernate** - It allows the integration of the backend with the database by mapping Java classes into database tables;
- **Velocity** - Apache velocity is used to build views from a template. These views are used to build files, such as XML, with a defined template. They are seen, for example, in the SNMP tooklit or for the reports.
- **Kafka** - Enables the event transition between layers. This is done between most layers in the AM and, in the case of the Processing, the events come from the Protocol Adapter e these are propagated to the Web layer, after being processed. These events have a retention period of 12 hours, i.e, they are kept stored for 12 hours.
- **Esper** - It is a Complex Event Processing (CEP) with the purpose of monitoring and analyse events from Kafka topics and database. Using statements, it is capable of checking if an event from a subsystem, machine and directory is being received, gather information on a machine's activity or lack of it, among others. With a set of rules it is capable of filtering data as needed.
- **REST API** - Grants users the ability to interact with the existing alarms, by **HTTP requests**. The Processing will then validate the request and, in case it is valid, execute the needed tasks. Given that this request is synchronous, when it is processed, the Processing sends back the REST response and updates the cache and database, if needed.
- **Cache** - The main purpose of the cache is to offer a rapid access to the necessary data for the AM system. For the Processing, there is a set of **loading caches** that include alarm parameterizations, such as rules, filters, among others. These are small caches used for the processing mechanism. Being an **eternal cache**, its data does not expire, and periodically it synchronizes with the database, through a Kafka topic, alarmsTopic. The cache can have two types: ehcache or Guava.
- **Database** - All the information regarding the alarms is stored in the database. Every time an alarm is updated or a new event received, the database is updated. However, in only two schenarios data is retrieved from the database:

when it is not contained in cache or in case of cache synchronization. It can be Oracle or PostgreSQL, depending on the client's choice.

- **Quartz** - This technology is used in the Processing for **scheduling** actions. A scheduled alarm goes off when a trigger is activated, for example, when there is no new events received in a defined period of time, so as to be archived, or change status to waiting or guard.
- **Webhooks** - These allow, resorting to URLs, to define custom operations in the form of **callbacks**, that run in the system when an action is executed.

The following figure 3.2 represents the relation between Processing, some of its components, and the other layers:

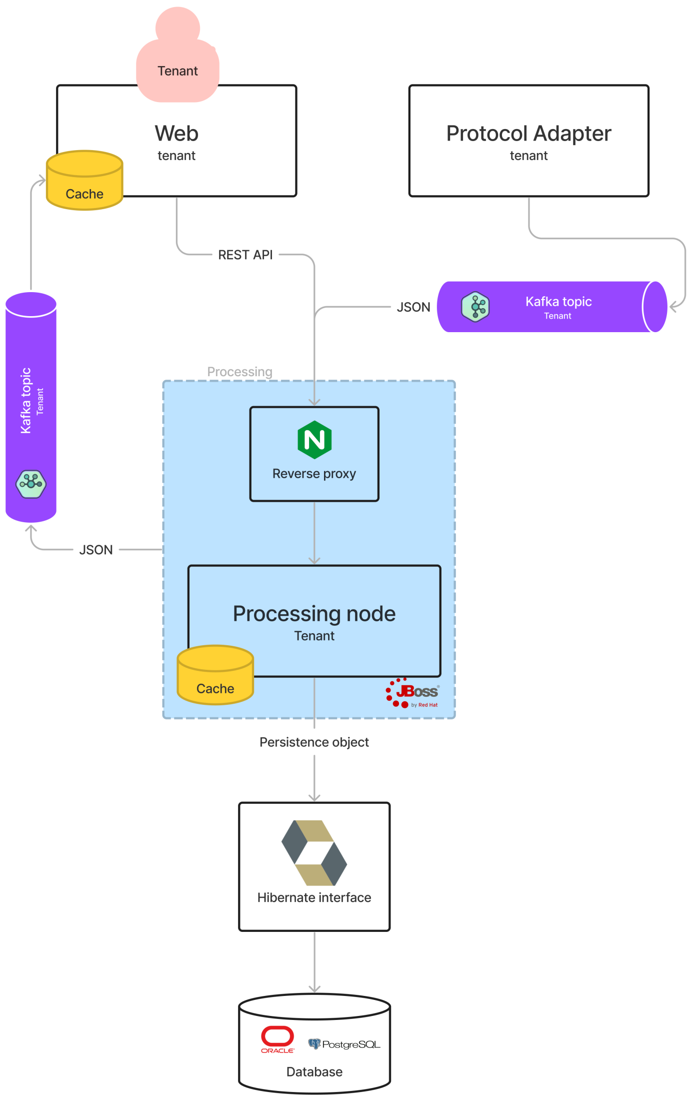


Figure 3.2: Relation between Processing and its components and other layers

3.4.3 Web Layer

Finally, the Web Layer puts on view the outcome of this process to the client, in the form of a list of all the active alarms. These alarms are stored in the Active Alarms Cache and are accessed when the client authenticates and enters the application.

Besides this, since the user can also apply parameterizations to the alarms, and create Trouble Tickets, TTKs, among other actions, the Web layer also has to send this information to the processing layer in order for this to be stored for later use.

3.5 Layers Complexity and Resource Cost

Because the layers are independent of each other it is possible to change their structure without affecting other layers, as long as the output structure is kept. For this reason, it is possible to define a phased migration to a multi-tenant approach.

Knowing that layers can be altered independently and that the system is too massive for a whole multi-tenant solution in the duration of this internship, some were prioritized. So, in order to understand development priority, it is important to study how costly migration is. Bear in mind that these qualitative attributions are assigned according to Altice's internal priorities and with the system expertise of the team's technical leader. For this comparison, the following parameters were used:

- **Algorithm Complexity (AC)** - How complex is the current layer implementation; It depends on the algorithm as well as its dependencies. It can have values of High or Low;

- **Implementation Complexity (IC)** - This is a prediction of how complex it may be to implement a multi-tenant solution; It depends on the algorithm, its dependencies, and possible solutions. However, a more thorough analysis is done in section 5. It can have values of Very High, High, Medium, or Low;

- **Resources Usage (RU)** - It refers to the resource consumption of each system layer. It depends on the total amount of the client's processed alarms. It can have values of High or Low.

A good layer to implement multi-tenancy is one that has a low algorithm complexity and high resource usage since it allows maximizing resource economy with the least implementation effort.

Layer	**AC**	**IC**	**RU**
Web	Low	Low	Low
Processing	High	Very High	High
Correlation Engines	High	High	High
Toolkits (SNMP, Correlation)	High	Very High	Low
Protocol Adapter	High	Very High	High
Kafka	Low	Medium	High
API	Low	Low	Low

Table 3.1: Layers complexity and resource usage

Even though it's not a layer, and thus it should not be included in the previous table, the **database** has a high implementation complexity and low resource usage.

Currently, Kafka is the layer that consumes resources the most, especially RAM, not only because it relies heavily on storing the messages in the filesystem but also because it is used in most communications between layers. This adds some complexity to the solution since it requires a lot of changes in a lot of points of the system/code, making this a relevant layer to study.

On the other hand, layers like Processing, Protocol Adapter, and Toolkits, have a high degree of complexity, meaning that changing these layers will be costly. However, in the Processing, Protocol Adapter, and Correlation Engines, the resource usage is high, making these important modules to focus on.

Chapter 4

Requirements

In order to understand what the system will offer and how it will behave, it is essential to identify, define, and document its requirements, both functional and non-functional, constraints, risks, system assumptions, issues, and dependencies. All this information is presented in this chapter.

4.1 Functional Requirements

Functional requirements describe what the system must do to meet the customer's needs. And so, considering that a multi-tenant approach in the system layers is supposed to be invisible to the user, meaning that he/she is not expected to notice any difference in how the system behaves, these requirements are limited to configuration and monitoring functions executed by an **Alarm Manager Admin**. Hence, even though the system has four types of users, only one is relevant to these requirements.

Figure 4.1, shows the requirements to be developed, followed by a simplified description of these processes from a user's perspective, also known as **user stories**. These follow the template proposed by Mike Cohn [52], responding to the questions **Who, what, and why** for each user story.

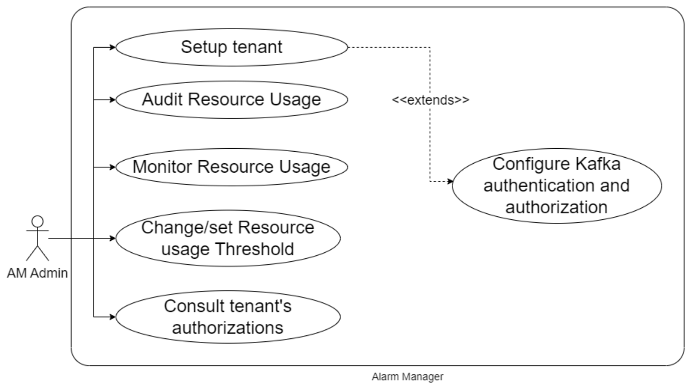

Setup tenant
Audit Resource Usage
<<extends>>
Monitor Resource Usage
Configure Kafka authentication and authorization
Change/set Resource usage Threshold
AM Admin
Consult tenant's authorizations
Alarm Manager

Figure 4.1: Use case diagram level 1

The implementation of these user stories is restricted by the changed layers. For example, being the Kafka layer multi-tenant, monitoring must be implemented for this part of the solution. Besides this, only user stories with a **high priority** are to be implemented. The rest will be converted into issues and serve as future work for the Alarm Manager team, as requested by the advisor from Altice labs.

Setup Tenant

Priority: High
As an Alarm Manager Admin
I want to be able to make the needed configurations
So that a new tenant admin has the needed resources, and authorizations to make use of the system.
Scenario: AM admin finishes configuration and guarantees that the procedure was done correctly. *"Given I'm in the role of AM administrator* *When I use create a new namespace in a Kubernetes cluster* *And I create a new kustomize overlay* *Then i define which replicas are needed* *And I configure the cluster variables such as the tenant topic prefix for Kafka, the database credentials, and its schema* *Then I create the NFS directory* *And set its path in the right file* *Then I configure the database schema* *Then I configure the Kafka authentication in the Kafka cluster and the authorization* *Then I install the instance* *And I wait for all pods to be up* *Then the cluster manager has the status "Running" for all pods*

Table 4.1: US1 - Setup Tenant

Audit Resource usage

Priority: Medium
As an Alarm Manager Admin
I want to receive notifications in case a tenant exceeds the limit of resources
So that the tenant can be notified and take an action.
Scenario: Tenant crosses the threshold of resources used. *"Given I'm in the role of AM administrator* *When the tenant exceeds the limit of resources* *Then the Grafana dashboard displays the notification of the alert.*

Table 4.2: US2 - Audit Resource usage

Monitor Resource usage

Priority: High
As an Alarm Manager Admin
I want to be able to consult how the resources are being used
So that I can collect and evaluate information.
Scenario: AM admin monitors resource usage. *"Given I'm in the role of AM administrator* *When I open the Grafana dashboard* *Then it shows me the statistics of the allocated resources* *When I select the tenant in the option "tenant"* *Then the dashboard shows me the tenant's resources usage*

Table 4.3: US3 - Monitor Resource usage

Change/Set Resource usage threshold

Priority: Low
As an Alarm Manager Admin
I want to set a threshold to the resources a tenant can make use of
So that I can take an action in case it is crossed.
Scenario: AM Admin sets a resource threshold. *"Given I'm in the role of AM administrator* *When I open the Grafana dashboard* *And go into the alerts page* *Then I select the variable I want to monitor* *And the threshold value* *Then I save* *When I go back to the graph information* *Then I see the created alert.*

Table 4.4: US4 - Change/Set Resource usage threshold

Consult tenant's authorizations

Priority: Medium
As an Alarm Manager Admin
I want to be able to consult tenant's authorizations
So that I can collect and evaluate information and change anything, if needed.
Scenario: AM admin monitors access authorizations. *"Given I'm in the role of AM administrator* *When I open the access monitoring page*

And look up the tenant Id *Then the System shows me all the authorizations the tenant has."*

Table 4.5: US5 - Access authorizations

Configure Kafka authentication and authorization

Priority: High
As an Alarm Manager Admin
I want to be able to configure a tenant's Kafka authentication and authorization
So that the AM admin grants or revokes tenant authorization.
Scenario: AM admin grants/revokes tenant authorization to a group of topics. *"Given I'm in the role of AM administrator* *When I open JAAS file inside the Kafka cluster* *Then I append the new tenant's credentials* *Then I run the command to create a new ACL entry to grant access to the tenant to all topics starting with the right prefix* *When I run the command to see the list of all ACL entries* *Then I see the newly created entry.*

Table 4.6: US6 - Configure Kafka authentication and authorization

4.2 Non Functional Requirements

Following what functions the system requires, it is also essential to specify the needed **characteristics** of the system and their **measurement criteria**. This process is the non-functional requirements gathering, so, in this section, for each of the attributes, a scenario is provided, along with a **MoSCoW** method rating (M for Must, S for Should, C for Could and W for Would). The following non-functional requirements are the most relevant for the system's architecture:

1. **Efficiency (M)** - Efficiency is a measure of how well the system utilizes its resources, such as processor capacity, disk space, memory, and communication bandwidth.

   - Description [Scenario 1]: In normal operation mode, when requested, the system provides the user with the requested resources from a set of shared pools. The CPU capacity stays below 80% and is able to respond to 100% of tenants' requests.

2. **Confidentiality (M)** - Deals with blocking unauthorized access to functions and information, ensuring that the software is protecting the privacy and safety of data, and delivering it as intended.

- Description [Scenario 2]: In normal operation mode, if a tenant tries to access another tenant's data, the system should prevent access. 100% of these unauthorized accesses should be prevented.

4.3 Risk Analysis

Before deciding whether a project should move forward or not, it's always required to do a risk analysis. This process serves to identify factors that can negatively affect the project's development or success. And so, identifying, addressing, and finding proper solutions from an early stage is an imperative task.

Gathering risks is part of a RAID analysis. A *RAID* log is a project management tool used to analyse and document project critical factors on an ongoing project, such as possible issues and project dependencies[11]. The acronym "RAID" stands for **R**isks, **A**ssumptions, **I**ssues, and **D**ependencies.

4.3.1 Risks

Risks are potential problems that may arise in the project and, therefore, it is important to identify, analyse and solve these before they have a negative impact on the project. In this section, risks shown in table 4.7 are divided into three categories: Low (severity 1-2), Medium (severity 3-4), and High (severity 6). **Severity**(Sev.) is calculated by multiplying **probability** of the occurrence (Prob.) with the **impact** (Imp.).

ID	Description	Prob.	Imp.	Sev.	Consequence
R1	Due to the problem and system complexity, a proper solution may not be found within the internship duration.	1	3	3	The final product is not completed.
R2	Due to the complexity of the problem, it may be too difficult to implement a solution with the planned features due to lack of experience and complexity of the product.	2	3	6	The requirements may be misunderstood and it may not be possible to implement all the target requirements.
R3	Since changes in the project objectives can occur, there may not be enough time to conclude all planned tasks.	2	2	4	The requirements may suffer changes. If there is no time, these may not be implemented as planned.

R4	Since an agile process is used, some requirements may be altered, reprioritized, or not implemented.	2	2	4	The final product may differ from the original plan.

Table 4.7: Risks Table

For each risk, there is a mitigation plan, i.e, a solution to the problem, as shown in table 4.8. These aim to either **avoid** the risk occurrence, **mitigate** in case it happens or if there is no possible solution, **accept** the problem.

ID	**Mitigation Plan**
R1	Accept
R2	By going through an extensive analysis of the problem, partitioning it, and solving each of the parts, with the help of more experienced people, it is possible to avoid this problem or at least minimize its impact.
R3	Identify and specify thoroughly all the relevant aspects in the early phase in order to minimize unknown issues that may cause changes in the project.
R4	Replan the tasks in order to understand whether the new requirements are feasable within the internship time and that at least the most important ones are assured.

Table 4.8: Risks' mitigation plans Table

4.3.2 Assumptions

Assumptions are aspects of the project that are known to be true. However, if a wrong assumption that leads to risk is made, it is simple to identify its root cause and discover if the assumption is true or false. Each assumption is detailed with a description, a reason for why it is an assumption, and its impact on the project, in case it is found to be false, as seen in the table 4.9:

ID	**Description**	**Reason for Assumption**	**Impact if Assumption Incorrect**
A1	The technologies used support a multi-tenant environment	Technologies documentation corroborate this assumption	Depending on the technology, some parts of the project may require a technology change or may not be included in the solution

A2	Moving from a single to a multi-tenant architecture will be beneficial for the project	A theoretical study indicates tha, when it comes to resource consumption, a multi-tenant solution is advantageous.	The project loses its interest.

Table 4.9: Assumptions Table

4.3.3 Issues

Issues refer to problems that are occurring during the project and were not expected. Hence, these are to be documented as they appear as the project progresses. To document these, a description of the issue and its impact is done, as well as the issue's priority.

ID	Issue Description	Impact Description	Imp.	Prior.
I1	Some of the used technologies offer a limited multi-tenant solution.	It impacts multi-tenant architecture nd its advantages.	H.	H.
I2	Some technologies don't offer sufficient tools to monitor or limit tenant-specific resource usage (For example, Kafka Multi-tenancy only allows applying quotas to streams that are used by multiple tenants.)	It impacts how precise the resource monitoring is.	M.	M.
I3	Lack of permissions to execute some tasks, such as apply changes in a Kubernetes pod.	Some tasks have to be accompanied by a team member that possesses the required permissions.	M.	M.
I4	Testing limitations	Testing was limited to the machine specifications and the number of tenant instances available.	M.	L.

Table 4.10: Issues Table

4.3.4 Dependencies

Dependencies are tasks that depend on the completion of other tasks, in order to be started or finished. This helps to define a timeline for the project tasks. Table 4.11 shows the dependencies for this project.

ID	**Dependency Description**	**Importance**
D1	Access to the project's source code	High
D2	Access to the Alarm Manager application in order to test implementations	Low
D3	Access and permissions for the Kubernetes cluster	Medium
D4	Access to the cluster's Dashboard	High

Table 4.11: Dependencies Table

Chapter 5

Multi-tenancy Analysis

This chapter has the purpose of presenting multi-tenancy for the Alarm Manager. It is noteworthy to take into account that being a large and complex product, it is impossible to detail a whole multi-tenancy solution within the duration of the internship. For this reason, only the parts considered most relevant, as seen in section 3.5, are studied. This is possible since the Alarm Manager is made up of separated layers, that can be altered and upgraded independently.

And so, for the proposed solution, five main points must be taken into account:

- **Resource management** between the tenants
- **Resource isolation** from other tenants, both data and workload;
- **Resource monitoring**, for supervision and troubleshooting;
- **Customization**, as the service may vary from tenant to tenant in certain circumstances;
- **Scalability**, for tenant growth.

5.1 Analysis of the different layers

The Alarm Manager is a large system with multiple and elaborated functionalities, so aiming for a full product solution is not possible in the given time. Due to this restriction, it is required to analyse the cost and complexity of the various layers in order to study those that can bring the most advantages in case of optimization, and establish a hierarchy.

As referred in the section 3.5, despite its complexity, both **Kafka** and **Processing** are layers that currently make a lot of resource usage, being good starting points for this project. Apart from these, it is also important to find a strategy for a multi-tenant database model, as requested by the internship advisor. Thus, the first phase of the project is expected to analyse multi-tenancy in the database, Kafka, and Processing layers.

5.2 Database

5.2.1 Database Models Overview

As presented in section 2.3.2 of the State of the Art, there are three multi-tenant database models: isolated databases, isolated schemas, and shared tables. In order to make an overview of these models' adequacy, the next table shows the advantages and disadvantages of each one.

	Isolated database	**Isolated Schema**	**Shared table**
Shared Schema among tenants	No	No	Yes
Data isolation	Yes	Yes	Possible; requires isolation mechanisms
Data customization/flexibility	Yes	Yes	Problematic
Query Complexity	Simple	Simple	Complex
Setup time for a tenant	Slow [1]	Fast	Fast
Infrastructure Cost	Most expensive	Medium	Cheapest
Operation Overhead	High	Low	Low
Version Migration Complexity	O(n)	O(n)	O(1)

Table 5.1: Database Models Comparison

As the table shows, due to its physical separation, keeping the databases apart allow for more tenant isolation and customization, since the multiple tenants do not need to share their data environment. However, this comes with a price, for it is more expensive to setup and has a higher operation overhead.

On the other hand, an isolated schema means that all tenants share the same database, but each has a different schema. This separation is enough to keep the tenant's data isolated and also maintain some customization. In the case all tenants have the same schema structure, an isolated schema achieves as much customization as an isolated database. However, it is easier to set up, since it is only needed to set up a new schema, instead of a whole database. Another advantage of an isolated schema compared with an isolated database, is the lower

[1]The setup time in isolated databases depends whether these are hosted in the same server or not. This is applicable for the scenario with multiple servers. If multiple databases are stored in the same server, the time complexity is the same as a shared schema

operation overhead, i.e, metadata files, memory allocation, RAM usage, among others.

Finally, having tenants sharing tables ends up being cheaper in terms of infrastructure as there is only the need to set up, start and host a single database and schema, but falls behind the other models in terms of isolation, customization and query complexity. Since all tenant data is stored in the same storage space, there must be mechanisms to isolate this data, being a common one the RLS, as discussed in 2.3.2. However, queries complexity is added, since they need to be filtered by their tenant. In the case in which queries are already complex as they are, this constraint adds computational work in order to retrieve the intended data.

In order to study which solution fits the problem the best, the product's considerations must also be taken into account.

5.2.2 Alarm Manager Considerations

For the Alarm Manager System case, it is important to have some considerations in mind:

1. Even though caches are used to restrict the number of accesses done to the database, a considerable amount of data is still stored in the database, such as the received alarms, and their configurations, reports, and statistics, among others. Considering that bigger clients generally receive about 100 events per second, this is indeed a lot of data to store and to query, for backup purposes or when the information is not cached. This way, **performance** is an important requirement;

2. For all clients there is only one schema structure, however, each one can configure and add custom alarm parameters; in the database, this is represented as columns in alarm tables, meaning that database **customization** at the **table level** is needed;

3. In terms of database services, these are paid for each database instance, meaning that, until a threshold of resources is crossed, the service price remains unaltered, independently of the number of schemas and tables;

4. Finally, tenants must be isolated, not only due to sensitive data regarding these but also due to workload.

To sum up, **performance**, **customization**, and **isolation** are imperative attributes that shall be kept in the new solution, while still decreasing the cost of the database service.

5.2.3 Models comparison

Having done a theoretical analysis of the possible models to implement, and the most relevant and required attributes in terms of the product itself, it is possible to compare all the possibilities and find a fit for the case.

As seen in table 5.1, both isolated database and isolated schema are the most **secure**, **customizable**, and that offer the **best performance** models. Having into consideration the requirements for the Alarm Manager, in 5.2.2, these three characteristics are extremely important for the system and must be guaranteed, for the client experience must not be affected by sharing the environment with multiple tenants. And so, in a multi-tenant environment, supporting the same level of performance and isolation is needed. However, in the case of a shared table, storing all the information together leads to higher query complexity. Thus, sharing tables among tenants is not the most appropriate solution.

The **query complexity** is a big concern in the system since all queries to the database are done at a level tenant, i.e, data is always retrieved per tenant. In the case of shared tables, since all tenant's data is stored together, the **complexity of the executed queries** is augmented, for these must be executed upon bigger tables and apply a more complex logic in order to filter what is tenant-related data. Row-level security, or RLS, offers a good solution for the filtering matter, yet, for more complex queries, it degenerates performance.

The same does not happen in both isolated database and schema, for there exists a separation between tenant's data. Therefore, the **tables size** is smaller, when compared with the shared table model, **lookup** is faster and there is no need to filter the data per tenant. In comparison with the current system, query complexity remains similar.

However, due to this isolation, **migrating database versions**, either upgrade or downgrade, is a laborious task and it must be done for each tenant individually. The same does not apply for a shared environment, in which this task is less time-consuming because it is done once and applied to all tables in the database at once. Nonetheless, this has the drawback of not allowing tenants to be in different versions of the database since they are forced to change the version as one.

Furthermore, as mentioned in point 3 of the previous section, 5.2.2, being that the service provider charges depend on the number of databases and not schemas or tables, the cost of keeping a schema for each tenant is the same as maintaining the necessary tables. This means that, regarding **costs**, isolating a schema and sharing data cost the same while isolating a database ends up being more expensive.

A major **problem** with having either a single database or a set of databases sharing multiple tenants, is that the number of **points of fault is narrowed** to how many databases are being used. When each tenant has its own database, one failing means that one tenant won't have access to its own data. With the other models, this problem will affect multiple tenants at once. Resolving this matter is possible with fault-tolerance approaches, such as redundancy, or disaster recovery mechanisms. However, these are expensive solutions.

The following table summarizes the discussed points, for each model:

	Isolated database	**Isolated Schema**	**Shared table**
Data isolation	✓	✓	✗
Data customization/flexibility	✓	✓	✗
Query Complexity	✓	✓	✗
Setup time for a tenant	✗	✓	✓
Service Provider cost	✗	✓	✓
Operation Overhead	✗	✓	✓
fault points	✓	✗	✗
Version Migration Complexity	✗	✗	✓

Table 5.2: Database Models benefits

As seen in the table 5.2, the **isolated schema model** offers the most advantages, for it balances the best the required characteristics and the model's own drawbacks, providing the best solution for the Alarm Manager database.

5.2.4 Schema isolation as a Solution

As mentioned earlier, isolating the database schema for each option is the option that best fits the presented problem. So, the next step is how to manipulate the database to an isolated schema model. Recall that, in this model, a single or a set of databases are shared by multiple tenants, where each has access to its own isolated schema, allowing for isolation and customization.

Currently, the Alarm Manager offers their clients two database options: PostgreSQL and Oracle. This internship focuses on PostgreSQL databases.

Before starting implementation, it is important to understand the important aspects to take into account, such as tenant migration from the current architecture to the new one, new tenant setup, authentication, and resource monitoring.

Data migration

The first step is to migrate the already existing data into a single database that will contain the various schemas. Attending that, currently, tenants have separated databases with a single schema, the default "public", it is possible to export every schema, one by one, with the **pg_dump** command. This command dumps a single database that can be output in scripts, aka plain-text files that contain the commands necessary to reconstruct the database, or archive file formats that, in order to reconstruct the database, must be used with **pg_restore** [59].

Using the flag **–schema** with the pg_dump allows choosing which schema shall be exported.

Database access

In order to associate each tenant with their own schema, a table to help associate a tenant id with its own schema is necessary. This is stored in the ALARMMGR schema, which is a global schema.

In the AM, there are two different ways to connect and communicate with the database: through **Persistence API (JPA)** or through a **datasource**. In both cases, users connect to their corresponding database using **Connection Strings**. A SQL connection String is an expression containing the parameters required for the application to connect to a database server, i.e, the server instance, database name, and authentication details, among other details[28]. So, it's possible to pass the **schema identification** as a parameter and have the user connect to the right schema. The schema id is obtained from the **schema_id** table that the system can look up since it knows the **client Tenant ID**. The Tenant ID is obtained from **tenant_id** metadata table that relates the tenant with the client Id.

This way, the client ID has only access to its own data schema, hence being both **authenticated** and **authorized**. Next is shown an example of a user named "alarmmgr" accessing the "alarmmgr_schema" schema in the "alarmmgr_db" database.

```
dbc:postgresql://almgrptc-pgsql-el7.c.ptin.corppt.com:5432/alarmmgr_db?
    currentSchema=alarmmgr_schema
user: alarmmgr
pass: alarmmgr
```

The parameter **currentSchema** specify which schema, or schemas are set in the search-path [42]. Since the data structure does not suffer many changes, the SQL queries don't suffer differences.

5.3 Kafka

As previously mention, the Kafka layer is an extremely important layer, since it is responsible for flowing great amounts of data throughout the whole system layers. For example, for a client that receives about two hundred events each second of a maximum size 4KB, having each of these events flowing through the whole system reflects in a lot of memory being used. However, and specially for smaller clients, these numbers are much minor and therefore it is possible to aggregate multiple tenants together.

5.3.1 Kafka solutions overview

Keeping a shared Kafka for multiple tenants means that these will share a set of clusters and a Zookeeper. Inside a cluster, topics and topic partitions can constitute sharing points for different clients, keeping always in mind their data security, customization, and performance. And so, three different solutions were found:

- Shared topics among tenants with dedicated partitions [51];
- Shared topics among tenants with an identifier in each message;
- Dedicated topics, inserted in different "namespaces" [10].

Next, each of these solutions is described more thoroughly.

Dedicated partitions

Tenants share event streams and topics, in this solution, having a group of dedicated partitions. Remember that a topic is composed of multiple partitions, responsible for orderly and evenly storing the received events. These events can be fairly stocked over multiple partitions due to balancing mechanisms, which calculates where to store the event depending on the offset, which is a sequential id number that uniquely identifies each record. Even though Round-Robin is the most used mechanism, it's up to the producer to decide the balancing mechanism [9].

With the described mechanism it is possible to achieve multi-tenancy. When the message is created, the producer, who is tenant aware, prefixes a tenant identifier to the message key. It is later sent to the stream, where it is stored depending on a defined hash function that maps the id to a particular partition. However, it is important to notice that different tenants will have different loads, leading to imbalanced partitions. Imbalanced partitions in a topic lead to reduced throughput and slower message processing. It is therefore essential that the number of partitions dedicated to each tenant depend on their load.

This solution offers both data and some workload **isolation**, since the messages are stored separately, as well as a good **resource usage**. However, there are many disadvantages: the **scalability** because, unlike topics, partitions must fit on the servers they're hosted in and therefore cannot grow infinitely [9], and also the **unbalanced topics**.

Shared topics

In the next solution, tenants share the same Kafka topics. However, contrary to the previous one, all tenants have access to the same resources, which minimizes the number of topics needed to respond to the clients' needs.

In order to identify the messages, the producer, which is tenant aware, must append an identifier. This can either come in the header or in the body of the message. Keeping the identifier in the header is the best solution, since it does not require the message to be parsed before knowing which tenant it belongs to.

However, and independently of how this identifier is sent, all consumers will have to consume the message, before filtering it and discard those not wanted. This constitutes a hazard to the application performance, since all messages, from all tenants, would have to be received and verified by all applications.

This solution is similar to the dedicated partitions, however there is a main advantage and a main disadvantage: it is a balanced solution but there is no isolation. And so, there must be a mechanism to protect data and to protect from noisy neighbors. Protecting data is possible with **encryption**, either transport encryption or end-to-end encryption:

- **Transport encryption** - The network transmission is encrypted and secured from eavesdropping between the client and the broker server;
- **End-to-end message encryption** - The message itself is encrypted during the message exchange. In the producer side, the message is encrypted alongside a symmetric AES key, and, in the consumer side, an ECDSA and RSA key pair decrypt the AES data key in order to decrypt the message itself.

Regarding the noisy-neighbor problem, quotas come in hand to limit tenant's bandwidth. This way, a client cannot surpass defined resource thresholds and so, when it tries to use more than what is supposed, its resources are throttled. There are multiple types of quotas, such as:

- **Producer and consumer bandwidth quotas** - Define the threshold for a tenant traffic/bandwidth, in bytes per second. When the client exceeds its producer/consumer bandwidth quota, it starts to throttle new requests;
- **Request quotas** - Limits the clock time that a broker spends on a request handler (I/O) and network threads, when a client exceeds the CPU usage permitted. It protects both the request processing as well as other tenants' bandwidth, since excessive CPU usage degrades the broker bandwidth;
- **Quotas on topic operations** - This quota prevents the cluster from being overwhelmed with operations, such as create, delete, or alter;
- **Broker/per-listener limits on connection creation rate** - Limit the amount of CPU spent on creating new connections;
- **Per-IP limits on connection creation rate** - Ensure that no client can create too many connections that add CPU overhead to the broker and may lead to high request latencies.
- **Limit on the number of active connections** - Limit the number of connections from different IPs accepted by the broker.

These quotas can be enforced at the broker level, in its configuration file, or at a client level, for example, using the *ClientQuotaCallback* interface.

Dedicated namespaces

Finally, the third solution entails the scenario where tenants maintain dedicated namespaces. This means that each is able to execute actions on a group of topics that is dedicated to themselves. Although the concept of *namespace* does not exist in Kafka, it is possible to simulate it with the tools given by Kafka itself. This is the solution proposed in the official documentation for a multi-tenant scenario.

In short, tenants produce and consume messages to/from dedicated topics. These topics follow a name convention in which an identifier is prefixed to the topic name. This **prefix** is what simulates the namespace, for Kafka is able to define authorization in a group of topics that have the same prefix.

For this to be put in action, some steps must be taken: Tenant authentication as well authorization for operations to be executed in the topics must be configured, and, at the application side, concatenate the tenant to the topic name.

Starting the the first step, **authentication**, Kafka uses **SASL** authentication, or Simple Authentication and Security Layer. There are four protocols supported by Kafka: plain, SCRAM, OAUTH and GSSAPI. The authentication can be enabled between brokers, between clients and brokers or between brokers and ZooKeeper. The access credentials are stored in a Java Authentication and Authorization Service (JAAS) file, allowing a decoupling of the authentication method. This is an indispensable step for the prefixed-ACLs. A prefixed-ACL entry is applies to all topics starting with a given prefix. For example, a tenant that is given write permission to all topics starting with "tenant.", can write to topic "tenant.topic" and all others with the same prefix, but not to topic "tenant1.topic". This crucial point is what creates the concept of namespace. It has the following format:

```
Principal P is [Allowed/Denied] Operation O From Host H On
Resource R.
```

This way, topics are secured from unexpected accesses. In order for the producer to send the messages to the right topic and the consumer to receive from the right ones, it is necessary to add the prefix when consuming/producing. This is a straightforward step, since the application is to be tenant aware.

This solution has the **advantage** of being an efficient, simple, and secure solution to implement. However, the number of topics is not minimized, as seen in previous solutions, and so, having a crescent number of tenants reflects in an exponential growth of small topics.

Other solutions

Outside Apache Kafka, **Apache Pulsar**, a open source messaging and streaming solution, offers a natively multi-tenant solution. It groups users into units that represent namespaces, as well as set policies such as message retention, expiry policy, and resource utilization as per namespace level or individual topic level. Besides this, it also offers a quoting tools more advanced than Kafka, at a namespace or individual topic level (in Kafka this is normally only possible at a broker level) [25].

Also, **Confluent Platform**, is a paid data streaming platform that utilizes Kafka as its core, and that adds better tools to implement and monitor a multi-tenant solution.

However, it's the team's decision not to change the currently used technologies. Hence, these will not be detailed and the solution is constrained to what's offered by Kafka.

5.3.2 Alarm Manager Considerations

Before deepen on each of the alternatives pros and cons, some consideration regarding the product itself must be taken into account:

- The Alarm Manager has few tenants, however, each generates a lot of data. The main purpose is for the system to support more concurrent users;
- Data isolation is required, for there must be some security mechanism. However, encryption is not a solution, since the passed packages are small and it is slow and complicated to encrypt a lot of small chunks.

Keeping these characteristics in mind, it is possible to study a solution that best fits the problem.

5.3.3 Solutions comparison

Having in mind the previous considerations, in 5.3.2, is possible to gather an answer to the problem.

Both solutions **sharing topics**, as seen, have the advantage of maximizing **resource efficiency** than otherwise. However, both have a major problem: topics do not grow infinitely. In one hand, partitions are limited by the server space where they are hosted, and in the other, the more partitions in a topic the worse the throughput. Hence, having too many tenants interchanging too much data in the same group of topics will lead to a **performance degradation**. On top of this, the solution regarding the dedicated partitions has also the problem of **imbalanced topics**, which, as well, causes a hazard to the system performance. The third solution, concerning dedicated namespaces, from the three presented, does not use resources as well, as it still needs to keep all topics for each of the tenants. Besides, it is preferable to have few bigger topics than multiple small ones [2]. However, it does offer some improvements by sharing the clusters.

Concerning **isolation**, keeping different topics for the tenants is the best solution, for both data and workload. Data is kept segregated in different topics that require authorization to be accessed, but also the lack of performance in a topic does not affect how the remaining function. So, if there is a problem in a topic, for example, lack of balance, the remaining topics will keep on working properly. In the other solutions, the isolation is weaker. For example, in the case of shared topics and partitions, there is no data isolation, for the encryption is not to be supported, meaning that the messages would flow through the same streams completely legible. This aspect makes this solution improper.

Regarding workload isolation, it was proposed the use of quotas to mitigate this problem. However, just as the encryption, it is a team decision not to throttle the clients when these exceed the number of resources allowed, namely CPU, memory, or bandwidth. Instead, the goal is to notify the Network Operations Center (NOC) in order to take some measure.

The table 5.3 summarizes the discussed points, for each solution:

	Dedicated partitions	**Shared topics**	**Dedicated namespaces**
Shared topics	Yes	Yes	No
Data isolation	Yes	No	Yes
Workload isolation	No	No	Yes
Performance	Medium	Low	High
Scalability	Hard	Hard	Yes
Resources gains	High	High	Low
Monitoring per tenant	Hard	Hard	Yes

Table 5.3: Kafka Solutions Comparison

Like other multi-tenant solutions, it would be possible to mix these, in order to create a solution in which small tenants could have separated partitions, and bigger tenants, that would require more workload, would have their own namespaces. But since in this phase of the project a homogeneous solution is the main goal, **dedicated namespaces** was determined to be the best solution, because of its scalability and isolation, even without being the most beneficial in terms of resource gain.

5.3.4 Dedicated namespaces as a Solution

Bearing in mind the solutions and the product limitations, having **different namespaces** for each client proved to be the best solution, for it is possible to maintain isolation without jeopardizing performance, even though the gains are smaller.

For this solution, the authentication protocol must be configured on each of the brokers *server.properties*. This includes specifying the protocol and setting its listeners ports. Besides, the JAAS has to be constructed manually on each of the brokers, and each of the tenants appended to the file. A **plaintext authentication** was established as a sufficient solution since the user does not have direct access to the Kafka cluster and therefore only the application interacts with the authentication mechanism. However, it is still necessary for the authorization. A JAAS entry has the following format:

```
 KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule
required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret"
    user_alice="alice-secret";
};
```

For every **new tenant**, a new entry must be added.

In order to authenticate, a producer/consumer in a Java application must pass its credentials like the following, in a Properties object:

```
        Properties props = new Properties();
        props.setProperty("bootstrap.servers",
"localhost:<port>");
        props.setProperty("security.protocol",
"SASL_PLAINTEXT");
        props.setProperty("sasl.mechanism", "PLAIN");
        props.setProperty("sasl.jaas.config",
"org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"<username>\" password=\"<password>\";");
```

The authorization is done using a command, from the command line. The following example shows how a Prefix ACL can be thought of: "*Principal User:Jane is allowed to produce to any Topic whose name starts with 'Test-' from any host*". This can be translated into a command the following way:

```
bin/kafka-acls.sh --authorizer-properties
zookeeper.connect=localhost:2181 --add --allow-principal
User:Jane --producer --topic Test- --resource-pattern-type
prefixed
```

Or it can also be done in a Java application with the AdminClient[0] library. When the producer/consumer wishes to produce/consume to/from a topic, the broker will search in the Access Control List whether the authenticated user has permission for the request or not.

The application, knowing which tenant it belongs to, can easily produce/consume to/from the right topic, with the format:

```
tenant<id>.<topic-name>
```

[0]Library documentation.

New tenant

For every new tenant most of the previous steps must be manual: appending a new user credential in the JAAS file, creating the new topics one by one with the right prefix, and setting the authorizations for the new tenant.

However, creating the new topics and setting the authorization can also be done in Java in a much more automatic way. The following snippet shows how to create a new ACL entry:

```
    AdminClient adminClient = AdminClient.create(props);

    // Define the resouce to apply the control to
    ResourcePattern resourcePattern = new ResourcePattern(
                ResourceType.TOPIC,
                <topic name>,
                PatternType.PREFIXED);

    // Define user and type of control
    AccessControlEntry accessControlEntry = new
AccessControlEntry(
                        "User:" + <principal>,
                        "*",
                        AclOperation.WRITE ,
                        AclPermissionType.ALLOW);
    AclBinding aclBinding = new AclBinding(resourcePattern,
accessControlEntry);

    // Create the ACL
    CreateAclsResult createAclsResult = adminClient.createAcls(
            Collections.singleton(aclBinding),
            new CreateAclsOptions().timeoutMs(5000)
    );
```

AclOperation defines the ype of ACL and can take the values ALLOW or DENY. *AclPermissionType* defines what the user, also known as principal, can or cannot do. It can have the values WRITE, CREATE, DELETE, ALTER, DESCRIBE, among many others.

Monitoring

Currently, monitoring the Alarm Manager is already done using both **Prometheus** and **Grafana**, to gather and display the metrics, and therefore supervise and troubleshoot. This step is possible configuring a Prometheus port, normally 9090, and is currently already supported by the system. Other solutions are widely used to gather metrics such as Yammer metrics.

With these metrics, its possible to gather data such as request latency, consumer, lag, and metrics on the quotas, among others, that can be grouped by the topic prefix or the logged in client. To note that some metrics are not done at a client or topic level and may complicate the supervision.

For the quota solution, as seen, there will be needed a mechanism to alert when a tenant crosses a threshold, which can be done in Grafana. By defining alerts, the Network Operations Center (NOC) shall receive a notification everytime a tenant surpasses its fair share of resources.

5.4 Processing Layer

The *Processing layer* is the main layer of the system, functioning as its core. It has the purpose of executing the various jobs for the application to work properly. This layer is thus responsible for processing the alarms, when these arrive and are handled by the Protocol Adapter, as well as execute manual actions upon the alarms that are requested by the client, and handle alarm timers. Manual actions from the user include, for example, recognize an alarm or change its priority rules. Given its importance and activity, the Processing must be efficient, since its operations, generally, must be completed in the order of milliseconds. In this section, possible solutions for the processing are proposed.

In order to study a solution, it is important to acknowledge tenant **isolation**, **performance** and **scalability**. It is essential that the Processing is capable of scaling if needed. This is already possible, for the layer escalates horizontally and a scheduler (nginx) helps schedule tasks. However, another problem arises regarding customization. By sharing a same layer architecture, processing different alarms with different formats can become tricky, requiring a more complex code structure.

Inside each Processing node, various components must be included in a solution, such as cache, API, among others. Additionally, since both the database and Kafka solutions have been already defined, for this layer it is also required the Processing to be acquainted with the current tenant and connect to the right Kafka topics and schema.

Due to its complexity, two possibilities are identified: keeping the current processing architecture, with minor changes due to the database and Kafka solutions, or change to a complete multi-tenant solution. Each have different pros and cons. In the following sections, both are described and analysed.

5.4.1 Shared Processing

For a complete multi-tenant solution, the Processing must be fully shared by multiple tenants. Thus, the layer should be composed of multiple Processing nodes that are able of scaling horizontally. For this solution, the layer shall have insight on which tenant an event belongs to.

The following figure presents this architecture. It is composed of a shared processing with multiple nodes. Besides these nodes, there is also as a nginx scheduler and a shared cache. All the nodes communicate with a single database. Events arrive in the layer through a single Kafka cluster, but from different topics, and HTTP requests arrive through the API.

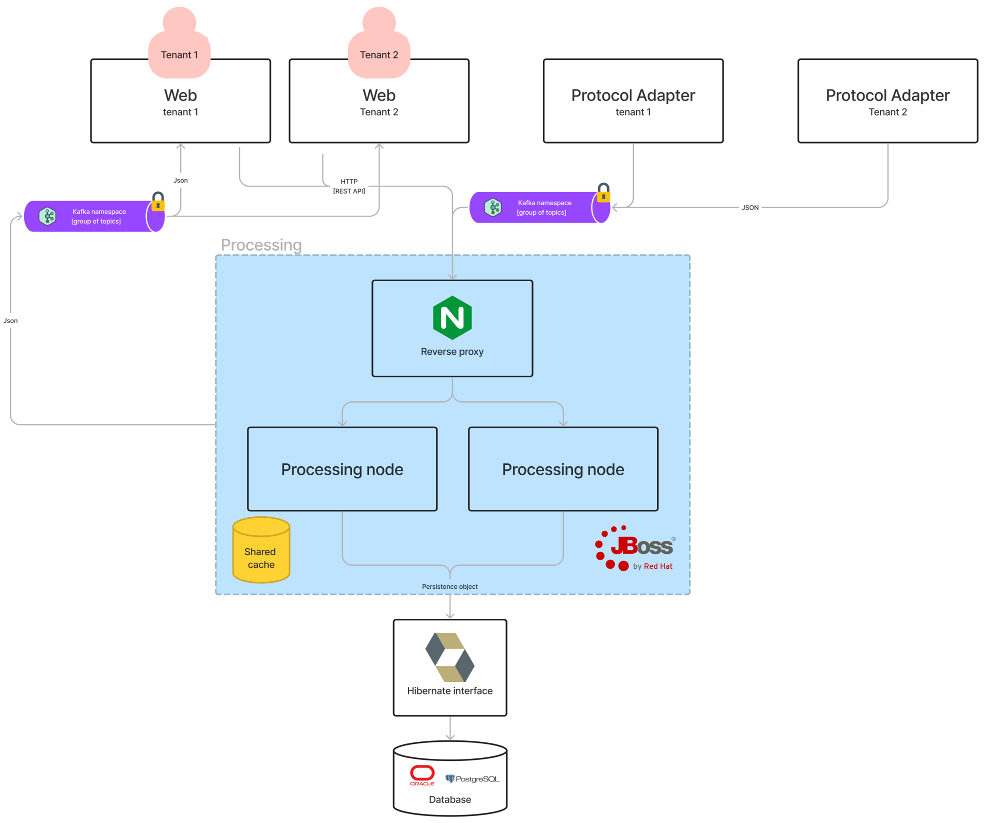


Figure 5.1: Shared processing architecture

Therefore, tenant identification and components behaviour are problems that arise. In the following sections these problems are analysed and new solutions are proposed.

Tenant identifier

Information arrives in the Processing in two different ways: by Kafka topics or by REST requests. In both cases, it is possible to identify the owner of the event.

For **Kafka**, given that various topics are dedicated to a single tenant, and that the name contains its tenant owner, it is possible to obtain its identifier. This will require that a consumer subscribe to multiple topics simultaneously. Knowing all existing tenants, it can add the tenant identifier to the beginning of the topic name and subscribe to all tenant's topics. In order to be aware of all the existing tenants, this information must be stored in a **general database table**.

Other options include adding the identifier to the header or message body, which implies that this information travels multiple application layers, unnecessarily. This would impose an overhead in system process.

Regarding the **REST API**, the tenant identifier can be passed in three different ways: in the URL, in the message header, or in its body.

1. **URL** - Data is passed as a URL parameter. It is a simple method, however, in a scenario where multiple endpoints are already used, multiplying by the number of tenants leads to an exponential growth and therefore low scalabiliy. It does not offer data security.
2. **Header** - In this case, the identifier is passed in the request's header. When a REST API authentication is done, the credentials are transmitted in the header.
3. **Body** - Information is sent in the message body. It's a simple process, allowing flexibility in the format of the data, however, is not efficient, due to parsing overhead.

Alarm Manager (AM) already supports both basic and bearer authentication. Therefore, the tenant identifier can be passed through the header.

Cache

The Processing layer is made up of multiple small caches that are an efficient way of storing data that needs to be swiftly accessed. These are *ConcurrentHashMap* objects, composed of a key and a value. The key is used to lookup a record when needed, and it can range from a String to any other application specific object type, such as *Target*, *TopicPartition*, among others. So, for a multi-tenant solution, a tenant identifier must be included in this pair, in order to identify the owner of a record.

To isolate tenant-specific data within each cache two approaches can be followed: prepend the tenant ID to the record key or map the current hash maps to their tenant IDs.

Since all searches are done resorting to the record key, a simple solution would be to add a **prefix to the key**. This would not require deep changes to the caches structures, however, since some of the keys are class objects, it would be required the addition of a new attribute to identify the tenant.

This solution has the advantage of being simple to implement and would keep the cache structure unaltered. Furthermore, even though each tenant's caches are small, storing them all together leads to an exponential growth of the data structure. However, being a hashing algorithm, ConcurrentHashMap has, in average, time complexity O(1), thus, not imposing a problem, for it is expected to maintain the look up, insert and delete time. The worst case scenario is achieved if the hash keys are too similar, leading the algorithm to behave as a linear searcher and thus taking O(n) time to look up/insert/delete data.

Another approach is to store each of the current caches as values in a hash pair, while keeping the **tenant identifier as key**. Contrary to the previous solution, this would require the addition of a level to the cache structure, leading to an average time complexity higher comparing to the previous case. The time complexity would still be O(1), but with a higher factor due to the need to perform two levels of hashing and additional memory access for the second-level data structures.

In terms of storing space, the first solution is more economic than the second one. Neither option offers any kind of data or performance isolation.

In conclusion, appending the tenant identifier to the both in terms of performance and space is the most efficient and convenient option.

REST API

REST requests are used by external users and between the web application and the system core. For multiple users to share the same API to connect to a set of shared Processing nodes, these can either have separate or shared endpoints. In 5.4.1, these solutions are already described.

Some authentication mechanism, with the purpose of identifying where the request comes from and what it has authorization to retrieve, is indispensable. Alarm Manager (AM) already supports both basic and bearer authentication. Therefore, a multi-tenant solution simply implies that when the node receives a request from an endpoint, the application will have acquired the tenant identifier, needed for authorization.

Its important to note that with this solution, tenants share the same endpoints, and therefore workload.

Database

In section 5.2, keeping separated schemas per tenant is the multi-tenant solution selected. And so, it is required that the application knows how to connect to the right schema. Knowing that in the Alarm Manager connections to the database can be done in two different ways, JPA or datasources, solutions for both cases must be defined.

Persistence API (JPA) in Hibernate is an interface that describes how objects are persisted in a Java application. In addition to this, it also gives the right tools to create a multi-tenant solution, allowing to connect to the right schema at runtime, depending on the tenant identifier. The persistence.xml is where the strategy and classes for the solution are defined.

There are two possible approaches: Either a **single JDBC Connection pool** to the database is done, and the connection is altered using the SQL SET SCHEMA commands, depending on the connected tenant; or **distinct JDBC Connection pool per-tenant**, in which each connections point to a different tenant schema.

A single connection is more flexible, for it can dynamically switch schemas without having to create additional connection pools, and is more resource efficient. However distinct JDBC connections offer stronger resource isolation as well as are easier to maintain and troubleshoot. For this reason, and to avoid the crescent maintenance complexity, having distinct JDBC connection pools is a safer option for a database connection solution.

For **distinct JDBC Connection pool per-tenant**, by passing a tenant identifier, a session is opened in the right schema. For this, it is required to implement two interfaces: A **connection provider** that will provide the connection for the tenant, and a **tenant resolver** in order to identify the current tenant.

The **connection provider** contains at least the methods for getting and releasing connections (either any connection or a specific one). If Hibernate is not able to resolve a tenant identifier, it will use the method **getAnyConnection** to retrieve some connection. These methods are used during startup.

Next is shown an example for implementing a tenant connection using JPAs, provided in [3].

```
    public class SchemaMultiTenantConnectionProvider
            extends AbstractMultiTenantConnectionProvider {

        private final ConnectionProvider connectionProvider;

        public SchemaMultiTenantConnectionProvider() throws
IOException {
            connectionProvider = initConnectionProvider();
        }


        ...

        @Override
        public Connection getConnection(String tenantIdentifier)
throws SQLException {
            Connection connection = super.getConnection(tenantIdentifier);
            connection.createStatement()
                .execute(String.format("SET SCHEMA \%s;",
tenantIdentifier));
            return connection;
        }
    }
```

SchemaMultiTenantConnectionProvider's purpose is to initiate the connection provider, i.e, to create a driver with the necessary multitenant configurations, or to open a connection when required.

As seen, it extends *AbstractMultiTenantConnectionProvider*, an interface from Hibernate used to implement a connection provider for a JPA. For datasources, *AbstractDataSourceBasedMultiTenantConnectionProviderImpl* is used instead.

For the **tenant resolver**, it must implement a tenant identifier and be able to validate the existing sessions. The following snippet shows a possible implementation for a resolver. As seen, it keeps a map of users, containing a key and a value, where the value is the name of the schema to connect to.

```
    public class CurrentTenantIdentifierResolver
            extends MultiTenantResolver {

        private Map<String, String> userDatasourceMap;

        ...

        @Override
        public String resolveCurrentTenantIdentifier() {
            if(this.tenantIdentifier != null
                    && userDatasourceMap.containsKey(this.tenantIdentifier)){
                return userDatasourceMap.get(this.tenantIdentifier);
            }
            return userDatasourceMap.get("default");
        }
    }
```

Having defined these classes, in the persistence.xml it is defined the strategy (DATABASE, DESCRIMINATOR, NONE, OR SCHEMA) and it's specified the class packages for the tenant resolver and connection provider, as seen in the following snippet.

```
    <property name="hibernate.multiTenancy" value="SCHEMA"/>
    <property name="hibernate.tenant_identifier_resolver"
value="multitenant.SchemaTenantResolver"/>
    <property name="hibernate.multi_tenant_connection_provider"
value="multitenant.SchemaMultiTenantProvider"/>
```

This file is where the datasource is also defined. To notice that, since the solution is to keep a single database with multiple schemas, there is only one datasource for all tenants.

Kafka

In section 5.3, it is defined that keeping dedicated namespaces per tenant is the multi-tenant solution proposed. For this, Kafka needs to be aware of all existing tenants. It can be fetched from the database. This is required for the consumption of messages, in order to **subscribe** the "same" topic in all tenant's namespaces simultaneously. In order to produce messages, the producer only needs to know the tenant it belongs to (either from the event header or the API request). In both cases, the tenant's identifier is prefixed to the topic name.

Quartz

As known, *Quartz* is used to trigger actions either periodically or in given circumstances. This mechanism is used, for example, for archiving alarms, in the database, by checking periodically the alarms' archive date, and moving all that are verify the condition. Every job must be dedicated to a tenant.

In a multi-tenant environment with multiple schemas, there are two solutions: A shared job that runs all schemas, or a different job per schema.

Alongside these jobs, the Alarm Manager contains a group of tables in the database that contain information on how to configure jobs and triggers, and it also stores some logs about them. It is possible to either store these tables in a global schema, or keep them in each tenant schema. Since different tenants can have different jobs and triggers configurations, the first solution would require the tenant id to be added to these table. For this solution, Quartz information is stored in each schema.

For a **global job** solution, it must run all tenant schemas. It requires that, during the job, the schema is switched for each of the tenants.

For **jobs per tenant**, a proposed solution is to add a tenant identifier to the job itself. Jobs are created with the *method createNewJob* in the class *TimerManagerEJB*. In order to pass the tenant identifier in the job, it would only be requried to call the method *UsingJobData*, as seen in the following snippet:

```
    private JobDetail createNewJob(Class<? extends Job>
wakeUpHelperClass, String jobName, String jobGroup, String
description) {
        return JobBuilder.newJob(wakeUpHelperClass)
                .withIdentity(jobName, jobGroup)
                .withDescription(description)
                .storeDurably(true)
                .requestRecovery(false)
        .UsingJobData("tenant_id", tenant_id_val)
                .build();
    }
```

With this tenant id, when a trigger is fired, it already contains to whom it is doing its task for, and because of this it can search the archive time in the alarms from a specific schema, for example. The tenant id is obtained by calling the following method:

```
    context.getJobDetail().getJobDataMap().getString("tenant_id")
```

To note that the number of jobs running at a given time is limited by the size of the thread pool. Therefore, it can not grow infinitely.

The first solution excludes the possibility of different tenants having different scheduling properties, such as different firing times. It requires to add the logic behind the schema switching and differences that exist between the schemas. Regarding the second solution, it is heavier, for every job must be multiplied by the number of existing tenants, and there is a threshold for the number of executing jobs. However, the number of tenants is not very high and it is not expected to grow very rapidly, for the chances of this threshold being exceeded are dim.

Having this in mind, and knowing that customization is an important property of Quartz, keeping **separated jobs** for the tenants is the most adequate solution for Quartz.

Monitoring

Custom metrics in the layer are obtained using the Prometheus client library. In the AM system, these metrics' names are statically defined, and so, instead of creating a new metric for each tenant, it is a better option to use **labels**. A label is a way to group metrics together that vary some parameter. Thereby, they allow to separate metrics per tenant.

The following example comes from the project's source code, and is used to obtain the number of requests being processed in a given moment.

```
    private static final Gauge requestsGauge =
Gauge.build().name("alarmmgr_requests_in_progress").help("Requests
In Progress").register();
```

In order to add a label, it is only needed to call the method "**labelNames**" with the label key and, when executing some action on this gauge, such as increment, decrement, start timer, among other, call the method to specify its value.

```
    private static final Gauge requestsGauge =
Gauge.build().name("alarmmgr_requests_in_progress").help("Requests
In Progress").labelNames("tenant").register();

    requestsGauge.labels(tenantID).inc();
```

This is a simple solution to split metrics per tenant, however, requires to change a lot of different points of the source code.

Shared processing flow

The proposed solutions lead to following flows:

- **Web/API Request** - Request is received from the Web layer or from the REST API

  1. Request is received from the REST API - The tenant identification is sent in the authentication mechanism;
  2. In the Processing, the nginx scheduler is responsible for deciding a Processing node;
  3. With the tenant identification, the node knows to whom the request belongs to, being thus able to connect to the right schema, access its data stored in the cache, and every other possible task.

- **New event** - Event arrives from Protocol Adapter
  1. Event arrives in the layer coming from a topic, in a JSON format. The header of the event contains the tenant identifier;
  2. The scheduler (ngix) gives the task to some of the available nodes;
  3. With the tenant identification, the node knows to whom the request belongs to, being thus able to connect to the right schema, access its data stored in the cache, and every other possible task;
  4. The needed validations are done to the event;
  5. It is stored in the right schema and sends the updated information to the right topic.

- **Automatic Operations** - Related to Quartz
  1. The tenant identifier is contained in the trigger.
  2. With the tenant identification, the node knows to whom the request belongs to, being thus able to connect to the right schema, access its data stored in the cache, and every other possible task.

Pros and cons

Explained the architecture, it's clear that less resources are required to execute the requested tasks, which is this solution's main advantage. However, problems regarding isolation, customization and complexity arise.

5.4.2 Dedicated processing

In this solution, processing nodes are dedicated to the tenants. Therefore, it is only required that the layer knows which tenant it belongs to. Only the database access and Kafka topics require modification for this solution.

Database

For a tenant to access its own schema in the database, it is changed the connection string, by adding the tenant's credentials and schema name.

For this, it is expected for the schemas with the expected tables to exist and a user with authorization to access them.

Kafka

Regarding the Kafka solution it should be prefixed the tenant ID to the topic name string for both the producer and the consumer. Since the processing node is dedicated to the tenant, it only subscribes and produces to the topics with its own tenant identifier.

Pros and cons

Having dedicated processing instances is a simpler solution that does not affect deeply the current Alarm Manager structure. Along the same lines, it does not offer any improvement, serving only to connect the previous solutions proposed.

5.4.3 Solutions comparison

Having presented the two solutions, these need to be weighted so as to understand the best option for the problem. Starting with a **shared Processing**, it is a cheaper option, needing less resources to complete efficiently its tasks. However, it presents some problems regarding performance isolation, customization and complexity. It is a solution that imply a lot of changes in the core of the project.

Due to its complexity, even though it is a cheaper solution, the changes that would be required to share the processing layer may be expensive for the possible gains. Because of this, before defining if changes are to be made to this layer, a **effort vs gain analysis** is relevant.

Resources analysis

In order to analyse how resources are used and how can the product scale, it is required to understand how much a Processing node currently uses, how expensive is the change, and how will a multi-tenant solution help make the processing cheaper.

Currently, each processing node consumes, in average, 1 CPU and 2 GB of RAM. However, in order to start an instance, these resources must be duplicated. Hence, each node instance requires 2 CPUs and 4GB of RAM. These resources grow linearly with the number of tenants' applications.

Each machine hosts at least **two instances** and has the following specifications:

8vCPU | 32 GiB Memory

General Purpose SSD (gp3) - 512Gb

10Gb outbound traffic

This reflects in 21 756.96 € each year, i.e, 10 878.48 € per tenant, every year. And so, currently, approximately **87 027.84 €** are spent every year to host the eight clients that use the Alarm Manager.

Cost machine per month (2 instances)	1 813,08 €
Total costs for a machine per year	21 756,96 €
Cost for a client per year (one instance)	10 878,48 €
Total costs per year (8 instances/4 machines)	87 027,84 €

Table 5.4: Machine costs per year

An estimation of the implementation days was obtained communicating with the team: the multiple tasks were gathered, validated, and estimated (see table 5.5). It is not an accurate value, however it reflects the team's expectations. Thereby, the implementation of a shared solution is estimated to last about **850 days**, for a single developer, it indicates a total of 180K euros. Divided in three years it is 60 000 € every year.

Task	**Duration (in days)**
Tenant identification in events	2 - 3
Configure cache 27 caches Change load structure, fetch from cache, answers from cache; adapt all caches	54 (2 per cache)
Configure external APIs (11 APIs) Handle requests, anomalies, inventories, maintenance, and prevention	110 (10 per API)
Configure external APIs (73 EJBs)	219 (3 per EJB)
Database - Implement and test solution (83 datasources, 48 JPAs)	393 (3 per datasource and JPA)
Database - migration for a multi-schema version	10
Configure Quartz (25 timers)	50 (1-2 days per timer)
Statistics	15
Total	850

Table 5.5: Planned tasks for the implementation of a shared processing: The tasks were gathered, validated, and estimated with the team's assistance

Now, for a multi-tenant, for the following resources,

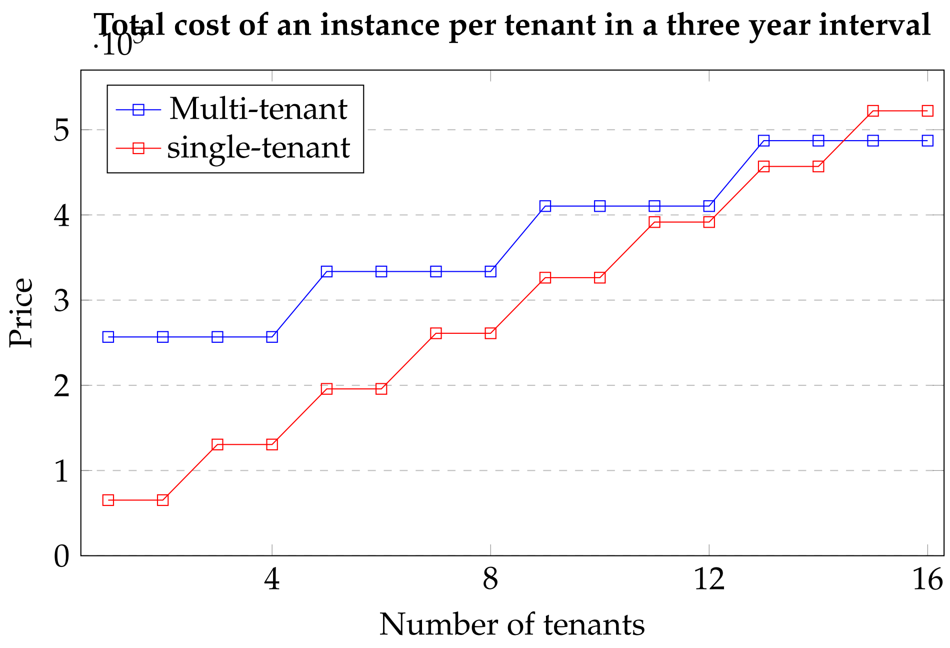
16vCPU | 64 GB Memory (+2CPUs/4Gb per tenant)

General Purpose SSD (gp3) - 1024Gb (+128Gb per tenant)

50Gb outbound traffic (+10gb per tenant)

Considering that each of these machines can host up to four tenants using the current amount of resources, the cost of the application is the same from one to four tenants. For the fifth, a new machine is required and therefore the cost goes up. Each of these machines costs up to 25 357.08 € per year.

For a three year case, it is estimated the following costs of maintaining a given number of tenants:



As seen, at the end of the three years, summing up with the amount required to implement multi-tenancy in the AM, single-tenancy ends up being cheaper even with fourteen tenants. Given that currently the Alarm Manager has eight clients and that a new one subscribes to the product every few years, fourteen tenants is not a conceivable value for the near future.

Therefore, the implementation of a shared processing is currently a task too expensive for the gains, given the number of existing tenants. However, it is still an option that the product might benefit from in the future.

5.4.4 Dedicated instances as a solution

Given the high costs for a multi-tenant solution for the Processing layer, dedicated instances for each client is shown to be the best option for the Alarm Manager case, in the near future. Thus, a solution for Processing includes changes in how the database is accessed, as well as the Kafka topics.

In chapter 7, this solution is implemented and reported.

Chapter 6

Development plan

Before starting to gather a concrete solution for the problem, it is crucial to plan the project. This is an important step that must be done at the beginning of the process since it's when its lifecycle is defined, which provides the basic framework for management, and the steps for the project completion are established. This chapter's focal point is the project's **approach**.

6.1 Lifecycle

Throughout the whole year, the team responsible for the Alarm Manager, and to which I was added, works in a **Scrum methodology**. Every day, the team meets to discuss what has been done and what is planned for the day. This is also known as **daily**. Moreover, every two weeks, a sprint is closed and a new one is opened, with tasks being estimated and delivered to team members.

Regarding the dailies, in the first semester, I only attended these twice a week (Thursdays and Fridays), and in the second semester, I participated daily. In both semesters these had the purpose of giving and receiving feedback about my work with the team, allowing sometimes to brainstorm about new approaches that I could explore. Since most of my work was researching or, when implementing, it was never directly in the final product, I was not expected to participate in tasks as such as estimate story points for requirements. The purpose of this lifecycle was to experiment the process of working in a real development team, such as how it is managed and its methodologies.

Besides these daily meetings, in the first semester I met with my internship adviser and two other team members to give updates and receive feedback on my work, every Friday. Furthermore, every two weeks, I also met with my thesis advisor, so as to discuss my dissertation work. During the second semester, I started meeting with my internship advisor twice a week (Thursdays and Fridays), and sporadically with my thesis advisor. Also, every month, throughout the whole internship, I met with my internship and thesis advisors, in order to guarantee that the interests of both the company and department were aligned. Finally, if requested or thought necessary, other meetings were scheduled with other *Altice* workers or project members, in order to get more insight into specific topics.

This project is then divided into four parts [37][49]:

1. **Starting the project** - Identify and study the problem;
2. **Organizing and preparing** - Breaking down the project into small tasks (requirements), analysing risks, and identifying the project timeline;
3. **Carrying out the project work** - Implement planned requirements and monitor the quality of the product, by executing a set of tests to the system;
4. **Closing the project** - Handoff of the work produced and terminate the project.

To schedule all tasks, a *Gantt* diagram for each of the semesters was created, which can be seen in the following sections.

6.2 Planning for the first semester

For the first semester, it was expected to complete the **first** two parts, i.e, "Starting the project" and "Organizing and preparing". The **first** breaks down into tasks such as thorough research on multi-tenancy, the methods to achieve it, and various other topics revolving around it, as well as a study on the Alarm Manager system and its architecture. Furthermore, the **second** part, "Organizing and preparing", focuses on planning both functional and non-functional requirements, the project's constraints, and risks, along with plans to mitigate these. In the first semester, it's expected to work 16 hours/week. All this considered, in figure 6.1, the *Gantt* chart with the planning for the first semester is shown.

The intermediate delivery was due on January 16th and marked the end of the first part of the project. That same day also marks the beginning of the **first sprint** after the intermediate delivery.

6.3 Planning for the second semester

The second semester was focused on detailing, implementing, and testing a solution for the database and the chosen Alarm Manager layers, as per the functional requirements. During the second semester, as opposed to the first one, it was expected to work full-time. The diagram 6.2 describes the planned scheduling for the second semester.

However, this scheduling was not strictly followed and therefore some tasks underwent changes, leading to an execution of the scheduling as followed in figure 6.3. These changes are due to the Processing solution as well as the Kafka implementation taking longer than estimated. When a planning was proposed, a solution for processing was not yet defined, and since the implementation was less deep than expected, it took less time that estimated. Also, testing was not included in the first planning. Therefore, requiring less time to implement, it was easily possible to schedule the testing tasks in the planning. Even so, all mandatory functional requirements were implemented: new tenant setup, authentication and authorization in both Kafka and the database and monitoring.

After the internship, time was allocated to writing the results, and finishing the dissertation, in order to have it completed and reviwed by July 10th.

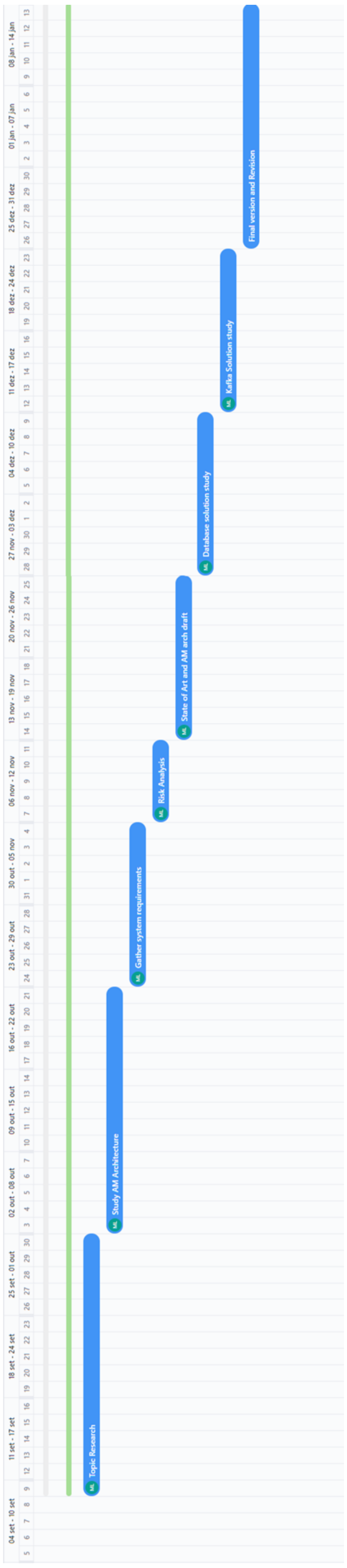


Figure 6.1: Planning for the first semester

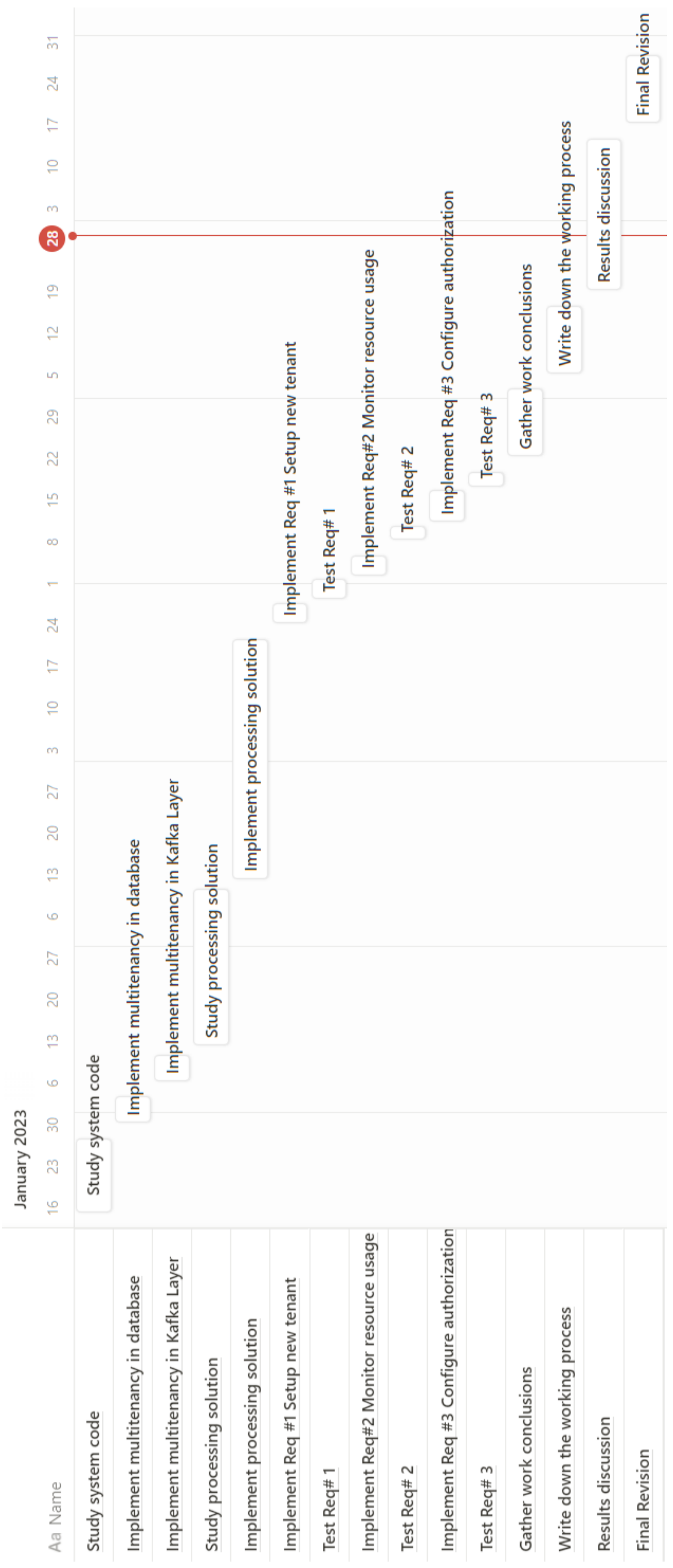


Figure 6.2: Original planning for the second semester

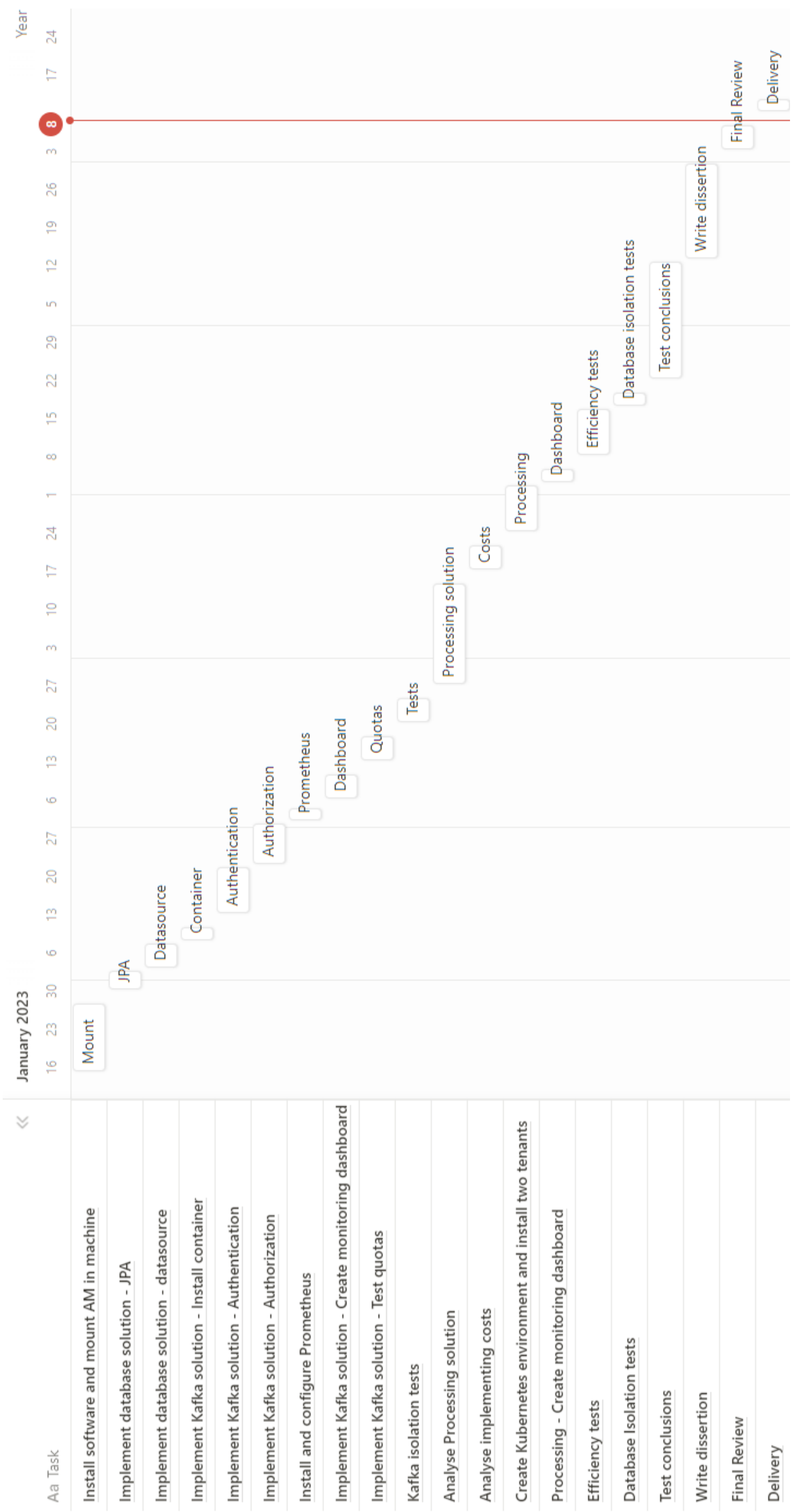


Figure 6.3: Followed schedule for the second semester

Chapter 7

Implementation

In this chapter, the development done throughout the internship is presented. It implements the selected solutions discussed in chapter 5. It has the purpose of supporting and testing their viability and impact. This chapter is therefore divided in four parts: Installing requirements, Kubernetes, Kafka, and other implementations.

As analysed and discussed in chapter 5, the solution is to keep dedicated processing nodes with a shared database and dedicated schemas for each of the tenants, and a shared Kafka namespaces. This scenario is shown in figure 7.1. As referred in chapter 6, some of the implementation is done with the assistance of team members, specially the team tech lead.

The **system** implemented is Kubernetes is composed of two Processing nodes, two Web applications, two Protocol Adapters, a database and a Kafka and Zookeeper clusters. This allows for two simultaneous tenants to receive, process, and display events.

This implementation has the purpose of responding to the **proposed use cases**. Table 7.1 presents the list of use cases from chapter 4, and their implementation status.

	Use case	**Priority**	**Required?**	**Implemented?**
US1	Setup Tenant	High	Yes	Yes
US2	Audit Resource usage	Medium	No	Yes
US3	Monitor Resource usage	High	Yes	Yes
US4	Change/Set Resource usage threshold	Low	No	No
US5	Access authorizations	Medium	No	No
US6	Configure Kafka authentication and authorization	High	Yes	Yes

Table 7.1: Use cases list

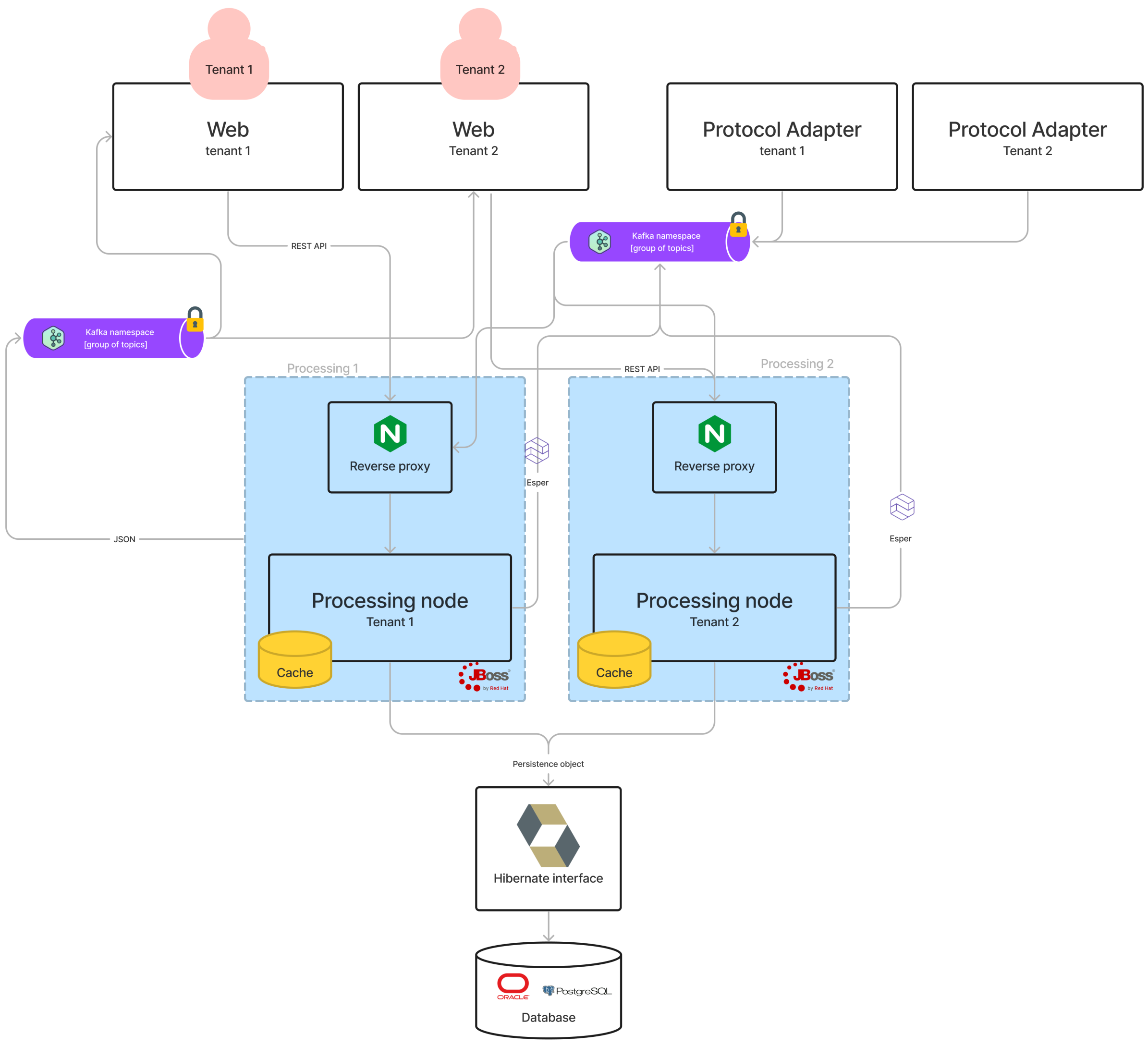


Figure 7.1: Proposed multi-tenant architecture

7.1 Required Tools

In order to implement the necessary changes, some software and tools are required.

Starting with the IDE, **IntelliJ** was used. **TortoiseSVN**, a subversion client that allows to pull and push the project's repository trunk was also used. It is helpful to obtain the project source code, but also to share the modifications done. **Dbeaver** offers a SQL client interface, that allows to view, manage, and execute SQL queries in the database. **Postman**, along with **jMeter** are powerful tools to test the development done. Both are capable of sending HTTP packages to the system. jMeter is used as a mass tester.

For the Kubernetes environment, **Kubectl** is a command line used to manage the cluster. It is required for **K9s**, a tool for providing supervision to the cluster status, as well as its pods. On the other hand, **Docker** is used for the Kafka development, allowing to create and manage a contained environment.

7.2 Kubernetes

Due to technical requirements, the whole AM system is built in a Kubernetes environment called **T-K8S**, a cluster generally used by the team for developing solutions and testing. As a result, a constraint is raised: other systems use the same Kafka pod, and for this reason it is not possible to implement client authentication nor, consequently, client authorization. This impacts the tests done as it generates noise.

The procedure for building the AM system followed a manual procedure using an overlay. An **overlay** is used as a part of **Kustomize**, a tool to customize Kubernetes objects. It is a directory with a *kustomization.yaml* file that is deployed to Kubernetes and that refers to other kustomization directories, known as bases. A **base** contains a set of resources and associated customizations. It combines Wordpress and MySQL format. The construction of the overlay was done from a sample already existing in the project and with the assistance of the tech lead. Other changes, which were done directly in the processing source code, were made by team members. This only requires the prefix to be added to the topic names.

7.2.1 Configure Overlay

As mentioned, most work was done in a Kustomize overlay, using an already existing sample that is able to simplify the deployment of a new instance.

And so, the **first step** for the creation of a tenant instance is to setup a new Kubernetes namespace that will host the instance. Since permissions are required to complete this step, it was required the assistance of a third party to create the namespace.

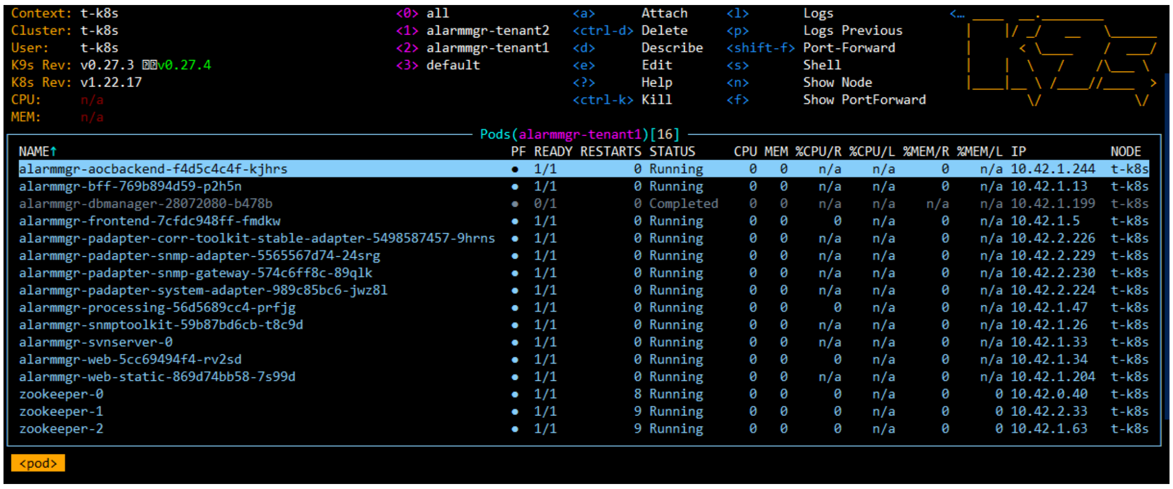
Afterwards, the next step is to modify the overlay as needed. The solution requires to link the overlay to the newly created namespace, define the application URL, which packages (which are **bases**) are desired for the tenant, and configure database and Kafka solutions. These steps are done by defining variables that are stored in a **ConfigMap** and that are called from inside the system.

Next, to configure the NFS path, it is required to create the directory manually, that must be specified in *storage-persistentvolume.yaml*.

Before finishing the process, a license must be generated. This licence is based on the namespace and is intended to verify its validity.

The last step is **installing the instance** using a **kubectl** apply command, that will patch the yaml file variables to the system variables. Every time a change is made, its just a matter of reapplying the overlay to update the instance.

The pod's status can be consulted in K9s. The following image 7.2 shows an example of the displayed information:



```
Context: t-k8s                                            <0> all               <a>       Attach     <l>       Logs              <…
Cluster: t-k8s                                            <1> alarmmgr-tenant2  <ctrl-d>  Delete     <p>       Logs Previous
User:    t-k8s                                            <2> alarmmgr-tenant1  <d>       Describe   <shift-f> Port-Forward
K9s Rev: v0.27.3 ⚡v0.27.4                                  <3> default           <e>       Edit       <s>       Shell
K8s Rev: v1.22.17                                                               <?>       Help       <n>       Show Node
CPU:     n/a                                                                    <ctrl-k>  Kill       <f>       Show PortForward
MEM:     n/a
                                                            Pods(alarmmgr-tenant1)[16]
 NAME↑                                                   PF READY RESTARTS STATUS     CPU MEM %CPU/R %CPU/L %MEM/R %MEM/L IP           NODE
 alarmmgr-aocbackend-f4d5c4c4f-kjhrs                     ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.1.244  t-k8s
 alarmmgr-bff-769b894d59-p2h5n                           ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.1.13   t-k8s
 alarmmgr-dbmanager-28072080-b478b                       ●  0/1          0 Completed    0   0    n/a    n/a    n/a    n/a 10.42.1.199  t-k8s
 alarmmgr-frontend-7cfdc948ff-fmdkw                      ●  1/1          0 Running      0   0      0    n/a      0    n/a 10.42.1.5    t-k8s
 alarmmgr-padapter-corr-toolkit-stable-adapter-5498587457-9hrns ●  1/1   0 Running      0   0    n/a    n/a      0    n/a 10.42.2.226  t-k8s
 alarmmgr-padapter-snmp-adapter-5565567d74-24srg         ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.2.229  t-k8s
 alarmmgr-padapter-snmp-gateway-574c6ff8c-89qlk          ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.2.230  t-k8s
 alarmmgr-padapter-system-adapter-989c85bc6-jwz8l        ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.2.224  t-k8s
 alarmmgr-processing-56d5689cc4-prfjg                    ●  1/1          0 Running      0   0      0    n/a      0    n/a 10.42.1.47   t-k8s
 alarmmgr-snmptoolkit-59b87bd6cb-t8c9d                   ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.1.26   t-k8s
 alarmmgr-svnserver-0                                    ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.1.33   t-k8s
 alarmmgr-web-5cc69494f4-rv2sd                           ●  1/1          0 Running      0   0      0    n/a      0    n/a 10.42.1.34   t-k8s
 alarmmgr-web-static-869d74bb58-7s99d                    ●  1/1          0 Running      0   0    n/a    n/a      0    n/a 10.42.1.204  t-k8s
 zookeeper-0                                             ●  1/1          8 Running      0   0      0    n/a      0      0 10.42.0.40   t-k8s
 zookeeper-1                                             ●  1/1          9 Running      0   0      0    n/a      0      0 10.42.2.33   t-k8s
 zookeeper-2                                             ●  1/1          9 Running      0   0      0    n/a      0      0 10.42.1.63   t-k8s

<pod>
```

Figure 7.2: Tenant 1 pod.

As discussed in 5.2, a solution for the **database** requires to change how the database is accessed. Currently, a user logs in and is redirected to a public schema. The purpose is to have all tenants accessing the same database, and having a different schema for each one.

Therefore, the only two differences for the current procedure is that all tenants will have the same host (variable *db_host*) with different schemas.

This solution also requires some additional steps, in order to create the schema's tablespaces owned by each tenant. This must be done in SQL.

In order to implement the Kafka selected solution, the application must be aware of what to prepend to the topic. As previously referred, configMap is an object that is able to patch values to the system from a yaml. Hence, this is possible by defining a new variable in *configmap-ansiblevars.yaml*. In the system's source code, every time the topic name is called, it comes with the tenant identifier prefixed. This is done by acquiring the tenant name value from the *configMap* and prepend it to the topic name every time it is requested.

Summarizing, this development responds to the **use case 1: "Setup tenant"**, by defining the manual steps to create a new instance so as to host a new tenant.

7.2.2 Monitoring

The AM system already offers many metrics for monitoring and troubleshooting. Since most can already be separated by namespace, and each tenant instance is hosted in a different namespace, it is easily possible to obtain metrics per tenant. Thus, a dashboard was created for the purpose of monitoring and testing the cluster behavior, containing the following metrics:

- **Cluster CPU utilization** - Percentage of CPU resources being used in a given moment. This is global to the whole cluster;
- **Cluster Memory utilization** - Percentage of the currently allocated memory from the available memory. This is global to the whole cluster;
- **Processed events** - Tells how many events are being processed in an interval per tenant;
- **Event Processing time** - Tells how much time was spent since the event came into the Processing and left it in an interval per tenant;
- **Query processing time** - Tells how much time was needed in average for a query to be processed in the whole process inside the layer in an interval per tenant;
- **Cache received messages** - Tells how many cache accesses are being done in a given instant per tenant.
- **CPU Usage** - Tells how many bytes of the CPU are being used in an interval per tenant;
- **Memory usage (GB)** - Tells how many bytes of the memory are being used in an interval by tenant;
- **Current rate of bytes received/transmitted** - Tells how many bytes from event packets are being received and transmitted in a given moment by tenant;
- **Received/transmitted bandwidth** - Tells the capacity of the cluster to receive/transmit data in an interval by tenant;
- **Rate of received/transmitted packets** - Tells how many event packets are being received and transmitted in an interval by tenant;
- **Rate of received/transmitted packets dropped** - Tells how many packets have been dropped in an interval per tenant.

The following figures, 7.3 and 7.4 , show the dashboard when events are being sent to two tenants:

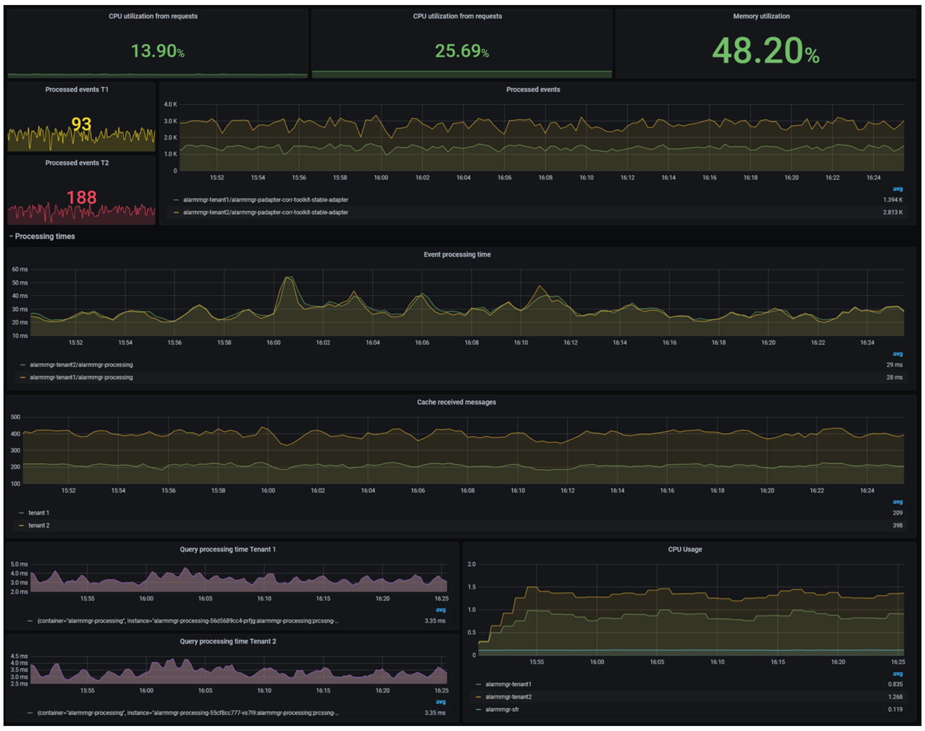


Figure 7.3: Monitoring Dashboard

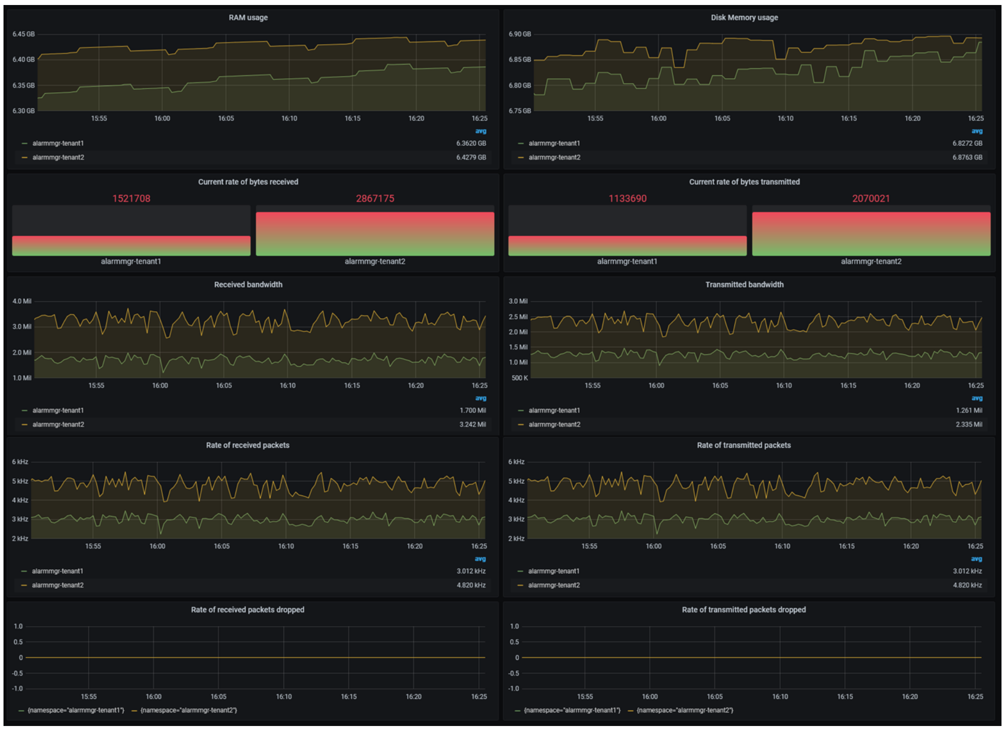

RAM usage
Disk Memory usage
Current rate of bytes received
1521708
2867175
Current rate of bytes transmitted
1133690
2070021
alarmmgr-tenant1
alarmmgr-tenant2
Received bandwidth
Transmitted bandwidth
Rate of received packets
Rate of transmitted packets
Rate of received packets dropped
Rate of transmitted packets dropped

Figure 7.4: Monitoring Dashboard

The dashboard responds to the **use case 3: "Monitor resource usage"**, by presenting a dashboard that displays the metrics required to monitor a client's resources.

7.3 Kafka Docker

A complete solution regarding Kafka was also implemented separately, since the namespace is shared with other applications, making it impossible to implement authentication and authorization. And so the development of the Kafka solution was done in a containerized environment, using an image from **bitnami**[1]. To notice that this development was done without support from the AM team.

Remember that the purpose is to build a Kafka solution such as depicted in figure 7.5, with multiple applications connecting to the same cluster, authenticating using SASL authentication (in this case plain SASL) and using different topics (depending on its prefix name).

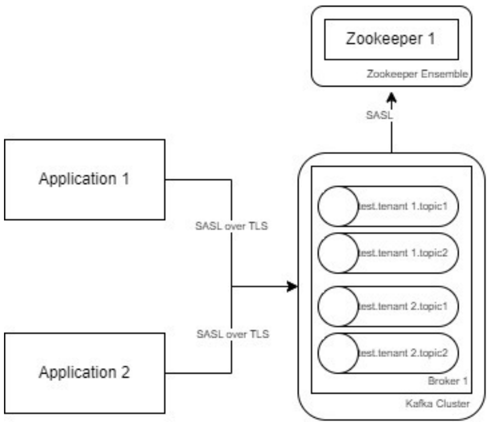


Figure 7.5: Kafka structure

For this implementation, it is required to install a Kafka, Zookeeper, Grafana, and Prometheus services. These services are created using a **docker compose** yaml file containing:

- A Zookeeper service (also bitnami), including a JAAS file and its configurations, so as to be mounted alongside the service.
- A Kafka service, which mounts a JAAS authentication file, custom server properties, users' property files and theirs credentials, as well as the required prometheus files: its javaagent and a configuration file.
- A Prometheus service, with its port configured and a yaml file containing its metrics rules (in this case, none was defined) and the specification of the job to be monitored, as well as its port.

[1]https://hub.docker.com/r/bitnami/kafka/

- A Grafana service, with its port configuration.

The configuration file for both the Kafka and Zookeper services is built for it differs from the original configuration, due to the authentication and authorization mechanisms. This way, when the container starts it is already totally configured. The following sections detail this configuration process.

7.3.1 Authentication

To enable authorization in a Kafka cluster it is required to enable the intended SASL authentication mechanisms in the server properties, in this case, **PLAIN** authentication. Because of this, it is expected that the variables, such as *sasl.enabled.mechanisms* and *sasl.mechanism.inter.broker.protocol*, contain the value PLAIN. It is required to configure listeners and protocols. In zookeeper, it is also needed to specify that the authentication scheme is SASL.

These changes only tell which authentication mechanism is being used. In order for users to authenticate, the servers must acknowledge them and therefore be aware of their credentials. For this, a JAAS configuration file contains the list of the existing users. However, it only specifies which users can authenticate, but not their credentials. These must be stored either separately or together in a properties files, so that, when authenticating, the service can link the user to its credentials. The following example shows the structure of a JAAS configuration file, giving server access to four users: admin, Alice, Bob, and Charlie.

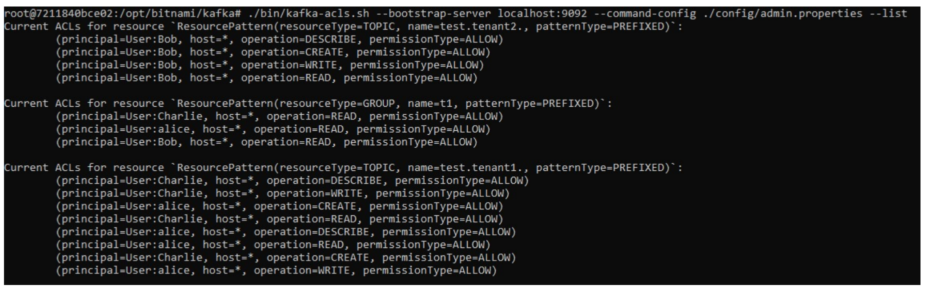
```
    Server {
        org.apache.kafka.common.security.plain.PlainLoginModule
required
        username="admin"
        password="admin"
        user_admin="admin"
        user_alice="alice"
        user_bob="bob"
        user_charlie="charlie";
    };
```

Thereby, every action that is done, either create or delete a new topic, produce or consume events, demands user credentials, passed alongside each command. When adding a new user, it is required to add manually the JAAS configuration and to provide the user credentials.

7.3.2 Authorization

Besides authentication configuration, some simple steps must be executed before creating ACLs for each tenant. In both Kafka and Zookeeper services, the authorization must be enabled in the configurations files, by adding the lines *authorizer.class.name=kafka.security.authorizer.AclAuthorizer* and *zookeeper.set.acl=true*, respectively. Besides this, it must be taken into account that users with no ACLs must have all access blocked and, since this is not the default configuration, it is mandatory to set this configuration.

Having completed these steps, it is possible to create ACLs for each existing user, either through the command line or using the Client Admin library in Java. These can be created using the topic prefix. ACLs can be listed, returning a list of all rules, divided by topic prefix, as seen in the following figure:



```
root@7211840bce02:/opt/bitnami/kafka# ./bin/kafka-acls.sh --bootstrap-server localhost:9092 --command-config ./config/admin.properties --list
Current ACLs for resource `ResourcePattern(resourceType=TOPIC, name=test.tenant2., patternType=PREFIXED)`:
        (principal=User:Bob, host=*, operation=DESCRIBE, permissionType=ALLOW)
        (principal=User:Bob, host=*, operation=CREATE, permissionType=ALLOW)
        (principal=User:Bob, host=*, operation=WRITE, permissionType=ALLOW)
        (principal=User:Bob, host=*, operation=READ, permissionType=ALLOW)

Current ACLs for resource `ResourcePattern(resourceType=GROUP, name=t1, patternType=PREFIXED)`:
        (principal=User:Charlie, host=*, operation=READ, permissionType=ALLOW)
        (principal=User:alice, host=*, operation=READ, permissionType=ALLOW)
        (principal=User:Bob, host=*, operation=READ, permissionType=ALLOW)

Current ACLs for resource `ResourcePattern(resourceType=TOPIC, name=test.tenant1., patternType=PREFIXED)`:
        (principal=User:Charlie, host=*, operation=DESCRIBE, permissionType=ALLOW)
        (principal=User:Charlie, host=*, operation=WRITE, permissionType=ALLOW)
        (principal=User:alice, host=*, operation=CREATE, permissionType=ALLOW)
        (principal=User:Charlie, host=*, operation=READ, permissionType=ALLOW)
        (principal=User:alice, host=*, operation=DESCRIBE, permissionType=ALLOW)
        (principal=User:alice, host=*, operation=READ, permissionType=ALLOW)
        (principal=User:Charlie, host=*, operation=CREATE, permissionType=ALLOW)
        (principal=User:alice, host=*, operation=WRITE, permissionType=ALLOW)
```

Figure 7.6: ACL list example

This defines which action each user can or cannot perform, and which resources it has access to or not. This way, **use case 4 : Change/Set Resource usage threshold dashboard** is implemented and authentication and authorization are possible inside the cluster.

7.3.3 Monitoring

In order for Prometheus to gather Kafka and Zookeeper metrics, it is required a JMX exporter jar, known as **javaagent**, and a JMX configuration file. On one hand, the javaagent exposes the application's JMX objects for Prometheus to consume, and on the other, the JMX configuration file itemizes a set of rules that define which metrics are caught and grouped. However, Kafka needs to know that these metrics are to be caught, and therefore, both *prometheus.jmx.enabled* and *prometheus.kafka.enabled* must be set to true.

These metrics are next used by Grafana, as displayed in figure 7.7, in order to create a dashboard to monitor and troubleshoot the tenant usage and brokers' health.

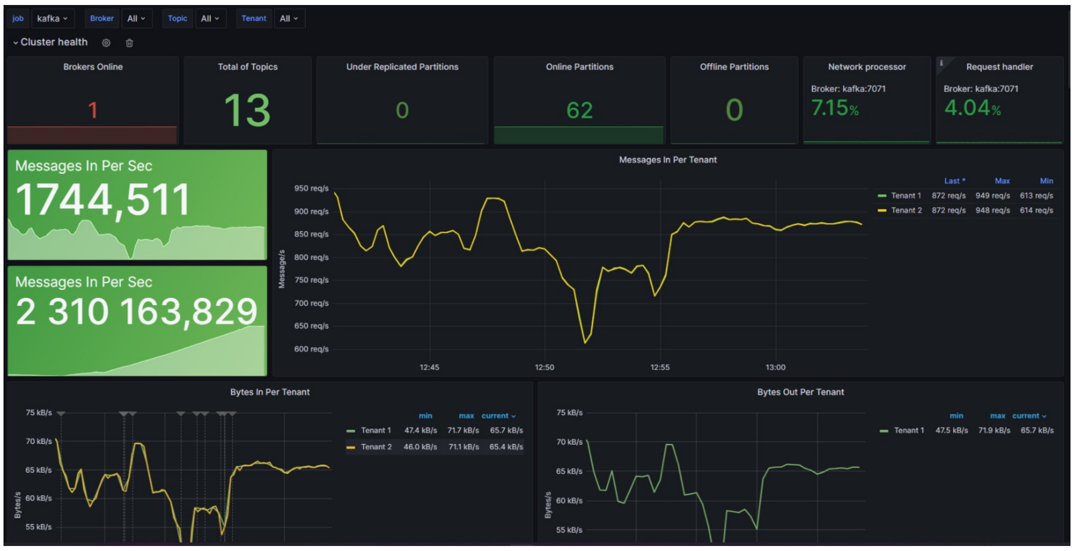


Figure 7.7: Kafka Dashboard

This dashboard also allows to create alerts. As previously referred, it is not expected for resources to be throttled when a user surpasses a threshold. Instead, it is preferable for a NOC employee to be notified and some measure be taken. This is done by setting threshold values in the queries. As seen in the following example 7.8, when a value exceeds the threshold for more than a given time (in the example is 20 seconds), alarm is fired:

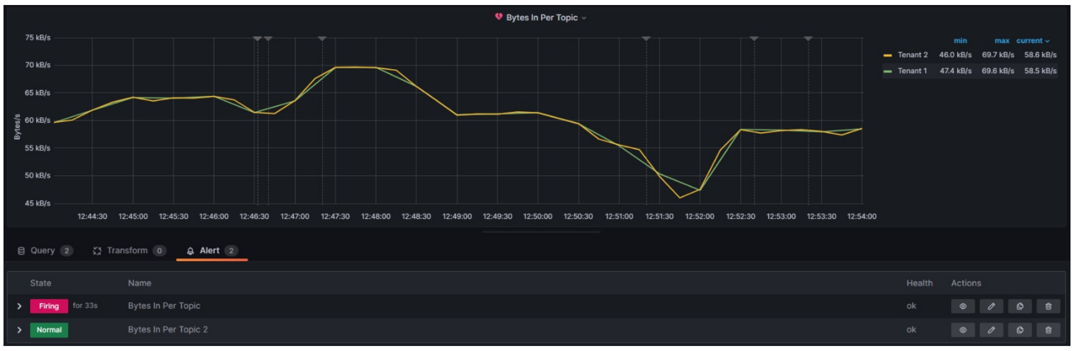


Figure 7.8: Alert for number of bytes being consumed surpassed

In this case, both tenants are making a similar utilization of the resources, however tenant 1 has the threshold set for 45KB/sec and tenant 2 set for 85KB/s, resulting in only tenant 1 firing the alarm for excess utilization of consumed bytes.

The dashboard responds to **use case 2: Audit Resource usage** and **use case 3: "Monitor resource usage"** , by presenting a dashboard that displays the metrics required to monitor a client's resources and defining alerts when resource usage is exceeded.

7.4 Other developments

In order to support the shared processing solution, a Maven project was created with all the necessary dependencies. It replicated one of the system packages, **Procedures Manager**, allowing to implement some parts of the solution in a sample of the project, in order to test whether the solution was viable or not, and for future work. And so, the solution for both datasources and JPAs, the authentication of a tenant in the REST API, and the labels in Prometheus metrics, in order to separate tenant specific metrics were implemented.

Chapter 8

Tests

Having assessed and discussed multiple solutions, identified, and implemented the most adequate, it is now important to test the resulting system. In accordance with the non functional requirements, it is required that the system is efficient, improves the resource usage, and assures confidentiality.

Therefore, the purpose of this chapter is to present and analyse the **tests** conducted to the system. It is divided into three parts: the first focuses on performance tests carried out in the Kubernetes environment examining **how the system reacts to different event loads**. The second section evaluates the **confidentiality aspects of both Kafka and the database**, by verifying whether tenants can or cannot access resources belonging to others. Finally, the third section analyses and discusses the results.

8.1 Performance tests

In order to test the performance of the proposed solution, it is considered important to answer the following questions:

1. How do resources such as CPU, memory, and bandwidth evolve with crescent load of events?
2. How does the system reacts to crescent load of events? At what point does it start degrading?
3. How does the system scales for supporting more tenants? What influences this behavior?

For this, events are built and injected using **jMeter**. A bean shell pre-processor allows to build a Json object by defining what variables are to be sent in the event, and their values, so as to send them to the system to be processed with different throughput. The system is composed of a Protocol Adapter, a Processing ,and a Web application.

All tests followed the same **procedure**:

- One or two simultaneous tenants receive events. A **tenant** is a complex instance that receives events and that is able of interacting with multiple users with permissions to. So, even though, it was not possible to test with more tenants due to technical restrictions, having two tenants per instance is already an advantageous scenario.
- Varying number of events are sent, ranging from 10 to 250 events per second, grouped by three categories: low load (10 to 50 events/sec), normal load (70 to 135 events/sec), and high load (170 to 220 events/Sec). These levels are representative of loads from real clients.
- All tests had the duration of 10 minutes and were executed three times. The presented values are the average of all the experiments. The result of the experiments of multi-tenancy are the average of the values obtained by the two tenants.

In most cases presented, both tenants were tested with the same loads simultaneously, which is not usual in a real situation. This represents the **worst case scenario**.

These tests were run in a Kubernetes test environment, used generally for testing. This means that on one side, the machine has more than the minimum resources, limiting to the analysis of performance degradation under high loads. On the other side, the environment is shared with other applications concurrently, which may influence the results.

This decision was done since the resources are already available and the time it would take to explicitly set up the environment merely for these preliminary tests was not worth the cost. Nevertheless, these tests are sufficient to draw conclusions regarding the behavior of the instance. Each tenant uses its own namespace, however, the Kafka namespace is also shared with other applications. To note, as well, that these results refer to the process of receiving, parsing, analysing, storing, and displaying each event.

Table 8.1 shows the resources available in the current environment, the minimum resources required, and the specifications of the machine being used to inject the events, since it affects the amount of tests sent:

	Current resources	Minimal resources	Testing machine
CPU (cores)	60	2	2
Memory (GB)	> 300	32	8
Bandwidth (GB/s)	-	10	-

Table 8.1: Environment current, minimal resources, and testing machine resources

It should be noted that these tests are run in **development mode**. This means that results are to be higher than expected. For example, an event that can be processed in 4 ms in production mode, might take about 30 ms in development mode.

To monitor the machine's behaviour, the cluster metrics are gathered and displayed in a Grafana dashboard, previously presented in 7.2.2.

8.1.1 Idle mode

Before diving into the test conclusions, it is important to note that even when idle, resources are always being consumed. This detail influences the results from the test and therefore must be taken into account. For example, tenant CPU usage is always between 0.13 and 0.3 CPUs, while Kafka CPU utilization is always around 0.103.

Also regarding the memory usage, its usage ranges from 6,2 and 6,7 GB. Following the next example, as depicted in the graph 8.1, from 2h30 to 8h00, there are no events being received and processed. However, from 8.2, the behaviour is not so linear, being that from 3h50 it starts growing steadily.

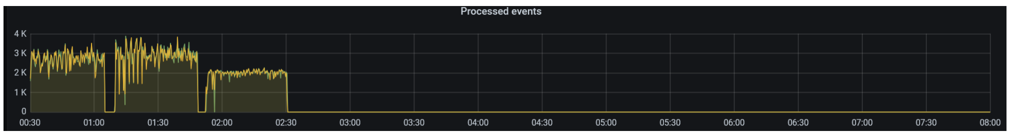
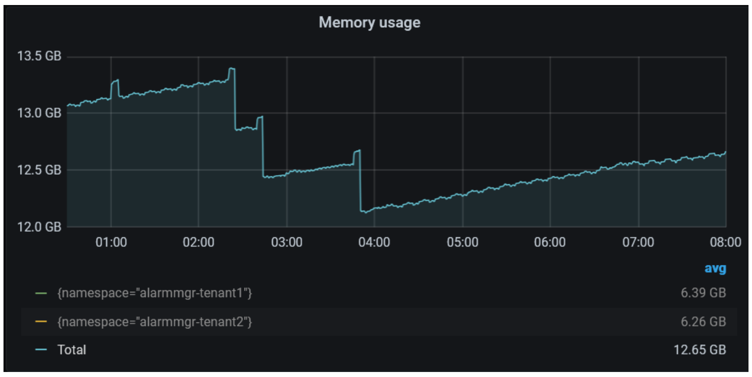


Figure 8.1: Processed events from 2h30 to 8h00



	avg
— {namespace="alarmmgr-tenant1"}	6.39 GB
— {namespace="alarmmgr-tenant2"}	6.26 GB
— Total	12.65 GB

Figure 8.2: Memory activity

This is due to external and automatic activities, such as Quartz jobs and cache updates.

8.1.2 Resources evolution

Starting with the first question: **How do resources such as CPU, memory, and bandwidth evolve with crescent load of events ?**, it is expected that the system is capable of distributing its resources equally per event, and that the quantity of used resources per tenant grow the more events are processed. Hence, for the same number of events, users utilize the same amount of resources to maintain performance. However, it is expected that different number of events being processing lead to imbalance between tenants resources. In a limited scenario, this would cause noisy-neighbors. This way, the curve for the three of the resources per tenant is expected to be linear, the more events are being sent.

However, it is not expected for the sum of all resources to be linear to the number of tenants. This is due to shared components, such as Kafka resources.

This way, it was obtained the CPU, memory and bandwidth usage for a single and a multi-tenant scenario with two tenants.

CPU

Graphs 8.3 and 8.4 displays the results regarding the **CPU usage per tenant**, depending on the number of tenants and events per second, and the case of **two concurrent tenants**, processing the same number of events, respectively.

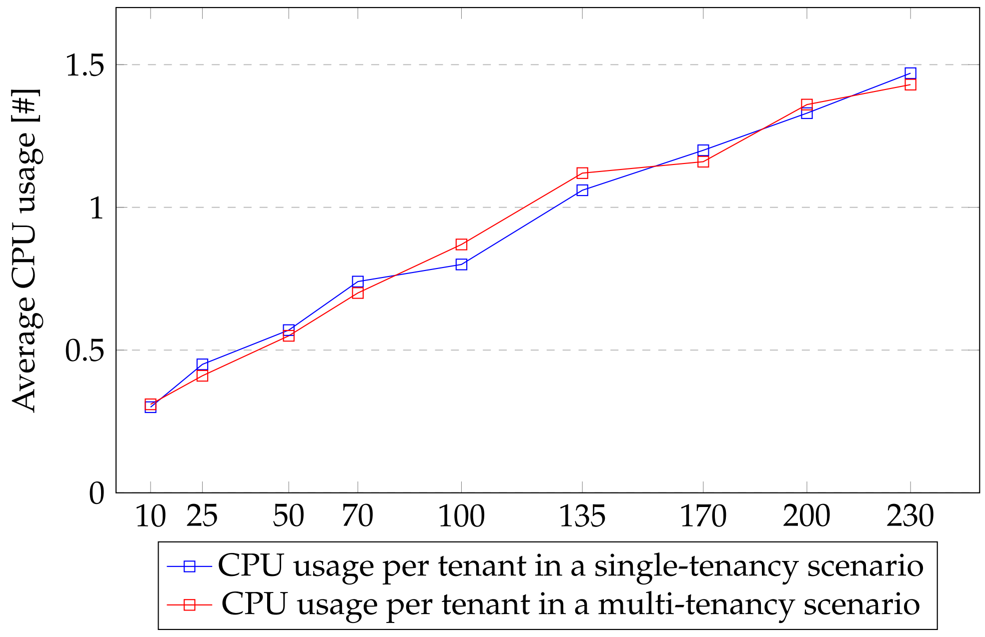


Figure 8.3: CPU evolution

As expected, the CPU values grow linearly, as the number of events that are sent per second increases. Also, in both cases tenants have access to the same quantity of resources, even though the number of tenants duplicated, which is easily seen in the second graph. This means that, even in a scenario in which resources are

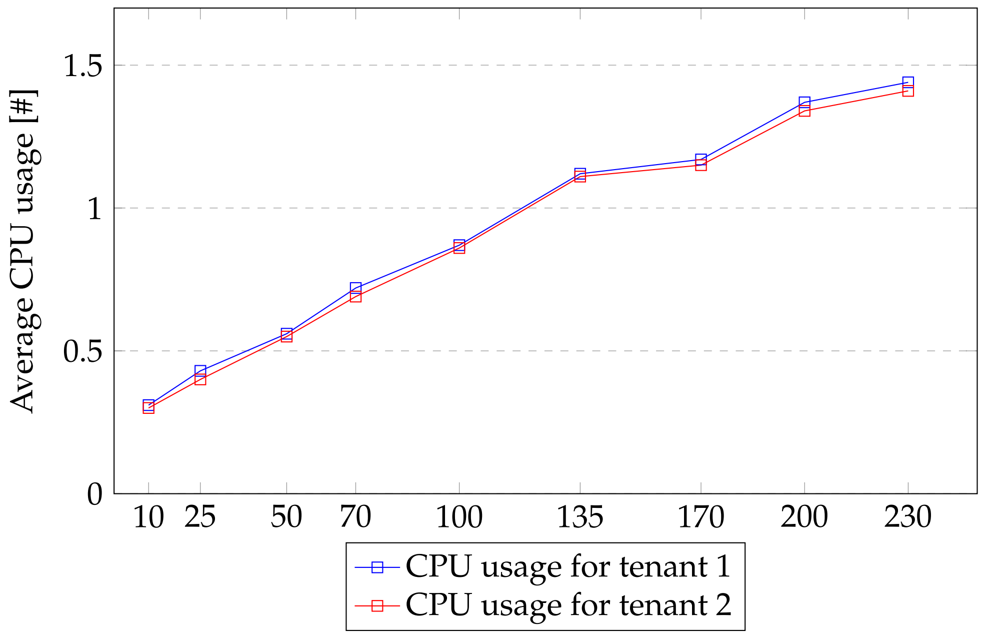


Figure 8.4: CPU usage per tenant

limited, it is expected that all events, whichever tenant they belong to, have access to the same amount of CPU resources and might have them throttled, imposing a threat to the workload isolation.

Considering, nonetheless, that most of the time events are not sent equally, it is yet notorious how the CPU usage depends on the number of events. For example, for a scenario in which a tenant is processing two times the events of another tenant, the CPU usage grows in about 50%. In case this factor grows to four times, the CPU usage increases to 100%. In both cases, receiving a constant number of events, the other tenant will have also constant resource usage. This variation in the CPU usage, and similarly for the memory and bandwidth, allows to conclude that resource usage is done per event and therefore both tenants can process events at a similar rate, independently of the number of events arriving.

However, in general, less CPU is utilized for the case of multi-tenancy, due to the shared Kafka namespace. The graph 8.5 shows that the CPU usage improves.

Table 8.2 presents the average decrease of CPU compared with the usage done by one tenant. Above two tenants, the values were estimated with the difference between the percentage of CPU usage between one and two tenants and by keeping a constant CPU usage for the Kafka layer. To note that the resource usage for Kafka, in a multi-tenant architecture, varies little with the increasing number of tenants, whereas, having multiple clusters, is more expensive. It is thus a **projection** that followed a **linear progression**.

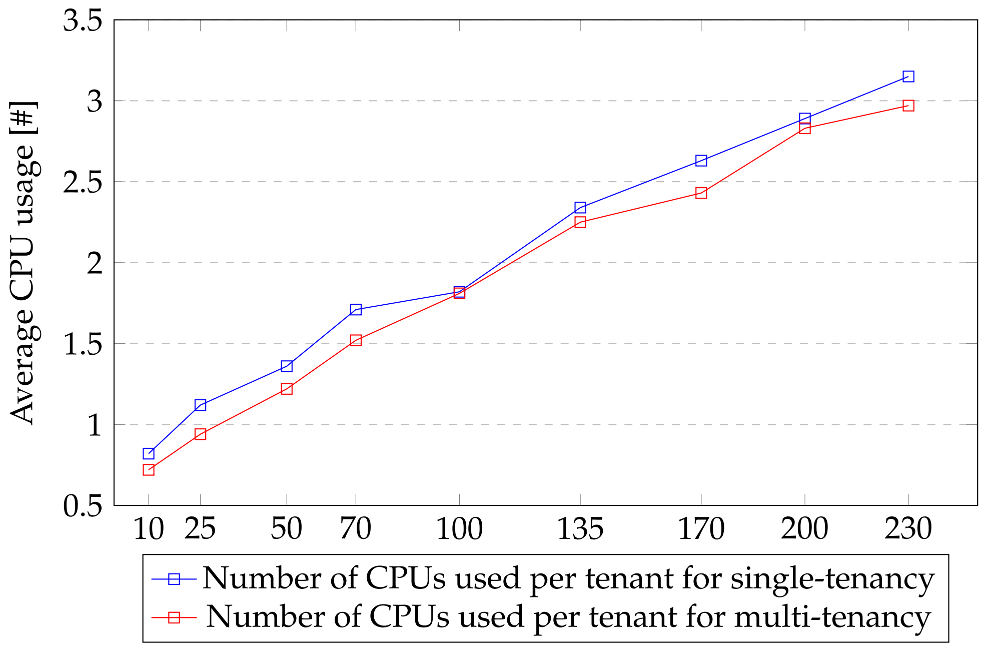


Figure 8.5: Comparison of CPU in a single and multi-tenant scenarios for two tenants

Tenants	Average decrease in CPU
2 Tenants	12 %
3 Tenants	22 %
4 Tenants	34 %
5 Tenants	45 %

Table 8.2: Projection of the general CPU reduction for number of tenants, due to the shared Kafka

And so, starting from two tenants, a multi-tenant scenario offers at least a decrease of 12% in CPU usage, mostly due to the shared Kafka. This value increases in about 11% by each tenant that is added. Therefore, the more tenants, the bigger the gains.

Transposing this to a minimal scenario, presented in 8.1, it is possible to notice that these improvements are not enough to host at least two tenants in minimal conditions in normal load. As depicted in the following graph that shows the CPU usage for one and two tenants, and a projection for a third one, at least 4 CPUs are required to host two tenants. For lower to normal usage, up to three tenants. For a higher number of tenants and in order to keep the stability of the distribution of the resources, more resources need to be added.

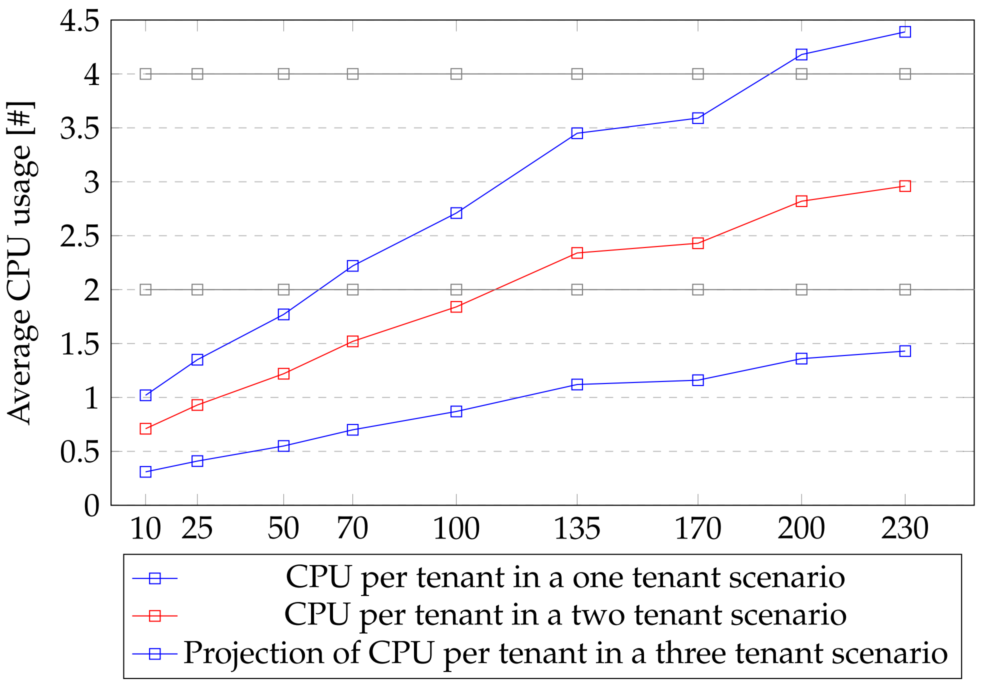


Figure 8.6: CPU usage for different concurrent tenants

Memory

Regarding **memory usage per tenant**, it is notorious that these values do not range much, normally between 6.4 to 6.85 for bigger loads. Furthermore, when comparing with two different tenants, is seen that they diverge little as well. The graph 8.7 shows how these memory resources are used depending on the number of tenants and events per second:

As shown, testing with two tenants in the same conditions, it is demonstrated that the amount of memory grows the more events are being processed. Compared with the single-tenant case, it is possible to see that, in general, more memory is used per tenant, in average 90MB. This small difference little impacts the system.

Bandwidth

Finally, related to the **network bandwidth per tenant**, which is the capacity of the network to transmit and receive data, the available resources for the scenario with one tenant is similar to two tenants, as shown in graph 8.8.

As shown, in both cases a tenant requires the same amount of resources to work properly. Having similar resource access is extremely important, for it is what will allow that, in both cases, the performance and thus time of processing are similar.

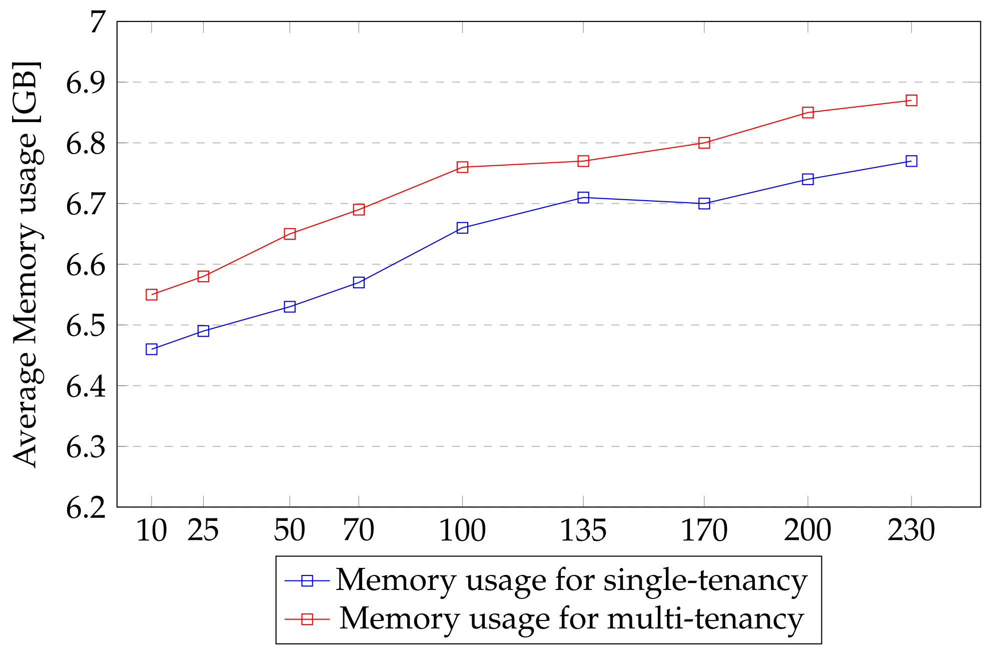


Figure 8.7: Comparison of Memory evolution

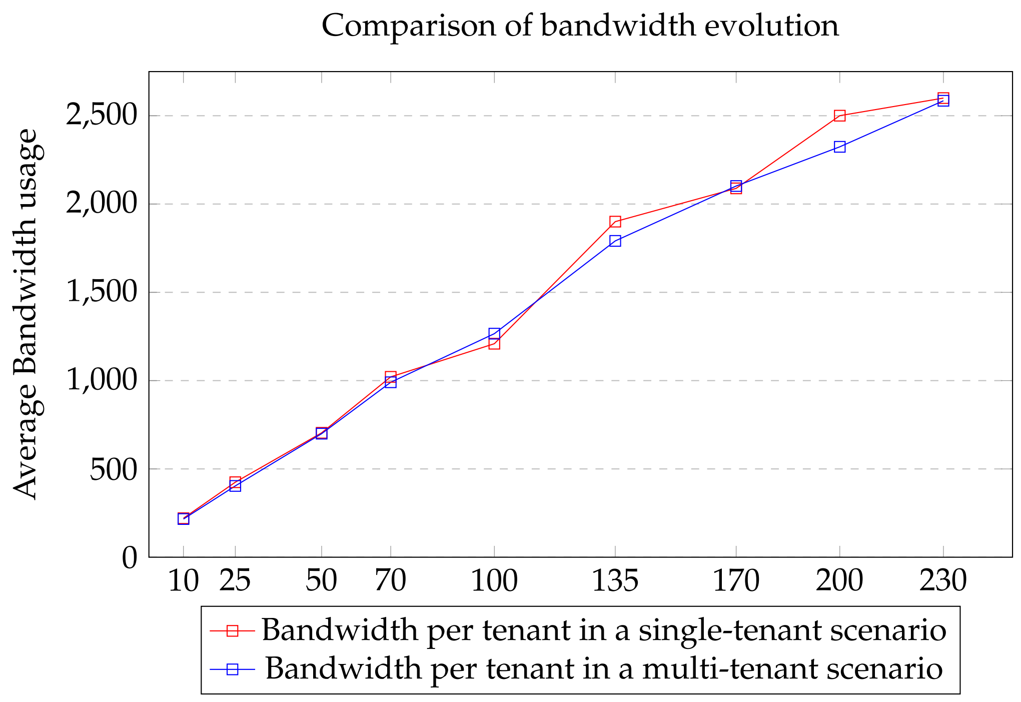


Figure 8.8: Comparison of bandwidth evolution

8.1.3 System performance

Answering the second question, **How does the system reacts to crescent number of events? In what point does it start degrading?**, it is expected that the more events are being processed, the longer it takes to process them. Besides this, also the number of tenants is expected to affect negatively the processing time for each event.

By analysing how long it takes for each event to be processed, it is shown in 8.9 that the processing time increases the more events are being processed.

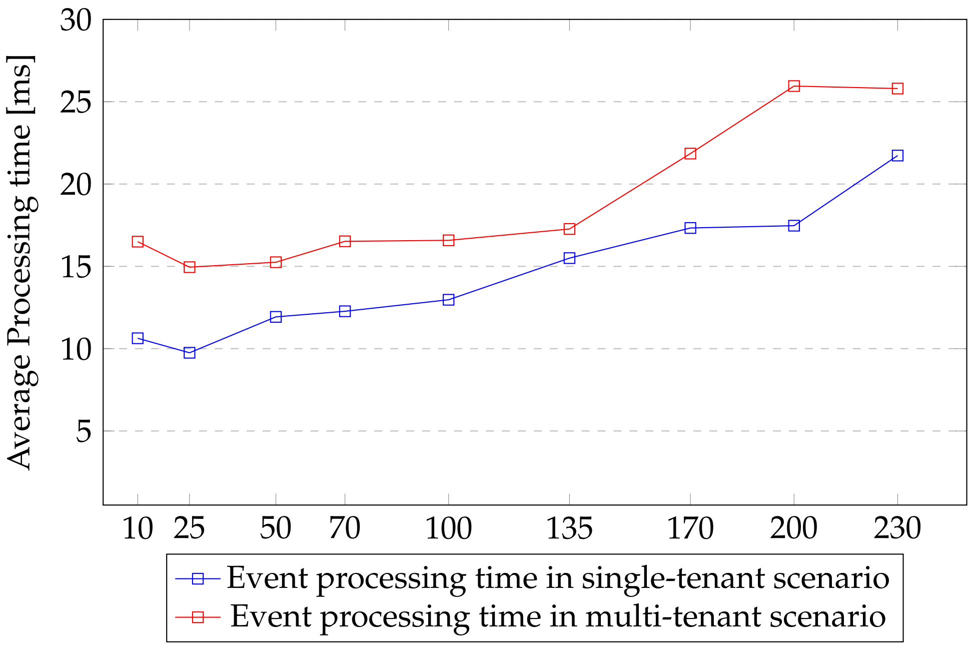


Figure 8.9: Processing time evolution

From these data it is calculated that, in average, events take more 3.42 ms to process than in the single-tenant scenario. This is caused, firstly, due to a **higher demand** for processing resources, that lead to events being received faster than the system is capable of processing, and therefore they wait momentarily. An analysis of the Kafka and database results, led to the conclusion that **Kafka operations are little impacted** by the multi-tenant solution, being that its performance depends more on the amount of events being processed in a given moment, but each **database queries take up to 1ms more to execute** and in average half a millisecond.

Analysing the throughput and comparing it with the number of events sent per second, we get that in both cases, with one or two concurrent tenants, the system is able to respond to the requests in due time. For example, having a 98% throughput means that, for 100 events sent in a second, the system can process 98% of them in a second.

	One tenant	Two tenants
Low load	96%	95%
Medium load	99%	94%
High load	97%	97%

Table 8.3: Capacity of event processing

From the table 8.3, it is not possible to infer a point in which the performance starts degrading. Since it is not possible to test with more tenants and a more limited environment in the current test conditions, this is a question to keep for future analysis. Nonetheless, it would be expected that, when the resources usage approximates the machine's limit, which depends on the load and number of tenants, the performance would be heavily degraded until a point in which packets would be dropped.

In conclusion, despite the performance being degraded in the current system, with the processing time increasing, this increase does not affect heavily its overall performance, since it is still capable of responding to the majority of received events (in average 95%) in the same interval they were received.

8.1.4 System scalability

How does the system scales for supporting more tenants? What influences this behavior? Since the system is being tested in a Kubernetes environment, it is challenging to answer the question. For a system with the minimum resources, it is expected that the system breaks with 2 tenants in high load. In the current test scenario this is impossible to test.

Analysing a comparison between a scenario with one and two tenants, it is observable that the same amount of CPU is being used per namespace. Supposing that this is applicable to three and more tenants, in the current scenario, graph 8.10 is created. Even though it does not represent the degradation caused by an approximation to the resources limits, it is clear that 2 CPUs are not capable of providing **two tenants** even in a normal load situation. 4 CPUs would be required. For **three tenants**, processing great amounts of events, 4 CPUs is once more not enough. However, in a scenario in which the three tenants process in average 100 events/sec, it is possible to host them three in a single instance. In short, tenants with smaller load sharing an environment require less resources to work properly and can therefore fit inside a same instance.

This analysis is possible comparing CPU since it is expected that this resource grow linearly, as seen in the analysis in 8.1.2. However, both RAM memory and bandwidth do not follow such a strict pattern and are not as accurate. The graph 8.11 displays the expected memory usage for up to four tenants.

And so, summarizing, the following tables, 8.4, 8.5, and 8.6, show how many resources are expected to be required for a given number of tenants for an average of 135 events per second, bearing in mind that these should not exceed 80% of the

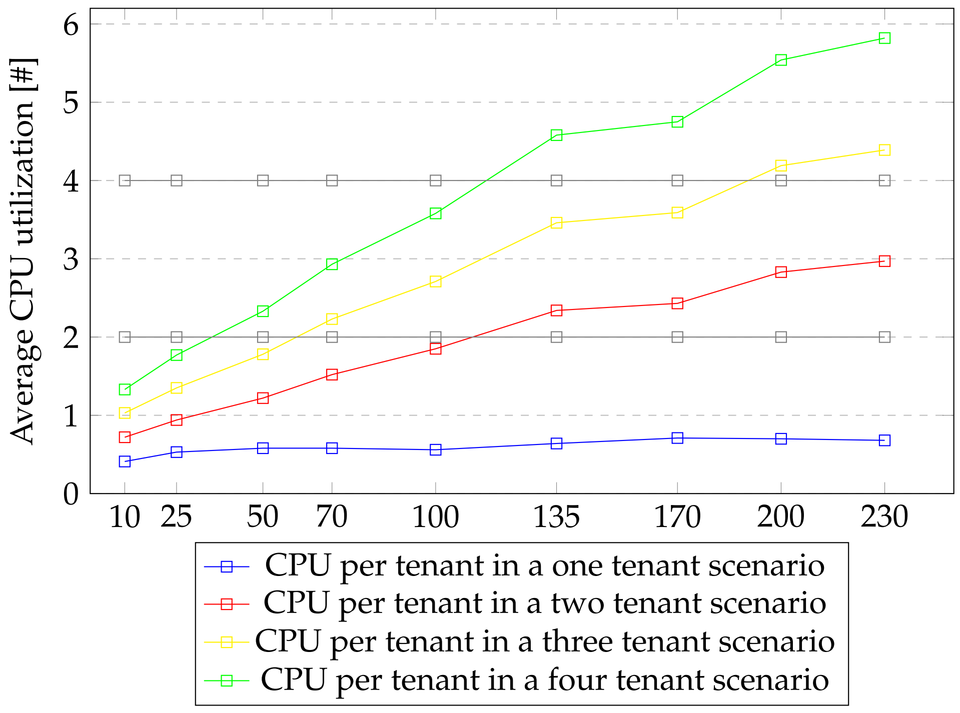


Figure 8.10: CPU expected scalability

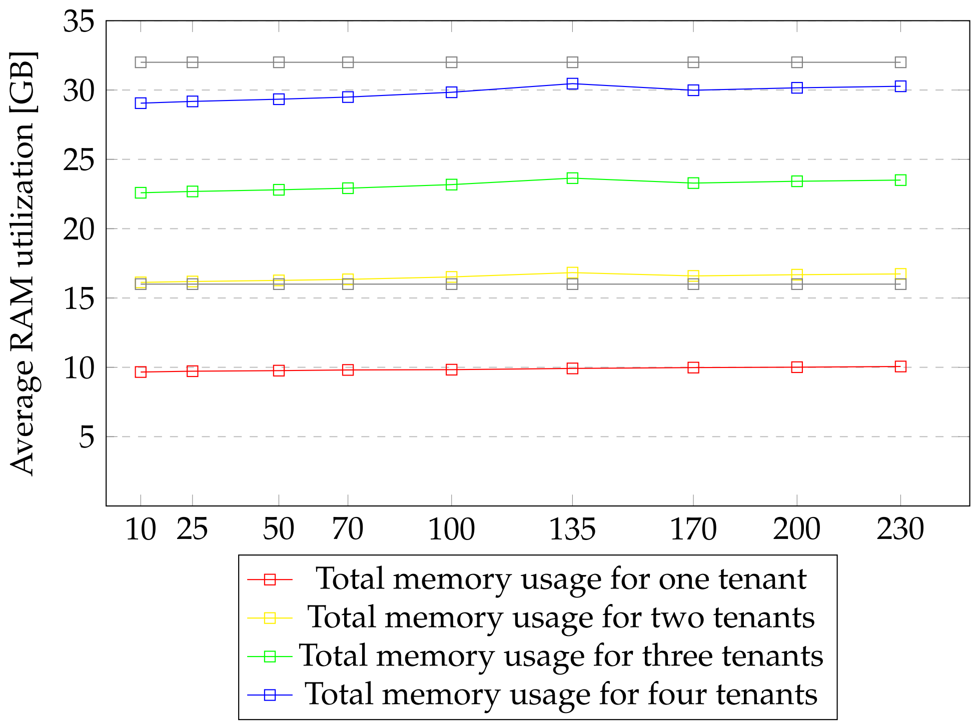


Figure 8.11: Memory expected scalability

available resources. As previously noted, these are not completely accurate and may not correspond to the reality.

Tenant	1	2	3	4
Utilization (%)	64	234	346	458
Min req	2	4	8	8
% Utilization	32%	59%	43%	57%

Table 8.4: Expected CPU utilization and minimum required

Tenant	1	2	3	4
Utilization (Gb)	6,72	16,83	23,64	30,46
Min req	16	32	32	64
% Utilization	42%	53%	74%	48%

Table 8.5: Expected memory utilization and minimum required

Tenant	1	2	3	4
Utilization (GB/s)	1,90	4,88	5,70	7,60
Min req	4	6	8	10
% Utilization	48%	80%	71%	76%

Table 8.6: Expected bandwidth utilization and minimum required

For most of the times, when the tenant count is increased, it is required to expand the minimum resources required. It is thus important to weight how relevant are these improvements considering the lost of performance.

8.2 Isolation tests

As proposed in the Non Functional Requirements, in 4.2, 100% of unauthorized accesses to unauthorized data are prevented. Therefore, the purpose of these isolation tests is to assure that a tenant has access to its own resources, guaranteeing that no other can access it as well.

8.2.1 Database

In order to test the database, both test users, tenant 1 and tenant 2, are logged into the database using their own credentials. This was done using **dbeaver**, a software that provides an interface to a database. So, to test the isolation in the database, the following tests were executed:

	Action	Expected output
db_1	Tenant fetches rows from its own schema.	The requested rows are returned.
db_2	Tenant fetches rows from an unauthorized schema.	A permission denied error message is returned.
db_3	Tenant creates an entry in a table from its own schema.	Row is created in the table. A same table in another schema does not contain the newly created row.
db_4	Tenant creates a row in an unauthorized schema.	An error message is shown. The row is not created.
db_5	Tenant updates a record in its own schema.	Row is updated.
db_6	Tenant updates a row in a table from another tenant's schema.	An error message is shown. Row is not updated.
db_7	Tenant deletes a row from its own schema.	Row is deleted. A record in another schema that fits the WHERE condition is not deleted.
db_8	Tenant deletes a row from an unauthorized schema.	An error message is shown. Row is not deleted.
db_9	Tenant fetches a view from its own schema.	The requested rows are returned.
db_10	Tenant fetches a view from another tenant's schema.	An error message is shown.
db_11	Tenant joins data from its own schema.	Fetched data is the result of the two joined tables.
db_12	Tenant joins data from its own schema and another tenant's schema.	An error message is shown.
db_13	Different tenants fetch data from their won schemas concurrently.	Both fetch their own data.
db_14	A tenant fetches data from its own schema while another tenant tries to access the exact same data.	Tenant with authorization fetches its own data. The other tenant receives a permission denied error.
db_15	Different tenants fetch each own data.	Both receive a permission denied error.

Table 8.7: Database Isolation tests

Test Results

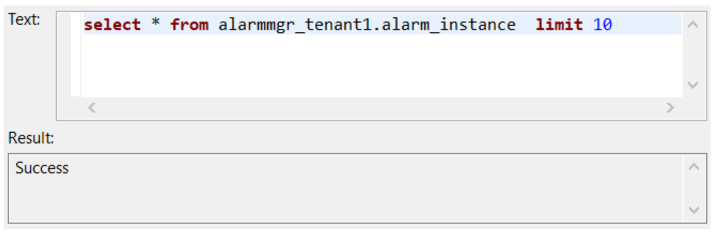
Summarizing, the database isolation test results are the following:

	Result		Result		Result
db_1	Passed	**db_2**	Passed	**db_3**	Passed

	Result		Result		Result
db_4	Passed	**db_8**	Passed	**db_12**	Passed
db_5	Passed	**db_9**	Passed	**db_13**	Passed
db_6	Passed	**db_10**	Passed	**db_14**	Passed
db_7	Passed	**db_11**	Passed	**db_15**	Passed

Table 8.8: Database Isolation results

In case of a successful operation, *dbeaver* displays the following log message 8.12:

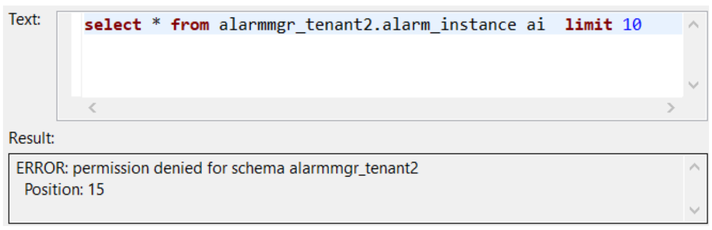


```
select * from alarmmgr_tenant1.alarm_instance  limit 10
```

```
Success
```

Figure 8.12: Database successful message

However, when permission is denied, it returns the message 8.13:



```
select * from alarmmgr_tenant2.alarm_instance ai  limit 10
```

```
ERROR: permission denied for schema alarmmgr_tenant2
  Position: 15
```

Figure 8.13: Database permission denied

In conclusion, 100% of unauthorized accesses to the database are prevented. Therefore, tenants can not access and interfere with one another's messages and hence privacy.

8.2.2 Kafka

In order to test the Kafka's solution isolation, at least two users, Alice and Bob, each part of a different tenant group, are logged into the cluster using their own credentials. This was done by creating a Producer and a Consumer. So, to test the isolation, the following tests were executed:

	Action	Expected output
kafk_1	Read/write permission is defined and tenant can now read and write to a topic.	Before, when trying to read/write, permission is denied. After permission, it can retrieve and send events.
kafk_2	Read/write permission is revoked and tenant cannot read and write to a topic.	Before, when trying to read/write, it can retrieve and send events. After, permission is denied.
kafk_3	Tenant reads from topics from its own namespace.	It consumes messages from the topics.
kafk_4	Tenant reads from topics from an unauthorized namespace.	A permission denied error message is returned.
kafk_5	Tenant writes to topics from its own namespace.	It produces messages to the topic.
kafk_6	Tenant writes to topics from an unauthorized namespace.	A permission denied error message is returned.
kafk_7	Tenant does not have permission to create topics (from its own or other namespaces).	A permission denied error message is returned.
kafk_8	Tenant does not have permission to delete topics (from its own or other namespaces).	A permission denied error message is returned.
kafk_9	Tenant joins data from its own topics.	Fetched data is the result of the two joined topics.
kafk_10	Tenant joins data with other tenant's namespace.	A permission denied error message is returned.
kafk_11	Different tenants fetch data from their won topics concurrently.	Both fetch their own data.
kafk_12	A tenant fetches data from its own topic while another tenant tries to access the exact same data.	Tenant with authorization fetches its own data. The other tenant receives a permission denied error.
kafk_13	Different tenants fetch each own data.	Both receive a permission denied error.

Table 8.9: Kafka Isolation tests

Test Results

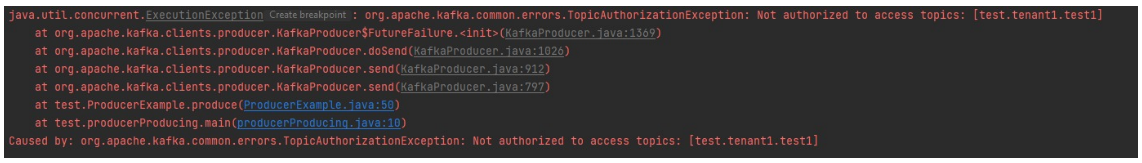
The following table 8.10 documents the results of these tests:

	Result		Result		Result
kafk_1	Passed	**kafk_5**	Passed	**kafk_9**	Passed
kafk_2	Passed	**kafk_6**	Passed	**kafk_10**	Passed
kafk_3	Passed	**kafk_7**	Passed	**kafk_11**	Passed
kafk_4	Passed	**kafk_8**	Passed	**kafk_12**	Passed

	Result		Result		Result
kafk_13	Passed				

Table 8.10: Kafka Isolation results

As seen, all tests are successful. In case of sending and receiving packets, it is verified if all is sent and received correctly. When a permission denied error is received, as expected, an exception is caught. 8.14 is an example of the exception.



```
java.util.concurrent.ExecutionException Create breakpoint : org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized to access topics: [test.tenant1.test1]
    at org.apache.kafka.clients.producer.KafkaProducer$FutureFailure.<init>(KafkaProducer.java:1369)
    at org.apache.kafka.clients.producer.KafkaProducer.doSend(KafkaProducer.java:1026)
    at org.apache.kafka.clients.producer.KafkaProducer.send(KafkaProducer.java:912)
    at org.apache.kafka.clients.producer.KafkaProducer.send(KafkaProducer.java:797)
    at test.ProducerExample.produce(ProducerExample.java:50)
    at test.producerProducing.main(producerProducing.java:10)
Caused by: org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized to access topics: [test.tenant1.test1]
```

Figure 8.14: Kafka permission denied example

These Kafka results prove that in all these cases data isolation between tenants is guaranteed in the Kafka topics. Therefore, tenants can not access and interfere with other tenant's messages and hence privacy.

8.3 Discussion

The **efficiency tests** shows that, comparing with the current Alarm Manager architecture, the new one offers slight improvements and therefore requires less resources to provide multiple tenants simultaneously. As proposed in the non functional requirements, it is expected that the system is able to process all events without surpassing the 80% mark of resources usage. It is possible to host more than one tenant in an instance without exceeding this mark and jeopardizing the instance integrity. Therefore, the first **non functional requirement**, regarding efficiency, is verified. However, as seen by the tests, this improvement is not substantial and in most cases, the increment of the number of tenants leads to a need to increase the available resources. For example, section in 8.1.4 it is shown that, regarding CPU, only when adding a fourth tenant it was not required to add more resources, and for the memory, when adding a third one.

The **isolation tests**, both at the database and Kafka level, verify confidentiality between different tenant's data. This means that a tenant's access to other unauthorized data is barred. Therefore, also the non functional requirement regarding tenant confidentiality is verified by these tests. And so, based on the analysis of the executed tests, it can be observed that **these fulfill the two proposed non functional requirements**.

The performance is deteriorated. From one to two simultaneous tenants, it is noticeable a 30% increase in the process of an event parsing and processing. Regardless, given these test conditions, it is not easy to predict how a tenant growth can lead to a bigger performance degradation. This would require deeper tests than those possible for this internship. Even so, it was possible to ascertain that the **improvements are below expectations** and a **multi-tenant solution as implemented** in this internship is not enough for the expected benefits. Nonetheless, these initiate the process towards a complete multi-tenant AM, being Processing the next step to follow.

Chapter 9

Conclusion

In conclusion, during the internship an investigation on the relevant **theoretical concepts** such as Cloud Computing and Single and Multi-tenancy was carried out, as well as the technologies associated with the current system that can be used to develop a solution. Multi-tenancy, as seen, is a paradigm of resource sharing between multiple tenants or clients. It is a widely used approach because it allows companies to reduce the need for infrastructure while maximizing the usage of the available resources. However, it also has drawbacks, such as complexity, resource competition, and lack of isolation.

This work was focused on the study of multi-tenancy in the **Alarm Manager**. As observed, it is a system capable of surveying, detecting, treating, and monitoring alarms from a network, and it is composed of multiple independent layers that communicate with each other through Kafka Streams. An analysis of the cost of changing each layer towards a more multi-tenant system was done, in order to understand which ones were more relevant, given the short internship time. And so, the Kafka and processing layer, as well as the database, were chosen as the layers with the highest priority. The requirements were gathered and a solution for the database,the Kafka, and Processing layer was planned, considering their complexity, gains, and cost.

The proposed solution was implemented for two tenants in the same Kubernetes cluster, by sharing a database and a Kafka cluster, a docker container for the Kafka solution, and two dashboards for monitoring. Finally, tests allowed to understand how the system responds to different tenant usage. Although these changes did not lead to significant improvements, these are architectural changes necessary for a multi-tenant AM.

It is considered that the goals of the internship were met and completed: Solutions for both Kafka and the database were studied, selected, and tested, allowing to conclude that minor improvements were achieved and isolation is kept. Regarding the Processing, even though it was decided not to proceed with its changes, it is an essential step towards a multi-tenant AM, and the multiple tasks required for its implementation were detailed and documented. Besides this, two dashboards were developed for monitoring and testing purposes.

The information gathered throughout the internship and contained in this document was also summarized and documented in the product's wikis, internal to Altice Labs, so as to facilitate its access for the team.

9.1 Future Work

Regardless of the conclusion of the internship, the team can still decide to move towards a multi-tenant architecture in the future, either due to necessity or updates in the used technologies that can facilitate the transition. This work can still serve as a foundation for a new and more adequate solution for the various Processing components: the cache, REST API, Quartz, among others. The developed dashboards will allow to test the implementation gains and drawbacks.

References

[1] Dockerfile reference. [Online; accessed October 12, 2022].

[2] Faq. [Online; accessed January 20, 2023].

[3] A guide to multitenancy in hibernate 6. [Online; accessed April 13, 2023].

[4] Overview. [Online; accessed October 12, 2022].

[5] The plaintext authenticator. [Online; accessed June 11, 2023].

[6] Salted challenge response authentication mechanism. [Online; accessed June 11, 2023].

[7] What is grafana? [Online; accessed April 23, 2023].

[8] Apache Kafka. Documentation. [Online; accessed October 29, 2022].

[9] Apache Kafka. Documentation. [Online; accessed January 20, 2023].

[10] Apache Kafka. Multi-tenancy. [Online; accessed October 29, 2022].

[11] Asana. Everything you need to know about creating a raid log, 2021. [Online; accessed December 19, 2022].

[12] AWS. Aws lambda. [Online; accessed January 11, 2023].

[13] AWS. Postgresql bridge model. [Online; accessed October 12, 2022].

[14] AWS. Postgresql pool model. [Online; accessed October 12, 2022].

[15] AWS. Postgresql silo model. [Online; accessed October 12, 2022].

[16] AWS. Tenant. [Online; accessed November 16, 2022].

[17] Azure. Cloud computing terms. [Online; accessed December 26, 2022].

[18] Azure. Multi-tenant saas database tenancy patterns. [Online; accessed October 12, 2022].

[19] Azure. Noisy neighbor antipattern. [Online; accessed December 28, 2022].

[20] Greg Boss, Padma Malladi, Dennis Quan, Linda Legregni, and Harold Hall. Cloud computing, 2007.

[21] Business Research Company. Alarm systems and equipment global market report 2022. [Online; accessed November 15, 2022].

[22] Christopher M. Judd. Getting started with docker. [Online; accessed October 12, 2022].

[23] CloudFlare. What is the cloud? [Online; accessed September 16, 2022].

[24] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2000.

[25] Confluent. Kafka vs. pulsar vs. rabbitmq: Performance, architecture, and features compared. [Online; accessed October 29, 2022].

[26] Docker. Docker overview. [Online; accessed October 12, 2022].

[27] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. pages 610–614. Institute of Electrical and Electronics Engineers Inc., 9 2014.

[28] Esat Erkec. Sql connection strings tips. [Online; accessed October 27, 2022].

[29] ET Money. What is compound annual growth rate (cagr)? [Online; accessed November 15, 2022].

[30] Google Cloud. What are containers? [Online; accessed January 11, 2023].

[31] Harish Somasundar. Database multi tenancy. [Online; accessed November 27, 2022].

[32] Dong Huang, Bingsheng He, and Chunyan Miao. A survey of resource management in multi-tier web applications. *IEEE Communications Surveys and Tutorials*, 16:1574–1590, 2014.

[33] IBM. Apache zookeeper. [Online; accessed October 29, 2022].

[34] IBM. User authentication and authorization in apache kafka. [Online; accessed October 29, 2022].

[35] IBM. What is virtualization? [Online; accessed December 26, 2022].

[36] IBM. Network file system, 2022. [Online; accessed December 11, 2022].

[37] Project Management Institute. *A Guide to the Project Management Body of Knowledge*. Project Management Institute, Inc., 2008.

[38] Ishwarya M. Installing apache kafka without zookeeper: Easy steps 101, 2022. [Online; accessed December 21, 2022].

[39] J. Rao J. Kreps, N. Narkhede. Kafka: a distributed messaging system for log processing.

[40] Dean Jacobs and Stefan Aulbach. Ruminations on multi-tenant databases.

[41] Jason Garman. Kerberos: The definitive guide. [Online; accessed June 11, 2023].

[42] JDBC PostgreSQL. Initializing the driver. [Online; accessed January 10, 2023].

[43] John Downs. Tenancy models to consider for a multitenant solution, 2022. [Online; accessed September 10, 2022].

[44] Michael Kavis, editor. *Architecting The Cloud*. John Wiley Sons, Inc., 1 2014.

[45] Kevin Cameron. How do alarm systems work? [Online; accessed November 15, 2022].

[46] Kubernetes. Kubernetes. [Online; accessed March 20, 2023].

[47] Kubernetes. Kubernetes components. [Online; accessed September 30, 2022].

[48] Kubernetes. Overview. [Online; accessed September 30, 2022].

[49] Lucid Content Team. The 4 phases of the project management life cycle. [Online; accessed November 22, 2022].

[50] Michael Cobb, StephanieMann. Oauth.

[51] Michael Seifert. Multi-tenancy data models in kafka. [Online; accessed October 30, 2022].

[52] Mountain Goat Software. User stories. [Online; accessed October 23, 2022].

[53] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service.

[54] Todd Palino Neha Narkhede, Gwen Shapira. *Kafka: The Definitive Guide*. O'Reilly Media, Inc., 9 2017.

[55] Oleksii Glib. How to build scale a multi-tenant saas application: Best practices, 2022. [Online; accessed September 10, 2022].

[56] Orbit Analytics. Multi-tenant. [Online; accessed October 27, 2022].

[57] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2:24–31, 5 2015.

[58] PostgreSQL. Row security policies. [Online; accessed October 28, 2022].

[59] PostgreSQL Documentation. pg$_dump.[Online;accessedJanuary10,2023]$.

[60] Prometheus. Alertmanager. [Online; accessed June 16 , 2023].

[61] Prometheus. Client libraries. [Online; accessed June 16 , 2023].

[62] Prometheus. Prometheus. [Online; accessed June 16 , 2023].

[63] Prometheus. Prometheus pushgateway. [Online; accessed June 16 , 2023].

[64] RedHat. What is saas? [Online; accessed November 16, 2022].

[65] Samuel Scott. What is docker? a revolutionary change in cloud computing. [Online; accessed October 12, 2022].

[66] Tal Perry. Database multi-tenancy for saas. [Online; accessed October 12, 2022].

[67] Wei Tek Tsai, Xiao Ying Bai, and Yu Huang. Software-as-a-service (saas): Perspectives and challenges. *Science China Information Sciences*, 57:1–15, 5 2014.

[68] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 5 2010.

Appendices