UCLouvain

LOUVAIN
School of Management

Louvain School of Management

# Microservices approach to build scalable and distributed systems

Auteur : Jean Bosco Rwibutso
Promoteur : Manuel Kolp
Année académique 2021-2022
Travail de fin d'études (TFE) en vue d'obtenir le titre de
Master (60) en Sciences de Gestion
Horaire décalé

# Abstract

Modern computer systems are built using microservices architecture, rather than the traditional way in which all the services were encapsulated in a single based system known as monolith. However, existing monolith applications are also constantly migrating to microservices mainly for their advantages to provide high scalable, distributed and performant system.

This thesis aims to provide detailed understanding of microservices approach. The focus will be the analysis of its distributed components and how the scalability is achieved. The components will be studied using $i*$ representation framework in order to have a clear view of their roles and dependencies. Moreover, a case study will be presented as an implementation of a messaging system using microservices architecture.

# Acknowledgments

I would like to thank my supervisor, Pr. Manuel Kolp, for his guidance and support in choosing this thesis subject. I also express my deepest gratitude to my family, colleagues and friends for their unfailing support and continuous encouragement.

# Contents


**1 Introduction** **1**

**2 Background and related work** **4**

2.1 Background . . . . . 4

2.1.1 Monolithic architecture . . . . . 5

2.1.2 Microservice architecture . . . . . 7

2.2 Related work . . . . . 11

2.2.1 iStar and Business Process Model Notation . . . . . 11

2.2.2 Virtual machines, containers and Kubernetes . . . . . 11

2.2.3 Scaling . . . . . 13

2.2.4 Inter-service communications . . . . . 14

2.2.5 High availability . . . . . 14

2.2.6 Monitoring and distributed tracing . . . . . 15

**3 Implementation** **17**

3.1 Functional analysis . . . . . 17

3.2 Technical analysis . . . . . 18

3.3 Architecture of the system . . . . . 19

**4 Validation** **21**

4.1 Method of validation . . . . . 21

4.2 Results . . . . . 22

4.2.1 The overview of the minikube cluster . . . . . 22

4.2.2 The scalability . . . . . 24

4.2.3 Caching . . . . . 25

4.2.4 The distributed tracing . . . . . 26

**5 Conclusion** **28**

# List of Figures

2.1 Store management system in a monolithic architecture [5, 9]. . . . . . . . . 5
2.2 Store management system in a microservice architecture [9]. . . . . . . . . 8
2.3 Comparison of virtual machines and containers technologies structures [16]. 13

3.1 BPMN diagram of a registration. . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 BPMN diagram of a chat and a notification. . . . . . . . . . . . . . . . . . . 18
3.3 SR diagram of the system components. . . . . . . . . . . . . . . . . . . . . . 20

4.1 Deployment resources on minikube dashboard. . . . . . . . . . . . . . . . . . 23
4.2 Service resources on minikube dashboard. . . . . . . . . . . . . . . . . . . . . 23
4.3 Pod resources on minikube dashboard. . . . . . . . . . . . . . . . . . . . . . . 23
4.4 Horizontal and vertical scaling in the deployment resource. . . . . . . . . . 24
4.5 Requests distributed between two running instances. . . . . . . . . . . . . . 25
4.6 Response time without using a cache vs response time using a cache . . . . 26
4.7 Distributed tracing of sending a chat message scenario. . . . . . . . . . . . . 27
4.8 Sequence diagram of sending a chat message scenario. . . . . . . . . . . . . 27

# Chapter 1

# Introduction

In this chapter, the context, the problem, the motivation and the objectives of this thesis will briefly be introduced.

**Context** This thesis is made in the context of the final dissertation in master in management science at Louvain School of Management[1]. Capabilities developed during the master program, especially information systems management courses[2], are used in this work.

**Problem** Build scalable and distributed systems using microservices approach.

According to Wikipedia, scalability is "*the property of a system to handle a growing amount of work by adding resources to the system*" [1].

A distributed system is "*a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system*" [2].

**Motivation** Information systems' world is fast-moving, new technologies are continuously being introduced in the ecosystem to ensure the delivery of high quality and large scale applications. As explained later in chapter 2, the core concepts of a

[1] *Louvain School of Management* is an international business school of the University of Louvain (UCLouvain).

[2] https://uclouvain.be/cours-2021-llsmf2018

good system are scalability, availability, efficiency, maintainability, serviceability and manageability [3].

The microservice approach aims to develop the whole system as a set of small distributed services. Each service is running on a separate process, and it is deployed and maintained in an independent way.

In contrast with the microservices, there is a traditional approach, known as "monolith", in which everything is developed and maintained in one service. This approach is not offering simple facilities in decoupling services, which makes maintenance more complex and not favouring the continuous development and improvement.

Systems have become extremely large, so that maintaining and deploying them as a single service is impractical[4][5]. A good example is to imagine the whole system of Facebook[6] or Google[7] which needs to be deployed for a small change in a particular service. Microservices architecture comes to rescue by decoupling the whole system into several independent services. In such case, only a new version of the impacted service will be deployed.

The distributed system involves a shift in complexity from inside the services to their connection and management [5]. Therefore, additional components such as service orchestration system, service discovery, load balancers and other communication gateways are involved in the system architecture.

**Objectives** This thesis aims to analyse and develop a system using microservices architecture. Thanks to the $i*$ representation framework, distributed components of the system will be presented.

We will have a strong interest for different configurations of the system components that help to deliver high scalable applications.

**Contributions** A small messaging system[3] is developed in order to illustrate the microservice architecture. Furthermore, we will put in place a Kubernetes[4] cluster which is nothing but an orchestrator and management tool of microservices.

**Roadmap** The remainder of this document is structured as follows. Firstly, chapter 2 provides the necessary background material and reports on related work. Requisite concepts to understand microservice architecture will be analysed and explained.

Secondly, in chapter 3, we will explore a case study as an implementation of microservices approach. However, a messaging system is developed using microservices architecture.

In chapter 4, we will evaluate and validate our implementation. In particular, scalability of the system is proven. Furthermore, we will expose how distributed applications in the system are managed and connected.

Lastly, we will draw our conclusions about our work in chapter 5.

[3] A messaging system with limited functionalities built in the context of this thesis as a proof of concept.

[4] Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [8]

# Chapter 2

# Background and related work

## 2.1 Background

A rapid growth of the internet and the digital technologies has changed the capabilities of information systems. As technology, methodologies, system management and environment keep changing, software companies face many challenges in the design of their systems to ensure that they are flexible, scalable and can easily be integrated into different environments.

Consequently, the system architecture plays a key role to support this evolution of information systems. For example, the feasibility and the complexity of integrating a new functionality or migrating from one service provider to another will rely on decisions made during the architectural design stage of the system.

The ***architecture of a system*** is what enables it to evolve, adapt and be flexible to change over its lifetime and therefore provides a standard and efficient service[5].

We distinguish two main types of system architecture; ***monolithic architecture***, in which all services are encapsulated in one component, and ***microservices approach***, in which the whole system is divided into a set of distributed services or components. The differences between both approaches are discussed in the following section.

### 2.1.1 Monolithic architecture

The monolithic architecture [5, 9, 10] is the traditional approach in which systems were built. In this approach, the system is built in one single unit and all services and business logics of the system are encapsulated in it and deployed as one component. In general, the monolith system is autonomous and independent of the other systems.

For example, let's consider a store management system which has four services as follows [5, 9] :

- **StoreFrontUI** : The service dedicated to manage the end-user interface.
- **Accounting** : The service dedicated to handle accounting business logics.
- **Inventory** : The service dedicated to handle inventory business logics.
- **Shipping** : The service dedicated to handle shipping business logics.

Figure 2.1 shows The representation of such system deployed as a monolithic application. All the four services are packaged in the same component and deployed as a single unit.

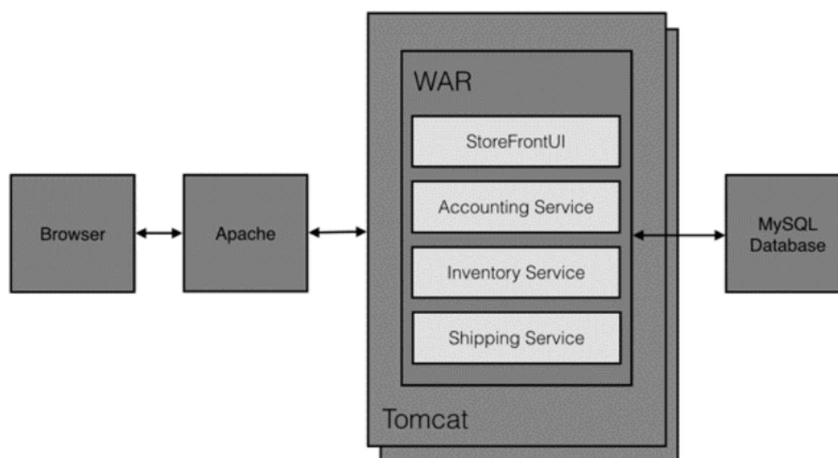



Figure 2.1: Store management system in a monolithic architecture [5, 9].

The monolithic approach is usually better for simple and lightweight applications. It is convenient for systems whose services are strongly dependent on each other and difficult to be distributed into separate components.

This approach presents some advantages due to the fact that the system is packaged and deployed as a single component. According to Richardson [9, 10], three main advantages of the monolithic systems are :

- **Simple to develop** : All source code is located in one place which ease its development as programming, debugging and testing processes involve only one service without any dependencies. Onboarding new developers is also simple, as they will find all the code in one place, and it facilitates their understanding of the system logics.
  In terms of evolution of the system, when new features are to be developed, the integration will be easy as some data are already present in the code and can easily be reused.

- **Simple to deploy** : Only one deployment unit should be deployed at runtime. Thus, whenever a new version of the system needs to be released, developers needs to perform a single chunk of deployable code instead of making updates in separate entities.

- **Simple to scale** : Scalability in a monolithic system can be achieved by replicating instances of the system on various servers behind a load balancer for traffic distribution[5, 11].

On the other hand, the monolithic approach presents drawbacks, especially when the system becomes larger or when the development team grows in size. Among the non-exhaustive list of the drawbacks, some being :

- **Code base difficult to understand** : The system functionalities keep changing and expanding as user demands and requirements change. Consequently, the system code base increases in size and in complexity. A such code base is difficult to understand for new developers. It becomes also difficult to evolve and to maintain due to its complexity[5, 12].

- **Code ownership cannot be used** : Once the system gets a certain size, it is useful to divide up the developers into teams that focus on specific functional areas. The problem with a monolithic system is that it will prevent teams to work independently, as they will have to share the same code base. They will need to coordinate whenever there is a change or a deployment of a new version. [9, 11].

- **Testing becomes harder** : As all the logics are encapsulated in one component, services cannot be tested independently. As a result, even after a small change, the regression testing for the full monolithic system is required.
- **Continuous development and continuous deployment are difficult** : Another disadvantage of a monolithic system relates to the versioning of its code base. It is very challenging to version independently the services of the system, as they belong to the same code base. A large code base impacts the speed of the development because the entire system needs to be built at each update. Consequently, this becomes an obstacle to frequent updates and deployments of the system [9].
- **Obstacle to scaling** : A monolithic architecture can only be scaled horizontally by running multiple instances of the system, respectively to the increasing of traffic load. This architecture can't scale with the increasing of the resource needs. For example, different system components or services could have different resource requirements. One service could be CPU intensive, while another might be memory intensive. With the monolithic architecture, it is difficult to scale each component independently[9].
- **Long-term commitment to a technology stack** : A monolithic architecture forces developers to commit to the technology stack that is chosen at the start of development. If the system uses a technology that subsequently becomes obsolete, then it can be very challenging to incrementally migrate to a newer or better technology. However, some components could only be compatible with specific technologies that the monolithic architecture can't integrate[9, 11].

### 2.1.2 Microservice architecture

The microservice architecture is a service-oriented approach in which a system is decomposed into a loosely-coupled collection of small services, each running in its own process and communicating with lightweight mechanisms[13]. Thus, microservices can be defined as *"individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines"*[5].

For example, on the figure 2.2, the store management system presented at the section 2.1.1 with the monolithic architecture, is re-illustrated using the microservices approach. The four services are now packaged and managed in different components. They are deployed separately and are communicating through REST API[1]. Each component handles a single responsibility.

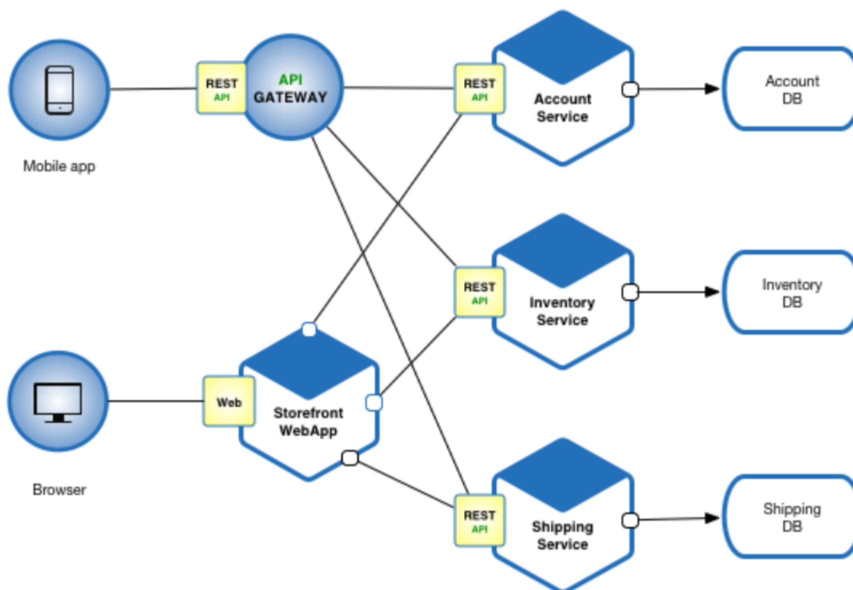



Figure 2.2: Store management system in a microservice architecture [9].

The microservice approach is better for large and complex systems managed by a certain size of developers. Microservices could also be the better approach for small systems that have loosely coupled services suspicious to evolve over time.

Some main characteristics and advantages of a microservice architecture are :

- **Single responsibility per service** : This is a principle implies that a service should handle only a specific logic and should only contain the code base related to it. It avoids having a million lines of code in one component and make simpler the code base. As a result, it makes the testing easier and facilitates the growth of the code base over time to accommodate more functionality [4, 5].

[1] a protocol that allows applications to communicate to each other on a network. it is explained further in the document.

- **Ownership and autonomy** : A large system often has a large team of developers working on it. Microservices help to organize developers into autonomous teams that own and work only specific microservices. Each team is independent and makes its own decision, therefore increasing motivation and efficiency [4, 5, 9].

- **Encapsulation** : Microservices own their business logics and data and keep them private. Microservices need to communicate through defined protocols to access each other data. This ensures loose coupling between services, therefore avoiding complexities [5, 9].

- **No commitment to a technology stack** : As a result of encapsulation, each microservice is independent to choose its own technology stack as long as it handles the assigned business logic. This is a benefit in case the system needs an incremental migration from an obsolete technology. It also gives a possibility to implement some components with suitable technologies that were not feasible with the current technology stack [4].

- **Services independently scalable** : With the microservice architecture, the system can be scaled horizontally and vertically based on each service needs. For example, multiple instances of a traffic intensive service could be run, while memory has been increased for a different service with memory intensive property [4].

- **Continuous development and deployment** : As microservices are independent, teams can independently develop and deploy new versions of system components. This facilitates frequent updates and deployments of the system [4].

- **System downtime reduction through fault isolation** : Fault isolation means that a critical system could stay up and running even when one of its modules or services fails. It becomes easy with microservices to isolate the failure to a single component and prevent cascading failures that would cause the whole system to crash or to fail [4].

The microservice architecture has although some drawbacks, especially due to the additional complexity of creating a distributed system. Some examples of the induced challenges are :

- **Data management** : Developers must implement the inter-service synchronizations to ensure data consistency in each service [9, 14].
- **End-to-end testing** : Even if testing an individual service would be easy, testing end-to-end the whole system could be challenging. This is due to the multiple interactions between the services. For example, when a problem is found during the testing process, it could be complicated to find the source of the problem [9, 14].
- **Coordination and dependencies between teams** : Microservices are encapsulated and owned by different teams. So when teams have dependencies, they have to align on how their microservices will communicate. For example, the nature of requests and responses between microservices is discussed. Therefore, the dependencies also involve a solution architect, who master the design and the business logic of the whole system, and will have to assign responsibilities to existing microservices or decide when a new microservice is to be created [14].
- **Common practices across teams** : Having autonomous teams could lead to a lack of standardization. If each team exposes functionality developed as best see fit, it might harm the availability or performance of the services consuming that functionality. So, microservices approach requires a minimum standardization and common practises shared across teams [14].
- **Operations complexity** : In the context of deployment operations, microservices architecture increases the number of deployments compared to a monolith system, which requires only a single deployment. Plus, an orchestration system, service discovery and load balancers are required to ensure correct deployment, communication and management of the microservices. This represents a total shift in complexity from inside the services to the connections between them and their management [9, 5].
- **Increased memory and resource consumption** : The amount of isolated deployable services increases in a microservice architecture. Resources for every service are provisioned separately, which increases the consumption of them. for example, if each service is running on its virtual machine, it will affect an overhead of memory, CPU and other resources provisioning. [9].

## 2.2 Related work

In the previous section, advantages and challenges of microservices have been mentioned. Various concepts and approaches are considered in order to build a scalable and distributed system. In this section, concepts related to microservices and distributed systems will be explained. Therefore, it will also put into context the implementation of a messaging system using the microservice approach that will be exposed later in chapter 3.

### 2.2.1 iStar and Business Process Model Notation

iStar and Business Process Model Notation are two concepts that are used in the analysis of business processes. The following paragraphs briefly define them. However, a general view and detailed about these two concepts are referenced in the lecture "Technological Project (IT)" given by Manuel Kolp and will not be developed in this thesis.

***Istar*** is a modelling language that describes conceptually interactions between business processes and supporting systems in terms of actors' goals and commitments, and the associated dependencies.

In this thesis, we will use **Strategic Rationale** (SR) model, which describes interactions between actors by presenting processes and resources inside their boundaries. SR model also illustrates how actors achieve their goal and highlight their dependences with other actors.

***Business Process Model Notation (BPMN)*** is used to model business processes of an organization or a system by describing conceptually the flows of activities. Thus, thanks to BPMN, functionalities of a system can be described by showing step by step actions and actors involved.

### 2.2.2 Virtual machines, containers and Kubernetes

Microservice applications are wrapped into different installation packages, which make difficult their deployment on one server. It is not a good choice to install many appli-

cations on top of the operating system. Applications add their own settings, libraries and other dependencies which could be incompatible with other applications or with the operating system itself.

***Virtual machines*** were used as a traditional model of installing and deploying several applications on the same server machine. Virtual machines are basically individual computers, sharing the physical resources of the host machine [15]. They are detached from each other and the operating system of the host. Each machine runs on its own operating system and installs the related applications on it. The setup process can be automated, but when the number of different virtual machines grows, the solution may not be feasible any more as the costs increase and the time required for completing the setup work multiplies [15].

***Containers*** are a lightweight, efficient and standard way for applications to move between environments and run independently. Everything needed (except for the shared operating system of the server) to run the application is packaged inside the container object: code, run time, system tools, libraries and dependencies [16]. Thus, containers appear quite similar to virtual machines, except that they share the operating system of the host server. The figure 2.3 illustrates the structure of virtual machines and containers technologies.

***Kubernetes*** is a portable, extensible, open source platform for managing containerized applications and services, that facilitates both declarative configuration and automation [8]. This management is usually called "container orchestration". It includes actions from basic procedures such as starting and stopping containers, all the way to creating the images, monitoring and more advanced configurations. Kubernetes provides suitable functionality, for example, to cover the aforementioned situation where service recovery is done by bringing up more instances of the strained service. The orchestration tool can automatically create more copies of a certain container and redirect the requests to different instances [16].

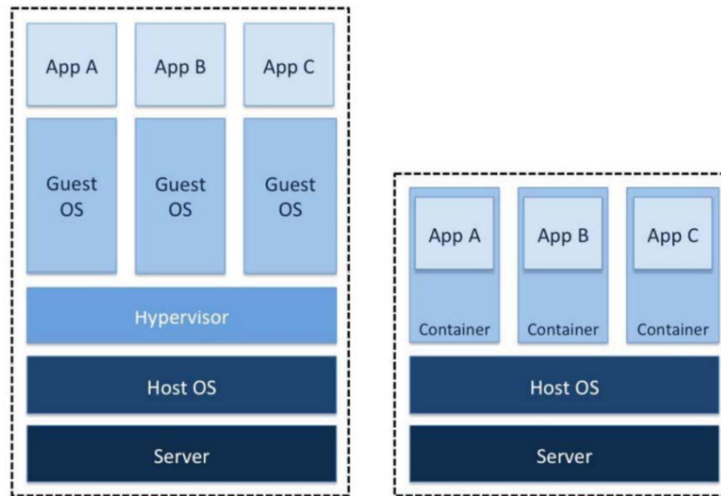



Figure 2.3: Comparison of virtual machines and containers technologies structures [16].

### 2.2.3 Scaling

Microservices applications that are running in a Kubernetes environment are scaled independently. There are two types of scaling: horizontal and vertical scaling. Horizontal scaling means that Kubernetes can configure the number of running containers of a given application or service. This is different from vertical scaling, which for Kubernetes would assign different resources parameters (for example: memory or CPU) to running containers [8].

Kubernetes achieves this by defining a dedicated resource called ”deployment”. This resource is used to tell Kubernetes how to create or modify instances of the running container that holds a containerized application. Thus, the number of replica and other resources, needed by the running container, are defined in the deployment resource.

Furthermore, Kubernetes has a feature ”autoscaling”, which automatically updates the deployment resource in the aim of scaling the applications to match demand [8]. The autoscaling algorithm defines conditions on which the resource will be updated. For example, update the number of replica when a certain traffic load is attained or update allocated CPU and memory of an existing application based on its utilization.

### 2.2.4 Inter-service communications

In the microservice architecture, services need to collaborate to handle requests from the system's clients. Multiple services have to interact to complete business activities. consequently, they must define some inter-service communication protocols.

These protocols can be classified into two basic messaging patterns that microservices use to communicate between each other:

- ***Synchronous messaging*** : It is the pattern in which the caller waits for a response from the receiver. For example, HTTP[2] protocols could be used when a microservice expects a response from another microservice in order to continue the processing of its initial request.
- ***Asynchronous messaging*** : It is the pattern in which the caller doesn't wait for the response from the receiver. For example, AMQP[3] protocols like RabbitMQ or Apache Kafka could be used when a microservice wants to notify or sending a request to another microservice without expecting a response from it to continue the processing of the initial request.

In Kubernetes, as explained before, services can be scaled up horizontally and run on multiple replica instances. Running instances have their own IP address, at which they can be accessible on the network. Kubernetes defines a resource called "service" which is an abstraction proxy of the logical set of all running instances of a microservice. A single DNS name is attributed to this resource. The service resource is responsible to load-balance the traffic across the running instances [8].

### 2.2.5 High availability

Microservices need to collaborate to handle request from the system's clients. When the number of requests are increased, It is important to accommodate whole requests with high availability.

[2]Hypertext Transfer Protocol
[3]Advanced Message Queuing Protocol

The basic technique to guarantee the availability of the system with scaling up horizontally some microservices. When the traffic will be load-balanced across the running replica, it will allow the system to process more requests simultaneously. Even better, some systems could be deployed on two different data centres behind a load-balancer to ensure the high availability. The benefit of this is no downtime when one data centre is out of service.

Furthermore, performance is a critical consideration for microservices high availability. To maintain a higher performance under heavy load, microservices could use caching techniques. Caching makes efficient, the reading of data that doesn't change frequently. Caching also provide to avoid re-calculation of heavy processes.

However, a distributed cache needs to be implemented in the microservice architecture. The distributed cache will help to help different microservices and replicas to benefit from the same cache. If one operation has been calculated by one instance, then other instances can consume calculated data from the distributed cache directly. Redis[4], Couchbase[5] and Cassandra[6] are examples of distributed caches.

### 2.2.6 Monitoring and distributed tracing

Monitoring systems built with microservices architecture is challenging as components are distributed. Monitoring helps to understand the overall health of the system, to glean insight into the performance and availability of each individual service that makes up the system, to isolate problematic transactions and endpoints and also to optimize the end-user experience [17].

There exists a lot of monitoring tools in addition to the dashboard provided by Kubernetes to monitor microservice systems. In general, the tools gather ***metrics, logs and traces*** from several services and centralize them into dashboards.

While collecting *metrics* from the system, the focus is on the resource consumption,

[4]https://redis.io
[5]https://www.couchbase.com/
[6]https://cassandra.apache.org/

latency, traffic, errors, and saturation of the services that will help in determining when there is a need for alerts in the system.

*Logs management* is very important in systems development as they help to identify issues or detect threats of an application. The tools collect and aggregate the logs from every service. Then, they will parse and store those logs in a searchable manner so that they can easily be visualized or manipulated through dashboards.

In microservice architecture, it is important to be able to follow the execution path of a system request as it could span multiple services. *Distributed tracing* helps to better reconstruct the requested journey through visualization of execution flow. It also offers insights on the duration of operations and how services relate to each other while performing the given task [9].

# Chapter 3

# Implementation

In chapter 2, some related work and concepts about microservice architecture were presented. However, in the context of this thesis, a messaging system was built using the microservice approach as a practical implementation of those concepts. Thus, in this chapter, the developed messaging system is discussed and analysed in details.

## 3.1 Functional analysis

The messaging system that was developed, has three functionalities :

- **Registration** : The system allows people to register to become users of the system.
- **Chat** : Users of the system can send messages between them.
- **Notification** : When a message is sent, the system notifies the corresponding recipient.

The functional requirement for registration is to only allow users who don't exist yet in the system to be registered. As for the functional requirement of a chat, the recipient user should always be registered in the system. Furthermore, for each message, the recipient user is notified through a dedicated channel.

The above requirements are analysed in detail thanks to BPMN diagrams illustrated on the following figures : 3.1, 3.2.

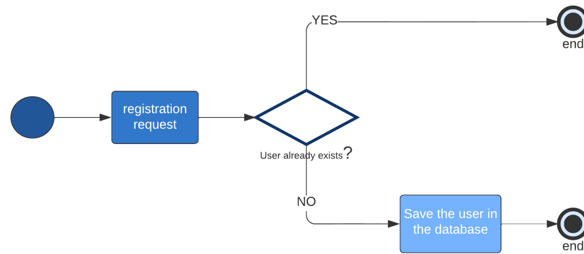



Figure 3.1: BPMN diagram of a registration.

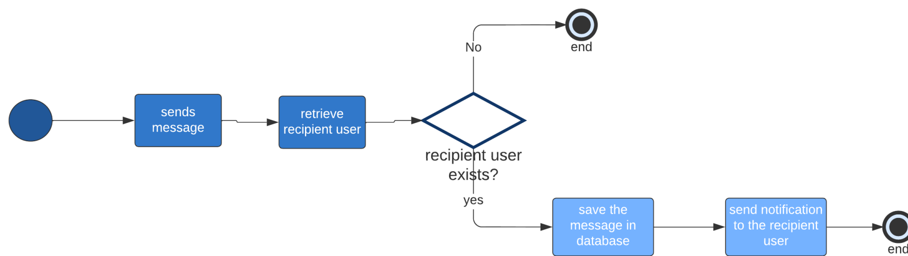



Figure 3.2: BPMN diagram of a chat and a notification.

## 3.2 Technical analysis

To build a system with functional requirements as explained at section 3.1, the following technical assumptions or requirements were made :

- Developers are organized into different teams that are responsible for specific components of the system.
- User related information is stored in a legacy datastore, and reading data from that datastore is a very slow and consuming operation.
- The system receives a huge amount of chat messages.
- The amount of notifications is directly proportional to the number of chat messages. However, the channel dedicated to send them, has a limited capacity of the notifications that can handled simultaneously.

## 3.3 Architecture of the system

Based on the functional and technical requirements analysed in sections 3.1 and 3.2, the system is built using the *microservices approach*. However, the application will be divided into three microservices: user, chat and notification microservices. **User microservice** handles user information related logics: creating a new user and retrieving existing user information. **Chat microservice** handles logics related to chat messages. **Notification microservice** handles logics related to notifications. The split of the application into distributive microservices will help to assign them to different teams of developers.

The *inter-service communication* will be achieved using two technologies, **HTTP calls** for synchronous communications and **Redis broker** [18] for asynchronous communications. The communications between chat and user microservices will be synchronous, as the chat microservice expects an immediate response while retrieving information of recipient users. On the other hand, the chat microservice communicates asynchronously with the notification microservice because it doesn't expect a response. Plus, the notification channel has a limited capacity, the Redis broker helps to store the notification requests in a queue so they can continuously be consumed as soon as the channel is free.

As reading user information is a heavy and slow operation, a **Redis cache**[19] is implemented to insure a *high availability* of user information. Once a user is successfully registered, his corresponding information is stored in the Redis cache. In fact, accessing data in the cache is much faster than retrieving them in the legacy datastore.

The system uses EFK (**ElasticSearch-Fluentd-Kibana**) technology stack for *monitoring logs* purposes. Fluentd [21] is responsible to parse the logs into a Json structured format. ElasticSearch [20] indexes those logs so that they can be easily manipulated and searchable. Kibana [22] provides a dashboard that helps to search and to visualize logs from elasticSearch. Therefore, all logs for the three microservices mentioned above will be collected in the same place thanks to these technologies. It helps performing a *distributive tracing* of system requests and operations.

The SR diagram, presented on the figure 3.3, illustrates in detail the conceptual design of the system architecture.

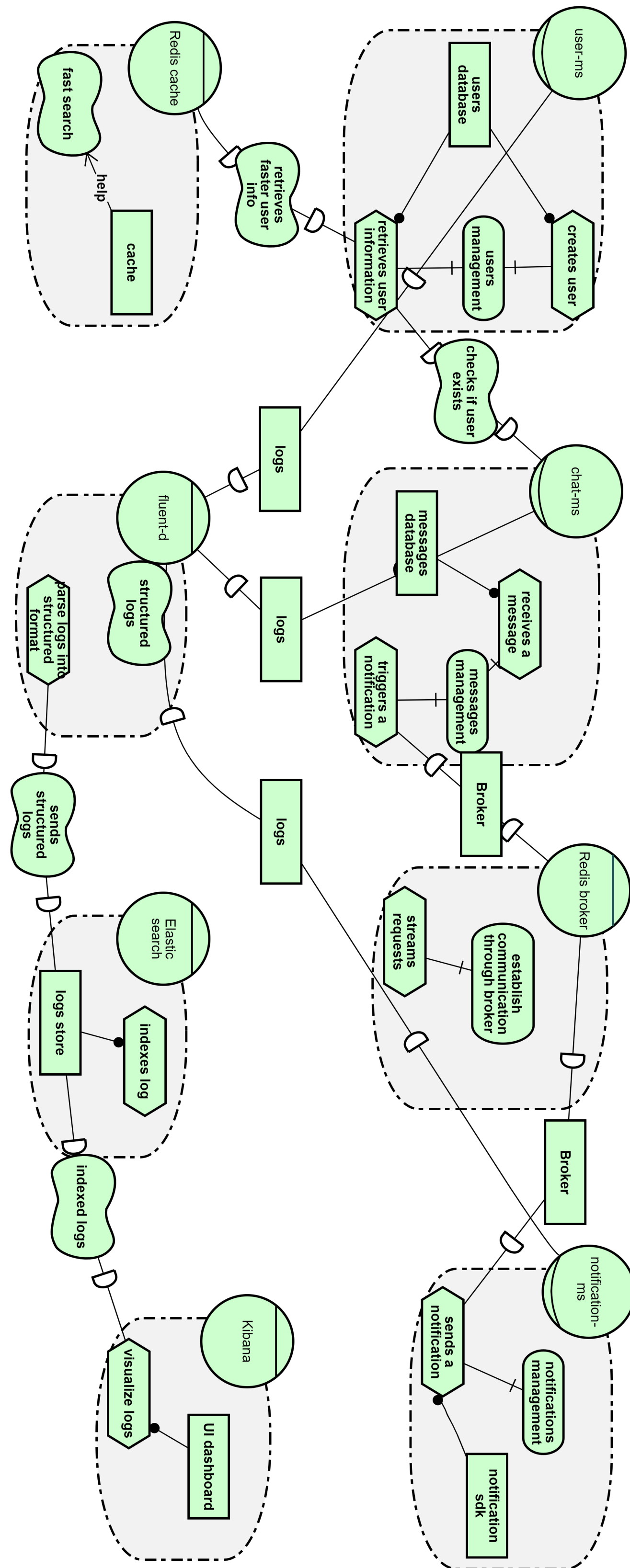



Figure 3.3: SR diagram of the system components.

# Chapter 4

# Validation

In this chapter, we will validate the described implementation, which conducted to prove that the distributed system built reaches the scalability and other microservice concepts. Firstly, the method of validation presents the set-up and different factors of the environment on which the system is built. Secondly, results will be discussed.

## 4.1 Method of validation

The messaging system described in the chapter 3 has been developed on a personal computer running on Windows 10 with an Intel Core i7 processor and 16 GB of RAM. **Docker** [23] and **minikube** [24] tools were installed on the same machine to manage application images and to orchestrate containerized applications.

A single node Kubernetes cluster was set up using minikube. The following containerized applications were deployed on the cluster: user-ms, chat-ms, notification-ms, redis, fluentd agent, elasticsearch and kibana. Only one instance is deployed for every application except chat-ms. In fact, chat-ms is scaled up to two instances due to high traffic expectations. The configuration of applications on the cluster is defined by kubernetes resource objects through .yaml [25] configuration files.

User-ms, chat-ms and notification-ms are the functional applications developed in the context of this thesis. Thus, they are written in JAVA using Spring boot framework [26]. Those applications are packaged into deployable images before being rolled out on the

cluster. As for the deployable images of redis, fluentd agent, elasticSearch and kibana applications, they are downloaded from the official docker images hub [27].

The system health is monitored by using the minikube dashboard. Thus, we can easily manage the status of the applications. Kibana dashboard will help to visualize the system logs. Furthermore, it will help us to trace the system requests and retrieve relevant information such as the execution time, inter-service communications etc.

Figures from kibana dashboard will highlight the *time* of a log entry, *traceid*, *message*, *kubernetes.labels.app* and *kubernetes.pod_name* properties. The *traceid* property represents a unique identifier of a request as it spans multiple microservices. The *message* property contains the actual log message launched from the application. It could contain a description, a response time and other information about a process that the application is explicitly logging. The *kubernetes.labels.app* describes the application name, while *kubernetes.pod_name* is the unique identifier of the corresponding running instance.

## 4.2 Results

In the following sections, the results are firstly presented by overviewing the distributed components on the minikube cluster. Secondly, scalability and caching outputs are assessed. Then, the distributed tracing of a chat messaging scenario is presented.

### 4.2.1 The overview of the minikube cluster

All the system applications were deployed as independent components running in a containerized environment. Separate configurations were applied for each application. In particular, the focus is on *deployments, services and pods* kubernetes resources. Deployments are resources that help us to deploy new versions of the application when the deployable image changes. Services help in the networking across the cluster by giving each application a specific host address. Lastly, the running instances of the applications are represented by pods, which have a unique identifier for each instance. These three resources, visible in the minikube dashboard, are illustrated on the following figures 4.1, 4.2 and 4.3.

Figure 4.1: Deployment resources on minikube dashboard.

Figure 4.2: Service resources on minikube dashboard.

Figure 4.3: Pod resources on minikube dashboard.

### 4.2.2 The scalability

The chat-ms has been scaled up to two running instances because it expects high traffic. As shown on the figure 4.4 of the .yaml configuration of the deployment resource, the horizontal scalability is achieved by defining the number of replica. Furthermore, the vertical scalability is also defined in the same file under resources specifications.

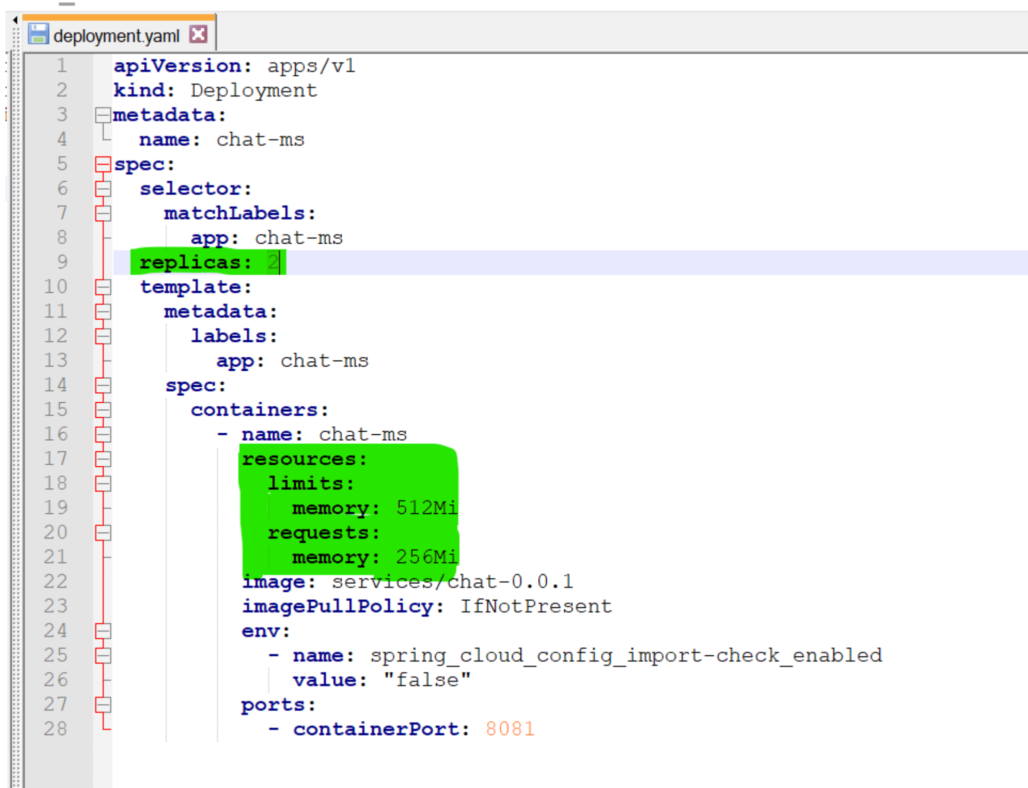



Figure 4.4: Horizontal and vertical scaling in the deployment resource.

Requests of chat-ms are randomly distributed between the two running instances. The figure 4.5 shows how the requests are being processed by chat-ms instances with different identifiers (kubernetes.pod_name). In consequence, the scaling helped to overcome the high traffic without overloading the whole system resources.

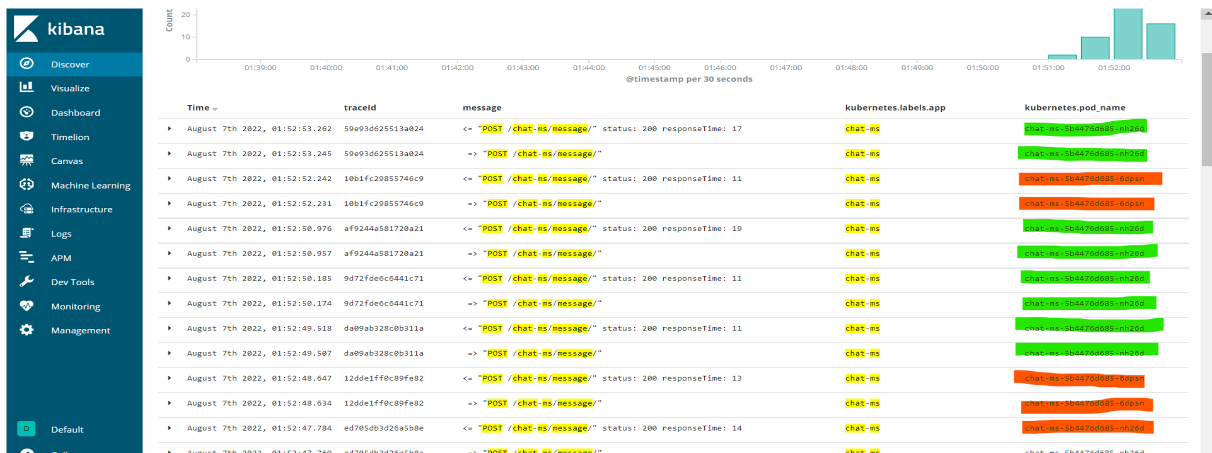
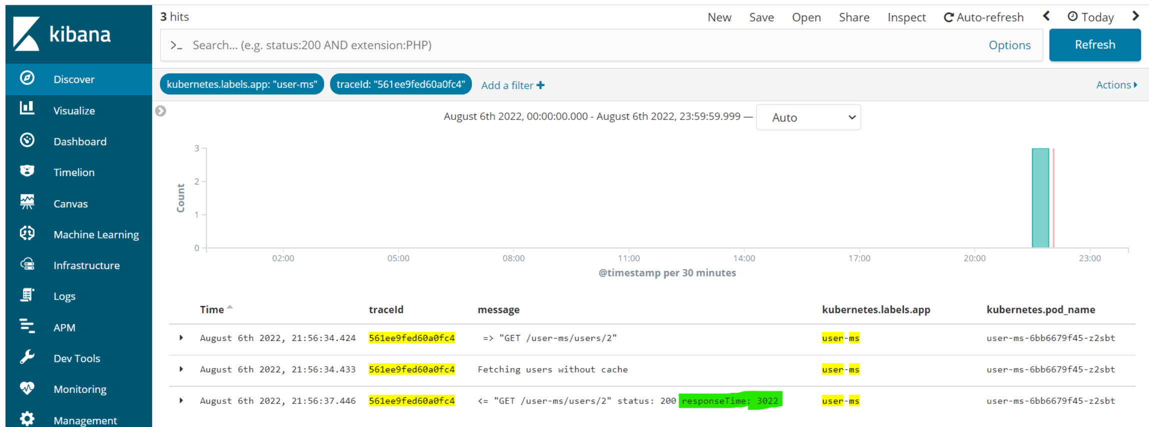



Figure 4.5: Requests distributed between two running instances.

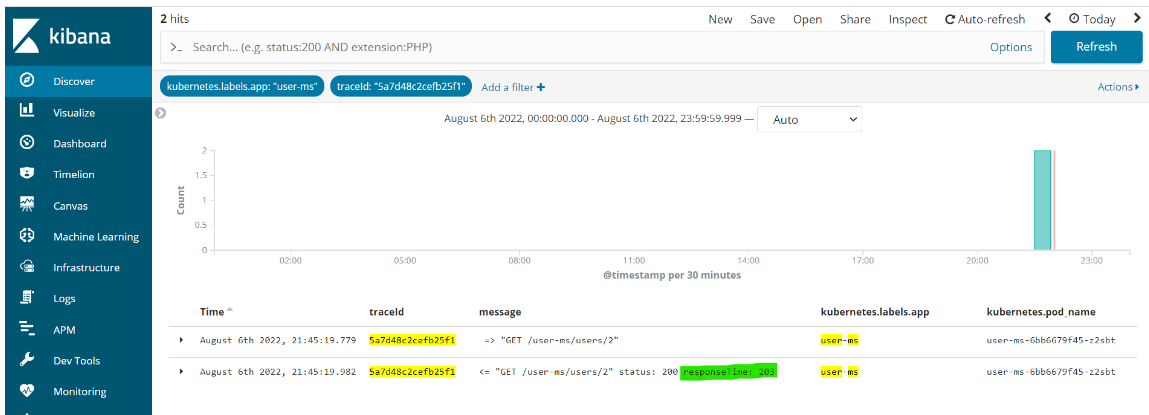
### 4.2.3 Caching

Retrieving user information through user-ms application is a very slow operation. A redis cache enables fast access to this information. It is a distributed cache, means that it can be shared between multiple instances or multiple applications. The cache is a separate component from the application, which means, the cached data will not be lost when the application restarts or crashes.

As presented on the figure 4.6, the request processed without using the cache takes much longer when compared to the request processed using the cache. Without a cache, the response time is 3022 milliseconds while it is only 203 milliseconds (16 times faster) when using a cache. Therefore, we can say that caching is a crucial factor in the availability and responsiveness of the whole system.




(a) Without using a cache.




(b) Using a cache.

Figure 4.6: Response time without using a cache vs response time using a cache

### 4.2.4 The distributed tracing

Finally, in this section, we will visualize the scenario of sending a chat message. Thanks to kibana dashboard, the scenario is traced from the initial request, and we can see how it crosses all system 's microservices. The resulting visualization will help us to interpret the trace into a sequence diagram.

A request of sending a message through the system was made and the corresponding traceid is "*d59f1c85873078a2*" as shown on the figure 4.7. By following this tracid in chronological order, the figure shows that the request is first received by chat-ms and then the latter calls user-ms to retrieve the recipient user information. Note that the user-ms responds back to chat-ms in 6 milliseconds thanks to the cache. Chat-ms pro-

cesses the message and then, initiates a request towards notification-ms. The chat-ms responds now back to the client without waiting a response from notification-ms because the communication between them is asynchronous. Chat-ms took 16 milliseconds before responding back to the client. Finally, when notification-ms receives the notification request, it processes it in background.

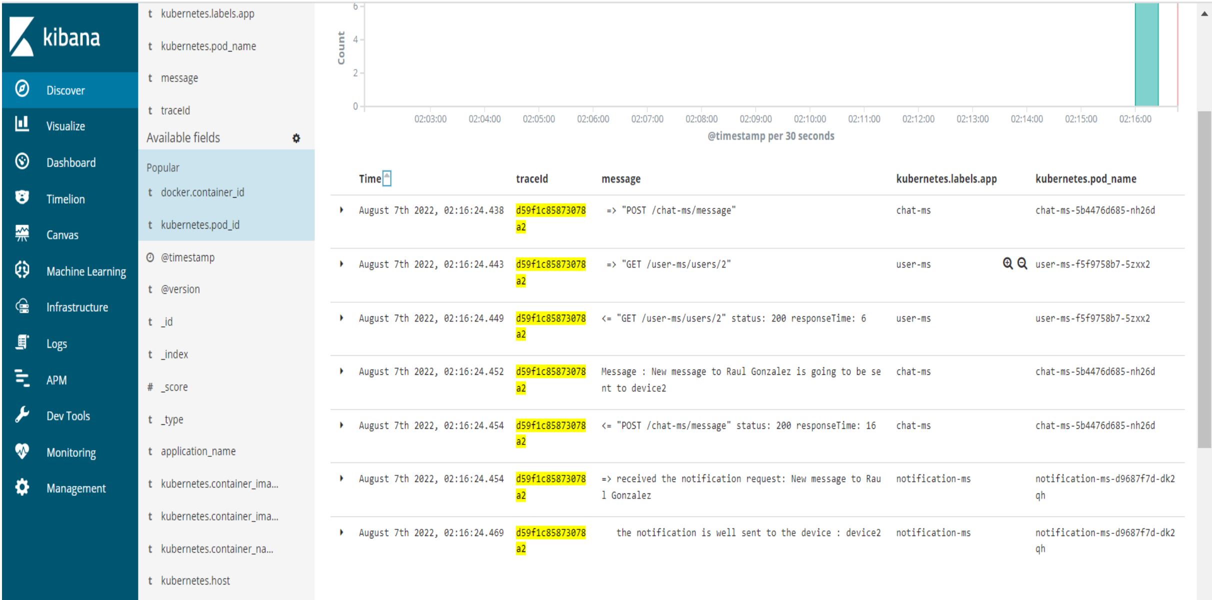



Figure 4.7: Distributed tracing of sending a chat message scenario.

Thanks to the above trace, the scenario can henceforth be interpreted in a conceptual sequence diagram presented at the figure 4.8.

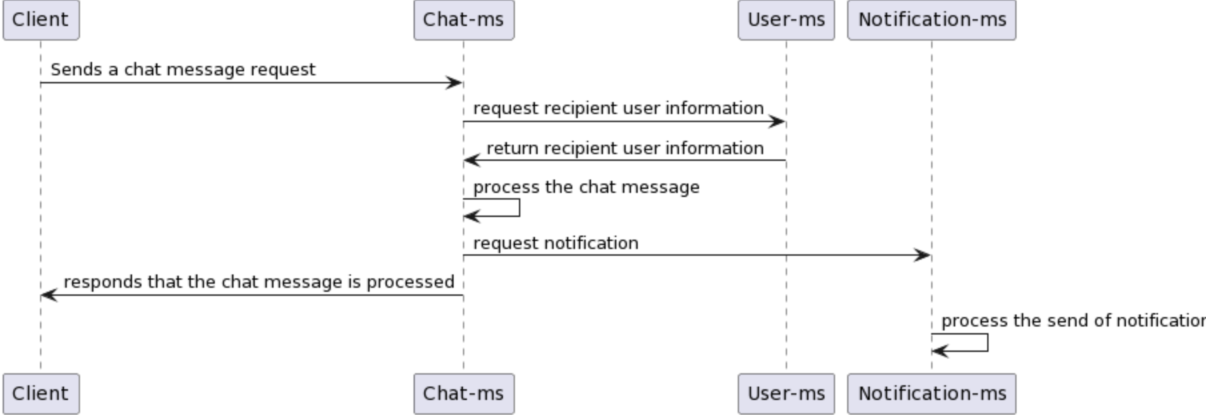



Figure 4.8: Sequence diagram of sending a chat message scenario.

# Chapter 5

# Conclusion

The objective of this thesis is to analyse how systems are built using microservices approach. The microservice architecture aims at dividing the system into small distributed services. In this paper, different concepts of microservice architecture have been presented. However, its advantages and disadvantages have been compared to those of a monolithic architecture. In addition, a messaging system has been developed using the microservices approach to demonstrate related concepts.

The microservices architecture is most suitable for large and complex systems. The modularization puts emphasis on complete independence in terms of autonomy, scalability, development and deployment for each microservice.

The developed messaging system revealed some technologies used to achieve a good scalable and distributed system. Although, the microservices shift the system complexity to the services connection and management. As a result of this challenge, a "Devops engineer" job position was created in information systems companies. Devops engineers are in charge of operational tasks regarding the management of microservices, such as coordinating the continuous deployments, setting up the infrastructure and monitoring the system.

To conclude, we highlight that this thesis requires a deep technical understanding of information systems management to reach better results. However, all the theoretical and practical concepts used in the analysis of the system's architectures, were referenced for whom wants a deeper understanding. As a software engineer, this work was very fruitful

and very interesting.

# Bibliography


[1] Scalability, Wikipedia. url : https://en.wikipedia.org/wiki/Scalability, 23/02/22.

[2] Distributed computing, Wikipedia. url : https://en.wikipedia.org/wiki/Distributed_computing, 23/02/22.

[3] Basic Concepts You Need to Know about Building Large Scale Distributed Systems, Larry — Peng Yang. url : https://medium.com/must-know-computer-science/basic-concepts-you-need-to-know-about-building-large-scale-distributed-system-595 27/02/22.

[4] Maryanne Ndungu.(2019)."*Adoption of the microservice architecture*". Master's Thesis. ÅBO AKADEMI

[5] Goetsch, K. (2017). "*Microservices for Modern Commerce*". California: O'Reilly Media Inc.

[6] Facebook, url : https://www.facebook.com/, 27/02/22.

[7] Google, url : https://www.google.com/, 27/02/22.

[8] Kubernetes, url : https://kubernetes.io/, 27/02/22.

[9] Richardson, C. & Smith, F. (2016). *Microservices From Design to Deployment.* California: NGINX Inc.

[10] Anfel Selmadji. (2019). *From monolithic architectural style to microservice one : structure-based and taskbased approaches.* Université Montpellier.

[11] Karbuja, R. (2016). *Designing a Business Platform Using Microservices.* TECHNISCHE UNIVERSITÄT MÜNCHEN.

[12] Dragoni, N.;Giallorenzo, S.;Lluch-Lafuente, A. & Mazzara, M. (2016). *Microservices: yesterday, today, and tomorrow.* Cham, Switzerland: Springer.

[13] Lewis, J., Fowler, M. (2014). *Microservices.* url : https://martinfowler.com/articles/microservices.html, 07/07/2022

[14] Tiago Costa Santos. (2018). *Adopting Microservices, Migrating a HR tool from a monolithic architecture*. Tecnico Lisboa.

[15] Kasper Stenroos. (2019). *Microservices in Software Development*. Metropolia University of Applied Sciences.

[16] avinetworks, url: https://avinetworks.com/what-are-microservices-and-containers/, 19/07/2022

[17] smartbear, url: https://smartbear.com/learn/performance-monitoring/monitoring-microservices/, 26/07/2022

[18] Redis Pub/Sub, url: https://redis.io/docs/manual/pubsub/, 26/07/2022

[19] Redis Cache, url: https://redis.com/fr/solutions/cas-dutilisation/cache/, 26/07/2022

[20] ElasticSearch, url: https://www.elastic.co/, 26/07/2022

[21] fluentd, url: https://www.fluentd.org/, 26/07/2022

[22] Kibana, url: https://www.elastic.co/fr/kibana/, 26/07/2022

[23] Docker, url: https://www.docker.com/products/docker-desktop/, 26/07/2022

[24] Minikube, url: https://minikube.sigs.k8s.io/docs/, 26/07/2022

[25] YAML, Wikipedia. url : https://en.wikipedia.org/wiki/YAML, 06/08/22.

[26] Spring Boot, url : https://spring.io/projects/spring-boot, 06/08/22.

[27] Docker hub, url : https://hub.docker.com/, 06/08/22.

# Appendix

- **Appendix 1:** microservices-architecture.zip ( (`https://github.com/jeanboscorwi/microservices-architecture`)

  This appendix contains 3 folders :

  - **Monitoring :** This folder contains 3 subfolders : *elasticSearch*, *fluentd* and *kibana*. Each subfolder contains configuration files of the corresponding application.
  - **Redis :** This folder contains the configuration file of redis application.
  - **Services :** This folder contains 3 subfolders: *chat*, *notification* and *user*. Each subfolder contains the source code of the corresponding microservice.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**Louvain School of Management**
Place des Doyens, 1 bte L2.01.01, 1348 Louvain-la-Neuve
Boulevard Emile Devreux 6, 6000 Charleroi, Belgique
Chaussée de Binche 151, 7000 Mons, Belgique
**www.uclouvain.be/lsm**