Heriot-Watt University

Masters Thesis

# Experimental comparison of auto-scaling cloud computing platforms for a micro-services web application

*Author:*
Yann Jaffrennou

*Supervisor:*
Dr. Arash Eshghi

*A thesis submitted in fulfilment of the requirements for the degree of MSc.*

*in the*

School of Mathematical and Computer Sciences

August 2020

# Declaration of Authorship

I, Yann JAFFRENNOU, declare that this thesis titled, 'Experimental comparison of auto-scaling cloud computing platforms for a micro-services web application' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Yann JAFFRENNOU

Date: August 2nd, 2020

# *Abstract*

Cloud computing technologies are increasingly used for deploying web applications and services. With their very own characteristics, they offer new ways to control and reduce hosting costs. These new practices and tools are detailed in this paper, and experiments are made to study the performances and costs of an experimental micro-services web backend application using Google Kubernetes Engine with an auto-scaling cluster, Google Cloud Run and a more traditional deployment using Google's auto-scaling Infrastructure as a Service. These experiments aim to evaluate the performances, ability to scale and cost of each deployment in a comparative way.

# *Acknowledgements*

I would like to sincerly thank my supervisor, Dr. Arash ESHGHI, for his interest for the subject and willingness to supervise me on a subject new to him.

I am also grateful to the Alana team, and especially Ioannis PAPAIOANNOU, for his responsiveness, the useful information he provided me concerning Alana, and his big contribution to finding a simple yet efficient experimental platform.

Finally, I would like to thank Dr. Jessica Chen-Burger for her trust and support.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **GCP** | **G**oogle **C**loud **P**latform |
| **AWS** | **A**mazon **W**eb **S**ervices |
| **MA** | **M**onolithic **A**rchitecture |
| **MSA** | **M**icro **S**ervices **A**rchitecture |
| **SOA** | **S**ervice **O**riented **A**rchitecture |
| **VM** | **V**irtual **M**achine |
| **OS** | **O**perating **S**ystem |
| **CSP** | **C**loud **S**ervices **P**rovider |
| **DBaaS** | **D**ata**B**ase **a**s **a** **S**ervice |
| **FaaS** | **F**unction **a**s **a** **S**ervice |
| **IaaS** | **I**nfrastructure **a**s **a** **S**ervice |
| **PaaS** | **P**latform **a**s **a** **S**ervice |
| **SaaS** | **S**oftware **a**s **a** **S**ervice |
| **CaaS** | **C**ontainer **a**s **a** **S**ervice |
| **NLU** | **N**atural **L**anguage **U**nderstanding |
| **WBS** | **W**ork **B**reakdown **S**tructure |
| **GCE** | **G**oogle **C**ompute **E**ngine |
| **GKE** | **G**oogle **K**ubernetes **E**ngine |

# Chapter 1

# Introduction

This chapter aims to define the context, motivations, aims and objectives of this Master Project.

## 1.1 Context and Motivations

As the new evidence for deploying and hosting web applications and services, cloud computing services are extensively used by companies [Tanni and Hasan, 2017]. These types of hosting services have their very own characteristics, including a price-for-usage billing [Huang et al., 2012] and an almost unlimited amount of resources available [Ko et al., 2014]. At the same time, a massive adoption of Agile project management in software development increased the need for practices permitting safe and easy continuous deployments and isolation of software components. To benefit from cloud computing characteristics and solve issues faced in Agile project management, a new model of software architecture is becoming very popular, and slowly evolves to the new default choice for any medium to big size web application: the micro-services architecture [Chen et al., 2017]. A focus on scalability brought by cloud computing services pricing models and micro-services architectures led to a massive adoption of new tools permitting the creation of ephemeral environments for software to run, built over new technologies and principles of virtualization, such as Docker and Kubernetes [Pereira Ferreira and Sinnott, 2019].

The combination of new software architectures, new virtualization technologies and new hosting services constitute entirely new technological stacks for any new application, and comparisons of their efficiency with traditional/historical technologies permit to determine what future improvements could be made in software development practices, and in the choice of technologies to use to achieve lower costs in production environments.

Nowadays, IT engineers are very likely to use cloud computing services to deploy web services or applications, and a good understanding of these technologies is crucial. Their new pricing models can be a threat for companies (especially startup companies with lower budgets). Thus choices have to be made between multiple software architectures and many cloud computing services for deploying, hosting and scaling applications and services. Each of these tools are generally based on different metrics for billing [Lehrig et al., 2015], implying another layer of confusion and difficulty to predict costs and optimize deployments. A good knowledge and understanding of these technologies is then very valuable for companies, permitting to achieve strategical choices leading to cost limitations and operational improvements.

## 1.2 Aims and Objectives

This Master Project aims to review and assess cloud computing services for deploying, hosting and scaling web services/applications proposed by the three main cloud services providers (Amazon Web Service, Microsoft Azure and Google Cloud Platform). With its experiments, a reusable protocol will be established to assess and compare different cloud services for any kind of web application or service, permitting to reduce their running cost.

The objectives of this project are the following:

1. Gather knowledge on new trends of cloud computing and scalable web applications
2. Evaluate the scalability performances of two cloud computing services
3. Experimentally compare the running costs of applications deployed using Infrastructure as a Service and Container as a Service (see section 2.1.3.3) on Google Cloud Platform, Amazon Web Services and Microsoft Azure (see section 2.1.3)

4. Experimentally determine the cheapest deployment for Alana Hub ensuring a good quality of service

# Chapter 2

# Literature review

This chapter aims to review all the existing technologies in use for this project, to assess and summarize relevant papers concerning different topics addressed in this paper, and to propose a few possible future work to go further in the objectives of this project.

## 2.1 Background

The purpose of this section is to gather all the knowledge needed to understand the objectives of this project, the problematic raised and the experiments that will be led to solve it. It aims to define and explain web applications and their architectures, existing practices and tools to achieve scalable web applications, and cloud computing infrastructures and services.

### 2.1.1 Web applications

#### 2.1.1.1 Principles

An application program can be defined as a "Complete, self-contained computer program [...] that performs a specific useful task, other than system maintenance functions" [BusinessDictionary, 2020]. Web applications are a sub-type of applications, that can be described as "a software program that runs on a web server" [Christensson, 2014]. Web applications often depends on web services. According to Christensson [2017], "A web

service is an application or data source that is accessible via a standard web protocol (HTTP or HTTPS). Unlike web applications, web services are designed to communicate with other programs, rather than directly with users". Web applications can integrate web services in their core, or rely on external web services. Thus a web application is a complete package that brings functionalities to its users in an interactive way. Web services are not meant to be used by users directly, but to interact with other web applications or services.

Consequently, a choice is often made by developers to isolate the interactive part of their applications (called front-end) and an ensemble of web services that permits these applications to benefit from server-side business logic and functionalities (e.g. dialogues with databases). This architecture permits to have multiple front-end application (e.g. android application, web application...) relying on the same back-end services.

#### 2.1.1.2 Monolithic architecture

One of the two possible software architectures for web applications is called 'monolithic architecure' (MA). A monolithic application can be defined as "an application with a single code base that includes multiple services" [Al-Debagy and Martinek, 2018]. Thus a monolithic application may include multiple services, but all these services are deployed and hosted as a whole. This software architecture is historical, and represented the "traditional approach to software development" [De Lauretis, 2019] for many years, used by the biggest companies in the world like Netflix, Amazon and Ebay.

Monolithic applications are generally simple to develop, test and deploy. However, their limitations are quickly reached when they get bigger, harder to debug, more complex and less comprehensible [Chen et al., 2017]. In other words, monolithic applications are convenient and easy to manage for small projects, but their strengths become limitations for larger applications.

#### 2.1.1.3 Micro-services architecture

As a new growing trend in software development, micro-services architecture (MSA) aims to solve the problems of monolithic applications. Micro-services can be defined as "small, autonomous services that work together" [Newman, 2015]. Micro-services

architectures constitute a part of the concept of Service Oriented Architecture (SOA), allowing to build complete applications as an agglomeration of independent and specific pieces of software defining services that work together [De Lauretis, 2019].

This type of software architecture is more and more adopted by big companies such as Amazon, Netflix, LinkedIn, Soundcloud and many others [Taibi et al., 2017], becoming the new default choice when developing an important and/or cloud-hosted web application.

In the context of web applications, monolithic and micro-services architectures can be represented as described in the figure 2.1.

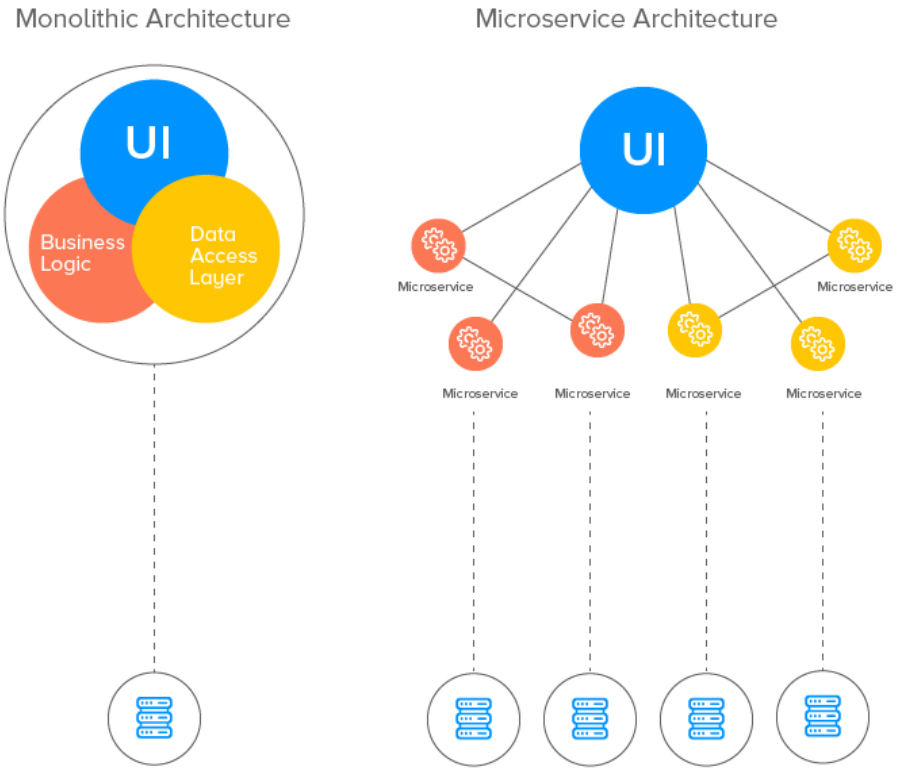



FIGURE 2.1: Monolithic vs Micro-services architecture (appinventiv.com)

### 2.1.2 Scalability

#### 2.1.2.1 Principles and problematics

The notion of scalability can be introduced with two fundamental known characteristics of any computing machine:

1. One machine with a given configuration can accomplish a limited and finite number of tasks per unit of time
2. A machine with a more powerful configuration will be able to accomplish more tasks per unit of time than a machine with a weaker configuration

In this context, an application can be considered scalable if an improvement to the hardware configuration of the server hosting it leads to an improvement in its performance in completing the tasks it is asked to [Barzu et al., 2017b]. Thus a server is not scalable if software limitations impede it from operating in a better way with higher physical resources (e.g. limitations in the use of multi-threading).

At any time, a scalable web server with a given configuration is able to handle a limited number of requests per unit of time. This number of requests can however be overpassed, as it does not depend on phenomenons on the server side but on the number of concurrent clients trying to interact with the server and the intensity of the activity of each client with the server. Consequently, if the amount of requests directed to a web application is bigger than the capacity of the servers hosting this application to handle them, there exists only two ways to upscale the servers' capacities to handle these extra requests:

1. Upgrade the configuration of the servers' nodes to increase individual computing capacities of each node: Vertical scalability (see 2.1.2.2)
2. Add more servers working in parallel to increase the global computing capacities of the server cluster: Horizontal scalability (see 2.1.2.3)

#### 2.1.2.2 Vertical scalability

Vertical scaling is the process of adding hardware resources to a server node to enhance its computing capabilities, thus enhancing the ability of the server node to answer requests [Barzu et al., 2017b]. However, scalability refers to both upscaling and downscaling, and physical resources can either be added or removed in vertical scaling processes. With this technique, an existing server node will have to be shut down to accomplish the upscaling or downscaling actions, and the effect of vertical scaling is similar to the replacement of a server node by another one with a different physical configuration. Thus vertical

scaling actions on a web server composed of a single node will result on a down time of the associated services, although server replacement should be made in a way that guarantees continuity of service. In a clustered configuration including multiple nodes, rebooting a server node to change its physical configuration implies that this node will not be able to handle any request during the upscaling process, resulting in a period of time with a lower computing capacity for the server before achieving better performances when the total number of node returns back to normal. Moreover, algorithms may not use correctly the amount of physical resources they have access to after upscaling actions [Aaqib, 2019].

Thus vertical scalability is a good choice for servers that do not require any continuity of service (e.g. to face an increasing number of employees in a company, for a web server used only by employees during the day), or with multiple node for slow upscaling actions (e.g. email server in a company facing an increasing emailing activity).

#### 2.1.2.3 Horizontal scalability

Horizontal scaling is the process of adding more machines (server nodes) to the server cluster to achieve better performances [Barzu et al., 2017a]. The use of a cluster-based architecture requires having a load balancer to split the traffic and distribute requests to each server node. Horizontal scalability solves the main issues of vertical scalability: there is no need to reboot any server node during upscaling and downscaling processes and no physical action is required on the server nodes, that only needs to be started up or shut down, ensuring a better continuity of service. Moreover, the application running on the servers can be designed to adapt to the physical configuration of each server node, replicating its behavior to each server node working simultaneously. However, the load balancing component has to dynamically adapt to the variable number of server nodes running.

Another great advantage of horizontal scalability is that it permits to implement auto-scaling systems on physical infrastructures: server nodes can be started up and shut down dynamically in a fully automated way, depending on the capacity of each working server node to handle the requests forwarded by the load balancer. Auto-scaling systems can also be used to restart server nodes presenting issues or bugs, by over-scaling the server cluster of one extra node while the inefficient node is restarted.

#### 2.1.2.4 Virtual machines

The purpose of virtual machines (VM) is to simulate physical machines inside real physical machines. These systems behave like any regular physical machine, despite using virtual components and resources. Virtual machines are isolated from any host operating system (OS) thanks to a hypervisor. A hypervisor is a software that allows kernels to run on top of the physical machine's kernel, by virtualizing all the physical components (CPU, RAM, storage...) needed for a guest kernel and OS to run [Joy, 2015]. The most know hypervisors in use nowadays include VMware, KVM, Xen and Hyper-V (Joy [2015]). According to Murugesan [2012], a typical virtual machine architecture can be represented as shown in figure 2.2.

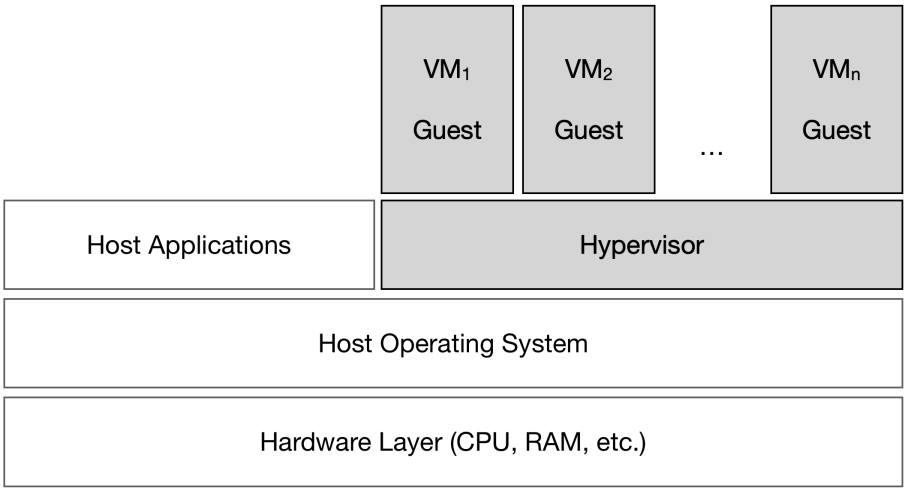



FIGURE 2.2: Typical virtual machine architecture [Murugesan, 2012]

Following this architecture, all the physical components needed by each VM to operate are virtualized by the hypervisor. Thus, a virtual machine's hardware configuration is static and any change in this configuration (comparable to vertical scaling described in section 2.1.2.2) will require the VM to be rebooted, in a way similar to physical machines.

The isolation provided by virtual machines is very useful in server environments. Indeed, OS images can be built to describe the entire hosting environment needed for the deployment of an application or service, permitting to integrate the OS-level configuration as a software component of the application's technological stack. Moreover, if an OS-level issue occurs, a virtual machine can simply be cleared out and recreated with its very own

system image. Consequently, virtual machines are a very good solution to achieve horizontal scalability (see section 2.1.2.3) by running multiple similar virtual server nodes in a single physical server, but they do not allow to achieve vertical scalability (see section 2.1.2.2) in a significantly better way than using physical servers.

#### 2.1.2.5 Container-based virtualization

Containers — or container-based virtualization — are comparable to virtual machines by bringing isolation between different applications' running environment. The key difference between container-based virtualization and hypervisor-based virtualization is the kernel-level architecture: while virtual machines use hypervisors to attribute static amounts of physical resources to each guest kernel, container services are built over the host kernel and have direct access to the physical resources of the host machine [Joy, 2015]. According to Joy [2015], container-based virtualization can be represented as described in figure 2.2.

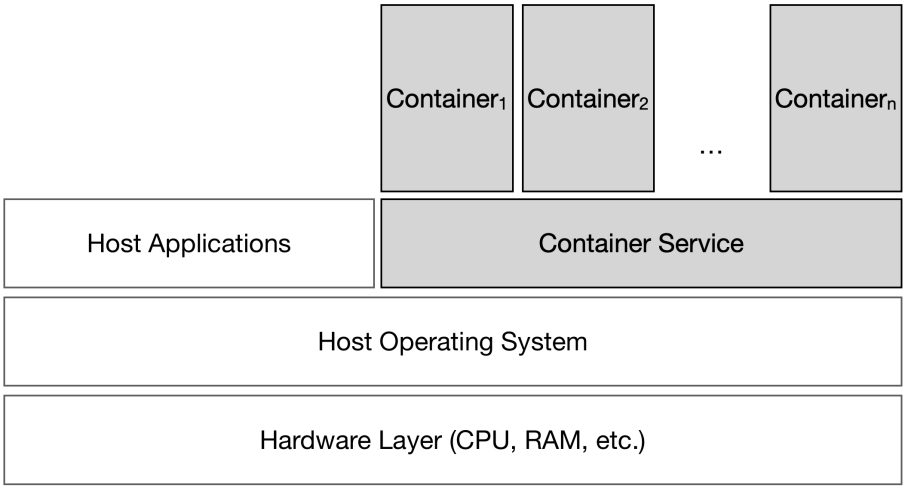



FIGURE 2.3: Container-based virtualization architecture [Murugesan, 2012]

Despite apparent similarities, this architecture is very different from hypervisor-based virtualization. In this architecture, each container runs a light OS without any guest kernel, thus hardware resources allocation does not have to be decided and locked before booting up containers. Consequently, each container can use computing resources like any regular application running on the host operating system, through the container service. In this way, containers can be compared to vertical auto-scaling (see section

2.1.2.2) virtual machines, increasing their access to and use of hardware resources depending on their needs. Container-based systems also allow the use of replication to scale horizontally [Joy, 2015].

Many other advantages of containers over virtual machines can be enumerated including portable deployments, fast application delivery and easy deployment [Joy, 2015].

The most popular and used container service nowadays is Docker, an open source platform for distributed systems that uses isolation features of the Linux kernel [Preeth E N et al., 2015].

#### 2.1.2.6 Container orchestration

With the possibility to use multiple similar containers in parallel to scale applications horizontally and the trend of micro-services software architecture (see section 2.1.1.3) leading to the use of a high number of containers for each service deployed, a need for container orchestration tools emerged [Pereira Ferreira and Sinnott, 2019]. According to Pereira Ferreira and Sinnott [2019], the purpose of these tools is to "control the automation tasks of deployment, scaling, and overall operation of containerised applications across the infrastructure". Thus container orchestration software provides all the required functionalities to manage complex container-based applications over multiple servers and/or server nodes, adding horizontal auto-scaling functionalities to the inherent vertical auto-scaling behaviors of containers.

The most popular container orchestration tool is Kubernetes, an open-source software developed by Google [Dewi et al., 2019]. Operating at the core of Google technologies, Kubernetes is responsible for starting up over two billion containers every week across their different business units [Pereira Ferreira and Sinnott, 2019].

Kubernetes uses a set of pods constituted of containers, and distributes them over an ensemble of workers (physical or virtual machines) depending on the health and work load of each worker. This distribution, scheduling and management of pods is done by the Kubernetes masters [Dewi et al., 2019]. A simple representation of Kubernetes is proposed in figure 2.4.

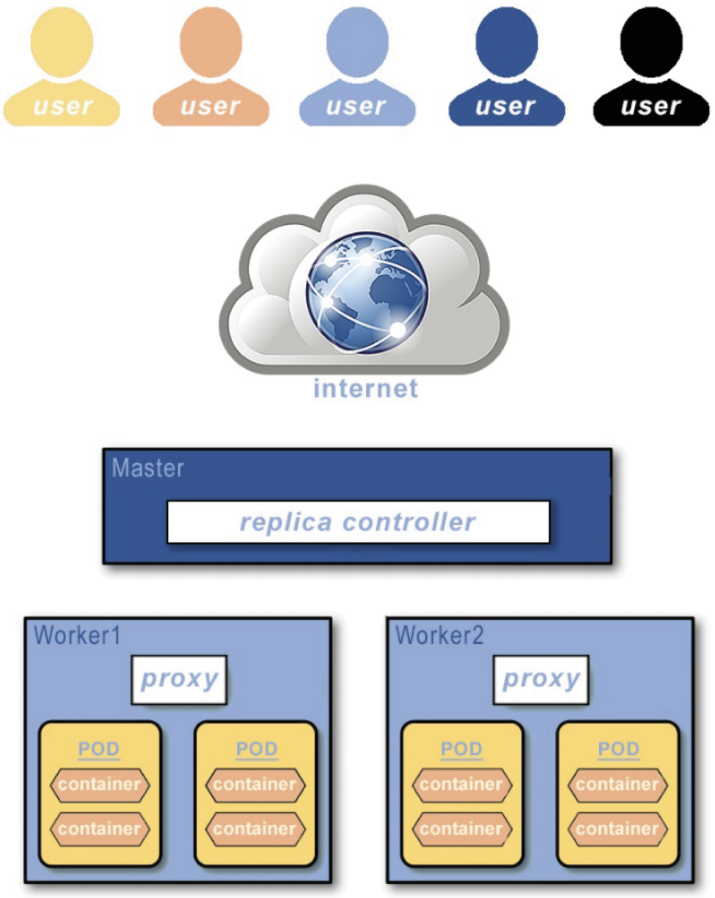



FIGURE 2.4: Simple Kubernetes architecture (Dewi et al. [2019])

### 2.1.3 Cloud computing

With its very own characteristics, cloud computing makes all virtualization and auto-scaling technologies a must-have. This section aims to describe the use of these technologies in cloud computing infrastructure, and to review different auto-scaling services proposed by the three main cloud actors.

#### 2.1.3.1 Principles

One of the key characteristics of cloud computing is the availability of resources and tools as a service to be used by clients. It represents a significant step in service oriented architectures (SOA), providing computational resources without any need for the users to understand the underlying technologies [Murugesan, 2012]. Cloud computing infrastructures differentiate by having massive resources available, and supporting on-demand

scalability of users' computational needs without the limitations of regular hosting or physical server clusters. They also bring new revenue models with users paying for the amount of resources they use rather than for a pre-defined amount of resources provided for a fixed price and paid every month/year [Murugesan, 2012].

Consequently, cost reductions of cloud hosting are achieved by reducing the amount of resources used. In this context, scaling applications is crucial: any application deployed on cloud computing infrastructures should always use the minimum amount of resources it needs to accomplish its given tasks [Moldovan et al., 2016].

#### 2.1.3.2 Public cloud computing actors

Among hundreds of cloud services providers (CSP), the three leading actors of cloud computing are Amazon Web Service (AWS, created in 2006), Microsoft Azure (created in 2010) and Google Cloud Platform (GCP, created in 2008) [Wahid and Banday, 2018]. In 2017, their market shares was divided as 47.1% for AWS, 10% for Azure and 3.95% for GCP. However, the evolution of these market shares has to be considered as each of these actors launched their services at different time. An analysis made by Canalys included a calculation of these annual growths as presented in the table 2.1

| CSP | Q4 2019 (US$ billion) | Q4 2019 Market share | Q4 2018 (US$ billion) | Q4 2018 Market share | Annual growth |
|---|---|---|---|---|---|
| AWS | 9.8 | 32.4% | 7.3 | 33.4% | 33.2% |
| Azure | 5.3 | 17.6% | 3.3 | 14.9% | 62.3% |
| GCP | 1.8 | 6.0% | 1.1 | 4.9% | 67.6% |

TABLE 2.1: Cloud Service Provider Growth

Following these annual growth, AWS, GCP and Azure have to be considered as equivalent in their potential adoption in the next few years. Therefore, the choice of a CSP for hosting a new web application has to be made following other criteria, such as pricing.

#### 2.1.3.3 Services

The services proposed by cloud services providers are numerous, including Databases as a Service (DBaaS) that permit to host databases in the cloud without having to deploy them on virtual machine instances, Function as a Service (FaaS) that permits to build

and call simple functions in the cloud and get responses without any hosting configuration, and many others. However, the three main services provided are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) (Wahid and Banday [2018]). These services express different level of management and responsibility shared between users (CSP's clients) and CSP's [Prajapati et al., 2018]. According to Nunnikhoven [2016], these different levels of services can be represented as shown in figure 2.5. This representation is however discussed, but provides a good overall understanding of the main differences between these services.

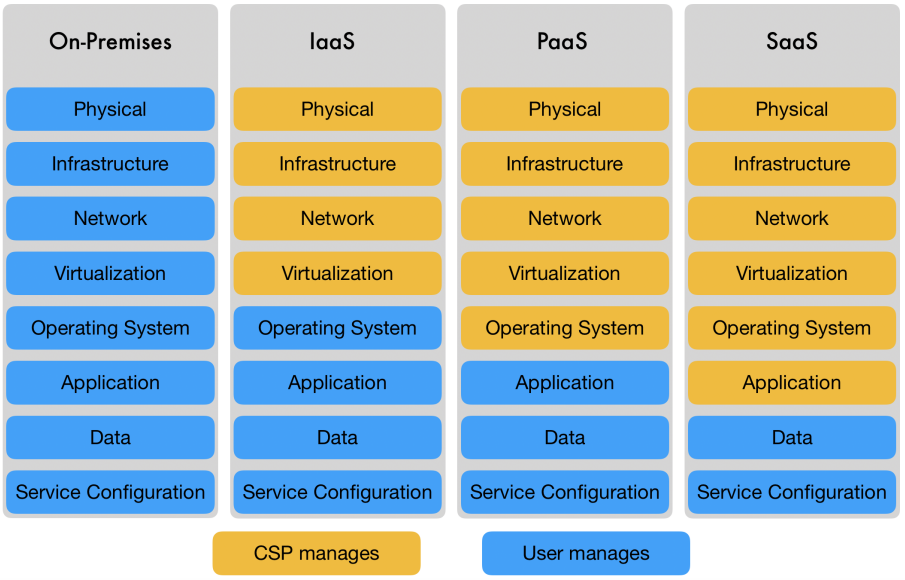



FIGURE 2.5: Service level comparison between On-Premises, Iaas, PaaS and SaaS

IaaS includes AWS EC2, Google Compute Engine and Microsoft Azure Cloud VMs ; PaaS includes Google App Engine, AWS Elastic Beanstalk and Microsoft Azure Web Apps ; SaaS includes GMail, Office 365... For all these services, auto-scaling tools are available with different characteristics: IaaS autoscaling tools usually work with a user-managed load balancer associated to a group of VM instances that grows up as the pressure on each running instance reaches a certain level, whereas PaaS and SaaS include hidden scalability services as the user is not responsible for managing the way VM instances operate. Consequently, PaaS and SaaS are qualified as 'serverless hosting'.

Another important type of services is Container as a Service (CaaS). These services often use the container orchestration tool Kubernetes, thus referred to as Managed Kubernetes Services [Pereira Ferreira and Sinnott, 2019]. Relying on the high levels of virtualization cloud computing technologies are built over, this specific type of service permits to

manage application deployments with a high focus on scalability and reliability using Kubernetes, without any need to manually manage any cloud virtual machine directly.

### 2.1.4 Alana

This subsection is dedicated to a review of the Alana service, as the objectives of this project include to apply and test auto-scaling deployment methods to one of its components: Alana Hub. All the knowledge brought in this subsection is extracted from a paper written by Alana's creators, Cercas Curry et al. [2018].

#### 2.1.4.1 Global architecture and services

According to its creators [Cercas Curry et al., 2018], Alana's architecture can be represented as described in figure 2.6.

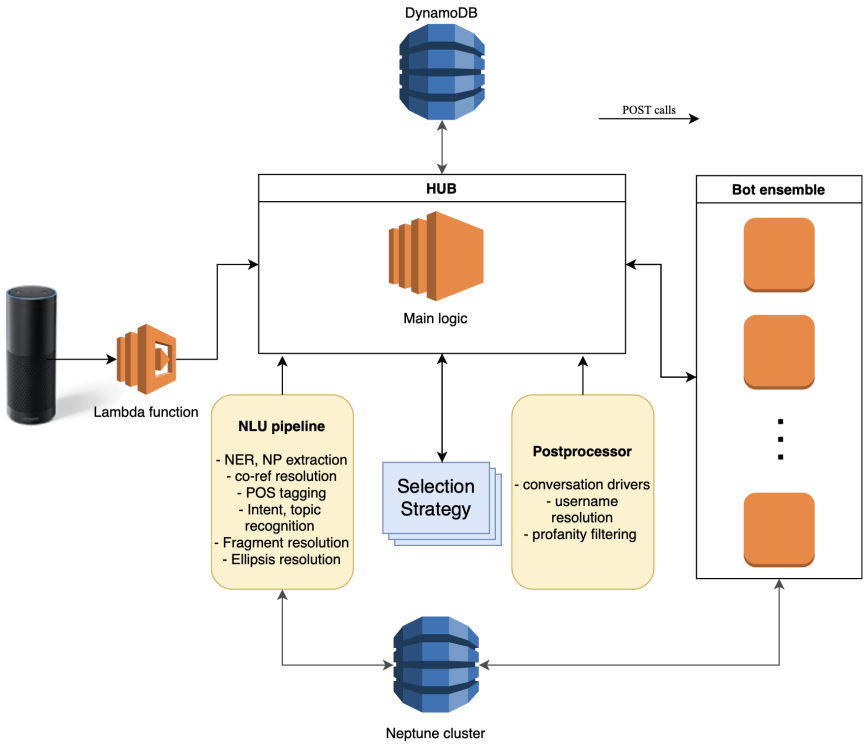



FIGURE 2.6: Alana Architecture [Cercas Curry et al., 2018]

This architecture makes Alana a modular application that can be enriched with bots to unlock new functionalities. One of the key component of this architecture is called the 'Hub', responsible for the main logic of Alana.

#### 2.1.4.2 Alana Hub

The Hub defines how all the other modules interact with each others, and integrates a Natural Language Understanding (NLU) pipeline and a Postprocessor. The NLU pipeline is responsible for understanding the written transcript of the oral user's request, whereas the postprocessor is responsible for transforming one of the response candidates proposed by the bots before transmitting it back to the user through lambda functions (FaaS type cloud computing, briefly described in section 2.1.3.3) and vocal assistants.

In the production configuration used during the 2018 Alexa Prize, Alana Hub was hosted on a simple AWS EC2 instance [Cercas Curry et al., 2018]. This limited configuration may be a bottleneck in the complete workflow of Alana, resulting in dropping off requests and leaving users unsatisfied. Moreover, the lack of scalability of this single IaaS instance could result in unnecessary hosting fees at idle.

Moreover, Alana Hub adopts a monolithic architecture and includes different services (called components) build over different technologies such as natural language processing, data querying and script-based algorithms. With the current hosting configuration, these components have to work in the same environment with a unique set of resources to share.

## 2.2 Related work

The purpose of this section is to gather information and conclusions from related studies on the topics of web applications architecture, containers and virtualization, managed container services and cloud services providers.

### 2.2.1 Monolithic vs. Micro-services architecture

Many studies have been led to compare the performances and deployment costs of web applications built over monolithic and micro-services architectures.

Villamizar et al. [2015] studied the deployments, performances and running costs of an application developed and implemented using the monolithic approach and the micro-services pattern. After deploying these two versions of their application using Amazon Web Services (IaaS, see section 2.1.3.3), they observed the following:

1. The latency of response is not considerably impacted by micro-services architecture
2. The running cost of micro-services version was 17% lower than the running cost of the monolithic version, due to its more granular scalability

Other conclusions were made on the advantages and disadvantages of both architecture in software development and deployment processes (see sections 2.1.1.2 and 2.1.1.3).

In another study led by Villamizar et al. [2016], the same conclusions were made on economical benefits of micro-services architecture operated by the CSP's client with a cost reduction of 13.42% compared to the monolithic architecture, and a cost reduction of 77.08% using the Amazon Lambda service (FaaS, see section 2.1.3.3) operated by the CSP.

### 2.2.2 Virtual Machines vs. Containers

Comparisons between virtual machines and Linux containers have been made, especially using Docker, Kubernetes and cloud VMs at IaaS level.

Joy [2015] led a study to compare the performances of an application hosted on virtual machines vs. containers.

Their first experiment consisted in deploying an application composed of a Joomla front-end and a PostgreSQL back-end with virtual machines in the cloud (EC2 instances on AWS) and containers (Docker running on two On-Premises servers).

Their second experiment aimed to compare the scaling performances of virtual machines and containers. They deployed a load-balanced Wordpress application using auto-scaling EC2 instances on AWS on one side, and Kubernetes and Docker running on two On-Premises servers on the other side.

From these two experiments, the following observations were made:

1. The container-based deployment of their application permitted to resolve more than 5 times as many requests as the VM based deployment
2. The virtual machine configuration takes more time to process one single request than the containers-based configuration
3. The container-based deployment was scaling 22 times faster than the virtual machines deployment

### 2.2.3 Managed container services

A very detailed study has been led by Pereira Ferreira and Sinnott [2019] on hosted and managed Kubernetes+Docker deployments. The aim of this study was to compare different key performances (computing performance, memory performance, disk performance, network performance...) between an application deployed with Kubernetes and Docker with IaaS level services, and the same application deployed using Managed Kubernetes Services (see section 2.1.3.3) on the main Cloud Services Providers: Google Cloud Platform, Amazon Web Services and Microsoft Azure (see section 2.1.3).

Their conclusions were the following:

1. Choosing a managed Kubernetes service does not imply any specific performance enhancement associated to the managed service itself
2. There is no unique best solution, each cloud service provider has its own pros and cons in terms of performance

### 2.2.4 Cloud services providers

Despite the studies led by Pereira Ferreira and Sinnott [2019] detailed in section 2.2.3, other studies comparing different cloud services providers shown differences between their performances. Some experiments led by Tanni and Hasan [2017] using CloudSim resulted in slightly better performances with AWS in comparison to GCP.

On the pricing side, a theoretical study led by Wahid and Banday [2018] concludes that pricing-to-performance ratio depends on the type of machines and resources selected.

AWS wins the pricing competition in many cases, but GCP and Azure are not defeated in all categories.

## 2.3 Conclusion

Cloud computing can be perceived as a revolution in the way web applications and services are hosted. Providing their users with an almost unlimited amount of resources to use, web entrepreneurs have to consider scalability from the very beginning of the development of any new application. Their architectural choices have implications on the development processes and the candidate services to use for production hosting. With numerous services offered by the main cloud services providers, the running cost of an application can vary and a good knowledge of these services is crucial. However, despite the best possible understanding of these technologies, the number of existing deployment alternatives for a whole application is still huge and only a few of them can be assessed.

The specific behaviors of each application have strong implications on the resources it will require, and the only reliable way to ensure the best cost optimization of a cloud infrastructure is experimental. The number of hosting configurations and their differences make the evaluation of all possibilities almost impossible, and a limited set of configurations has to be determined before leading any experiment. In this context, it is impossible to ensure that the hosting infrastructure built and used for a web application on the cloud is the most optimized, and a constant effort has to be put in hosting optimization throughout the whole life cycle of every application.

# Chapter 3

# Requirement Analysis

This chapters aims to describe the requirements for this project, and bring explanations on the technologies chosen for assessment and the experiments that will be led to evaluate solutions for deploying Alana in an auto-scaling environment.

## 3.1 MoSCoW Requirements

The requirements for this project are presented in table 3.1 according to the MoSCoW scheme. Further explanations on these requirements are given in this chapter.

## 3.2 Deployments

### 3.2.1 Architecture of Alana Hub

In its current form, Alana Hub has a monolithic software architecture. Thus the monolithic Docker image should only emerge from the current production version of Alana, compacted in a unique Docker image. The micro-services version of Alana Hub is a new version that has to be developed from the monolithic version, splitting its components into micro-services with one Docker image per micro-service. The possible micro-services architectures for Alana Hub will be determined as part of this project (requirement R3, see table 3.1), and some details about its components and their interactions will be

| Requirement ID | Requirement description | MoSCoW | Priority |
|---|---|---|---|
| R1 | Create a test bench composed of 100 GCE instances | M | H |
| R2 | Create a tool to schedule and operate 9 stressing sessions for each experiment, with 3 different values for each of the 3 experimental factors | M | H |
| R3 | Propose at least one micro-services architecture for Alana Hub | M | H |
| R-D1.1.1 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on GCP with Compute Engine | M | H |
| R-D1.1.2 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on GCP with Kubernetes Engine | M | H |
| R-D1.2.1 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on AWS with EC2 | M | H |
| R-D1.2.2 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on AWS with Kubernetes Service | M | H |
| R-D1.3.1 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on Azure with cloud VMs | S | M |
| R-D1.3.2 | Execute one experiment composed of 9 stressing sessions on the monolith of Alana Hub on Azure with Kubernetes Service | S | M |
| R-D2.1.1 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on GCP with Compute Engine | M | H |
| R-D2.1.2 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on GCP with Kubernetes Engine | M | H |
| R-D2.2.1 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on AWS with EC2 | M | H |
| R-D2.2.2 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on AWS with Kubernetes Service | M | H |
| R-D2.3.1 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on Azure with cloud VMs | S | M |
| R-D2.3.2 | Execute one experiment composed of 9 stressing sessions on the micro-services version of Alana Hub on Azure with Kubernetes Service | S | M |
| R4 | Retrieve logs from the slave machines | M | H |
| R5 | Retrieve logs from Alana | M | H |
| R6 | Retrieve data from the cloud hosting services consoles | M | H |
| R7 | Analyse the collected data to propose conclusions on the best deployment solution to choose | M | H |
| R8 | Use any Unix based computer as the master | S | M |
| R9 | Propose a unique application to manage all the requirements | C | S |
| R11 | Test any other type of deployment | W | M |

Table 3.1: MoSCoW requirements

required to propose a possible architecture following these criteria. A possible presentation of such information could be a diagram representing all the modules of the NLU pipeline [Cercas Curry et al., 2018] and their interactions with each others and with the rest of the software. This would allow to assess the possibilities to parallelize the NLU pipeline components to achieve a faster resolution, calling independently each component depending on the specific resources they need.

From the new micro-services architecture, all the choices concerning its deployment (allocation of services to Kubernetes pods, use of cloud VMs to dedicated services...) will be made. The experiments will permit to determine if these choices are good, and if some further adjustments should be made to this new architecture to optimize its behavior in the contexts reproduced in the experiments.

Preferably, all these docker configurations should be provided in a docker-compose format, consisting of at least two components per architecture:

1. docker-compose.yaml: Yaml file containing the upper level configuration of all services with volume mapping, ports, reference to the adequate Dockerfile...
2. Dockerfile: Yaml file containing details on lower level configurations of the Docker images, allowing Docker to build appropriate images to run Alana at start-up

Some additional files may be provided as needed by the docker configuration, such as environment variable files, specific configuration files for some services...

### 3.2.2 Cloud deployments

The experiments of this project consist in simulating different levels of load to the Alana service, and measuring characteristics about its behaviors, performances and running costs. This imply to define different scenarios that will be assessed and compared, with fixed parameters and variables. With the constraints of using Docker in all deployments and using auto-scaling technologies, and the desire to compare deployments with services from the three main cloud services providers (Google Cloud Platform, Amazon Web Service and Microsoft Azure), a choice has been made to evaluate two types of deployments proposed by these three cloud services providers (autoscaling VM group+load balancer ;

Kubernetes with autoscaler), for the two different architectures of Alana. A total of 12 deployment types will then be tested, as described in the table 3.2.

| **Architecture** | **Deployment type** | **Google Cloud Platform** | **Amazon Web Services** | **Microsoft Azure** |
|---|---|---|---|---|
| Monolithic architecture | Autoscaling VM group + Load balancer | Load Balancer Compute Engine instance groups | Network Load Balancer Auto Scaling Group of EC2 instances | Azure Load Balancer Azure autoscale with Virtual Machine |
| | Kubernetes engine | Google Kubernetes Engine cluster with cluster autoscaler | Amazon Elastic Kubernetes Service with cluster autoscaler | Azure Kubernetes Service with cluster autoscaler |
| Micro-services architecture | Autoscaling VM group + Load balancer | Load Balancer Compute Engine instance groups | Network Load Balancer Auto Scaling Group of EC2 instances | Azure Load Balancer Azure autoscale with Virtual Machine |
| | Kubernetes engine | Google Kubernetes Engine cluster with cluster autoscaler | Amazon Elastic Kubernetes Service with cluster autoscaler | Azure Kubernetes Service with cluster autoscaler |

TABLE 3.2: Alana Hub Deployments

Each of these deployments will be associated with one separate and independent factorial experiment, and their factors are detailed in subsection 3.4.1.

Each cloud services provider proposes a free plan to let users try and use their services without having to pay. The services offered by each one of them vary.

Google Cloud Platform proposes the most elastic offer for free: a USD 300 free credit available for a duration of 12 months that can be used with any service, without any limitation or restriction on what can be used and tested [Google, 2020a]. However, a limitation set the maximum number of vCPU cores running at the same time to 8. This will be a limitation for this project, and a budget will be required to cover the fees it could imply.

Amazon Web Services proposes a quite restricted free plan, that only includes a free use of a limited set of services for a duration of 12 months. With this free plan, the configuration of EC2 instances is limited to t2.micro, including only one vCPU and 1Gb of memory [Amazon, 2020b]. These instances will not be sufficient for the experiments of this project .

Microsoft Azure proposes a hybrid free plan for the same duration of 12 months, including a free access to a limited set of services (same VM instances as AWS, not sufficient for Alana), and a USD 200 credit that can be used freely over the first 30 days, comparable to Google Cloud free credit [Microsoft, 2020b].

Thus Google Cloud Platform and Microsoft Azure could be used with free credits for these experiments, but AWS will bill for the use of their services from the very beginning.

A financial credit should then be provided to cover AWS' costs, and the extra costs of GCP ans Azure.

## 3.3 Experimental platform

An experimental platform is required to stress Alana in all its deployment configurations. The purpose of this test bench is to simulate dialog sessions with multiple users, with a focus on minimizing any internal factor that could limit the overall performances of the system from the client's point of view. This experimental platform will be composed of multiple Google Compute Engine instances (highest free credit, this will be managed from a separate Google account) acting as slaves supervised by a master machine ordering tasks. This configuration can be represented as shown in figure 3.1

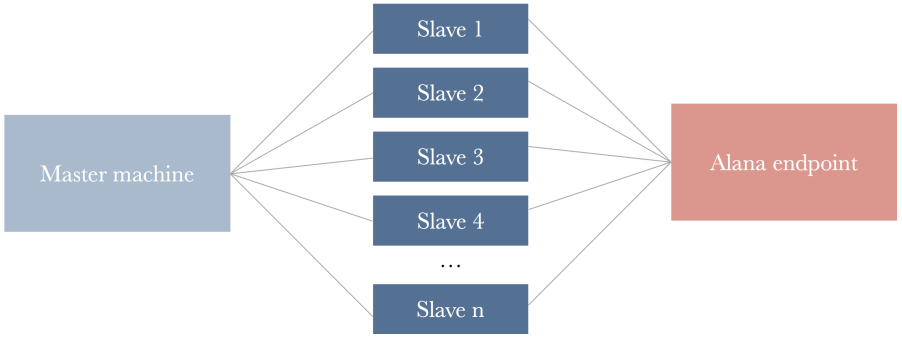



FIGURE 3.1: Experimental platform architecture

Each slave machine composing this test bench represents one user, and each experiment will use a number of these machines to send requests and get responses from Alana.

## 3.4 Methodology for the experiments

### 3.4.1 Description of the experiments

The 12 experiments led will inherently provoke different behaviors of Alana, depending on the evolution of the number of requests it has to respond to. These four behaviors are as follows:

1. Idle: Alana is deployed, but no user is interacting with it

2. Upscaling: Alana is facing an increasing number of requests per minute, and has to scale up to handle them
3. Continued load: Alana handles a big amount of requests per minute, and has to stay stable and coherent during this period
4. Downscaling: Alana is facing a decreasing number of requests per minute, and has to scale down to limit costs

Experiments with different intensity of load for the "continued load" phase should be made, to determine the relations between the number of requests received by Alana on a given period of time (request frequency), the number of VM instances (or Kubernetes workers) required to handle them, and the cost of the cloud infrastructure. Thus the intensity of each stressing session has to be controlled. From Alana's point of view, the intensity of a session depends on the 3 following characteristics:

1. Number of concurrent users
2. Time elapsed between Alana's response and the user's next request
3. Complexity of each request in the way Alana computes it

This last parameter is not controlled, and the sample conversations provided (described in section 3.4.2) should be made ensuring that the dialogues provided represent a general ensemble with a great variety of requests and subjects. The characteristics 1 and 2 can be integrated as variables for each stressing session, with a transfer of parameter from the master machine to each slave machine.

Finally, a total of 12 stressing sessions will be led for each deployment scenario described in section 3.2.2. For each of these sessions, a total of 3 factors will be modified. These 3 factors are as follows:

1. Number of concurrent users
2. Request frequency
3. Duration of the session

For each of these 3 factors, 3 separate values will be used. These values will be determined as part of the project.

### 3.4.2 Utilization of the experimental platform

Each slave machine will run a shell script responsible for making curl calls to Alana endpoint, collecting the responses and log the time taken between each request and the associated response. Depending on the factors defined in section 3.4.1, the next request will be sent to Alana after a certain delay. To ensure that all slaves are working in the same time slot, stressing sessions will be scheduled and their characteristics will be transferred from the master machine with a crontab file defining the exact starting time of the session. The most simple way to schedule and launch stressing sessions is to run a shell script on the master machine responsible for building shell scripts and cron files for each slave, before transferring them to the slave machines using scp commands. The slave machines are not ephemeral, and the list of their IP addresses will be exported from Google Cloud Platform. These instances will be created with an instance model providing the public SSH key needed to make these transfers from the master machine to each slave.

To simulate user interactions with Alana, samples of existing conversations should be provided. The number of these samples is not limited as conversations can be looped to simulate longer interactions. However, with a probable number of 100 users simulated and an average duration per human-to-Alexa conversation of 2.20 minutes with 11 rounds [Cercas Curry et al., 2018] and the intuition that conversations with a same number of rounds will be approximately 2 times shorter (no reflexion/formulation time for the simulated user, and bypass of Alexa's speech recognition), a minimum number of 1000 conversations should be provided to simulate sessions with 100 users for a duration of 10 minutes, with an average duration per session of 1.10 minutes, without reusing any conversation. For longer simulation sessions, discussion samples will be reused.

The format of these discussion samples can be chosen by the Alana team among many options: SQL database, no-SQL database, CSV file, text file... Each conversation should be clearly distinguished, and indications of a user identifier, a session identifier and a round identifier could be used to make each interaction unique.

Finally, these samples should be as representative as possible of real-world interactions, with a consideration of the variety of subjects and conversation types real-world scenarios could include.

## 3.5 Data retrieving and analysis

To achieve analyses on the behavior, scalability and cost of Alana at scale, data has to be collected from Alana Hub, Alana's hosting and the experimental platform. The purpose of this data collection is to include all the information needed to characterize the links from causes to consequences at every level. This data is described in table 3.3.

| Data ID | Data description | Source |
|---|---|---|
| 1 | Time taken by Alana to respond to each request | Alana Hub |
| 2 | Time taken by each bot to provide a response to Alana Hub for each request | Slave machines |
| 3 | Time taken by Alana Hub's NLU pipeline to process each request | Alana Hub |
| 4 | Number of server instances used over time for each session | Cloud hosting provider |
| 5 | Amount of CPU resources used by Alana Hub over time for each session | Cloud hosting provider |
| 6 | Amount of RAM used by Alana Hub over time for each session | Cloud hosting provider |
| 7 | Cost of Alana Hub cloud infrastructure over time for each session | Cloud hosting provider |

TABLE 3.3: Experimental data

The nature of these analyses will be determined as part of the R7 requirement.

# Chapter 4

# Professional, Legal, Ethical and Social Issues

This project and its context bring some professional, legal, social and ethical issues that have to be covered. All the necessary precautions are taken to ensure that these issues are controlled, and will not raise any problem. This chapter brings details on these issues and the way they should be handled.

## 4.1 Legal

The main legal issue for this project concerns the use of Google Cloud Platform, Amazon Web Services and Microsoft Azure with free trial accounts for a research project. Each of these cloud services providers have their own contracts and our use of their services has to be adequate to their policies. This section provides details on the contracts for these three services, to ensure that no contractual abuse will be made.

### 4.1.1 Amazon Web Services

Amazon presents on its website, with a Question/Answer format, a clear mention ensuring that their free trial can be used without any specific limitation in comparison with a paid account. Moreover, examples are provided on the use of this free tier for different

purposes including "development and test projects" [Amazon, 2020c]. This project consists in testing deployments for development purposes, and falls into the scope of this particular use. Thus our activity with AWS is perfectly legal for this matter.

### 4.1.2 Google Cloud Platform

The creation of multiple Google accounts by one same person to get access to multiple free trials is technically possible, but not allowed by the Terms and Conditions of Google Cloud Platform. As described in the Supplemental Terms and Conditions For Google Cloud Platform Free Trial [Google, 2020c], "Only new Google Cloud Platform customers are eligible to participate in the Free Trial". A misconduct to this rule can lead to a suspension for the user concerned. If a project has only one owner, and this owner gets suspended, the project will be considered as orphaned and marked for deletion [Google, 2020b]. The solution proposed by Google to avoid a project from getting orphaned (and potentially deleted) is to make sure that at least two users are set as owners of the project.

Two solutions emerge from this limitation: create new free GCP accounts against their terms of services and accepting the exposure to the risk of project deletion, or opt for a paid account without any risk on the integrity of the projects.

### 4.1.3 Microsoft Azure

Microsoft Azure's free trial is limited to one per new customer: "If you've never tried or paid for Azure before, you're eligible. [...] There is a limit of one account with 12 months free access to products and $200 credit per new customer" [Microsoft, 2020a]. Thus, the creation of multiple accounts by one customer may be technically limited, and prohibited if technically feasible. Alana should then expect to pay for the Microsoft Azure resources used for this project.

## 4.2 Professional

A professional issue that could be raised concerns the possession of proprietary docker images of the Alana Hub service. These ready-to-deploy images are the property of

Alana, and their use should be limited to what Alana explicitly allows. Thus, these files should be kept safely with a great attention to avoiding any kind of leaks. Solutions to ensure this security will be decided with Alana.

## 4.3 Ethical

The use of cloud computing resources implies an energy consumption and environmental cost. The global energy footprint of the IT sector is estimated to consume 7% of global electricity [Cook et al., 2017], thus ethical issues associated to energy consumption and environmental impact have to be considered. According to Greenpeace [Cook et al., 2017], Google matches its energy consumption with an equivalent or larger supply of renewable energy. Thus Google is able to provide its customer with entirely environmentally clean services in terms of energy consumption. Amazon is not transparent enough to assess their environmental impact correctly, but claims to have produced renewable energy at a height of more than 50% of their energy consumption for the year 2018 [Amazon, 2020a]. Things could however have changed since 2018, and the real environmental impact of Amazon Web Services is still difficult to estimate. Finally, Microsoft's energy consumption has also passed the target of at least 50% renewable energy equivalence in 2018 [Smith, 2019].

With the purpose of this project and considering the current hosting of Alana, it is important to note that "[Cloud computing data centers] are typically operating far more efficiently than most independently operated data centers due largely to much higher server utilization rates and better data center design, requiring a much smaller percentage of energy spent on cooling and other non-computing power demand" [Cook et al., 2017]. Moreover, one of the purposes of this project is to determine the less costing — thus most computing resources efficient — way to deploy Alana Hub. Consequently, and despite a potential consumption estimated to a maximum of 50% non-renewable energy, the overall environmental impact of this project is positive and serves the ecologic transition and its ethical principles.

## 4.4 Social

The main social issue with this project concerns the use of conversations between real-life users and Alana as samples for our experiments. To ensure that any impact of the use of these conversations are minimized and the users privacy is respected, all these conversations should be anonymized according to the GDPR definitions, thus containing no information permitting to identify a natural person [European Commission, 2018].

Dispositions should also be taken to ensure that this data will remain under control, avoiding any kind of leaks from this project.

# Chapter 5

# Implementation

This chapter is dedicated to the implementation of the project. It brings details on the application used and the 3 different cloud computing deployments made and assessed.

## 5.1 Application used: architecture, endpoints and responses

To exploit horizontal scalability in the best possible way for the experiments of this project, and in consideration of the current progress of Alana Hub as a dockerised micro-services application, a choice has been made to use an ensemble of microservices APIs constituting a whole web backend application. This application was not built for this specific project, but modifications were made to ensure a good fit between the application and its running environments. The final application used is described in this section.

The Cinema Application used for these experiments finds its origins in a project written in Golang, created by Morejon and shared publicly on GitHub. Its purpose is to represent a simple version of a complete booking system that could be used by any cinema. It includes 4 independent micro-services: users, movies, showtimes and bookings.

Two distinct versions have been created from this original project. Databases apart, the first version (docker-compose file in appendix C) has been designed to run as a standalone application and includes a reverse proxy (ingress) service to distribute requests over its 4 other services distinguished by different host names. The second version (docker-compose file in appendix D) requires external services to direct incoming requests to

the appropriate service. The implementation of a reverse proxy is the only difference between these two versions. Finally, both versions rely on external MongoDB databases (docker-compose file in appendix B) populated with one thousand sample records each. The architectures of these two versions are represented respectively in figure 5.1 and figure 5.2.

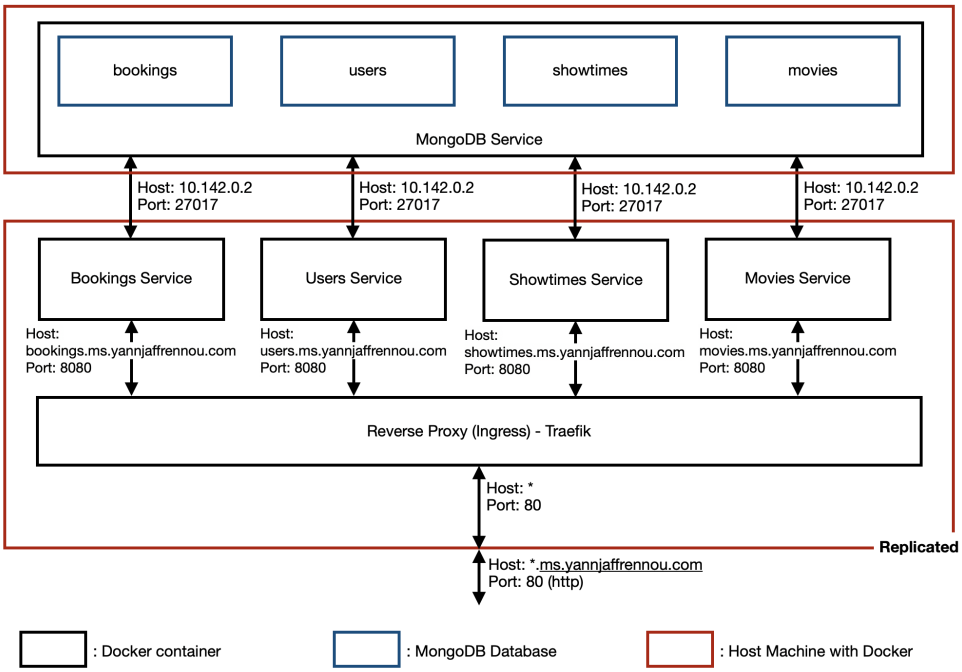



FIGURE 5.1: Cinema Application V1 (standalone)

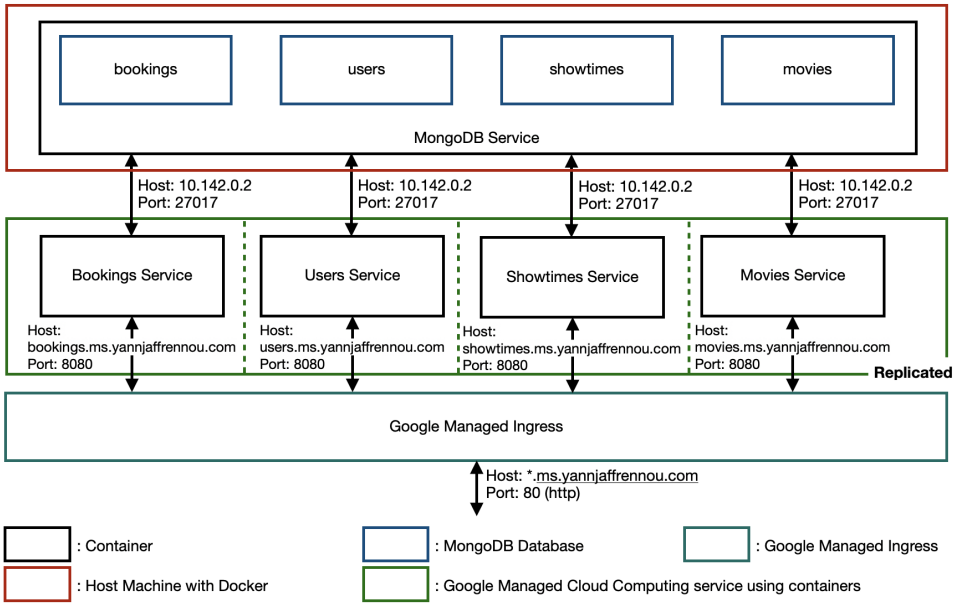



FIGURE 5.2: Cinema Application V2 (using managed services)

With the appropriate domain name and host name configuration, the endpoints of the Cinema Application are described in table 5.1.

| Service | URL | Method | Description |
|---|---|---|---|
| Users | http://users.ms.yannjaffrennou.com/users | GET | Get all users |
| Users | http://users.ms.yannjaffrennou.com/users | POST | Create user |
| Users | http://users.ms.yannjaffrennou.com/users/{id} | DELETE | Remove user by id |
| Movies | http://movies.ms.yannjaffrennou.com/movies | GET | Get all movies |
| Movies | http://movies.ms.yannjaffrennou.com/movies | POST | Create movie |
| Movies | http://movies.ms.yannjaffrennou.com/movies/{id} | GET | Get movie by id |
| Movies | http://movies.ms.yannjaffrennou.com/movies/{id} | DELETE | Remove movie by id |
| Showtimes | http://showtimes.ms.yannjaffrennou.com/showtimes | GET | Get all showtimes |
| Showtimes | http://showtimes.ms.yannjaffrennou.com/showtimes | POST | Create showtime |
| Showtimes | http://showtimes.ms.yannjaffrennou.com/showtimes/{id} | GET | Get showtime by id |
| Showtimes | http://showtimes.ms.yannjaffrennou.com/showtimes/{id} | DELETE | Remove showtime by id |
| Bookings | http://bookings.ms.yannjaffrennou.com/bookings | GET | Get all bookings |
| Bookings | http://bookings.ms.yannjaffrennou.com/bookings | POST | Create booking |

Table 5.1: Cinema Application Endpoints

Example responses from each one of the 4 services included in the Cinema Application for GET methods are presented in JSON listings 5.1 (movies), 5.2 (users), 5.3 (showtimes) and 5.4 (bookings).

```
{
    "data": [
        {
            "id": "57b37b7c377dd100054f9f91",
            "title": "El padrino",
            "director": "Francis Ford",
            "rating": 9.5,
            "createdon": "2016-08-16T20:45:48.704Z"
        },
        ...
    ]
}
```

Listing 5.1: Example response for GET movies.ms.yannjaffrennou.com/movies

```
{
    "data": [
        {
            "id": "57b378da202bba0005a61b87",
            "name": "Manuel",
            "lastname": "Morejon"
        },
        ...
    ]
}
```

Listing 5.2: Example response for GET users.ms.yannjaffrennou.com/users

```
{
    "data": [
        {
            "id": "57b37e39d88780000587358a",
            "date": "2016-08-15",
            "createdon": "2016-08-16T20:57:29.339Z",
            "movies": [
                "57b37c7b377dd100054f9f94",
                "57b37ba8377dd100054f9f92",
                "57b37b7c377dd100054f9f91"
            ]
        },
        ...
    ]
}
```

Listing 5.3: Example response for GET showtimes.ms.yannjaffrennou.com/showtimes

```
{
    "data": [
        {
            "id": "5ef38094a960b0000107bb86",
            "userid": "5ef36ebc8affb40001d156c3",
            "showtimeid": "5ef379751d8cb9000129a7d1",
            "movies": [
                "5ef36fae622bf60001fecd3c",
                "5ef36f28622bf60001fec9f5"
            ]
        },
        ...
    ]
}
```

Listing 5.4: Example response for GET bookings.ms.yannjaffrennou.com/bookings

## 5.2 General Google Cloud environments: Projects and accounts

To ensure a minimum interference between each deployment and the experimental platform, two separate Google Cloud accounts are used. One account is dedicated to the experimental platform, while the other one is dedicated to the experimental deployments. From the deployment account, one project is created for each deployment. As described in section 5.1, a MongoDB setup is required for both versions of the Cinema Application. To ensure a maximum equity between each deployment, this MongoDB setup is replicated in each project, allowing each deployment to rely on its own databases. This MongoDB setup requires a Cloud Firewall rule and a Google Compute Engine instance, and the parameters of these services are detailed in table 5.2.

| **Service** | **Parameter** | **Value** |
|---|---|---|
| VPC Network - Firewall | Target Tag(s) | mongodb |
| | Network | Default |
| | Source IP ranges | 0.0.0.0/0 |
| | TCP Port(s) | 27017 |
| GCE - VM Instance | Zone | us-east1-b |
| | Instance Type | n1-standard-1<br>(1 vCPU, 3.75 GB memory) |
| | Network Tag(s) | mongodb |
| | External IP address | None |
| | Internal IP address | 10.142.0.2 |
| | Boot Disk (OS) | Container-Optimized<br>OS 81-12871.181.0 |
| | Disk Size | 10GB |
| | Startup Script | docker start $(docker ps -aq)<br><br>docker exec $(docker ps -q)<br>/bin/bash /backup/restore.sh |

TABLE 5.2: MongoDB Setup Parameters

A first setup is required for each Google Cloud project to deploy MongoDB. This setup is made using a docker-compose file and SSH connections. All MongoDB instances of all projects (each relying on their own private network) use the same private (internal) IP address, permitting to use the same environment variables for the Cinema Application in all deployments. With this setup, MongoDB runs in a docker container inside of the VM instance and the default MongoDB port (27017) is exposed, as described in figure 5.1 and 5.2. Databases are backed up every time the VM instance restarts, and traffic

on the default MongoDB port is authorized from all networks. This setup is strictly equivalent in all projects, for all deployments. The MongoDB deployment does not auto-scale.

## 5.3 Infrastructure as a Service: Google Compute Engine & Load Balancer

### 5.3.1 General architecture

The general architecture of the Infrastructure as a Service deployment used is presented in figure 5.3.

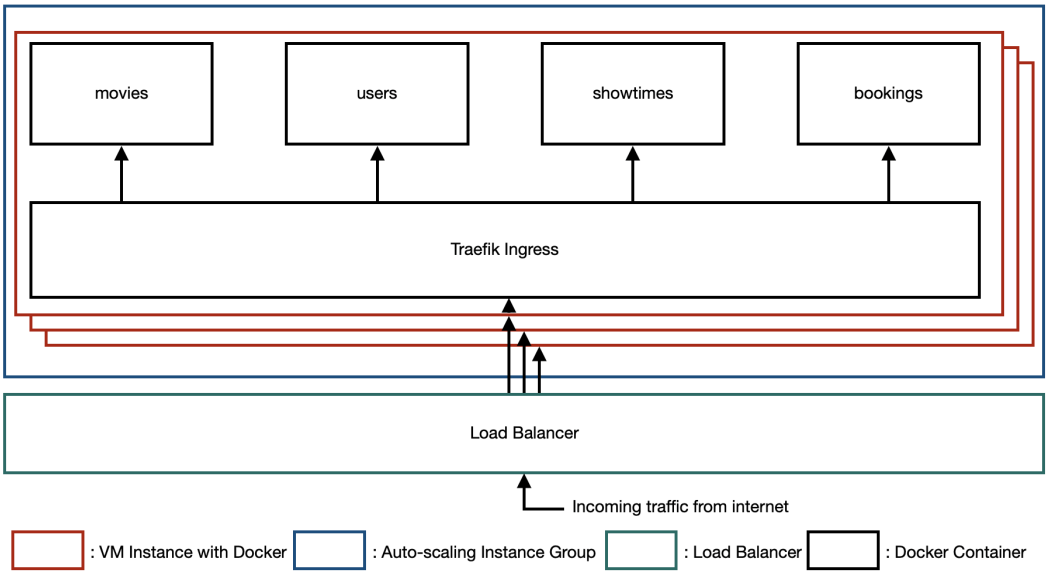



FIGURE 5.3: IaaS Architecture

Replication is managed at VM instance level, and each VM instance runs exactly one copy of each container. The IaaS deployment is the only one running the first version of the Cinema Application described in figure 5.1.

### 5.3.2 Resources used and setup process

The setup process to achieve this IaaS deployment is described in this section. All sensible parameters of the complete configuration of each service are described in table 5.3.

To create this complete architecture, a VM instance is firstly created using Google Compute Engine VM Instances service. The purpose of this instance in only to build a reusable system image that will be used to create new instances in auto-scaling context. For this matter, the first version of the Cinema Application is deployed to the image-builder VM instance. Once this setup is finished, all useless content is deleted (Cinema App code, SSH keys and useless docker images) to only keep one copy of each container. The image-builder is then stopped, and a system image is created from its disk using Google Compute Engine Images service.

Once the system image is ready to use, the image-builder VM instance is deleted and a Google Compute Engine Instance Template must be created to define the complete configuration that will be used for all instances of the auto-scaling deployment. This instance template includes the choice of the previously created system image as a boot disk, a startup script to start all Docker containers, and a network configuration with no external IP address. At this stage, no infrastructural costs are occurring (no service deployed, no running instance, no request to handle...). Dynamic costs will start to occur with the next steps, including the creation of VM instances as part of the production environment.

An auto-scaling Compute Engine Instance Group is created, with a targeted CPU utilization set to 60%. Finally, a Load Balancer is created from Google Network Services. A unique backend service is associated to this load balancer, directing all the traffic to the previously created instance group. A DNS record is added to the DNS Zone of yannjaffrennou.com to resolve all host names following *.ms.yannjaffrennou.com with the external IP address provided by the frontend http service of the lastly created load balancer.

| Service | Parameter | Value |
|---|---|---|
| GCE VM instance (Image Builder) | Zone | us-central1-a |
| | Instance type | n1-standard-1 (1 vCPU, 3.95GB memory) |
| | Boot Disk (OS) | Container-Optimized OS 81-12871.181.0 |
| | Disk size | 50GB |
| | Allow HTTP/HTTPS traffic | true |
| GCE Images | Name | cinemams-image |
| GCE Instance Template | Name | cinemams-template |
| | Instance type | n1-standard-1 (1 vCPU, 3.95GB memory) |
| | Boot Disk (OS) | cinemams-image |
| | Allow HTTP/HTTPS traffic | true |
| | Startup Script | docker start $(docker ps -aq) |
| | External IP | none |
| GCE Health Check | Name | cinemams-healthcheck |
| | Protocol | TCP |
| | Port | 80 |
| | Interval | 60 seconds |
| | Timeout | 5 seconds |
| | Healthy threshold | 2 consecutive successes |
| | Unhealthy threshold | 3 consecutive failures |
| GCE Instance Group | Name | cinemams-group |
| | Zone | us-central1-a |
| | Instance Template | cinemams-template |
| | Autoscaling mode | Autoscale |
| | Number of instances | Minimum 1, Maximum 6 |
| | Target CPU utilization | 60% |
| | Cool down period | 60 seconds |
| | Health check | cinemams-healthcheck |
| Load Balancing - Backend | Name | cinemams-backend |
| | Protocol | HTTP |
| | Timeout | 30 seconds |
| | Instance Group | cinemams-group |
| | Port Number | 80 |
| | Maximum backend utilization | 80% |
| | Capacity | 100% |
| | Health check | cinemams-healthcheck |
| Load Balancing - Frontend | Name | cinemams-frontend |
| | Protocol | HTTP |
| | IP address | IPv4 ephemeral |
| | Port | 80 |
| Load Balancing - HTTP(S) Load Balancing | Backend(s) | cinemams-backend |
| | Frontend | cinemams-frontend |
| | Host and path rules | default |

Table 5.3: IaaS complete setup parameters

## 5.4 Managed Kubernetes Service: Google Kubernetes Engine

### 5.4.1 General architecture

The general architecture of the Kubernetes cloud deployment used is represented in figure 5.4.

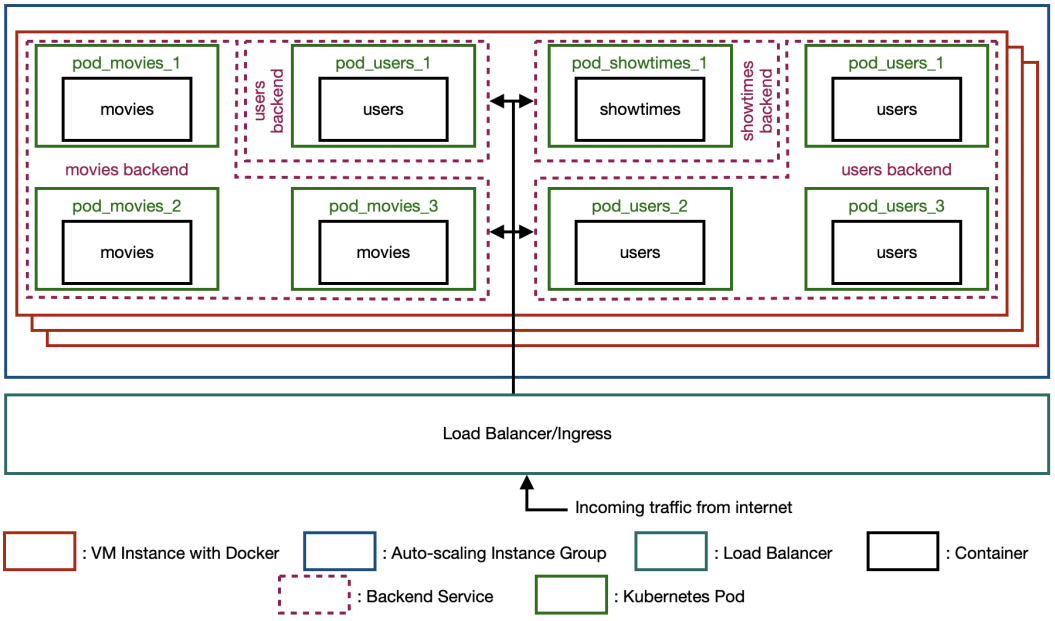



FIGURE 5.4: Kubernetes Architecture

The Kubernetes deployment is based on the second version of the Cinema Application. Each container is replicated independently through pods, which are then distributed over an auto-scaling instance group.

### 5.4.2 Resources used and setup process

The Kubernetes deployment uses more services than the IaaS deployment, and it is based on higher levels of management. The setup process to achieve this deployment is described in this section. All sensible parameters of the complete configuration of each service are described in table 5.4.

First, container images are created on a local machine from the docker-compose file of the second version of the Cinema Application. These docker images are then pushed

to Google Container Repository to be used in cloud computing contexts. In parallel, a Kubernetes Cluster is created using Google Kubernetes Engine (GKE). This cluster scales between 3 and 6 instances (to satisfy the requirement of a minimum of 3 pods replicated over distinct nodes for each service), and its nodes are using the same instance type and OS as the ones used in the IaaS deployment. The disk space is equivalent too, and set to 50GB. Once the Kubernetes cluster is ready, running costs start to occur and the 4 services of the Cinema Application can be deployed.

Each container is associated to one pod configuration called Workload in Kubernetes. Thus a total of 4 workloads are created. For each one of them, the appropriate container image is selected from the container registry. Once the 4 workloads are created and have associated pods running on some Kubernetes nodes, they are exposed to port 80 (with target port 8080 as described in figure 5.2). The service type parameter is set to Node Port to allow all pods associated to one workload to be exposed on a dedicated port of each node. This step leads to the creation of 4 services in GKE.

Finally, a Kubernetes Ingress is set up to route incoming traffic to appropriate pods and expose the Cinema Application publicly. 4 host rules are created for each one of the 4 services, depending on the host names defined in figure 5.2. A Load Balancer is automatically created with one backend per service, implying 4 health checks. The URLs targeted by these health checks must be modified to target the endpoints described in table 5.1, to ensure that a response with code 200 (success) will be sent instead of a 404 (not found).

| Service | Parameter | Value |
|---|---|---|
| GKE Cluster | Name | cinemams-kube |
| | Zone | us-central1-a |
| | Master version | 1.14.10-gke.36 |
| | Node version | 1.14.10-gke.36 |
| | Number of nodes | Minimum 3, Maximum 6 |
| | Node image type | Container-Optimized OS (cos) |
| | Machine type | n1-standard-1(1 vCPU, 3.95GB memory) |
| | Disk size | 50GB |
| GKE Workloads | Application name | cinemams-<serviceName> |
| | Container image | us.gcr.io/cinemams-kubernetes-282410/ cinema/<serviceName>:latest |
| | Cluster | cinemams-kube |
| | Service port | 80 |
| | Target port | 8080 |
| | Protocol | TCP |
| | Service type | Node port |
| | Service name | cinemams-<serviceName>-service |
| GKE Ingress | Ingress type | External HTTP/S load balancer |
| | Name | cinemams-ingress |
| | Backend services | cinemams-users-service<br>cinemams-movies-service<br>cinemams-showtimes-service<br>cinemams-bookings-service |
| | Host rules | movies.ms.yannjaffrennou.com ->cinemams-movies-service<br>users.ms.yannjaffrennou.com ->cinemams-users-service<br>showtimes.ms.yannjaffrennou.com ->cinemams-showtimes-service<br>bookings.ms.yannjaffrennou.com ->cinemams-bookings-service |
| | Frontend protocol | TCP |
| | Frontend port | 80 |
| | Frontend IP | IPv4, automatically allocated |
| GCE Health checks | Path | /<serviceName> |

Table 5.4: Kubernetes complete setup parameters

## 5.5 Serverless deployment: Google Cloud Run

### 5.5.1 General architecture

The general architecture of the Cloud Run deployment used is represented in figure 5.5.

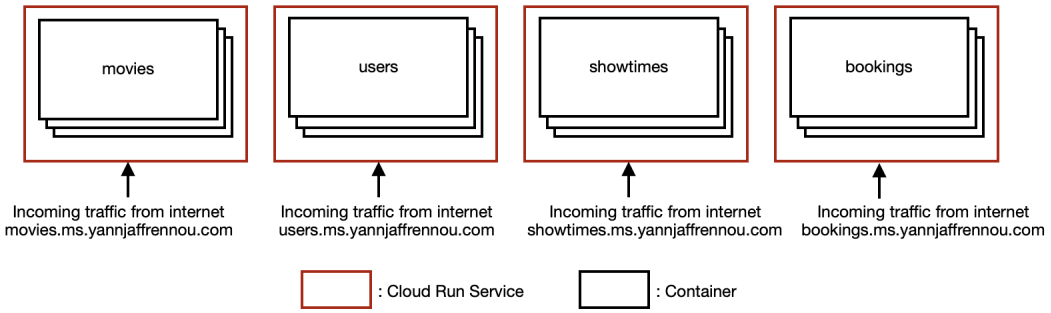



FIGURE 5.5: Cloud Run Architecture

The Cloud Run deployment is based on the second version of the Cinema Application, running at container level. Each container is replicated independently and uses its own resources (one vCPU cannot be shared across multiple containers). This architecture is the most simple from the user's point of view, as each container is deployed independently and traffic is routed at DNS Zone level.

### 5.5.2 Resources used and setup process

The Cloud Run deployment is the easiest to set up. All sensible parameters of the complete configuration of each service are described in table 5.5.

First, a network connector is required to enable communication between Cloud Run containers and the MongoDB service running in Google Compute Engine. This network connector is created using the Serverless VPC Access service. This connector is configured to allow traffic through the default network, from region us-central1, for all machines with an IP address within the IP range 10.8.0.0/28 (matching with the IP addresses allocated to Cloud Run instances running on the default cloud network). In parrallel, container images for each one of the 4 services of the Cinema Application have to be made available in the Container Registry of the Cloud Run project by pushing them from a local machine. These images are strictly identical to the ones used for

the Kubernetes deployment, created from the second version of the Cinema Application described in figure 5.2.

Once the network configuration is done and the container images are ready, each one of the 4 services of the Cinema Application can be deployed independently using their container images. Each service is set to target port 8080 of their running containers, and the lastly created VPC connector is added to their connections.

Finally, the last step required to make all services accessible from internet is the DNS setup. 4 mappings have to be added to the Domain Mapping, for each one of the 4 services. This step requires to add one CNAME record per service to the DNS Zone following the instructions provided by Google Cloud.

| **Service** | **Parameter** | **Value** |
|---|---|---|
| Serverless VPC Access | Name | cinemams-connector |
| | Region | us-central1 |
| | Network | default |
| | IP range | 10.8.0.0/28 |
| | Minimum throughput | 200Mbps |
| | Maximum throughput | 500Mbps |
| Cloud Run Services | Service Name | cinemams-<serviceName> |
| | Region | us-central1 |
| | Authentication | Allow unauthenticated |
| | Container image | us.gcr.io/cinemams-cloud-run/cinema/ <serviceName>:latest |
| | Container port | 8080 |
| | Memory allocated | 256MiB |
| | CPU allocated | 1 |
| | Number of instances | Minimum 0, Maximum 1000 |
| | VPC Connector | cinemams-connector |
| Cloud Run Domain Mappings | Mappings | movies.ms.yannjaffrennou.com ->cinemams-movies users.ms.yannjaffrennou.com ->cinemams-users showtimes.ms.yannjaffrennou.com ->cinemams-showtimes bookings.ms.yannjaffrennou.com ->cinemams-bookings |

Table 5.5: Cloud Run complete setup parameters

## 5.6 Experimental Platform

The experimental platform used for the experiments of this project is quite different from the one described in section 3.3. This new experimental platform, its architecture and behaviour are described in this section.

### Global architecture and services used

The experimental platform has been built from an existing service called Locust. Locust is an open source load testing tool that permits to define user behaviours before simulating multiple users interacting with the application to test. Locust is available as a docker image on Docker Hub, and it can be set up with a master/worker architecture using multiple docker containers (docker-compose file in appendix A). In this configuration, the users simulated are distributed over all workers. With Locust and Docker, the complete experimental platform is handled by one Google Compute Engine VM instance. It resides in the same region as all deployments (us-central1), and it is configured with 8 vCPUs and 30GB of memory (n1-standard-8 instance type) to make it ready to handle its high needs of resources. A simple docker-compose file is used to start Locust (with one master and 10 workers), and the web interface of Locust can be accessed through any browser as soon as these containers are all running.

### User behaviour

The user behaviour is described through python code. A set of candidate requests is defined along with a delay (that can be randomized), and every time a user receives a response to its lastly sent request, it waits for the delay before sending a randomly picked request. This cycle is represented in figure 5.6.

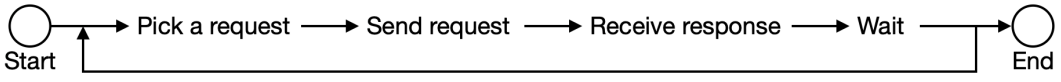



FIGURE 5.6: Locust user behaviour

### Swarming

Three parameters have to be defined before starting any new swarming session: the number of users to simulate, the user hatching and the URL to swarm. Thus the number of users simulated increases every second before reaching a maximum value. For instance, a session configured with 1000 users and a user hatching of 100 will take 10 seconds to reach the total number of users to simulate before stabilizing at its full load of 1000 users. A running session can be stopped at any time, by the simple press of a button. While a session is running, metrics on the requests and responses are updated in real time and saved in a table that can be downloaded as a CVS file. For every session and each endpoint targeted, these metrics include the number of requests sent, the average request frequency, the number of success and failures, statistics on response time, inter alia.

# Chapter 6

# Experiments and results

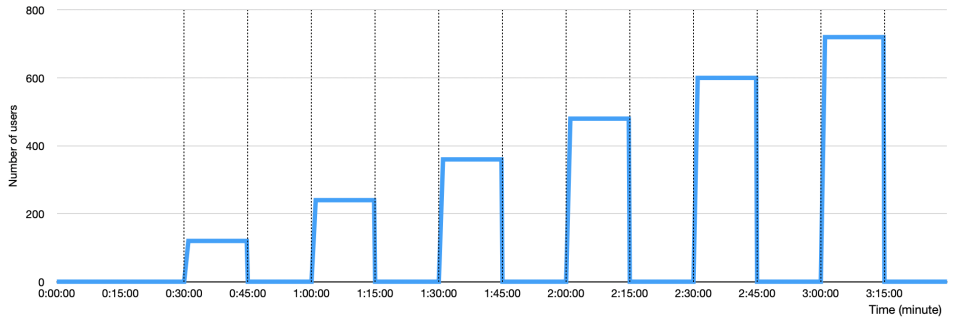
## 6.1 Experimental protocol

To experimentally measure the performance and cost of each deployment throughout different utilization schemes, one experiment is lead for each deployment. The only variable in this set of experiments is the cloud computing infrastructure used, thus permitting to compare these deployments in similar stressing conditions determined through preliminary experiments on the Cinema Application. Consequently, the experimental protocol must be identical for all experiment, and this protocol is described in this section.

The experiments lead are composed of 13 phases defined over time, for a total of 7 idle periods and 6 swarming sessions. Swarming sessions are separated by a 15 minutes idle period, and include a 1 minute period of scale out and 14 minutes of full load, no matter the number of users simulated (the higher the number of users is, the higher the user hatching is). The number of users simulated over time is described in figure 6.1.

For each deployment, the total duration of the experiment is 3 hours and 30 minutes. The first 30 minutes of idle period permit to set up the deployment and send a test request before starting swarming. Once the experiment is finished, the infrastructure is shutdown to avoid any extra cost.

To simplify the requests sent to the Cinema Application and minimize the need of high computational resources of the MongoDB instance, only the "GET all" endpoints described in table 5.1 are triggered. Thus no write lock have to be managed by the




| Phase | Unit | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of users | user | 0 | 120 | 0 | 240 | 0 | 360 | 0 | 480 | 0 | 600 | 0 | 720 | 0 |
| User hatching | usr/sec | / | 2 | / | 4 | / | 6 | / | 8 | / | 10 | / | 12 | / |
| Duration | min | 30 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| Scale out duration | min | / | 1 | / | 1 | / | 1 | / | 1 | / | 1 | / | 1 | / |

FIGURE 6.1: Users simulated over time

MongoDB service, and no consistency check have to occur. The timeout before each request is set to a fixed value of 1 second. This user behaviour is presented accordingly to the experimental platform description (section 5.6) in the python listing 6.1.

```
import random
import json
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    ids = {}
    wait_time = between(1, 1)

    @task
    def get_all_users(self):
        self.client.get("http://users.ms.yannjaffrennou.com/users")

    @task
    def get_all_movies(self):
        self.client.get("http://movies.ms.yannjaffrennou.com/movies")

    @task
    def get_all_showtimes(self):
        self.client.get("http://showtimes.ms.yannjaffrennou.com/showtimes")

    @task
    def get_all_bookings(self):
        self.client.get("http://bookings.ms.yannjaffrennou.com/bookings")
```

LISTING 6.1: User behaviour: locustfile.py

## 6.2 Results

### Performance

A summary of the performance delivered by the Cinema Application in each deployment configuration is presented in table 6.1.

| Measure | Users | Request Count | Failure Count | Median Response Time | Average Response Time | Requests/s |
|---|---|---|---|---|---|---|
| **Unit** | **user** | **request** | **failure** | **ms** | **ms** | **req/s** |
| IaaS | 120,00 | 66106,00 | 0,00 | 89,00 | 103,94 | 72,42 |
| | 240,00 | 128432,00 | 0,00 | 94,00 | 145,99 | 141,37 |
| | 360,00 | 185286,00 | 0,00 | 94,00 | 198,60 | 205,03 |
| | 480,00 | 242516,00 | 0,00 | 95,00 | 234,74 | 268,17 |
| | 600,00 | 285942,00 | 0,00 | 98,00 | 327,28 | 317,22 |
| | 720,00 | 321435,00 | 1,00 | 110,00 | 464,24 | 354,83 |
| | **SUMMARY** | **1229717,00** | **1,00** | / | **294,50** | **226,51** |
| Kubernetes | 120,00 | 65184,00 | 0,00 | 92,00 | 110,58 | 72,15 |
| | 240,00 | 128136,00 | 0,00 | 110,00 | 138,58 | 141,77 |
| | 360,00 | 135125,00 | 1,00 | 150,00 | 819,91 | 150,08 |
| | 480,00 | 119152,00 | 1,00 | 160,00 | 2023,59 | 131,27 |
| | 600,00 | 163239,00 | 6,00 | 120,00 | 1684,05 | 181,59 |
| | 720,00 | 145564,00 | 765,00 | 140,00 | 2802,94 | 160,80 |
| | **SUMMARY** | **756400,00** | **773,00** | / | **1401,08** | **139,61** |
| CloudRun | 120,00 | 62858,00 | 0,00 | 170,00 | 175,08 | 69,41 |
| | 240,00 | 124977,00 | 0,00 | 180,00 | 187,38 | 137,77 |
| | 360,00 | 187307,00 | 0,00 | 170,00 | 186,32 | 206,84 |
| | 480,00 | 246935,00 | 9,00 | 180,00 | 198,28 | 273,21 |
| | 600,00 | 308375,00 | 40,00 | 170,00 | 197,55 | 341,49 |
| | **720,00** | **345308,00** | **0,00** | **180,00** | **281,78** | **389,22** |
| | **SUMMARY** | **1275760,00** | **49,00** | / | **216,74** | **236,32** |

TABLE 6.1: Summary of the Cinema Application performance for all swarming sessions

As described in table 6.1, the Cloud Run configuration can be considered as best performing by responding to the highest number of requests, closely followed by the IaaS configuration. The Kubernetes configuration performed less, and presented the highest number of failures of all configurations. Despite a very small median response time for the IaaS configuration, the Cloud Run deployment presents the lowest average response time and Kubernetes is, here again, less performant.

### Cost

In terms of cost, and according to the results presented in table 6.2, the IaaS configuration was the less expensive to run, followed by Kubernetes, and Cloud Run. For all configurations, the highest cost factor corresponds to the actual computing resources (CPU), representing more than half of the cost of each infrastructure. The rest of these costs are distributed between memory utilization and load balancing. For the Cloud

Run deployment, the N1 instance cost is only due to the MongoDB instance running in Google Compute Engine.

| **Deployment** | **IaaS** | **Kubernetes** | **Cloud Run** |
|---|---|---|---|
| **Unit** | **euro** | **euro** | **euro** |
| N1 Predefined Instance Core running in Americas | 0,53 | 0,41 | 0,10 |
| N1 Predefined Instance Ram running in Americas | 0,27 | 0,20 | 0,05 |
| HTTP Load Balancing: Global Forwarding Rule Minimum Service Charge | 0,07 | 0,07 | / |
| CPU Allocation Time | / | / | 1,15 |
| Memory Allocation Time | / | / | 0,03 |
| **TOTAL** | **0,87** | **0,68** | **1,33** |

TABLE 6.2: Cost breakdown for all deployments

### Scalability and Resource utilization

As the main cost factor and scalability parameter, the CPU available and used by the Cinema Application is a key element to determine if a deployment is able to scale properly. For the IaaS and Kubernetes deployment, this CPU availability is represented by the number of Compute Engine instances running, whereas the CPU available to the Cloud Run configuration is observable through dynamic CPU allocation (expressed in seconds of CPU core availability per second of time elapsed).

For the IaaS deployment, the CPU capacity and CPU utilization are represented over time in figure 6.2. With the targeted CPU utilization of 60% described in table 5.3, one running instance (with one vCPU) represents 60% of CPU availability, two instances are 120%, and so on. Thus this CPU availability metric is always a multiple of 60%.


FIGURE 6.2: IaaS Autoscaler Utilization

Concerning the Kubernetes deployment, the CPU utilization monitoring does not cover auto-scaling clusters. Consequently, it is expressed as a percentage of the total CPU availability of the cluster. In other words, a 80% CPU utilization can represent less CPU consumption than a 60% CPU utilization if the Kubernetes cluster has scaled up between these two measures, increasing the total CPU availability. Thus the CPU

utilization has to be considered along with the number of running instances of the Kubernetes cluster, respectively represented in figure 6.3 and 6.4.


FIGURE 6.3: Kubernetes CPU utilization


FIGURE 6.4: Kubernetes cluster size

Finally, the Cloud Run scalability can be represented directly through the CPU allocation, corresponding to the number of seconds of one vCPU allocated to a service each second of time elapsed. In other words, if 4 vCPUs are allocated to a service at a certain time, this metric will be of 4 seconds per second at this precise time. As the Cloud Run service manages each service independently, this metric is expressed per service. The Cloud Run CPU allocation for all 4 services over time is presented in figure 6.5.

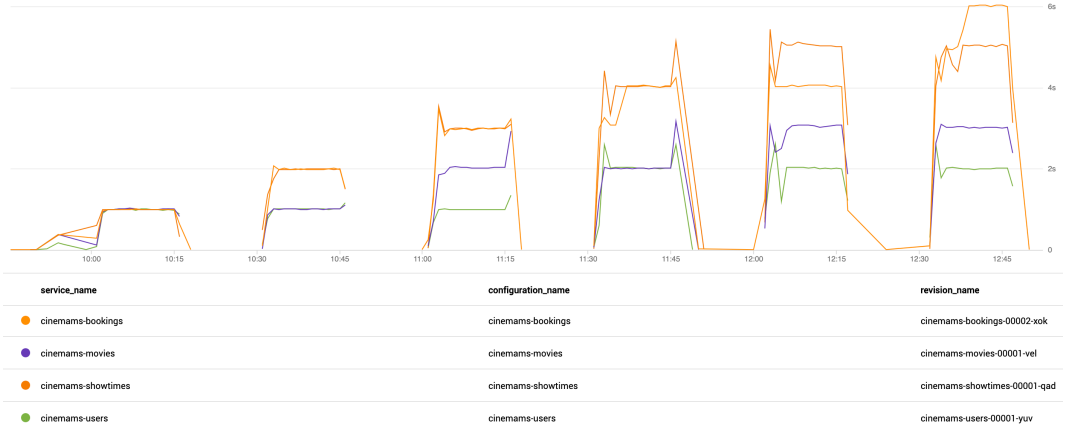



FIGURE 6.5: Cloud Run CPU allocation

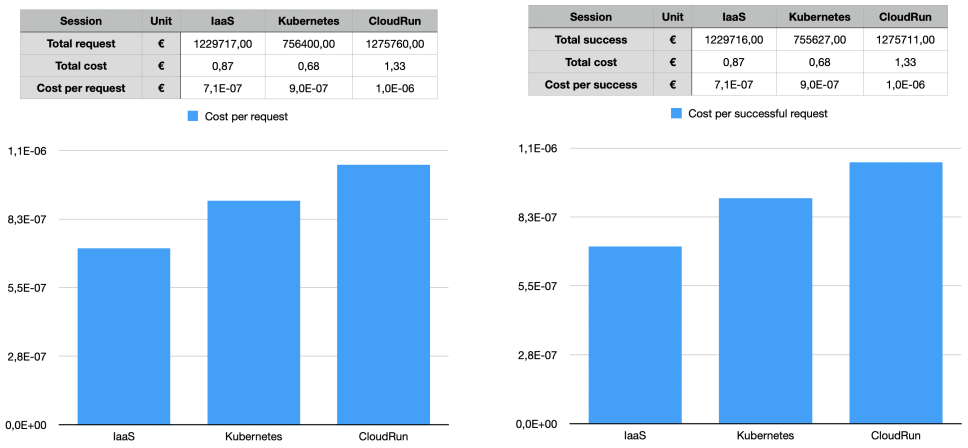
As Google Kubernetes Engine relies on the same technology of auto-scaling instance group as the IaaS deployment, similar scaling behaviour are observable: it takes around 13 minutes before the instance group starts to shutdown one instance when the CPU

capacity is higher than the CPU utilization. This results in unnecessary costs for unused VM instances. On the other side, Cloud Run allocates computing resources with a significantly higher precision, avoiding any extra cost. Moreover the difference of targeted CPU utilization between IaaS and Kubernetes (respectively 60% and 80%) implies a lower number of running instances for the Kubernetes configuration in comparison with the IaaS configuration: the IaaS instance group peaked at 6 instances whereas the Kubernetes instance group peaked at 4 instances. The Cloud Run CPU allocation went up to a total of 16 vCPUs, way higher than the IaaS and Kubernetes configurations.

## 6.3 Findings

### Cost per performance

The cost per performance of each cloud computing platform compared in this study can be expressed as a cost per request, or cost per successful request. As the number of failures is overall pretty low (the highest being 0,1% for Kubernetes), these two means of calculation present pretty similar results, presented in figure 6.6.




(A) Cost per request (B) Cost per successful request

FIGURE 6.6: Cost per performance

According to these results, the price per request of each deployment matches its level of service. In other word, the cloud hosting providers' clients can benefit from extra services allowing easiest deployment or maintenance at a cost. A Simple Infrastructure as a Service presents the lowest cost per request, and further analyses of the performances

of each system must be made in consideration of the scale and scalability performance of these systems.

### Scalability and performance at scale

The ability to auto-scale of each infrastructure can be expressed as its ability to deliver the same performances for different number of users simulated at different time. In other words, the response time observed for each request should be invariable, and the number of requests handled per second should be proportionate to the number of users simulated. The number of requests handled by each infrastructure for each swarming session is presented in figure 6.7. According to this graph, the IaaS and Cloud Run deployments handled an increasing number of requests while the number of users was increasing, proving a good ability to scale. On the other side, Kubernetes was able to scale only from 120 to 240 users, and its performance started decreasing from 360 users.

| Session | Number of users | IaaS | Kubernetes | CloudRun |
|---|---|---|---|---|
| 1 | 120 | 72,4 | 72,2 | 69,4 |
| 2 | 240 | 141,4 | 141,8 | 137,8 |
| 3 | 360 | 205,0 | 150,1 | 206,8 |
| 4 | 480 | 268,2 | 131,3 | 273,2 |
| 5 | 600 | 317,2 | 181,6 | 341,5 |
| 6 | 720 | 354,8 | 160,8 | 389,2 |

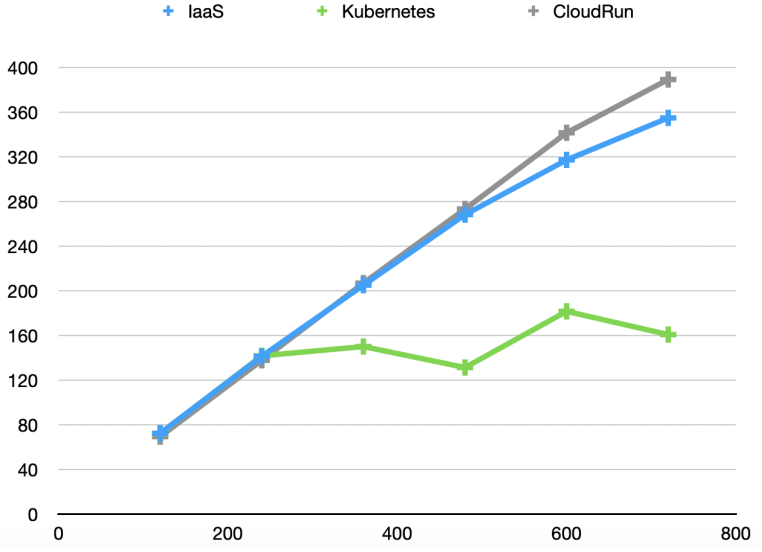


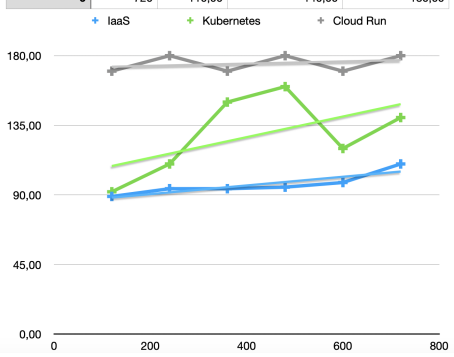

FIGURE 6.7: Number of requests handled per second

The median and average response time of each configuration and for each swarming session are presented in figure 6.8. These graphs show, once again, that the IaaS and

Cloud Run were able to scale very well and maintain a constant response time while the load was increasing. On the other side, Kubernetes presents surprising response time variations that can be explained by a poor ability to get access to and use a correct amount of computing resources.

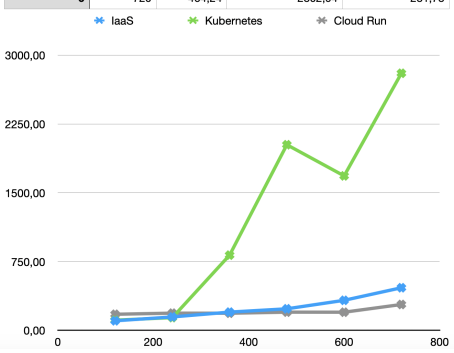
Aside from scalability, the median response time of each deployment seems to be proportionate to the level of service it relies on. Indeed, a higher level of service implies a higher number of layers for each request to go through before arriving to a computing instance. This phenomenon does not benefit to the Cloud Run service, that presents the best scalability but the highest median response time.

| Session | Users | IaaS | Kubernetes | Cloud Run |
|---|---|---|---|---|
| 1 | 120 | 89,00 | 92,00 | 170,00 |
| 2 | 240 | 94,00 | 110,00 | 180,00 |
| 3 | 360 | 94,00 | 150,00 | 170,00 |
| 4 | 480 | 95,00 | 160,00 | 180,00 |
| 5 | 600 | 98,00 | 120,00 | 170,00 |
| 6 | 720 | 110,00 | 140,00 | 180,00 |




(A) Median response time

| Session | Users | IaaS | Kubernetes | Cloud Run |
|---|---|---|---|---|
| 1 | 120 | 103,94 | 110,58 | 175,08 |
| 2 | 240 | 145,99 | 138,58 | 187,38 |
| 3 | 360 | 198,60 | 819,91 | 186,32 |
| 4 | 480 | 234,74 | 2023,59 | 198,28 |
| 5 | 600 | 327,28 | 1684,05 | 197,55 |
| 6 | 720 | 464,24 | 2802,94 | 281,78 |




(B) Average response time

FIGURE 6.8: Average response time

To summarize, the IaaS configuration delivered the best performance having the smallest response time and a very good ability to scale, despite a slightly lower number of requests handled per second than the Cloud Run deployment at certain scale. The Cloud Run infrastructure showed the best ability to scale with perfectly constant median and average response times, but its response time is higher than the two other configurations. Finally, Google's managed Kubernetes Service presented a very bad ability to scale, resulting in bad overall performance. It was however able to deliver competitive performances at small scale, relying on a correctly adjusted amount of computing resources.

### Summary: Strengths and weaknesses

A summary of the results observed for each 4 deployments in terms of ability to scale, fixed scale performance and cost is presented in table 6.3 (ranking, lower is better).

| | **IaaS** | **Kubernetes** | **Cloud Run** |
|---|---|---|---|
| **Performance at fixed scale** | 1 | 2 | 3 |
| **Ability to scale** | 2 | 3 | 1 |
| **Cost** | 1 | 2 | 3 |

TABLE 6.3: Ranking of IaaS, Kubernetes Engine and Cloud Run

The results observed for Kubernetes Engine are the most surprising, with limited performances and a poor ability to scale, despite its numerous benefits aside from pure performance (continuous integration, continuous deployment, flexibility...). It should preferably not be used in a fast-scaling context, and the size of the cluster should ideally be fixed. Cloud Run is perfectly adapted to fast-scaling context, but may imply a trade off on response time and increased costs. Finally, a simple Infrastructure as a Service is the best value for money, with the best performances, overall good scalability and low cost.

# Chapter 7

# Conclusion

## 7.1 Conclusion

This project permitted to reveal some key characteristics of some of Google's auto-scaling cloud computing technologies: Google Kubernetes Engine, Google Cloud Run and Google IaaS with more classic auto-scaling instance groups and Docker. Each one of these technologies presents its own set of strengths and weaknesses, and a special attention should be paid to the choice of a production environment for deploying a micro-services application in the cloud. Along with considerations on the flexibility and level of service proposed by each technology, an estimation of the requirements of the application to be deployed must be made in terms of performance, scalability and cost.

Containerised applications are very easy to deploy using all of these technologies, and architectural decisions should be taken from the very beginning of the development phase to ensure a good transition to auto-scaling production environments.

From big changes in the application tested, repercussions on the financial coverage of the experiments, and structural changes on the experimental platform, this project had to strongly evolve. Most of its Must requirements were not met, and a Won't requirement was satisfied. According to the requirements described in section 3, the following were met:

- Retrieve data from the cloud hosting services consoles

- Analyse the collected data to propose conclusions on the best deployment solution to choose
- Test any other type of deployment

However, thanks to the flexibility of the project and its general aim, the following objectives were fully reached:

- Gather knowledge on new trends of cloud computing and scalable web applications
- Evaluate the scalability performances of two cloud computing services
- Experimentally compare the running costs of applications deployed using Infrastructure as a Service and Container as a Service (see section 2.1.3.3) on Google Cloud Platform (see section 2.1.3)

The findings and conclusions of this project must be considered within its context, the application used for its experiments, and the specific load simulated, although some conclusions could be extrapolated to other applications and contexts with further studies.

## 7.2 Future work

From the experiments of this project, it appeared that Kubernetes' ability to deliver good performance in auto-scaling context was not good compared to concurrent deployments. Further studies could be led to study the influence of Kubernetes Engine parameters (such as targeted CPU utilization) on its performance at scale.

The same experiment could also be led on other auto-scaling services proposed by Google (such as App Engine), or by other cloud services providers (Amazon EC2, beanstalk...) to extend the observations of this project to a broader ensemble.

Finally, some experiments on micro-services applications deployed in auto-scaling cloud computing infrastructures could be led at a much bigger scale through A/B testings on a real-life application delivered to its users.

# Appendix A

# Locust - Docker

```
version: '3'

services:
  master:
    image: locustio/locust
    ports:
     - "80:8089"
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py --master -H http://master:8089

  worker:
    image: locustio/locust
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py --worker --master-host master
```

Listing A.1: docker-compose.yml

# Appendix B

# MongoDB - Docker

```
version: '3.7'

services:
  db:
    image: mongo:3.3
    ports:
      - target: 27017
        published: 27017
        protocol: tcp
        mode: host
    volumes:
      - ./backup:/backup
```

Listing B.1: docker-compose.yml

# Appendix C

# Cinema Application V1 - Docker

```
version: '3.7'
services:
  proxy:
    image: traefik:1.7.4-alpine
    command:
      - "--api"
      - "--docker"
      - "--docker.watch"
    labels:
      - "traefik.frontend.rule=Host:monitor.ms.yannjaffrennou.com"
      - "traefik.port=8080"
    volumes:
      - type: bind
        source: /var/run/docker.sock
        target: /var/run/docker.sock
    ports:
      - target: 80
        published: 80
        protocol: tcp
        mode: host

  movies:
    build: ./movies
    image: cinema/movies
    labels:
      - "traefik.backend=movies"
      - "traefik.frontend.rule=Host:movies.ms.yannjaffrennou.com"

  bookings:
    build: ./bookings
    image: cinema/bookings
    labels:
```

```
      - "traefik.backend=bookings"
      - "traefik.frontend.rule=Host:bookings.ms.yannjaffrennou.com"

  showtimes:
    build: ./showtimes
    image: cinema/showtimes
    labels:
      - "traefik.backend=showtimes"
      - "traefik.frontend.rule=Host:showtimes.ms.yannjaffrennou.com"

  users:
    build: ./users
    image: cinema/users
    labels:
      - "traefik.backend=users"
      - "traefik.frontend.rule=Host:users.ms.yannjaffrennou.com"
```

Listing C.1: docker-compose.yml

# Appendix D

# Cinema Application V2 - Docker

```
version: '3.7'
services:
  movies:
    build: ./movies
    image: cinema/movies
    ports:
      - 8080:8080

  bookings:
    build: ./bookings
    image: cinema/bookings
    ports:
      - 8080:8080


  showtimes:
    build: ./showtimes
    image: cinema/showtimes
    ports:
      - 8080:8080

  users:
    build: ./users
    image: cinema/users
    ports:
      - 8080:8080
```

Listing D.1: docker-compose.yml

# Bibliography


Aaqib, S. M. (2019). An efficient cluster-based approach for evaluating vertical and horizontal scalability of web servers using linear and non-linear workloads. pages 287–291.

Al-Debagy, O. and Martinek, P. (2018). A comparative review of microservices and monolithic architectures. pages 000149–000154.

Amazon (2020a). Aws & sustainability. Url: https://aws.amazon.com/about-aws/sustainability/.

Amazon (2020b). Aws free tier. Url: https://aws.amazon.com/free/.

Amazon (2020c). Aws free tier faqs. Url: https://aws.amazon.com/free/free-tier-faqs/.

Barzu, A., Barbulescu, M., and Carabas, M. (2017a). Horizontal scalability towards server performance improvement. pages 1–6.

Barzu, A., Carabas, M., and Tapus, N. (2017b). Scalability of a web server: How does vertical scalability improve the performance of a server? pages 115–122.

BusinessDictionary (2020). Businessdictionary. Url: http://www.businessdictionary.com/definition/application-software.html.

Cercas Curry, A., Papaioannou, I., Suglia, A., Agarwal, S., Shalyminov, I., Xinnuo, X., Dusek, O., Eshghi, A., Konstas, I., Rieser, V., and Lemon, O. (2018). Alana v2: Entertaining and informative open-domain social dialogue using ontologies and entity linking.

Chen, R., Li, S., and Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. pages 466–475.

Christensson, P. (2014). Web application definition. Url: https://techterms.com/definition/web_application.

Christensson, P. (2017). Web service definition. Url: https://techterms.com/definition/web_service.

Cook, G., Lee, J., Tsai, T., Kong, A., Deans, J., Johnson, B., and Jardim, E. (2017). Clicking clean: who is winning the race to build a green internet?

De Lauretis, L. (2019). From monolithic architecture to microservices architecture. pages 93–96.

Dewi, L. P., Noertjahyana, A., Palit, H. N., and Yedutun, K. (2019). Server scalability using kubernetes. pages 1–4.

European Commission (2018). General data protection regulation. Url: https://gdpr-info.eu.

Google (2020a). Google cloud free tier. *Journal*, 2020(2020-04-04). Url: https://cloud.google.com/free/docs/gcp-free-tier.

Google (2020b). Project suspension guidelines. Url: https://cloud.google.com/resource-manager/docs/project-suspension-guidelines.

Google (2020c). Supplemental terms and conditions for google cloud platform free trial. Url: https://cloud.google.com/terms/free-trial/.

Huang, Y., Yang, Y., Rossi, M., and Xu, B. (2012). Towards a unified architecture of cloud service delivery platform. 01:360–364.

Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. pages 342–346.

Ko, R. K. L., Tan, A. Y. S., and Ng, G. P. Y. (2014). 'time' for cloud? design and implementation of a time-based cloud resource management system. pages 530–537.

Lehrig, S., Eikerling, H., and Becker, S. (2015). Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. pages 83–92.

Microsoft (2020a). Azure free account faq. Url: https://azure.microsoft.com/en-us/free/free-account-faq/.

Microsoft (2020b). Create your azure free account today. Url: https://azure.microsoft.com/en-us/free/.

Moldovan, D., Truong, H., and Dustdar, S. (2016). Cost-aware scalability of applications in public clouds. pages 79–88.

Morejon, M. (2020). microservices-docker-go-mongodb. Url: https://github.com/mmorejon/microservices-docker-go-mongodb Commit: 5c371d5.

Murugesan, S. (2012). Cloud computing: A new paradigm in it that has the power to transform emerging markets. *International Journal on Advances in ICT for Emerging Regions (ICTer)*, 4.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.*

Nunnikhoven, M. (2016). Defending the whole, iaas, paas, and saas. Url: https://www.slideshare.net/marknca/defending-the-whole-iaas-paas-and-saas.

Pereira Ferreira, A. and Sinnott, R. (2019). A performance evaluation of containers running on managed kubernetes services. pages 199–208.

Prajapati, A. G., Sharma, S. J., and Badgujar, V. S. (2018). All about cloud: A systematic survey. pages 1–6.

Preeth E N, Mulerickal, F. J. P., Paul, B., and Sastri, Y. (2015). Evaluation of docker containers based on hardware utilization. pages 697–700.

Smith, B. (2019). We're increasing our carbon fee as we double down on sustainability. Url: https://blogs.microsoft.com/on-the-issues/2019/04/15/were-increasing-our-carbon-fee-as-we-double-down-on-sustainability/.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32.

Tanni, T. I. and Hasan, M. S. (2017). A performance analysis of a typical server running on a cloud. pages 1–6.

Villamizar, M., Garcés, O., Castro, H., Verano Merino, M., Salamanca, L., Casallas, R., and Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., and Lang, M. (2016). Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. pages 179–182.

Wahid, A. and Banday, M. T. (2018). Machine type comparative of leading cloud players based on performance pricing. pages 2364–2368.