ESIEA

Migration of a cloud-based microservice platform to a container solution

Migration d'une plateforme de microservices dans le cloud vers une solution conteneurisée

by

Pablo Piedrafita Castañeda

Aymeric Barantal Renaud Perrier Nadjim Benali

Master's thesis
Mémoire de fin d'études

Internship done at
GANDI SAS

October, 4th 2018

"Ninety percent of everything is crud."

Theodore Sturgeon

Remerciements

J'adresse mes remerciements aux personnes qui m'ont aidé durant mon stage et lors de la réalisation de ce mémoire, mais aussi aux personnes qui m'ont accueilli à l'ESIEA pendant cette année.

Tout d'abord, j'adresse mes remercimients à mon maître de stage, Aymeric Barantal, qui m'a offert cette opportunité et m'a guidé tout au long du stage, avec une énorme patience face à mon impetuosité.

Merci à l'équipe Caliopen, pour avoir fait du stage une expérience amène, énormémment amusante et très instructive. Merci à Gandi aussi pour avoir rendu possible ce stage.

Je tiens à remercier aussi mon tuteur pédagogique, Renaud Perrier, pour son exquise vision analytique qui m'a montré à prendre du récul dans la rédaction du mémoire.

Merci à l'ESIEA pour m'avoir acueilli durant mon Erasmus et avoir rendu possible cette expérience. À mes camarades de classe je remercie avoir fait de mon passage à l'école un excellent séjour.

Enfin, je tiens à remercier MC pour toutes ses corrections et conseils malgré son manque de connaissances sur le sujet.

Contents

List of Figures

List of Tables

Abbreviations

IaaS	**I**nfrastructure **A**s **A** **S**ervice
PaaS	**P**latform **A**s **A** **S**ervice
FaaS	**F**unctions **A**s **A** **S**ervice
CaaS	**C**ontainers **A**s **A** **S**ervice
KaaS	**K**ubernetes **A**s **A** **S**ervice
SaaS	**S**oftware **A**s **A** **S**ervice
PI	**P**rivacy **I**ndex
CI	**C**ontinous **I**ntegration
CD	**C**ontinous **D**elivery / **D**eployment
SPOF	**S**ingle **P**oint **O**f **F**ailure
IMAP	**I**nternet **M**essage **A**ccess **P**rotocol
LAN	**L**ocal **A**rea **N**etwork
VLAN	**V**irtual **L**ocal **A**rea **N**etwork
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
SDLC	**S**oftware **D**evelopment **L**ife **C**ycle
IT	**I**nformation **T**echnology
OS	**O**perating **S**ystem
ELK	**E**lasticsearch **L**ogstash **K**ibana
DNS	**D**omain **N**ame **S**ystem
POC	**P**roof **O**f **C**oncept
TLS	**T**ransport **L**ayer **S**ecurity
VPS	**V**irtual **P**rivate **S**erver

Résumé analytique

Caliopen est la plateforme de mail Open Source qui veut récupérer la confidentialité dans les communications de ses utilisateurs. Réincarné en 2013 par Laurent Chemla, le projet se situe comme une alternative aux services de mail gratuits comme Gmail ou Outlook, qui ont oublié l'importance de la privacité dans les communications personnelles. Le projet est ambitieux, il cherche à atteindre le grand publique avec deux fonctionalités principales: leur permettre d'agréger tous leurs moyens de communication numériques sous une seule interface et leur aider à améliorer la confidentialité dans leurs échanges grâce à la gamification. Cependant, il ne se limite pas au publique. Caliopen cherche à offrir une solution capable et innovante aux grandes entreprises aussi, à travers des solutions privées ou SaaS.

Le pilier principal du projet est l'aspect social. Toutefois, Caliopen s'appuie sur une base extrêmement technique pour arriver à offrir à ses utilisateurs un outil à la fois apte et simple à utiliser. Caliopen se trouve depuis fin 2017 en phase alpha, avec l'idée de passer en beta en octobre 2018. À fin d'améliorer la plateforme avant le lancement de la beta, certaines améliorations ont été prévues pour celle-ci. Ce travail se situe dans le contexte de ces aménagements, qui ont pour objectif de créer une plateforme résiliente, autonome et à la pointe, pour simplifier sa gestion et augmenter la productivité de l'équipe. La technologie principale derrière le nouveau produit est la conteneurisation, grace à laquelle le projet pourra intégrer d'autres concepts comme la livraison continue (continuous delivery en Anglais).

Ce travail introduit tout d'abord les problèmes auxquels on fait face dans la plateforme Caliopen actuelle, principalement liés au provisionnement et la gestion du cluster de machines, en ce moment en production. Il se trouve aussi que la plateforme est très hétérogène, on retrouve des environnements de développement, de test et production non unifiés et technologiquement disperses. On présente une solution basée sur la connue plateforme d'orchestration de conteneurs Kubernetes avec l'idée de créer un appui sur lequel la plateforme pourra continuer à évoluer. De plus, on incorpore de nouveaux outils qui serviront comme prémisse pour améliorer le processus de mise en production et de publication de nouvelles versions. Ces outils ont pour objectif d'automatiser une grande partie des processus de tests et de déploiement. Kubernetes permet d'unifier les environnements de tests et de production et de les rapprocher considérablement de l'environnement de developpement. De plus, il a permis de créer une architecture sécurisée.

Il existent toujours des axes d'amélioration dans la solution présentée. La migration des services de l'ancienne plateforme n'est pas complète, notamment la partie de stockage qui présente de nouveaux challenges à résoudre. L'automatisation des processus est d'autre part aussi assez limitée et requiert de nouvelles études et améliorations qui devront être intégrées par l'équipe Caliopen.

Executive summary

Caliopen is an Open Source mail platform that aims to recover the privacy in the communication of its users. Reborn in 2013 by Laurent Chemla, the project intends to become an alternative to free mail services such as Gmail or Outlook, that have long forgotten about the importance of confidentiality in personal communication. The project has two axes of commercialization. The first one focuses on the general public, offering them the possibility of aggregating their means of communication. The second one is oriented to big businesses, through private or SaaS based solutions.

Even though the project's main objective is social, a strong technical backbone is needed to provide its users with a capable and easy to use application. Caliopen is currently in alpha, with the expectation to go into a beta phase in October 2018. With the intention of improving the platform before the launch of the beta, some enhancement have been planned. This work takes place within the context of these modifications, that intend to create a resilient, autonomous and state-of-the-art platform. The main concepts behind the new product are containerization and continuous delivery.

This work first studies the problems the current platform faces, mostly related to cluster management, and deployment environments and tries to solve them with a Kubernetes-based solution. The new cluster will serve as a base to build upon and improve the delivery process of the project, bringing automation to multiple aspects of the development. There are, nevertheless, many axes of improvement for the solution deployed. For instance, many services in the platform are still not running on Kubernetes, notably storage, that presents new unresolved challenges. Automation remains limited, requiring improvements from Caliopen's team.

Chapter 1

Introduction

1.1 Gandi SAS

1.1.1 Introducing the company

GANDI SAS (Gestion et Attribution des Noms de Domaine sur Internet in French, or Management and Allocation of Domain Names on the Internet) was founded in 1999 by Pierre Beyssac, Laurent Chemla, and Valentin Lacambre. In 2005, Gandi was bought by an European management team within the same field, in order to create an alternative and independent line of Internet services based around domain names. Gandi has offices in Paris (France), San Francisco (USA), Bissen (Luxembourg) and Taipei (Taiwan).

Gandi provides domain name registration, web hosting, and VPS cloud hosting. As of May 2015, it manages around 2,000,000 domain names from 192 countries, which places them as first among domain name registrars in France, sixth in Europe, and in the top fifteen worldwide.

Gandi uses and advocates for open-source software. The company has a program to support financially, technically, administratively, or morally, projects and organizations that meet their criteria of being concrete, open and an alternative to a dominant mass commercial supplier.

1.1.2 Gandi & Caliopen

Caliopen meets the criteria to be supported financially, technically and administratively by Gandi. It's a project almost in beta, completely open-source and a privacy-oriented alternative to dominant mail services such as Gmail. The BPI is the main financier of the project, in partnership with Gandi. Gandi provides Caliopen with a physical emplacement in their central headquarters in Paris, workforce (the core team is hired by them), and unlimited access to their cloud services. Nevertheless, the team does not directly depend on Gandi's organization as it works independently with their own structure. Caliopen is managed by the originator of the project, Laurent Chemla, but the team follows a horizontal organization. Core technical decisions are still made by a core member of the team, Aymeric Barantal. Caliopen's reduced size makes it easy to introduce new technologies that can be used as a base for future Gandi projects.

1.2 Caliopen

It was in 2003 that Laurent Chemla, co-founder of Gandi, conceived the first foundation of Caliopen with "Caliop", a mail service intended to be an alternative in a market dominated by Internet service providers. At the time, mail addresses were tied to service providers and when changing providers the mail account was lost. Caliop intended to end this dependency and set the bar higher with a more sophisticated mail service that had technological improvements such as database storage, data replication or high availability. Google launches in 2004 Google mail, free, and prior to Caliop, putting an end to the project. Fast forward to 2013, the project is reborn with a new objective, protecting the private life of its users. Due to legal complications the project is renamed Caliopen but it is not until October 2016 that it gets its first big funding with the "Banque publique d'investissement" in cooperation with Gandi SAS. Caliopen is currently under active development with a beta phase expected for October 2018.

1.2.1 What is Caliopen?

Caliopen is an open-source messaging platform focusing on privacy in communications, and aggregation of the user's means of communication. It allows the user to communicate with his contacts through any of his imported accounts, with a common interface regardless of the protocol, mail account or mail service used. Each protocol having its privacy and security implications, Caliopen helps the user learn about them, thanks to a "Privacy Index", and improve the privacy in his exchanges, through behavioral modification.

1.2.2 Project's objectives

The first and most important objective of the project is a social one. Making the user aware of the privacy in his daily communications is a first step, followed by a learning process in which the user is taught how to improve it. This privacy is measured and constantly shown to the user as the Privacy Index: a graphical way of representing how well the user is doing in terms of keeping his personal communications safe. This Privacy Index is continually evolving, increasing when the user makes good choices in terms of privacy and decreasing when his choices expose his or his contacts' privacy.

Going further into the privacy aspect, the project envisions a network of Caliopen nodes: secure relays that assure an even higher level of privacy between Caliopen users. Decentralizing the platform limits the risks, both social and technical, of a single point of failure, mass surveillance being one of the main risks the project fights against. Creating a confidential distributed platform will encourage the user to use his Caliopen mail address to enhance his privacy and the one of his contacts.

But Caliopen is not only limited to privacy, it has the intention to bring complementary solutions such as protocol aggregation. In the current society, instant messaging platforms are taking over email for casual communication. Emails become more and more a professional exchange method. The classic email approach to discussions gets old when we see how instant messaging platforms handle it: conversations are contact-oriented, a conversation is tied to a person or a group of persons and new messages are always integrated within those contexts. Caliopen tries to make email evolve with a similar approach, that redefines the concept of discussion. The idea is also that a discussion is tied to a set of contacts: when a new user is added or removed from the recipients in an ongoing message exchange, this new set of contacts becomes a new conversation. Changing the mentality that users have towards mail exchange will surely be difficult, and one of Caliopen's main challenges is creating a satisfactory and comprehensible way of addressing this new vision.

Caliopen not only intends to reach the public as an alternative, more private mail service, it could also be deployed as a private solution for a company's mail service. Deploying its own self-hosted mail platform has become a complicated task, with many security and technical implications. Because of the complications, many companies chose to delegate this task to external services such as Gmail or Outlook. Simplifying the installation and maintenance of a mail system, offering an opensource alternative with technical support gives the option to deploy a private mail solution that avoids external storage of private information. This approach would also put an emphasis on the learning aspect, teaching workers how to keep secrets within the company, and limiting leaks.

Caliopen's business model is divided into business to client solutions and business to business solutions, like the one previously presented. Business to client follows only a freemium model, where users can create free accounts but are proposed incentives to upgrade to a paid account. However, business to business offers can be presented under four main models: private deployments, introduced previously; specific developments, to fit peculiar functional needs; SaaS solutions, cloud-hosted private platforms; and bundles and packages, for example a Caliopen account bounded to another product.

1.2.3 Working on Caliopen

The project Caliopen is always presented with a focus on the social aspects it brings as a private mail service but also has a strong technical backbone to support its principles. The internship focuses mainly on the technical elements, but integrating the team means working on every aspect of the project. The work done is related to making the platform evolve, to implement new solutions and accelerate the development process of Caliopen. This work takes place during the alpha phase of the project, this makes it easier to test new technologies since instability should be expected. To start off, there is a need to get in touch with the current platform and services, while preparing and learning about the tools that will be used to migrate to the new platform.

Chapter 2

State of the art

Throughout this document many challenges tied to technological concepts are exposed. To fully understand them it is essential to present the underlying technologies and their current state. The first part of this chapter analyzes the key differences between **container and classic virtualization**, and a view of **container orchestration** as an abstraction tool. A view on the evolution of the **Cloud Computing** continuum and offer will be presented next, followed by a brief history on software development life cycles and the current state with **CI/CD** will be presented. Lastly, a summary will show how all this concepts relate to each other, enabling one another.

2.1 Virtualization, Containers and Orchestration

Hardware virtualization exists since the late 60s and is being used to isolate multiple systems and share compute resources to this day. For this purpose, a **hypervisor** or **virtual machine manager** is needed, a layer between hardware and the operating system that creates virtual representations of physical resources, later allocated into virtual machines. The hypervisor will share the underlying hardware between the guest VMs, that can be created and destroyed on demand, avoiding the waste of resources non-virtualized hardware can encompass (i.e. applications that run once every week with a machine running 24/7). Every provisioned VM then runs its own OS, completely isolated from other virtual machines concurrently running in the same hardware. Virtualization can be implemented at different levels [6], but this section will focus on the virtualization of physical resources as it is the norm on Infrastructure as a Service platforms.

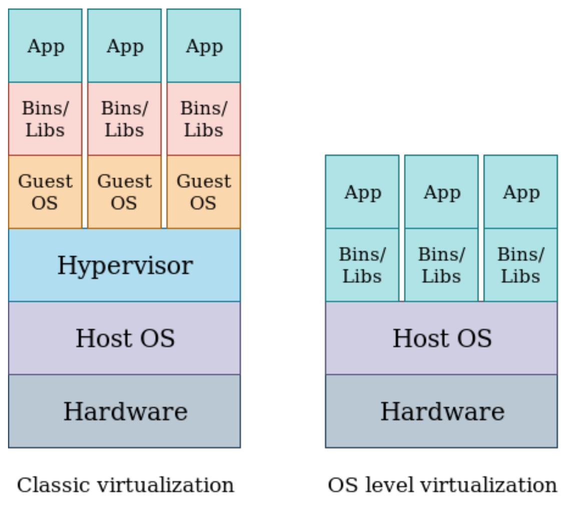


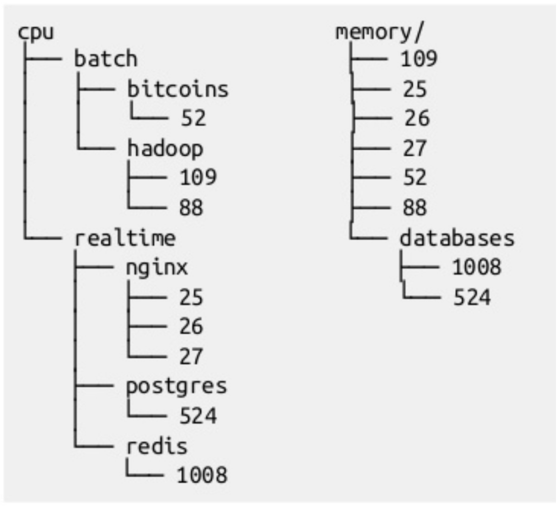
FIGURE 2.1: Hardware virtualization compared to OS level virtualization

Overall, virtualization is a very well known technology of which limitations are also well known, mostly related to performance. Some hardware virtualization types allow a more efficient interaction between the VM and the hardware than others, such as para-virtualization compared to full virtualization [7], but on the other hand, they require adaptation of the OS to interact with the hypervisor, effectively creating an unwanted dependency. Virtualization systems run in production are usually type 1 hypervisors that run directly on bare-metal (2.1 shows a type 2 hypervisor), given that they have better performance.

Hardware virtualization also shows some limits when it comes to resource sharing and allocation. Static allocation of resources resurfaces the problem of misuse of CPU and memory, as most of the time a virtual machine won't be using all of the resources at its disposal. Users usually over-provision virtual machines looking at a worse case scenario of the application they want tu run, extreme cases would be applications that require intensive CPU usage for short periods of time. To palliate this problem, hypervisors allow CPU and memory overcommitment and solutions exist for efficient allocation and migration in datacenters [8][9][10]. Nevertheless, overbooking is shown to have an impact on performance [11].

Classic virtualization is very useful when running multiple guest operating systems, but it is not optimized to run multiple instances of the same kernel, as it can be the case when running multiple Linux based applications. For the latter case, isolation at kernel level, or OS level virtualization, becomes much more efficient.

OS level virtualization, as in container technology, differs firstly of formerly presented virtualization models in the lack of hardware virtualization and hypervisor, as shown in figure 2.1. As previously mentioned, the hypervisor allows to run multiple kernels on top of the same hardware. However, containers share a unique kernel, and it is the operating system's job to isolate the processes, and limit resource usage. The idea of OS virtualization is to provide the same isolation virtual machines give without the need for hardware virtualization and the overhead it entails. Solutions for this purpose exist since the early 2000s with Solaris Zones [12] and FreeBSD Jails [13], although the original idea of filesystem isolation comes from *chroot*. It is not until 2007 that os level isolation is integrated into the Linux kernel, and in 2008 used by LXC[1] for the first Linux container implementation.



```
cpu                          memory/
├── batch                    ├── 109
│   ├── bitcoins             ├── 25
│   │   └── 52               ├── 26
│   └── hadoop               ├── 27
│       ├── 109              ├── 52
│       └── 88               ├── 88
└── realtime                 └── databases
    ├── nginx                    ├── 1008
    │   ├── 25                   └── 524
    │   ├── 26
    │   └── 27
    ├── postgres
    │   └── 524
    └── redis
        └── 1008
```

Figure 2.2: CPU and memory cgroups hierarchy example, source: 1

The two main principles that allow the isolation in Linux systems are **Namespaces** and **Cgroups** [1]. Cgroups tells a process how much resources it can use. They allow memory, cpu, I/O and network limiting and metering, and device node access control, defined through hierarchies (Figure 2.2). On the other hand, Namespaces provide processes with their own, isolated, view of the system. In 2.3, the process inside the child namespace would see the process with pid 3 as the first started process (pid 1).

[1]`https://linuxcontainers.org/`

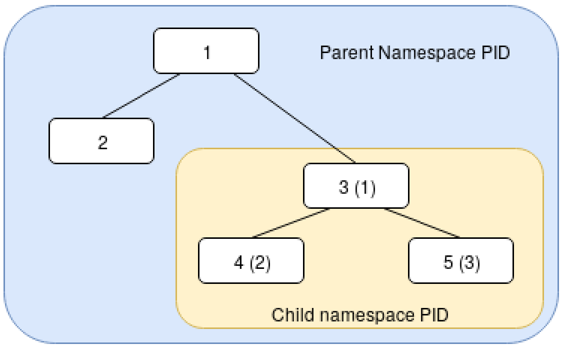


FIGURE 2.3: PID namespace isolation example

Although virtualization performance has considerably improved since its conception, containers' approach to isolation has taken over given that containers add virtually no overhead, improve performance and allow for much quicker scalability [14][15][16]. Containers have another advantage: they are based on images, prepackaged applications that can be deployed on the go. This simplification in the deployment of applications is an enormous acceleration to the release process and fits the needs of developers and ops alike. In spite of these clear advantages, the rise of containers occurred only after the release of Docker[2], supposedly to a well timed release that matched a maturation of kernel namespaces [17]. Since then, alternative container runtimes have seen daylight, rkt[3] is one of the most notorious examples.

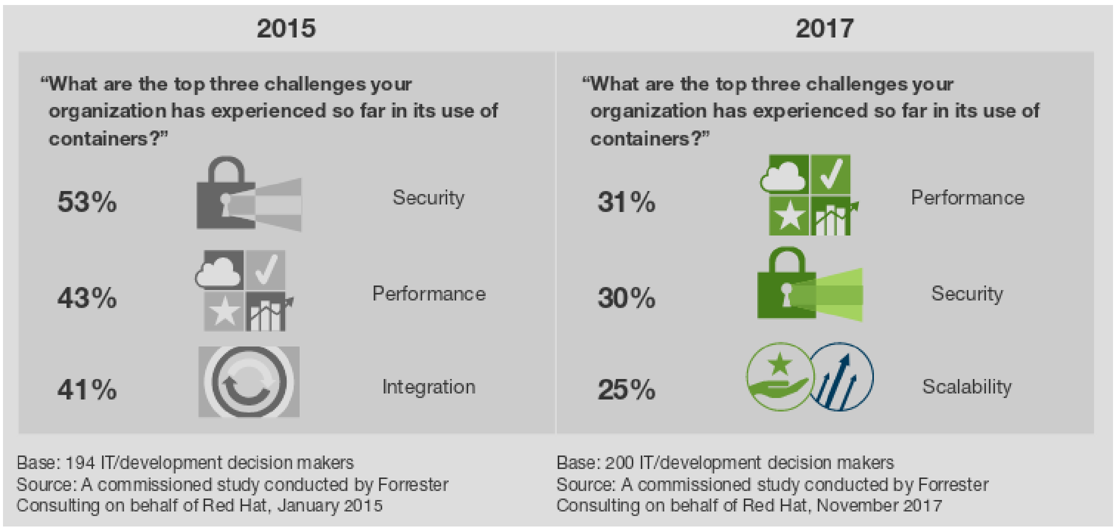


FIGURE 2.4: Evolution of container challenges from 2015 to 2017

In the end, containers presume VM levels of control and isolation, with bare-metal performance and simplified deployment. This isn't always the case. As an example,

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

[3]https://coreos.com/rkt/

Docker's NAT introduces overhead for workloads with high packet rates [14] and companies still raise concern over security, performance and scalability 2.4. Although most of these concerns are unfounded, as shown in [14][15][16] and could be tied to issues outside the scope of containers. Isolation remains an apprehension, containers cannot prevent interference in resources that the operating-system kernel doesn't manage, such as level 3 processor caches and memory bandwidth [18]. Containers also share a unique operating system kernel, a breach in a container that affects the base OS could very easily propagate to other containers in the same machine. Some solutions to this problem involve isolating container contexts on virtual machines in multi-tenancy situations, which could also explain the performance challenges shown in 2.4. It has to be noted that virtualization is not immune to exploits either [19]. [20] shows some good practices for running containers in production from a security standpoint.

Containers are seen by many as the future of virtualization, and while not completely appropriate for running multiple operating systems, they are very convenient for running multi-service applications, even though they become hard to manage and maintain at larger scales. They do not provide scalability, scheduling or clustering capabilities, there is a requirement for a higher level management system.

Container orchestrators provide a higher level of abstraction for managing multi-service containerized applications. Solutions such as Kubernetes, Mesos or Docker Swarm try to simplify the life cycle of containers for the user, at the expense of some learning.

Mesos was first introduced in 2011 [21] as a platform for resource sharing in Data Centers, focusing on cluster computing frameworks such as Hadoop and MPI, it wasn't until 2016 that support for Docker, rkt and Appc containers was announced.

Kubernetes[4] was made publicly available in 2014 [22] as a production-grade container orchestrator, completely focused on containers. Kubernetes was originally a Google product that originated from two other internal container orchestration tools: Borg and Omega [18]. It introduces horizontal scaling, self-healing, load-balancing, automated roll-out and roll-back, resource management, scheduling, and other features, to extend container capabilities. The project is open-source and is now maintained by the Cloud Native Computing Foundation[5].

[4] `https://kubernetes.io`

[5] `https://www.cncf.io/`

A Google Trends search comparing three of the main orchestration solutions shows the growth in popularity of Kubernetes (Figure 2.5) and how it has overturned the ecosystem. Kubernetes is rapidly becoming the industry de facto orchestration platform [23], and companies want to set some standards to simplify cross-cloud portability of Kubernetes applications [24].

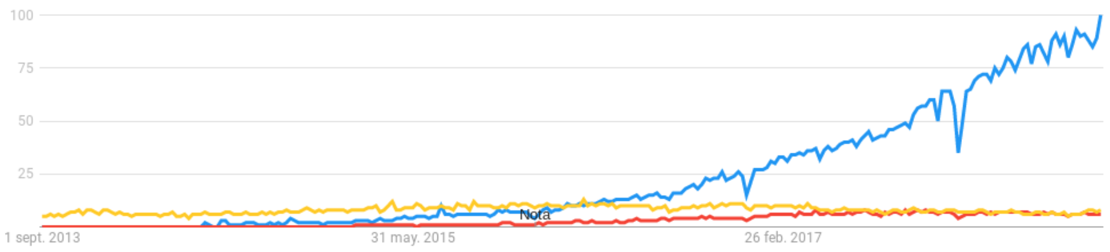


Figure 2.5: Google Trends: Kubernetes in blue, Docker Swarm in red and Mesos in yellow

To conclude, containers show a shift in the process of deploying applications. Thanks to containers and enabled by orchestration, micro-services architectures take the lead. It is no longer about nodes, machines or servers, but about applications and services. Releasing a new service is no longer about instantiating a virtual machine having to take into account how many resources, which os and libraries it needs, etc, it's about preparing an image and deploying it to a cluster, with a platform abstraction for the developer, who ignores the underlying infrastructure and doesn't have to worry about its maintenance.

2.2 Cloud Computing: from IaaS to Serverless

As defined by the National Institute of Standards and Technology [25] Cloud Computing stands for:

> *"A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

Five characteristics are essential to the cloud computing model:

- **On-demand self-service**: computing is provisioned automatically on user demand
- **Resource pooling**: resources are unified using virtualization, containerization or other resource sharing techniques
- **Rapid elasticity**: provisioned resources can quickly be released or new ones allocated
- **Measured service**: resource usage is monitored for the provider and the user
- **Broad network access**: services are accessible over the Internet

And three basic kind of services are also defined:

- **Software as a Service**: providers offer access to an application running on their cloud
- **Platform as a Service**: providers offer the possibility to deploy user-created applications on their cloud, the user does not manage underlying virtual machines or operating systems
- **Infrastructure as a Service**: providers offer the option to provision a virtual machine where the client choses the "physical" resources attached to it and the operating system to run

The services offered by cloud providers are not written in stone, and slight variations around those 3 principles are common. Furthermore, this definition dates from 2011 and in the last decade service providers have increased the number and type of services they offer: Containers as a service, an in-between of IaaS and PaaS, Storage as a service, Functions as a service, and more technology specific solutions such as Kubernetes as a service, in the realm of CaaS.

Each service gives a different level of abstraction, as shown in Figure 2.6, but abstraction comes at the cost of customization:

Infrastructure as a service offers the most flexibility, leaving to the user's choice network configuration, disk space, CPU and memory, operating system, and everything

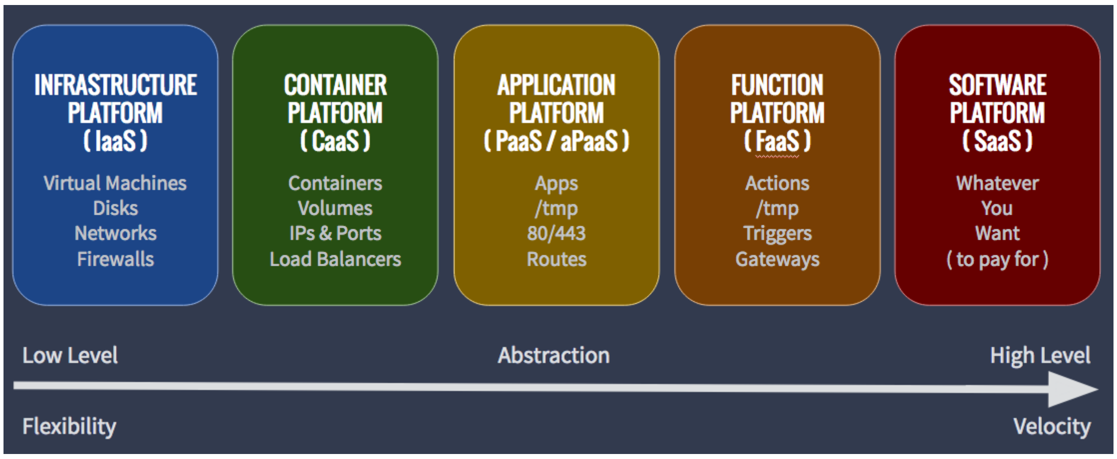


FIGURE 2.6: Cloud offers and abstraction levels, from [2]

on top of it up to his application, he is also in charge of maintaining the platform and handle related problems (high availability, scalability, etc). Hardware virtualization is at the base of this service and comes with its pros and cons.

Containers as a service is a solution to deploy custom containers. It is implemented with an orchestrator behind the scenes and since this orchestrator can affect the way containers are managed, cloud providers usually specify which one is used (e.g KaaS) or offer a proprietary solution (e.g Amazon ECS[6]). This approach gives the abstraction of the orchestrator behind it.

Platform as a Service abstracts away the orchestration behind it and offers the client an already functional platform with the packages he needs, usually from a list of options, pre-installed. Containers are the easiest way of implementing this solution thanks to pre-packaged images and how fast they can be spawned.

A step further we find **Functions as a Service** and **Serverless**. FaaS completely abstracts away containers and servers. The client choses a language and gets billed on execution of his function and not instance size, scalability is then managed without developer intervention. Functions need to be stateless and permanent storage needs to live elsewhere as containers are constantly destroyed and spawned. Serverless adds the principle of event-driven programming to the mix, tightly integrating with other provider solutions. On the other hand, the complete abstraction of the underlying platform ends up being a proprietary lock-in [23]. As an example, Amazon Lambda [7] integration with

[6]`https://aws.amazon.com/ecs/`

[7]`https://aws.amazon.com/lambda/`

the rest of their services[8] (e.g S3, Kinesis, DynamoDB, etc) makes it complicated to migrate to other solutions. A key difference between PaaS and FaaS is that while the former is always running and scales on demand, the latter only runs on function request, scaling transparently.

The cloud has been based since its inception mainly on Linux kernel's abstractions, either bypassing it through virtualization or with new abstractions such as containerization. Serverless is relatively new and as shown in [26], current kernel abstractions are not adapted to it. The paper goes on suggesting the possibility of developing new unikernels instead of adapting the Linux kernel, getting rid of the dominance Linux currently has in the Cloud.

2.3 CI/CD and software development life cycles

Introduced in 1970 by Dr. Winston W.Royce [27], the waterfall model has been used as the model to develop software for more than 25 years. It breaks the life cycle of software development into 6 stages 2.7:

- **Requirements** Analysis of the requirements and definition of what the application should do
- **Analysis** System analysis is made to generate models and business logic used in the application
- **Design** Technical design requirements are specified, such as programming language and services
- **Coding** Source code is written implementing specifications of previous stages
- **Testing** QA and beta testers report issues within the application, often forcing a come back to the Coding phase
- **Operations** The application is deployed, this entails also the support and maintenance of the application

[8] `https://aws.amazon.com`

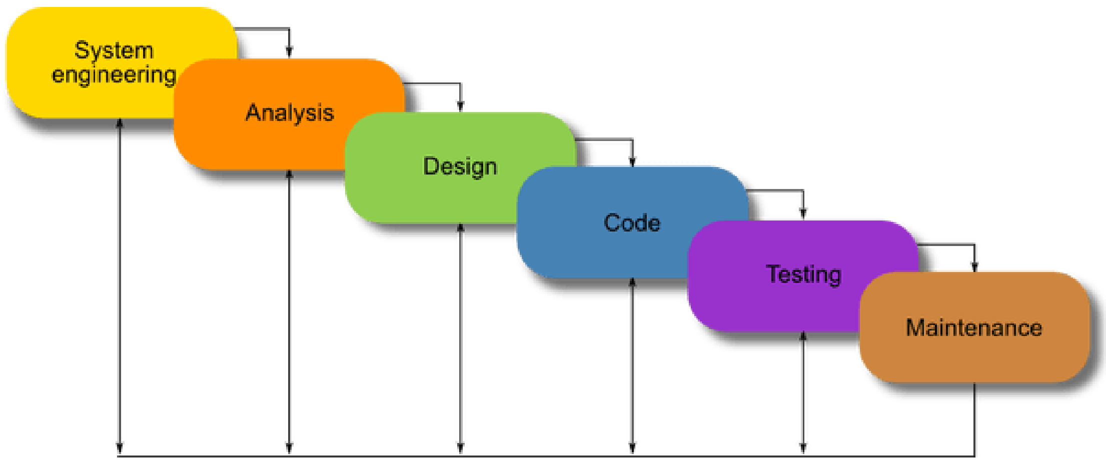


Figure 2.7: Stages of Waterfall SDLC, from Aibrake.io Blog[3]

Waterfall focuses on developing the complete functionality of a software before releasing it. Once the testing stage is passed, software is considered ready to be deployed and goes into the operation process. This procedure's main problem is the lack of adaptiveness. Finding a flaw in the design in later stages is often extremely difficult to fix, requiring a leap backwards in the development process, same applies to client feedback, often coming too late in the development process. David L. Parnas and Paul C. Clements give more details on the idea behind a rational design process in their article [28]. While this model is still used nowadays, or at least modified waterfall models that try to fix the main flaws, it has been phasing out in favor of more agile methods.

In February 2001, a group of seventeen software practitioners write The Agile Manifesto [29], a document that reshaped the landscape of software development. This new vision focuses much more on values and principles instead of requirements and guidelines, emphasizing the need for adapting projects to the teams behind them. Contrary to Waterfall development, releases in an agile context are done at periodic intervals, called sprints (Figure)2.8), usually shorter than a month. This cyclic approach gives more flexibility and allows the reviewing of design choices after each iteration. The 4 main values in Agile development [29] are presented in contrast with traditional ones:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

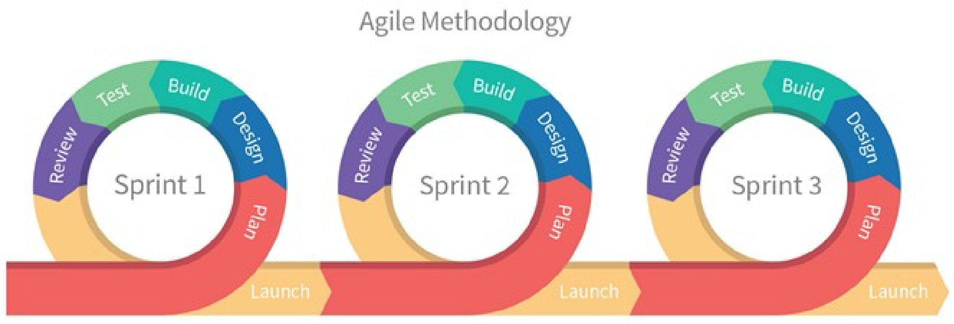


Figure 2.8: Agile SDLC iterations, from Cobalt.io blog [4]

From this principles other SDLC have been born: Kanban Model [30], Scrum, Extreme Programming, Iterative Model, etc. Continuous Delivery/Deployment is yet another subset of agile method in which the core idea is to have software that is always ready for release, without dedicating time at the end of each iteration making a releasable build of the software.

At the heart of every Continuous X approach is automation. Freeing all the steps in a deployment from human interaction is the ideal situation, but automation may not be always possible, or at least, it is proved to be complicated. A fully automated deployment is qualified as Continuous Deployment, while the automation only of the services' tests, be it unit or functional, is called Continuous Integration. In the middle ground we find Continuous Delivery which, as described by Carl Caum in his puppet blog post [5]:

> *"Continuous Delivery doesn't mean every change is deployed to production as soon as possible. It means every change is proven to be deployable at any time."*

However, being proven to be ready for release does not mean it actually needs to be released, and thus the step in which the software is published and made available for the user is manual. The release cycle depending on the company, deploying constantly may not fit every team's needs.

CI/CD is another subset of agile methodologies, but as presented in Kief's blogpost [31], it doesn't prevent it from having conflicts with its base methodology: with CD it is expected to release work in progress, while at the end of an agile iteration only finished

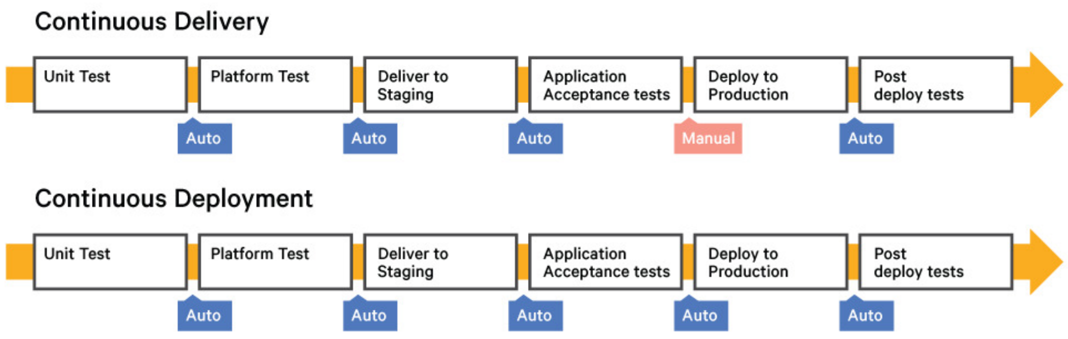


FIGURE 2.9: Continuous Delivery vs Continuous Deployment comparison, from Puppet Blog [5])

features are delivered; for CD there are no release points, code is always in a releasable state.

Jenkins is one of the most used CD tools that does everything from testing to deployment, thanks to their recently released platform Jenkins X, they even allow the integration into a Kubernetes cluster. This kind of solution has two inconveniences: even though they simplify the task, they are still not easy to put in place; and every piece of the tool is so tightly integrated that they don't allow for much flexibility. Drone.io approach is much more simpler and modular, with a core functionality easily expandable with plug-ins, everything running in containers. It is not nearly as powerful as Jenkins, but is easier to put in place and to understand. Concourse CI follows the same container-first approach that Drone.io does, and is easily scalable but configuration can be steeper.

In the end, adaptability to a team's needs is at the core of agile methodologies and as such every team should seek its own approach, finding a solution that fits their needs and capabilities. Implementing real continuous deployment solutions is a big challenge, and even though many technical solutions exist to facilitate it, they remain complex, and as previously mentioned, do not fit every team's capabilities.

2.4 Summary

Three sides of modern development process have been shown: a technological base with the abstractions it creates, what this abstraction means for Cloud Computing and developers, and how, thanks to the simplifications it provides, developers are able to respond faster to management and release requirements.

Container technology has been proven to provide many benefits over traditional VMs from which we can highlight:

- Agility in the deployment thanks to container images
- Decoupling of applications from infrastructure
- Environmental consistency across development, stage and production
- Portability across clouds and distributions
- High level of application-centric abstraction
- Easier micro-service based development
- Higher resource utilization

Adoption of this technology has allowed Cloud providers to offer fast provisioning and highly abstracted solutions for developers, shifting the focus to micro-service oriented architectures in which the developer has to worry less and less about platform problematics such as scalability and resource utilization. Containers have also accelerated delivery times, making testing and deploying new applications easy to automate and forcing the rethink of software development life cycles. The evolution of this technology doesn't end there, as a new trend has emerged with Serverless, in which even the current technological solutions don't seem to fit perfectly the expected needs.

Chapter 3

The current state of Caliopen

The Caliopen platform already has a production environment and a delivery process well established. The comprehension of this environment is crucial to understand the purpose and objectives of this study. This chapter will go through the different facets of the current product. In the four first sections, we will take a look at the team organization, the contribution process together with a service-oriented schema and a view of the topology of the platform. A second part will focus on the progress of a feature to get to production and will take a critical look at some limitations the current structure presents. The final section will exhibit the contribution of the present work to the monitoring and logging in the current platform, some of which will persist through the migration.

3.1 Team and organization

Caliopen's functionalities go from the message handling to the user interaction. In fact, Caliopen provides both the messaging server, in charge of processing, storing and sending messages and a web client that integrates with all the back-end functionalities. This wide offering divides the team into 2 development axes.

There are currently 4 members dedicated to the Web-client. A UI designer and a UX designer take the lead when a new view of the website needs to be developed, they create the first interface concepts, with design, usability and accessibility in mind. The outcome of this first design process then passes on to the integration stage, where two front-end developers transform the idea into code. After this first idea is implemented, it moves onto a proof of concept state, that shows a view of the interface implemented and can be reviewed by the team to improve it.

Two back-end developers work on the implementation of new functionalities server-side (integration of new protocols, message handling, etc.), but are also in charge of maintaining and administrating the servers in the platform. The present work is integrated in the context of the latter problematic, as it will focus on the administration and improvement of the platform from an operational point of view, thus leaving more room for back-end developers to focus on developing new functionalities.

There is an innovation aspect to the project that comes from Caliopen's partners Qwant and UPMC, they work on semantic indexing, and trust and confidentiality on a network. The first one will provide a solution for automatic tagging of mails while the second project aims to develop the privacy aspect of the decentralized platform. It has already been stated that the project has a strong focus on user behavior and as such, sociological studies have been made within the project to better understand client's needs and expectations.

The team's global objectives (i.e. which functionalities need to be implemented first) are fixed by Caliopen's owner, Laurent Chemla. Although the team is centralized in Gandi's office in Paris, there are some remote collaborators that are also part of the project. To improve communication within the team, a daily meeting is organized where everyone discusses progress and objectives.

3.2 Workflow: from feature to release

As an opensource project, Caliopen's source code and tools are publicly available on Github [32]. Even though Caliopen is composed of many services, the choice has been made to centralize all the code on a unique monolithic repository, or monorepo, instead of decentralizing each application. This is done for easier dependency management, but can show some limitations when the repository grows too much in size.

Contributions to the repository follow a feature branch workflow, reflected in Figure 3.2, with a stable default branch for release code (master) and an experimental branch for

```
├── doc                        : documentation for developers, administrators and users
├── devtools                   : scripts, fixtures and other tools
└── src                        : source code
       ├── backend             : Server-side application
       |   ├── brokers         : protocol conversion
       |   ├── configs         : config files
       |   ├── components      : plugin-like application
       |   ├── defs            : interfaces, objects, models...
       |   ├── interfaces      : public apis
       |   ├── main            : interaction with the storage
       |   ├── protocols       : modules that implement standard protocols
       |   └── tools           : standalone programs to manage the backend
       └── frontend            : applications that run on client devices
           └── ...
```

FIGURE 3.1: GitHub repo tree layout

modifications (develop). Changes to the code are generally only made on the develop branch, except for hotfixes, that are also introduced in the master branch. Contributions to the project are made through pull requests, where the contributor explains what changes his branch introduces. These pull requests need to be reviewed by internal members of the project that can request changes or give the green light to merge the modification. Most commonly, a new branch corresponds to a unique feature or fix, hence the name of the workflow: branch-feature. Figure 3.2 shows the develop branch, where features are constantly introduced and the master branch, for which a new addition implies a new version.

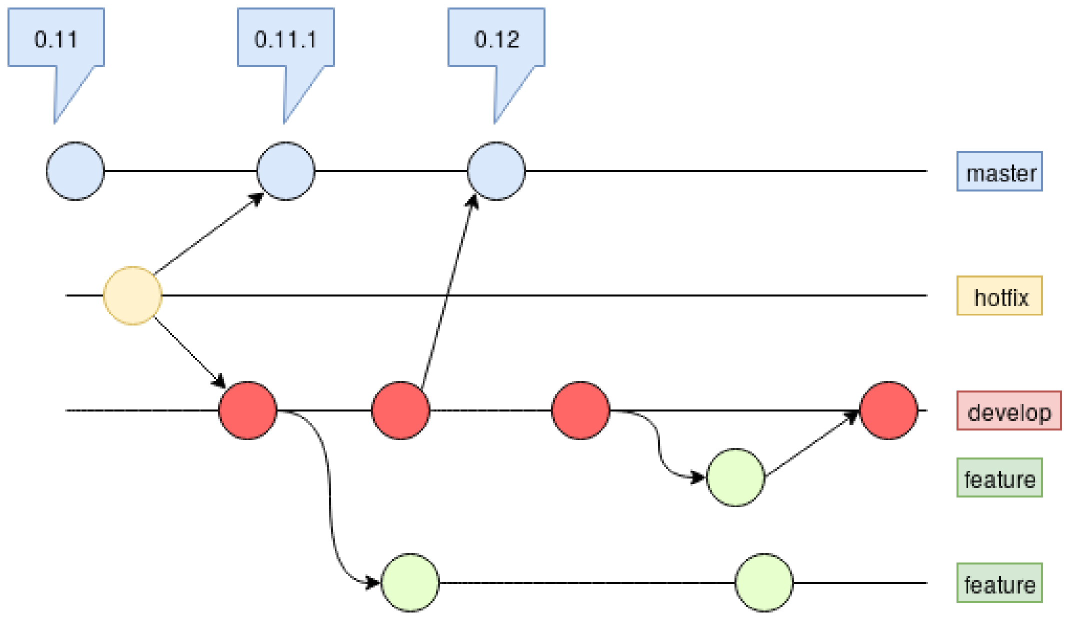


FIGURE 3.2: Branch feature git workflow

Travis is a CI tool that is currently used within the project to automate testing. Every pull request has to pass unit and/or functional tests in addition to the code review before getting merged into the develop branch. A script is used to check if the changes made are to the front-end or to the back-end and proceed with the appropriate tests.

Releases are not made at constant intervals, instead the date is chosen based on number of features implemented and it can be delayed until an important feature is finished if needed. The day of the release chosen, a date of freeze is fixed, from that moment on no new features are added to the develop branch that is going to be tested, merged and released.

3.3 The services that make Caliopen

Caliopen is at the bottom an email platform, and every new service has been built upon that. At the time of writing, Caliopen only supports email protocols: users can create their own Caliopen email address and import any external IMAP accounts. Support for other protocols such as Twitter, Mastodon or IRC is planned, but won't be reflected on the topology presented as it is yet to be implemented.

The platform follows a micro-service architecture with each service having a unique function. The lack of monolithic applications allows for a smoother evolution, and easier extensibility and integration of new services. Communication between every service is made with standard protocols, that way, communication between services written in different languages is seamless. A list of the services developed by the team is shown in Table 3.1.

Table 3.1: Applications developed by Caliopen

Service	**Language**	**Description**
APIV2	GO	Public REST API, acts as a proxy for APIv1
APIv1	Python	Public REST API
LMTP	GO	Transforms messages between internal/external formats
Identity Poller	GO	Checks for new user remote accounts
IMAP Worker	GO	Awaits orders to go fetch remote mails
Message Handler	Python	Treats new incoming messages
Frontend	Node+web	Webclient and webserver
CLIs	Go/Python	Multi-purpose command line tools

Services used on the platform but not developed by Caliopen are presented in Table 3.2.

Table 3.2: Other services not developed by Caliopen

Functionality	**Application**	**Usage**
Cache	Redis	Stores auth and session information
Main storage	Cassandra	Stores users, mails, etc
Index	Elasticsearch	Allows for advanced searches on the data
SMTP	Postfix	Mail server
Process communication	NATS	Message queue for service communication
Object Store	Minio	Store for >1MB files
Secure storage	Vault	Store for sensitive information

Figure 3.3 shows how services interact with each other and how they communicate; mostly done through NATS messages and the Cassandra storage. The platform exposes three main entrypoints: two for mail exchange and one for the weblient. Postfix acts as a central point for incoming and outgoing messages while the IMAP Worker exclusively retrieves external accounts' emails. The api and the web server are exposed to interact with the web-client. Sensible information such as user passwords are stored securely through Vault[1].

[1]`https://www.vaultproject.io/`

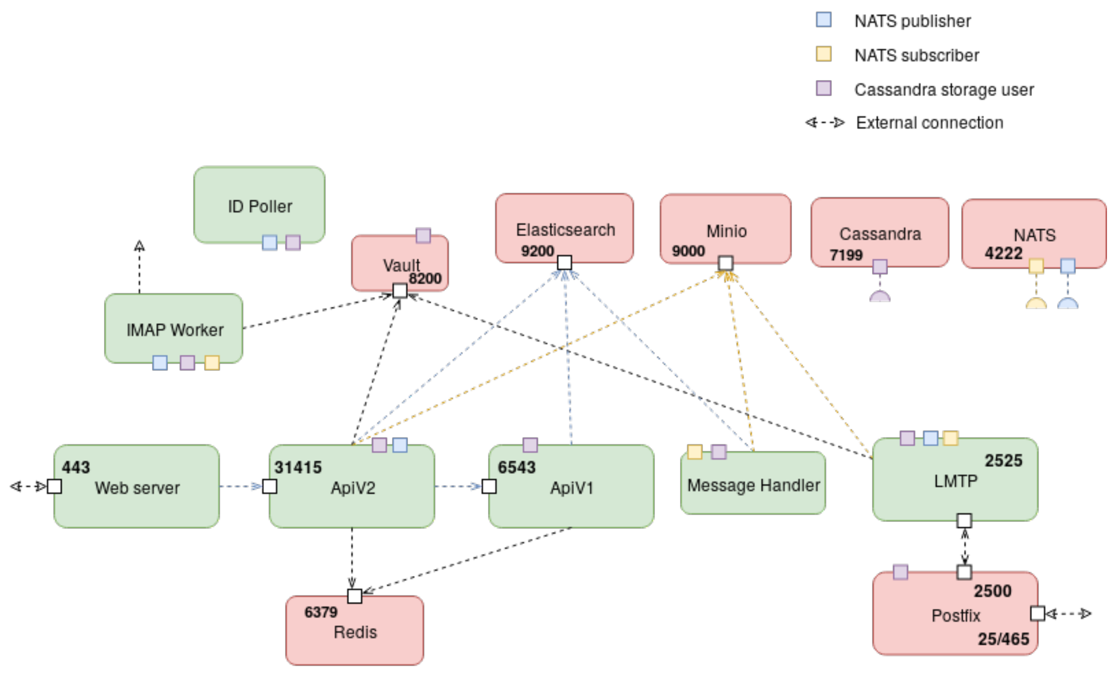


FIGURE 3.3: Caliopen's service view and affiliation

3.4 Caliopen's infrastructure

To support such a large amount of services, the platform includes a total of 27 machines. Figure 3.4 presents the topology of the current Caliopen platform from a "physical" point of view. The machines are not actual physical servers, they are all deployed on Gandi's IaaS cloud platform and are all virtualized.

Virtual machines shown in blue correspond to storage. Because the main storage holds most of the user's information, it is composed of five servers to provide high availability and assure there will be no loss of information. These servers are also regularly backed-up. The red colored machines are two identical servers where both api services are installed. The green color represents the frontend machine. Purple is used for the group of machines dedicated to mail services. There is not a direct 1-to-1 mapping between previously shown services and virtual machines, some machines are used for multiple services. The machines used for monitoring will be presented later on and the next chapter will also introduce the registry. The two DNS machines are exclusively used for name resolution and marketing ones are independent from the platform.

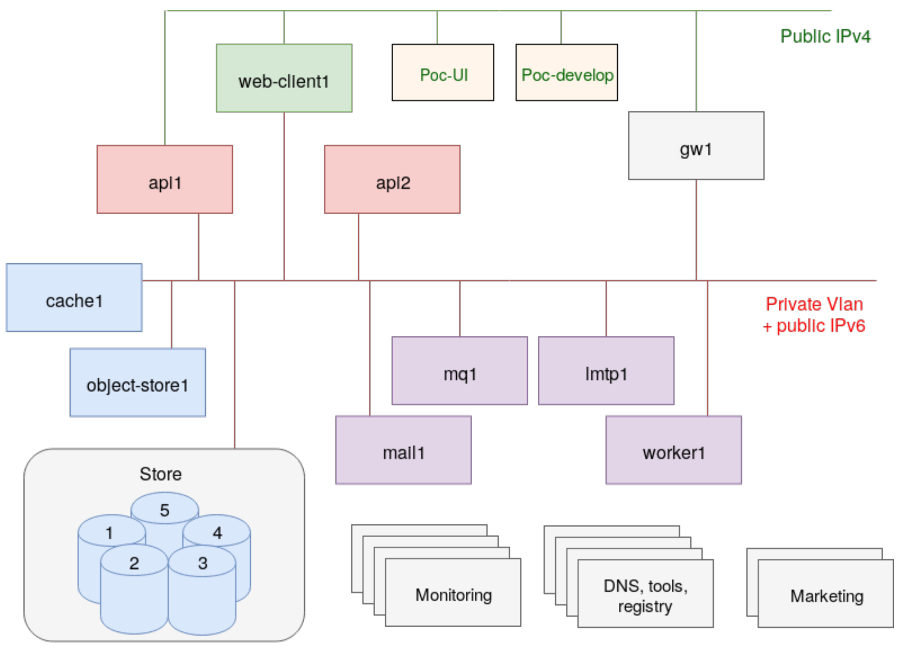


FIGURE 3.4: Caliopen's virtual machine topology

3.5 Development and stage environments

Developers need a quick way of deploying Caliopen services on their machines for basic testing. For this, containers are quick solution to avoid installing every library, dependency and application locally. Managing multiple Docker containers even at lower scale is impractical so Caliopen currently uses a Docker-compose solution that helps building and starting every Caliopen service. A Docker-compose file defines basic building, storage and networking rules to interconnect all the containers. Every time a developer wants to put in place the environment he builds the Docker images locally and starts the containerized services. It tries to emulate production at a lower scale and only with the essential services.

When a release 3.5 is made it is first published to GitHub and then deployed to production. The usual process of deployment involves multiple stages. First off, there are unit and functional tests that are associated to each service, then the code has to be deployed to a pre-production environment or stage, as close to production as possible to endure platform tests and identify any stability problems or bugs. Finally, after every check has passed, the release is ready for production.

A couple of proof of concept machines are used to quickly put in place these new versions: one for UI changes, to allow quicker reviews, and one for the actual platform, so the release is briefly tested before pushing it to production. POC uses the same Docker-compose solution developers use. There are currently three problems with the platform POC.

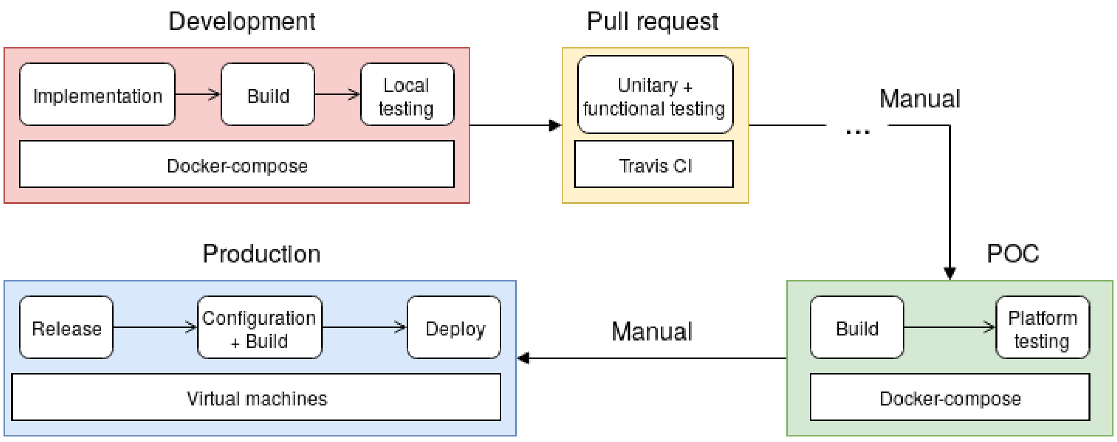


FIGURE 3.5: Release process of a new version

The environment is not used consistently, sometimes POC is out of date because it requires manual updating, and even when it is up-to-date, it doesn't run under the same circumstances as production. Production is not based on a container solution, the binaries run directly on the virtual machine, and although the binaries are the same, network, storage, and configuration are completely different. Furthermore, there is not a real set of tests for the whole platform. Lastly, the current POC deployments do not allow to receive mail, which is necessary to test some functionalities. Not only is the passage between the different stages presented (developer, stage and production) manual, but the phases in each stage are also manual. We will show on the next chapter how a CI/CD solution in combination with our Kubernetes cluster can help automate some of these steps.

3.6 Scalability and resource utilization

As pointed out in the state of the art, one of the problems with virtual machines is over-provisioning. Caliopen currently faces this problem, instances are allocated with a margin to cover extreme usage cases but most of the time they remain idle. Non-elasticity is a concern because machines are billed on resources allocated, not used, and they are not prepared to adapt to user demand. This exposes a second problem, scalability. Even though machines are provisioned anticipating more charge than they currently hold they are not prepared to go further. Current resources suffice for actual use but this doesn't envisage an augmentation in the number of concurrent users.

The first solution to palliate resource waste is to share machines between multiple services. While this diminishes resource waste, it doesn't resolve scalability and elasticity problems at all. The problem comes from the base, virtual machines are not prepared to be flexible. Once provisioned they remain with the resources allocated until recreation or at least, require rebooting to be modified (vertical scaling). But neither deletion and creation nor rebooting are an option, the platform needs to be able to respond dynamically to demand, without downtimes. A classic approach would be to watch the platform and manually create more virtual machines when the number of concurrent users goes above a threshold (automation for this would be a pain to implement in the current platform), also known as horizontal scaling, although this is impractical from an operational standpoint because it requires manual intervention.

Hopefully, Caliopen's services are prepared to be easily scalable. Next chapter will present how a container-based solution provides all the tools needed to automatically do horizontal scaling, limited only by the resources attached to the cluster, and how it uses the virtual machines much more efficiently.

3.7 Monitoring the platform

Now that a general vision of the platform has been shown, monitoring and logging services can be added to get a view of how the system is watched over. Although a monitoring solution already existed, some improvements have been made to it. The logging solution was previously non-existent and has been put in place during the internship. This part shows a general view of the systems used to monitor the platform and the logging solution put in place.

3.7.1 Monitoring and alerting

A platform that operates on a set of 27 machines needs to have real-time information about the status of the cluster. Monitoring lets you know when there is something wrong with the system, shows changes and trends in usage, helps debugging, and all this can feed other systems such as automation or security. Prometheus [2] is an open-source monitoring and alerting toolkit which has its own data model and query language. Prometheus focuses on monitoring services, not machines, and shines for its simplicity to integrate with any service and its scalability. Figure 3.6 shows Caliopen's monitoring configuration.

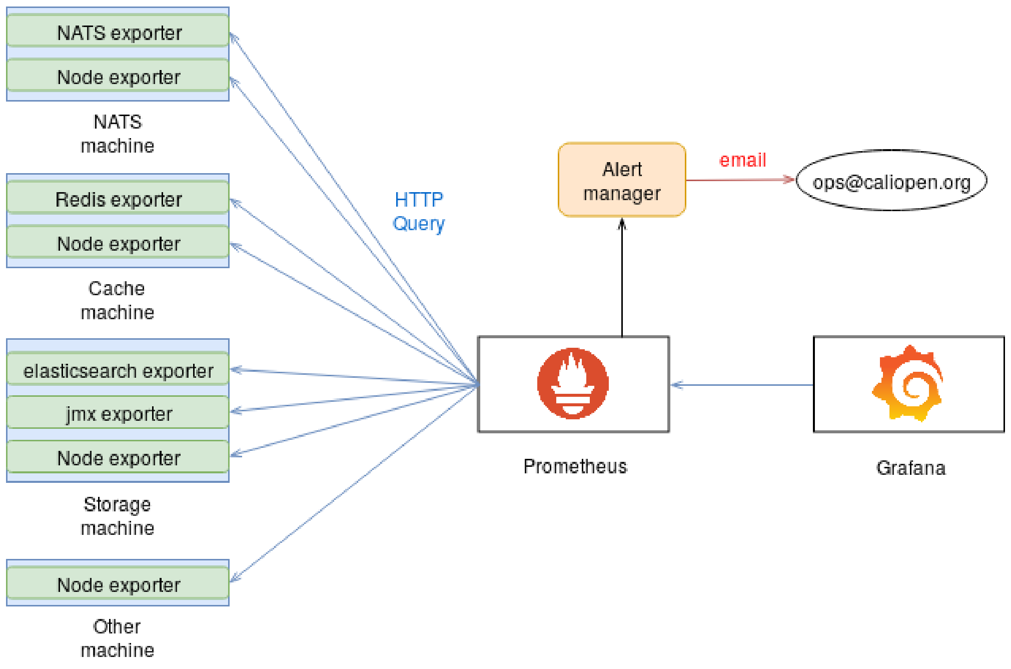


FIGURE 3.6: Monitoring and alerting configuration for Caliopen platform

Prometheus is based on scrapping. To retrieve service metrics from a node, Prometheus needs an exporter that the central server will then query on a specified port at defined intervals through standard HTTP. It includes a local on-disk time series database where the exported metrics are stored. Alert manager, an integrated solution for alerting, is then configured to alert the administrators when certain monitored values go above a threshold, or certain events take place. On top of this data, Grafana is used for visualization of metrics. It gives an easy interface to create dashboards with graphical representations of data (e.g. graphs, tables, heatmaps, etc.).

3.7.2 Service logging

When it comes to logs, the first problem that we face is that services are heterogeneous, they are written in different languages and log in distinct ways and formats. Additionally,

[2]`https://prometheus.io/`

there is a localization problem because those logs remain on the machine that contains the service, decentralizing the access to them. The platform needs a way of centralizing the logs for easier access, and a way of indexing them for future analysis.

We need a solution that facilitates getting every log in a central server. The whole platform working on Linux, syslog is a very appropriate solution for this as it allows sending local syslog logs to a remote server. The complication with this technique is that every application has to go through the local syslog so it is then forwarded to the remote, requiring changes to the way applications in the cluster log. With an already in place Elasticsearch cluster, the most straightforward method of providing both centralization and indexation is putting in place the rest of the ELK stack [3]. The three new components will be Filebeat, Logstash and Kibana, as shown in Figure 3.7.

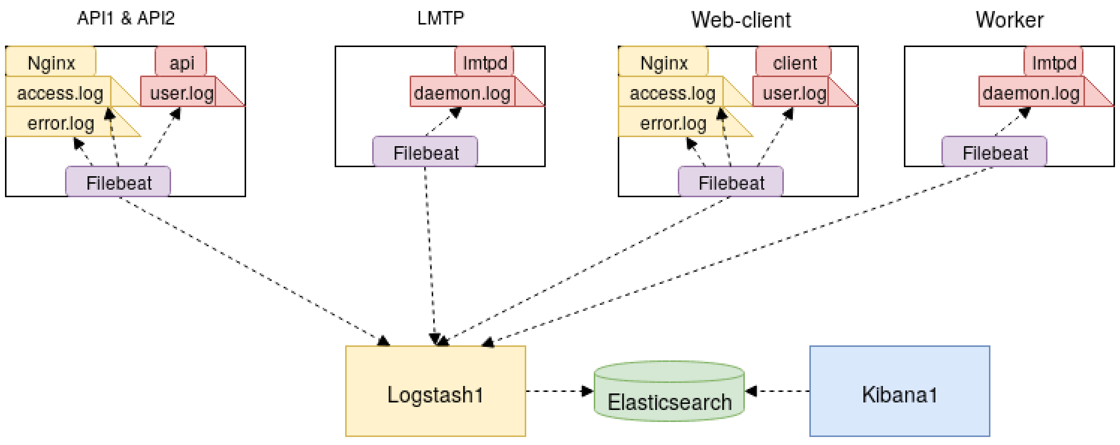


FIGURE 3.7: ELK stack configuration used for logging

Filebeat is a lightweight shipper for logs that has to be present on every machine we want to collect logs from. It comes with internal modules for common formats, but can be used to export any file. Our configuration will be used to fetch Nginx access and error logs, and Caliopen services, that usually log on common files. Filebeat can be used to filter unwanted lines in a file before forwarding it to Logstash, which can be really useful if we want to get rid of info logs and just want to save warnings and errors.

A Logstash pipeline (Figure 3.8) consists of three steps: an input, a filter and an output. In our case, we want input from Filebeat services running on multiple machines and we want to output the result to Elasticsearch. As previously mentioned, we need to create homogeneous logs, and given the diversity of logs we will need to apply different filtering

[3]`https://www.elastic.co/elk-stack`

rules. Hopefully, Logstash allows the definition of multiple filters and the dispatch of input files to the filters based on tags Filebeat provides, resulting in different parsing for each type of log. Output is then sent to Elasticsearch following a unique format, to allow common indexing.

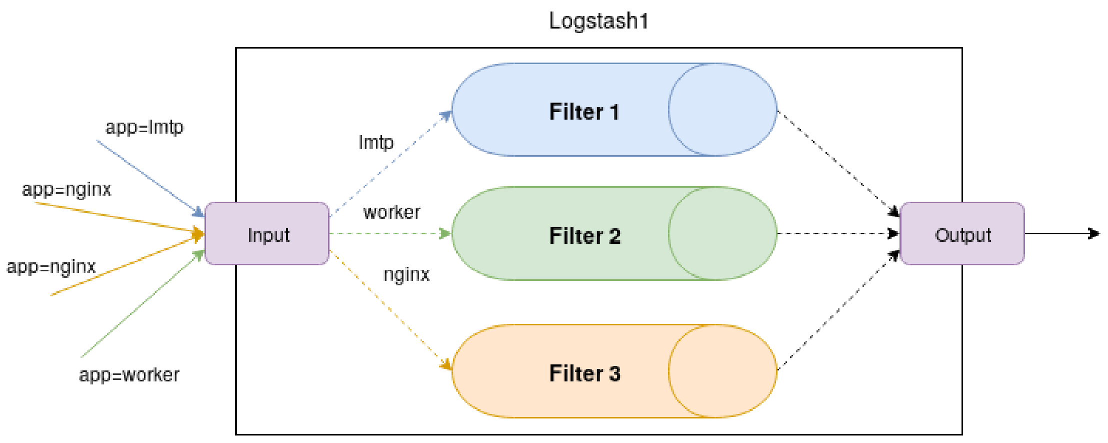


FIGURE 3.8: Logstash pipeline example

Chapter 4

The evolution of the platform

Different visions of the current platform have been presented: a service view has helped with the comprehension of the interaction between the services in the platform. An infrastructure view has shown a lower level focusing on resource usage and virtual machine provisioning, and a presentation of the workflow and pipeline displayed how the product evolves. Some problems have been identified with the expectation that this chapter will provide an alternative solution to mitigate them. Kubernetes, the container orchestrator, will be introduced first, with the advantages it gives over the old architecture. Details on the tools used to deploy the new cluster will follow, to end up presenting the resulting architecture. Lastly, a brief view on the evolution of the workflow will be shown and the improvements that could be made to the new platform.

4.1 Kubernetes

While many container orchestrator exist in the market, truth is that Kubernetes is dominating it. It has become an industry standard, with a huge community of contributors and supporters, which for an open-source project is essential. A big community assures quick bug fixes, technical support and a rapid development of new features. The choice between container orchestrators is clear when comparing not only communities and adoption, but also features and simplicity. Kubernetes origin has already been introduced, with a basic overview of the features that make it a powerful container orchestrator, section 4.1.4 will go into more details. Kubernetes can be thought of as a container orchestrator, but also a micro-services, cloud portable platform. It provides the simplicity of PaaS together with the flexibility of IaaS, simplifying the orchestration of computing, networking and storage. The following sections will introduce the Kubernetes technological concepts used to create and manage applications.

4.1.1 Kubernetes principles

A Kubernetes cluster is divided into Master Nodes, usually containing exclusively services proper to the cluster, or control plane, and Worker Nodes, that provide the runtime environment for containers in the cluster. The division of the control plane (the portion of the platform in charge of the state of the cluster) and the "user plane" (where user containers are deployed), mitigates situations where both planes could interfere with each other. This way, an unavailable control plane does not necessarily prevent running user application from responding to client requests, although it prevents recovery from an unstable state.

The Kubernetes API acts as the entrypoint for interacting with the cluster both for users and administrators. Creating resources is done through the definition of yaml files containing a desired state for that specific resource. The API is then capable of translating the configuration file and storing it on a database dedicated to the state of the cluster. Once stored, the control plane is in charge of attaining this defined state with the help of multiple controllers. Given the distributed nature of a cluster, this declarative approach, as opposed to an imperative, event-driven one, gives more resilience to failures. A declarative definition tells a component **what** it wants, while an imperative definition declares **how** a resource should be treated. The former approach requires a craftier logic and that is why the Kubernetes control plane requires an aggregation of multiple services collaborating tightly to reach an user-defined state. Next section will present the services that make Kubernetes' control plane and their functionalities.

4.1.2 Kubernetes components

Master nodes run the control plane components and, while not exactly part of the control plane, the worker nodes run some components that are also required for container provisioning and node networking. These components are presented below.

On the master nodes

- **Kube-apiserver**: Main entrypoint for cluster administration, also serves as a bridge for the communication between Kubernetes components.
- **Kube-scheduler**: Element in charge of assigning workloads to nodes, based on resource utilization and availability, making sure application requests don't exceed a node's capacity.
- **ETCD** [1]: Distributed key-value store that holds the cluster state and configuration, accessible by every node in the cluster either directly or through the apiserver.
- **Kube-controller-manager**: Service in charge of the multiple controllers that keep the cluster in the desired state. There are multiple controllers, each in charge of maintaining the desired state for a different workload.
- **DNS**: Addon that provides name resolution for the cluster. It is not actually part of the control plane but is still essential.

On the worker nodes

- **Kubelet**: Service that runs on every worker node and acts as a communication point with the control plane. It receives orders in the form of manifests that specify the workload for the node. Directly responsible of creating and destroying containers.
- **Kube-proxy**: Service managing the networking on a node. It allows external requests to reach the proper container through Linux iptables rules.

Figure 4.1 shows how these components interact with each other. In addition, it shows kubectl, the tool to manage workloads on a Kubernetes cluster. Communication with the Kubernetes apiserver is made through standard HTTP requests. Kubectl simplifies making requests to the api with already implemented functionalities such as inspecting cluster resources; creation, deletion, and update of the components; and has client-side validation, so malformed commands are not sent to the apiserver; among others.

[1] `https://coreos.com/etcd/`

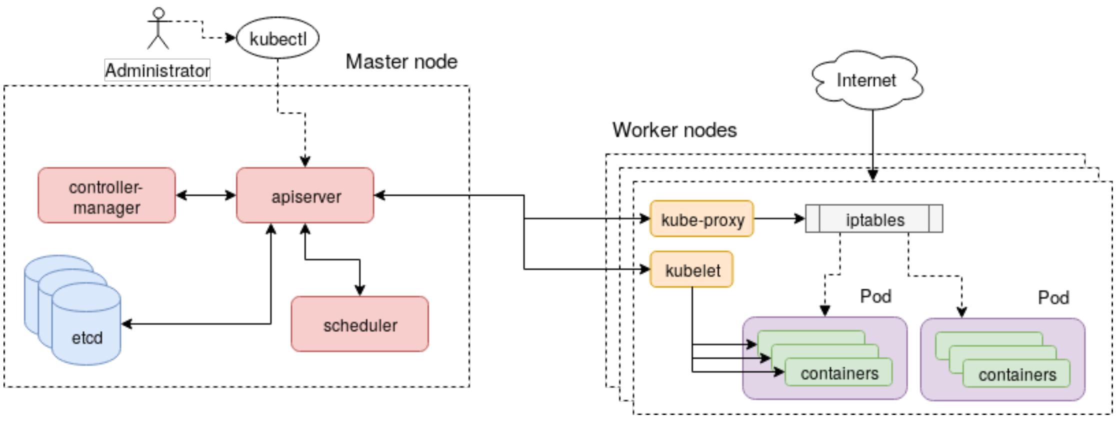


FIGURE 4.1: Kubernetes cluster architecture

4.1.3 Kubernetes workloads

Kubernetes widespread adoption comes, not only from its canny control plane, but the easily understandable abstractions it brings for the user. These abstractions provide high level representations of the state of the cluster. Following, some basic Kubernetes concepts will be described.

- **Pod**: Smallest deployable unit of computing that can be created in Kubernetes. A pod is a group of one or more containers with shared storage and network, always co-located and co-scheduled. The most common use case is one container per Pod.
- **ReplicaSet**: Group of identical pods. It makes sure that the specified number of pods is always available.
- **Deployment**: Declarative description for Pod and ReplicaSet creation, deletion and update. A Deployment file specifies the Pod description and the number of Pods the user wants to run. A Deployment controller will then make sure this specification is met.
- **Service**: Abstraction that defines a logical set of Pods. Contrary to a Pod that is not perennial, a service defines a permanent way of interacting with a set of Pods that lasts through recreation.
- **Namespaces**: Virtual cluster within a physical cluster. Provides isolation, resource division and a different scope for different groups of workloads. A use-case would be multi-tenancy situations or dividing a cluster into stage and production.

- **Job**: Runs one or more pods until completion, once the pods have completed successfully, the job is marked as completed and the pods are deleted. If pods don't successfully complete, a Job can restart them as many times as needed.
- **Configmap**: A ConfigMap allows the decoupling of configuration files from the application image, keeping containerized applications portable. Created ConfigMaps can be mounted into Pods providing a way of configuring applications before initialization.

An application is most of the time ran through the definition of a Deployment. A simple use case for a Deployment is shown in Figure 4.2, it is specifying that our application needs to have two pods running at all times. The Deployment Controller is then in charge of keeping the desired number of pods available. Because those pods can be scheduled anywhere on the cluster, a unifying abstraction of those pods has to be created under a Kubernetes service. This abstraction will provide a unique entrypoint for interacting with all the pods, independently of the worker node where they are running, and implementing basic loadbalancing capabilities. Kubernetes services are essential because node failures imply rescheduling pods, so their location can change frequently. It is important to make a distinction between pod loadbalancing, implemented thanks to Kubernetes services, and node loadbalancing, which will be explained in section 4.4.2. Examples of a Deployment[2] file and a Service[3] file can be found on Caliopen's repository.

4.1.4 Kubernetes features

Kubernetes control plane is a powerful combination of services intervening to keep a permanent state in the cluster. Controllers assure the high availability of applications and a cluster with a big pool of worker nodes guarantees a run environment for those applications. Because of this large run environment, putting a node in maintenance to reboot, upgrade or change it does not bring down any service, that can be instantly rescheduled to other nodes. The abstractions Kubernetes introduces simplify a number of tasks. Scaling horizontally applications can easily be done updating configuration files, and the possibility of scaling automatically based on resource usage also exists.

[2]`https://git.io/fANvK`

[3]`https://git.io/fANfz`

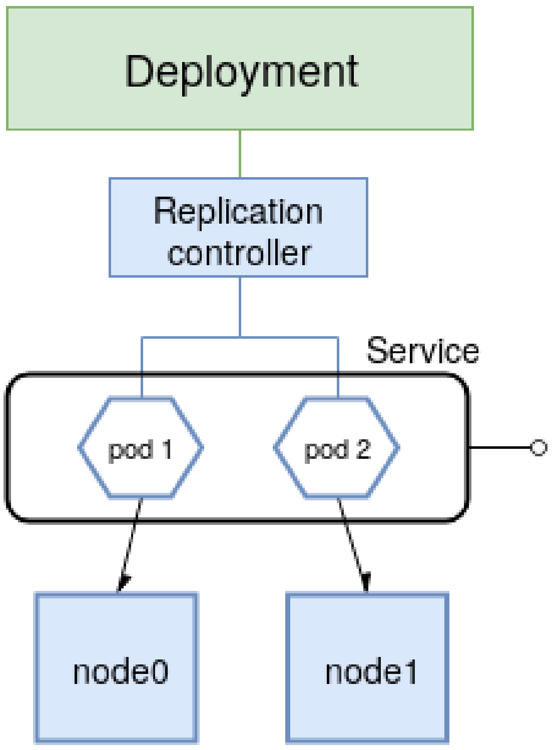


FIGURE 4.2: From Deployment to Pod

Thanks to namespaces, multiple environments can coexist on the same hardware, completely isolated from each other, useful for having almost identical stage and production environments. All the platform is running on containers, that simplify version upgrading and task automation, as section 4.5 will show.

4.2 Kubernetes for devs: Minikube

In section 3.5, the current development environment was presented, a solution based on Docker-Compose that deployed a minimal Caliopen stack locally for developers. An equivalent, Kubernetes-based solution, to deploy a simple local developer environment exists, Minikube. Minikube bootstraps single node (acting both as master and worker) Kubernetes clusters, either on a virtual machine or running directly on the host machine with Docker. One of the advantages of this solution is that Kubernetes can be run on any OS able to run a virtual machine while Docker-compose requires the host to run Docker.

After the first contact with Caliopen's applications, it was time to start working on migrating to a Kubernetes solution to start familiarizing with its concepts and constraints. Minikube is the quickest way to put in place a cluster, but adjustments to every service were required to fit Kubernetes specification: each application needed to become a deployment, each deployment needed to be exposed through a service, and every application needed to load configuration during runtime, and not be integrated during build-time in the container (see later in section 4.5.1). Furthermore, a Caliopen instance requires setting up the storage with Caliopen's CLI, a task perfectly adapted for Kubernetes Jobs.

The first idea that comes to mind is that this migration of the development environment means more complexity than simplification for the developer. Whereas adapting to a Kubernetes mentality requires time at first, in the long run it allows the team to get more in touch with Kubernetes and get a system that applies concepts closer to the production environment. Moreover, within the context of the project, simplifying the deployment of a developer environment was also intended, and as such, the solution prepared favors the use of a single command to get a ready to use platform. For this task, some scripts have been developed[4]. When using the script, a developer only needs to choose what kind of development he intends to do (frontend, backend, or both). The result is a developer environment easy to put in place and that acts on the same principles that the actual production platform.

4.3 Creating the Caliopen cluster from scratch

Although the production environment may only be deployed once from scratch, a consistent, repeatable, and automated way of putting it in place is useful in case of disaster, or simply in case another deployment needs to be set up with a different configuration. There are three main steps to get to the point of having Caliopen's applications running on the new cluster: virtual machine creation, bootstrapping the Kubernetes cluster on the newly created virtual machines, and finally, the deployment of the Caliopen stack. This section will present three tools used for this purpose: LibCloud, Ansible and Kubeadm.

[4]`https://git.io/fA5EO`

4.3.1 VM creation with Libcloud

Libcloud is a Python library meant to interact with any cloud providers, but hiding the differences between them. It allows the user to manage different cloud resources though a unified API, providing an abstraction of the underlying cloud specific implementation of basic concepts such as computation, storage, DNS and containers. For instance, a virtual machine under AWS may be called an EC2 instance and under Gandi's IaaS a virtual machine, Libcloud would refer to them both as Nodes, serving a single api to interact with them.

The interest in this library comes in the integration with Ansible. A combination of both tools allows the automation in the process of creation and deletion of virtual machines through Ansible Playbooks, which will be later explained.

Support for new cloud providers in Libcloud comes from modules, and although Gandi compute module (the one used to interact with the IaaS platform) was already developed it was not up to date. The platform has been, since the development of the module, expanded with private networking possibilities, very useful for the future platform. Changes to the module have been made to be able to use this new functionalities from within Libcloud. The changes made are currently under review to be integrated upstream in the main Libcloud repository[5].

4.3.2 Kubernetes cluster bootstrapping

Creating a production ready Kubernetes cluster requires putting in place and correctly configuring all of the aforementioned components, see section 4.1.2. This task is complicated and Kubernetes provides tools to simplify the bootstrapping. In fact, it can be done in a variety of ways, with some tools being more powerful but requiring more setup than others. Three main approaches are presented in the official documentation: kubeadm, kubespray and kops.

Kops being the most advanced option, it is currently only available to deploy clusters on AWS, GCE or DigitalOcean. It requires Golang drivers for the interaction with the cloud provider platform, but due to the lack of drivers, it is not an option available for this work.

[5]`https://git.io/fAdlQ`

Kubespray provides a more generic deployment option through Ansible playbooks and inventories, and customizable configurations. It can also provision virtual machines but requires a Terraform[6] driver.

Kubeadm is a bootstrapping solution to deploy a minimum viable Kubernetes cluster that conforms to best practices. Behind the scenes, both Kops and Kubespray use this tool, expanding it to provide more functionalities. The idea behind Kubeadm is to provide a basic, but solid Kubernetes cluster to build an infrastructure on top of. This is the solution that will be used for this project.

4.3.3 Automating tasks with Ansible

Two pieces of the puzzle have been shown: Libcloud, providing an api to interact with Gandi IaaS, and Kubeadm, to create a Kubernetes cluster from a given pool of nodes. But this tools still require manual interaction to use and the objective is being able to automate the process to the point where a Caliopen stack can be started from scratch through a single command. Ansible[7] is the piece that puts it all together.

Ansible is a simple automation tool for infrastructure provisioning, configuration management and application deployment. Through the definition of playbooks it facilitates the repeated execution of commands on a defined set of hosts, through ssh. Playbooks consist of a series of tasks that are executed on a remote host, they can be seen as instruction manuals. It is important to note that running a Playbook twice guarantees the same state, because Ansible does not execute already satisfied tasks.

After the modifications made to Libcloud to support Gandi vlans, the Ansible modules[8] had also to be adapted to represent these modifications. To expand the functionalities, a dynamic inventory script was also implemented[9]. This script retrieves information about virtual machines directly from the cloud provider (IP addresses, resources, configuration parameters, location, etc.). This is useful to avoid having a static file that needs manual updating when a machine or some of its configuration changes.

[6]`https://www.terraform.io/`

[7]`https://www.ansible.com/`

[8]`https://git.io/fAdlW`

[9]`https://git.io/fAdlE`

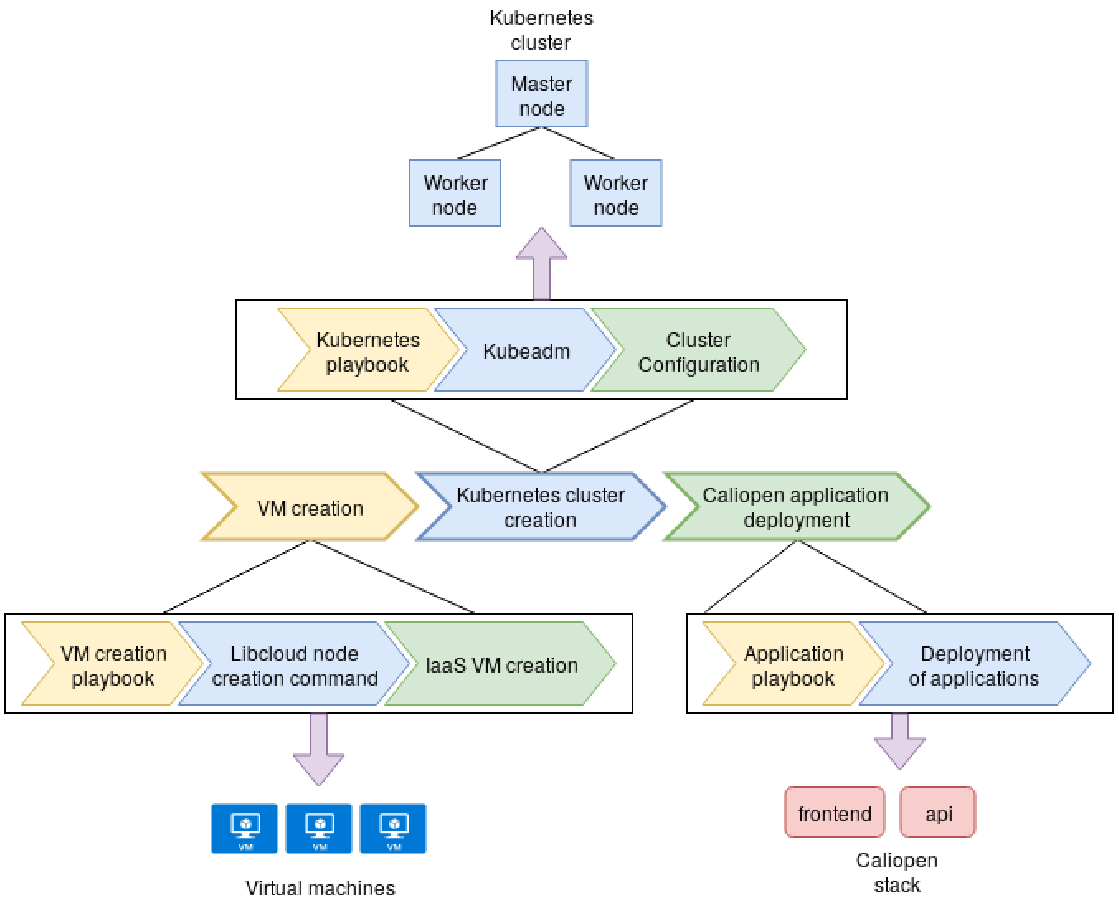


FIGURE 4.3: Steps of the automatic deployment

Figure 4.3 shows the instructions that we will configure to reach our end goal. First step creates simple virtual machines on Gandi IaaS, second step uses kubeadm to bootstrap and configure the Kubernetes cluster on those virtual machines. The third step deploys the applications to the cluster, all this through the definition of Ansible playbooks.

4.4 First Kubernetes cluster

4.4.1 A secure architecture

Thanks to the combination of the tools presented above, it is now possible to deploy the Kubernetes cluster on Gandi IaaS platform. The objective architecture consists of a single master node and four worker nodes, with the possibility of expanding it later (see section 4.6). Figure 4.4 presents the physical view of the solution. The newly deployed virtual machines that are part of the cluster are shown in blue and only have an IP within the private network of the platform. The schema also introduces a new machine, *lb1*, which will provide external access to Caliopen's applications, more details on this will be presented in the next section.

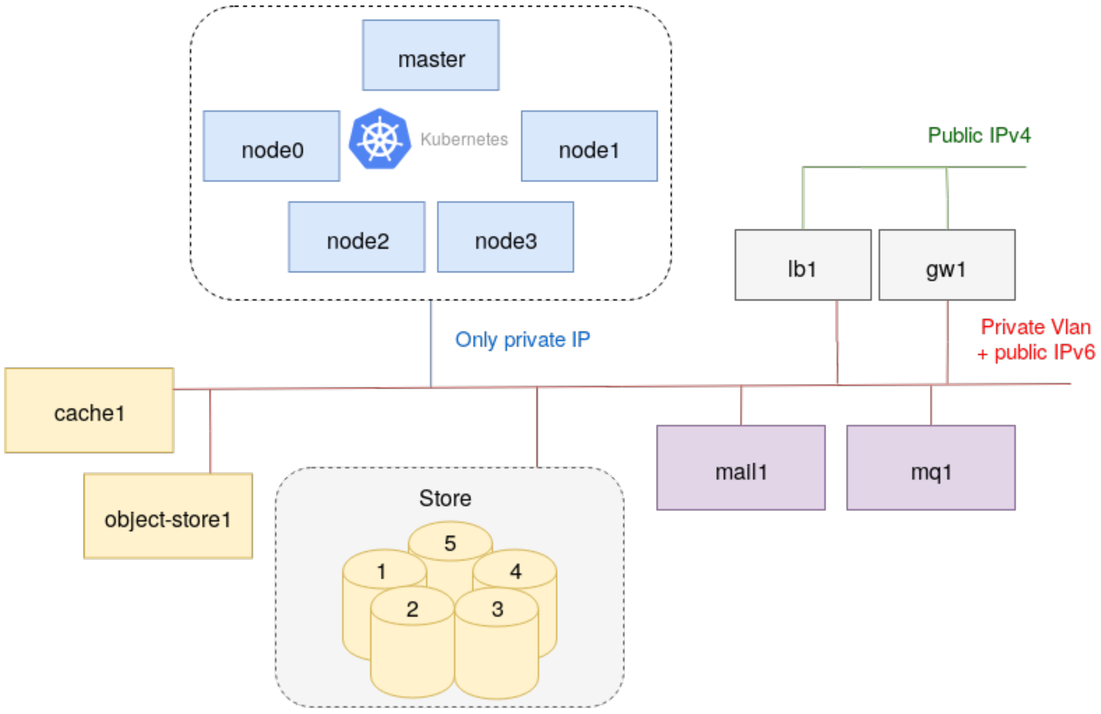


FIGURE 4.4: Architecture of the new platform

The idea in mind when making the cluster fully private is to provide it with a security barrier, no public IP, no possible intrusion or unwanted exposition. Although this limits undesirable access, it also limits the entrance for an administrator. To solve this problematic, a bastion host is needed, that is, a virtual machine that is exposed both externally with a public IP, and internally with a private IP, granting it access to the machines in the Kubernetes cluster. Of course, this machine is also vulnerable to intrusions, but it is easier to secure and monitor a single machine than a pool of machines. Figure 4.7 shows how an administrator does to get access to the cluster: through ssh to the bastion node that contains every tool needed to interact with the pool of machines in the cluster. The bastion host exposes the ssh port but limits access to a list of trusted keys that grant access exclusively to the persons in the team. The bastion host in the case of Figure 4.4 is *lb1*, that also operates as a load balancer. Ideally, load balancer and bastion host should be separated on two different machines.

Kubernetes scheduling by default uses any worker node in the pool to deploy pods. When a platform starts to scale in number of worker nodes, having a pod moving around anywhere in the cluster can complicate load balancing and increase dramatically

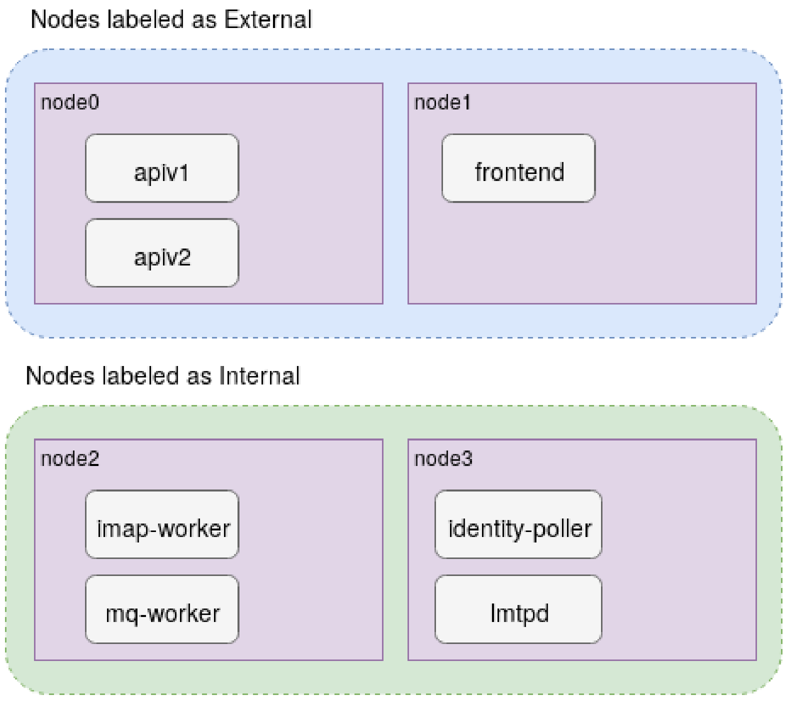


FIGURE 4.5: Separation of services in the cluster

in-cluster network traffic. This entropy can be limited thanks to what Kubernetes calls NodeAffinities, that reduce the number of nodes a pod can be scheduled into thanks to labels in those nodes. In the case of Caliopen's cluster, there are four worker nodes which can be used for scheduling applications. NodeAffinites are used to separate client-facing applications that are meant to be accessible from the Internet and local applications that do not need exposition outside of the Caliopen platform. Figure 4.5 shows this division, having nodes zero and one dedicated to external applications, and nodes two and three for the rest of the applications. Next section will present how an application is made accessible outside the cluster and how separating pods in two categories helps with loadbalancing.

4.4.2 External access to the cluster's services

It has been shown that pods are by nature ephemeral and that a Kubernetes service represents a logical set of pods or a micro-service, which has to be used as the entry-point to interact with those pods. By default, Kubernetes services are only exposed within the Kubernetes cluster network, meaning that they are only visible by other pods in the cluster. Multiple ways of exposing them externally exist. On Cloud platforms with specific support for Kubernetes clusters, external load balancers that know how to comunicate with pods can be dynamically provisioned. The cloud controller manager is in charge of the provisioning and the exposition can be done without the administrator's interaction. Cloud controller managers and cloud specific implementations are out of the scope of this work simply because the cluster is being deployed on Gandi's IaaS platform which currently does not support any of those functionalities. Even though in this work's context it cannot be automatically created, a load balancer is a key piece in a Kubernetes cluster. The distributed nature of the platform makes it possible to lose nodes due to eventual failures and a load balancer can keep track of alive worker nodes. More on the load balancing configuration later in this section.

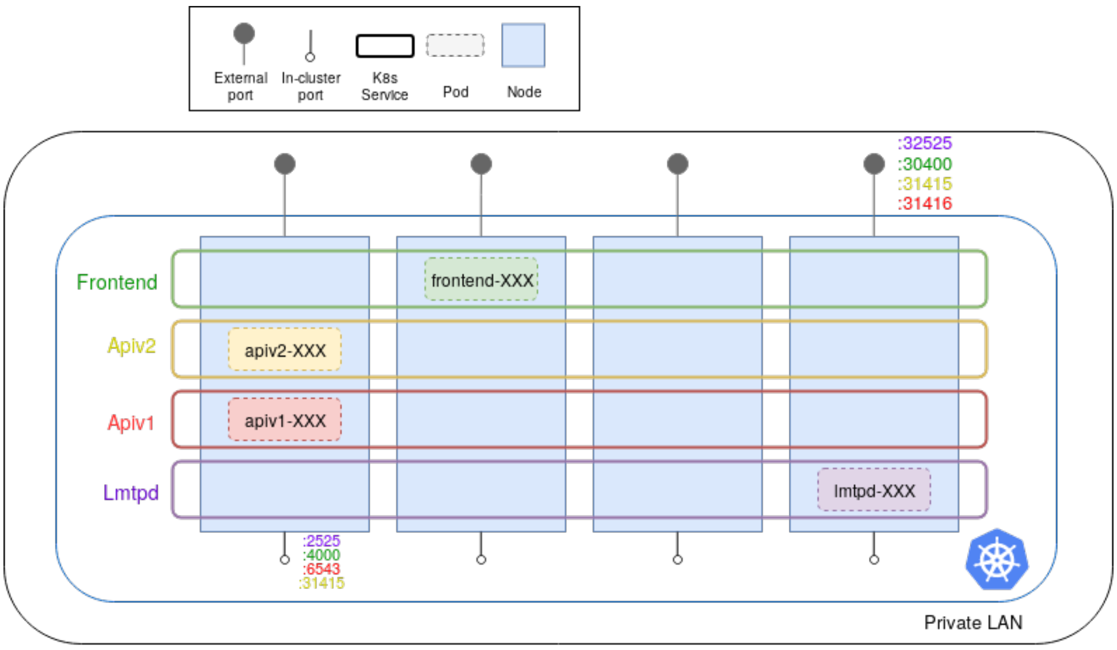


FIGURE 4.6: How pods are exposed outside and inside the cluster

We saw that by default a Kubernetes service is only visible to pods running inside the cluster, but we need a way of exposing them to the private network, with the rest of the platform. The way Kubernetes Services can be exposed to the private LAN is through NodePorts. Defining a Service as a NodePort exposes the service on a given port on every node in the cluster, worker or master, even if the pod is not running on that specific node. How a pod is found within the cluster is up to kube-proxy and the service, this article [33] gives more details on the network implementation of this solution. Figure 4.6 gives a vision on the multiple layers of exposition there are. On a first instance, a service is exposed exclusively inside Kubernetes, being only reachable by pods running inside the cluster. This is useful for applications that require communication exclusively with other applications that are inside the cluster, but no machine in the Caliopen platform can access it; this is represented by the in-cluster ports. When the service is exposed through a NodePort, the service is being exposed at a node level, inside the private network. This makes the application reachable by other applications inside Caliopen's platform, but the nodes in the Kubernetes cluster having only private IP, our applications are still not accessible from the Internet; this is represented by external ports in the schema. The objective of the architecture was exactly to limit this exposition and to carefully chose what gets seen from the outside, the point of exposition being the load balancer.

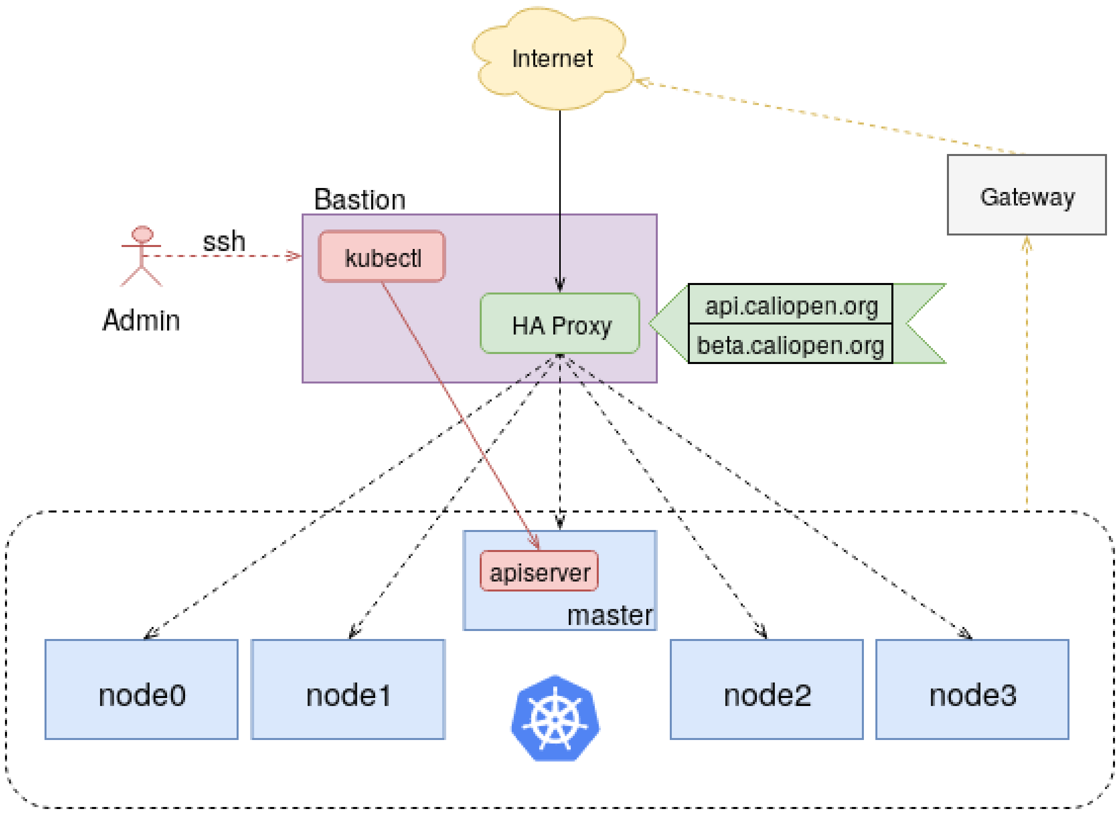


FIGURE 4.7: Service exposure and cluster administration

The HAProxy load balancer shown in Figure 4.7 acts as a TLS endpoint, a client-facing entry-point for Caliopen's Frontend (beta.caliopen.org) and API (api.caliopen.org), and an access point to the applications that run in the Kubernetes cluster but need to be accessible by machines outside of its scope. Thanks to the definition of NodeAffinites separating nodes in two blocks, we have enclosed applications that are to be used internally. The load balancer can then grant access to those nodes only to machines in the private network. Figure 4.8 shows that even though every node exposes every service, the load balancer only queries for internal services (in blue) nodes two and three, and for external services (in purple) nodes zero and one.

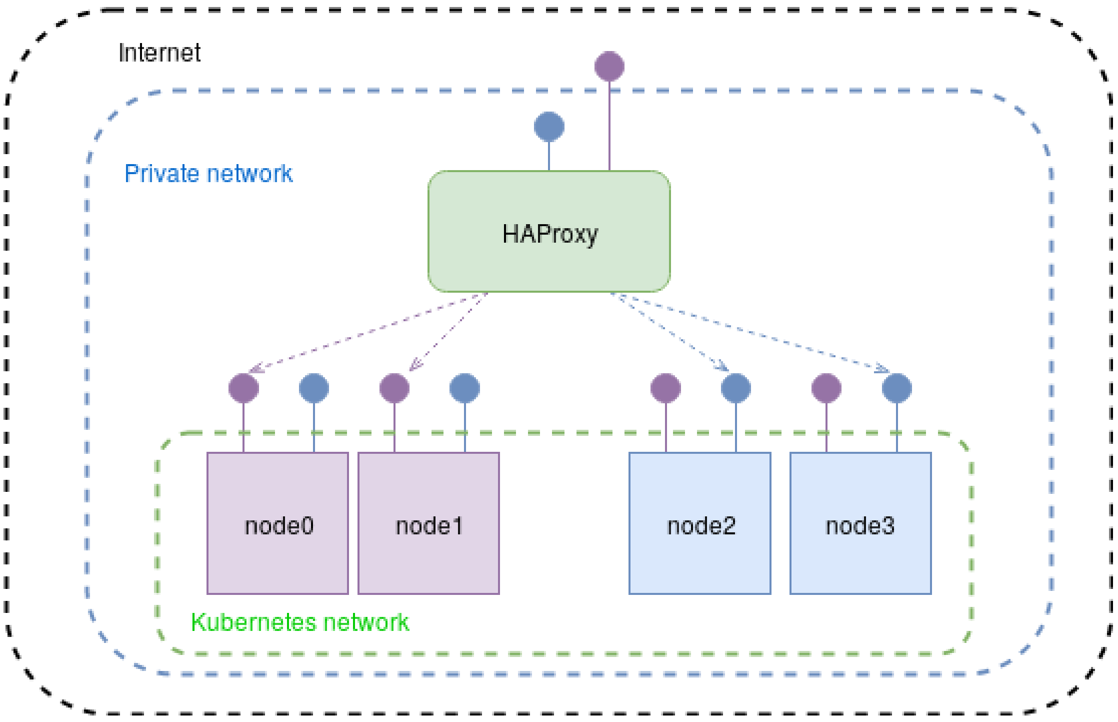


FIGURE 4.8: Load balancer configuration

4.5 CI/CD

One of the main objectives of the project was to improve the release process. Chapter 2 presented many aspects of the old platform and how the release process could sometimes be long, incomplete or untested. This section will focus specifically on the development of a CD pipeline, and how the new platform integrates into this pipeline. First, a vision on the evolution of Caliopen's Docker images, followed by the new CD tool introduced and its current and envisioned scope.

4.5.1 Docker and Dockerfiles

The state of the art introduced container images as prepackaged multi-platform solutions that accelerated deployment times thanks to their ready to deploy nature and the fast instantiation of containers. Docker has helped introduce developers to the process of release thanks to concepts such as Dockerfiles and Docker registries. Dockerfiles on one hand are files with a simple syntax that define how to assemble and run an image, and help automate the build of an application. Registries on the other hand can be seen as an equivalent to *apt* repositories, facilitating the "installation" of applications.

The idea of a Docker image is to be portable and lightweight. Images are by nature portable as they can be ran on any host with a Docker runtime, but getting an image of the right size, including only the basic packages needs some extra attention. Caliopen services are mostly Python and Go applications, with the exception of the frontend, developed in javascript. Python being closer to being an interpreted language and Go being compiled, they have totally different requirements to run, and should be treated differently when optimizing their Docker images. In the case of Go, creating a minimal image requires only a statically compiled binary inside an empty container, while python requires its VM to run.

There are two problems with the original Docker images. First, they are not built with the minimal packages needed. This increases dramatically the size of the final image as well as the build time, shown in Table 4.1. Secondly, they package configuration files inside, binding a build to a specific configuration, an approach totally contrary to containers' philosophy of portability. The work done to improve those images shows the improvements in size and build time in Table 4.2.

Table 4.1: Image size and build time before modifications

Image	**Size**	**Build time (min)**
apiv2	1.25GB	2:47
lmtpd	1.22GB	2:26
imap_worker	1.25GB	2:34
identity_poller	1.21GB	2:23
apiv1	568MB	7:14
cli	577MB	6:58
message_handler	562MB	7:01

Thanks to Docker multistage builds, final images do not contain the packages used to build them, reducing greatly the size. Furthermore, having those "builder" images available in the Docker registry reduces the build time avoiding rebuilding shared libraries between images.

Table 4.2: Image size and build time after modifications

Image	**Size**	**Build time (min)**
apiv2	38.1MB	0:19
lmtpd	27.4MB	0:14
imap_worker	49.7MB	0:27
identity_poller	20.2MB	0:13
apiv1	375MB	1:38
cli	380MB	1:16
message_handler	364MB	1:11

4.5.2 Docker registry

One of the strong points of Docker are registries. A Docker Registry is a centralized server that facilitates storing and distributing built Docker images, with the possibility of versioning those images through tags. The resulting image of a build is usually stored in Docker's official hub to give public access to them, but Caliopen's applications are still in alpha and not yet ready for public release. Instead, we put in place our own registry.

The first use case of our registry is avoiding building in a developer environment. Up until the deployment of the registry, when a dev started the Caliopen stack every image needed to be built locally, without taking advantage of the fact that the same image had already been built by another person at one point. This problem extends to POC, images were built the moment it had to be updated. Having the images stored reduces the time to deploy the environment and guarantees common images for all developers. The images used in those environments are based on the develop branch and as such will be tagged develop.

The second use case is storing images for every version of Caliopen's application's, so they can be deployed in the Kubernetes cluster, either in the stage or production environments. A historic of images facilitates rolling back to a previous version in case of bugs. This way we have the newer release of the images, tagged latest, and prior version, tagged with the number of the version (0.11, 0.12, etc.). We will integrate the build of images into an automated process, as next section will show.

4.5.3 CI/CD pipeline

The principles of continuous delivery and deployment have been presented in the state of the art, it improves a team's productivity by automating some steps of the delivery process. Section 3.5 presented the old release process where most of the steps required manual intervention. This section will show the capabilities of a CD tool such as Drone, integrated during the internship into the project. Drone.io is the choice made first and foremost for its simplicity. Compared to other solutions, Drone provides a way of defining a pipeline very close to human language. However, this simplicity comes at a cost.

A pipeline is a series of instructions that are executed on reception of a specific event associated generally to a code repository. The pipeline procedure is triggered when events such as pushing code or creating pull requests are detected. The instructions can be shared between every type of event or can be unique. An example of instruction very frequently implemented on a pipeline is passing a test or building the source code. To consider a build as passing most the time every instruction in the pipeline has to be successful. We have currently planned to automate the following situations, highlighted in bold:

- **Pull request opened**: test code and service build
- **Pull request merged into develop**: test code; build modified services and update develop registry images; update the stage environment with the new images; eventually, make platform tests
- **Release made**: build stable images and add them to the registry; eventually update production containers

The three situations in Figure 4.9 represent a continuous Delivery situation and not a continuous Deployment one, because releases are still triggered manually. The schema shows a target pipeline and not the current reality of this work's implementation. At this time, platform tests are still not implemented nor designed and the stage environment is still a work in progress. The only manual interaction involved in these processes is the first trigger: opening a PR, merging a PR or making a release.

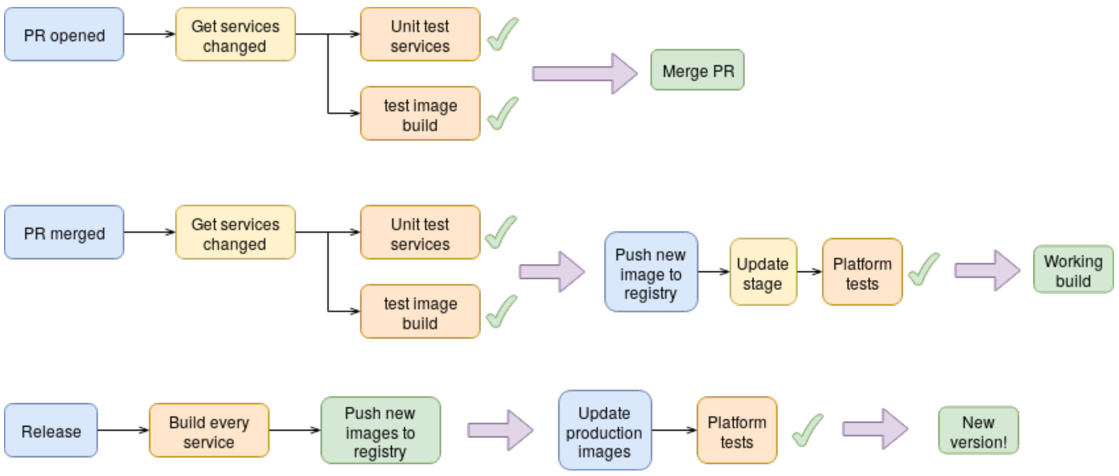


FIGURE 4.9: Three automated pipeline events

Reaching a perfectly adapted pipeline is a very tedious process and it's outside of the reach of this work, but the solution designed sets a base for future improvements. Even though the CD tool helps integrating with a GitHub repository, a lot of the logic behind the pipeline still needs to be implemented by the developer. The scripts developed for the pipeline are available in Caliopen's GitHub respository [10].

The introduction of this section anticipated the cost simplicity would have on the capacities of a continuous delivery solution, and truth is, the choice of Drone.io may not have perfectly fit the project. While it has provided an easy way to implement a CD pipeline for starters, we have rapidly found its limitations, specially when integrating it into a monorepository. We knew from the beginning that it was a product in alpha, as its documentation showed. Nevertheless, we were still hopeful with the advancement of the project, and expected it to develop in the months to come. Only time will tell if Drone gets to a point where it is more suited for our needs or other solutions will end up replacing it. The last section of this chapter will present all the planned improvements that can be made to the new platform.

4.6 Future Improvements

The architecture presented provides high availabilty, fault-tolerance and easy scalability for Caliopen's services, absent in the old architecture. It also makes a better use of a machine's capacity, sharing resources much more effectively thanks to containers.

[10]`https://git.io/fAxqF`

Nevertheless, it introduces new points of failure that need to be taken care of. Our architecture consists of a unique master node that represents a single point of failure for the cluster's control plane, thankfully remediable adding extra master nodes to the cluster. The loadbalancer is also a SPOF and being the only entrypoint for users, downtime on this machine implies the Caliopen application is inaccessible. A classic architectural solution could be having a second passive load balancer that goes live when the original fails.

Leaving behind failure management, the cluster is in a elemental state. The Kubernetes cluster still hasn't been integrated into the logging nor the monitoring platform and services such as storage or the mail server remain outside of the cluster. Storage in a Kubernetes cluster is a vast subject that could not be addressed in the scope of this work and requires further studies and preparation. A public SMTP server also shows some limitations when deployed behind Kubernetes network abstractions that the project can't currently resolve.

Chapter 5

Conclusion

The introduction of a Kubernetes cluster has allowed the project to move away from some limitations the original platform presented, notably resource utilization. It doesn't end there as the new platform introduces new features such as high availability or easier management. The new environment, together with the newly introduced CD tool, Drone, will serve as a base for future developments. Drone defines a starting point for building a continuous deployment pipeline, still limited by the lack of platform tests, but in the path for automating the process of delivery.

The new technologies introduced still have lots of room to improve. Most notably, many services used in the Caliopen platform are still not migrated to the Kubernetes cluster, such as storage or the mail server. Migrating the storage cluster to Kubernetes presents many challenges that require study before integration, and have been left out of the scope of this work. There are also some limiting points to the automation of the release process that won't be solved exclusively by Kubernetes and Drone.io, for example, automatic data and index migration between versions of the application is currently impossible. Nevertheless, the ambition remains to move every service to the Kubernetes cluster, simplifying by a huge amount the management of the cluster and reducing availability or scalability problems.

Unfortunately, the internship has ended before the studied platform could be completely put in production, and validation and testing of the new Cluster are still pending. Hopefully, I will be able to close this chapter in the months to come.

This internship has been a first experience collaborating on an Open Source project, an invaluable help to enter fascinating communities. Working on a multidisciplinary team has provided a priceless experience on communication, thanks to the limited size, the independence of the team and the close, day to day, relationship established with the members of Caliopen. Such practical and technical work has proven the need for an abstract, technology-independent vision when affronting unknown situations, it has changed my personal view on facing new challenges.

Bibliography

[1] Jérôme Petazzoni. Anatomy of a container: Namespaces, cgroups & some filesystem magic. 2015.

[2] Karl Isenberg. Iaas vs caas vs paas vs faas: Choosing the right platform, 2017. URL `https://mesosphere.com/blog/iaas-vs-caas-vs-paas-vs-faas/`. Last Accessed: September 2nd, 2018.

[3] Andrew Powell-Morse. Waterfall model: What is it and when should you use it? *Airbrake Blog*, December 2016. URL `https://airbrake.io/blog/sdlc/waterfall-model`. Last Accessed: August 22nd, 2018.

[4] Ante Gulam. Integrating crowsourced information security into agile sdlc. 2016. URL `https://blog.cobalt.io/`. Last Accessed: August 23rd, 2018.

[5] Carl caum. Continuous delivery vs. continuous deployment: What's the diff? *Puppet Blog*, August 2013. URL `https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff`. Last Accessed: August 22nd, 2018.

[6] J. E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5): 32–38, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.173.

[7] Hasan Fayyad, Luc Perneel, and Martin Timmerman. Full and para-virtualization with xen: A performance comparison. *Journal of Emerging Trends in Computing and Information Sciences*, Volume 10:719–727, 09 2013.

[8] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, May 2007. doi: 10.1109/INM.2007.374776.

[9] T. Wo, Q. Sun, B. Li, and C. Hu. Overbooking-based resource allocation in virtualized data center. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 142–149, April 2012. doi: 10.1109/ISORCW.2012.34.

[10] Jin Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 630–637, June 2009. doi: 10.1109/INM.2009.5188871.

[11] D. Hoeflin and P. Reeser. Quantifying the performance impact of overbooking virtualized resources. In *2012 IEEE International Conference on Communications (ICC)*, pages 5523–5527, June 2012. doi: 10.1109/ICC.2012.6364669.

[12] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1052676.1052707`.

[13] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015. doi: 10.1109/ISPASS.2015.7095802.

[15] A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, March 2015. doi: 10.1109/ICACEA.2015.7164727.

[16] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111): 2, 2014.

[17] E. W. Biederman. Multiple instances of the global linux namespaces. In *2006 Ottawa Linux Symposium*, pages 102–112, 2006.

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, January 2016. ISSN 1542-7730. doi: 10.1145/2898442.2898444. URL `http://doi.acm.org/10.1145/2898442.2898444`.

[19] Jason Geffner. Virtualized environment neglected operations manipulation. URL `http://venom.crowdstrike.com`. Last Accessed: September 4th, 2018.

[20] Jérôme Petazzoni. Docker, linux containers (lxc), and security. 2014.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[22] jbeda. Kubernetes. `https://github.com/kubernetes/kubernetes/`, 2014.

[23] Thomas Claburn. Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity, November 2017. URL `https://www.theregister.co.uk/2017/11/06/coreos_kubernetes_v_world/`. Last Accessed: September 3rd, 2018.

[24] Ron Miller. 36 companies agree to a kubernetes certification standard, 2017. URL `http://tcrn.ch/2mlZMb0`. Last Accessed: September 2nd, 2018.

[25] U.S. Department of Commerce. Final version of nist cloud computing definition published, 2011. URL `https://www.nist.gov/news-events/news/2011/10/final-version-nist-cloud-computing-definition-published`. Last Accessed: September 2nd, 2018.

[26] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 169–173, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103008. URL `http://doi.acm.org/10.1145/3102980.3103008`.

[27] Dr. Winston W. Royce. Managing the development of large software systems. *Proceedings, IEEE WESCON*, August 1970.

[28] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, Feb 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312940.

[29] Mike Beedle et al. Manifesto for agile software development. September 2001.

[30] David J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.

[31] Kief. The conflict between continuous delivery and traditional agile. January 2012. URL `http://kief.com/the-conflict-between-continuous-delivery-and-traditional-agile.html`. Last Accessed: August 22nd, 2018.

[32] CaliOpen. Caliopen github repository. `https://github.com/CaliOpen/`.

[33] Mark Betz. Understanding kubernetes networking: ingress. URL `https://goo.gl/EEGzbE`. Last Accessed: September 22nd, 2018.