Mémoire de Projet de Fin d'Études

Pour l'Obtention du Titre

D'Ingénieur d'État en Informatique

Option

Ingénierie des Télécommunications et Réseaux

Sujet

Conception et implémentation d'une architecture Cloud à base de conteneurs.

Soutenu par :

M. Ahmed MAJIDI

M. Ahmed ELGHAZAL

Sous la direction de :

M. Abdelaziz DOUKKALI SDIGUI (ENSIAS)
M. Aminearrahmane ACHARGUI (Sayoo)

Année Universitaire 2014 – 2015

سُبْحَانَكَ لاَ عِلْمَ لَنَا إِلاَّ مَا عَلَّمْتَنَا ۖ إِنَّكَ أَنْتَ الْعَلِيمُ الْحَكِيمُ

Dédicace

À mes très chers parents
Pour votre amour, vos encouragements, vos sacrifices et vos prières
Aucun mot ne saura exprimer mes sentiments d'amour, de respect et de gratitude envers vous ;
À mes chers frères et sœurs
Pour tout ce que vous avez fait pour moi, Je vous souhaite une vie pleine de
bonheur et de succès ;
À mes chers cousins
Pour votre soutien, vos encouragements et votre confiance en moi ;
À mes chères amies
Pour votre gentillesse, votre compréhension, pour les beaux moments qu'on a passé ensemble, ...bref pour votre amitié ;
À toute ma grande famille ;
À mon binôme Ahmed ;
À toute personne qui m'est chère ;
À tous ceux qui m'aiment ;
À tous les musulmans ;
À ALLAH avant tout.

Un grand Merci à vous.

Majidi

À ALLAH avant tout

À ma très chère maman

Aucune dédicace ne saurait exprimer l'amour, l'estime, le dévouement et le respect que j'ai pour toi.

Rien au monde ne vaut les efforts fournis jour et nuit pour mon éducation et mon bien être.

Ce travail est le fruit de tes sacrifices que tu as fait pour mon éducation et ma formation. « Je t'aime très fort »

À mon frère Titif

tu étais toujours à mes côtés pour me soutenir et me motiver. Je suis fière de t'avoir.

Tu as participé dans mon succès par tes encouragements. Un grand merci de tout mon cœur.

À mes chers tantes, oncles et toute ma grande famille

Pour votre gentillesse, votre soutien, vos encouragements et votre confiance en moi.

À mes chers co-chambres Fatih et Bachiri

Merci pour les moments partagés. Nous formons sans doute une famille.

À mes chers amis

J'ai passé avec vous de bons moments de ma vie, je vous aime.

Elghazal

Remerciements

Au terme de ce travail, nous tenons à exprimer notre profonde gratitude et nos sincères remerciements pour tous ceux qui nous ont aidés de près ou de loin pendant la durée de notre stage.

Nous tenons également à remercier Monsieur Abdelaziz DOUKKALI SDIGUI, professeur à l'ENSIAS pour son encadrement, son enthousiasme vis-à-vis notre projet et ses encouragements. Nos sincères remerciements sont à adresser à M. Mohamed OUHAMMI et M. Aminearrahmane ACHARGUI qui nous ont fait l'honneur de nous encadrer tout au long de ce travail. Nous leurs en sommes très reconnaissants pour ses fructueux conseils et précieuses directives, et pour le grand souci qu'ils portent à l'égard de notre sujet.

Remerciements spéciaux à tout le corps professoral de l'ENSIAS, pour la formation de qualité qu'ils nous ont prodiguée durant ces trois années. Que tous ceux et celles qui ont contribué de près ou de loin à l'accomplissement de ce travail trouvent l'expression de nos remerciements les plus chaleureux.

Résumé

Le présent rapport constitue le fruit d'un travail de quatre mois, réalisé dans le cadre de notre projet de fin d'études, et effectué au sein de *Sayoo*. Le projet a pour objectif de mettre en place une architecture Cloud basée sur les conteneurs.

Il s'agit en effet d'étudier, puis réaliser une architecture qui va permettre la gestion automatisée d'un service propre à la société . Pour arriver à cette fin, nous avons pour mission d'étudier les dernières technologies dans le domaine du Cloud pour mieux aborder la phase de conception et déploiement. Pour bien mener ce projet, nous avons choisi de définir les notions fondamentales et nécessaires pour la compréhension du projet notamment dans le domaine du Cloud.

Durant ce projet, nous avions pour mission dans un premier temps de cerner le sujet, étudier sa faisabilité et définir le cahier de charges, ainsi que rédiger le dossier de spécifications fonctionnelles aussi bien que techniques. Ensuite nous avons entamé l'analyse approfondie et la conception de notre projet, nous avons par la suite élaboré plusieurs architectures intermédiaires. Finalement nous avons passé à l'implémentation, le test et le déploiement de l'architecture.

Mots clés : Cloud, Cluster, Scalabilité, Conteneur, Docker, PaaS, SaaS, Odoo, Deis.

Abstract

This submission is the result of our work at *Sayoo*, conducted as part of our graduation project. The purpose of our project is to develop a container-based cloud architecture.

In fact, this project aims to reach an architecture that will enable automated management of the company's service. Then, our mission is to study the latest cloud technologies to get a better knowledge of further parts. In order to have a clear view of this project, we've defined several concepts and aspects of the cloud and its technologies.

Initially in this project, we had to identify the issue then define the functional and technical specifications. After, we've focused on the analysis and design parts. We've developed several intermediate architectures that finally lead us to the appropriate architecture.

Mots clés : cloud, Cluster, Scalability, Container, Docker, PaaS, SaaS, Odoo, Deis.

Table des figures

Liste des tableaux

Liste des abréviations

API Application Programming Interface. 36, 62, 65
AWS Amazon Web Services. 57, 61, 79

CPU Central processing unit. 36, 66

DNS Domain Name System. 20, 61

ERP Enterprise Resource Planning. 15, 18–20

GCE Google Cloud Engine. 61, 63, 79

HTTP HyperText Transfer Protocol. 62, 65
HTTPS HyperText Transfer Protocol Secure. 9, 62–64

IaaS Infrasturcture As a Service. 36, 79
IP Internet Protocol. 57, 68

LXC Linux Containers. 33

PaaS Platform As a Service. 13, 59, 61, 64, 68, 79
PKI Public Key Infrastructure. 63

SaaS Software As a Service. 49, 52, 69, 76, 79, 80
SLA Service level agreement. 20
SMTP Simple Mail Transfer Protocol. 77
SSH Secure Shell. 46
SSL Secure Socket Layer. 13, 62

TLS Transport Layer Security. 13, 62

URL Uniform Resource Locator. 53

YAML YAML Ain't Markup Language. 40, 75

Table des matières

Introduction

Toutes les entreprises fournissent des services à leurs clients, évidemment avec l'objectif de générer des bénéfices ainsi qu'une gestion efficace des ressources. En effet, l'entreprise doit tracer une stratégie efficace au court, moyen et long terme pour améliorer cette gestion et promouvoir sa relation avec ses clients et partenaires. Le concept de planification des ressources de l'entreprise (*ERP*) est une réponse à ce besoin car il met en place un système modulaire englobant l'ensemble du système d'informations (SI). Cependant les *ERP* ne recouvrent pas entièrement les besoins, c'est pourquoi certaines entreprises se penchent sur le développement personnalisé de nouveaux modules.

Dans cette optique, la société Sayoo a décidé de s'investir dans le développement des modules *ERP* ainsi que leurs déploiements. Ce service a pour objectif de faciliter la gestion et le suivi des ressources de l'entreprise ainsi que l'évolution de leurs relations avec les clients, en automatisant plusieurs processus métier. Sayoo a décidé d'améliorer ce service. Dans ce cadre, notre stage de fin d'étude a donc pour objectif de mettre en œuvre une architecture Cloud basée sur les conteneurs qui facilitera à Sayoo la provision, l'administration et le déploiement des instances *ERP*.

Le présent mémoire expose les résultats de notre travail durant notre stage. Il comporte cinq chapitres organisés comme ceci :

— L'objet du premier chapitre est de cerner le projet dans son contexte général, à travers la présentation de la société *Sayoo*, ensuite décrire la motivation du projet et les objectifs visés. La dernière section du chapitre sera consacrée à la conduite et la planification du projet.

— le deuxième chapitre est consacré à la définition du Cloud computing et l'explication des différents concepts ainsi que l'introduction de la notion de containérisation.

— Le troisième chapitre s'attaque à l'orchestration des applications distribuées qui

présentera les différentes technologies permettant une gestion multi-conteneurs ainsi qu'un benchmark à la fin du chapitre.

— Au niveau du quatrième chapitre, nous expliciterons les phases d'analyse et de conception de l'architecture. Nous exposerons plusieurs architectures intermédiaires pour modéliser les fonctionnalités réalisées.

— Le cinquième et dernier chapitre décrit d'une façon détaillée, la phase de la mise en œuvre du projet, à travers quelques interfaces réalisées.

Le mémoire se termine par une conclusion générale qui présente le bilan du travail réalisé et ses principales perceptives.

CHAPITRE 1

Contexte général du projet

Le présent chapitre permet de situer le projet dans son contexte général, à travers la présentation de l'organisme d'accueil *Sayoo* et des différents services qu'il offre. Nous exposerons aussi les motivations, les objectifs visés ainsi que la planification du projet.

1.1 Présentation de l'organisme d'accueil

Sayoo est une Start-up qui agit dans le domaine des technologies de l'information. Lancée en décembre 2013, elle vise à fournir aux clients les solutions adéquates à leurs besoins en utilisant les dernières technologies. *Sayoo* repose principalement sur l'open-source afin de fournir leurs services à prix abordable et bénéficier des communautés actives.

1.1.1 Services

La société *Sayoo* fournit des services de qualité qui touchent plusieurs domaines.

Développement logiciel *Sayoo* utilise des outils et frameworks performants et récents pour mieux gérer les projets et gagner en termes de coût et de temps.

Développement Web/Mobile Le développement mobile tient une place importante au sein de l'entreprise parce que le marché mobile est en constante progression en offres et en demandes.

Conception des design *Sayoo* ne se focalise pas totalement sur l'aspect technique. la société conçoit et délivre des interfaces réactives et séduisantes pour les clients.

Intégration des ERP et formations *Sayoo* se penche aussi sur la gestion des entreprises via Odoo (anciennement Open ERP). Ce service tient une place importante pour la société notamment dans sa personnalisation.

1.2 Cadre du projet

1.2.1 Présentation du projet

la Société *Sayoo* tient à fournir des services de qualité à ses clients. Le projet consiste à améliorer le service *ERP* fournit pour les entreprises.

Le service *ERP* tient une importance capitale au sein de la société parce qu'il déploie des ressources importantes. Par ailleurs, toute relation avec une entreprise nécessite une rigueur et une grande attention.

Un client qui désire gérer les ressources de son entreprise doit rencontrer un responsable de la clientèle de *Sayoo* pour spécifier ses besoins. Le responsable propose une version

personnalisée de la suite *Odoo* (anciennement *Open ERP*) au client pour une gestion efficace. Par la suite, la société qui s'est déjà procurée 3 différents serveurs (Développement, Évaluation, Production) doit déployer une version d'*Odoo* sur le serveur de développement pour que les développeurs effectuent les premières personnalisations. Le client doit avoir accès au serveur Développement pour confirmer ou refuser. En cas de confirmation, l'administration doit déployer la version souhaitée sur le serveur production et conserver la version d'évaluation au cas d'une nouvelle mise à jour ou maintenance. Le serveur Évaluation a pour but de faire découvrir le client une version standard ou améliorée de la suite *Odoo*.

Pour une meilleure fiabilité, la société se procure 2 serveurs (Développement, Production) par client. elle doit périodiquement effectuer des maintenances pour assurer la disponibilité de son service. Chaque client de la société se différencie seulement par sa personnalisation de la suite *Odoo* donc les étapes de configuration sont redondantes pour tous les clients. Le temps de configuration et de déploiement reste trop important donc le client doit attendre pour accéder à son service après chaque maintenance. Les pré-requis de ce service sont relativement coûteux et affecte par conséquent le prix final du client.

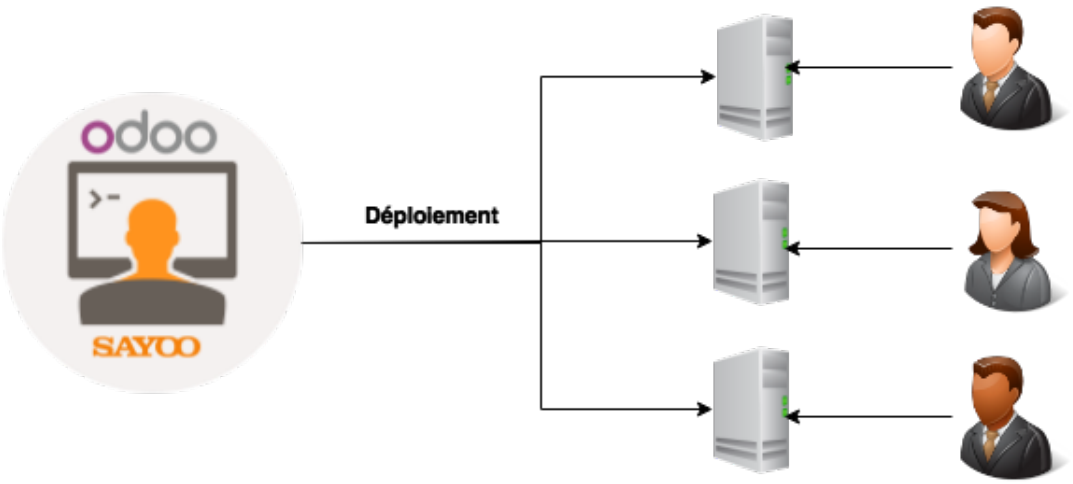

FIGURE 1.1 – Processus actuel de déploiement des services Odoo

1.2.2 Motivations

Le processus de la provision des applications ERP pour les clients n'est pas optimale. D'où les motivations d'opter vers une solution Cloud. Voici une liste des problèmes que rencontrent *Sayoo* lors du processus du déploiement actuel.

— La provision des instances *Odoo* est complètement manuelle. Quand un client de-

mande une version de test d'*Odoo* par exemple, l'équipe de *Sayoo* est contrainte d'exécuter une longue série d'étapes répétitives : la commande, l'installation et la configuration du serveur, la mise en place de l'application *Odoo*, la configuration DNS, etc...

— Le besoin récurrent des administrateurs qui doivent garder en permanence l'œil sur les applications des clients.

— La supervision est manuelle. En effet, au fur et à mesure que le nombre de clients augmente, l'administrateur doit ajouter les applications au système de supervision. Ce processus devient faible et non évolutif. Par conséquent, le suivi est pénible en termes de ressources et de temps.

— L'absence d'automatisme fait que l'équipe *Sayoo* consacre la plupart du temps aux tâches répétitives au lieu de se consacrer au développement et la personnalisation des applications *Odoo*.

— La nécessité d'acheter un serveur pour chaque client pour y installer son application, ceci est un "*overkill*" pour atteindre le but. En conséquence, cela affecte le coût final de la solution.

— L'absence d'une architecture robuste qui peut garantir des exigences non fonctionnelles énormes, à savoir la configuration automatique, la haute disponibilité, la sécurité et la scalabilité. Ainsi, Sayoo ne garantie pas un SLA à ses clients.

1.3 Objectifs du projet

Ce projet vise principalement à améliorer le service *ERP* au sein de *Sayoo* par l'automatisation de certaines tâches, notamment la configuration et le déploiement. L'objectif de ce projet est de déployer une architecture qui se caractérise par une haute disponibilité, une scalabilité et une sécurité. Par conséquent, cela revient à déployer une architecture *Cloud* qui répond aux exigences suivantes.

— Faciliter la configuration et le déploiement des instances *Odoo* pour les clients.

— Éviter les tâches répétitives et faciliter la personnalisation d'*Odoo*.

— Réduire le coût en termes de temps et d'argent pour *Sayoo*.

— Inclure un mécanisme de supervision fiable et évolutif.

— Gérer aisément plusieurs clients en même temps.

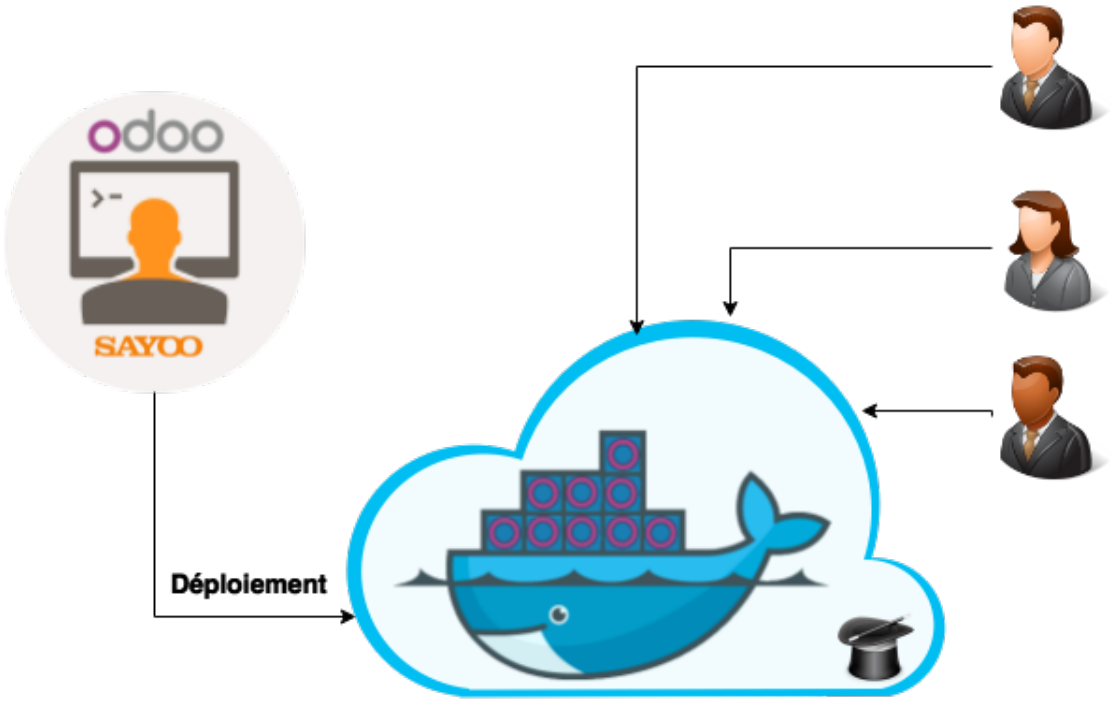

FIGURE 1.2 – Processus souhaité de déploiement des services Odoo

1.4 Planification du projet

Notre projet est loin d'être classique. En effet, il fait appel aux technologies non abordées le long de notre parcours scolaire. Du coup, une planification rigoureuse s'est imposée pour prévoir le déroulement du projet. Grâce aux réunions tenues avec les encadrants au sein de *Sayoo*, nous avons été éclairés sur les différentes étapes du projet ainsi que leurs séquencements. Le projet est partagé en trois grandes étapes : la première est une phase de documentation dont les objectifs est de bien assimiler les nouveaux concepts concernant le Cloud Computing, d'autre part de délimiter le périmètre du projet au niveau fonctionnel qu'au niveau technique. La seconde partie est consacrée à la conception de la solution, quant à la troisième étape, elle traite la mise en œuvre de la solution à travers la réalisation, les tests, et le déploiement.

Le stage a débuté le 23 mars pour une durée de 4 mois. Il en résulte le planning suivant :

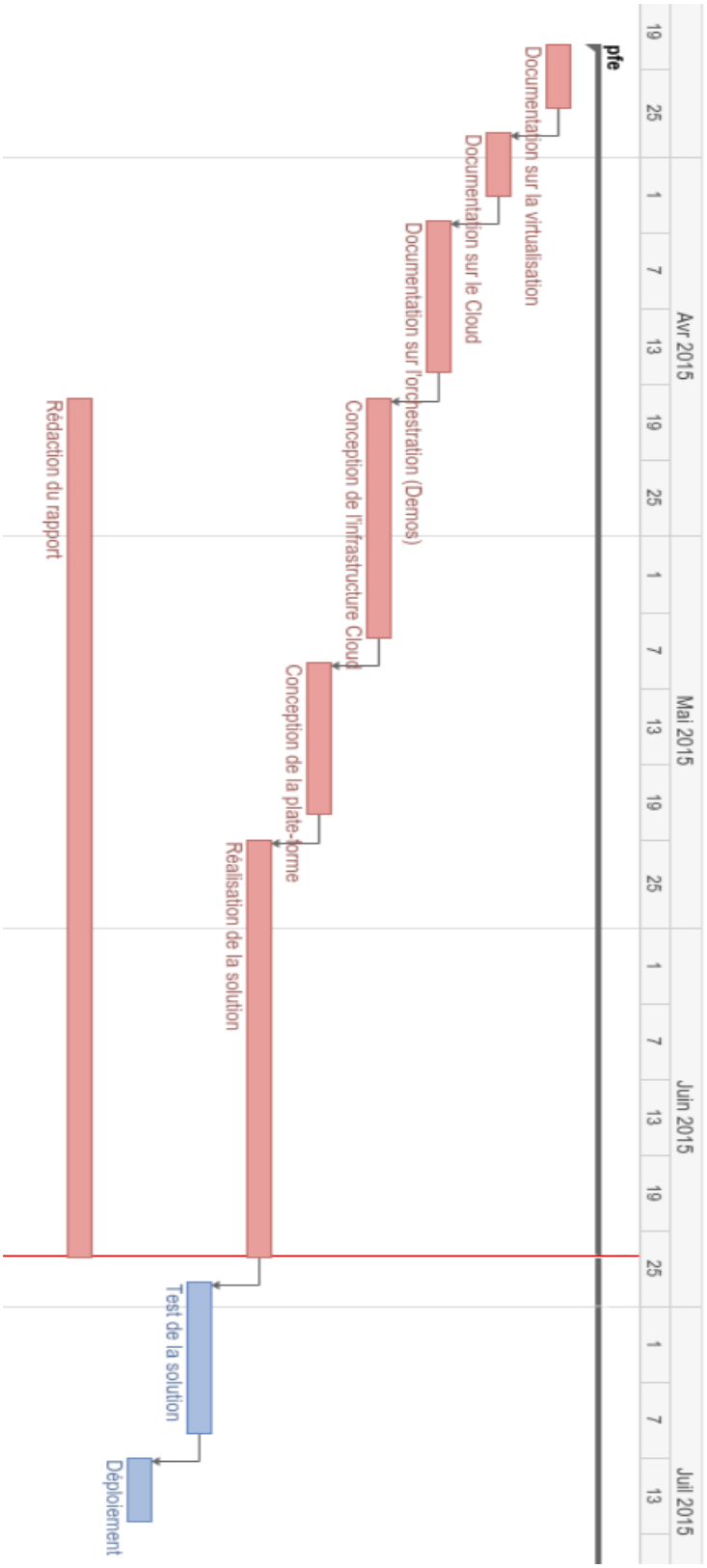

FIGURE 1.3 – Diagramme de Gantt

CHAPITRE 2

Le cloud computing

Le but de ce chapitre est d'exposer le travail fait au niveau de la documentation. Ce chapitre est très riche en concepts incontournables et nécessaires à la suite de ce rapport. En effet, nous allons commencer par expliquer concrètement la signification du Cloud. Ensuite, nous présenterons les types de Cloud les plus classiques, à savoir IaaS, PaaS et SaaS. Enfin, nous verrons les notions de virtualisation et containérisation ainsi que la différence entre les deux.

2.1 Le Cloud Computing [1]

2.1.1 Définition

Le **Cloud Computing** est l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'intermédiaire du réseau internet. Ces serveurs sont loués à la demande selon des critères techniques (Bande passante, puissance, etc.). Le Cloud Computing se caractérise par sa grande souplesse. En effet, il est destiné et ouvert à tous les utilisateurs.

Le Cloud est rendu possible grâce à la virtualisation, l'ubiquité des réseaux à grande vitesse, les capacités des navigateurs actuels et l'évolution des piles de développement Web. Avec ces choses en place, il devient moins nécessaire de posséder sa propre infrastructure, ou même de posséder son propre logiciel. Il est possible de subvenir à ses besoins à partir du Cloud Computing.

2.1.2 Caractéristiques

Le Cloud donne la capacité aux utilisateurs finaux d'utiliser des pièces de ressources, elles doivent être acquises rapidement et facilement. NIST définit plusieurs caractéristiques qu'il juge essentiel pour qu'un service soit considéré comme «Cloud». Ces caractéristiques comprennent les points suivants :

— **Service à la demande** : La capacité pour un utilisateur final de s'inscrire et recevoir des services sans les longs délais qui ont caractérisé l'informatique traditionnelle.
— **Accessible au réseau large** : La capacité d'accéder au service via les plates-formes standard (bureau, ordinateur portable, mobiles, etc.).
— **La mise en commun des ressources** : Les fournisseurs servent plusieurs clients ou «locataires» avec des services provisoires et évolutifs. Ces services peuvent être ajustés pour répondre aux besoins de chaque client, sans aucune modification apparente pour l'utilisateur.
— **Rapide élasticité** : La capacité des ressources doit évoluer pour faire face aux pics de la demande.
— **Service mesuré** : La facturation est mesurée et livrée.

Sans ces caractéristiques, l'informatique en nuage n'apporte rien par rapport à l'informatique traditionnelle. Une solution Cloud doit comporter ces caractéristiques. Par

conséquent, notre projet se doit de satisfaire les critères du Cloud. Dans ce qui suit, nous détaillerons les types classiques du Cloud.

2.1.3 Les types de Cloud

Le Cloud Computing est un concept qui décrit une large collection de services. Dans ce rapport, nous allons exposer les différents types de services de Cloud communément appelé *Software as a Service* (SaaS), *Platform as a Service* (PaaS) et *Infrastructure as a Service* (IaaS) et décrire plus en détails ceux qui touchent à notre sujet.

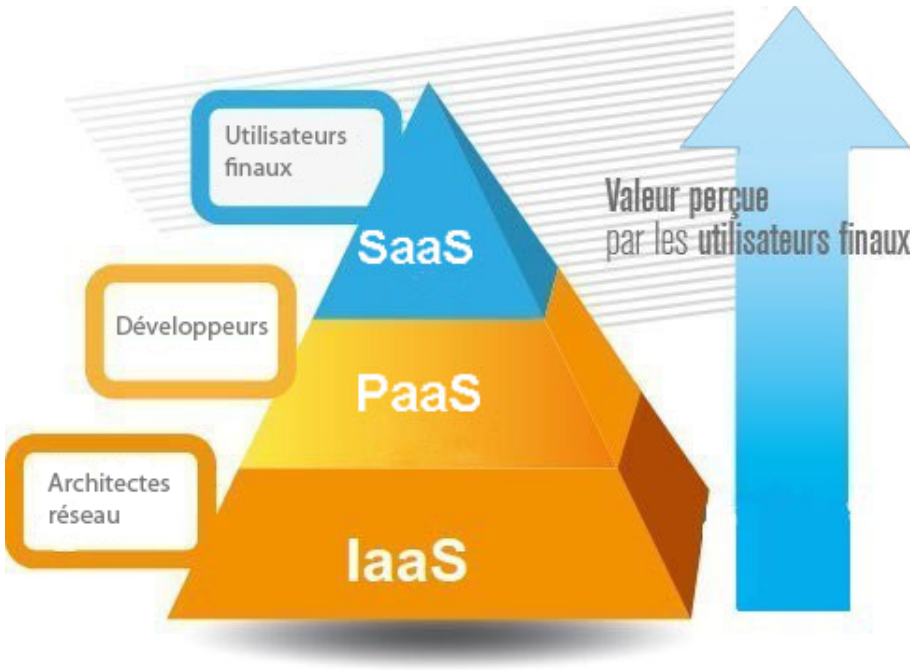

FIGURE 2.1 – Pyramide des services Cloud

Software as a Service

La meilleure façon de comprendre ces services est de commencer avec le SaaS, la couche la plus abstraite et celle qu'on utilise peut-être déjà aujourd'hui, même à un niveau personnel. Un exemple simple de SaaS est un service de messagerie en ligne, comme Gmail. Lorsqu'un client utilise Gmail, il n'héberge pas son propre serveur de messagerie. C'est Google qui l'héberge, et l'accès est simplement via un navigateur comme un client.

SaaS est orienté vers les utilisateurs finaux de l'entreprise et ne nécessite pas beaucoup de compétences pour l'utiliser. Le fournisseur décide sur le nombre de ressources

à consacrer pour l'utilisation de l'application, les serveurs, les machines virtuelles, et les équipements réseaux. Enfin, il suffit de pointer le navigateur à l'application.

Infrastructure as a Service

IaaS est à l'autre bout du pyramide du Cloud. Lorsque l'on souhaite garder le contrôle de l'environnement logiciel, mais on ne veut pas maintenir aucun équipement ni acheter des serveurs et les mettre dans une pièce à température contrôlée. On consulte une offre IaaS d'un fournisseur IaaS pour commander et acheter tout simplement une machine virtuelle.

On peut mettre n'importe quel logiciel que l'on souhaite au-dessus d'IaaS. En arrière plan, le fournisseur fournit les ressources nécessaires pour satisfaire le besoin. Ceci est rendu plus facile avec les technologies de virtualisation, qui séparent les ressources physiques de la machine virtuelle qui exécute le logiciel. IaaS est disponible sur Amazon EC2, GCE de Google et bien d'autres.

L'infrastructure en tant que service ou l'IaaS ne touche pas directement à notre sujet. Du coup, il ne sera mentionné que rarement par la suite.

Platform as a Service

PaaS se situe entre IaaS et SaaS. Il n'est pas un produit fini comme SaaS, encore moins une simple ressource virtuelle vierge comme IaaS. PaaS est destiné pour les développeurs, il leur donne des outils et des interfaces de haut niveau pour mieux développer. Par exemple, Windows Azure de Microsoft vous donne des outils pour développer des applications mobiles, des applications sociales, sites Web, jeux et plus encore. Vous construisez ces choses, mais vous utilisez les API et les outils pour les accrocher et exécuter dans l'environnement Azure.

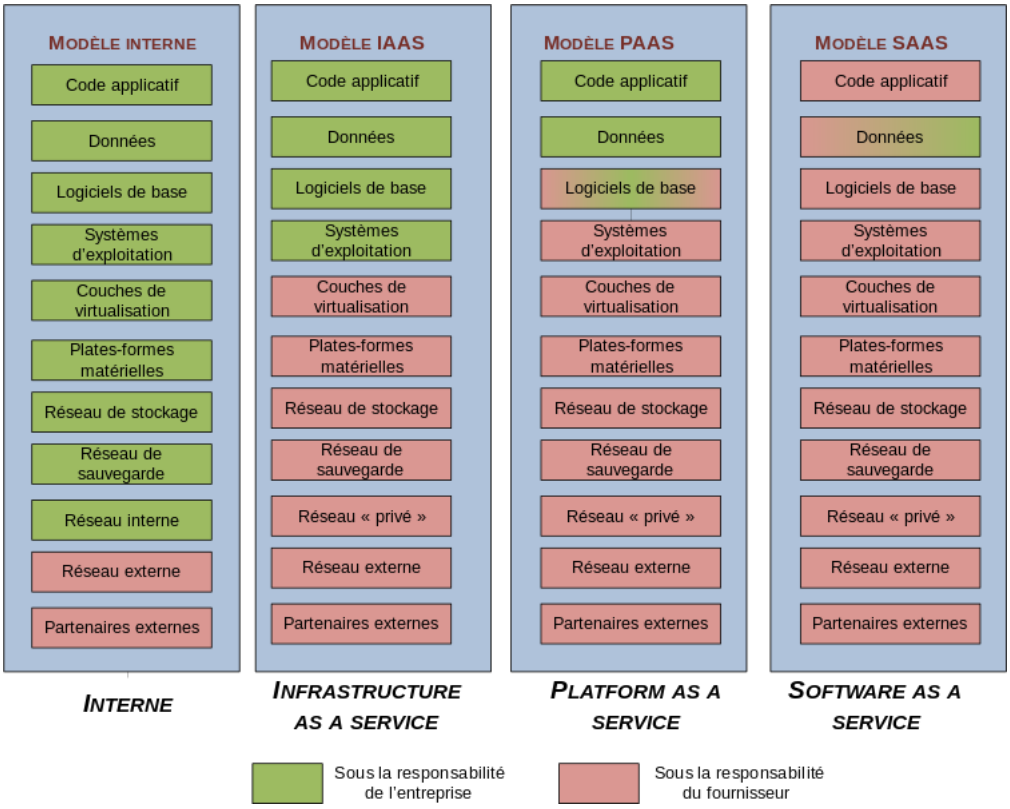

FIGURE 2.2 – L'éxternalisation de l'informatique en Cloud [2]

2.2 Virtualisation ou containérisation

2.2.1 Virtualisation

La virtualisation sert à partitionner un seul serveur physique en plusieurs machines virtuelles comme un espace de stockage ou un réseau. Elle permet une consolidation de serveurs avec une grande souplesse d'utilisation. Dans le contexte de l'informatique en nuage, la virtualisation est importante pour la mise en service et le retrait rapide de serveurs. Par ailleurs, la scalabilité est la principale caractéristique du Cloud moderne. Ainsi la virtualisation a permis la création de machines virtuelles, la montée en charge à la demande, la migration à chaud, etc. La virtualisation est une technologie permettant d'arriver à une utilisation rentable des serveurs tout en prenant en charge la séparation entre de multiples locataires d'un matériel physique.

Il existe plusieurs solutions de virtualisation, l'on peut citer à titre informatif :

— **VMware** : Cette solution, séduisante, pose certains problèmes. En effet, il propose des outils propriétaires et incompatible avec le noyau Linux.

— **Xen** : C'est un hyperviseur libre de machines virtuelles, il est considéré comme un outil de virtualisation des plus performants.
— **KVM** :, Kernel-based Virtual Machine est devenue rapidement la solution de virtualisation de référence pour Linux. Elle est basée sur les architectures Intel ou les architectures AMD.
— **QEMU** : C'est un émulateur machine générique. Il est une solution de virtualisation à utiliser si le processeur de système hôte ne possède pas d'extension matérielle spécifique à la virtualisation.

2.2.2 Containérisation

Le noyau Linux permet de lancer plusieurs instances isolées de l'espace utilisateur. Un conteneur est un environnement isolé où un ou plusieurs processus peuvent être exécutés. Les conteneurs se concentrent sur l'isolation des processus au lieu d'émuler une machine physique complète.

Historiquement, *chroot* dans le noyau Linux a fourni un certain niveau d'isolation en fournissant un environnement pour créer et héberger une copie virtualisée d'un logiciel, et ceci depuis le début des années 80. Mais le terme «conteneurs» n'est apparu que vers la fin de l'année 2006. Il a été renommé «*Control Groups*» (cgroups) pour éviter toute confusion causée par de multiples significations du terme «conteneurs» dans le noyau Linux. «*Control Groups*» est une fonctionnalité du noyau Linux qui est disponible depuis la version v2.6.24, elle limite et isole l'utilisation des ressources d'un ensemble de processus. Par la suite, l'isolation de l'espace de noms a été ajoutée.

Cela a conduit à l'évolution de Linux Containers (LXC), un environnement de virtualisation au niveau du système d'exploitation qui est construit sur les fonctionnalités du noyau Linux comme *chroot*, *cgroupes*, l'isolation de l'espace de noms, etc.

Cgroups

— Isolation de l'usage des ressources (CPU, mémoire, E/S, etc.).
— Limitation des ressources : un groupe peut être configuré pour ne pas dépasser une certaine limite de la mémoire.
— Priorité : certains groupes peuvent obtenir une plus grande part de CPU ou de débit E/S disque ;
— Mesure de l'usage des ressources.
— Contrôle : le gel des groupes ou des points de reprise et le redémarrage.

Namespaces

- Partitionnement des structures de Kernel pour créer des environnements virtuels
- Des espaces de noms différents
  - pid (processus)
  - net (interfaces réseaux, routage, ...)
  - ipc (communication inter-processus)
  - mnt (points de montage, système de fichiers)
  - uts (Nom de hôte)
  - user (UIDs)

Contrairement à la virtualisation, la liste des solutions de containérisations n'est pas aussi longue. La plupart d'eux se basent sur ou convergent vers *cgroups* et *namespaces*.

- **LXC** : Linux containers, il combine cgroups et namespace pour fournir un environnement isolé pour les applications ;
- **OpenVZ** : il permet de créer multiples conteneurs pour Linux. Dorénavant, tous les efforts des développeurs vont aller dans le sens de fusionner les fonctionnalités d'*OpenVZ* avec *LXC*.
- **lmctfy** : *Let Me Contain That For You* est une solution open source de containérisation de Google qui est basé sur *cgroups*. Le futur de *lmctfy* est flou vu que Google avait commencé à migrer les concepts de *lmctfy* vers Docker.
- **Docker** : ce n'est pas une autre solution, mais il est basé sur *LXC* et fournit une couche de haut niveau accessible pour l'utilisateur.

2.2.3 Etude comparative

Le choix de la solution a été proposé fortement par la société. On a opté pour une solution de containérisation (Docker) dans la mesure où un *Benchmark* prouve son efficacité et sa légèreté.

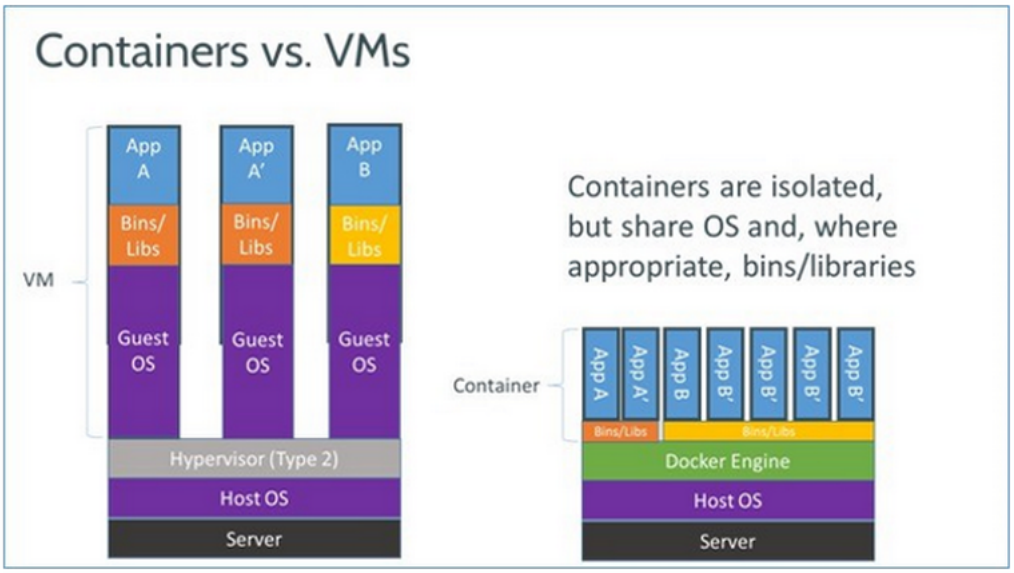

FIGURE 2.3 – Virtualisation VS Containérisation [3]

Dans la containérisation, ou la virtualisation au niveau du système d'exploitation, l'hôte et les invités partagent le même noyau. Cette approche réduit le gaspillage de ressources puisque chaque conteneur contient uniquement l'application, les bibliothèques et les binaires nécessaires. Le rôle de l'hyperviseur est assuré par un moteur léger de la containérisation comme Docker, qui est installé au-dessus du système d'exploitation hôte.

Le principal avantage des conteneurs c'est qu'ils sont libres de l'overhead du système d'exploitation contrairement aux machines virtuelles, ce qui leur rend considérablement plus légers, plus faciles à télécharger ainsi que plus rapides à lancer. Cet avantage permet également au serveur d'héberger potentiellement beaucoup plus de conteneurs que de machines virtuelles. Scalabilité, portabilité et facilité de déploiement sont quelques avantages.

Lors de cette étude, nous allons se baser sur un travail réalisé par IBM [4], où nous allons mettre l'accent sur les performances de Docker et *KVM*. Le *Benchmark* concerne 15 machines virtuelles simultanément en marche sous *OpenStack*.

Tests	Docker	KVM
Temps moyen de démarrage	3.9s	5.88s
Usage CPU démarrage (sys)	0.44%	2.08%
Usage CPU démarrage (usr)	1.14%	12.6%
Usage de mémoire au démarrage	45.8Mb/VM	185Mb/VM
Temps moyen de redémarrage	2.54s	124.45s
Usage CPU redémarrage (sys)	0.11%	0.19%
Usage CPU redémarrage (usr)	0.34%	0.82%
Usage de mémoire au redémarrage	57Mb	467Mb
Temps moyen de suppression	4.09s	4.45s
Temps moyen de Snapshot (VM to Image)	26.39s	42.93s
Usage CPU de Snapshot (sys)	0.11%	1.07%
Usage CPU de Snapshot (usr)	0.4%	1.58%
Usage de mémoire de Snapshot	48Mb	114Mb
Usage de mémoire état stable	3.52s	5.87s
Calcul des nombres premier jusqu'à 20000	15.11s	15.08s
Débit du réseaux (Mbits/s)	940.26	940.56

TABLE 2.1 – Performances : Docker VS KVM

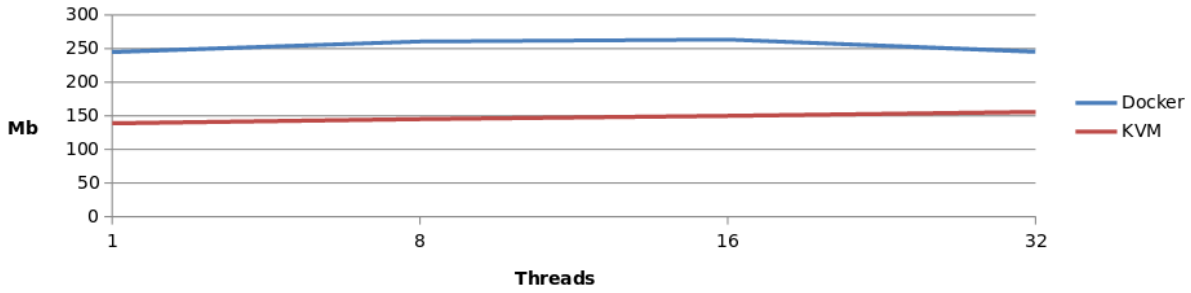

FIGURE 2.4 – Lecture du disque dur [4]

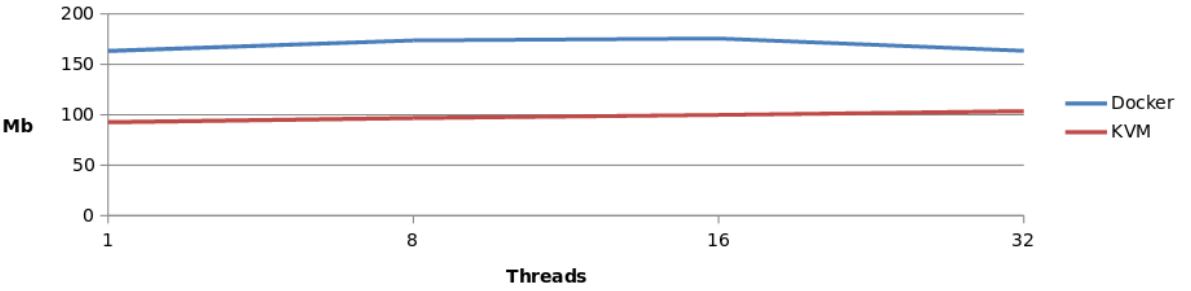

FIGURE 2.5 – Ecriture dans le disque dur [4]

En somme, il va s'en dire que Docker l'emporte largement par rapport à *KVM*, tant qu'en termes de la gestion des ressources (CPU, RAM) qu'en termes du temps consommé pour le démarrage, le redémarrage et la suppression. Par ailleurs, Docker est le meilleur dans la lecture/écriture dans le disque dur et s'approche à des performances natives. Ainsi, Il serait donc judicieux de virtualiser les bases de données dans des conteneurs que dans des machines virtuelles.

2.3 L'outil Docker

2.3.1 Présentation de Docker

Docker est un logiciel open source qui automatise le déploiement d'applications dans des conteneurs logiciels. C'est un outil qui peut empaqueter une application et ses dépendances dans un conteneur virtuel, qui pourra être exécuté sur n'importe quel serveur Linux ou Windows (*Boot2Docker*). En fait, il a pour objectif de **faciliter le déploiement**

d'applications, d'avoir plusieurs versions d'une même application sur les serveurs (phase de développement, tests), mais aussi d'**automatiser le packaging d'applications**. Avec Docker, on s'oriente vers de l'intégration et du déploiement en continu grâce au système de conteneur. De plus, Docker permet de garder son système de base propre, tout en installant de nouvelles fonctionnalités au sein de conteneurs, on part d'une base qui est le système d'exploitation et on ajoute différentes briques conteneurisées. Docker a été distribué en tant que projet open source à partir de mars 2013.

2.3.2 Utilisation de Docker

Docker propose de nombreux templates qui permettent de déployer des applications très rapidement en conteneur. La communauté est très active, ce qui permet aux utilisateurs de disposer de nombreux conteneurs applicatifs pré-faits. Docker est basé sur LXC qui est une référence sous Linux quant à l'utilisation des conteneurs. Par ailleurs, il intègre les éléments suivants :

- **Control Groups** : Fonctionnalité du noyau Linux pour limiter, compter et isoler les ressources (CPU, RAM, etc.) utilisées par un groupe de processus.
- **AppArmor et SElinux** : Gestion avancée des permissions aussi bien au niveau des applications qu'au niveau du système de fichiers.
- **Kernel namespace** : Fonctionnalité du noyau Linux qui permet l'isolation, afin de s'assurer qu'un conteneur ne puisse pas en affecter un autre.
- **chroot** : Fonctionnalité qui permet de changer la racine d'un processus, afin de l'isoler sur un système par mesure de sécurité.

Docker propose des services pour effectuer facilement différentes actions : créer, éditer, publier et exécuter des conteneurs. On parlera souvent des notions de dockerfiles, conteneurs et images.

- **DockerFile** : Fichier source qui contient les instructions, éléments à installer, c'est un fichier de configuration.
- **Image** : Compilation d'un fichier DockerFile pour former une image portable, prête à être déployée.
- **Conteneur** : Exécution d'une image, mise en conteneur d'une image.

Le schéma suivant décrit le fonctionnement de l'outil Docker :

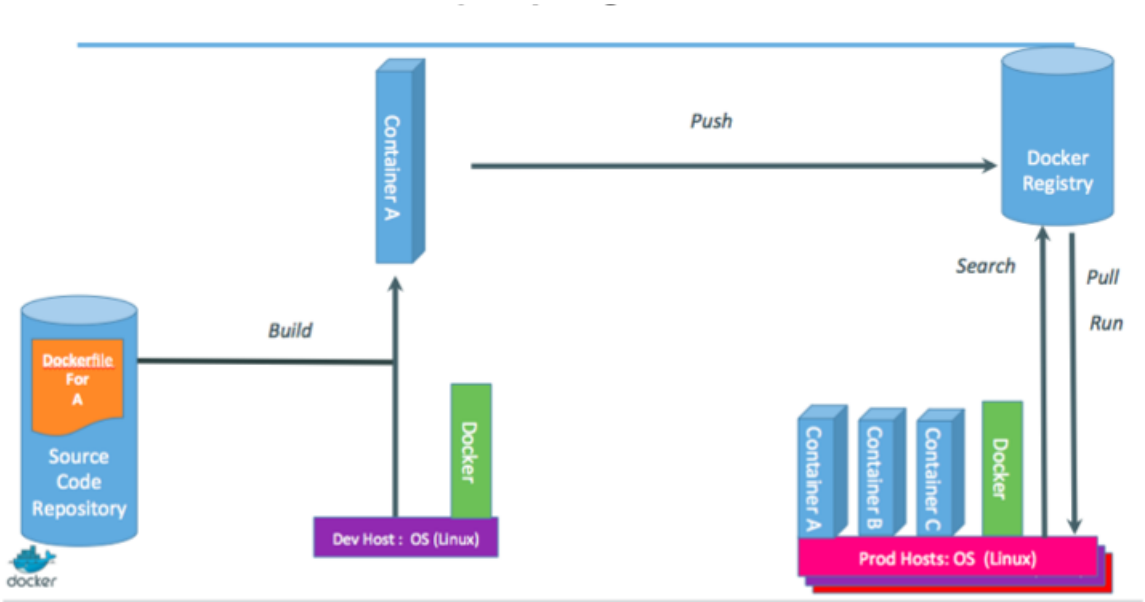

FIGURE 2.6 – Fonctionnement de Docker [5]

La figure 2.6 cache derrière elle toute la philosophie de Docker. Cette philosophie assume que toute application qui marche dans l'ordinateur du développeur doit marcher aussi dans les serveurs de production. En effet, le développeur, quand il aura fini de développer son application, va faire un *build* pour construire une image de son application. Enfin, il va faire un *push* pour pousser l'image vers un registre des images centralisé. Dans un environnement de production, une copie de l'application peut-être lancée en faisant un simple *pull* et *run*.

CHAPITRE 3

Orchestration des applications distribuées

Docker est un outil qui a révolutionné le monde informatique par l'introduction de la notion de conteneur, donc il est normal de chercher à l'utiliser en production. De nos jours, les développeurs utilisent de plus en plus Docker pour déployer des applications qui s'exécutent sur plusieurs conteneurs et plusieurs hôtes. Orchestrer ces applications distribuées nécessite une approche **multi-conteneur** et **multi-hôte** native avec une interface utilisateur et de l'outillage commun qui fonctionnent sur toutes les infrastructures. Plusieurs outils ont été présentés par de grandes entreprises et qui sont toujours en cours de développement. Dans ce chapitre, les projets prometteurs dans l'orchestration des applications distribuées seront exposés.

3.1 Le Clustering

Nous avons présenté dans le chapitre 2 le Cloud Computing ainsi que ses services classiques. Dans cette section, nous allons parler du Cluster comme un service et le situer dans le pyramide des services du Cloud (Figure 3.1) [6].

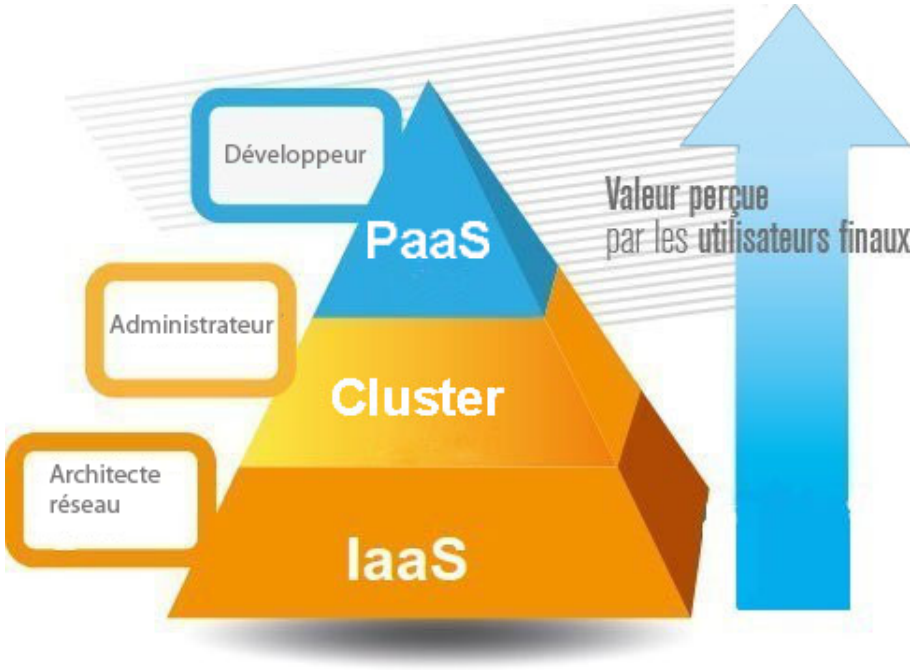

FIGURE 3.1 – Le Cluster dans le Cloud

— **Plate-forme** : Un développeur crée une application dans un ensemble particulier de langages de programmation à l'aide des APIs spécifiques et pousse ce code vers la plate-forme. La plate-forme l'exécute et lance une seule copie du service ou éventuellement plusieurs copies dépendamment du trafic. En effet, ce service monte en charge automatiquement.
— **IaaS** : A l'autre bout du pyramide se trouve l'IaaS composé seulement de machines virtuelles qui résident essentiellement dans l'internet. Quand un utilisateur consulte un fournisseur IaaS et demande un ordinateur avec des spécifications données (quantité de mémoire et de disque, nombre de CPU, une version spécifique Linux, etc.). En quelques minutes, L'utilisateur pourra exécuter n'importe quel logiciel sur le serveur mais il est entièrement responsable de la gestion de la machine.
— **Cluster** : C'est lorsqu'il y a un besoin de gérer un ensemble de machines comme **une seule entité**, un Cluster de machines.

	IaaS (VMs)	**Cluster**	**Plate-forme**
A quoi sert ?	Créer et déployer les images des machines virtuels	Créer et déployer des conteneurs	créer et déployer des applications (services)
Automatise	-	Ordonnancement	Tout
gère	Un seul atome	Des serveurs	Des applications
flexibilité	++	+	-
Agilité	-	+	++

TABLE 3.1 – Le Cluster dans le Cloud

En général, un cluster comporte plusieurs éléments qu'on définit ci-dessous [7] :

— **Master** : C'est la machine qui est responsable du Cluster. Il veille sur son bon fonctionnement grâce au système de découverte des services et l'ordonnanceur.

— **Ordonnanceur** : Dans un ordinateur ordinaire, pour démarrer un processus, on a besoin d'un gestionnaire des processus pour les faire fonctionner et les garder en cours d'exécution. L'ordonnanceur le fera sur l'ensemble des machines (Cluster).

— **Nœud** : Une machine qui appartient au Cluster, elle peut être réelle ou virtuelle.

— **Gestionnaire des conteneurs** : C'est la partie de logiciel qui est distribuée dans chaque machine du cluster. Il supervise les conteneurs ordonnancés, les redémarre en cas de nécessité.

— **Conteneur ordonnancé** : Un service fonctionne à l'intérieur d'un conteneur ordonnancé. En effet, ce conteneur se trouve dans une telle machine parce que le développeur a déclaré à l'ordonnanceur du Cluster (maître) les besoins du service en termes de ressources (nombre de CPU, RAM, etc...). L'ordonnanceur fournit pour le service un endroit où il peut vivre en paix.

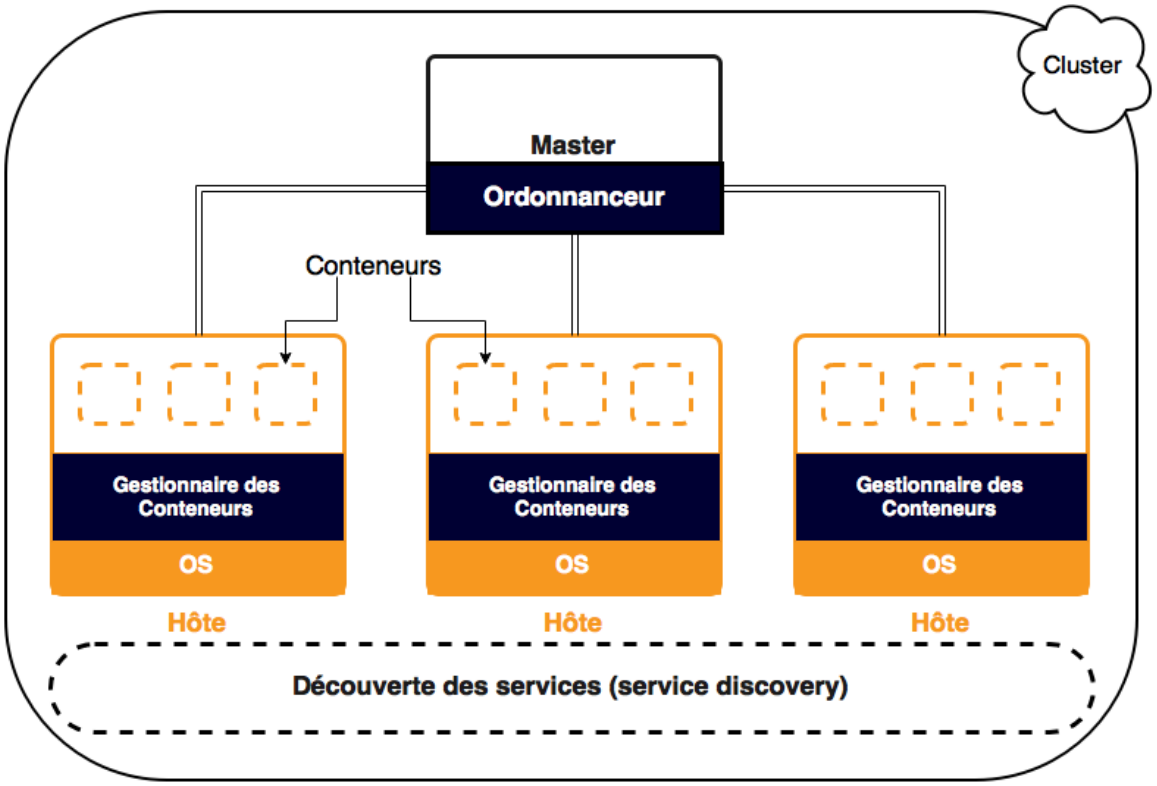

FIGURE 3.2 – Un cluster en général

Nous avons vu la notion de Cluster et les processus d'automatisation et de coordination. De ce fait, la gestion des services informatiques distribués se trouve incontournable au fonctionnement du cluster, elle est appelé **l'orchestration**. Dans la section suivante, nous allons détailler les différents solutions d'orchestration qui existent dans le monde de la containérisation.

3.2 Orchestration Docker

Les capacités d'orchestration de Docker sont construites sur les fondations existantes de Docker. Ces capacités sont assurées par trois nouveaux services de la plate-forme qui sont conçus pour couvrir tous les aspects du cycle de vie dynamique des applications distribuées. Selon l'entreprise Docker, toutes ces caractéristiques sont conçues avec la philosophie de conception "*Batteries Included, but Removable*" qui indique qu'ils peuvent fonctionner avec des services tierces. Ceci offre le choix aux clients de choisir entre les différents outils d'orchestration de Docker et les alternatives communautaires.

Docker Machine

Ce service facilite l'approvisionnement d'un hôte avec Docker installé dans une variété d'environnements. Les développeurs peuvent rapidement lancer les machines hôtes exécutant Docker ; sur un ordinateur portable, un centre de données de VMs, ou une instance Cloud. Cela évite la tâche de se connecter à un hôte pour installer et configurer le démon Docker et le client. Bien que toujours en version alpha, Docker machine prend en charge l'approvisionnement de Docker localement avec VirtualBox et à distance sur les instances Digital Ocean. Le support pour AWS, Azure, VMware, OpenStack et d'autres infrastructures devrait arriver rapidement.

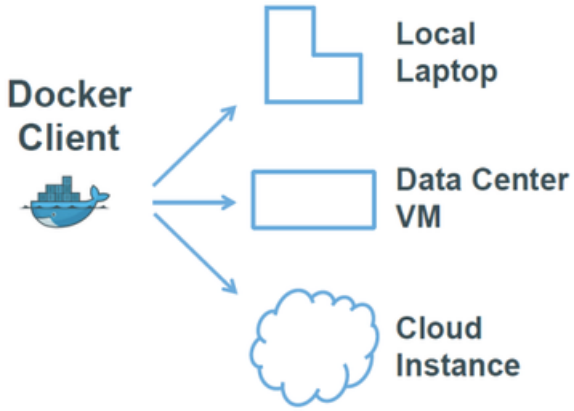

FIGURE 3.3 – Docker Machine [8]

Docker Swarm

Docker Swarm est un service de Clustering natif de Docker qui fonctionne avec le moteur Docker standard, et qui crée un ensemble de ressources sur lesquels les applications distribuées s'exécutent. Cela permet aux développeurs et aux équipes opérationnelles de considérer un cluster de machines Docker comme une collection de ressources unique. Les administrateurs peuvent planifier des conteneurs qui seront lancés dans l'un des hôtes qui répond aux exigences. Docker Swarm fournit des contraintes standard et personnalisées pour répondre aux besoins et à la planification basée sur des règles, cela permet de déclarer des exigences et contraintes spécifiques à chaque conteneur. Docker Swarm est conçu pour évoluer avec le cycle de vie de l'application. Il peut prendre en charge un hôte dans l'environnement de développement et d'autres s'exécutant dans l'environnement de production.

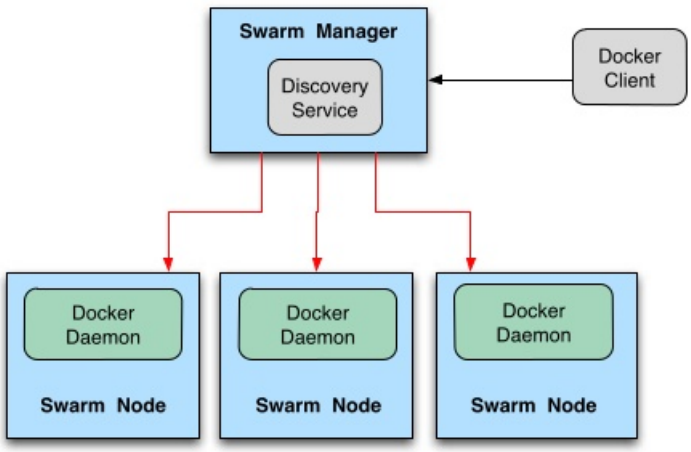

FIGURE 3.4 – Architecture de Docker Swarm [9]

Docker Compose

Docker Composer permet aux développeurs d'assembler un ensemble de conteneurs. Du coup, ils agissent de façon autonomes, interopérables et indépendantes des infrastructures. Avec cette approche déclarative, il est facile de définir des piles de conteneurs qui sont portables. Une pile d'applications distribuées est déclarée dans un simple fichier de configuration YAML qui contient la définition de chaque conteneur.

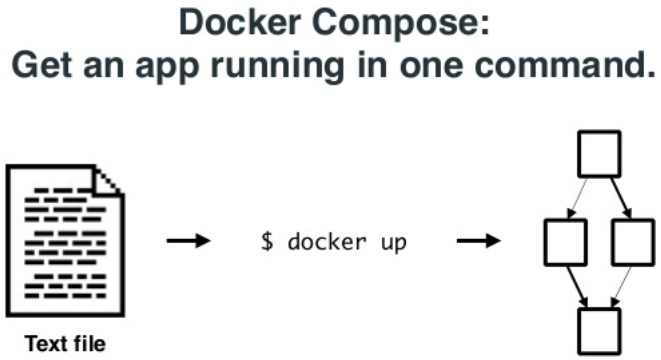

FIGURE 3.5 – Docker Compose [10]

3.3 Orchestration Google

L'entreprise Google est aussi intéressée par l'orchestration des conteneurs Docker, elle a d'ailleurs lancé un projet nommé **Kubernetes**. Ce projet est récent mais prometteur et pourrait révolutionner le domaine du Cloud s'il est suffisamment mature pour la production. Kubernetes a pour objectif de fournir un outil de supervision unique capable de déplacer des conteneurs Docker d'un Cloud à un autre. Autrement dit, de proposer une forme d'interopérabilité dans le nuage, via un framework de gestion de conteneurs solide, ouvert et adapté à toute application sur tous types d'environnements, qu'il s'agisse de Cloud privé, public ou hybride. Ce projet a le soutien et l'attention des géants du Cloud notamment Microsoft, IBM, RedHat qui tiennent à s'assurer que cet outil sera compatible avec leurs Clouds et tentent de l'associer au projet **Openstack** qui est l'orchestrateur du Cloud hybride open source.

3.3.1 Architecture de Kubernetes

Kubernetes est un outil open-source pour l'orchestration des conteneurs Docker. Il gère l'ordonnancement des nœuds dans un Cluster et gère les ressources pour mieux correspondre à l'intention de l'utilisateur. En utilisant les concepts de "*labels*" et "*pods*", il permet un regroupement optimal de conteneurs pour une meilleure gestion et une découverte facile.

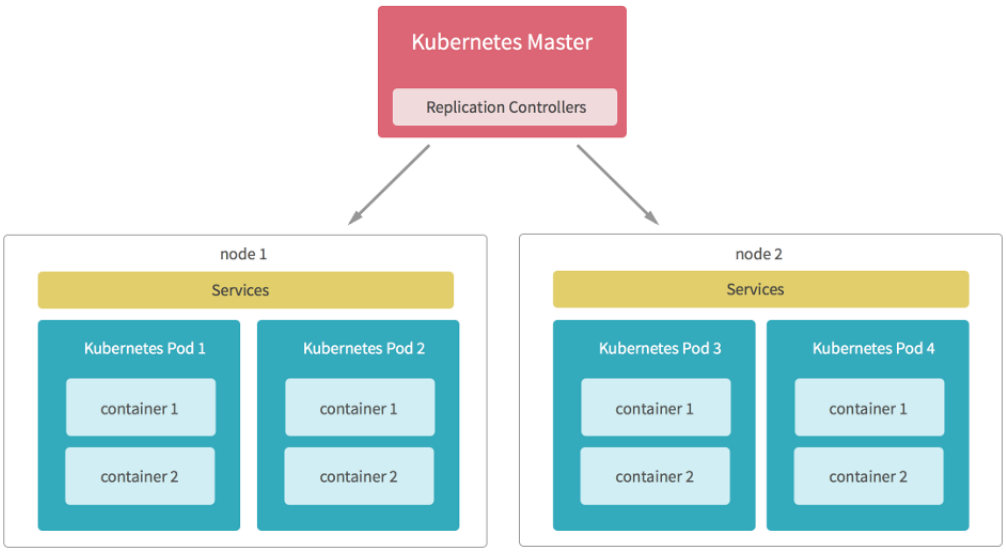

FIGURE 3.6 – Architecture de Kubernetes [11]

— **Kubernetes Master** : il contrôle l'ensemble du Cluster et exécute l'API. Fondamentalement, il est responsable du Cluster.
— **Nodes** : un nœud est un serveur physique (ou une machine virtuelle) à l'intérieur du cluster. Il communique avec les maîtres et contient des conteneurs, on peut ajouter ou supprimer des nœuds à volonté.
— **Pods** : un "*Pod*" est le bloque basique de construction dans *Kubernetes*.Dans un pod, il est possible de lancer plusieurs conteneurs. L'allocation des CPU, mémoire et des autres ressources est gérée dans un pod. Chaque pod possède sa propre adresse IP et son nom d'hôte pour éviter d'éventuels conflits de ports.
— **Replication Controller** : Il est vrai que le Pod est un composant puissant au sein de *Kubernetes* mais il ne permet pas la gestion des échecs. Les échecs sont des événements inévitables bien qu'il faudrait que le service soit toujours disponible. C'est de ce fait qu'intervient *Replication Controller*. Ce dernier s'assure qu'un nombre donné de pods sont exécutés au sein du Cluster, il peut enlever et ajouter des pod du Cluster donc il faudra définir un pod template pour assurer sa fonction.
— **Services** : les Pods sont ajoutés ou supprimés donc il faudrait permettre un **load-balancing** du trafic dans ses pods. "*Service*" agit comme un *load-balancer* dynamique pour un ensemble de pods, il est très efficace et utilise différentes techniques (IP Tables, ...) pour éviter la surcharge.

3.4 Orchestration Apache

L'entreprise Apache a toujours été intéressée par le domaine du Cloud, le lancement de son projet **Mesos** visait principalement à améliorer l'orchestration des *"datacenters"* ceci bien avant l'émergence des conteneurs et Docker. Le projet Mesos est très prometteur pour le futur avec les conteneurs bien qu'il soit déjà mature avec les machines virtuelles. Plusieurs grandes entreprise utilisent Mesos notamment *Twitter*, *Paypal* et *Airbnb*. Mesos agit comme un "Cluster Manager" et offre de nombreuses fonctionnalités telles que :
— Une scalabilité à plus de 10000 nœuds.
— Isolement des ressources pour les tâches via les conteneurs linux (LXC).
— Gestion efficace du CPU et de la mémoire interne.
— Haute disponibilité du master via Apache Zookeeper.
— Une interface Web pour le monitoring des Clusters.

3.4.1 Architecture de Mesos

Mesos possède une architecture composée principalement de maîtres, esclaves et frameworks. Nous définirons dans cette partie les composants pertinents de l'architecture suivante.

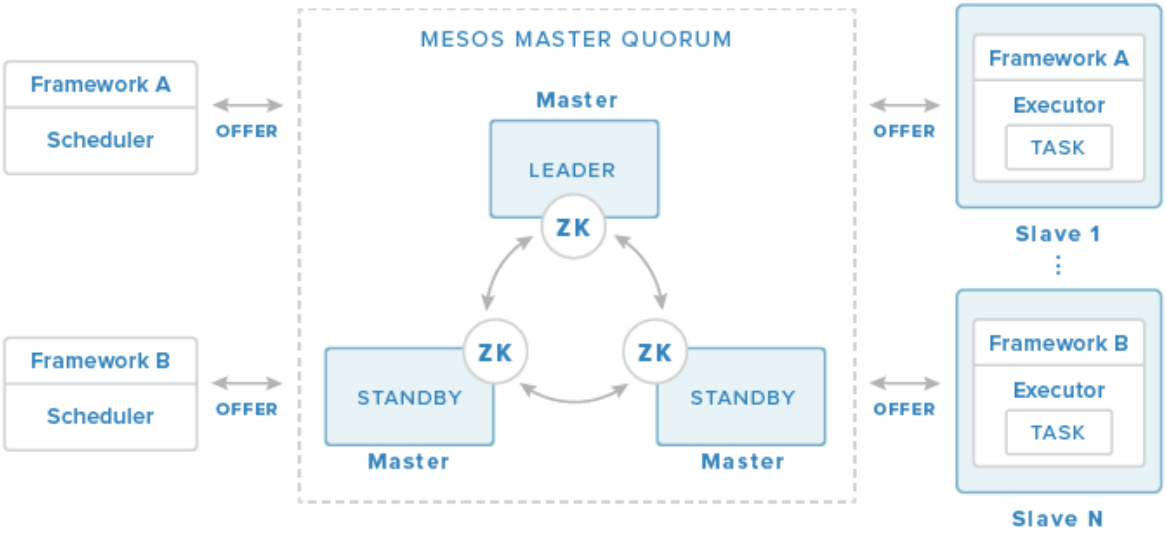

FIGURE 3.7 – Architecture de Mesos [12]

— **Master daemon** : il tourne dans noeud "master" et gère les esclaves.
— **Slave daemon** : il tourne aussi dans un noeud "master" et exécute les tâches des frameworks.
— **Framework** : connu aussi par l'application Mesos, il est composé d'un ordonnaceur qui gère les offres d'allocation de ressources, et d'un ou plusieurs exécuteurs qui lancent les tâches sur les esclaves.
— **Offer** : c'est une liste des ressources disponibles de la mémoire et du CPU propre à un nœud esclave. Tous les nœuds esclaves envoient des offres pour le maître et ce dernier les transmet aux frameworks disponibles.
— **Task** : c'est une tâche qui est ordonnancée par un framework et qui est exécutée dans un nœud esclave. Cette tâche peut être de n'importe quel type (commande, script bash ,requête SQL, Hadoop Job, ...)
— **Apache ZooKeeper** : C'est un logiciel qui coordonne les nœuds maîtres.

3.4.2 Frameworks Mesos

Le framework ou l'application Mesos tient une importante place au sein de l'architecture. Dans cette partie , on présentera deux frameworks importants qui sont **Marathon** et **Chronos**.

FIGURE 3.8 – Composants de l'architecture Mesos [13]

Marathon

Marathon est un framework Mesos développé pour exécuter les applications "*long-running*". Il sert de remplacement pour le système *init*. Il possède de nombreuses fonctionnalités qui simplifient les applications en cours d'exécution dans un environnement en Cluster telles que la haute disponibilité, les contraintes de nœuds, les contrôles de santé de l'application, une API pour scriptabilité et la découverte de service ainsi qu'un outil facile à utiliser l'interface utilisateur Web UI. Il ajoute ses capacités de mise à l'échelle et d'auto-guérison à l'ensemble des fonctionnalités de Mesos.

Chronos

Chronos est une application Mesos qui a été initialement développée par Airbnb comme remplacement pour le système *cron*. Il est un ordonnanceur pleinement fonctionnel, distribué et tolérant aux pannes pour *Mesos*, ce qui facilite l'orchestration des "*jobs*" qui sont des collections de tâches. Il comprend une API qui permet d'exécuter les scripts de la planification de tâches et une interface web pour la facilité d'utilisation. *Chronos* est complémentaire à *Marathon* car il fournit une autre façon d'exécuter des applications selon un calendrier ou d'autres conditions telles que la réalisation d'un autre emploi. Il

est également capable de la planification des tâches sur plusieurs nœuds esclaves *Mesos*, et fournit des statistiques sur les échecs et les réussites d'emploi.

3.5 coreOS

coreOS est un système d'exploitation open-source basé sur le noyau *Linux* et construit pour fournir des infrastructures aux déploiements en Cluster. En effet, il est un système minimaliste, sur lequel est greffé des conteneurs applicatifs indépendants et sécurisés. coreOS est conçu pour être léger et performant afin de gérer les datacenters. coreOS consomme 40% moins de mémoire au démarrage en moyenne par rapport à un serveur *Linux*. coreOS est disponible sur plusieurs plates-formes comme Amazon Compute Cloud, Google Compute Engine et plusieurs autres [14].

3.5.1 Composants de coreOS

Le service Systemd

systemd est un service de gestion de système et ses processus. Il a pour but d'offrir un meilleur cadre pour la gestion des dépendances entre services, de permettre le chargement en parallèle des services au démarrage, et de réduire les appels aux scripts shell. Il est contrôlé dans le système d'exploitation CoreOS par le service *Fleet*.

Le service Etcd

Le service *etcd* est une base de données distribuée de clé-valeur qui fournit une configuration partagée et un système de découverte de services pour les Clusters *CoreOS*. Il fonctionne sur chaque machine dans un Cluster et gère l'élection du maître lors de la production du Cluster ou la perte du maître.

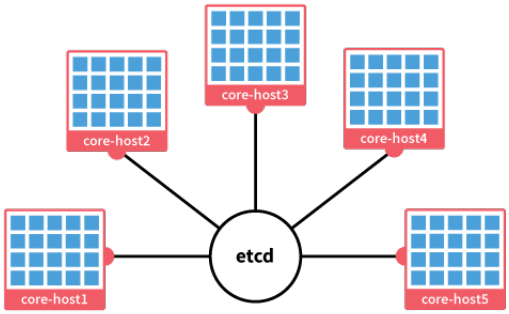

FIGURE 3.9 – Service etcd [15]

Le service Fleet

Fleet permet de traiter le Cluster *Coreos* comme s'il partageait un unique système d'initialisation. Il encourage les utilisateurs à écrire des applications sous forme de petites unités éphémères qui peuvent facilement migrer autour d'un Cluster. En utilisant *Fleet*, les développeurs peuvent se concentrer sur la gestion des conteneurs sans soucier des machines qui tournent les conteneurs. Le service *Fleet* a plusieurs avantages :

— Déployer les conteneurs Docker sur des hôtes arbitraires dans un cluster.
— Distribuer des services vers un Cluster depuis une machine locale.
— Maintenir un nombre fixe d'instances d'un service, ré-ordonnancer lors d'une panne d'une machine.
— Découvrir les machines fonctionnant dans le Cluster.
— Se connecter automatiquement via SSH dans la machine exécutant un service.

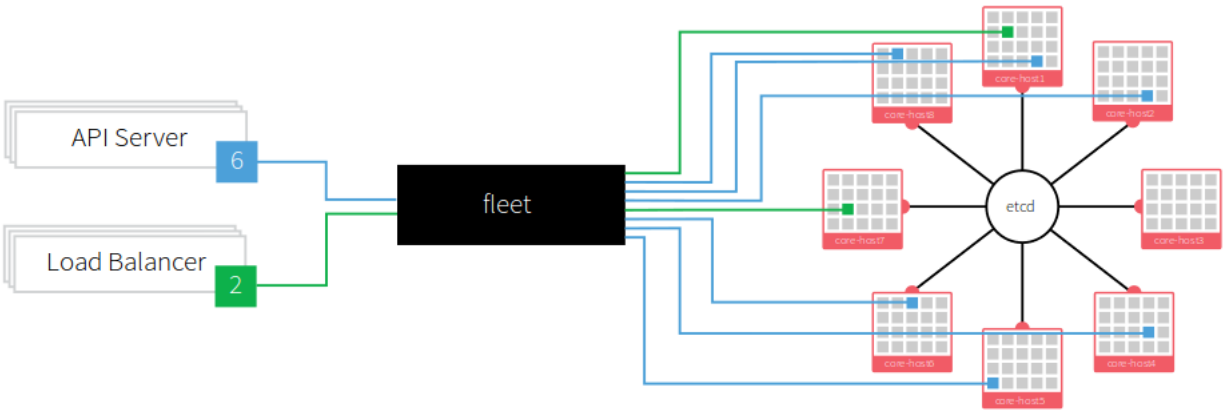

FIGURE 3.10 – Service management Fleet [15].

3.6 Récapitulation

CoreOS	Kubernetes	Mesos
Etcd élit le leader du cluster	Un master par cluster	Haute disponibilité des masters
Orchestration bas niveau	Orchestration haut niveau	Orchestration bas niveau
Stable	Version de pré-production	Stable
Rapidité de production du cluster	Moins rapide	Moins rapide
Bien documenté	Documentation pauvre	Bien documenté
Inconscience vis-à-vis les ressources du cluster	Inconscience vis-à-vis les ressources du cluster	Ordonnancement conscient des ressources du cluster
Axé sur les conteneurs	Il rassemble des années d'expériences de Google avec l'orchestration des conteneurs	moins axé sur les conteneurs
Destiné pour Docker	Destiné pour Docker	Utilisé depuis des années pour l'orchestration de Hadoop, il vient de subir un ré-usinage de code pour supporter Docker
Limité	Complet (encore plus dans le futur)	Management générique du cluster, extensibilité par les différents ordonnanceurs
Pas de proxy et d'équilibreur de charge	Dispose de son propre proxy	Des proxy matures enfichables

TABLE 3.2 – Matrice de décision pour le choix de la solution d'orchestration

Mesos, Kubernetes et CoreOS essaient tous de résoudre des problèmes très similaires, chacun agit dans un niveau d'abstraction. L'on peut imaginer faire des combinaisons des trois solutions d'orchestration en cas de nécessité :

— **CoreOS + Kubernetes** : CoreOS est parfait pour produire rapidement un Cluster. Il va ordonnancer les composants de base de Kubernetes dans le Cluster. L'ordonnancement des conteneurs va être délégué à Kubernetes.

— **Mesos + Kubernetes** : Mesos n'inclut pas nativement un ordonnanceur. L'extensibilité de Mesos fait qu'il est possible de laisser Kubernetes agir comme l'ordonnanceur de Mesos.
— **CoreOS + Mesos + Kubernetes** : L'idée est de jouir de tous les points forts des solutions. En contrepartie, on perd en simplicité, il est préférable de ne l'adopter qu'en cas de nécessité.

Faute de maturité, nous allons adopter CoreOS pour sa stabilité, sa simplicité et sa documentation disponible en dépit qu'il est bas niveau.

CHAPITRE 4

Conception

Ce chapitre a pour objectif de présenter la conception du projet. Cette phase est un point de fusion entre l'analyse des besoins et l'étude approfondie de la documentation. Dans un premier temps, une problématique des bases de données sera soulevée. Ensuite, nous allons expliquer comment un SaaS doit être conçu. Enfin, une architecture Cloud pour porter les conteneurs sera présentée.

4.1 Base de données

Odoo, comme toute autre application, a besoin d'une base de données *PostgreSQL* pour persister des informations. Malheureusement, la communauté de Docker n'est pas si convaincue vis-à-vis l'idée de containériser les bases de données. Nous allons prouver à travers deux scénarios qu'il serait une imprudence de tourner les bases de données dans des conteneurs.

Le fait de la rapidité et la légèreté de Docker et des conteneurs en général a poussé CoreOS d'adopter une certaine philosophie à l'égard de la gestion et l'ordonnancement des conteneurs. En effet, lorsqu'un conteneur s'est arrêté pour une raison quelconque, il va être automatiquement écrasé et remplacé par un nouveau qui va tourner le même service. Une philosophie, certes, moins prudente et naïve, mais sa simplicité épargne beaucoup de problèmes à tout le monde.

Par conséquent, on ne peut en aucun cas mettre les données dans le système de fichiers d'un conteneur sous peine de les perdre. La figure 4.1 montre qu'après avoir tuer un conteneur, ses données sont perdues.

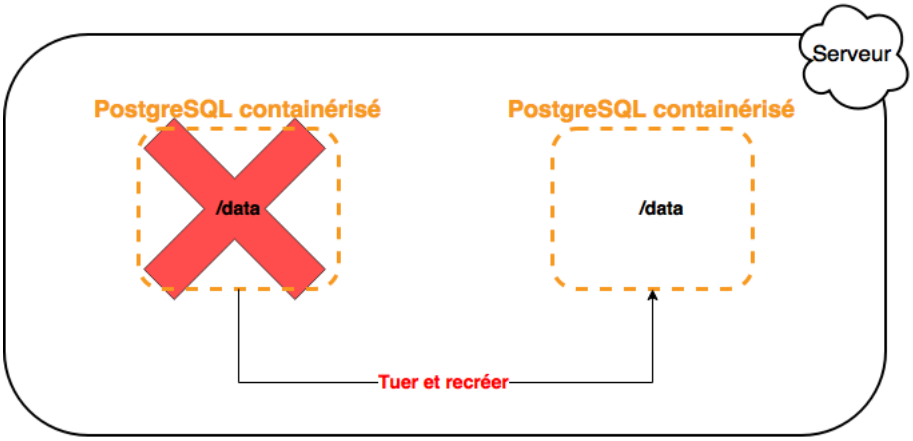

FIGURE 4.1 – Données perdues - scénario 1

Pour y remédier, l'on peut penser à monter le volume des données dans le serveur hôte, ainsi les données sont plus ou moins persistantes. Outre le fait que les données seront certainement perdues quand le serveur tombe en panne, le conteneur n'aura plus accès aux données quand il sera réordonnancé. En effet, le caractère imprévisible de l'ordonnanceur *Fleet* fait qu'il n'y a pas de garantie qu'un conteneur réside dans le même serveur. La figure 4.2 montre ce scénario.

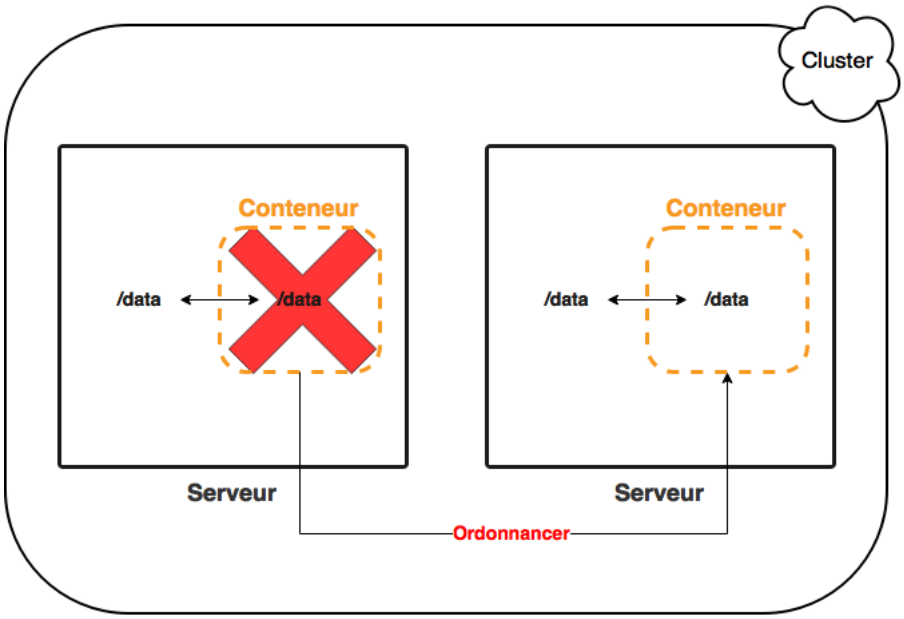

FIGURE 4.2 – Données perdues - scénario 2

Dans un environnement de production, il est encore tôt pour mettre les bases de données et les applications à états (stateful) en général dans des conteneurs. De ce fait, il serait judicieux de les faire sortir de l'architecture à base de conteneurs. Par conséquent, deux solutions se portent à nous, les bases de données vont être mis en place suivant l'informatique classique ou elles vont être consommées comme un service à partir d'une partie tierce Cloud (Délégation de l'informatique). La dernière solution est coûteuse mais il est plus fiable.

4.2 Scalabilité et disponibilité

Parmi les contraintes qui ont poussé Sayoo à migrer vers le Cloud, c'est de garantir une haute disponibilité et une scalabilité infinie.

La haute disponibilité fait référence au fait qu'un service est en quelque sorte tolérant à un échec ou à une panne. La disponibilité va être assurée en faisant la redondance des services dans différents serveurs du Cluser.

La scalabilité ou l'évolutivité, c'est la capacité d'une application à pouvoir exploiter de nouvelles ressources récemment déployées pour répondre à un pic de charge. En fait, la scalabilité n'est pas apporté par le Cloud, qui apporte l'élasticité, mais c'est l'application elle-même qui l'apporte. Par conséquent, l'idée que toute application déployée dans un milieu élastique Cloud peut être évolutive est fausse.

Pour faire une bonne conception d'un SaaS évolutif, il faut distinguer d'abord deux types d'applications :

— Application **stateful** ou à états : C'est une application dont les processus ne stockent aucune donnée qui doit persister, même dans un temps aussi petit qu'il soit, dans l'espace mémoire ou le système de fichiers.

— Application **stateless** ou sans états : C'est une application qui stocke tous les données persistantes dans un service de support externe (Base de données, queue, mémoire cache, etc.).

Une application évolutive doit être **stateless**, plus que cela, il doit respecter les douze-facteurs définies dans l'annexe C. Une application évolutive suppose qu'aucune donnée mise en cache (mémoire, disque) ne sera disponible dans une requête ultérieure. En effet, les chances sont élevées qu'une future requête sera servie par un processus différent du même service. L'idée c'est qu'une application doit être capable de renvoyer les réponses de façon consistante à partir de n'importe quelle instance exécutant l'application.

La figure 4.3 montre un service qui n'est pas évolutif contrairement au service présenté dans la figure 4.4.

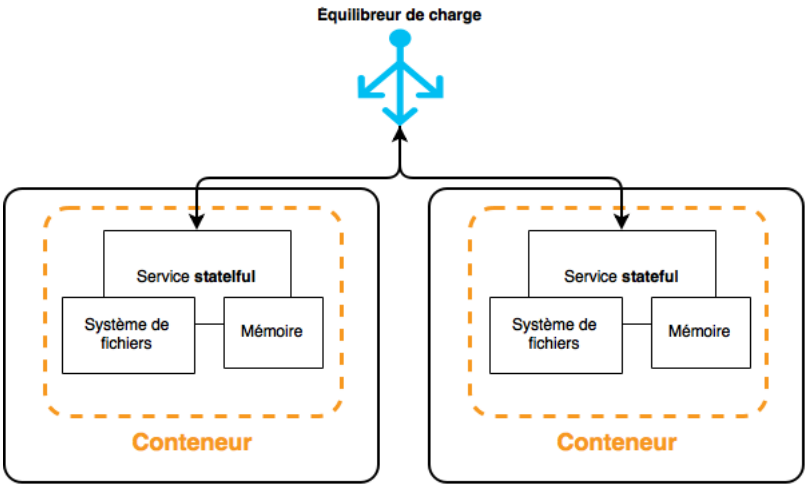

FIGURE 4.3 – Service non évolutif

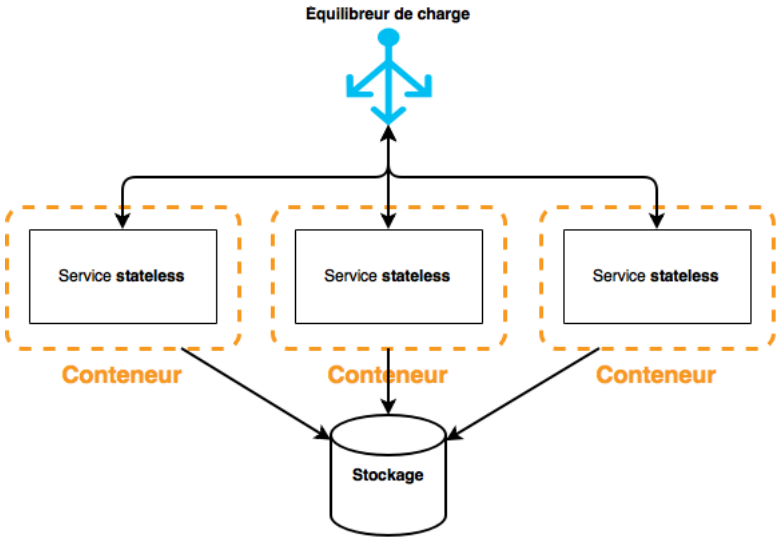

FIGURE 4.4 – Service scalable

4.3 Architecture

L'architecture est très simplifiée car nous avons fait une abstraction du comportement interne des composants de CoreOS. Cette architecture est inspirée de la plate-forme de Nuxeo [16] basée sur Docker et CoreOS et conçue pour la résilience des pannes. Elle décrit trois processus et implique trois acteurs :

— **Développeur** : Il développe, personnalise et adapte l'image Odoo de base aux besoins du client. Une fois le développement est fait, il pousse l'image Odoo vers le registre des images de Docker. Cette image est étiquetée par un nom, une version et un environnement (développement, test, production).

— **Exploitant** : Il exploite l'ordonnanceur Fleet pour déployer l'application que le développeur vient de pousser dans le registre. L'exploitant va demander à l'ordonnanceur, par exemple, de lancer le service en mode haute disponibilité. L'ordonnanceur lance plusieurs instances du service dans des serveurs différents. L'exploitant, remarquant un pic de charge sur ce service, va demander à l'ordonnanceur de lancer une autre instance qui compense la surcharge.

— **Client** : C'est l'utilisateur final de l'application. Il va saisir l'URL de son application. Son requête va être interceptée par l'équilibreur de charge (Load-Balancer) qui va faire suivre la demande à un des serveurs du Cluster. Quand la requête est arrivée, elle sera interceptée par le proxy de ce serveur qui va la faire suivre à l'instance du service la moins chargée, quitte même à la faire suivre vers une instance

d'un autre serveur.

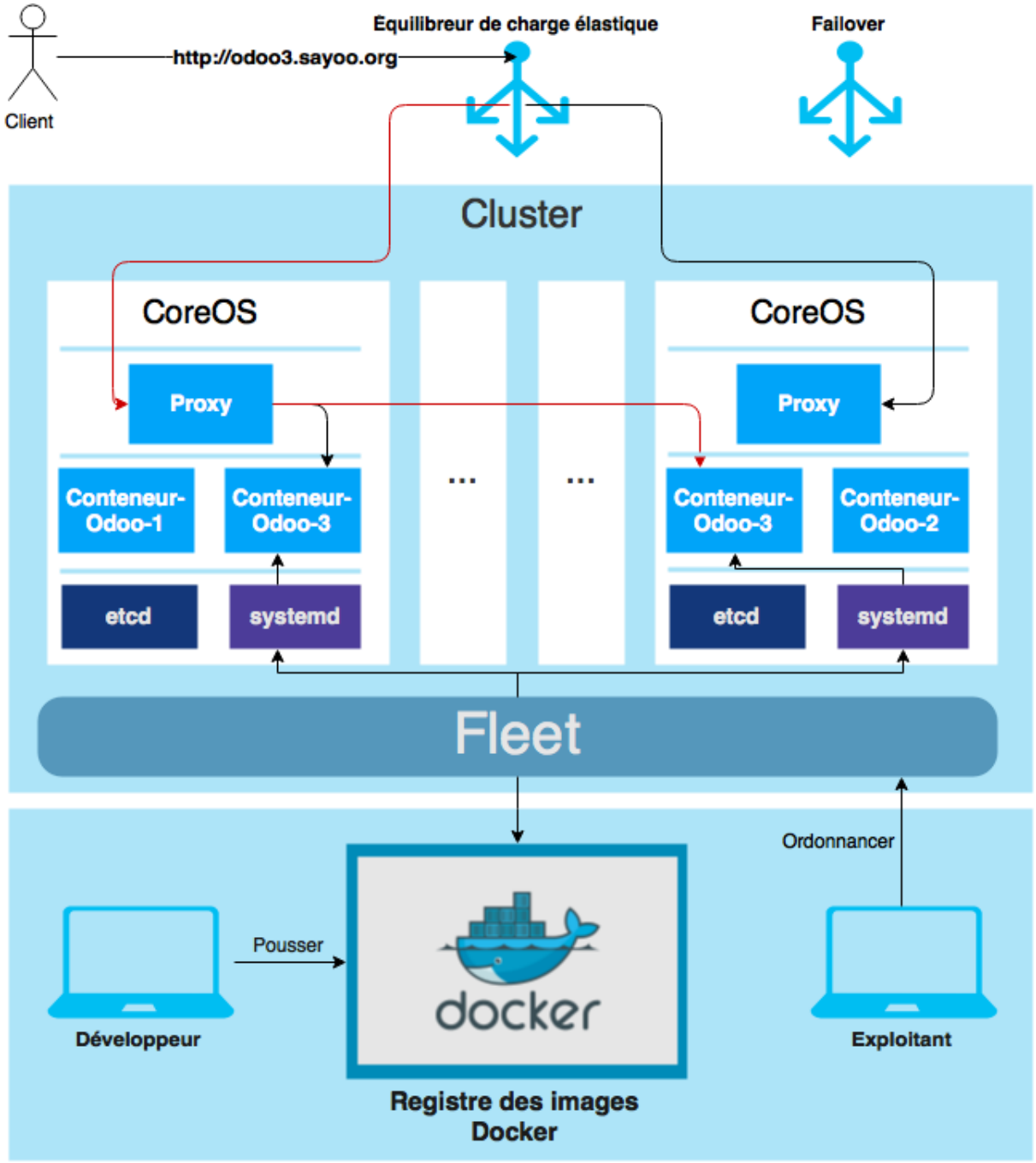

FIGURE 4.5 – Architecture de la solution

Conclusion

En somme, nous avons prouvé l'incompatibilité des bases de données avec les conteneurs dans une solution fiable. Une architecture qui répond aux exigences du Cloud a été présentée, mais il ne faut pas la gâcher par une mauvaise conception des applications.

CHAPITRE 5

Réalisation

Ce chapitre est consacré à la description de la phase de la réalisation. La démarche globale de deux solutions sera décrite, puis les avantages de l'une par rapport à l'autre seront illustrés. Enfin, La sécurisation et l'installation d'un système de supervision évolutif clôturera le chapitre.

5.1 A base d'un Cluster

5.1.1 Installation du Cluster

Nous allons mettre en place l'architecture décrite dans la figure 4.5. Le Cloud public AWS sera utilisé pour la production d'un Cluster de trois serveurs [17]. Il est important de noter que le Cluster peut-être installé en tant qu'une entité privée ou publique :

— **Réseau privé** : Les serveurs du Cluster communiquent entre eux avec des adresses IP privées. Cette solution est contraignante. En effet, les serveurs doivent être installés dans la même zone. De plus les fournisseurs chargent des coûts supplémentaires pour l'installation d'une interface réseau privée. En contrepartie, on gagne en termes de sécurité.

— **Réseau public** : Les serveurs du Cluster peuvent se communiquer mutuellement via une interface réseau publique et donc une adresse IP publique. De ce fait, beaucoup d'avantages se dégagent. D'une part, les serveurs peuvent être distribués dans le monde entier et ainsi garantir une disponibilité maximale. D'autre part, une migration à chaud de toute la plate-forme peut être faite. Cette migration est silencieuse et transparente pour tous les clients.

5.1.2 Installation des proxy

Gogeta est un proxy dynamique dont la configuration est basée sur la base de donnée Etcd. En effet, Gogeta écoute en permanence Etcd pour assurer une reconfiguration dynamique en temps réel des routes sans redémarrage. L'installation de Gogeta sur tous les serveurs est très simple. En effet, il est installé dans un conteneur et ordonnancé par Fleet [18].

Quand l'ordonnanceur Fleet crée un service en mode haute disponibilité, Il stocke les données (IP et port) dans Etcd sous forme de clé-valeur. Les proxy Gogeta, étant en écoute permanente d'Etcd , se reconfigure pour prendre en considération le nouveau service. Ainsi, si une requête arrive sur l'un des proxy Gogeta, il la transmet vers une des instances avec la stratégie *Round-robin* après avoir fait des bilans de santé.

```
1 /domains/odoo.sayoo.org/type  = service
2 /domains/odoo.sayoo.org/value   = odoo
3 /services/odoo/1/location   = {"host":"10.240.210.86", "port": 42654}
4 /services/odoo/2/location   = {"host":"10.240.228.23", "port": 42669}
```

Script 5.1 – Contenu de la base de donnée Etcd

5.1.3 Utilisation du Cluster

Autant la mise en place du Cluster et ses composants est simple, autant son utilisation est compliqué. Un exemple sera présenté pour illustrer la procédure d'ordonnancement d'un service Odoo dont l'image est stockée dans le registre public de Docker.

```
[Unit]
Description=Odoo
After=docker.service
Requires=docker.service
[Service]
EnvironmentFile=/etc/environment
ExecStartPre=-/usr/bin/docker kill odoo
ExecStartPre=-/usr/bin/docker rm odoo
ExecStartPre=/usr/bin/docker pull sayoo/odoo:latest
ExecStart=/usr/bin/docker run -p="49069:8069" --name odoo sayoo/odoo:latest
ExecStartPost=/usr/bin/etcdctl set /domains/odoo.sayoo.org/type service
ExecStartPost=/usr/bin/etcdctl set /domains/odoo.sayoo.org/value odoo
ExecStartPost=/usr/bin/etcdctl set /services/odoo/%i/location "{\"host\":\"
    ${COREOS_PRIVATE_IPV4}\", \"port\": 49069}"
ExecStop=/usr/bin/docker stop odoo
ExecStopPost=/usr/bin/etcdctl rm --recursive /domains/odoo.sayoo.org
ExecStopPost=/usr/bin/etcdctl rm --recursive /services/odoo/%i/
Restart=always
RestartSec=5s
[X-Fleet]
X-Conflicts=odoo@*.service

```

Script 5.2 – Fichier d'ordonnancement d'un service Odoo

Le fichier décrit le service Odoo. Premièrement, il requiert le service Docker. Ensuite, avant de lancer Odoo, il faut télécharger la dernière image d'Odoo. Puis, l'exécuter et stocker des couples clés-valeurs dans la base de données qui sont utiles pour l'équilibreur de charge Gogeta. Le service est ordonnancé sous la contrainte de lancer une seule instance par serveur (*X-Conflicts=odoo@*.service*).

Le service est ordonnancé en mode haute disponibilité en lançant deux instances du service. L'application sera enfin accessible via le lien *http ://www.odoo.sayoo.org*.

```
    $ fleetctl submit odoo@.service
    $ fleetctl start odoo@{1,2}.service

```

Script 5.3 – Ordonnancement de deux instances

La solution à base d'un Cluster CoreOS motorisé par des équilibreurs de charges élastiques est très puissante, mais son utilisation demeure très difficile. En effet, interagir directement avec le Cluster nécessite des administrateurs systèmes très compétents. C'est la raison pour laquelle une solution à base d'une PaaS sera adoptée.

5.2 A base d'une PaaS

5.2.1 Plate-forme Deis

Deis

Deis est une Plate-forme en tant que service, PaaS, open-source qui facilite la gestion et le déploiement des applications sur des serveurs. Deis se base sur Docker et CoreOS pour fournir une PaaS très légère inspirée par la fameuse plate-forme Heroku [19].

Applications supportées

Deis peut déployer n'importe quelle application ou service qui peut fonctionner à l'intérieur d'un conteneur Docker. Pour que l'application soit évolutive horizontalement, elles doivent suivre la méthodologie de douze-facteurs de Heroku (Voir Annexe C, page 76).

Plates-formes supportées

Deis peut être déployée sur n'importe quelle système qui supporte CoreOS : poste de travail, ainsi que la plupart des Clouds publics, privés et les centres de données.

Pourquoi Deis ?

— **Langage-agnostique** : Deis déploie des services écrites dans n'importe quels langages de programmation ;
— **Extensibilité** : Deis est extensible, les outils de supervisions et de la gestions des message logs peuvent être intégrés facilement ;
— **Scalabilité** : Une application est montée en charge avec une seule commande. Par ailleurs, La capacité de la plate-forme Deis peut être montée en charge en ajoutant simplement des hôtes au Cluster ;

— **100% Open Source** : Deis est un logiciel libre, open-source publié sous la licence Apache 2.0.

5.2.2 Architecture

l'architecture de Deis présentée dans la figure 5.1 a tellement de points de commun avec celle que nous avons proposé. Néanmoins, elle cache beaucoup de détails, notamment à l'intérieur du contrôleur. Par ailleurs, Deis intègre ses propres routeurs (*Load Balancer*) et un registre privé des images Docker. Les services de support, la supervision, la gestion des messages logs et l'équilibreur de charge principal ne sont pas intégrés par la plate-forme Deis.

Un développeur, voulant déployer une application, va pousser son code vers le contrôleur. Ce dernier construit l'image et la stocke dans le registre des images. Ensuite, le contrôleur passe le relais à l'ordonnanceur qui se débrouille pour lancer le service à l'intérieur d'un conteneur. Le conteneur est crée à base de l'image Docker.

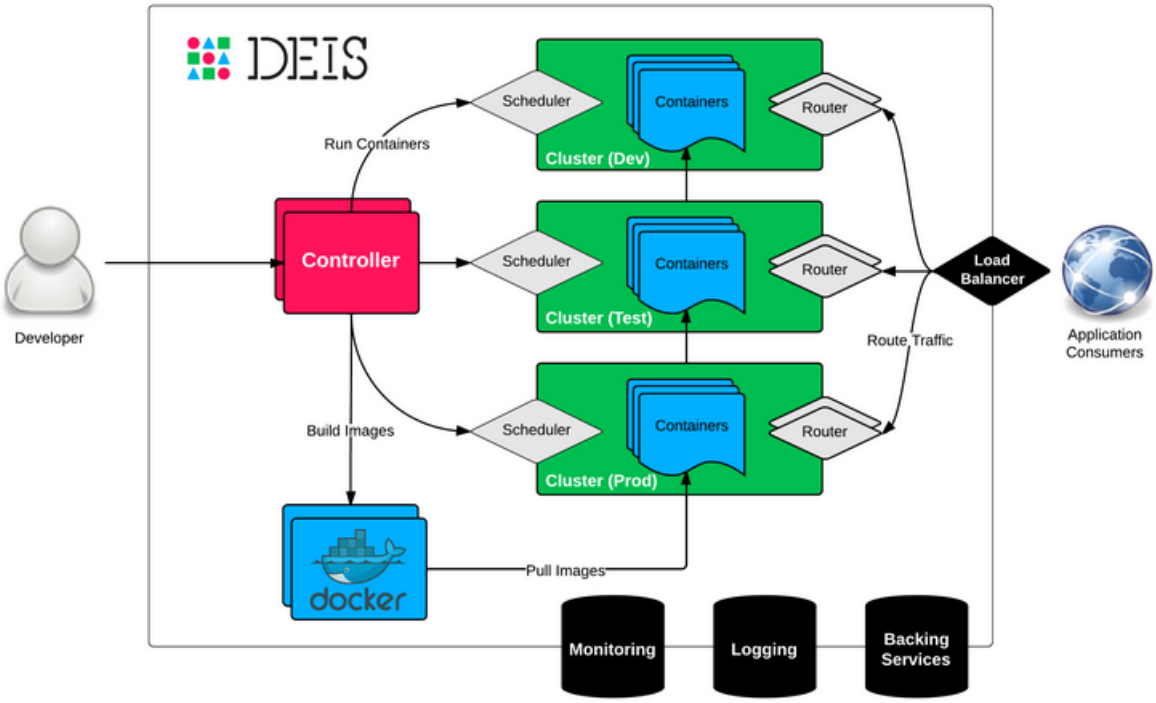

FIGURE 5.1 – Architecture de la PaaS Deis [20]

5.2.3 Installation

Deis nécessite un Cluster de CoreOS pour s'installer dessus. Nous aurions pu installer Deis au-dessus du Cluster que nous avons déjà réalisé, mais Deis nécessite des ressources (au moins 4GB de RAM) que la version d'essai d'AWS n'offre pas. Nous avons opté pour GCE.

L'installation de Deis sur GCE est détaillée dans la documentation [21]. Les grandes étapes d'installation :

— Lancer des serveurs CoreOS avec le fichier d'initialisation de Deis fourni par la documentation.
— Faire pointer l'équilibreur de charge de Google vers nos serveurs. Cette équilibreur de charge est élastique et donc reconnaît dynamiquement des nouveaux serveurs.
— Configurer le DNS pour faire pointer le nom de domaine et tous les sous-domaines vers l'équilibreur de charge de Google (Figure 5.2).

Type	Name	Value	TTL
A	*.cloud	points to 104.197.95.7	Automatic
A	cloud	points to 104.197.95.7	Automatic

FIGURE 5.2 – Configuration du DNS

5.2.4 Avantages du PaaS par rapport au Cluster

Déploiement avec une seule commande

Quand le développeur a terminé le développement ou la personnalisation d'une application, il peut la déployer rapidement.

```
1  $ git push deis master
2
```

Script 5.4 – Déploiement avec Deis

Montée en chargé avec une seule commande

Après avoir lancé un service, l'exploitant, remarquant un pic de charge, va pouvoir ajouter d'autres instances du service. Les routeurs sont dynamiquement reconfigurés pour diffuser la charge à travers toutes les instances.

```
$ deis scale cmd=3

```

Script 5.5 – Montée en charge

Déploiement avec zéro temps d'arrêt

Durant le déploiement d'une nouvelle version d'un service, Deis garantit un temps quasi nul d'indisponibilité. En effet, les routeurs ne sont reconfigurés que lorsque le déploiement est terminé. Enfin, l'ancienne version est supprimée.

Des API HTTP

Deis offre des APIs REST pour un interfaçage facile et standard. Du coup, il est possible de développer dans le futur des interfaces web ou mobile qui cache les difficultés de la ligne de commande. D'ailleurs, une interface web est déja crée par la communauté pour la gestion du Cluster [22].

5.2.5 Sécurité avec le protocole SSL/TLS

5.2.5.1 SSL/TLS

SSL est un protocole qui permet d'échanger des informations entre deux ordinateurs de façon sûre. SSL assure trois choses :

— **Confidentialité** : Il est impossible d'espionner les informations échangées ;
— **Intégrité** : Il est impossible de truquer les informations échangées ;
— **Authentification** : Il permet de s'assurer de l'identité du programme, de la personne ou de l'entreprise avec laquelle on communique.

Nous allons installer SSL dans la plate-forme pour établir un lien crypté entre le navigateur et le serveur (HTTPS). Ce qui est intéressant, c'est que lorsque SSL est activé, il le sera pour tous les services qui s'exécutent sur le Cluster.

5.2.5.2 Installation

L'installation de SSL dans la plate-forme se fait en deux étapes :

— Générer une clé privée et un certificat signé. Normalement, le certificat doit être signé par les PKI, des parties tierces aux-quelles l'on fait confiance, qui chargent une somme d'argent annuelle. Dans un environnement de test, on peut se contenter d'un certificat signé par nous-même ;

— Attacher la clé privée et le certificat avec la plate-forme. En fait, ils doivent être installés dans l'équilibreur de charge. Il y a deux choix qui se portent à nous, soit les installer dans l'équilibreur de charge principal de GCE ou dans les routeurs de Deis. Nous avons décidé de l'installer dans les équilibreurs de charge propres à Deis vu que c'est une solution portable et ne dépend pas de l'environnement où Deis est installé.

```
$ openssl genrsa -des3 -passout pass:x -out server.pass.key 2048
$ openssl rsa -passin pass:x -in server.pass.key -out server.key
$ openssl req -new -key server.key -out server.csr
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out
  server.crt
```

Script 5.6 – Génération de la clé privée et le certificat

```
$ deisctl config router set sslKey=server.key sslCert=server.crt
$ deisctl config router set enforceHTTPS=true
```

Script 5.7 – Activation du protocole SSL

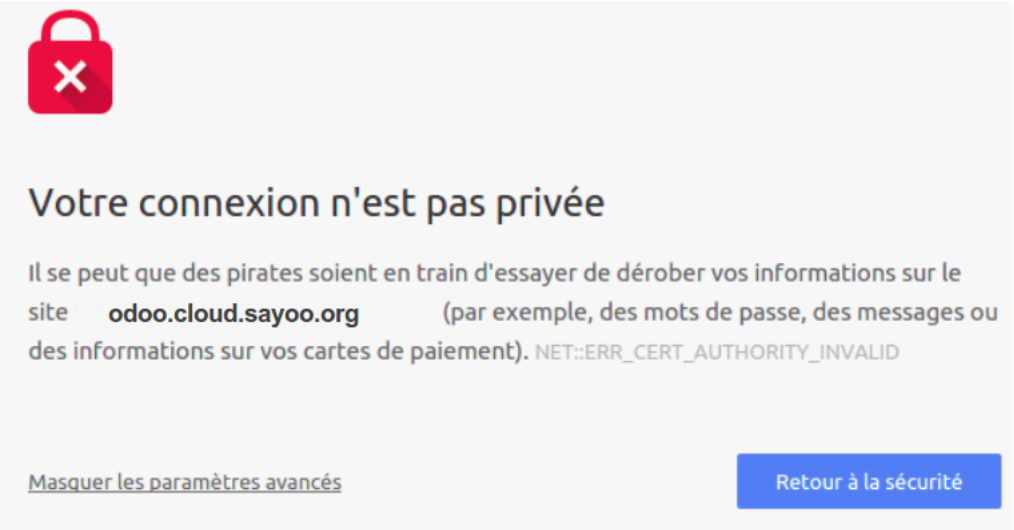

FIGURE 5.3 – Certificat non fiable

Après avoir activé le protocole HTTPS, le navigateur envoie un avertissement (Figure 5.3, page 63). En effet, nous avons signé le certificat nous-même au lieu de la déléguer à une partie tierce de confiance (PKI), ce qui devrait être fait en production. En dépit de cela, nous avons bel et bien sécurisé la communication (Figure 5.4, page 64).

FIGURE 5.4 – Communication sécurisée avec le protocole HTTPS

5.3 Supervision

La PaaS *Deis* ne dispose pas d'un système de supervision intégré. Puisque les services déployés dans Deis fonctionnent entièrement à l'intérieur des conteneurs Docker, cela signifie que les outils de supervision des conteneurs Docker devraient fonctionner avec Deis. Les plus matures d'entre eux seront utilisés.

5.3.1 Outils de supervision utilisés

cAdvisor

cAdvisor (Container Advisor) fournit aux utilisateurs de conteneurs une vue de l'utilisation des ressources et des caractéristiques de performance de leurs conteneurs en cours d'exécution. C'est un démon, toujours en exécution, qui recueille, agrège, analyse, et exporte les informations sur l'exécution des conteneurs. Plus précisément, pour chaque conteneur, il conserve l'historique de l'utilisation des ressources, les histogrammes des historiques complets d'utilisation des ressources et des statistiques du réseau. Ces données peuvent être exportées pour chaque conteneur ou pour toute la machine.

cAdvisor a un support natif pour les conteneurs Docker et devrait supporter à peu près tout autre solution de containérisation. Par ailleurs, cAdvisor fournit une interface utilisateur ainsi qu'une REST API.

Heapster

Heapster est un projet open source de Google qui comble la limitation de cAdvisor qui supervise un seul serveur. Heapster recueille les données à travers les HTTP API fourni par les cAdvisor installés sur les serveurs, puis les stocke dans la base de données InfluxDB. Ainsi, Heapster est considéré comme un superviseur au niveau du Cluster.

InfluxDB

Influxdb est une base de données distribuée destinée pour le stockage des métriques, des événements et des analyses de performances. Influxdb est simple à installer et à gérer, et rapide car elle vise à répondre aux requête en temps-réel. En effet, chaque point de données est indexé et disponible pour des requêtes qui retournent des réponses dans < 100 ms. Enfin, elle dispose d'une HTTP API intégrée qui va être sollicitée par l'interface utilisateur Grafana.

Grafana

Grafana est l'un des meilleurs projets open source pour la visualisation des données de mesure. Il fournit un moyen puissant et élégant pour créer, partager et explorer les données et les tableaux de bord à partir des bases de données de métriques. Grafana assure un support riche pour les bases de données Graphite, InfluxDB et OpenTSDB. Grafana sera utilisé pour visualiser les mesures stockées dans InfluxDB.

5.3.2 Démarche d'installation

Cette solution de supervision s'installe au-dessus du Cluster de CoreOS et cohabite avec la plate-forme Deis. Comme c'est déjà mentionné, tous les services sous CoreOS sont installés en tant que conteneurs. De même, cAdvisor, InfluxDB, Heapster et Grafana vont être installés dans des conteneurs. Les fichiers d'installation se trouvent dans le guide officiel de Heapster [23]

— cAdvisor doit être installé sur tous les serveurs du Cluster. Heureusement, l'ordonnanceur Fleet effectue cette opération de façon automatique et évolutif. En effet, il suffit de mentionner *Global=true* dans le fichier d'installation ;

— InfluxDB, Heapster et Grafana doivent être installés dans le même serveur. Cette contrainte est exprimée par *X-ConditionMachineOf=influxdb.service*. Ainsi, Fleet

va s'en sortir pour faire coexister les trois services.

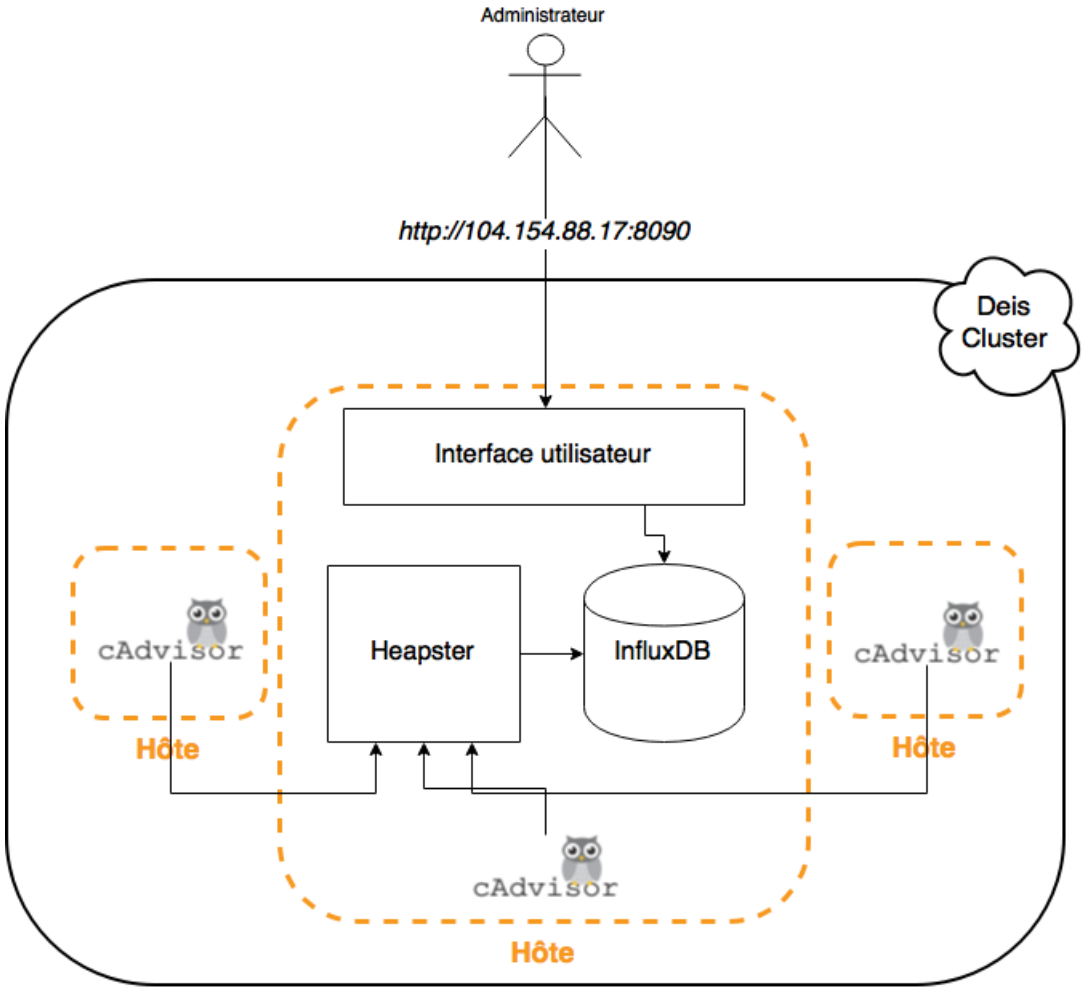

FIGURE 5.5 – Architecture de la supervision

Le code suivant montre comment charger et installer cAdvisor dans le cluster. C'est la même procédure pour les autres services.

```
$ ssh core@104.154.75.6
$ fleetctl load cadvisor.service
$ fleetctl start cadvisor.service
```

Script 5.8 – Lancement du service cAdvisor

5.3.3 Interfaces graphique

Notre Cluster se compose de trois serveurs. Les figures suivantes présentent la quantité totale de CPU et de la mémoire consommée par le cluster ainsi que les quantités

consommées par chaque serveur.

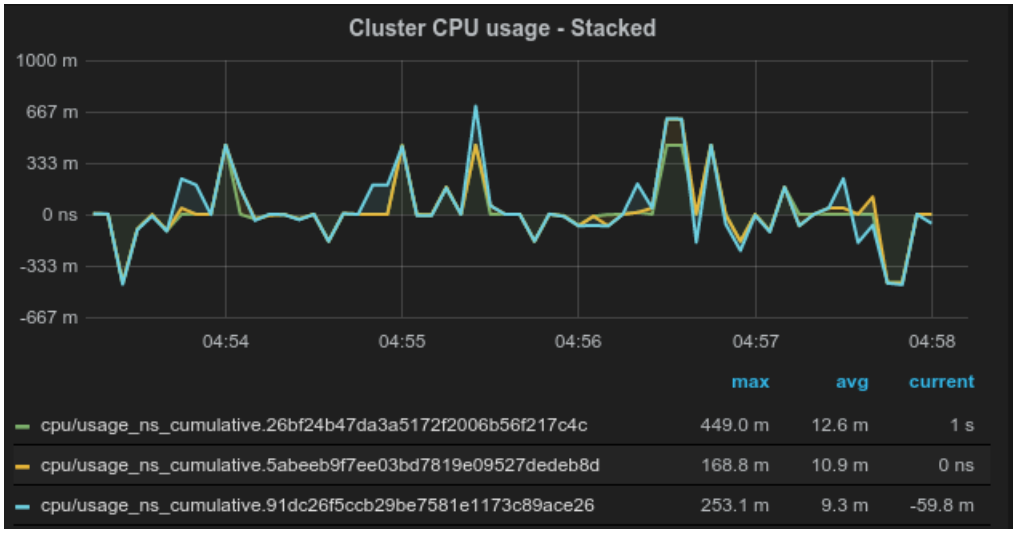

FIGURE 5.6 – Quantité de CPU consommée par le cluster

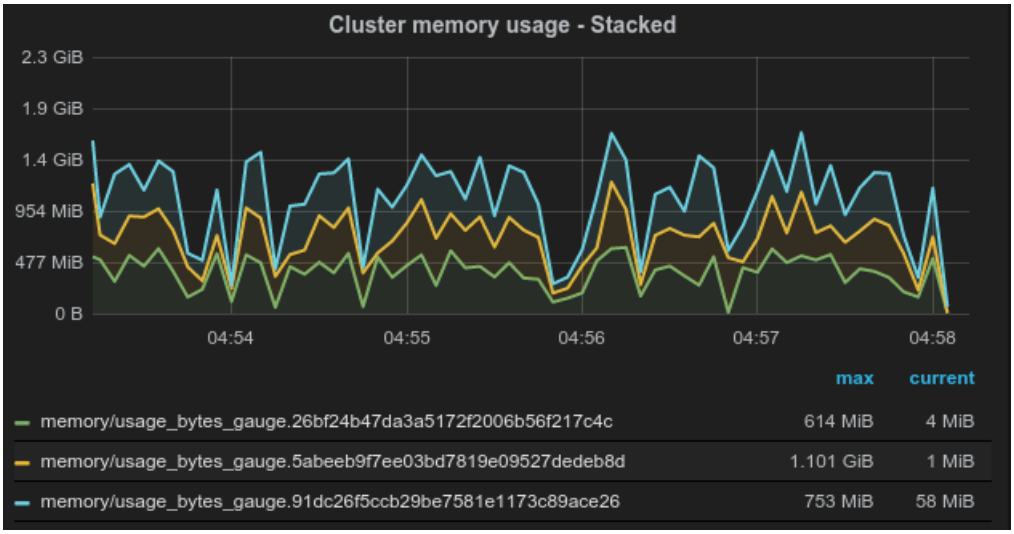

FIGURE 5.7 – Quantité de mémoire consommée par le cluster

5.3.4 Supervision et scalabilité

La scalabilité était toujours parmi nos aspirations tout au long de ce projet. Le système de supervision réalisé est conçu pour fonctionner de façon indépendante, complètement automatisée et hautement évolutive. Cela va être prouvé à travers ces deux scénarios :

— Quand un nouveau service est lancé sur un des serveurs du Cluster, cAdvisor se rendra compte de ce nouveau conteneur et mettra à la disposition de Heapster

les données qui va les stocker dans la base de données pour des consultations via l'interface graphique. Ainsi, l'auto-découverte des services est assurée.

— Lors d'un pic de charge, on a intérêt à lancer les nouveaux conteneurs dans une nouvelle machine. On va tout d'abord lancer un nouveau serveur avec le fichier d'initialisation du Cluster. Désormais, le serveur appartient au Cluster et, de façon automatique, le superviseur le reconnaît parfaitement. Ainsi, les conteneurs lancés sur ce nouveau serveur sont supervisés automatiquement comme le décrit le scénario 1.

D'autre part, Grafana, l'interface utilisateur, sera disponible dans la machine dans laquelle *Fleet* décide d'installer le trio Heapster, InfluxDB et Grafana. Cela signifie que Grafana peut changer de position dans le Cluster à l'improviste et n'aura donc pas une adresse IP fixe pour y accéder. Comme perspective, il serait préférable de mettre en place une sorte de proxy dont l'IP est fixe par lequel on accède à Grafana partout où il soit.

Conclusion

Nous avons décrit l'installation et l'utilisation d'une plate-forme en tant que service au-dessus d'un Cluster de CoreOS. La PaaS, couplée avec son interface utilisateur et un système de supervision, devient plus complète et favorise le déploiement continue dans un monde containérisé.

Conclusion

Notre stage de fin d'études consistait à mettre en œuvre une architecture Cloud basée sur des conteneurs. Le but de ce projet est de faciliter et simplifier aux développeurs de *Sayoo* le développement, la personnalisation de nouveaux modules *Odoo* (Open ERP), et faciliter le processus de déploiement continue.

Certes, c'est *Odoo* qui a poussé à avoir recours au Cloud, mais la solution réalisée est conçu de façon général pour la provision des SaaS légers (containérisés). Toutefois, elle n'est pas complète. En effet, Nous aimerions ajouter un service de mesure et de facturation ainsi que d'améliorer le processus de scalabilité et de l'automatiser.

Par ailleurs, nous suivrons avec attention le projet *Flocker* qui s'intéresse à la containérisation des bases de données dans le but de perfectionner l'orchestration des applications *Stateful*. Nous continuerons à surveiller le projet Deis qui intégrera Kubernetes et Mesos dans un futur proche, ce qui rendra l'architecture encore plus performante et optimale. Enfin, nous développerons des interfaces utilisateurs (UI) afin de simplifier la gestion entière de l'architecture et éviter de passer par les lignes de commandes.

Webographie

[1] Rackspace SUPPORT. *Understanding the Cloud Computing Stack : SaaS, PaaS, IaaS*. URL : http://www.rackspace.com/knowledge_center/whitepaper/understanding-the-cloud-computing-stack-saas-paas-iaas.

[2] WIKIPEDIA. *Cloud Computing*. URL : https://fr.wikipedia.org/wiki/Cloud_computing.

[3] ANAND MANI SANKAR. *Containers (Docker) : A disruptive force in cloud computing*. URL : http://anandmanisankar.com/posts/container-docker-PaaS-microservices/.

[4] IBM. *KVM and Docker LXC Benchmarking with OpenStack*. URL : http://fr.slideshare.net/BodenRussell/kvm-and-docker-lxc-benchmarking-with-openstack.

[5] MICHAEL PAGES. *Docker Présentation – Part 1*. URL : http://blog.ippon.fr/2014/04/14/docker-presentation-part-1/.

[6] GOOGLE. *Scaling clusters declaratively with Kubernetes and Docker realisation*. URL : https://www.youtube.com/watch?v=IR9UhW8k6Ag.

[7] GOOGLE. *Containerizing the Cloud with Docker on Google Cloud Platform*. URL : https://www.youtube.com/watch?v=tsk0pWf4ipw.

[8] Kasia LORENC. *Docker Goes After Enterprise With Orchestration Services And Hub*. URL : http://www.tomsitpro.com/articles/docker-enterprise-hub-orchestration,1-2375.html.

[9] DOCKER. URL : https://www.docker.com/.

[10] Sreenivas MAKAM. *Docker Compose and Interworking of Docker Machine, Swarm, Compose*. URL : https://sreeninet.wordpress.com/2015/05/31/docker-compose-and-docker-machine-swarm-compose-interworking/.

[11] METEORHACKS. *Kubernetes : The Future of Cloud Hosting*. URL : https://meteorhacks.com/learn-kubernetes-the-future-of-the-cloud.

[12] Mitchell ANICAS. *An Introduction to Mesosphere*. URL : https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere.

[13] Ben LORICA. *Running batch and long-running, highly available service jobs on the same cluster*. URL : http://radar.oreilly.com/2013/09/running-batch-and-long-running-highly-available-service-jobs-on-the-same-cluster.html.

[14] COREOS. *Container management and deployment for your cluster using fleet*. URL : https://coreos.com/using-coreos/clustering/.

[15] COREOS. *Containerizing the Cloud with Docker on Google Cloud Platform*. URL : http://coreos.com.

[16] DAMIEN METZLER. "CoreOS and Nuxeo : How We Built nuxeo.io". In : *nuxeo* (2014). DOI : http://www.nuxeo.com/blog/coreos-nuxeo-build-nuxeoio/.

[17] COREOS. *Running CoreOS on EC2*. URL : https://coreos.com/docs/running-coreos/cloud-providers/ec2/.

[18] ARKENIO. *Gogeta*. URL : https://github.com/arkenio/gogeta.

[19] DEIS. *Docker based PaaS to deploy and manage applications on cluster*. URL : http://deis.io/.

[20] DEIS. *Deis architecture*. URL : http://docs.deis.io/en/latest/understanding_deis/architecture/.

[21] DEIS. *Installing Deis on Google Compute Engine*. URL : http://deis.io/.

[22] JUMBOJETT. *full client-side web app that interfaces with deis api*. URL : https://github.com/jumbojett/deis-ui.

[23] GOOGLE. *Installing Heapster on CoreOS*. URL : https://github.com/GoogleCloudPlatform/heapster/blob/master/docs/coreos.md.

[24] HEROKU. *The twelve-factor app methodology for building robust SaaS*. URL : http://12factor.net/.

[25] WIKIPEDIA. *Amazon Web Services*. URL : https://fr.wikipedia.org/wiki/Amazon_Web_Services.

[26] Dominique FILIPPONE. "Google Compute Engine : ses atouts pour l'emporter face à AWS". In : *journaldunet* (2014). DOI : http://www.journaldunet.com/solutions/cloud-computing/google-compute-engine.shtml.

Index

ANNEXE A

Odoo Sous Docker

Dans ce qui suit, nous allons donner une idée sur l'interface que fournit Docker pour la provision des services (conteneurs). Nous allons réaliser la mini-architecture suivante :

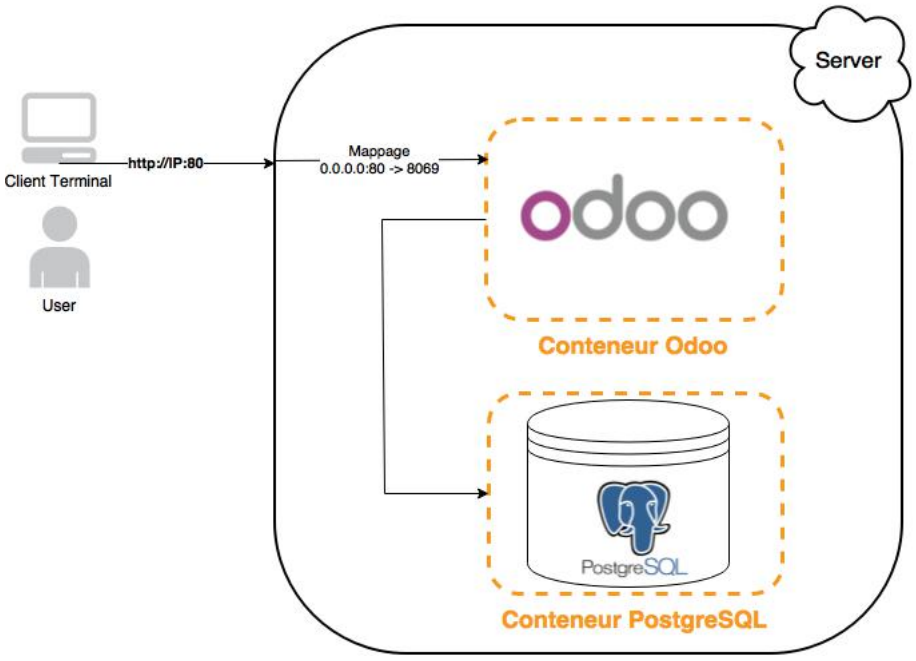

FIGURE A.1 – Odoo sous Docker

Environnement technique

Nous allons partir d'un serveur Ubuntu 14.04-64 bits avec un noyau >= 3.8 et Docker installé (L'installation ne sera pas détaillée), ou de préférence, un serveur CoreOS où Docker est déjà intégré nativement avec ce système d'exploitation.

Installation de PostgreSQL

Odoo a besoin d'un serveur de base de données PostgreSQL, la commande suivante permet de l'installer :

```
1  $ docker run -d -e POSTGRES_USER=odoo -e POSTGRES_PASSWORD=odoo --name
   db postgres
2
```

Script A.1 – Installation de PostgreSQL

Installation d'Odoo

L'installation d'Odoo se fait grâce à la commande suivante :

```
1  $ docker run -p 80:8069 --name odoo --link db:db -t odoo
2
```

Script A.2 – Installation d'Odoo

En quelques secondes, nous avons pu lancer le service Odoo prêt à être utilisé, personnalisé, et éventuellement porté vers un autre serveur. Pour accéder au service il suffit de taper dans le navigateur :

```
1  http://IP_SERVEUR:80
2
```

ANNEXE B

Docker-compose

Nous allons réaliser l'installation détaillée dans l'annexe A avec l'outil *Docker-compose*. Cet outil permet de définir dans un fichier YAML l'architecture de l'application. Enfin, on peut faire des manipulations (démarrage, redémarrage, arrêt) sur toute l'architecture comme si c'était un seul service.

```
    odoo:
        image: odoo
        links:
          - db
        ports:
          - "80:8069"
    db:
        image: postgres
        environment:
          - POSTGRES_USER: odoo
          - POSTGRES_PASSWORD: odoo

```

Script B.1 – Installation d'Odoo avec Docker-compose

```
    $ docker-compose up -d

```

Script B.2 – Lancement d'Odoo avec Docker-compose

ANNEXE C

Douze-facteurs d'un SaaS

Dans l'informatique moderne, le logiciel est généralement livré comme un service : appelés SaaS. Les douze-facteurs d'un SaaS est une méthodologie pour la création d'applications qui :

- Réduit le temps et le coût pour les nouveaux développeurs participant au projet ;
- offre une **portabilité maximale** entre les environnements d'exécution ;
- conçoit des applications pour le **déploiement** sur les **plates-formes de Cloud** modernes, éliminant ainsi la nécessité de serveurs et d'administration de systèmes ;
- **Réduit la divergence** entre le développement et la production, permettant un déploiement continu ;
- favorise **monter en charge** sans toucher aux outils, à l'architecture, ou à les pratiques de développement.

La méthodologie des douze-facteurs synthétise l'expérience de l'équipe de Heroku, un des tout premiers services Cloud [24]. Elle peut être appliquée à des applications écrites dans n'importe quel langage de programmation, et qui sont combinées à n'importe quel services (base de données, file d'attente, mémoire cache, etc.).

code source

Le code source de l'application doit être unique, mis et suivi depuis un système de contrôle de version comme Git.

Dépendances

La plupart des langages de programmation offre un système de gestion de dépendances. Ainsi, une application doit déclarer explicitement et isoler les dépendances. Un SaaS robuste ne repose jamais sur l'existence implicite des dépendances.

Configuration

Les variables de configurations peuvent varier entre les environnements (développement, test, production, etc). Ces variables ne doivent jamais être stockées dans des constantes à l'intérieur du code. Une bonne pratique c'est de les stocker dans des variables d'environnement.

Services de support

Traiter tous les services de support (base de données, file d'attente, serveur SMTP, etc.) comme des ressources liées.

Build, release, run

Pour déployer une application, il faut passer par trois étapes :
— build : transformer le code source en exécutable ;
— release : combiner l'exécutable avec les configurations ;
— run : lancer les processus dans l'environnement d'exécution.

Processus

Les processus d'une application doivent être **sans état** (stateless). Les données persistantes doivent être stockées dans une base de données.

Attachement de port

Chaque service doit être exporté via un port.

Concurrence

Les processus d'une application ne doivent jamais lancés comme un démon. Au lieu de cela, il faut compter sur le gestionnaire de processus du système d'exploitation (systemd, init.d, upstart, etc.).

Jetable

Les processus d'une application sont jetables, ils peuvent être stoppés ou démarrés à n'importe quel moment rapidement et sans causer de problèmes.

Parité Dev/Prod

Il faut garder tous les environnements (développement, test, production, etc) aussi similaires que possible.

Messages logs

Les messages logs sont très importants, il faut les traiter comme des flux d'événements.

Processus admin

Les tâches d'administration doivent être exécutés dans un unique processus avec un environnement identique à celui du processus principal.

ANNEXE D

Fournisseurs du Cloud

Amazon Web Services

AWS est une collection de services informatiques distants fournis via internet par le groupe américain de commerce électronique Amazon.com. Lancé officiellement en 2006, Amazon Web Services fournit des services en lignes à d'autres sites internet ou applications clientes. La plupart d'entre eux ne sont pas directement exposés à l'utilisateur final, mais offrent des fonctionnalités que d'autres développeurs peuvent utiliser. Parmi les services qu'offre AWS, on trouve Elastic Compute Cloud (EC2), fournissant des serveurs virtuels évolutifs utilisant Xen [25].

AWS offre aux nouveaux clients une version d'évaluation de 12 mois, ce qui a permis d'utiliser le service EC2 dans ce projet pour créer un cluster de trois serveurs.

Google Compute Engine

Google est présent depuis de nombreuses années sur les services SaaS et PaaS via Google Apps et Google App Engine. Ce n'est qu'en décembre 2013 que Google a fait son entrée sur le marché du IaaS avec une version finale.

Sur le marché du IaaS, Amazon EC2 est aujourd'hui la référence en termes de nombre d'utilisateurs, de qualité de service et de fonctionnalités, mais Google a fait une entrée assez fracassante avec une offre qui dépasse déjà le reste du marché. En effet, Google propose une volumétrie de stockage plus importante, une meilleure bande passante ainsi qu'un temps de démarrage des machines virtuelles optimisé [26].

GCE offre aux nouveaux clients 300 dollars virtuels à consommer en moins de 2 mois. Ce qui a permis de tester cette IaaS dans ce projet pour créer un cluster où Deis a été installé.

Heroku

Heroku est un service de Cloud Computing de type plate-forme en tant que service. Créé en 2007, il était l'un des tout premiers services Cloud, puis a été racheté par Salesforce.com. Cette plate-forme est destinée pour les développeurs pour y mettre leurs applications programmées dans la plupart des technologies web.

Heroku n'a pas été utilisé durant notre projet, mais il a influencé le projet indirectement. En effet, Heroku a tellement inspiré Deis de sorte qu'elle a été décrite comme un mini-Heroku privé. Par ailleurs, l'équipe de Heroku a résumé son expérience dans la scalabilité des applications et a introduit les douze-facteurs d'un SaaS