Introduction à Kubernetes

CLOUD COMPUTING
IATIC 5 2019/2020

Clément LEFEVRE

Étienne SAUVEE

Pierre VERDURE

ISTY

Table des matières

Table des figures

Introduction

1. Contexte du projet

Ce projet s'intègre dans le cadre du module *Cloud Computing* de notre dernière année de cycle ingénieur en informatique à l'ISTY (Institut des Sciences et Techniques des Yvelines), et est encadré par M. Nadjib AIT SAADI, notre enseignant.

Le but principal du projet était de développer un socle de connaissances important sur une technologie liée au *Cloud Computing*, et de le partager avec l'ensemble de la promotion ensuite, par le biais d'une présentation orale et d'un rapport écrit.

Parmi les nombreux sujets proposés, nous avons choisi de travailler sur *Kubernetes*, car nous le connaissions relativement peu, et il était intimement lié aux problématiques *DevOps* sur lesquels nous avions eu l'occasion de travailler en milieu professionnel.

2. Historique de Kubernetes

A. Origines

Les origines de *Kubernetes* remontent aux années 2003-2004, avec l'introduction de *Borg*, un système interne de gestion de clusters par *Google*. Issu d'une équipe composée initialement de quelques personnes, *Borg* s'est développé au fil des années, jusqu'à devenir un système de gestion de clusters à grande échelle, exécutant des centaines de milliers de jobs, de plusieurs milliers d'applications, sur plusieurs clusters, chacun composé de plus de 10 000 machines.

En 2013, soit 10 ans après le lancement de *Borg*, *Google* introduit un autre système de gestion de clusters, *Omega*, un ordonnanceur flexible et évolutif dédié aux grands clusters.
Né d'une volonté d'améliorer l'ingénierie logicielle de l'écosystème de *Borg*, *Omega* a été totalement repensé, dans le but d'obtenir une architecture plus cohérente, fondée sur des principes, tout en conservant les modèles qui avaient fait leurs preuves sur Borg.

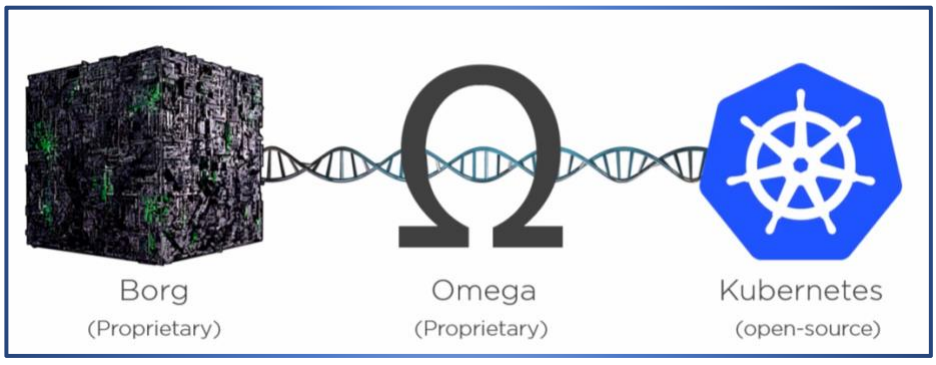


Figure 1 : Origines de Kubernetes

B. Naissance et premières versions

A la mi-2014, naît *Kubernetes*, présenté comme une version open-source de Borg. Le terme *Kubernetes* est d'ailleurs issu du grec κυβερνήτης et signifie ''pilote''.
A cette époque, les containers Linux se démocratisent dans le monde professionnel, et l'infrastructure cloud public de *Google* (*Google Cloud Platform*) se développe. En réalisant *Kubernetes*, *Google* offre donc son savoir-faire au monde open-source, tout en étoffant son offre cloud.

En comparaison avec *Borg* et *Omega*, *Kubernetes* a lui été développé afin d'améliorer l'expérience des développeurs d'applications s'exécutant dans un cluster. Son principal objectif est de faciliter le déploiement et la gestion de systèmes distribués complexes, tout en bénéficiant des avantages de la containerisation.
Les premiers collaborateurs, *Microsoft*, *RedHat*, *IBM*, et *Docker* rejoignent le projet en Juin.
Le 21 Juillet 2015, la version 1.0 de *Kubernetes* est réalisée. D'autres contributeurs majeurs, parmi lesquels *Huawei*, *OpenShift*, et *Deis* rejoignent le projet en novembre.
Enfin, le même mois se tient la première *KubeCon* à San Fransisco, conférence visant à rassembler la communauté de *Kubernetes*.

C. Démocratisation et croissance

2016 peut être considérée comme l'année de la démocratisation de *Kubernetes*. En Février, *Helm*, son propre gestionnaire de paquets, est réalisé.
Le projet atteint son rythme de quatre versions majeures par an, toujours maintenu actuellement.
La version 1.2, publiée en Mars, vise à simplifier le déploiement d'applications, et apporte des améliorations majeures telles que la mise à l'échelle automatique.
Enfin, en décembre, *Kubernetes* supporte *OpenAPI*, permettant ainsi aux fournisseurs d'APIs de définir leurs opérations et leurs modèles.

2017 est pour *Kubernetes* l'année du soutien aux entreprises, avec des versions majeures telles que la 1.7, apportant de nouvelles fonctionnalités pour l'orchestration des containers : chiffrement des secrets, extensibilité, etc.
En août, *Github* tourne sous *Kubernetes*, et en septembre, la *CNCF* (*Cloud Native Computing Foundation*) annonce les premiers fournisseurs d'accès certifiés (les KCSP pour *Kubernetes Certified Service Provider*).
Oracle rejoint le projet en tant que collaborateur, et *Docker*, bien que possédant son propre ordonnanceur, *Docker Swarm*, embrasse totalement *Kubernetes*.

Enfin, 2018 est l'année d'adoption des fournisseurs d'accès cloud, avec notamment l'adoption de *Kubernetes* par *Digital Ocean*.
Amazon développe un service (*EKS* pour *Elastic Kubernetes Service*) dédié à *Kubernetes*, et permettant de simplifier le processus de création, de sécurisation, d'exploitation et de maintenance, en ajoutant une couche d'abstraction supplémentaire.
De son côté, *Microsoft* développe *AKS* (*Azure Kubernetes Service*), qui lui aussi facilite le déploiement et la gestion d'application containerisées, en offrant une expérience CI/CD (intégration et déploiements continus).

Prérequis

1. Cluster

Un cluster est un groupe de serveurs indépendants fonctionnant comme un seul et même système. Les serveurs d'un cluster sont généralement situés à proximité les uns des autres, et interconnectés par un réseau.

Les clusters sont généralement conçus pour les applications dont les données sont fréquemment mises à jour, par exemple pour des serveurs de fichiers, des serveurs d'impression, des serveurs de bases de données, etc.

En règle générale, un cluster est constitué de nœuds de calcul, de nœuds de stockage, et de nœuds frontaux. On compte parfois des nœuds additionnels dédiés au monitoring

Un cluster *Kubernetes* doit avoir un maître (master) : le système qui commande et contrôle toutes les autres machines du cluster. Nous détaillerons l'architecture d'un cluster *Kubernetes* dans la partie dédiée de ce rapport.

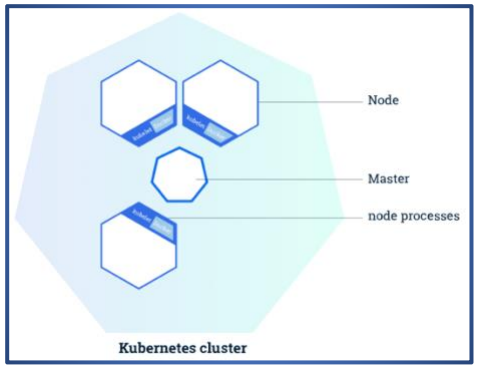


Figure 2 : Cluster Kubernetes

2. Namespace

Un *namespace* (espace de noms) désigne un ensemble virtuel de clusters sauvegardés par le même cluster physique. Les *namespaces Kubernetes* peuvent ainsi être considérés comme des entités logiques utilisées afin de représenter les ressources du cluster pour un ensemble d'utilisateurs donné.

Par défaut, *Kubernetes* possède trois *namespaces* :

- *default* : le namespace par défaut pour les objets sans autre *namespaces*.
- *kube*-public : *namespace* pour les ressources accessibles et lisibles par tous les utilisateurs. Généralement, ce *namespace* est réservé à l'utilisation du cluster, au cas où certaines ressources devraient être accessibles et lisibles depuis l'extérieur du cluster.
- *kube*-system : *namespace* pour les objets et les ressources créés par le système *Kubernetes*.

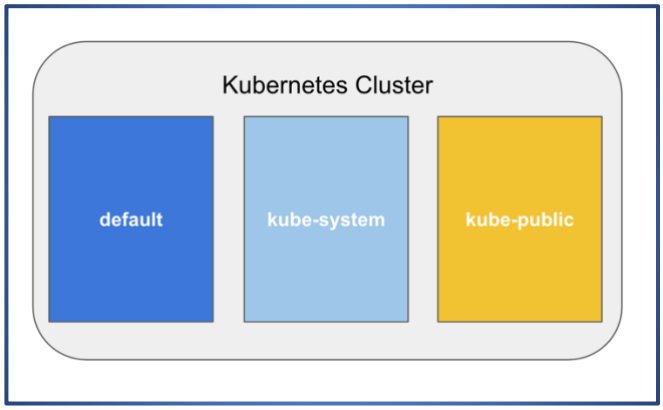


Figure 3 : Namespaces dans Kubernetes

3. Containers

Les containers sont un moyen standard de joindre le code, les configurations et les dépendances d'une application en un seul objet.

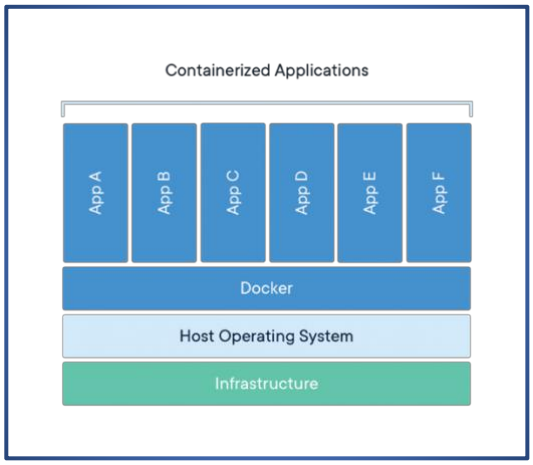


Figure 4 : Architecture d'un container

Les containers partagent un système d'exploitation installé sur le serveur et s'exécutent en tant que processus à ressources isolées, assurant des déploiements rapides, fiables et cohérents, quel que soit l'environnement.

Ainsi, plutôt que d'avoir des dépendances disséminées dans tout l'environnement, les containers permettent, grâce à la containerisation, de séparer complètement des applications qui se seraient exécutées simultanément sur un serveur.

Néanmoins, dans la mesure où tout est géré au sein d'une unité unique, l'architecture du container a une incidence majeure sur la sécurité.

Le principe de fonctionnement des containers est présenté dans la figure ci-dessous.

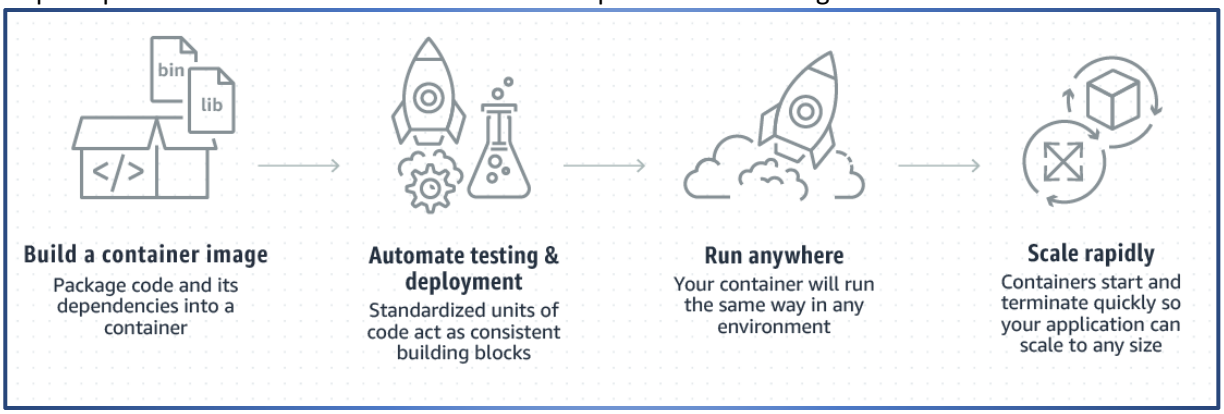


Figure 5 : Fonctionnement des containers

Description

Kubernetes est l'orchestrateur le plus populaire pour gérer les containers, c'est un outil permettant de les contrôler harmonieusement.

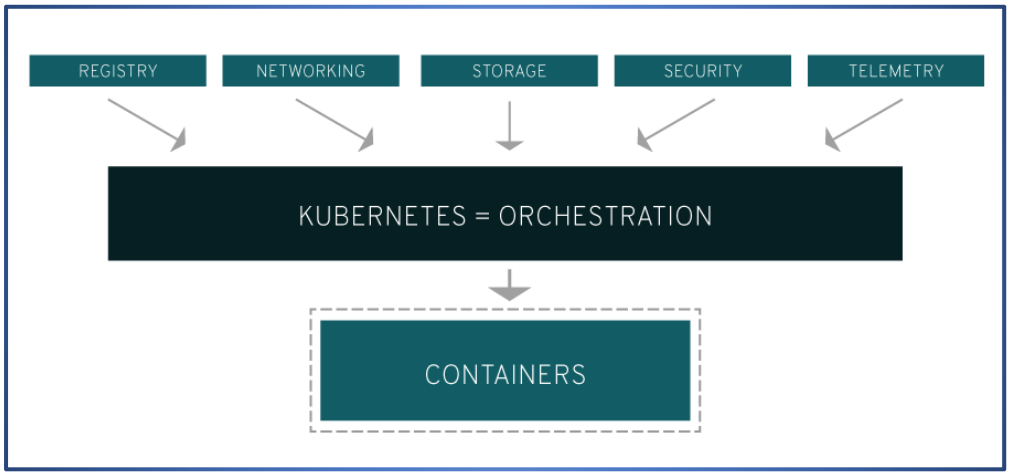


Figure 6 : Description de Kubernetes

Il a été conçu pour gérer complètement le cycle de vie des applications et services containerisées à l'aide de méthodes qui fournissent la prévisibilité, l'évolutivité et la haute disponibilité.

Kubernetes permet notamment de déployer des applications plus rapidement grâce à différents mécanismes explicités ci-dessous :

- Mise à l'échelle horizontale de l'infrastructure : de nouveaux serveurs peuvent être ajoutés ou supprimés facilement.
- Mise à l'échelle automatique de l'infrastructure : le nombre de containers en cours d'exécution est modifié automatiquement, en fonction de l'utilisation du processeur ou d'autres mesures fournies par l'application.
- Mise à l'échelle manuelle : le nombre de containers en cours d'exécution est également configurable manuellement.
- Contrôle d'intégrité et autoréparation : l'intégrité des nœuds et des containers est constamment vérifiée, en s'assurant que votre application ne rencontre aucune défaillance.
- Routage du trafic : permet d'envoyer des requêtes aux containers appropriés.
- Déploiements et restaurations automatisés : les déploiements de nouvelles versions ou de mises à jour sont gérés sans temps d'arrêt et l'intégrité des containers est surveillée.
  Si le déploiement rencontre des erreurs, il est automatiquement annulé et les équipes sont notifiées.
- Déploiements *Canary* : Les déploiements *Canary* permettent de tester le nouveau déploiement en production en parallèle avec la version précédente.

Cas d'utilisations

Au sein de cette partie, nous détaillerons quelques exemples de cas d'utilisations de *Kubernetes*, en montrant les apports réels de cette technologie pour chacun d'eux.

1. Orchestration

Kubernetes est un orchestrateur de containers. La principale mission de ce dernier est d'automatiser le cycle de vie des containers : le déploiement, les mises à jour, le contrôle d'état et les procédures de basculement.
Kubernetes se focalise sur la façon dont les applications doivent fonctionner, plutôt que sur des détails spécifiques d'implémentation.
De plus, il permet d'éviter l'enfermement propriétaire : les applications sont agnostiques à l'infrastructure physique. Seule la configuration du cluster *Kubernetes* est spécifique à chaque hôte.

2. Déploiement d'une simple application

Le premier cas d'utilisation est relativement simple, il consiste à déployer une simple application, par exemple d'architecture 3-tiers à l'aide de *Kubernetes*.
Bien que *Kubernetes* soit assez complexe à prendre en main, il ne faut pas négliger le but applicatif et éducatif. En effectuant ce déploiement, on apprend à utiliser *Kubernetes*, et par la suite, si l'application se développe, sa gestion, sa configuration et son administration seront fortement simplifiées.

3. Déploiement d'une application micro-services

Ce second cas d'utilisation est la raison précise de la création de *Kubernetes*. Considérons une application relativement complexe déployée sur un cluster *Kubernetes*.

Cette application possède de nombreux composants, qui communiquent les uns avec les autres. *Kubernetes* aide à gérer ces communications, à travers des tâches telles que la détection de problèmes de communication entre les composants, ou la gestion des processus d'authentification.
De plus, *Kubernetes* gère la mise à l'échelle, et c'est un atout majeur pour ce type d'applications, pour lesquelles il est souvent nécessaire de mettre à l'échelle un seul composant plutôt que l'application entière.
Ainsi, pour les applications de cette architecture, *Kubernetes* simplifie l'ensemble du processus de gestion des composants de l'application et réduit considérablement le travail nécessaire pour que l'application soit opérationnelle.

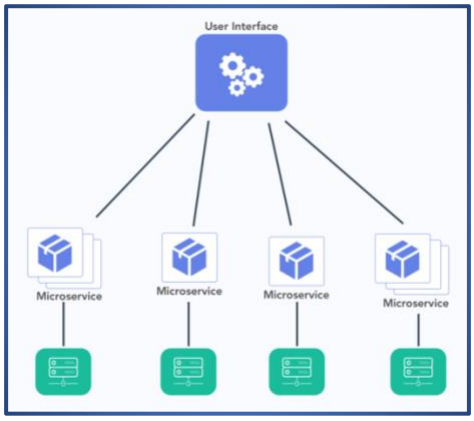


Figure 7 : Déploiement d'une application micro-services

4. Lift & Shift

Le lift & shift est un cas particulièrement utilisé actuellement, puisque de nombreuses applications sont migrées vers le cloud. Imaginons la situation suivante : nous avons une application déployée sur des serveurs physiques dans un centre de données classique. Pour des raisons économiques et pratiques, il a été décidé de la transférer dans le cloud (soit sur une VM, soit sur des *Pods* dans *Kubernetes*).

Dans un premier temps, l'application (qui fonctionne en dehors du cloud), est transférée dans *Kubernetes*. Elle est ensuite divisée en composants plus petits pour devenir une application cloud native.
Kubernetes agît donc ici comme une passerelle pour la migration d'une application dans le cloud.

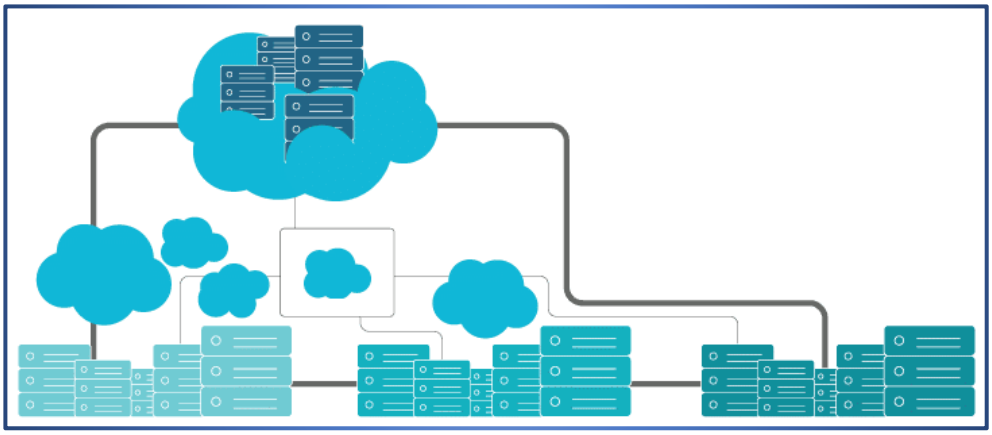


Figure 8 : Lift & Shift

5. Cloud-native Network Functions (CNF)

Il y a quelques années, les services réseaux des grandes entreprises de télécommunications étaient basés sur du matériel tel que des pare-feux ou des équilibreurs de charge fournis par des sociétés spécialisées dans le matériel réseau.
Si de nouvelles fonctionnalités étaient développées, les opérateurs devaient mettre à niveau le matériel existant. De la même manière, lorsqu'une mise à jour du micro-logiciel de l'appareil n'était pas possible, du matériel supplémentaire devait être acheté.
Pour remédier à cet inconvénient, les opérateurs télécoms ont choisi de disposer de tous ces services réseau en tant que logiciels et d'utiliser les machines virtuelles et *OpenStack* pour la virtualisation des fonctions réseau (VNF).
Ils souhaitent désormais aller plus loin, et utiliser des containers dans le même but. Cette approche s'appelle CNF (Cloud-native Network Functions). Dans cette optique plusieurs projets R&D sont à l'étude, pour la migration des VNF depuis des VM vers des containers.
Pour ce cas d'utilisation, *Kubernetes* serait responsable de l'orchestration des containers, mais également de la gestion du trafic réseau.

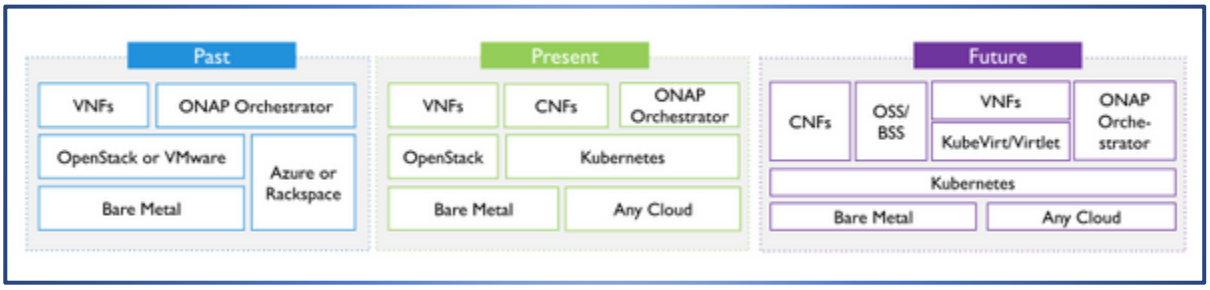


Figure 9 : Transition des VNFs vers les CNFs

6. Machine Learning

De nos jours, les techniques d'apprentissage automatique sont massivement utilisées pour résoudre des problèmes concrets, dans de nombreux domaines (véhicules autonomes, reconnaissance d'images, etc.). Cependant, le processus de construction d'un modèle d'IA efficace et de son utilisation en production est compliqué et chronophage.
Pour accélérer l'ensemble du processus, on peut utiliser *Kubernetes*, afin de réduire considérablement le nombre d'opérations nécessaires au déploiement d'une telle application. De plus, les calculs nécessaires à la formation du modèle d'apprentissage machine sont effectués à l'intérieur du cluster *Kubernetes*, ce qui simplifie considérablement les tâches des *data scientistes*.

Une des grandes annonces de la conférence *KubeCon + CloudNativeCon* de décembre 2017 concernait *Kubeflow* : un projet open source dédié à rendre l'apprentissage machine sur *Kubernetes* facile, portable et évolutif. *Kubeflow* s'adresse aux utilisateurs qui souhaitent des plateformes portables, plus de contrôle, une simplification des composantes technologies d'apprentissage machine et des déploiements via *Kubernetes*.

7. Puissance de calcul

Récemment, une employée de *Google*, Emma Haruka Iwao, a battu le record du monde de calcul du nombre de décimales du nombre Pi, en dénombrant plus de 31 000 milliards de chiffres.
De tels calculs nécessitent une énorme puissance de calcul, c'est pourquoi l'utilisation d'un cluster *Kubernetes* serait pertinente pour gérer la répartition des calculs sur plusieurs ordinateurs. Il suffirait donc d'écrire le programme, puis de laisser *Kubernetes* gérer les machines automatiquement.

Figure 10 : Emma Haruka Iwao

8. DevOps (CI/CD)

Kubernetes apporte également des avantages considérables à la méthodologie d'intégration et de déploiements continus. C'est d'ailleurs une suite logique des cas d'utilisation 1, 2, et 3.
Une fois l'application déployée en production, son fonctionnement doit être surveillé en permanence, afin d'offrir le plus de feedback possible aux développeurs.
Nous développerons cet aspect de *Kubernetes* au sein de la partie *Intégration dans un workflow DevOps*.

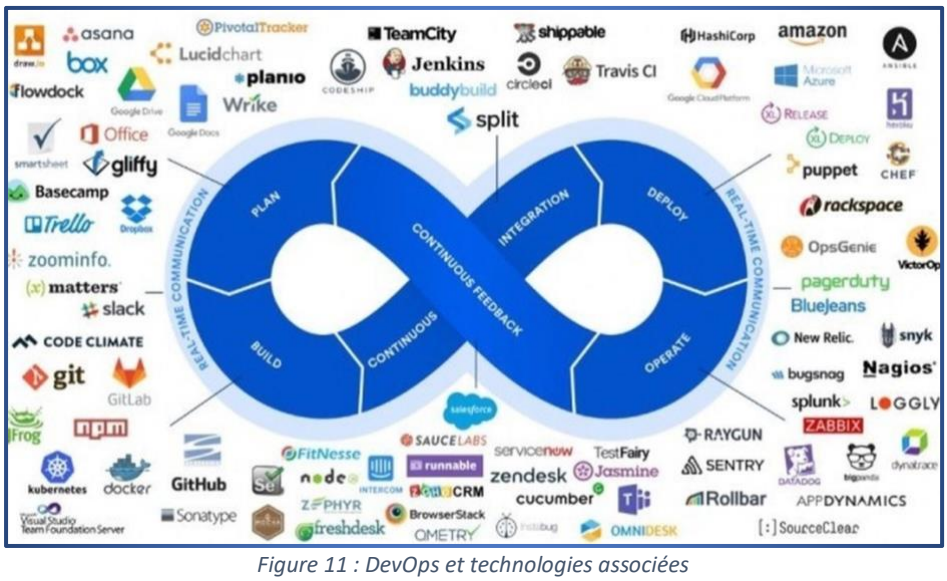


Figure 11 : DevOps et technologies associées

Architecture

1. Macroarchitecture

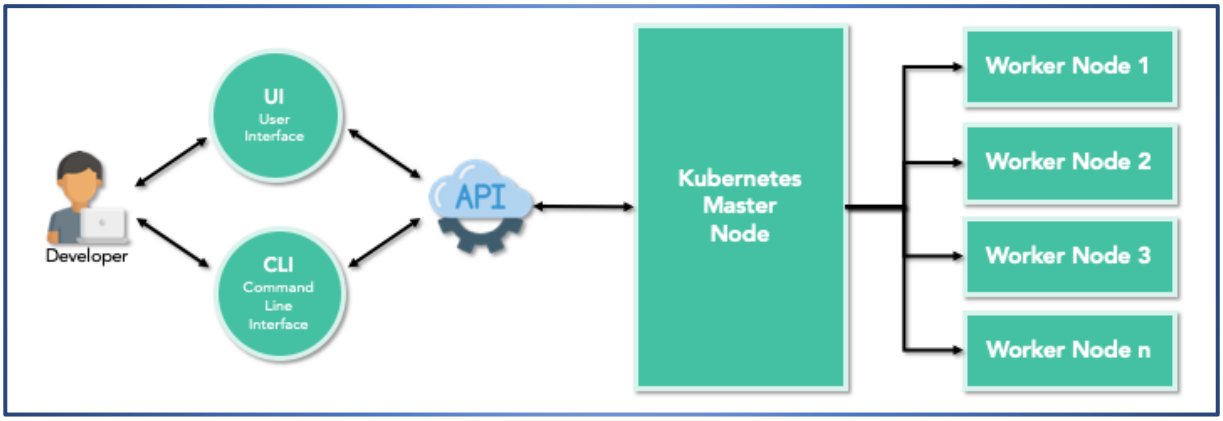


Figure 12 : Macroarchitecture

Comme la plupart des plateformes distribuées, Kubernetes est composée d'au moins un maître, le *Master Node*, et de plusieurs nœuds esclaves, les *Worker Nodes*.

L'utilisateur peut interagir avec *Kubernetes* de deux manières :

- Graphiquement, via une interface utilisateur (UI),
- Depuis un terminal, via une interface en ligne de commande (CLI).

Ces deux entités communiquent ensuite avec *Kubernetes* via une API REST, exposée par le *Master Node*.

Le *Master Node* est donc responsable de l'exposition de l'API avec laquelle interagît l'utilisateur, mais également de la planification des déploiements, et de la gestion du cluster dans son ensemble. Ainsi, il **identifie** et **surveille** l'état des *Worker Nodes*.

Les *Worker Nodes* **exposent** des **ressources de calcul**, **de stockage**, et **de réseau** aux applications. Ils possèdent chacun un environnement d'exécution containerisé (par exemple Docker ou RKT), et communiquent avec le *Master Node* par le biais d'un agent.

Pour la suite de cette partie, afin de détailler l'architecture, nous considèrerons uniquement un *Master Node* et un *Worker Node*. Toutefois, cette configuration est assez rare, puisqu'il est relativement commun d'avoir plusieurs *Master Nodes* et *Worker Nodes* au sein d'un cluster *Kubernetes*.

2. Master Node

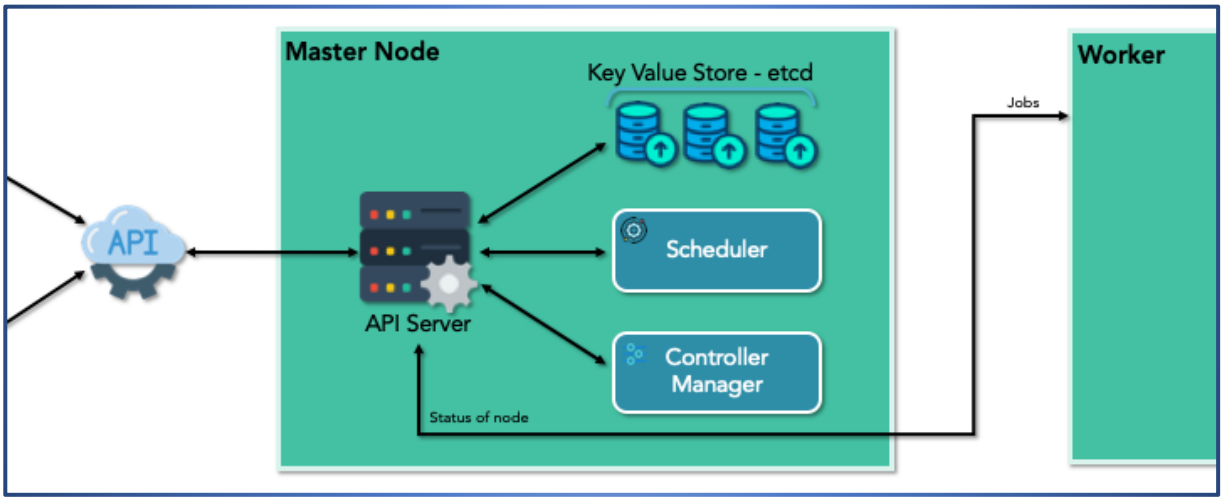


Figure 13 : Architecture du Master Node

Comme nous l'avons mentionné précédemment, le *Master Node* permet de charger les informations du cluster pour ainsi planifier le chargement, identifier, surveiller l'état des *Worker Nodes* et de tracer leurs informations.

Pour permettre cela, le *Master Node* est composé de plusieurs composants.

B. API Server

L'*API Server* est l'entité de gestion centrale qui reçoit l'ensemble des requêtes REST pour les modifications (*Pods*, *Services*, réplication, etc.).
Il peut être considéré comme étant le front-end (devanture) du cluster *Kubernetes*, et sert de pont entre les différents composants pour maintenir l'intégrité du cluster et diffuser les informations et les commandes.
Ainsi, il s'agît du seul composant qui communique avec le cluster *etcd*, pour s'assurer que les données sont correctement stockées et cohérentes (en accord avec la réalité).

C. L'etcd

L'*etcd* est un cluster de bases de données qui stocke les données liées au cluster *Kubernetes* dans un format *Key-Value* (clé-valeur). Les informations stockées sont par exemple :

- Le nombre de *Pods,*
- Leur état,
- Leur *namespace*, etc.

Ils sont uniquement accessibles depuis l'API Server, pour des raisons de sécurité.

Comme la plupart des autres composants du plan de contrôle, l'*etcd* peut être configuré sur un seul *Master Node*, ou, dans les scénarios de production, réparti sur plusieurs machines. Il doit simplement être connecté en réseau depuis chacune des machines du cluster *Kubernetes*.

D. Le Scheduler

Le *Scheduler* affecte les charges de travail aux *Nodes* du cluster. Il est responsable de l'ordonnancement des *Pods*. Il permet de sélectionner quel *Node* devrait faire tourner un *Pod* non ordonnancé, en se basant sur la disponibilité des ressources.
Pour ce faire, il lit les exigences opérationnelles d'une charge de travail, analyse l'environnement et affecte le travail sur un ou plusieurs *Nodes* en fonction de ces analyses.

Il gère l'utilisation des ressources sur chaque *Node* afin de s'assurer que la charge de travail n'est pas en excès par rapport aux ressources disponibles.
Il identifie les *Worker Nodes* selon leur taille, leur capacité, leur stockage actuel et le proxy auquel ils sont attachés ainsi que d'autres contraintes pour leur attribuer les containers adéquats.

E. Le Controller Manager

Enfin, le *Controller Manager* gère principalement les différents contrôleurs qui régulent l'état du cluster, gère les cycles de vies des charges de travail, et effectuent des tâches de routine.

On retrouve les contrôleurs suivants :

- *Replication Controller* : responsable du maintien du bon nombre de *Pods*.
- *Node Controller* : responsable de la détection et de la résolution d'une panne de *Nodes*.
- *Endpoints Controller* : responsable du remplissage des objets *Endpoints.*
- *Service Account & Token Controllers* : responsables des comptes et des tokens pour les nouveaux *namespaces*.

De manière générale, il permet de vérifier l'attribution des tâches (i.e. il vérifie que quelque part dans le cluster un certain nombre de *Worker Nodes* travaillent sur l'accomplissement d'une tâche).
Il communique avec l'*API Server* afin de lui indiquer les *Pods* à créer ou supprimer en fonction de la charge restante pour la complétion d'une tâche.

3. Worker Node

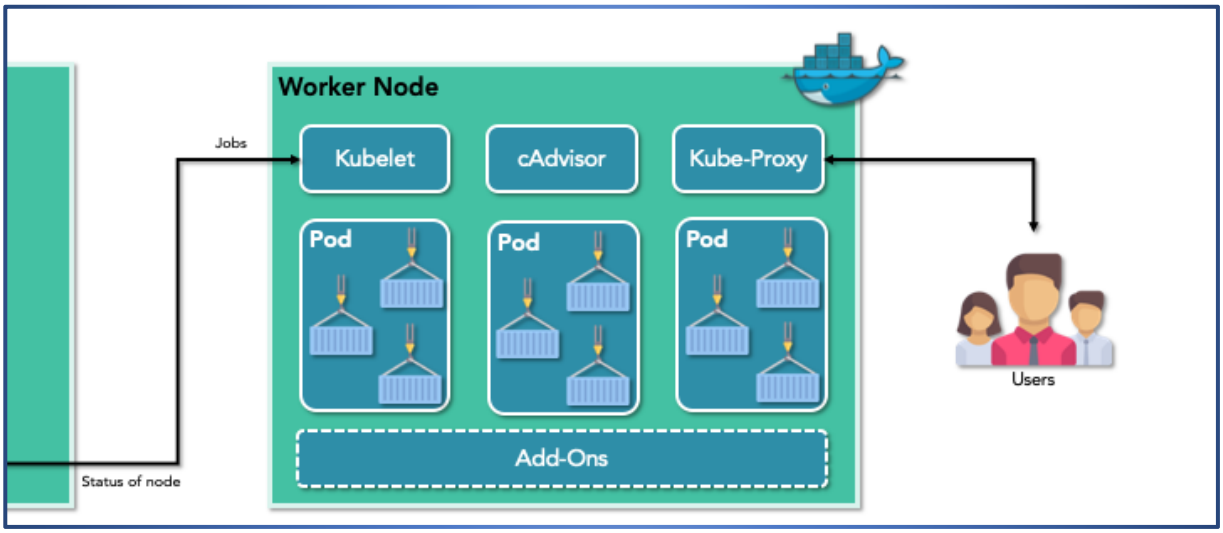


Figure 14 : Architecture du Worker Node

Le *Worker Node*, également appelé *Minion*, est une machine au sein de laquelle les containers sont déployés.

A. Environnement de containerisation

Chaque *Worker Node* du cluster *Kubernetes* doit exécuter un environnement de containerisation, tel que *Docker*, *rkt* ou *runc*.
Cet environnement de containerisation est responsable du démarrage et de la gestion des applications containerisées. Chaque unité de travail sur le cluster est, à son niveau de base, implémentée comme un ou plusieurs containers qui doivent être déployés.
L'environnement d'exécution du container sur chaque *Worker Node* est le composant qui exécute enfin les containers définis dans les charges de travail soumises au cluster.

B. Les Pods

Un *Worker Node* est composé d'un ou plusieurs *Pods*. Un *Pod* fait généralement référence à un ou plusieurs containers, qui doivent être contrôlés comme une seule application. En effet, les containers d'un Pod partagent le même espace de stockage et sont exécutés dans un contexte partagé. A ce titre, les containers d'un même Pod peuvent s'échanger des ressources.

Un Pod encapsule un certain nombre d'éléments, parmi lesquels :

- les containers d'applications,
- les ressources de stockage,
- un identifiant réseau unique,
- et toutes les configurations liées à l'exécution des containers.

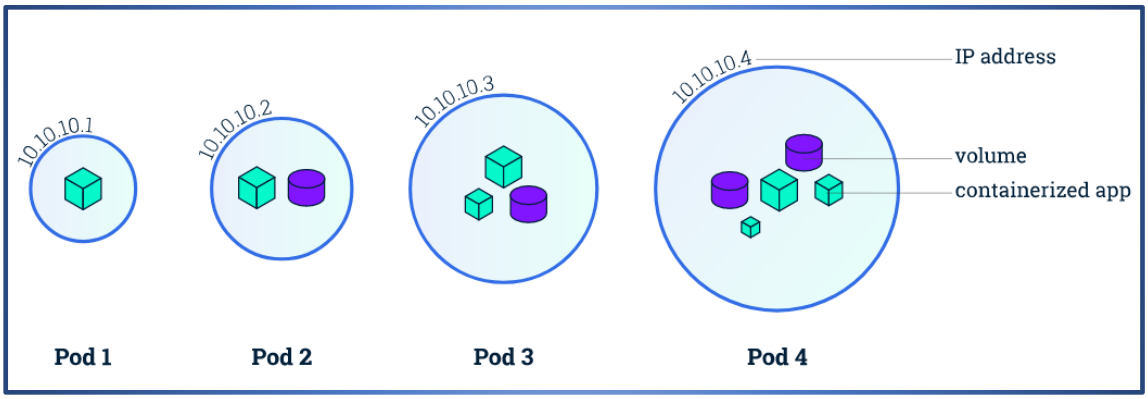


Figure 15 : Ensemble de Pods

Les *Pods* sont volatiles, Kubernetes les gère en fait à sa manière, et ne garantit pas qu'un *Pod* physique donné sera maintenu. Par exemple, le contrôleur de réplication, que nous avons mentionné précédemment, peut tuer, et démarrer un nouveau jeu de *Pods*.
On introduit ici la notion de *service*, qui représente un ensemble logique de *Pods*. Un *service* agît en fait comme une passerelle, permettant aux *Pods* clients de lui envoyer des requêtes, sans se soucier des *Pods* physique le constituant. Cette notion ajoute donc une couche d'abstraction supplémentaire.

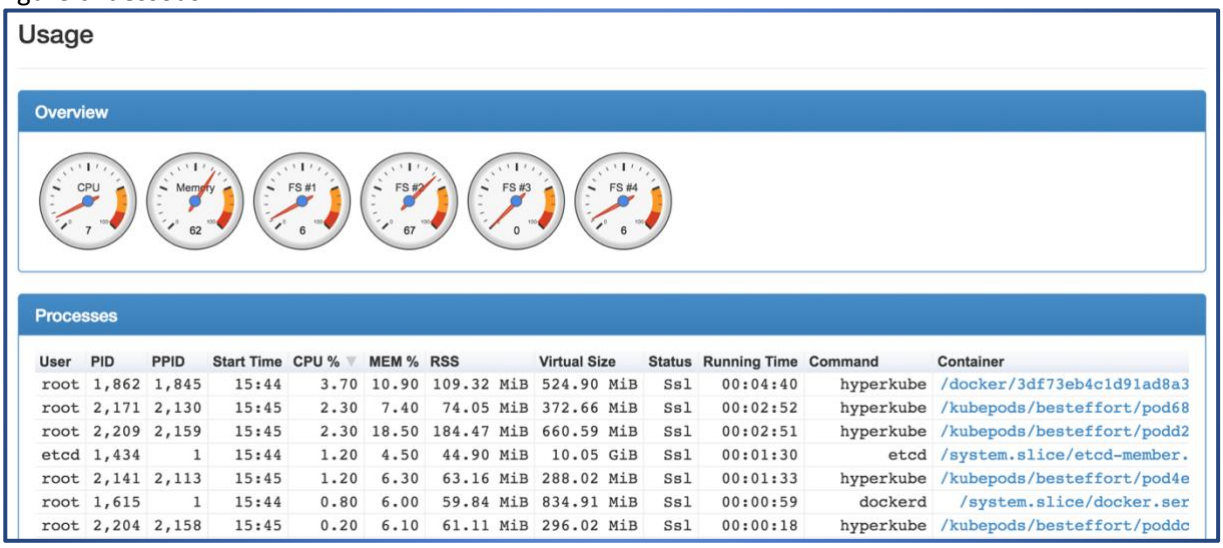
C. Kubelet

Le *Kubelet* est un agent s'exécutant sur chaque *Worker Node* du cluster. Il est responsable de l'état d'exécution de chaque *Worker Node*. Il s'assure que tous les containers présents sur le *Node* sont fonctionnels et en bonne santé.
Il prend en charge le démarrage, l'arrêt, et la maintenance des containers d'applications (organisés en *Pods*), dirigé par le plan de contrôle, un ensemble de *PodSpecs*.

Le *Kubelet* communique avec les composants principaux pour s'authentifier auprès du cluster et recevoir des commandes et travailler. Le travail est reçu sous la forme d'un manifeste qui définit la charge de travail et les paramètres de fonctionnement. Il assume alors la responsabilité de maintenir l'état du travail sur le *Worker Node*. Il contrôle le temps d'exécution du container pour lancer ou détruire des containers selon les besoins.

D. cAdvisor

Le *cAdvisor* est un composant permettant de surveiller et de récupérer les données de consommation de ressources, et des performances (CPU, RAM, stockage, réseau, etc.) du *Node* sur lequel il s'exécute.
Il intègre également une interface graphique, exposant les métriques recueillies, comme le montre la figure ci-dessous.



User	PID	PPID	Start Time	CPU %	MEM %	RSS	Virtual Size	Status	Running Time	Command	Container
root	1,862	1,845	15:44	3.70	10.90	109.32 MiB	524.90 MiB	Ssl	00:04:40	hyperkube	/docker/3df73eb4c1d91ad8a3
root	2,171	2,130	15:45	2.30	7.40	74.05 MiB	372.66 MiB	Ssl	00:02:52	hyperkube	/kubepods/besteffort/pod68
root	2,209	2,159	15:45	2.30	18.50	184.47 MiB	660.59 MiB	Ssl	00:02:51	hyperkube	/kubepods/besteffort/podd2
etcd	1,434	1	15:44	1.20	4.50	44.90 MiB	10.05 GiB	Ssl	00:01:30	etcd	/system.slice/etcd-member.
root	2,141	2,113	15:45	1.20	6.30	63.16 MiB	288.02 MiB	Ssl	00:01:33	hyperkube	/kubepods/besteffort/pod4e
root	1,615	1	15:44	0.80	6.00	59.84 MiB	834.91 MiB	Ssl	00:00:59	dockerd	/system.slice/docker.ser
root	2,204	2,158	15:45	0.20	6.10	61.11 MiB	296.02 MiB	Ssl	00:00:18	hyperkube	/kubepods/besteffort/poddc

Figure 16 : Mesures d'utilisation de cAdvisor

E. Kube-Proxy

Le *Kube-Proxy* permet d'assurer la cohérence au niveau réseau. Il s'agît d'un proxy réseau présent sur chaque *Node*. *Kube-Proxy* est donc chargé de maintenir les règles réseau sur les nœuds. Ces règles permettent une communication réseau vers les *Pods* depuis des sessions à l'intérieur ou à l'extérieur du cluster.
Le Kube-Proxy permet donc les communications *Pod-Pod*, *Node-Node*, container-container, etc.

F. Add-Ons

Enfin, il est également possible d'installer des composants supplémentaires (*add-ons*) sur les *Worker Nodes*. Parmi les fonctionnalités supplémentaires, permises par ces composants, on peut notamment mentionner la journalisation (*logging*), la surveillance des ressources de container, ou encore la possibilité de piloter *Kubernetes* depuis une interface web utilisateur (*dashboard*).

4. Architecture générale

A. Architecture classique

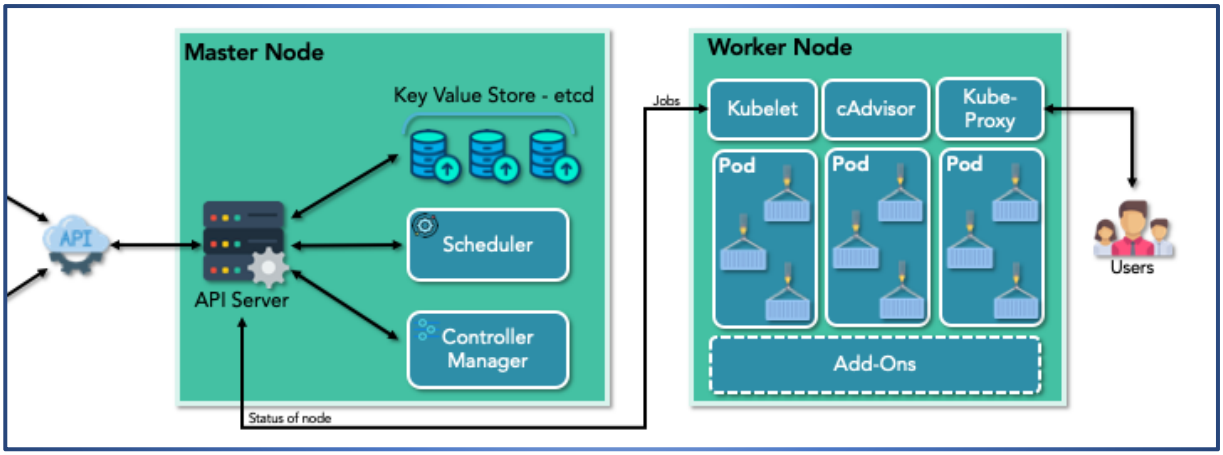


Figure 17 : Architecture générale

La figure ci-dessus représente donc l'architecture classique d'un cluster *Kubernetes*.

De manière générale, un cluster *Kubernetes* s'utilise avec bien plus de *Worker Nodes*, mais il est primordial de comprendre que tous les canaux de communication du cluster au *Master Node* se terminent à l'*API Server*, qui est configuré pour écouter les connexions distantes sur son port d'écoute HTTPS avec des règles d'authentification client.
Par exemple, les informations fournies au *Kubelet* sont regroupées sous forme de certificat client.

Ainsi les *Pods* souhaitant se connecter à l'*API Server* peuvent le faire de manière sécurisée en utilisant un compte de service afin que *Kubernetes* injecte le certificat dans le *Pod* instancié.
Les certificats peuvent être générés à l'aide de deux outils :

- *CFSSL*
- *CFSSLJSON*

On dénombre donc deux principaux canaux de communication :

- De l'*API Server* vers le *Kubelet* afin récupérer les logs de statut des *Pods* qui lui sont rattachés.
- De l'API server vers les *Nodes*, *Pods* et *Services* par des connexions HTTP qui peuvent être protégées par la mise en place de tunnels SSH pour éviter que le trafic ne soit exposé en dehors du réseau dans lequel les *Nodes* s'exécutent.

Enfin l'utilisateur peut configurer le *Kube-Proxy* par l'intermédiaire d'un service rattaché à l'*API Server* afin de gérer plus facilement le DNS et les IP du cluster.

Il peut également se connecter via une interface *kubectl* pour exécuter des commandes sur le cluster de *Kubernetes* pour par exemple créer, lister des ressources ou encore exécuter une commande depuis le container d'un *Pod*.
C'est ainsi que nous procéderons pour la *Proof of Concept*.

B. Architecture dans le Cloud

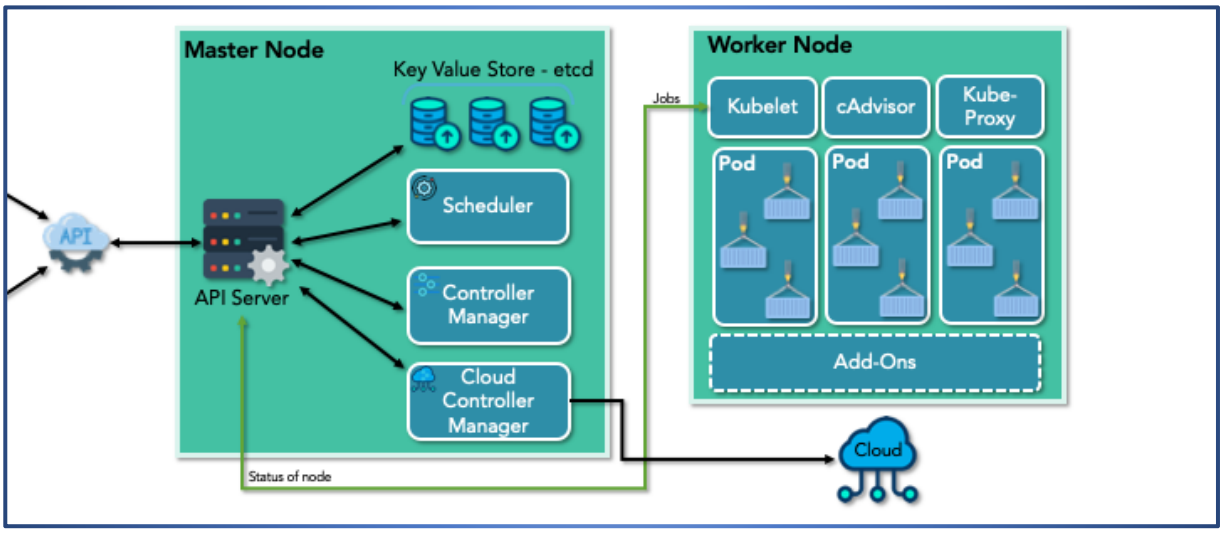


Figure 18 : Architecture dans le Cloud

L'architecture du cluster *Kubernetes* dans le cloud est légèrement différente de la précédente. Cette différence avec l'architecture originelle réside dans le composant *Cloud Controller Manager***.** Cette fonctionnalité a été intégrée dans la version 1.6 de *Kubernetes*, en 2017.

Le *Cloud Controller Manager* permet au code du fournisseur cloud et au code de *Kubernetes* d'évoluer indépendamment l'un de l'autre. Dans des versions antérieures, le code de base de *Kubernetes* dépendait du code spécifique du fournisseur cloud.
Désormais, le code spécifique des fournisseurs cloud doit être maintenu par les fournisseurs cloud eux-mêmes et lié au *Cloud Controller Manager* lors de l'exécution de *Kubernetes*.

Afin d'intégrer le composant *Cloud Controller Manager*, il est nécessaire d'apporter quelques modifications au cluster existant. Ainsi, le principe de communication reste inchangé, mais le marquage des informations relatives aux *Nodes* n'est plus récupéré à l'aide de métadonnées locales mais par des appels à l'*API Server*, qui transmet ensuite l'information au *Cloud Controller Manager*.

Concernant le choix du fournisseur d'accès cloud, on distingue principalement deux cas :

- Si le fournisseur fait partie du référentiel existant, il faudra donc simplement importer le projet existant
- Le cas échéant, il faudra vérifier que le fournisseur puisse s'exécuter avec le paramétrage *cloud-provider* associé au *Cloud Controller Manager* de *Kubernetes*.

A l'heure actuelle, *Digital Ocean*, *Oracle*, *AWS*, *Azure*, *GCP*, *BaiDuCloud* et *Linode* sont les fournisseurs ayant implémenté leur Cloud Controller Manager.

Positionnement sur le marché

1. Concurrence

A. Nomad

Nomad est un manager de cluster et un orchestrateur. Il a été pensé aussi bien pour les services à vie longue que pour les courts traitements par lots. *Nomad*, comme *Kubernetes*, est un projet open source. Nomad suit le précepte *UNIX* de faire une seule chose et de la faire correctement, et ne propose que le management de clusters et l'orchestration. En revanche, il permet de ne pas se limiter aux applications containerisées, car *Nomad* permet également l'utilisation d'applications virtualisées ou *standalone*. D'après certains avis d'utilisateurs, il peut se révéler complexe à prendre en main, à configurer et monitorer.

B. Docker Swarm & Compose

Docker Swarm permet de convertir un groupe d'hôtes *Docker* en un unique hôte virtuel. Comme il s'agit d'un outil natif Docker, il expose l'*API Docker*, ce qui rend possible de l'intégrer et de le faire communiquer avec d'autres outils *Docker* (*Docker CLI*, *Compose*, *Krane*, etc.).
Avec *Compose*, on définit une application multi-container dans un seul fichier, on la crée rapidement avec une seule commande qui effectue tout le nécessaire pour le bon fonctionnement de l'application.
L'API *Docker* limite les fonctionnalités de ces deux outils, et la communauté derrière le projet est plus petite que celle de *Kubernetes*.

C. Apache Mesos

Apache Mesos est un gestionnaire de cluster qui simplifie la complexité d'exécuter des applications sur une pool de serveurs partagée. Les fonctionnalités clés d'*Apache Mesos* sont la mise à l'échelle et l'isolation des tâches avec les containers *Linux*.
De nombreuses applications de traitement de données (*Hadoop*, *Kafka*, *Spark*) s'intègrent facilement sur *Mesos*. Cela est particulièrement utile car ces applications peuvent toutes être lancées sur un même pool de ressource, avec de nouvelles applications dans des containers.
Mesos est un outil ayant fait ses preuves dans des projets à grande échelle (utilisé par *Twitter*, *eBay* et *AirBnB*) et peut supporter des centaines de milliers de nœuds. Cependant, l'outil *Mesos* est peut-être une solution trop complexe pour des petits clusters de moins d'une douzaine de nœuds puisqu'il possède un faible niveau d'abstraction et est prévu pour des grands clusters.

D. OpenShift

OpenShift est une *Platform as a Service* (*PaaS*) de cloud computing proposée par *Red Hat*. Il offre tout un écosystème permettant de créer, tester, déployer et exécuter des applications.
Apprendre à utiliser *OpenShift* est relativement simple : la plupart des environnements peuvent être installés en peu d'étapes et la documentation ainsi que les ressources externes sont très fournies.
Grâce à sa flexibilité et son pouvoir de personnalisation, *OpenShift* peut être utilisé afin de créer des tâches spécialisées et permet l'utilisation de n'importe quel langage, ce qui le rend particulièrement polyvalent.

2. Positionnement de Kubernetes

A. Avantages

Un des avantages clés est la possibilité d'avoir une architecture en microservices, c'est à dire de séparer l'application en de multiples services, chacun avec son propre exécutable. Cela permet de faciliter les changements et la maintenance. *Kubernetes* permet une haute mise à l'échelle, facilite la gestion de containers et aide à réduire les délais de communication grâce à un *load balancer*.

Kubernetes permet d'effectuer des *rolling updates* (mises à jour ''roulantes'') : le déploiement des mises à jour est en continu par opposition au système par version. Il propose aussi le *rollback*, qui permet d'annuler les changements de code si nécessaire et ainsi facilement revenir à un état précédent.

Un des avantages majeurs de Kubernetes est qu'il s'agit d'un environnement sûr grâce à l'immutabilité, signifiant qu'une instance de *Kubernetes* est verrouillée et ne peut pas être changée. Les applications dans *Kubernetes* sont des outils jetables : une fois fini avec une ancienne version, on la remplace par la nouvelle. Si une version échoue, on la détruit simplement et la remplace par une nouvelle.

Un autre avantage de *Kubernetes* est qu'il est basé sur la programmation déclarative et non impérative. Cela signifie qu'au lieu de décrire comment parvenir d'un point A à un point B, on définit B comme but à atteindre et le système décide de la meilleure façon d'y parvenir.

Kubernetes possède une fonctionnalité d'autoréparation permettant de récupérer les instances tombées. Imaginons une application possédant trois instances, avec l'une d'entre elle qui tombe à cause d'un problème tel qu'un manque de mémoire.
La fonctionnalité d'autoréparation va alors créer une nouvelle instance pour restaurer les trois instances de l'application.

B. Rapport à la concurrence

D'après un sondage réalisé par *TheNewStack*, 69% des entreprises interrogées utiliseraient Kubernetes pour gérer leurs containers d'applications.

Il est de plus important de mentionner que certains concurrents mentionnés sur ce graphique sont également basés sur *Kubernetes*, comme *OpenShift*, et *Google Container Engine*.

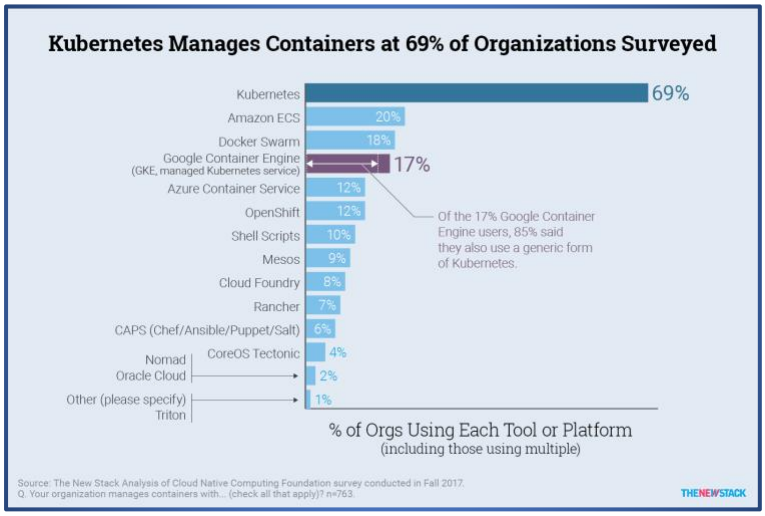


Figure 19 : Outils utilisés par les entreprises pour la gestion des containers

Bien que *Kubernetes* soit en position de force par rapport à ses concurrents, il ne faut pas pourtant les négliger.

Pour certains cas d'utilisations, comme de l'orchestration combinant des applications non-containerisées et des applications containerisées, une solution comme *Mesos* peut s'avérer plus performante.
Pour l'orchestration d'un nombre limité de containers sur un nombre limité de serveurs, *Docker Swarm* peut être une meilleure alternative, plus simple à mettre en place et à prendre en main.

En revanche, pour une solution fiable, robuste, scalable et durable, *Kubernetes* est le choix à faire.

3. Importance et reconnaissance de la technologie

A. Un projet en pleine croissance

Avec à l'heure actuelle plus de **204 000 commits**, plus de **139 000 requêtes pull**, pour un total de **plus d'1.8 Millions de contributions**, effectuées par quelques **39 000 contributeurs**, on peut aujourd'hui affirmer que *Kubernetes* est un **projet en pleine croissance**.
En effet, le projet est aujourd'hui le troisième projet open-source le plus important, derrière *Linux* et *React*, un framework développé par *Facebook*.

Parmi ces contributeurs, on retrouve évidemment de nombreux **développeurs indépendants**, représentant **16.2%** des commits, mais également **plus de 2 000 entreprises**, dont de nombreux leaders dans divers domaines de l'informatique.
Ainsi, on peut mentionner **Google**, à l'origine du projet, qui est le **principal contributeur**, avec **42.3%** de commits, *Red-Hat*, pointure du monde open-source, mais également des entreprises du monde des réseaux, comme *Huawei* et *ZTE*.

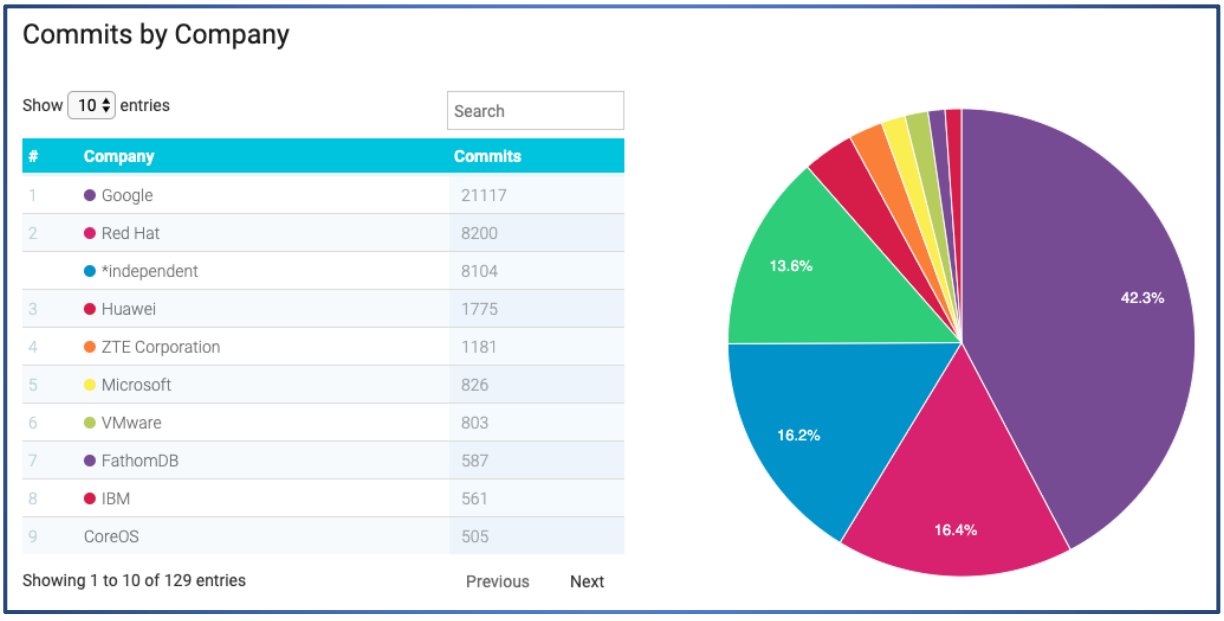


Figure 20 : Principaux contributeurs au projet

B. Communauté

La *CNCF* (*Cloud Native Computing Foundation*), en charge du projet, organise deux conférences liées à *Kubernetes* :

- La *KubeCon*
- La *CloudNativeCon*

En 2019, ces deux conférences ont rassemblé plus de **21 000 personnes**, sur **trois continents**. Elles sont financées par des sponsors, et la quasi-totalité des bénéfices est reversée à la communauté, pour financer divers programmes (formations, certifications, organisation d'autres conférences de taille moindre, etc.)

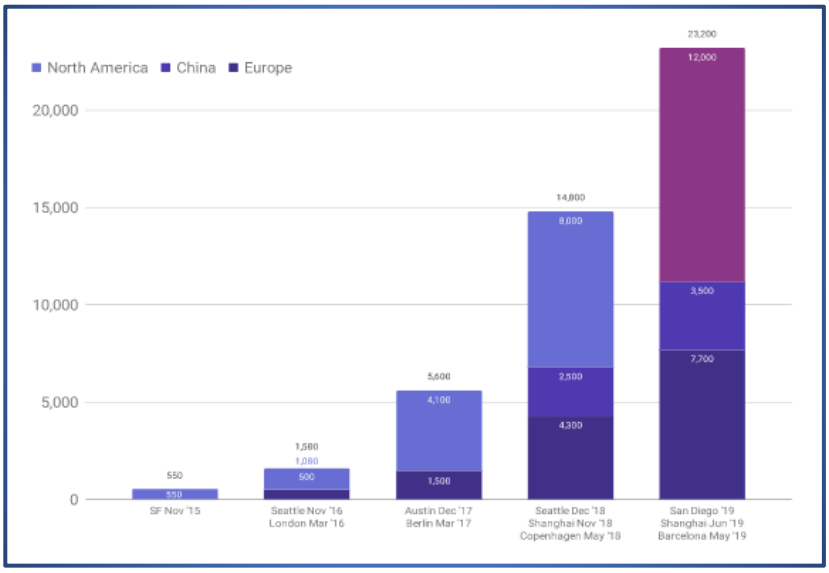


Figure 21 : Affluence mesurée aux conférences KubeCon et CloudNativeCon

C. Certifications

Il existe actuellement deux certifications officielles pour *Kubernetes*, attribuées par la *CNCF* :

- La **CKA** (*Certified Kubernetes Administrator*), destinée aux exploitants, devenue rapidement une des certifications les plus populaires de l'organisme, avec plus de **9 000 inscriptions** et environ **1 700 certifiés**.
- La **CKAD** (*Certified Kubernetes Application Developer*), destinée aux développeurs d'applications utilisant *Kubernetes*, comptant plus de **2 700 inscrits**.

Ces deux certifications coûtent environ $300, et incluent une formation de préparation en ligne. Valides sur une durée de deux ans, avec une possibilité de renouvellement, elles sont particulièrement reconnues en milieu professionnel et apportent un vrai plus en termes de compétences et de salaire.

La *CNCF* propose également une certification destinée aux entreprises : la **KCSP** (*Kubernetes Certified Service Provider*). Les KCSP sont des fournisseurs de services agréés qui possèdent une vaste expérience dans l'accompagnement des entreprises à utiliser *Kubernetes* et au moins trois administrateurs certifiés *Kubernetes* (CKA) en interne. A ce jour, 125 entreprises partenaires possèdent cette certification.
Ces entreprises sont mises en avant sur le site web de *Kubernetes* (https://kubernetes.io/partners/#kcsp), et bénéficient ainsi d'une exposition importante, et donc d'opportunités commerciales.

D. Utilisateurs

Enfin, les utilisateurs de *Kubernetes* font aussi sa force, certains rédigeant régulièrement des cas d'études publiés sur le site de la plateforme, relatant ainsi leur transition vers *Kubernetes*, et les avantages constatés.

Le cas de l'application *Pokemon GO* est certainement l'un des plus frappants. *Niantic*, l'entreprise propriétaire de l'application, a choisi *Kubernetes* pour sa façon de gérer le multi-clustering au niveau mondial et laisser les équipes de développeurs se concentrer sur d'autres sujets comme l'amélioration du jeu.

Sur le graphique ci-contre, on constate que l'équipe avait estimé un trafic cible, et une fourchette haute, équivalent à 5 fois ce trafic. Pourtant, la popularité du jeu a fait exploser les statistiques, à tel point que le trafic constaté était 50 fois supérieur à l'estimation, et 10 fois supérieur à la fourchette haute.
Grâce à *Kubernetes*, et notamment à sa mise à l'échelle automatique, l'application a su répondre à cette demande, et satisfaire les besoins des utilisateurs.

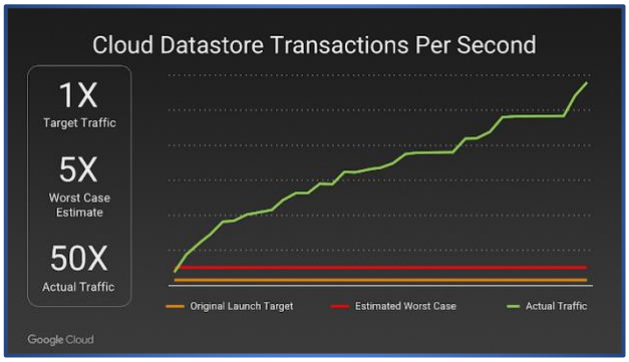


Figure 22 : Trafic sur l'application Pokemon GO

De plus, comme évoqué au sein de la partie *Concurrence* de ce rapport, d'après un sondage mené par *TheNewStack* en 2017, 69% des entreprises interrogées managent leurs containers à l'aide de *Kubernetes*. On peut par exemple citer *Google*, *Huawei*, *Spotify*, *SAP*, *ING*, *Samsung*, *ebay*, *IBM*, etc.

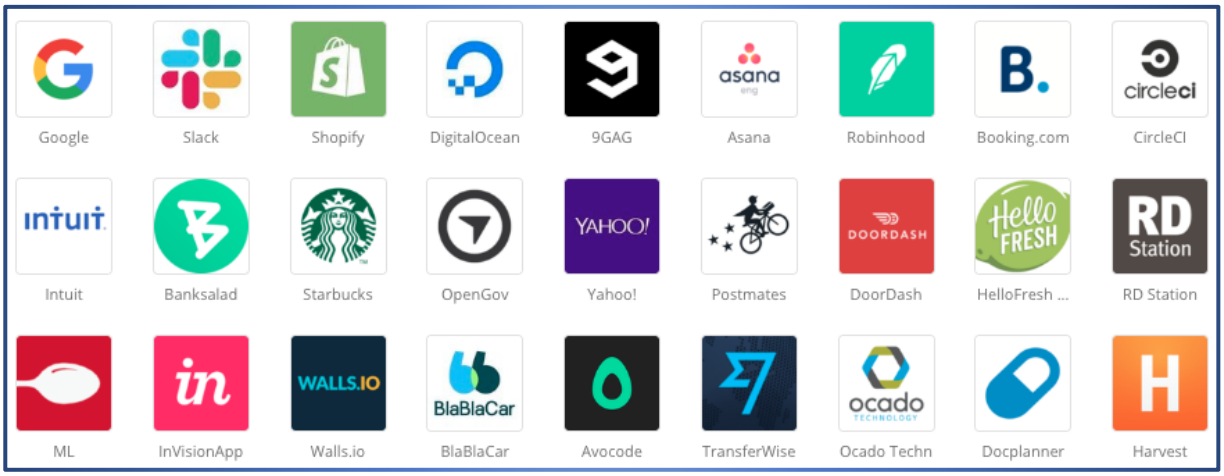


Figure 23 : Quelques utilisateurs de Kubernetes

4. Intégration dans un workflow DevOps

Aujourd'hui, le cycle de livraison des logiciels se veut de plus en plus court, là où la taille des applications, elle, ne cesse d'augmenter.

En réponse à cela naît le mouvement *DevOps*, à la fin des années 2000. Son principal objectif est d'unifier le développement et l'exploitation des applications, sur la totalité du cycle de vie du logiciel (de la planification à l'exploitation et la surveillance, en passant par le développement, les tests, et le déploiement).

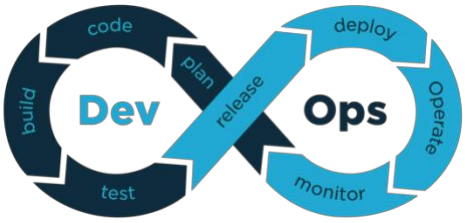


Figure 24 : Cycle de vie du logiciel

Les containers facilitent l'hébergement et la gestion du cycle de vie des applications sur un environnement. Ils regroupent les dépendances du code applicatif dans des blocs de construction afin d'assurer cohérence, efficacité et productivité. L'outil *Docker* permet de déployer des containers à l'intérieur d'un cluster et de les traiter comme une seule entité.
Toutefois, le véritable défi concerne le déploiement d'applications multi-containers (i.e. qui sont réparties sur plusieurs containers). D'autres problématiques rentrent en jeu : la mise à l'échelle, la gestion des clusters, de manière à gérer le stockage et éviter la congestion du réseau...

Grâce à *Kubernetes*, les développeurs peuvent partager leurs logiciels et leurs dépendances avec les exploitants. L'orchestration de containers rapproche donc les développeurs et les exploitants, permettant ainsi à l'équipe de collaborer efficacement.

Les tâches manuelles, liées aux déploiements et à la mise à l'échelle des applications containerisées, souvent sources d'erreurs, et très chronophages, sont supprimées afin que le logiciel soit déployé de manière plus fiable d'un environnement à un autre.
Ainsi, il est possible de planifier et de déployer un nombre quelconque de containers sur un nœud, sur des clouds privés/publics/hybride), et *Kubernetes* gère ces charges de travail. Il simplifie également de manière drastique les tâches liées au containers (mise à l'échelle automatique, mises à jour progressives, ...).

Pour démontrer l'utilité de *Kubernetes* dans un pipeline CI/CD, on peut se baser sur l'exemple suivant : le déploiement d'une mise à jour d'une application.

Considérons un pipeline CI/CD relativement basique, composé des éléments suivants :

- Un dépôt *GitLab*, permettant le versionnage du code source
- Un serveur d'intégration continue basé sur *Jenkins*
- Un dépôt *Docker*, permettant le stockage des images
- Un cluster *Kubernetes*

Dans le cadre d'une mise à jour, une fois les développements terminés, le nouveau code source est poussé sur le dépôt *GitLab*. Le serveur d'intégration continue va détecter ces changements, et lancer le pipeline d'intégration continue.

Le pipeline va dans un premier temps construire l'image Docker à partir du code source à jour, puis tester cette image, par exemple à l'aide d'outils tels que *JUnit 5* ou *Cucumber*.
Si les tests n'ont pas rencontré d'erreurs, l'image *Docker* est poussée sur un dépôt *Docker* prévu à cet effet, puis *Kubernetes* est à son tour notifié.

Kubernetes va dans un premier temps créer un nouveau *Pod* à partir de l'image *Docker* récupérée sur le dépôt distant, puis vérifier l'état du *Pod* démarré. S'il s'est correctement lancé, alors l'ancien *Pod*

est stoppé, la migration s'est correctement terminée. Le cas échéant, *Kubernetes* ne va pas stopper l'ancien *Pod*, puisque le nouveau Pod a rencontré une anomalie. *Kubernetes* va donc tenter de le redémarrer, puis de vérifier à nouveau son état. S'il reste inchangé, alors *Kubernetes* notifie les développeurs de l'échec du déploiement de la mise à jour.

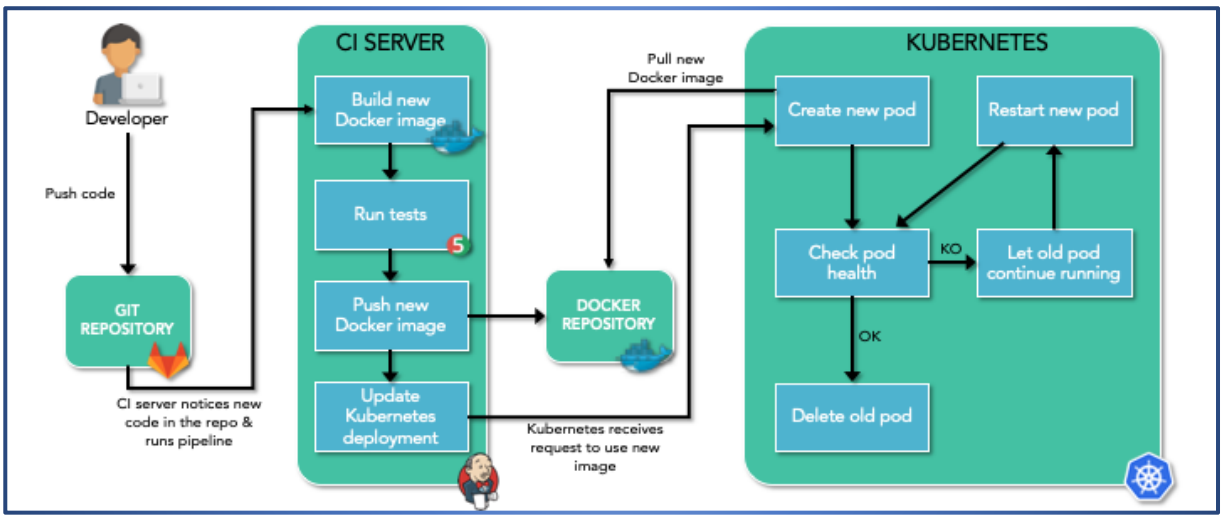


Figure 25 : Exemple de pipeline CI/CD basé sur Kubernetes - Déploiement d'une mise à jour d'une application

Perspectives

1. Limites des containers

Kubernetes souffre indirectement d'une limite, l'architecture multi-tenant des containers qu'il manipule.
En effet, si l'on compare les VMs et les containers, on remarque que bien que ceux-ci possèdent de nombreux avantages, notamment en termes de performances et de consommation de ressources, ils souffrent d'une isolation au niveau processus, les rendant moins sécurisés que les VMs.

VMs	Containers
Poids lourd	Poids léger
Performance limitée	Performance native
Propre OS	Partage de l'OS
Virtualisation niveau matériel	Virtualisation au niveau de l'OS
Temps de démarrage en secondes	Temps de démarrage en millisecondes
Alloue la mémoire requise	Nécessite moins d'espace mémoire
Entièrement isolé = Plus sécurisé	**Isolation au niveau processus = Moins sécurisé**

Figure 26 : Comparatif entre les containers et les VMs

2. Émergence des VM containers

Conscients de cette limite, plusieurs entreprises ont développé une nouvelle technologie, les VM containers, qui sont des machines virtuelles légères, fonctionnant comme des containers, mais possédant une meilleure utilisation des ressources.

Une des entreprises pionnières en la matière est *OpenStack*, avec *Kata Containers*, mais les géants du web s'y sont rapidement intéressés, à la manière de *Google* avec *gVisor*, et *Amazon* avec *Firecracker*. *Kata Containers* a reçu le soutien de plus de 40 partenaires, dont les fabricants de puces *Intel*, *ARM* et *AMD*.
Comme le montrent les figures ci-après, la principale différence entre les containers classiques, et les *Kata Containers* réside dans le partage du noyau (*Kernel*).

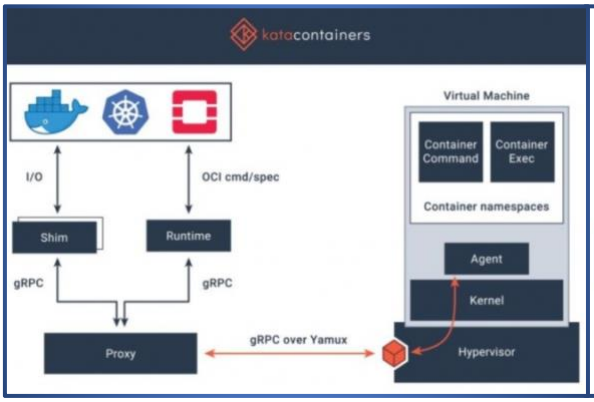


Figure 28 : Architecture des Kata Containers

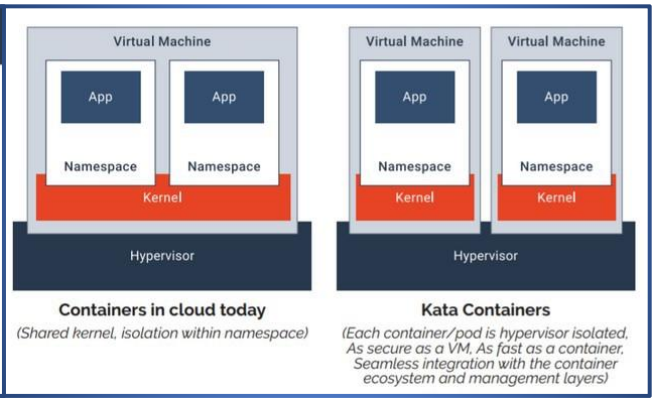


Figure 27 : Différences entre containers et Kata Containers

Sur des containers classiques, le noyau est partagé, l'isolation a lieu au sein d'un *namespace.* En revanche, sur les Kata Containers, chaque container est isolé au niveau de l'hyperviseur, à la manière d'une VM, c'est pourquoi ils sont donc aussi sécurisés.

L'enjeu pour *Kubernetes* est de s'adapter à cette nouvelle technologie, en fournissant un isolement plus strict pour les VM containers, sans pour autant délaisser les containers, qui sont, et seront encore durant les prochaines années, massivement utilisés.

Proof of Concept

Cette partie vise à détailler et expliciter la Proof of Concept de la présentation de *Kubernetes*, accessible via le fichier *Kubernetes_PoC.mp4*.

0. Prérequis

Au sein de cette Proof of Concept, nous utiliserons *Minikube*, un outil facilitant l'exécution locale de *Kubernetes*.
Pour installer *Minikube* sur votre machine, veuillez suivre les instructions au lien suivant :
https://kubernetes.io/docs/tasks/tools/install-minikube/

1. Création d'un cluster Kubernetes

La commande ci-dessus permet de créer un cluster, en spécifiant son nom. Une VM va ainsi être configurée, avec *Docker* et *Kubernetes*.

```
minikube start -p demo-cluster
```

Pour le stopper, on peut utiliser la commande :

```
minikube stop -p demo-cluster
```

Il est également possible de consulter le statut du cluster :

```
minikube status -p demo-cluster
```

Le retour de la commande va mentionner le statut des différents composants, notamment l'hôte, le *kubelet*, et l'*apiserver*.

Enfin, une autre fonctionnalité intéressante liées au cluster *Kubernetes* est le *dashboard* (interface graphique) :

```
minikube dashboard -p demo-cluster
```

Ceci permet d'accéder et de configure de nombreuses fonctionnalités de *Kubernetes* de manière graphique. Ainsi, on peut par exemple afficher les différents *Services*, *Nodes*, *Pods*, visualiser les *Jobs*, les *déploiements*, etc. L'interface se révèle assez complète et rapide à prendre en main.

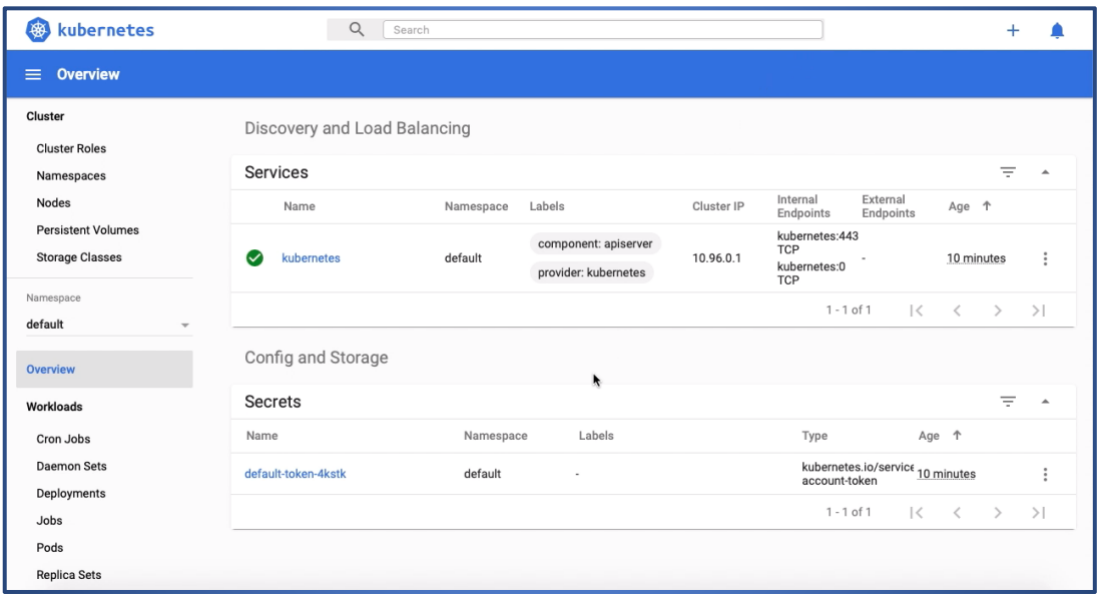


Figure 29 : Dashboard du cluster Kubernetes

2. Création d'un déploiement

Au sein de cette partie, nous allons créer un déploiement à partir d'une image *Docker* relativement simple sous *Kubernetes*, puisqu'il s'agît d'un projet de type *hello world*.

L'image *Docker* est mise à disposition par *Kubernetes*, et est accessible au lien suivant : gcr.io/hello-minikube-zero-install/hello-node

Cette fois, nous n'utiliserons plus *minikube* mais *kubectl*, l'outil en ligne de commandes mentionné au sein de la partie *Architecture* de ce rapport.

```
kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node
```

Il est possible de visualiser les différentes entités de *Kubernetes* (*déploiements*, *Nodes*, *Pods*, etc.) et leurs états grâce aux commandes suivantes :

```
kubectl get deployments
kubectl get nodes
kubectl get pods
```

En affichant de nouveau le *dashboard*, on obtient le résultat suivant :

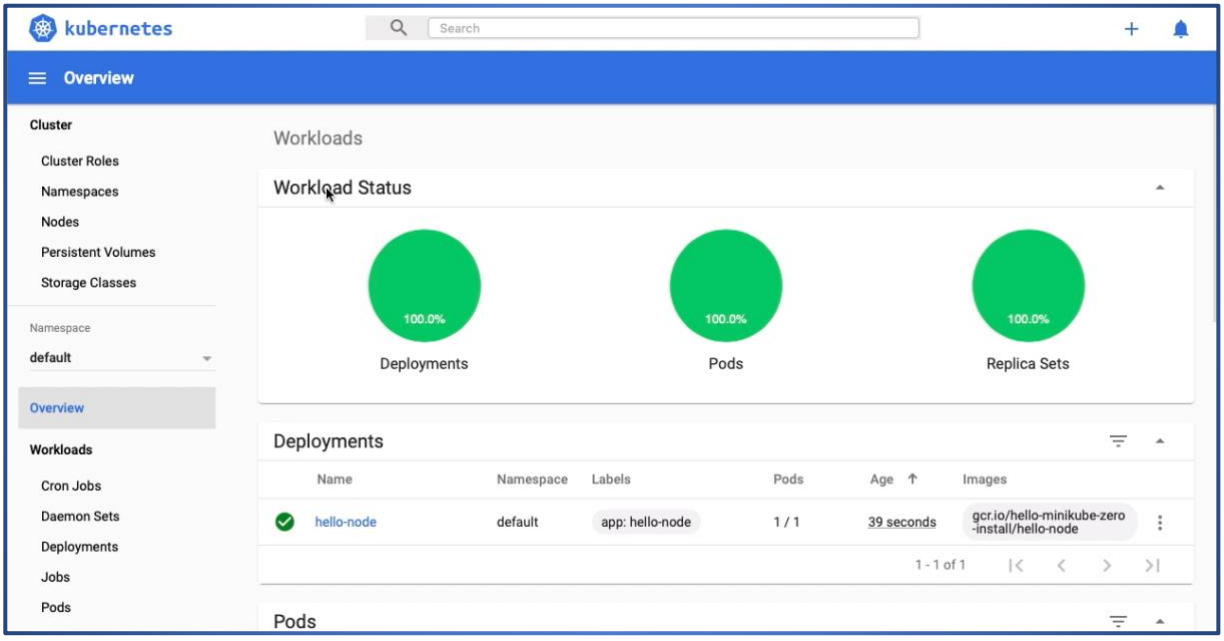


Figure 30 : Dashboard du cluster Kubernetes après déploiement

On visualise donc assez rapidement l'état de nos déploiements, de nos *Pods* et de nos *Replicas*.

3. Exposition d'un service

On va désormais exposer l'application précédemment déployée. Pour ce faire, on utilise la commande :

```
kubectl expose deployment hello-node --type=LoadBalancer --port=8080
```

Il est nécessaire de spécifier deux options :

- Le type du service, il en existe 5, dans notre cas on choisit *LoadBalancer* (équilibreur de charge)
- Le port d'exposition du service (ici 8080).

On peut ensuite accéder au service à l'aide de *minikube* :

```
minikube service hello-node -p demo-cluster
```

L'application s'exécute, et on observe bien le résultat attendu (Hello World !).

4. Création d'un déploiement à partir d'un fichier de configuration

Nous allons désormais effectuer un autre déploiement, dont la configuration sera lue depuis un fichier de configuration.
L'application déployée sera *nginx,* qui est un logiciel libre de serveur web.

```
kubectl apply -f https://kubernetes.io/examples/controllers/nginx-deployment.yaml
```

Comme précédemment, on peut visualiser les *Pods*, et les réplicas à l'aide des commandes :

```
kubectl get pods
kubectl get rs
```

Enfin, il est possible de mettre à jour le déploiement, en modifiant l'image. Ici, on va simplement changer de version de *nginx*, à l'aide de la commande suivante :

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1 --record
```

5. Mise à l'échelle

A. Mise à l'échelle manuelle

On peut choisir de mettre à l'échelle un déploiement manuellement, à l'aide de la commande suivante :

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=12
```

B. Mise à l'échelle automatique

Néanmoins, il est plus commun de mettre à l'échelle de manière automatique, en fixant une condition sur l'utilisation des ressources, tel qu'un pourcentage d'utilisation du processeur (CPU) à ne pas dépasser :

```
kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --cpu-percent=80
```

Conclusion

Comme nous l'avons montré à plusieurs reprises dans ce rapport, *Kubernetes* est actuellement la solution d'orchestration de containers leader du marché.

De fait, en milieu professionnel, ses compétences sont particulièrement recherchées. D'après une étude menée au Royaume-Uni, *Kubernetes* serait aujourd'hui mentionné dans 5% des offres IT (soit 1 sur 20), et dans 35% des offres de gestion des SI (Systèmes d'Informations).
De plus, le salaire moyen de ces offres serait de £70,000 annuel, soit environ 82 100€.

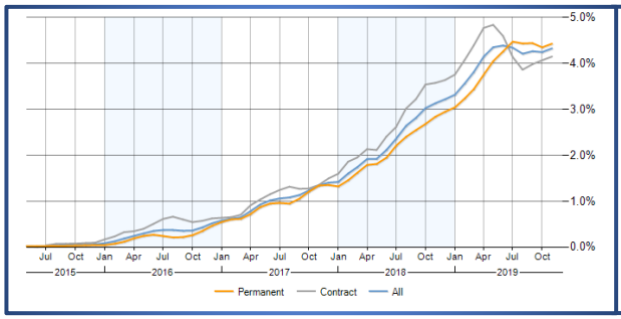


Figure 32 : Pourcentages des offres IT citant Kubernetes

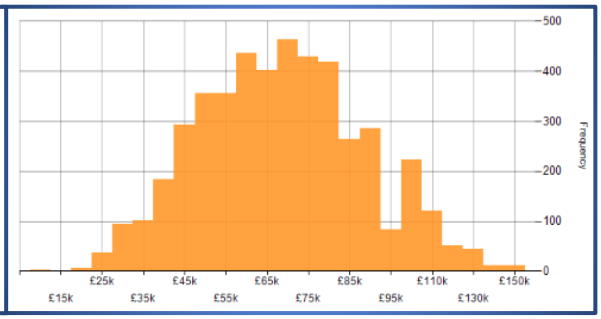


Figure 31 : Répartition des salaires des offres IT citant Kubernetes

Ainsi, nous sommes très heureux d'avoir pu travailler sur ce sujet, innovant et en pleine croissance.

Ce projet de *Cloud Computing* nous a permis de développer un socle de connaissances et de compétences relativement important sur *Kubernetes*, et nous espérons pouvoir les mettre en application et les développer en milieu professionnel.

En termes de perspectives personnelles, nous songeons à suivre une formation professionnelle sur *Kubernetes*, en entreprise, ou en ligne, afin de développer nos acquis.
A la suite de celle-ci, nous pourrions être intéressés par les certifications de Kubernetes, et plus particulièrement la *CKAD* (*Certified Kubernetes Application Developer*), qui semble la plus adaptée à nos profils.

Enfin, nous serions également intéressés par les conférences *KubeCon*, *CloudNativeCon*, et *DevOpsDays* si elles venaient à se produire en France, puisqu'elles traitent de sujets particulièrement innovants et liés aux problématiques DevOps qui nous tiennent à cœur.

Lexique

CI/CD : Ensemble de pratiques menant à une intégration et un déploiement continu.

CONTAINER : Unité logicielle standard qui regroupe le code, les configurations, et toutes les dépendances d'une application afin qu'elle s'exécute rapidement et de manière fiable d'un environnement à un autre.

CLUSTER : Ensemble de serveurs indépendants fonctionnant comme un seul et même système.

DEVOPS : Approche visant à unifier le développement informatique et l'administration des infrastructures.

MISE A L'ECHELLE (SCALABILITY) : Capacité d'un système à gérer une quantité croissante de travail, en ajoutant des ressources.

NAMESPACE : Ensemble virtuel de clusters sauvegardés par le même cluster physique.

NODE : Un *Node* est une machine de travail dans *Kubernetes*. Il peut s'agir d'une machine virtuelle ou physique, selon le cluster.

POD : Unité de base de l'ordonnancement de *Kubernetes*. Un *Pod* contient un ou plusieurs containers, se partageant contexte d'exécution et ressources.

SERVICE : Ensemble logique de *Pods*. Les services offrent une couchent d'abstraction supplémentaire, en fournissant une unique adresse IP et un nom DNS permettant d'accéder aux *Pods*.

VM (VIRTUAL MACHINE) : Programme logiciel ou un système d'exploitation présentant le comportement d'un ordinateur physique, capable de tâches telles que l'exécution d'applications et de programmes.

Sources- Webographie

- **Documentation officielle**
  https://kubernetes.io/docs/home/
- **Page Github du projet**
  https://github.com/kubernetes/kubernetes
- **Site de la CNCF**
  https://www.cncf.io/
- **Page Open Hub du projet – Statistiques**
  https://www.openhub.net/p/kubernetes
- **RedHat – Qu'est-ce que Kubernetes**
  https://www.redhat.com/fr/topics/containers/what-is-kubernetes
- **Cas d'utilisations**
  https://platform9.com/blog/kubernetes-use-cases/
- **Kubernetes & OpenStack**
  https://vexxhost.com/blog/openstack-kubernetes/
- **Reconnaissance de la technologie**
  https://www.weave.works/technologies/the-journey-to-kubernetes/
- **Certifications**
  https://www.cncf.io/certification/cka/
  https://www.cncf.io/certification/ckad/
- **Cas d'etude – Pokemon GO**
  https://cloud.google.com/blog/products/gcp/bringing-pokemon-go-to-life-on-google-cloud
- **Futur de Kubernetes**
  https://tech.paulcz.net/blog/future-of-kubernetes-is-virtual-machines/
- **Perspectives d'emploi**
  https://www.itjobswatch.co.uk/jobs/uk/kubernetes.do