26/08/2022

DEVELOPPEMENT D’UNE PALEFORME KUBERNETES

Sous l’encadrement de Monsieur AFIFI HOSSAM

EQUIPE PROJET :

* GEORGES PAULIN ABIA
* AHOUA BECHEHIN
* OUMAR AMANE
* STÉPHANE COULIBALY
* HAKIM MOULAY

Table des matières

[**REMERCIEMENTS** 2](#_Toc112469986)

[**Liste des Figures** 3](#_Toc112469987)

[**Introduction** 5](#_Toc112469995)

[**Chapitre 1 : KUBERNETES** 6](#_Toc112469996)

[I- **Présentation Docker** 6](#_Toc112469997)

[**i.** **Introduction** 6](#_Toc112469998)

[**ii.** **Docker** 7](#_Toc112469999)

[**a.** **Architecture Docker** 8](#_Toc112470000)

[**b.** **Composantes internes de Docker** 9](#_Toc112470001)

[**c.** **Docker (Containers) Vs Hyperviseurs (Machines Virtuelles)** 10](#_Toc112470002)

[**iii.** **Kubernetes** 10](#_Toc112470003)

[**a.** **Architecture Kubernetes** 11](#_Toc112470004)

[**b.** **Les composants de Kubernetes** 11](#_Toc112470005)

[ **Kubelet** 13](#_Toc112470006)

[ **kube-proxy** 13](#_Toc112470007)

[ **Container Runtime** 13](#_Toc112470008)

[ **kube-apiserver** 14](#_Toc112470009)

[ **Etcd** 14](#_Toc112470010)

[ **Kube-scheduler** 14](#_Toc112470011)

[ **Kube-Controller-manager** 14](#_Toc112470012)

[ **Le Service *Ingress*** 18](#_Toc112470022)

[ **Le Service *NodePort*** 19](#_Toc112470026)

[**iv.** **Conclusion :** 24](#_Toc112470034)

[**Chapitre 2 : Déploiement Canary sur kubernetes** 25](#_Toc112470035)

[**1.** **Présentation :** 25](#_Toc112470036)

[**2.** **Déploiement de Canary du kubernetes** 26](#_Toc112470037)

[**3.** **Mise en place de Canary kubernetes** 29](#_Toc112470038)

[**4.** **Conclusion :** 34](#_Toc112470039)

[**Conclusion Générale** 35](#_Toc112470040)

[**Références bibliographiques :** 36](#_Toc112470041)

# **REMERCIEMENTS**

Nos remerciements vont particulièrement non seulement à l’endroit de **Monsieur HOSSAM AFIFI**, Enseignant du pôle Réseaux et Services Télécom qui nous a encadré durant toute la période du projet malgré ses importantes occupations et engagements académiques, mais aussi au **Doctorant Monsieur ANTOINE BERNARD** qui n’a ménagé aucun effort pour partager des ressources, nous conseiller et débloquer même à des heures très tardives des disfonctionnements dans nos configurations.

Nous remercions les responsables de la formation de Mastère en Réseaux et Services de Telecom SudParis pour le choix des thèmes et l’organisation des enseignements qui permettent la réalisation d’un projet aussi important dans le cadre de notre cursus. Nous pensons particulièrement au **Dr Ing Joséphine KOHLENBERG,** et au **Maître de conférences/Associate Professor,** **Abdelwaheb MARZOUKI**, respectivement Coordinatrice et Responsable du Mastère.

Nous ne pourrions clôturer cette partie sans remercier nos familles pour les appuis multiformes notamment le soutien moral.

# **Liste des Figures**

Figure 1.1 : Conteneur Docker

Figure 1.2 : Comparaison conteneurs Docker vs machines virtuelles VM

Figure 1.3 : Communication entre le client et le démon de Docker

Figure 1.4 : Composantes internes de Docker

Figure 3.1 : Principe de fonctionnement Canary

Figure 1: Schéma de fonctionnement de Kubernetes

Figure 2 : Conteneur Docker

Figure 3 : Communication entre le client et le démon de Docker

Figure 4 : Composantes internes de Docker

Figure 5 : Comparaison conteneurs Docker vs machines virtuelles VM

Figure 6 : Architecture de Kubernetes.

Figure 7 : Pod

Figure 8 : Worker node

Figure 11 : Illustration de service Kubernetes

Figure 12 : Composition et fonctionnement d’’un service Kubernetes

Figure 13 : Composition et fonctionnement d’’un service Ingress Kubernetes

Figure14 : Illustration d’’un service Ingress Kubernetes

Figure15 : Illustration d’’un service Node Port Kubernetes

Figure 16 : Composition et fonctionnement d’’un service Node Port Kubernetes

Figure 17 : Composition et fonctionnement d’’un service LoadBalancer Kubernete

Figure 18 : Illustration d’’un service LoadBalancer Kubernetes

Figure 3.1 : Principe de fonctionnement Canary

Figure 3.2 : Activation paramètre VT -x/ AMD-V

Figure 2 : VirtualBox

Figure 3 : Docker

Figure 4 : Kubectl version

Figure 5 : Minikube version

Figure 6 : Etat cluster Minikube

Figure 7 : Nœud cluster

Figure 8 : Info cluster

Figure 9 : Image Nginx

Figure 10 : Déploiement Nginx

Figure 11 : Exemple code html

Figure 11 : Déploiement pods

Figure 12 : Service

Figure 13 : Adresse ip service

Figure 14 : Application version 1

Figure 15 : Fichier YAML

Figure 16 : Code HTML version 2

Figure 17 : Service modifié

Figure 18 : Application version 2

Figure 19 : Kubernetes dashboard

# **Introduction**

Le challenge permanent des directions des systèmes d’information et des ingénieurs IT est d’offrir des services efficaces et efficients aux utilisateurs. En entreprise, cet objectif n’était pas la priorité des premiers déploiements. L’informatique traditionnelle se basait généralement sur le modèle « une application – un serveur ». L’un des plus grands inconvénients de cette approche est la non-rationalisation des ressources (Mémoire, Processeur, Disque, Réseau). Il n’est pas effectivement rare de trouver des serveurs utilisés à seulement 30% de leurs potentiels.

Afin d’apporter un début de solution à cette problématique, des opérations de consolidation sont préconisées. Elles consistent à passer du modèle susmentionné vers le modèle « un serveur plusieurs applications ». Cette tendance s’est beaucoup développée avec l’émergence et la vulgarisation des architectures N tiers.

De nouvelles technologies voient le jour pour accompagner l’implémentation de cette nouvelle approche. Parmi les plus connues, on peut évoquer la virtualisation. Elle vise à simuler/déployer, dans certains cas, sur un même ordinateur physique, plusieurs ordinateurs virtuelles (machines virtuelles) ou dans d’autres cas, plusieurs applications, chacune dans un environnement étanche (conteneur) sur un même ordinateur physique.

La gestion des machines virtuelles et/ou des conteneurs peut devenir complexe lorsque leur nombre augmente notamment dans de grands déploiements. Il est donc nécessaire de mettre en place un orchestrateur (Kubernetes) de conteneurs ou un outil de gestion d’inventaire des machines virtuelles pour gérer l’environnement informatique.

Dans le cadre de notre projet, nous allons travailler sur la conteneurisation des serveurs web NGINX via les conteneurs dockers et l’orchestration à partir de Kubernetes. L’objectif principal étant d’assurer la haute disponibilité en cas d’incident sur l’un des serveurs.

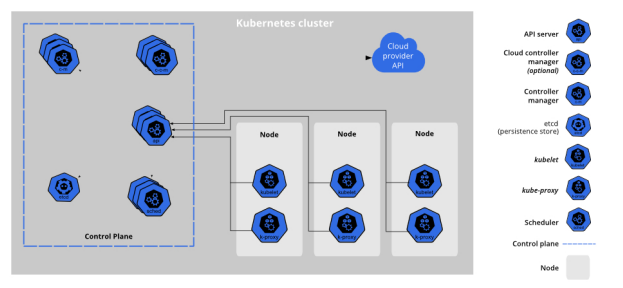
# **Chapitre 1 : KUBERNETES**

## **Présentation Docker**

### **Introduction**

Kubernetes est une plate-forme d’orchestration de conteneurs. Elle est libre et open-source et permet d’automatiser l’exploitation de conteneurs. Elle permet de simplifier les processus manuels de publication en exécutant les applications « packagées » dans un format spécifique de conteneurs. Ces derniers sont déployés et exécutés dans un cluster Kubernetes. Kubernetes fonctionne avec plusieurs environnements d’exécution de conteneurs à savoir: Docker, containerd, CRI-O et toutes les autres implémentations de Kubernetes Container Runtime Interface (CRI).

Kubernetes se charge du déploiement, de la mise à disposition de l’application ainsi que d’un tableau de contrôle, de la répartition de charge (load balancing), de la surveillance des conteneurs exécutés et du redémarrage de ces derniers en cas de problème. Minikube est un outil permettant d’exécuter un cluster Kubernetes avec un seul nœud en local. Il fonctionne sur Windows, Mac OS et Linux à travers plusieurs modes de fonctionnements à l’aide d’une VM, d’un conteneur ou en hyperviseur.

****

*Figure1: Schéma de fonctionnement de Kubernetes*

### **Docker**

Docker est une plateforme open source qui permet à des applications d’être déployées à partir de conteneurs. Le concept de Docker est d’avoir une couche d’abstraction permettant aux développeurs d’applications d’empaqueter n’importe quelle application et c’est ensuite la technologie de conteneurisation qui gère le déploiement, indépendamment de l’infrastructure physique.

La plateforme Docker facilite la portabilité et l’intégration d’applications dans différents environnements. Avec Docker, un développeur peut configurer une application sur son ordinateur et la transmettre à une autre personne qui pourra la déployer et l’exécuter sur un serveur Docker sans différence de fonctionnement avec la version du développeur.

À l’origine, le concept de conteneurisation utilisé par Docker pour ses solutions logicielles vient des conteneurs physiques utilisés depuis des dizaines d’années dans le domaine du transport de marchandises.

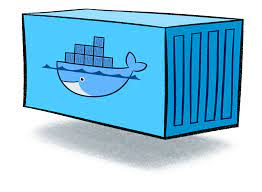
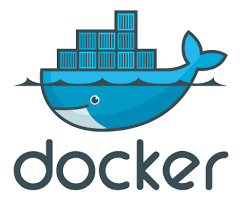


Figure 2 : Conteneur Docker

Un conteneur est une unité logique, dans laquelle on empaquète une application avec toutes les dépendances dont elle a besoin. Elle est donc capable de s’exécuter de manière isolée par le noyau du système hôte.

### **Architecture Docker**

L’architecture de Docker est basée sur une architecture de type client-serveur (voir Figure : Architecture de Docker). Il y a un client Docker et un Docker daemon qui prend le rôle de serveur.

Le client Docker ne peut pas communiquer directement avec les conteneurs. Pour cela, il communique avec un élément de l’architecture appelé Docker daemon. Il peut être installé sur la même machine que le Docker daemon ou sur une machine différente.

Le Docker daemon est un élément du présent sur le serveur Docker considéré comme un gestionnaire de conteneurs qui fait le lien entre le client, les conteneurs et les images qui composent les conteneurs.

C’est un composant essentiel qui a pour responsabilité d’effectuer toutes les opérations liées à un conteneur comme la création, l’exécution ou l’arrêt de celui-ci.

Le Docker daemon peut être exécuté sur plusieurs serveurs dans une infrastructure et un client peut interagir avec plusieurs serveurs également. La gestion des communications appartient totalement aux clients, mais les serveurs Docker peuvent interagir directement avec les Docker images si un client le demande.

Les clients sont responsables et doivent gérer les tâches exécutées par les serveurs. Le rôle des serveurs est d’héberger les applications sous forme de conteneurs.

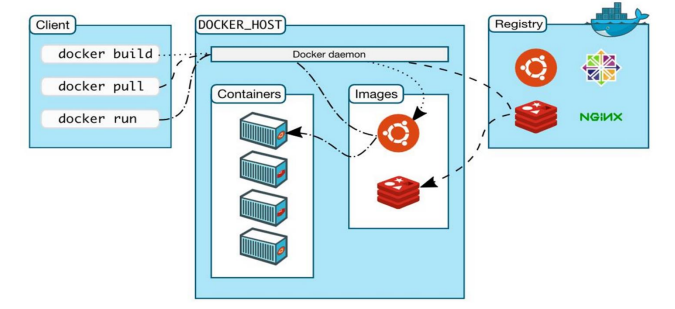


Figure 3 : Communication entre le client et le démon de Docker

La communication entre le client et le serveur se fait généralement par l’outil de ligne de commande de Docker.

C’est un outil qui est fonctionnel avec la majorité des systèmes d’exploitation.

Voici les principales actions qu’il est possible de réaliser avec l’outil de ligne de commande de Docker

* Récupérer une image depuis un registre et la transférer vers un Docker daemon ou transférer une image vers un registre à partir d’un Docker daemon.
* Créer une image Docker
* Démarrer un conteneur sur un serveur Docker

L’outil de ligne de commande de Docker n’est pas le seul moyen de communication entre un client et un serveur Docker. L’utilisation d’APIs permet aussi de communiquer avec le serveur et de gérer et contrôler l’exécution des applications.

### **Composantes internes de Docker**

À l’interne, Docker est composé des trois éléments suivants :

* Les images : Elles sont les composantes de construction de Docker et représentent des modèles accessibles en lecture seulement,
* Les registres : Ils sont les composantes de distribution de Docker qui hébergent les images,
* Les conteneurs : Ils sont les composantes d’exécution de Docker et sont similaires à des répertoires. La figure ci-dessous présente les composantes internes de Docker et les interactions entre elles.

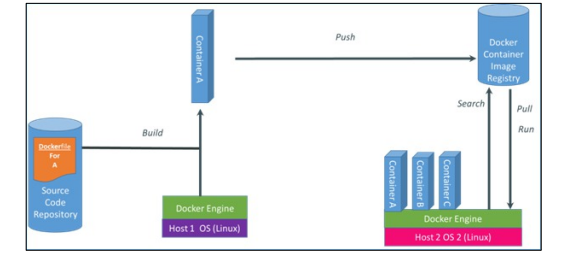
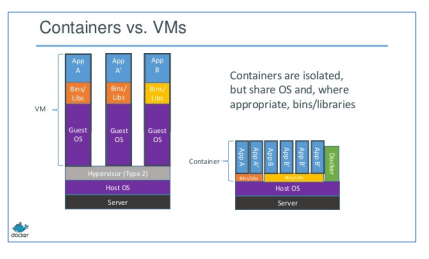


Figure 4 : Composantes internes de Docker

### **Docker (Containers) Vs Hyperviseurs (Machines Virtuelles)**

La virtualisation par hyperviseurs (c.-à-d. par machine virtuelle) diffère de la visualisation par conteneurs de Docker par différents points. Premièrement, au niveau de l’architecture : La couche Hyperviseur est éliminée dans la virtualisation par conteneur. Les applications utilisent le système d’exploitation de l’hôte et ses pilotes, ce qui rend les conteneurs Docker plus rapide que les machines virtuelles.



*Figure5 : Comparaison conteneurs Docker vs machines virtuelles VM*

Deuxièmement, au niveau de performance et de la vitesse, la virtualisation par hyperviseur permet que les machines virtuelles, étant une abstraction des machines physiques, nécessitent un système d’exploitation, des démons («daemons» en anglais), de la gestion de mémoire et des pilotes de matériels, ce qui les rend plus lourds et occupent plus d’espace mémoire par rapport aux conteneurs qui utilisent le système d’exploitation (SE) de l’hôte.

### **Kubernetes**

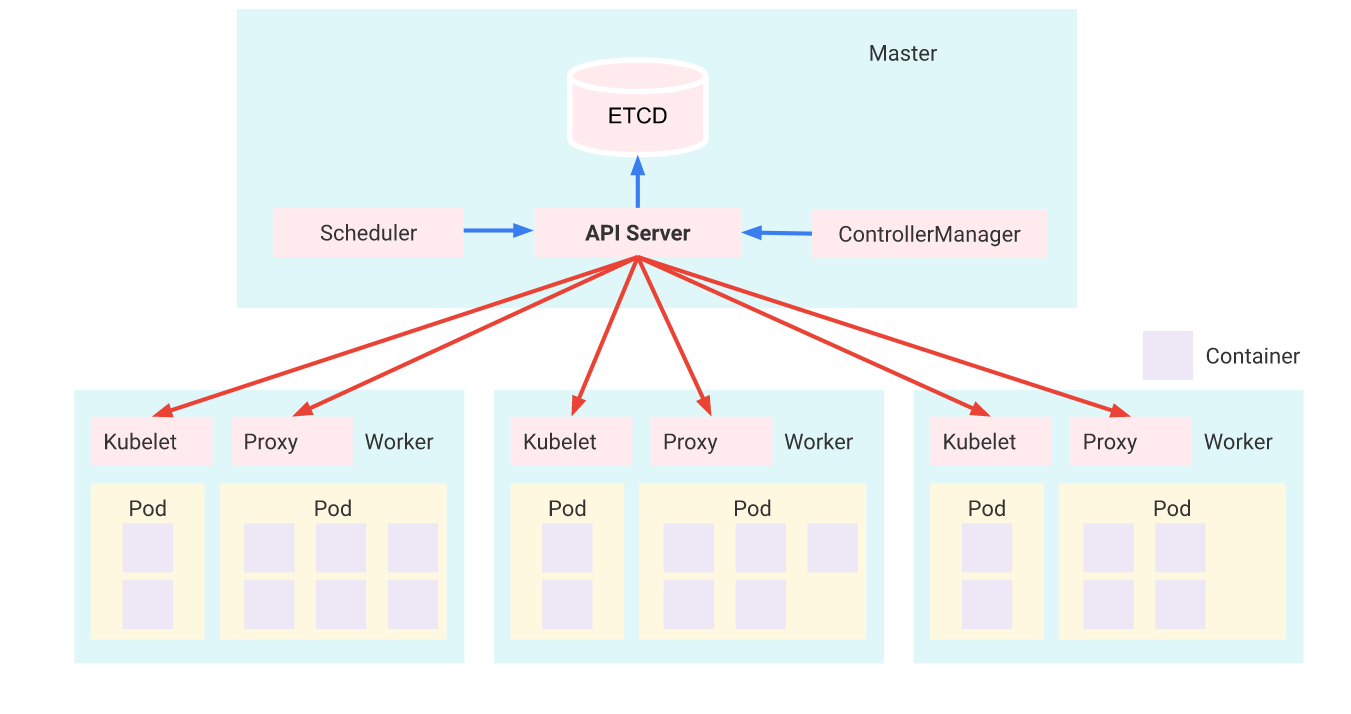
Kubernetes est un système open source extensible et portable qui vise à fournir une [plate-forme](https://fr.wikipedia.org/wiki/Plate-forme_(informatique)) permettant d'automatiser le déploiement, la montée en charge et la mise en œuvre de [conteneurs d'application](https://fr.wikipedia.org/wiki/Conteneur_(homonymie)#conteneur_d'application) sur des [clusters de serveurs](https://fr.wikipedia.org/wiki/Grappe_de_serveurs) fournissant une infrastructure centrée sur les conteneurs.

Avec Kubernetes, on peut répondre rapidement et efficacement aux demandes des clients

* Déployez vos applications rapidement et de manière prévisible.
* Mettez à l'échelle vos applications à la volée.
* Déployer de manière transparente de nouvelles fonctionnalités.
* Optimisez l'utilisation de votre matériel en utilisant uniquement les ressources dont vous avez besoin.

### **Architecture Kubernetes**

L’architecture d’un cluster de Kubernetes forme une relation maître/esclave, c’est-à-dire qu’il y a un composant principal qui contient le plan de contrôle (master node) et d’autres composants appelés nœuds (worker nodes) qui forment les machines de calculs, Plusieurs composants entrent en jeu dans le plan de contrôle, Les nœuds peuvent être répartis sur plusieurs machines physiques ou virtuelles. Les nœuds contiennent les pods qui sont les plus petits composants de Kubernetes et dans lesquels on retrouve les applications conteneurisées.



*Figure6 : architecture de Kubernetes*.

### **Les composants de Kubernetes**

Nous allons maintenant voire un à un les principaux composants de Kubernetes.

* **Pods** :

Un pod Kubernetes est la plus petite unité de gestion d'un cluster Kubernetes. Les containers sont installés dans une ressource de Kubernetes appelé pod. Un pod est un ensemble d’un ou plusieurs containers. En effet, Kubernetes ne fonctionne pas en interagissant directement avec les containers mais avec des pods. Ces pods peuvent être considérés comme des machines avec des adresses IP uniques.



*Figure7 : pod*

Sur la Figure : Pod, les composants « volume » se trouvant dans les pods représentent un espace de stockage de données. Il en existe différents types. Chaque pod contient un fichier « descriptor » qui indique les paramètres d’installation du pod.

Le pod est la ressource avec la plus petite granularité du système Kubernetes. Ils seront toujours hébergés au sein d’un worker node.

* **Worker Node :**

Un Node est là où sont placées toutes les charges de travail dans Kubernetes, c’est-à-dire qu’il exécute les tâches que le plan de contrôle lui assigne. Un nœud peut être une machine physique ou virtuelle qui détient toutes les ressources nécessaires afin de garantir l’exécution d’un ou plusieurs pods . Un clusterest l’appellation que l’on donne à un ensemble de nodes. Cette entité va héberger tous les services qu’un développeur aura décidé de déployer (figure).

****

*Figure8 : worker node*

Chaque WN est composé d’un Kubelet, d’un kube-proxy et d’un Container runtime.

### **Kubelet**

Le Kubelet a comme responsabilité de gérer tous les pods ainsi que leurs containers contenus dans le WN. Lui-même contient un composant appelé « cAdvisor » qui lui s’occupe de récupérer des métriques sur chaque container de chaque pods.

### **kube-proxy**

[kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) est un proxy réseau qui s'exécute sur chaque nœud du cluster et implémente une partie du concept Kubernetes de [Service](https://kubernetes.io/fr/docs/concepts/services-networking/service/). Kube-proxy maintient les règles réseau sur les nœuds. Ces règles réseau permettent une communication réseau vers les Pods depuis des sessions réseau à l'intérieur ou à l'extérieur du cluster. kube-proxy utilise la couche de filtrage de paquets du système d'exploitation s'il y en a une et qu'elle est disponible. Sinon, kube-proxy transmet le trafic lui-même.

### **Container Runtime**

Le Container runtime est extérieur à Kubernetes. C’est le composant qui gère les containers dans un WN, Docker dans le cadre de notre mémoire.

Pour créer un container, le Kubelet inspecte le descriptor d’un pod du WN et communique les images qui y sont mentionnées au Container runtime via le protocole REST. Ce dernier télécharge l’image à partir de DockerHub si l’image n’est pas disponible localement et instancie un container avec cette image.

Si un pod est supprimé ou dysfonctionne, le Kubelet a la responsabilité d’arrêter le ou les containers qui y étaient hébergés via le container runtime.

* **Master node**

Le master node a la responsabilité d’administrer le cluster. Il coordonne les activités telles que la mise en échelle des applications, la maintenance des applications à l’état désiré et la propagation des mises à jour.

**Composants du master node**

### **kube-apiserver**

Est le principal centre de communication pour tous les composants du cluster. Il est chargé d’orchestrer toutes les opérations au sein des composants du cluster et expose également l’API Kubernetes utilisée par les utilisateurs externes qui gèrent les charges de travail de conteneur. Les composants du nœud de travail communiquent également avec le plan de contrôle via le serveur d’API Kube.

### **Etcd**

Est une base de données qui stocke toutes les informations de nœud et les informations de charge de travail de conteneur entant que paire clé-valeur consistance et hautement disponible.

### **Kube-scheduler**

Est responsable de la planification des charges de travail de conteneur pour les nœuds de travail en tenant compte de la demande de ressources informatiques des charges de travail, des ressources disponibles dans les nœuds, du type décharge de travail autorisé dans les nœuds de travail et des autres politiques et contraintes de gestion.

### **Kube-Controller-manager**

Se compose de plusieurs gestionnaires de contrôleurs. Le contrôleur de nœud est responsable de l’intégration des nouveaux nœuds au cluster et de la gestion de l’indisponibilité des nœuds. Le contrôleur de réplication garantique le nombre souhaité de charges de travail de conteneur est en cours d’exécution tout le temps. En plus de ces deux contrôleurs, de nombreux autres contrôleurs aident à gérer différentes fonctionnalités.

Les composants du cluster communiquent avec l’API Server, car il est le seul à pouvoir communiquer avec l’etcd et par conséquent est le seul à pouvoir mettre à jour l’état du cluster.



Les composants de Kubernetes écoutent les évènements qui les intéressent selon leurs responsabilités au sein du système et vont réagir quand ils auront reçu un changement de l’état du cluster.

* **Fonctionnalité Kubernetes**

**Auto-guérison** : redémarre les conteneurs qui échouent, remplace et re-planifie les conteneurs lorsque les nœuds meurent, tue les conteneurs qui ne répondent pas au contrôle d'intégrité défini par l'utilisateur et les publie auprès des clients seulement lorsqu’ils sont prêts.

**Conditionnement en bac automatique** : place automatiquement les conteneurs en fonction de leurs besoins en ressources et d'autres contraintes, sans sacrifier la disponibilité. Co-localisez les charges de travail critiques et “best-effort” afin de maximiser l'utilisation des ressources.

**Mise à l'échelle horizontale et mise à l'échelle automatique** : Faites passez votre application à l’échelle à l'aide d'une simple commande, d’une interface graphique, ou automatiquement en fonction de l'utilisation du processeur ou de métriques personnalisées.

**Déploiements et restaurations automatisés :** Kubernetes déploie progressivement les modifications apportées à votre application ou à sa configuration, tout en surveillant l'intégrité de l'application afin de s'assurer qu'elle ne tue pas toutes vos instances en même temps. En cas de problème, Kubernetes annulera les changements pour vous.

**Découverte de services et équilibrage de charge :** inutile de modifier votre application pour utiliser un mécanisme de découverte de service tiers. Kubernetes donne aux conteneurs leurs propres adresses IP et un seul nom DNS pour un ensemble de conteneurs, et peut équilibrer la charge entre eux.

**Gestion des secrets et de la configuration :** Déployez et mettez à jour les secrets et la configuration de l'application sans reconstruire votre image et sans exposer les secrets les fichiers de configuration de votre déploiement.

**Orchestration du stockage** : monte automatiquement le système de stockage de votre choix, qu'il s'agisse du stockage local, d'un fournisseur de cloud public tel que GCP ou AWS, ou d'un système de stockage réseau tel que NFS, iSCSI, Gluster, Ceph, Cinder ou Flocker.

**Exécution par lots :** En plus des services, Kubernetes peut gérer vos batch et votre CI, en remplaçant les conteneurs qui échouent, si vous le souhaitez.

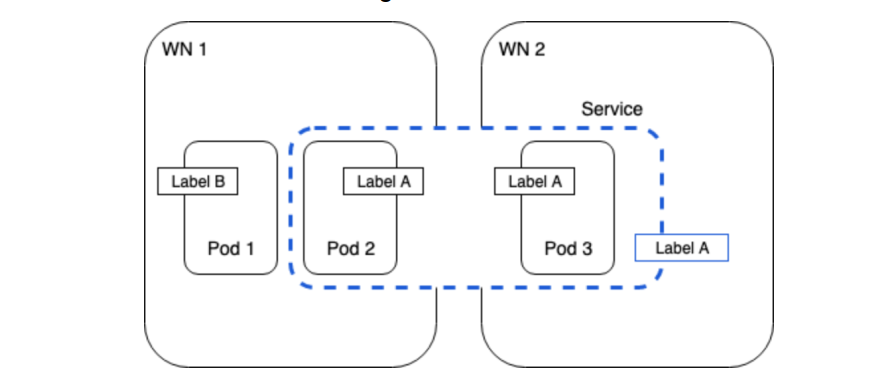
* **Concepts de Kubernetes**

**Les Différents Services de Kubernetes**

Un Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) identifiant un ensemble de pods à l'aide de sélecteurs d'étiquettes. Sauf indication contraire, les services sont supposés avoir des adresses IP virtuelles routables uniquement dans le réseau du cluster.

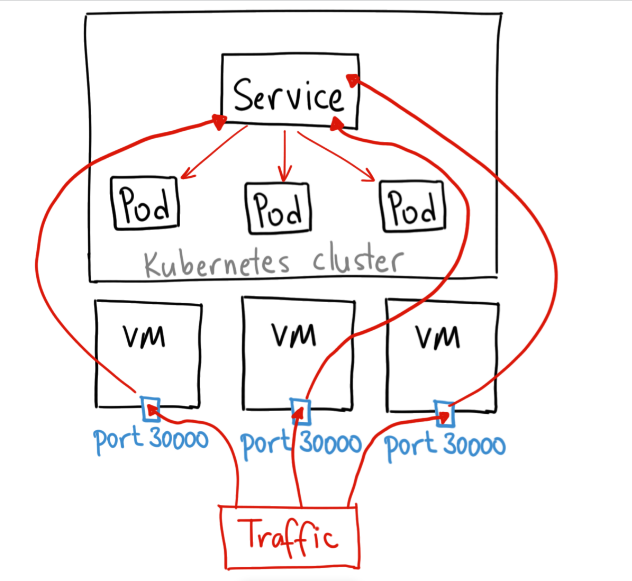
Un service dans Kubernetes est un objet REST, semblable à un pod. Comme tous les objets REST, vous pouvez effectuer un POST d'une définition de service sur le serveur API pour créer une nouvelle instance. Un Service Ingress est un objet Kubernetes qui gère l'accès externe aux services dans un cluster, généralement du trafic HTTP.

Comme les ressources sont gérées dynamiquement par Kubernetes, il faut pouvoir disposer d’un point d’entrée sur les microservices car on ne connait pas à priori leurs adresses.



*Figure11 : d’illustration de service Kubernetes*

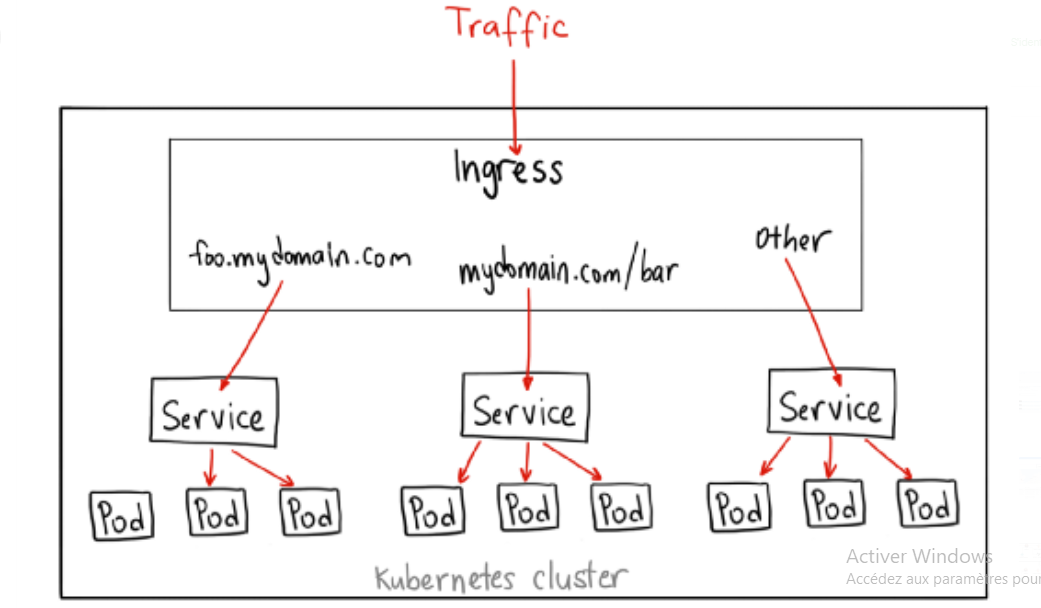
Un « Service » c’est le point d’accès aux microservices contenus dans les images des containers contenus dans les pods. Ces derniers possèdent un label afin de pouvoir les identifier. La Figure11 : Service montre que les pods avec le Label A sont regroupés dans un même « Service », même s’ils sont localisés sur différents worker nodes. En s’adressant à ce « Service », un client externe peut effectuer une requête et atteindre un microservice dans le pod de label A sans connaître sa localisation exacte. Comme il y a plusieurs Pods par « Service », Kubernetes réalise ainsi une forme de loadbalancing : quand une requête arrive sur un « Service », Kubernetes lui assigne un Pod disponible.



*Figure12 : Composition et fonctionnement d’’un service Kubernetes*

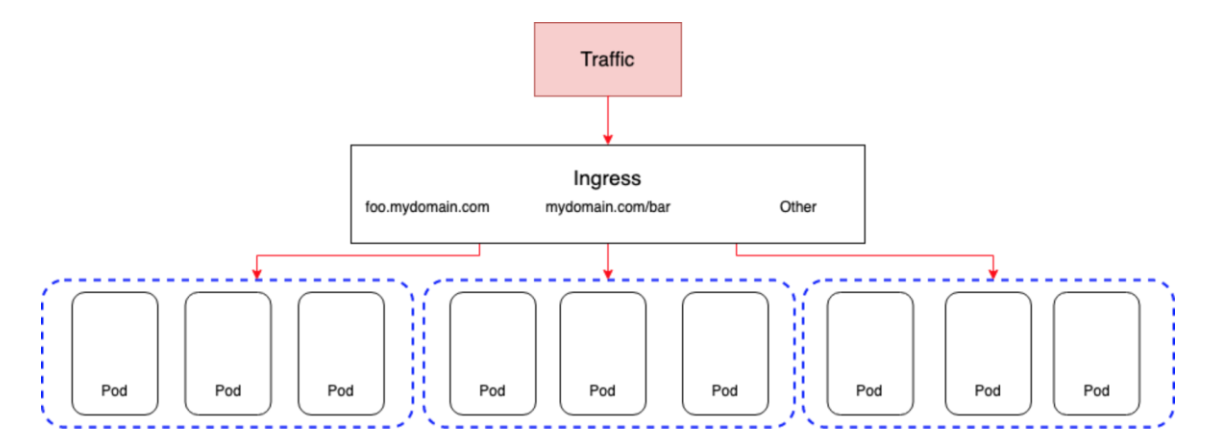
Il existe plusieurs façons d’atteindre un Service, selon les techniques appelées :

* **Le Service Ingress**



*Figure13 : Composition et fonctionnement d’’un service Ingress Kubernetes*

Dans ce mode interne à Kubernetes, il n’y a qu’un seul point d’entrée au trafic et donc une seule adresse IP pour l’ensemble des « Services ». L’Ingress redirige les requêtes externes au bon « Service » en utilisant un chemin fourni dans la requête, car chaque service possède un nom de domaine. Par exemple : AdresseIPIngress.foo.mydomain.com redirigera la requête vers le « Service ».



*Figure14 : Illustration d’’un service Ingress Kubernetes*

**ExempleVersion API**

**apiVersion**: networking.k8s.io/v1

**kind**: Ingress

**metadata**:

**name**: test-ingress

**annotations**:

**nginx.ingress.kubernetes.io/rewrite-target**: /

**spec**:

**rules**:

- **http**:

**paths**:

- **path**: /testpath

**pathType**: Prefix

**backend**:

**service**:

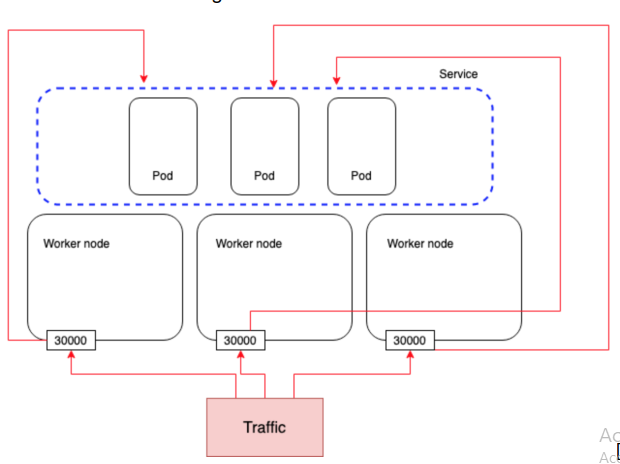
**name**: test

**port**:

**number**: 80

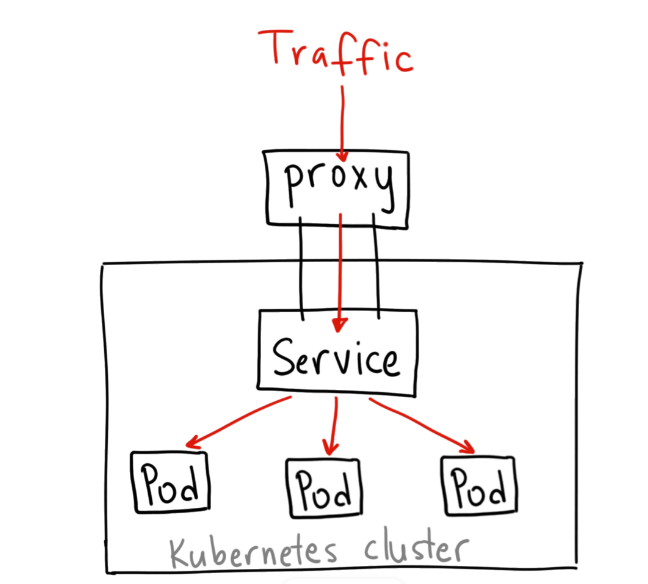
* **Le Service NodePort**

Dans ce mode, le Service est exposé via un port spécifique et commun sur chaque worker node. Sur la Figure 1 : NodePort Service nous voyons que le « Service » est atteignable via le port 30000 des WNs. Ce port sera disponible sur tous les WNs, même ceux qui ne contiennent pas un pod qui est contenu par le « Service ».



*Figure15 : Illustration d’’un service Node Port Kubernetes*

Lorsqu’une requête d’un client externe arrive sur ce port, la requête est redirigée vers le « Service » qui la redirigera vers un des pods capable de répondre à la requête. C’est une redirection en deux étapes.



*Figure16 : Composition et fonctionnement d’’un service Node Port Kubernetes*

**ExempleVersion API**

Un service peut mapper *n'importe quel* port entrant vers un targetPort. Par défaut et pour plus de commodité, le targetPort a la même valeur que le champ port. Par exemple, supposons que vous ayez un ensemble de pods qui écoutent chacun sur le port TCP 9376 et portent une étiquette app.kubernetes.io/name=MyApp:

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app.kubernetes.io/name**: MyApp

**ports**:

- **protocol**: TCP

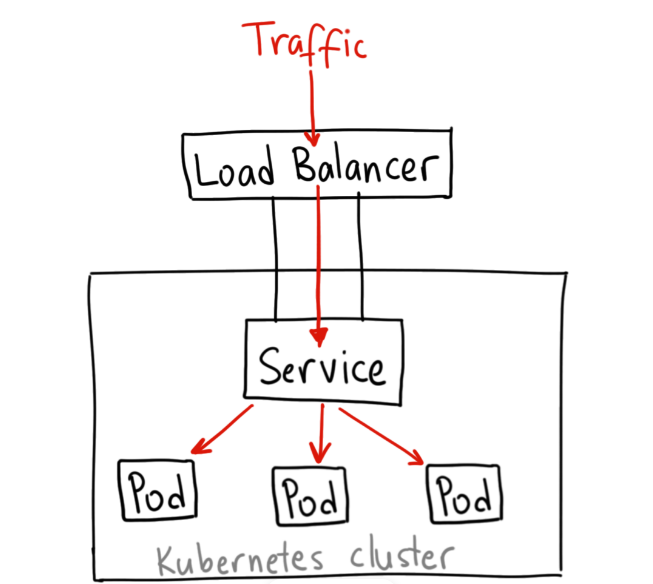
**port**: 80

**targetPort**: 3000

* **Le Service LoadBalancer**

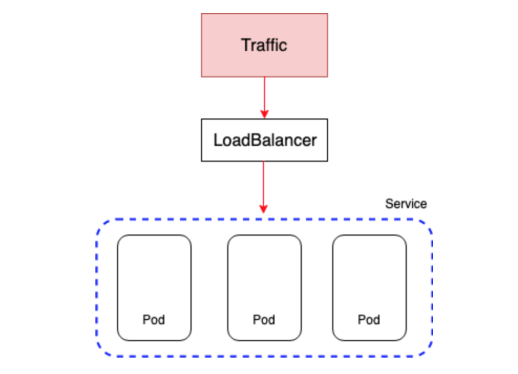
Le LoadBalancer est un service externe à Kubernetes. Il permet de définir une adresse IP pour chaque « Service » et redirige toutes les requêtes vers le service. On doit donc se reposer sur un fournisseur externe pour pouvoir disposer d’un LoadBalancer.

Les clusters provenant d’infrastructure cloud et payantes comme Google Kubernetes Engine (GKE) ou Azure Kubernetes Service, l’intègrent à leur offre. Avec ce mode d’accès, il faut un LoadBalancer par « Service », qui va être facturé par le provider de l’infrastructure cloud ce qui est coûteux si notre cluster offre beaucoup de services. Le LoadBalancer peut supporter plusieurs protocoles, HTTP, TCP et UDP, par exemple.

****

*Figure17 : Composition et fonctionnement d’’un service LoadBalancer Kubernetes*

Le trafic provenant du loadbalancer externe est dirigé vers les Pods backend. Le fournisseur de cloud décide de la répartition de la charge. Certains fournisseurs de cloud vous permettent de spécifier le loadBalancerIP. Dans ces cas, le loadbalancer est créé avec le loadBalancerIP spécifié par l'utilisateur. Si le champ loadBalancerIP n'est pas spécifié, le loadBalancer est configuré avec une adresse IP éphémère. Si vous spécifiez un loadBalancerIP mais que votre fournisseur de cloud ne prend pas en charge la fonctionnalité, le champ loadBalancerIP que vous définissez est ignoré.



*Figure18 : Illustration d’’un service LoadBalancer Kubernetes*

**ExempleVersion API**

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app.kubernetes.io/name**: MyApp

**ports**:

- **protocol**: TCP

**port**: 80

**targetPort**: 3000

**clusterIP**: 10.0.171.239

**type**: LoadBalancer

**status**:

**loadBalancer**:

**ingress**:

- **ip**: 192.0.2.127

**Replicaset**

Un ReplicaSet (ensemble de réplicas en français) a pour but de maintenir un ensemble stable de Pods à un moment donné. Cet objet est souvent utilisé pour garantir la disponibilité d'un certain nombre identique de Pods.

Un ReplicaSet est défini avec des champs, incluant un selecteur qui spécifie comment identifier les Pods qu'il peut posséder, un nombre de replicas indiquant le nombre de Pods qu'il doit maintenir et un modèle de Pod spécifiant les données que les nouveaux Pods que le replicatSet va créer jusqu'au nombre de replicas demandé.

Un ReplicaSet va atteindre son objectif en créant et supprimant des Pods pour atteindre le nombre de réplicas désirés. Quand un ReplicaSet a besoin de créer de nouveaux Pods, il utilise alors son Pod template.

Le lien d'un ReplicaSet à ses Pods est fait par le champ [metadata.ownerReferences](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/" \l "owners-and-dependents), qui spécifie la ressource de l'objet par lequel il est détenu. Tous les Pods acquis par un ReplicaSet ont leurs propres informations d'identification de leur Replicaset, avec leur propre champ ownerReferences. C'est par ce lien que le ReplicaSet connait l'état des Pods qu'il maintient et agit en fonction de ces derniers.

Un ReplicaSet identifie des nouveaux Pods à acquérir en utilisant son selecteur. Si il y a un Pod qui n'a pas de OwnerReference ou que OwnerReference n'est pas un controller et qu'il correspond à un sélecteur de ReplicaSet, il va immédiatement être acquis par ce ReplicaSet.

## Quand utiliser un ReplicaSet ?

Un ReplicaSet garantit qu’un nombre spécifié de réplicas de Pod soient exécutés à un moment donné. Cependant, un déploiement est un concept de plus haut niveau qui gère les ReplicaSets et fournit des mises à jour déclaratives aux Pods ainsi que de nombreuses autres fonctionnalités utiles. Par conséquent, nous vous recommandons d’utiliser des déploiements au lieu d’utiliser directement des ReplicaSets, sauf si vous avez besoin d'une orchestration personnalisée des mises à jour ou si vous n'avez pas besoin de mises à jour.

Cela signifie qu'il est possible que vous n'ayez jamais besoin de manipuler des objets ReplicaSet : utilisez plutôt un déploiement et définissez votre application dans la section spec

N.B : Un *pod* (terme anglo-saxon décrivant un groupe de baleines ou une gousse de pois) est un groupe d'un ou plusieurs conteneurs (comme des conteneurs Docker), ayant du stockage/réseau partagé, et une spécification sur la manière d'exécuter ces conteneurs.

**Stockage - Volumes**

Docker a également un concept de [volumes](https://docs.docker.com/storage/), bien qu'il soit, dans une certaine mesure, plus relâché et moins géré. Avec Docker, un volume est simplement un dossier sur le disque ou dans un autre conteneur. Les durées de vie ne sont pas gérées et, jusqu'à très récemment, seuls les volumes supportés par un disque local l'étaient. Docker fournit maintenant des pilotes de volume, mais la fonctionnalité est très limitée pour le moment (par exemple, à partir de Docker 1.7, seulement un pilote de volume est autorisé par conteneur et il n'est pas possible de passer des paramètres aux volumes).

Un volume Kubernetes, en revanche, a une durée de vie explicite - la même que le Pod qui l'inclut. Par conséquent, un volume survit aux conteneurs qui s'exécutent à l'intérieur du Pod et les données sont préservées lorsque le conteneur redémarre. Bien sûr, lorsqu'un Pod cesse d'exister, le volume va également cesser d'exister. Peut-être plus important encore, Kubernetes supporte de nombreux types de volumes et un Pod peut en utiliser plusieurs simultanément.

À la base, un volume est juste un dossier, contenant possiblement des données, qui est accessible aux conteneurs dans un Pod. La manière dont ce dossier est créé, le support qui le sauvegarde et son contenu sont déterminés par le type de volume utilisé.

Pour utiliser un volume, un Pod spécifie les volumes à fournir au Pod (le champ .spec.volumes) et où les monter dans les conteneurs (le champ .spec.containers.volumeMounts).

Un processus dans un conteneur a une vue système de fichiers composée de son image et de ses volumes Docker. L'[image Docker](https://docs.docker.com/userguide/dockerimages/) est à la racine de la hiérarchie du système de fichiers et tous les volumes sont montés sur les chemins spécifiés dans l'image. Les volumes ne peuvent pas être montés sur d'autres volumes ou avoir des liens physiques vers d'autres volumes. Chaque conteneur dans le Pod doit spécifier indépendamment où monter chaque volume.

**Types de Volumes**

Kubernetes supporte plusieurs types de Volumes :

* [awsElasticBlockStore](https://kubernetes.io/fr/docs/concepts/_print/#awselasticblockstore)
* [azureDisk](https://kubernetes.io/fr/docs/concepts/_print/#azuredisk)
* [azureFile](https://kubernetes.io/fr/docs/concepts/_print/#azurefile)
* [cephfs](https://kubernetes.io/fr/docs/concepts/_print/#cephfs)
* [cinder](https://kubernetes.io/fr/docs/concepts/_print/#cinder)

### **Conclusion :**

Dans ce chapitre nous avons présenté en détail la plateforme d’orchestration des conteneurs qu’est Kubernetes. Nous avons étudié son architecture et vu comment ses différents éléments fonctionnement entre eux. Nous avons vu également les différents services et concepts de kubernetes.

# **Chapitre 2 : Déploiement Canary sur kubernetes**

## **Présentation :**

En génie logiciel, le déploiement Canary consiste à créer des versions échelonnées, nous déployons d’abord une mise à jour logicielle pour une petite partie des utilisateurs, afin qu’ils puissent la tester et fournir des commentaires, une fois la modification acceptée, la mise à jour est déployée pour le reste des utilisateurs.

Les déploiements Canary nous montrent comment les utilisateurs interagissent avec les changements d'application dans le monde réel, la stratégie Canary offre des mises à niveau sans temps d'arrêt et des restaurations faciles.

Dans ce chapitre, nous allons mettre en place dans un environnement local, le déploiement Canary du kubernetes.

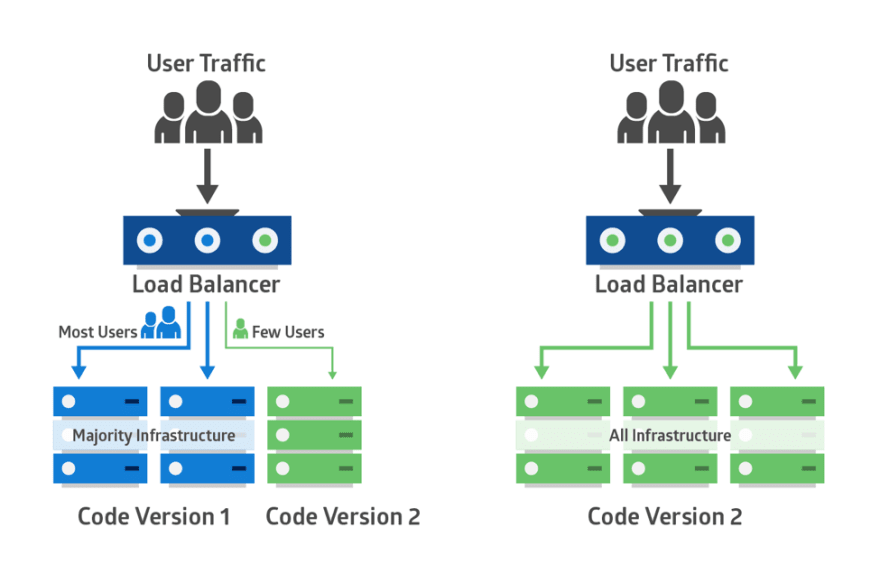
Plusieurs étapes sont nécessaires pour la mise en œuvre du déploiement Canary : on site les 3 étapes initiales :

* Création de deux clones de l’environnement de production ;
* Mise en disposition d’un équilibreur de charge qui est nécessaire pour router initialement tout le trafic vers une version ;
* Création de nouvelles fonctionnalités dans l’autre version.

Lors du déploiement de la nouvelle version logicielle, on transfert un certain pourcentage de la base des utilisateurs par exemple 15% vers cette nouvelle version et garder 85% des utilisateurs sur l’ancienne version. Si les 15% ne signalent aucune anomalie, on peut désormais commencer l’étape de déploiement progressivement sur un grand nombre d’utilisateurs jusqu’à déployer la nouvelle version pour tout le monde. Si les 15% signalent des problèmes, on peut faire retour arrière et 85% des utilisateurs n’auront jamais rencontré ces problèmes.

Le déploiement progressif de Canary est facile avec une restauration rapide et offre une sécurité supplémentaire puisque l’environnement reste en marche sans temps d’arrêt.

Le déploiement de Canary est plus intéressant et utile lorsque l’équipe de développement logiciel n’est pas sûre de la nouvelle version et un déploiement progressif est un avantage qui offre la possibilité de corriger les bugs. Dans le cas où, ils sont sûrs que la nouvelle version fonctionne correctement sans erreurs ni problèmes, un déploiement bleu/vert s’impose.



*Figure 1 : fonctionnement Canary*

## **Déploiement de Canary du kubernetes**

Les étapes ci-dessous nous montrent comment configurer un déploiement Canary. Nous allons créer un cluster Kubernetes simple de pods Nginx avec une page HTML statique de base de deux phrases. Les versions du déploiement varient en fonction du contenu qu’elles affichent sur la page web.

**Etape 1 : Installation et manipulation d’un cluster kubernetes avec Minikube.**

Nous avons travaillé avec la solution Minikube dans le déploiement de Canary kubernetes parce que c’est un outil facilitant l’exécution locale de kubernetes et il exécute un cluster kubernetes à nœud unique dans une machine virtuelle (VM), dès lors la VM devient à la fois un nœud de type Master et Worker.

**Prérequis**

* **Préparation de l’environnement locale**

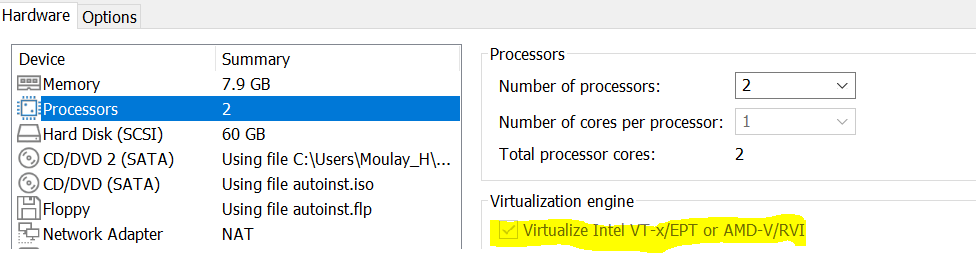
Nous avons installé sur une machine locale l’outil de virtualisation Vmware workstation afin de créer une machine virtuelle. Ensuite on a installé la dernière version de système d’exploitation Ubuntu (22.04).

* **Hyperviseur**

On a d’abord commencé par vérifier si la virtualisation est prise en charge sous notre machine linux par l’utilisation de la commande

# **egrep –color ‘vmx|svm’ /proc/cpuinfo**

Après la sortie est vide et on a activer la fonctionnalité sur le paramètre VT -x/ AMD-V de la machine virtuel comme montre la figure 3.1.

****

**Figure 1 : activation paramètre VT -x/ AMD-V**

Après on a installé l’hyperviseur VirtualBox sur la VM comme montre la figure 3.2

Une image contenant texte

Description générée automatiquement

**Figure 2 : VirtualBox**

* **Docker**

On a installé l’outil docker pour créer, déployer et exécuter des applications dans des conteneurs.

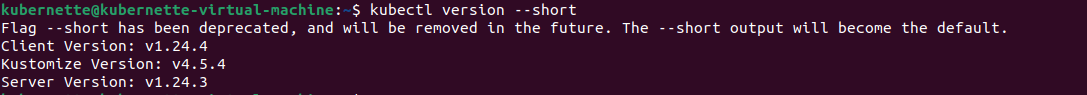
Une image contenant texte

Description générée automatiquement

**Figure 3 : Docker**

* **Kubectl**

On a installé la dernière version de l’outil en ligne de commande de kubernetes « Kubectl » afin de permette d’exécuter des commandes dans les clusters Kubernetes, cet outil va nous permettre de déployer des applications, inspecter et gérer les ressources du cluster et consulter les logs.



**Figure 4 : kubectl version**

**Etape 2 : Installation de minikube**

Une fois les prérequis satisfaits, On a installé l’outil Minikube qui fait tourner un cluster Kubernetes en un nœud unique dans une machine virtuelle sur la machine virtuelle qu’on a créée.

Une image contenant texte

Description générée automatiquement

**Figure 5 : Minikube version**

**Etape 3 : Création du cluster Kubernetes avec Minikube**

Pour créer le cluster kubernetes, on a utilisé la commande #minikube start pour créer et configurer une machine virtuelle qui exécute un cluster Kubernetes à un seul nœud, et configuré également notre installation de kubectl de manière à communiquer avec notre cluster.

Une image contenant texte

Description générée automatiquement

**Figure 6 : état cluster Minikube**

On va utiliser l’outil kubectl afin de récupérer la liste des nœuds de notre cluster Kubernetes

**#kubectl get nodes**

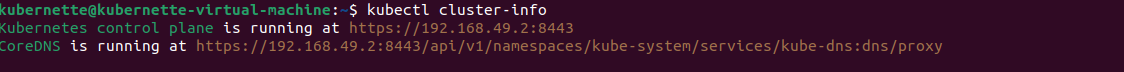
On ne récupère qu’un seul nœud de type Master comme montre la figure 5



**Figure 7 : Nœud cluster**

On peut récupérer des informations sur notre cluster kubernetes grâce à la commande ci-dessous

**#Kubectl cluster-info**

****

**Figure 8 : info cluster**

## **Mise en place de Canary kubernetes**

**Etape1 : Pull Docker Image**

La première étape consiste à extraire ou à créer l’image des conteneurs de notre cluster Kubernetes. Puisque nous construisons des conteneurs Nginx dans cet exemple, nous utilisons l’image Nginx disponible sur Docker Hub

1.1 Téléchargement de l’image avec la commande **#docker pull nginx**

1.2 Vérification de l’image comme montre la figure

****

**Figure : image Nginx**

**Etape 2 : création le déploiement Kubernetes**

On a créé la définition de déploiement à l’aide d’un fichier yaml, on a nommé le fichier nginx-deployment.yaml par l’éditeur de texte VI.

**#vi nginx-deployment.yaml**

On a ajouté le script dans le fichier qu’on a créé

**Une image contenant texte

Description générée automatiquement**

**Figure : déploiement Nginx**

A partir de ce déploiement, on a créé 3 répliques de pods Nginx pour le cluster Kubernetes, tous les pods ont la version d’étiquette (label) : « 1.0 », De plus ils ont un volume hôte contenant le fichier index.html monté sur le conteneur.

On a créé un code html.

Une image contenant texte

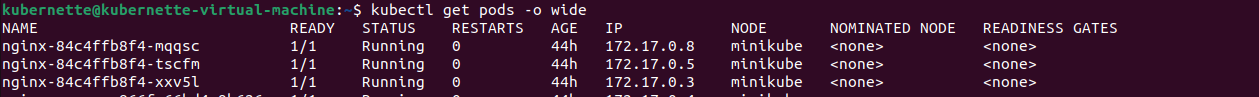
Description générée automatiquement

**Figure : exemple code Html**

**Etape 3 : Création de déploiement en utilisant la commande suivante :**

**#Kubectl apply -f nginx-deplyment.yaml**

**Etape 4 : Vérification la réussite de déploiement des pods**

****

**Figure : déploiement pods**

On voit que les trois pods sont en cours d’exécution

**Etape 5 : Création de service**

On va créer une définition de service pour le cluster kubernetes afin que le service achemine les demandes vers les pods spécifiés.

* Création d’un nouveau fichier yaml et on ajoute le script comme suit :

**vi nginx-deployment.service.yaml**

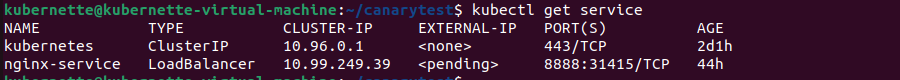
Dans le fichier yaml, on a spécifié le type de service « loadbalancer » son fonctionnement est d’équilibrer les charges de travail entre les pods avec l’application de libellés nginx et la version : « 1.0 ».

La commande pour créer de service : Kubectl apply -f nginx-deployement.service.yaml.

**Etape 6 : Vérification de la première version du cluster**

Pour vérifier le service, on doit récupérer leur adresse et pour faire ça on tape la commande ci-dessous :

**#Kubectl get service**

****

**Figure : adresse ip service**

Après on tape l’adresse ip sur l’url avec le port qu’on a défini comme montre la figure ci-dessous :

**Une image contenant texte

Description générée automatiquement**

**Figure : application version 1**

1. **Création de déploiement Canary**

La version 1 de l’application est mise en place, on va suivre les mêmes étapes pour le déploiement Canary de la version 2.

* Création de fichier yaml pour le déploiement Canary avec le script comme montre la figure suivante :

**Une image contenant texte

Description générée automatiquement**

**Figure : fichier YAML**

Le contenu du fichier de déploiement Canary diffère par trois paramètres importants :

Le nom dans les métadonnées est nginx-canary-deployement.

L’étiquette de la version : « 2.0 »

Tel est le contenu de fichier html de la version 2 :

Une image contenant texte

Description générée automatiquement

Figure : code HTML version2

Création de déploiement avec la commande :

Kubectl apply -f nginx-canary-deployement.yaml

**Verification**

**Kubectl get pods -o wide**

1. **Exécution de déploiement canary**

Quand on ouvre le navigateur web et on accède à la même adresse IP qu’à l’étape précédente, on remarque qu’il n’y a aucun changement sur la page web. En effet, le fichier de service est configuré pour équilibrer la charge uniquement des pods avec la version d’étiquette : « 1.0 »

Pour tester le pods mise a jours on aura besoin de modifié le fichier de service par la suppression de la ligne de la version : « 1.0 ».

Le fichier doit inclure les informations suivantes

**Une image contenant texte

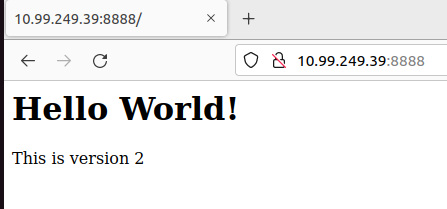
Description générée automatiquement**

Figure : service

Après on créer le service en utilisant la commande suivante :

**#Kubectl apply -f nginx-deployment.service.yaml**

Maintenant le trafic est splitté entre les pods version 1 et version 2, si on actualise la page web, on verra différents résultats fonction de la redirection de la requête par le service.



**Figure : application version 2**

## **Conclusion :**

Nous avons constaté qu’une partie de trafic est redirigée vers le déploiement Canary. En effet, il existe trois répliques de la version 1 et trois répliques de la version 2. Cependant, on peut rediriger un plus petit pourcentage des demandes par la configuration de déploiement Canary pour avoir moins de pods et on peut augmenter progressivement le nombre de répliques si on est sûr que le canary peut gérer plus de trafic.

# **Conclusion Générale**

Aujourd’hui, la haute disponibilité constitue la norme. Les utilisateurs s’attendent donc à une accessibilité constante des applications. La conception des logiciels et applications basée sur des conteneurs limite les interruptions de service. De plus, elle permet un gain de performance et d’évolutivité.

La stratégie de déploiement de Canary kubernetes est largement utilisée car elle réduit le risque de déplacer les modifications en production tout en réduisant le besoin d'infrastructure supplémentaire. Les organisations utilisant Canary peuvent tester la nouvelle version dans un environnement de production en direct sans exposer simultanément tous les utilisateurs à la dernière version.

La phase d’étude de cette technologie était assez complexe car cette technologie est en constante évolution et très vaste. On a dû passer par une étude de l’architecture de Kubernetes : ses composants et leurs interactions, afin de comprendre le fonctionnement du système global.

Durant la partie pratique, on a rencontré beaucoup des difficultés dans le début mais avec le temps on a pu dépasser les obstacles avec l’aide de non seulement de l’encadreur, mais aussi du doctorant. Fort heureusement aussi, Kubernetes possède une grande communauté d’utilisateurs, très active et une excellente documentation.

Pour conclure, on est très satisfait par les connaissances et compétences acquises lors de la rédaction de ce mémoire. On espère pouvoir les approfondir dans l’insertion professionnelle.

# **Références bibliographiques :**

[**https://kubernetes.io/docs/home/**](https://kubernetes.io/docs/home/)

[**https://fr.scribd.com/search?query=kubernete**](https://fr.scribd.com/search?query=kubernete)**s**

<https://www.entrepriseprevention.com/kubernetes/>

<https://kubernetes.io/fr/docs/concepts/workloads/controllers/replicaset>

Auto-Formation chez Alphorm.com : Administrer le cluster, les labels et les Pods.