

Formation :

Orchestrer ses conteneurs virtuels avec Kubernetes



Formateur
Brahim Hamdi
brahim.hamdi.consult@gmail.com

Juin 2024

Brahim HAMDI

- Consultant/Formateur DevOps & Cloud
- Expert DevOps, Cloud, CyberOps et Linux



- Introduction à kubernetes
- Architecture Kubernetes
- Exploiter Kubernetes
- Kubernetes en production

- Décrire les principes de l'orchestration de conteneurs
- Manipuler les ressources de base Kubernetes
- Déployer des applications et les mettre à disposition.

- Développeurs
- Architectes
- Administrateurs systèmes

- Avoir de sérieuses compétences en système en Linux / Unix
- Connaître les technologies de conteneurs.

- <https://kubernetes.io/docs>

Introduction à Kubernetes

- CNCF
- Qu'est ce que Kubernetes ?
- Kubernetes : Points forts et faiblesses
- Quelques fonctionnalités de Kubernetes
- Quelques orchestrateurs de conteneurs
- Kubernetes vs Docker Swarm
- Solutions d'installation

CNCF: Cloud Native Computing Foundation

- Une fondation à but non lucratif pour promouvoir le Cloud Natif.
- Elle réunit des projets open source, des entreprises, des passionnés.
- CNCF a été fondée en Décembre 2015 et fait partie de la Fondation Linux.



- **Les projets hébergés par la CNCF**



Orchestration



Prometheus

Monitoring



OPENTRACING

Distributed Tracing API



fluentd

Logging



linkerd

Service Mesh



Remote Procedure Call



CoreDNS

Service Discovery



Container Runtime



rkt

Container Runtime



CNI

Networking API



envoy

Service Mesh



JAEGER

Distributed Tracing

Les membres Platinum de la CNCF



- Un système robuste d'orchestration de conteneurs.
- 100% Open source, écrit en Go.
- Créé par Google, basé sur Borg et Omega, les systèmes qui fonctionnent aujourd'hui chez Google et dont la robustesse est éprouvée depuis plus de 10 ans.
- Il a grandi de façon exponentielle et est devenu le premier projet à être donné à la CNCF.
- La première version de production: v1.0.1 en Juillet 2015.
- Une nouvelle version mineure tous les trois mois depuis la v1.2.0 en Mars 2016.

- Open source et très actif.
- Une communauté très visible et présente dans l'évolution de l'informatique.
- Les pods tendent à se rapprocher plus d'une VM du point de vue de l'application (ressources de calcul garanties, une ip joignable sur le réseau local, multiprocess).
- Hébergeable de façon quasi-identique dans le cloud, on-premise ou en mixte.
- K8s est pensé pour la scalabilité et le calcul distribué.

- Beaucoup de points sont laissés à la décision du fournisseur de cloud ou des admins système :
 - Pas de solution de stockage par défaut, et parfois difficile de stocker « simplement » sans passer par les fournisseurs de cloud, ou par une solution de stockage décentralisé à part entière (iSCSI, Gluster, NFS...)
 - Beaucoup de solutions de réseau qui se concurrencent, demandant un comparatif fastidieux même si plusieurs leaders émergent comme Calico, Flannel, Weave ou Cilium
 - Pas de solution de loadbalancing par défaut : soit on se base sur le fournisseur de cloud, soit on configure un MetalLB

■ **Self-healing:**

- 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'il sont prêts.

■ **Horizontal scaling and autoscaling:**

- Faites passer 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.

■ **Service Discovery and Load Balancing:**

- 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.



■ **Secret and configuration management:**

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

■ **Automated rollouts and rollbacks:**

- Kubernetes déploie progressivement les modifications apportées à l'application, 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.

- **Docker-compose**: surcouche à Docker, simple (mise en place et appréhension), capacités assez limitées.
- **Docker Swarm**: gestion de cluster de machines, scaling, intégré à Docker.
- **Mesos/Marathon/DCOS**: gestion de (très gros) clusters de machines, scaling, gestion fine des ressources, templates et bibliothèques de déploiement.
- **Kubernetes (k8s)**: nombreuses fonctionnalités (clusters, scaling, templates, ressources, stockage), templates et bibliothèques (via helm), contrôle d'accès.

|   | Kubernetes | Docker Swarm |
|--|---|---|
| Installation & Configuration | Installation difficile et compliqué | Installation très facile: Intégration dans l'environnement Docker déjà réalisée |
| GUI | Offre un outil intégré : taableau de bord clair et facile à utiliser. | GUI uniquement avec un logiciel supplémentaire |
| Scalability | Un peu lent dans la mise à l'échelle | Mise à l'échelle extrêmement rapide: 5× plus rapide que Kubernetes |
| Auto-Scaling | Offre une mise à l'échelle automatique. | N'offre pas une mise à l'échelle automatique. |
| Logging & Monitoring | Fait partie intégrante de l'application | Uniquement avec un logiciel supplémentaire |

- **Google Kubernetes Engine (GKE)** (Google Cloud Platform)
 - L'écosystème Kubernetes développé par Google. Très populaire car très flexible tout en étant l'implémentation de référence de Kubernetes.
- **Azure Kubernetes Services (AKS)** (Microsoft Azure)
 - Un écosystème Kubernetes axé sur l'intégration avec les services du cloud Azure (stockage, registry, réseau, monitoring, services de calcul, loadbalancing, bases de données...).
- **Elastic Kubernetes Services (EKS)** (Amazon Web Services)
 - Un écosystème Kubernetes assez standard à la sauce Amazon axé sur l'intégration avec le cloud Amazon (la gestion de l'accès, des loadbalancers ou du scaling notamment, le stockage avec Amazon EBS, etc.)

■ **Rancher**

- Un écosystème Kubernetes très complet et entièrement open-source, non lié à un fournisseur de cloud. Inclut l'installation de stack de monitoring (Prometheus), de logging, de réseau mesh (Istio) via une interface web agréable.

■ **K3s**

- Un écosystème Kubernetes fait par l'entreprise Rancher et axé sur la légèreté.
- Il remplace etcd par une base de données Postgres, utilise Traefik pour l'ingress et Klipper pour le loadbalancing.

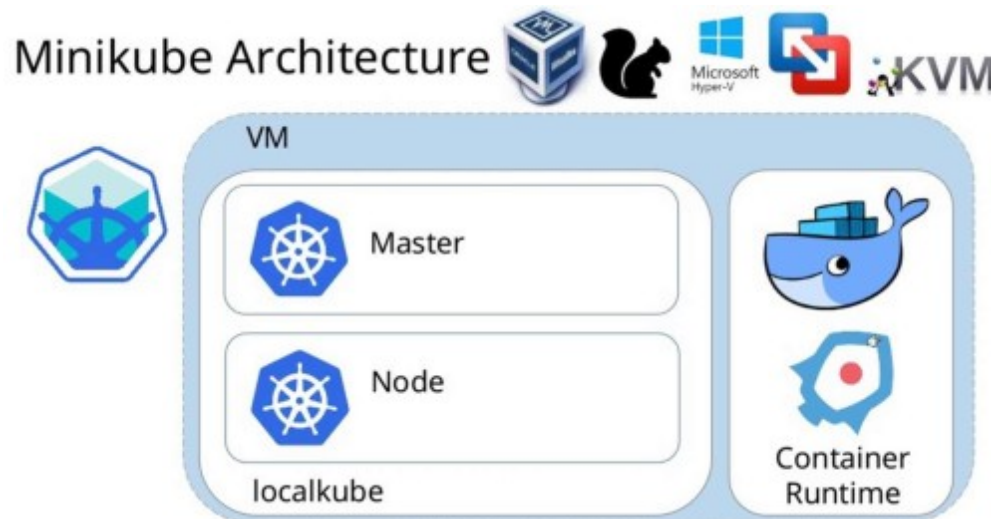
■ **Openshift**

- Une version de Kubernetes configurée et optimisée par Red Hat pour être utilisée dans son écosystème. Tout est intégré donc plus guidé, avec l'inconvénient d'être un peu captif de l'écosystème et des services vendus par RedHat.

■ **MiniKube**

■ Minikube

- Une solution qui vous permet d'utiliser Kubernetes sur votre machine de bureau ou votre laptop.
- N'est pas vraiment fait pour tourner en production
- Exécute un cluster Kubernetes à nœud unique dans une machine virtuelle (VM), dès lors votre VM devient à la fois un nœud de type Master et Worker.



- CNCF
- Qu'est ce que Kubernetes ?
- Kubernetes : Points forts et faiblesses
- Quelques fonctionnalités de Kubernetes
- Quelques orchestrateurs de conteneurs
- Kubernetes vs Docker Swarm
- Solutions d'installation

Architecture de Kubernetes

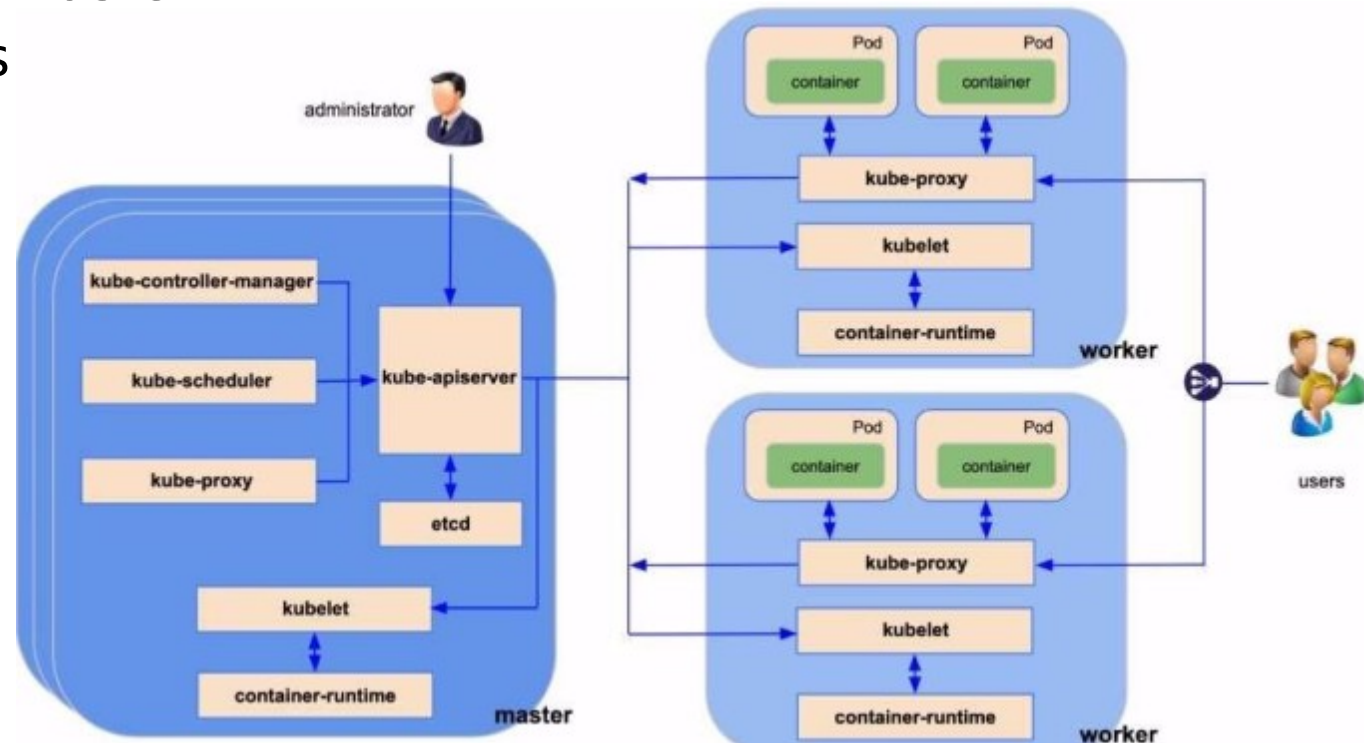
- Les différents types de nœud
- Architecture de master
- Composants de Control Plane
- Autres composants
- Architecture de Worker/node
- Les réseaux de Kubernetes
- Définition des objets
- Les pods

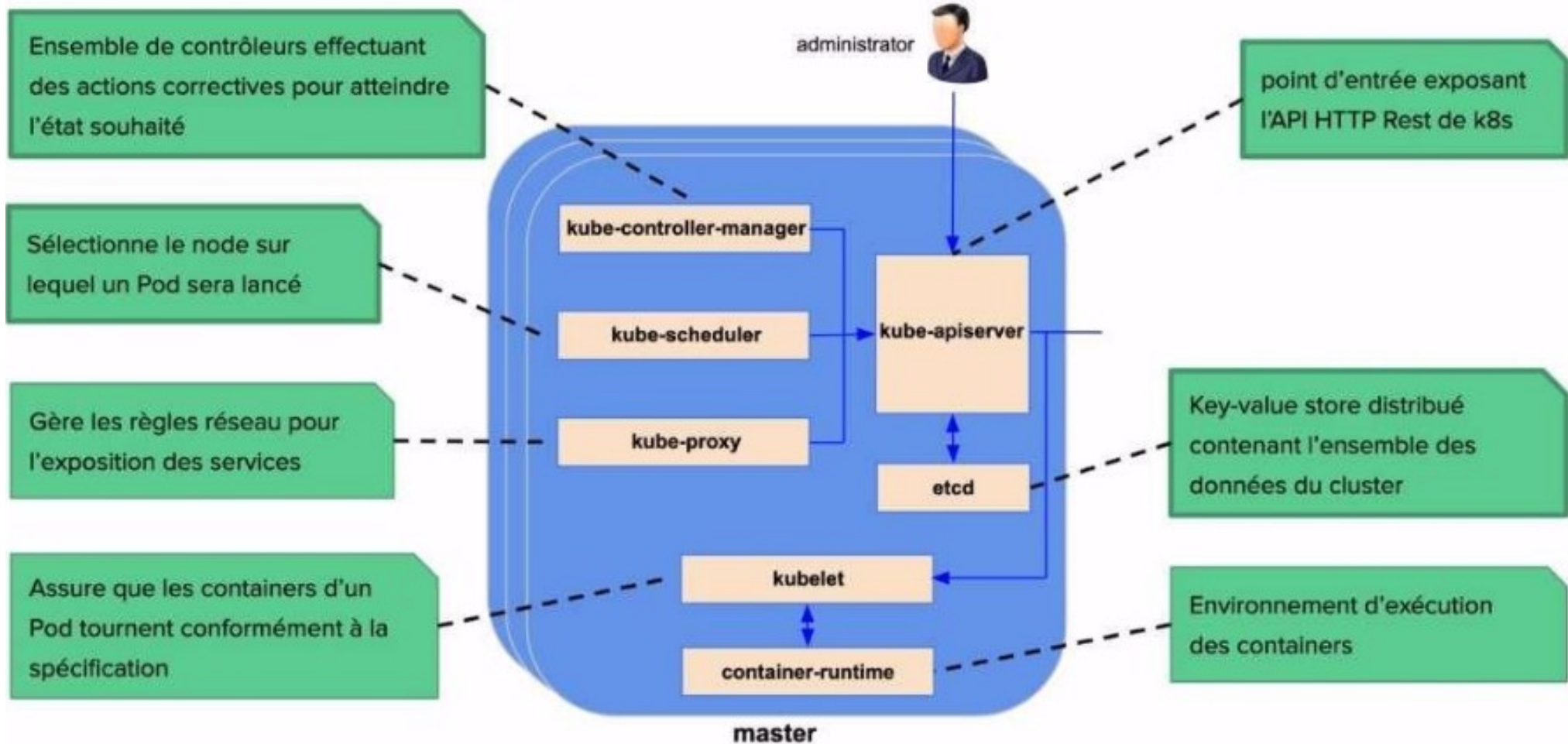
■ Master:

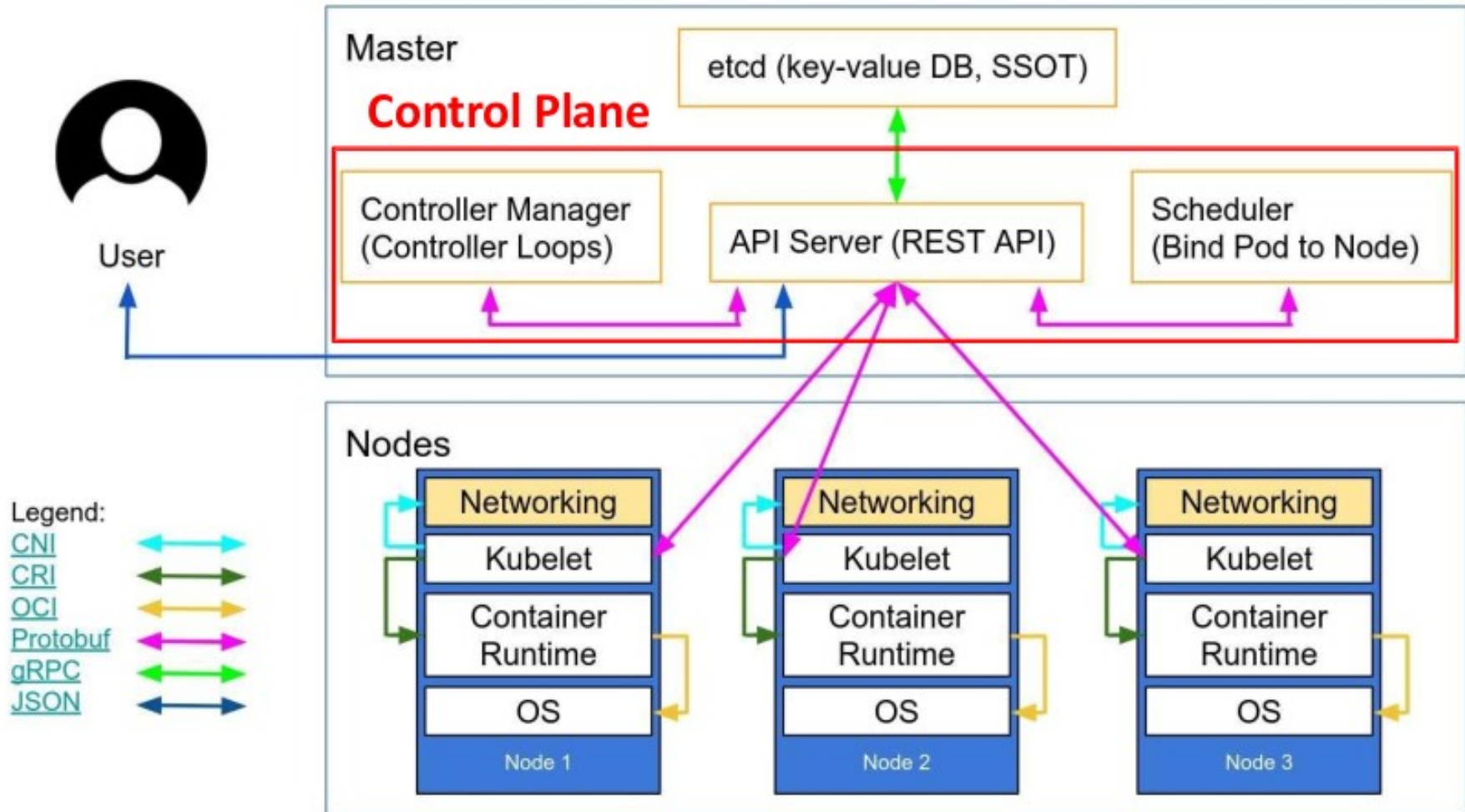
- Responsable de la gestion du cluster ("control plane")
- Expose l'API server
- Schedule les Pods sur les nodes du cluster

■ Worker/Node :

- Node sur lequel sont lancés les Pods applicatifs
- Communique avec le Master
- Fournit les ressources
- aux Pods

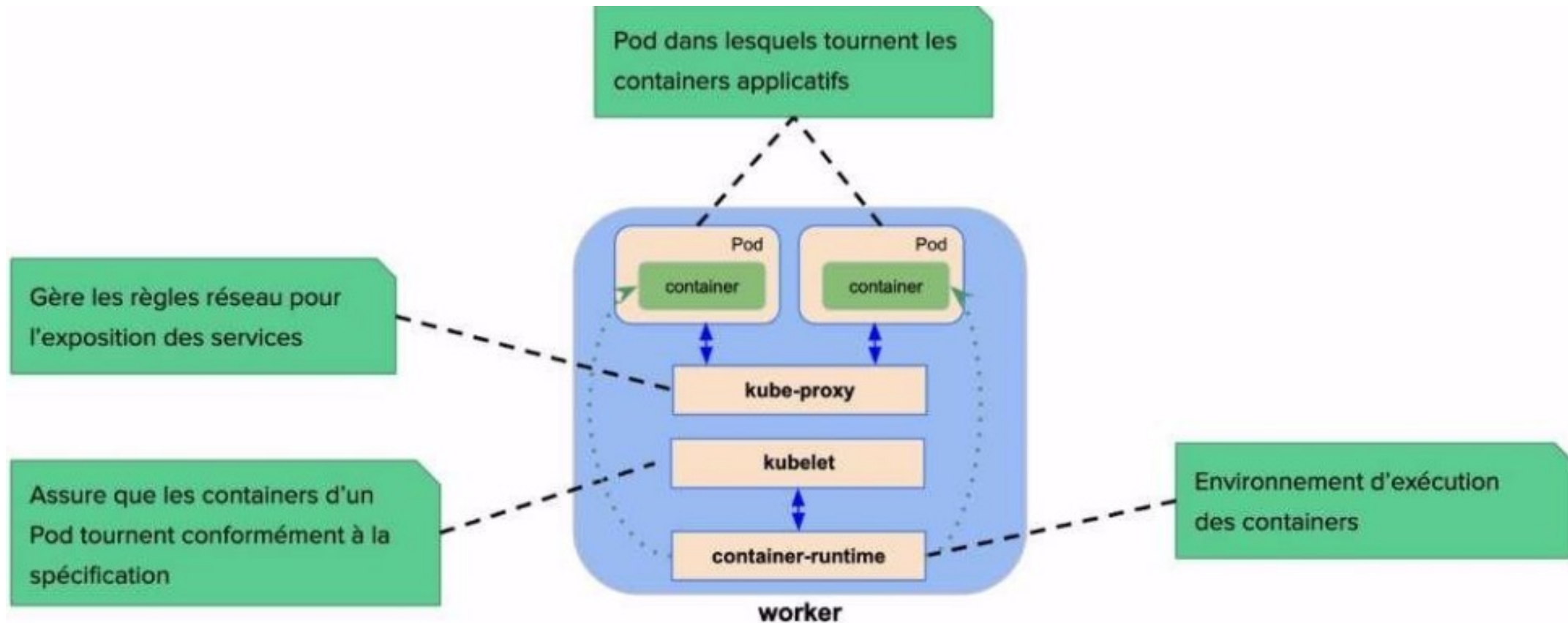






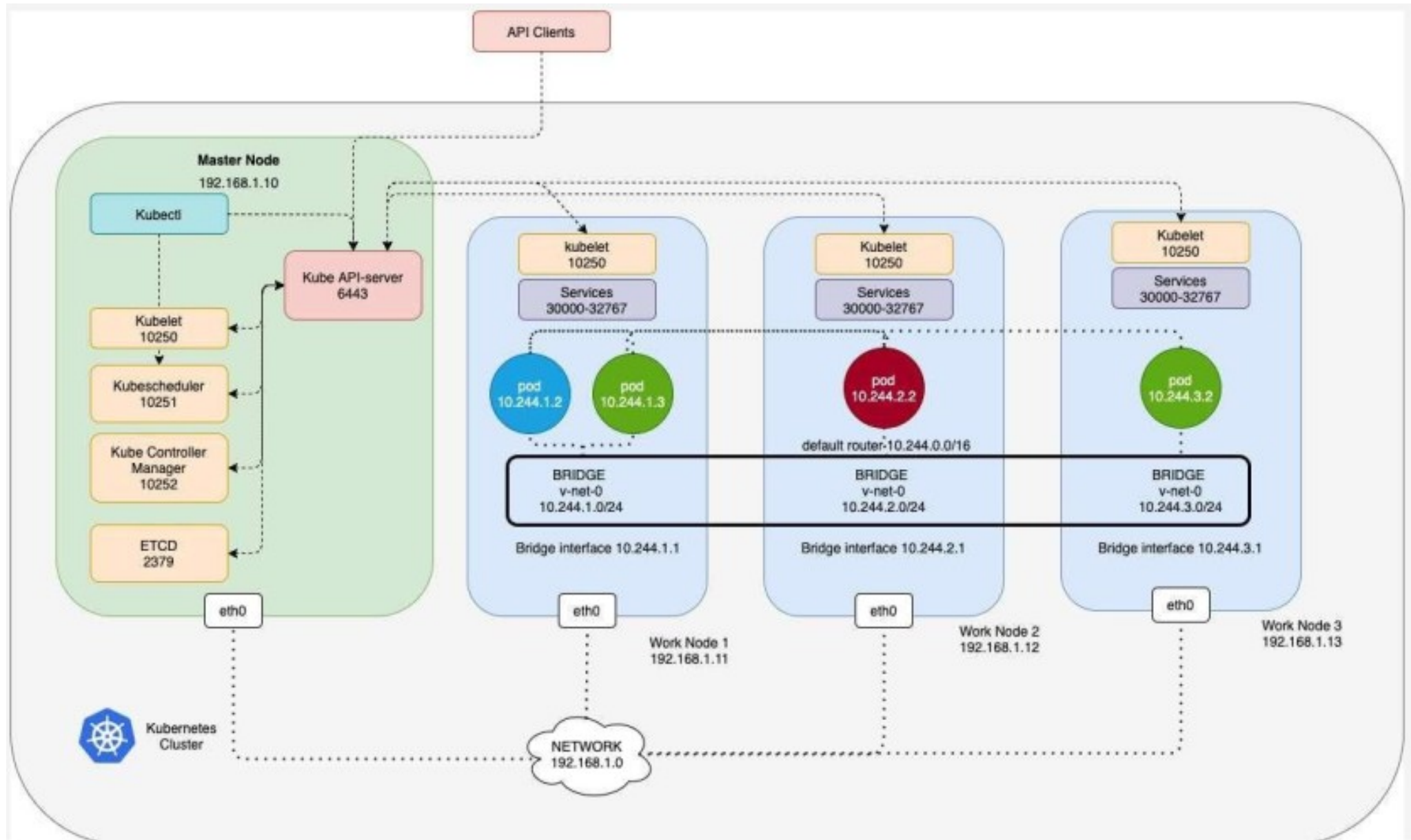
- KUBE-CONTROLLER-MANAGER
 - Boucle infinie qui contrôle l'état d'un cluster
 - Effectue des opérations pour atteindre un état donné
- KUBE-APISERVER
 - Les configurations d'objets (Pods, Service, RC, etc.) se font via l'API server
 - Tous les composants sont reliés à l'API serve
- KUBE-SCHEDULER
 - Planifie les ressources sur le cluster
 - En fonction de règles implicites (CPU, RAM, stockage disponible, etc.)
 - En fonction de règles explicites (règles d'affinité et anti-affinité, labels, etc.)

- KUBE-PROXY
 - Responsable de la publication des Services
 - Utilise iptables
 - Route les paquets à destination des conteneurs et réalise le load balancing TCP/UDP
- ETCD
 - Base de données de type Clé/Valeur (Key Value Store)
 - Stocke l'état d'un cluster Kubernetes
 - Point sensible (stateful) d'un cluster Kubernetes
 - Projet intégré à la CNCF
- kubelet : Service "agent" fonctionnant sur tous les nœuds et assure le fonctionnement des autres services
- kubectl : Ligne de commande permettant de piloter un cluster Kubernetes



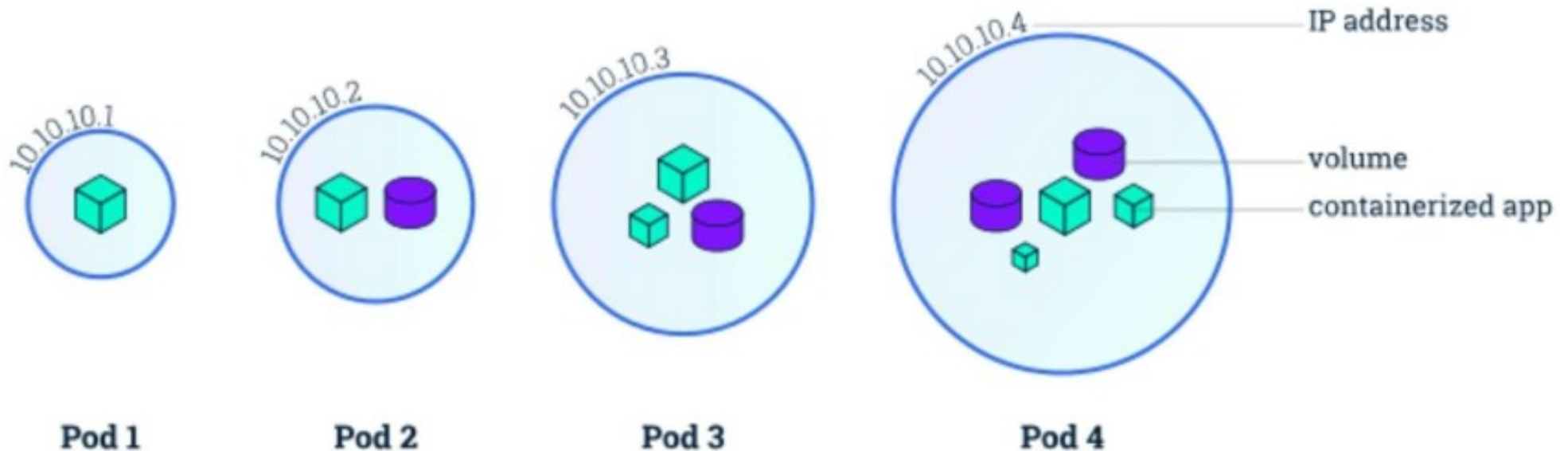
- Kubernetes n'implémente pas de solution réseau par défaut, mais s'appuie sur des solutions tierces ou plugins qui implémentent les fonctionnalités suivantes :
 - Gèrent le réseau pour la communication Pod-to-Pod conformément au standard CNI
 - Chaque pod reçoit sa propre adresse IP
 - Les pods peuvent communiquer directement sans NAT
- CNI : Container Network Interface
 - Projet dans la CNCF
 - Standard pour la gestion du réseau en environnement conteneurisé
 - Les trois solutions les plus utilisées sont Calico, Flannel et WeaveNet.

Les réseaux de Kubernetes



- Kubernetes contient un certain nombre d'abstractions représentant l'état de votre système: applications et processus conteneurisés déployés, leurs ressources réseau et disque associées, ainsi que d'autres informations sur les activités de votre cluster.
- Les contrôleurs s'appuient sur les objets de base et fournissent des fonctionnalités supplémentaires. Ces objets de base de Kubernetes incluent: Pod, Service, Volume et Namespace.
- Autres objets de Kubernetes:
 - ReplicaSet
 - Deployment
 - StatefulSet
 - DaemonSet
 - Job
 - Cron job

- Le Pod représente la plus petite unité déployable et exécutable au sein d'un cluster.
- C'est la brique de base qui encapsule un ou plusieurs conteneurs ayant une seule et unique @IP.
- Il a son propre réseau et volume interne.



- Les différents types de nœud
- Architecture de master
- Composants de Control Plane
- Autres composants
- Architecture de Worker/node
- Les réseaux de Kubernetes
- Définition des objets
- Les pods

Exploiter Kubernetes

- Les labels
- Pod avec plusieurs containers
- Le ReplicaSet
- Le Deployment
- Le DaemonSet
- Mise en réseau du cluster
- Les Services
- ConfigMap
- Secret
- Les Volumes



- Système de clé/valeur
- Organisent les différents objets de Kubernetes (Pods, RC, Services, etc.) d'une manière cohérente qui reflète la structure de l'application
- Corrèlent des éléments de Kubernetes : par exemple un service vers des Pods.
- Exemple des labels:
 - "release": "stable", "release": "canary"
 - "environment": "dev", "environment": "production"
 - "tier": "frontend", "tier" : "backend", "tier": "cache"
 - "partition": "customerA", "partition": "customerB"

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    environment: staging
    team: kube-team
spec:
  containers:
    - name: my-container
      image: "k8s.gcr.io/my-app:v0.1"
```

cat wp-pod.yaml

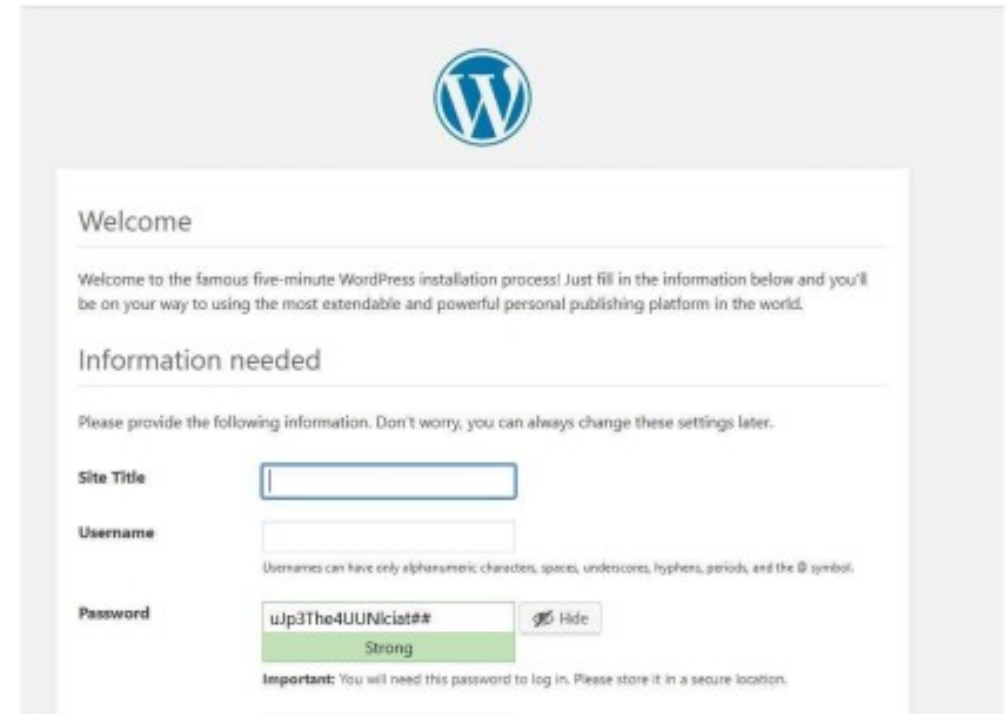
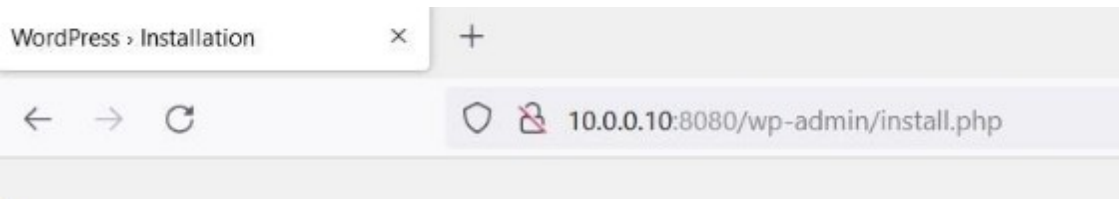
```
apiVersion: v1
kind: Pod
metadata:
  name: wp
spec:
  containers:
    - image: wordpress:4.9-apache
      name: wordpress
      env:
        - name: WORDPRESS_DB_PASSWORD
          value: mysqlpwd
        - name: WORDPRESS_DB_HOST
          value: 127.0.0.1
    - image: mysql:5.7
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: mysqlpwd
      volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
  volumes:
    - name: data
      emptyDir: {}
```

kubectl apply -f wp-pod.yaml

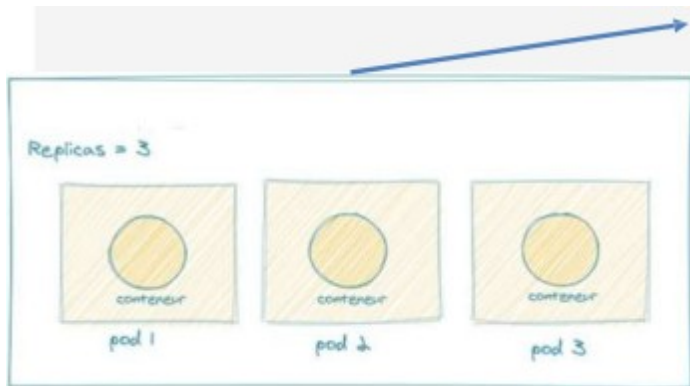
Kubectl get pods

| NAME | READY | STATUS | RESTARTS | AGE |
|------|-------|---------|----------|-----|
| wp | 2 / 2 | Running | 0 | 18s |

kubectl port-forward --address 0.0.0.0 wp 8080:80



- Un **ReplicaSet** pilote des **pods** et s'assure qu'un certain nombre de pods sont **lancés**
- Si un pod vient à **mourir**, le ReplicaSet va lancer **un nouveau pod**.
- Dans la réalité, on ne se sert que rarement directement des ReplicaSet , au profit d'un autre contrôleur, qui vient piloter un ReplicaSet : le **Deployment**.
- Par conséquent, il est recommandé d'utiliser des **Deployments** au lieu d'utiliser directement des ReplicaSets, sauf si vous avez besoin d'une orchestration personnalisée des mises à jour.



```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
  
```

```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3

```

ReplicaSet

Pod

```
# Lancement du ReplicaSet
$ kubectl create -f frontend.yaml
# Liste des ReplicaSets
$ kubectl get rs
```

| NAME | DESIRED | CURRENT | READY | AGE |
|----------|---------|---------|-------|-----|
| frontend | 3 | 3 | 3 | 11m |

Le ReplicaSet gère les 3 Pods (replicas) définis dans la spécification du ReplicaSet



```
# Liste des Pods
$ kubectl get pod
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|---------|----------|-----|
| frontend-5dmcr | 1/1 | Running | 0 | 13m |
| frontend-7kwfj | 1/1 | Running | 0 | 13m |
| frontend-l6mk8 | 1/1 | Running | 0 | 13m |

```
#Scaler les répliques de vos Pods
$ kubectl scale --replicas=4 replicaset frontend
```


- Un Deployment contrôle et pilote des ReplicaSets et des pods.
- Il ajoute des fonctionnalités aux ReplicaSet comme :
 - Déployer (rollout) des ReplicaSet : les ReplicaSets vont créer des pods en tâche de fond et le Deployment va s'assurer que le déploiement (rollout) se passe bien
 - Déclarer un nouvel état des pods : le Deployment va piloter la mise à jour des pods, des anciens ReplicaSet vers les nouveaux. Ce déploiement va créer une nouvelle révision du Deployment
 - Revenir à une ancienne version du Deployment (rollback).: si le déploiement engendre des instabilités, le Deployment nous permet de revenir à une précédente révision.
 - Mettre à l'échelle le Deployment pour gérer les montées en charge par exemple
 - Mettre en pause un déploiement : pour fixer des éventuelles erreurs et reprendre le rollout

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deploy
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook
  tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      Labels:
        app: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_sample_app-engine/php-frontend:v3

```

Deployment

ReplicaSet

Pod

Lancement du Deployment

\$ **kubectl create -f vote-deployment.yaml**

Liste des Deployments

\$ **kubectl get deploy**

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------|-------|------------|-----------|-----|
| vote-deploy | 3/3 | 3 | 3 | 11m |

Liste des ReplicaSet

\$ **kubectl get rs**

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------|---------|---------|-------|-----|
| vote-deploy-7c4d84d659 | 3 | 3 | 3 | 15m |

\$ **kubectl get pod**

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------|-------|---------|----------|-----|
| vote-deploy-7c4d84d659-4jblj | 1/1 | Running | 0 | 18m |
| vote-deploy-7c4d84d659-95nwp | 1/1 | Running | 0 | 18m |
| vote-deploy-7c4d84d659-mcd8x | 1/1 | Running | 0 | 18m |



Un ReplicaSet est créé et associé au Deployment



Le ReplicaSet gère les 3 Pods (replicas) définis dans la spécification du Deployment

#Scaler les répliques de vos Pods

```
$ kubectl scale --replicas=4 deploy vote-deploy
```

Détails du Deploy

```
$ kubectl describe deploy vote-deploy
```

```
Name:                vote-deploy
...
NewReplicaSet:        vote-deploy-7c4d84d659 (4/4 replicas
Events:               created)
Type    Reason          Age    From                      Message
----    -
Normal  ScalingReplicaSet  2m49s  deployment-controller  Scaled up replica set vote-deploy-7c4d84d659 to 3
Normal  ScalingReplicaSet  70s    deployment-controller  Scaled up replica set vote-deploy-7c4d84d659 to 4
```

Changer la valeur de replicas par 2

```
$ kubectl apply -f vote-deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deploy
spec:
  replicas: 2
...
```

- Un **DaemonSet** est un contrôleur qui va s'assurer qu'un seul et unique pod s'exécute sur un node
- Ainsi quand un node est ajouté au cluster, le DaemonSet va lancer lui même le pod qu'il définit
- À l'inverse quand le **DaemonSet est supprimé**, il **supprime avec lui le pod qu'il gère**
- Ne connaît pas la notion de réplicas.
- Utilisé pour des besoins particuliers comme :
 - l'exécution d'agents de collection de logs comme fluentd ou logstash
 - l'exécution de pilotes pour du matériel comme nvidia-plugin
 - l'exécution d'agents de supervision comme NewRelic agent ou Prometheus node exporter

apiVersion: apps/v1

kind: DaemonSet

metadata:

labels:

app: elasticsearch

name: elasticsearch

namespace: kube-system

spec:

selector:

matchLabels:

app: elasticsearch

template:

metadata:

labels:

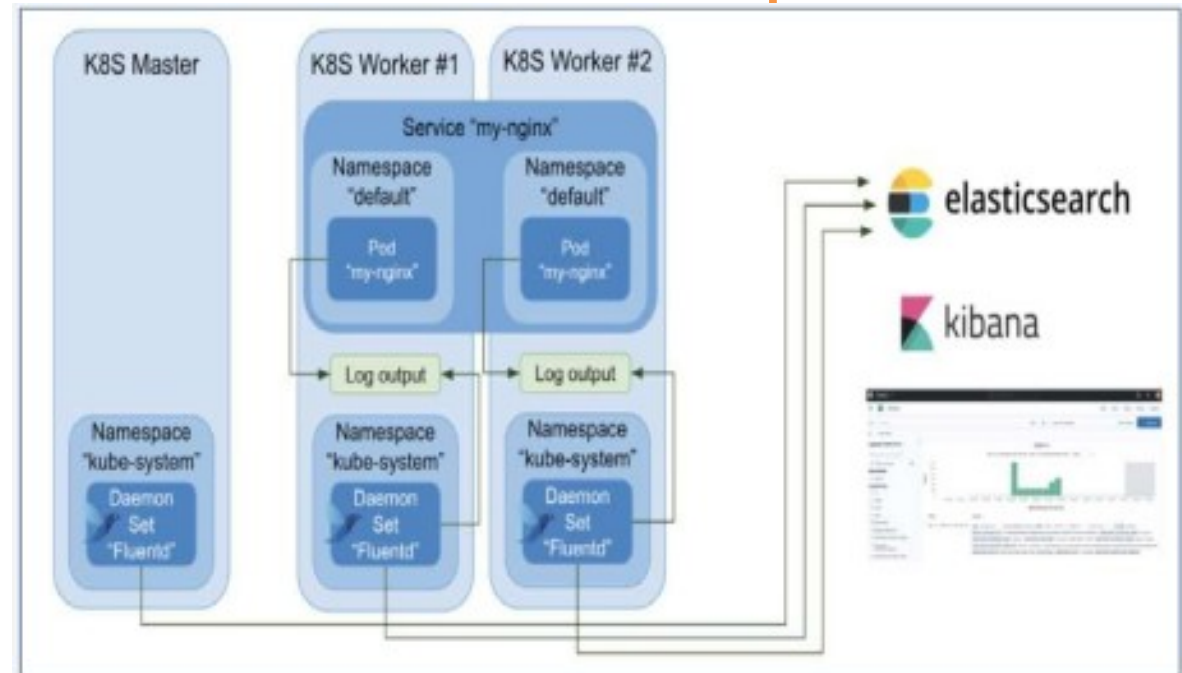
app: elasticsearch

spec:

containers:

- **image:** quay.io/fluentd_elasticsearch/fluentd:v2.5.2

name: fluentd-elasticsearch



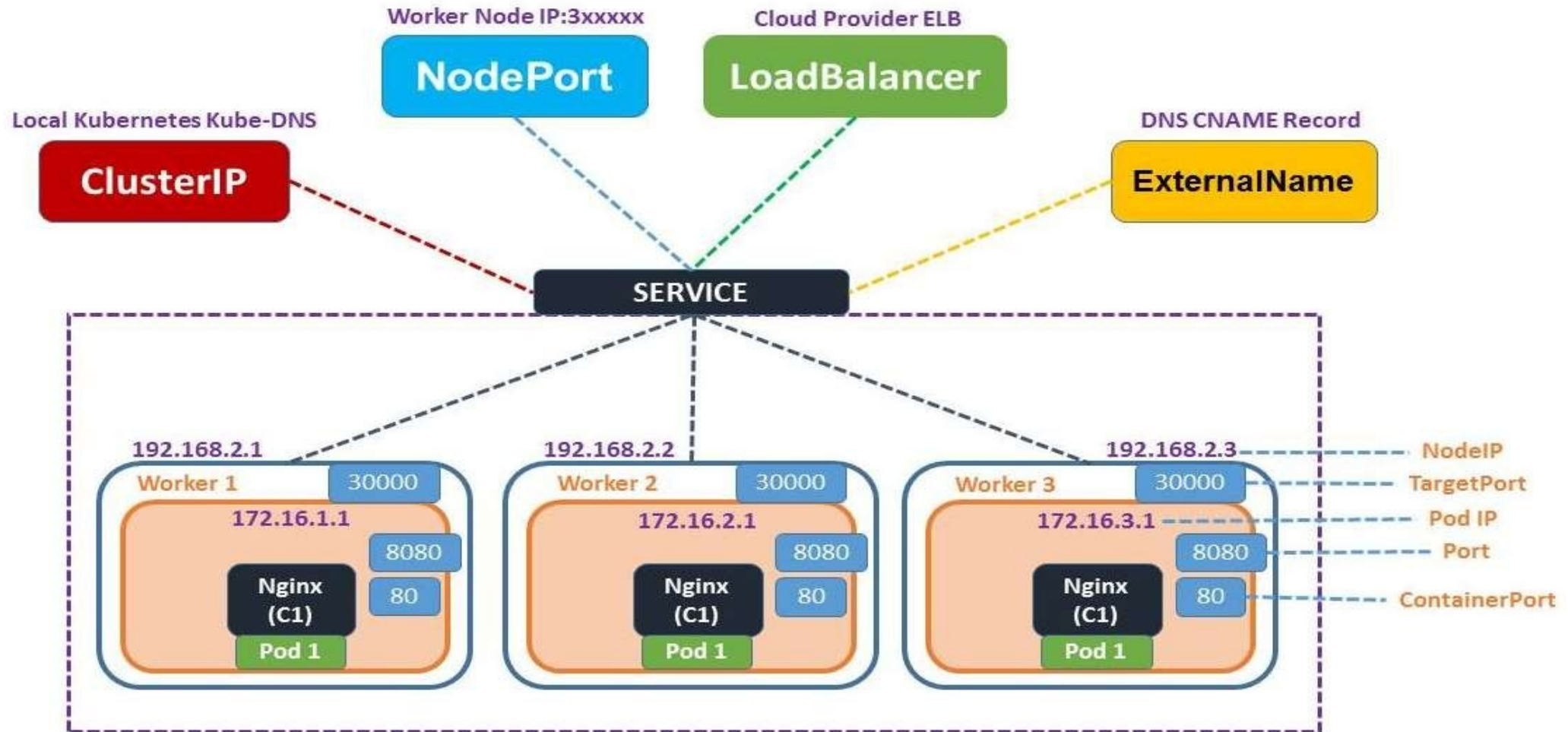
- Étant donné les similitudes entre les **DaemonSets**, les **StatefulSets** et les **Deployments**, il est important de comprendre quand les utiliser.
- **Les Deployments** (liés à des ReplicaSets) doivent être utilisés :
 - lorsque votre application est complètement **découplée du nœud**
 - que vous pouvez en exécuter **plusieurs copies sur un nœud** donné sans considération particulière
 - que l'ordre de création des replicas et le nom des pods **n'est pas important**
 - lorsqu'on fait des opérations **stateless**
- **Les DaemonSets** doivent être utilisés :
 - lorsqu'au moins **une copie de votre application doit être exécutée sur tous les nœuds du cluster** (ou sur un sous-ensemble de ces nœuds).

- Il y a 4 problèmes de réseau distincts à résoudre dans Kubernetes :
 - Communications de conteneur à conteneur hautement couplées: ce problème est résolu par les pods et les communications localhost. Les conteneurs à l'intérieur d'un pod peuvent communiquer entre eux à l'aide de localhost
 - Communications de pod à pod.
 - Communications de pod au service.
 - Communications externes au service



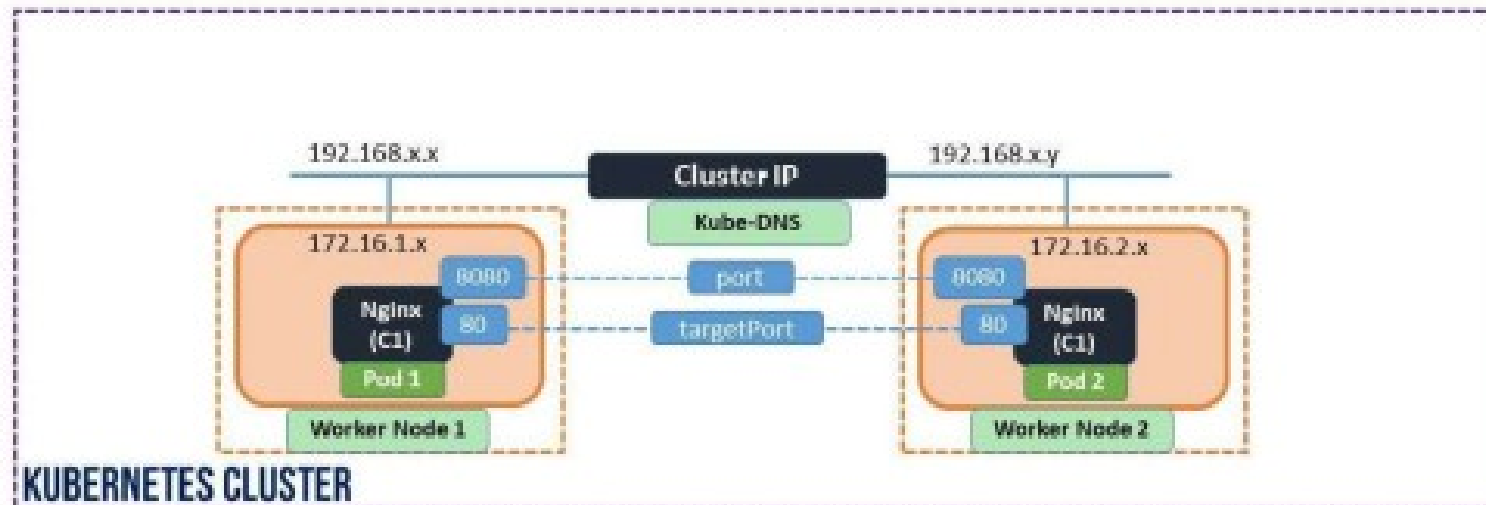
- Il existe quatre types de service:
 - ClusterIP
 - NodePort
 - LoadBalancer
 - ExternalName

- Vous pouvez également utiliser [Ingress](#) pour exposer votre service:
 - Ingress n'est pas un type de service, mais il sert de point d'entrée pour votre cluster.
 - Il vous permet d'exposer plusieurs services sous la même adresse IP.



- Expose le service sur une IP **interne au cluster**.
- Le choix de cette valeur rend le service uniquement accessible à partir du cluster.
- Il s'agit du ServiceType par **défaut**.

CLUSTER IP



```
$ cat vote-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: web
  labels:
    app: web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - containerPort: 80
```

```
$ cat vote-service-clusterIP.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
```

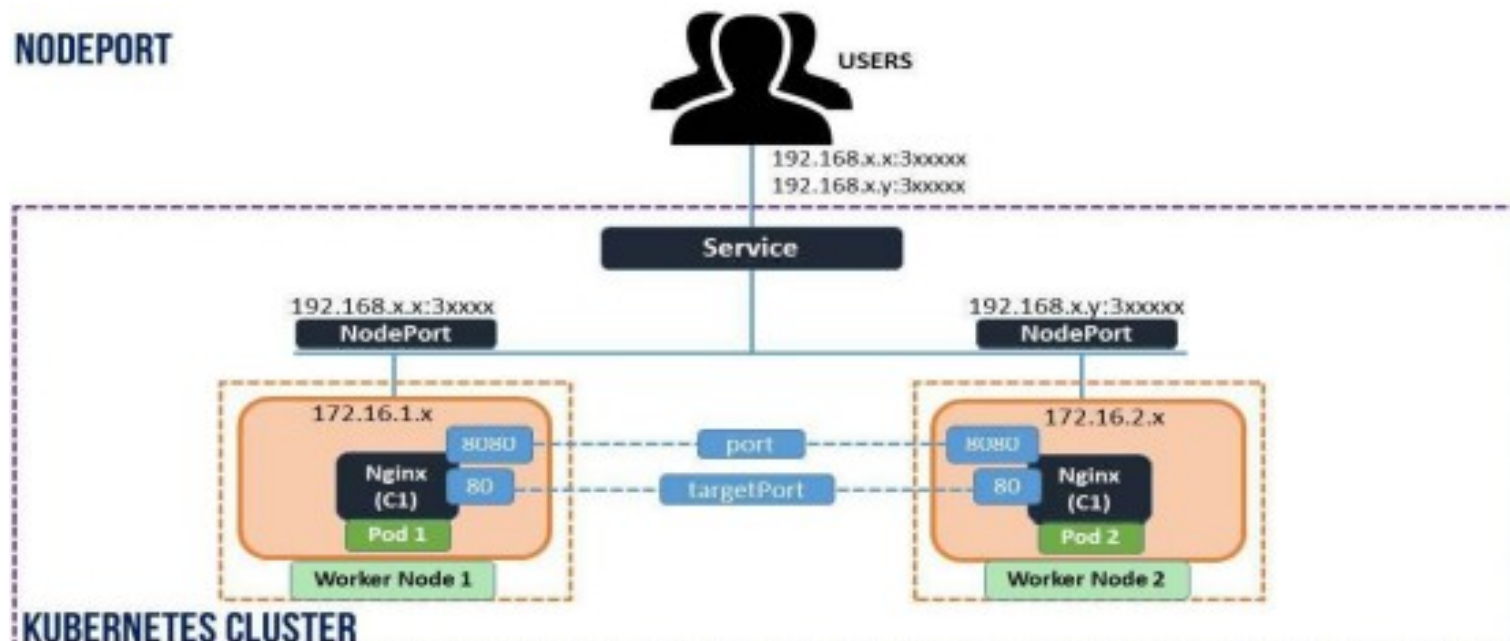
🔍 Chaque requête reçue par le service est envoyée sur l'un des Pods ayant le label spécifié

```
# Lancement du Pod vote
$ kubectl create -f vote-pod.yaml
# Lancement du Service de type ClusterIP
$ kubectl create -f vote-service-clusterIP.yaml
# Lancement d'un Pod utilisé pour le debug
$ kubectl create -f pod-debug.yaml
# Accès au Service vote depuis le Pod debug
$ kubectl exec debug -ti -- sh
/ # apk update && apk add curl
/ # curl http://web-service
(code html de l'interface vote)
...
```

```
$ cat pod-debug.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: debug
spec:
  containers:
  - name: debug
    image: rancher/curl
    command:
      - "sleep"
      - "10000"
```

- Chaque nœud du cluster **ouvre un port statique** et redirige le trafic vers le port indiqué via le contrôleur Kube-proxy.
- Le nœud maître du Kubernetes alloue un port statique à partir d'une plage, et chaque nœud va proxy ce port (le même numéro de port sur chaque nœud) pour notre service. Cela se fait via iptables.

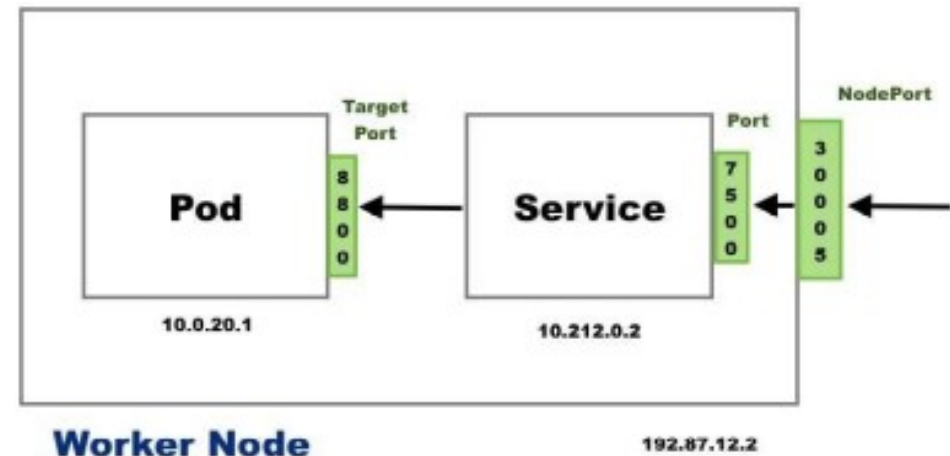


- Spécification d'un Service nommé vote-service de type NodePort.
 - Kubernetes alloue un port à partir d'une plage spécifiée: 30000-32767
- \$ cat vote-service-nodePort.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: web-srv
spec:
  selector:
    app: web
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 31000

```



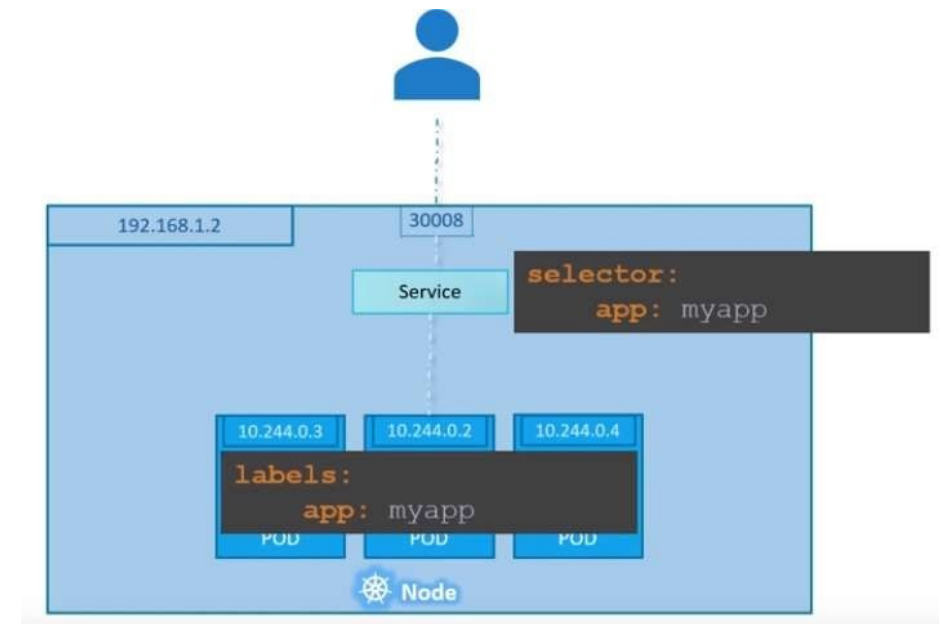
- Il est possible d'exposer le déploiement vote-deploy avec le service vote-service de type NodePort.

Lancement du Service

\$ kubectl create -f vote-service-nodePort.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deploy
spec:
  replicas 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
...
```

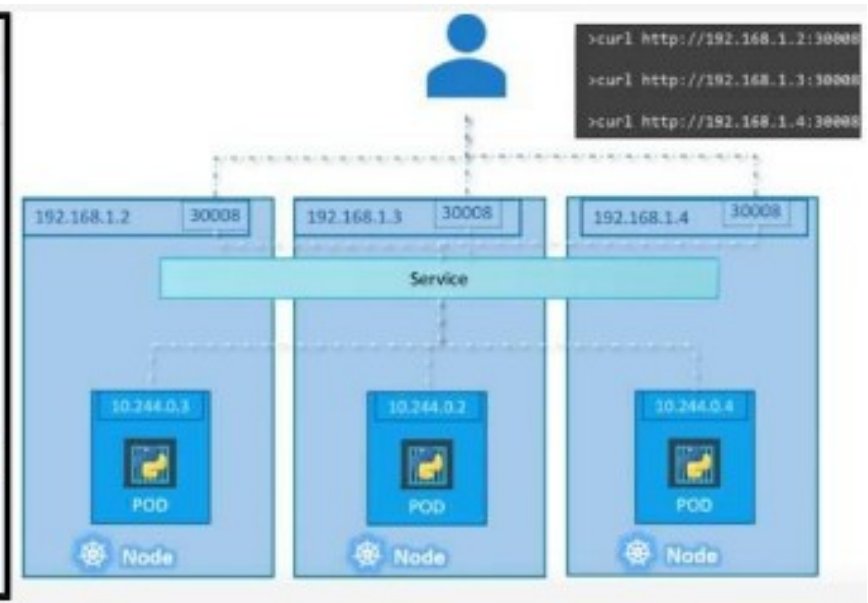
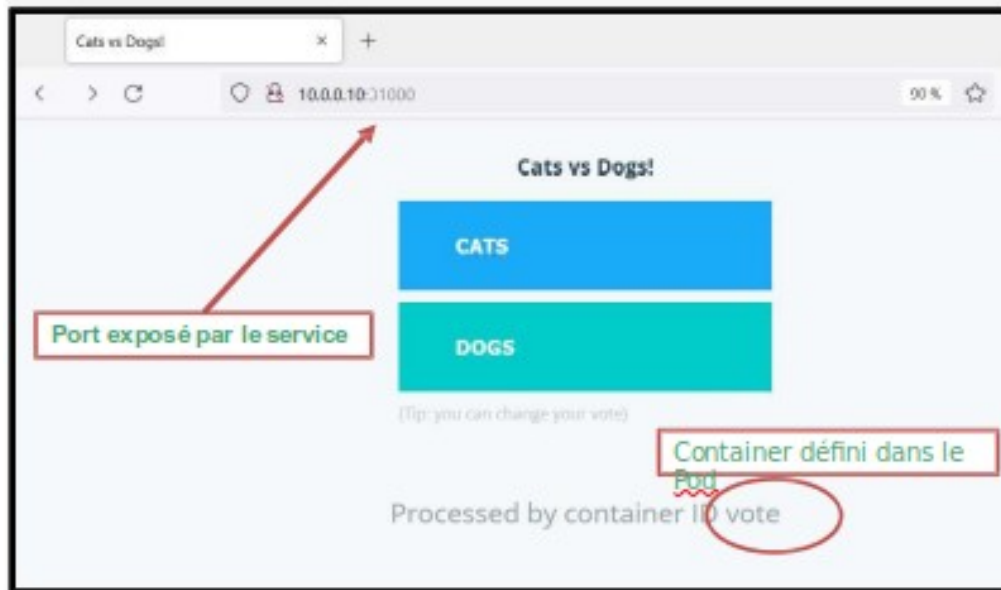
```
apiVersion: v1
kind: Service
metadata:
  name: web-srv
spec:
  selector:
    app: web
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 31000
```



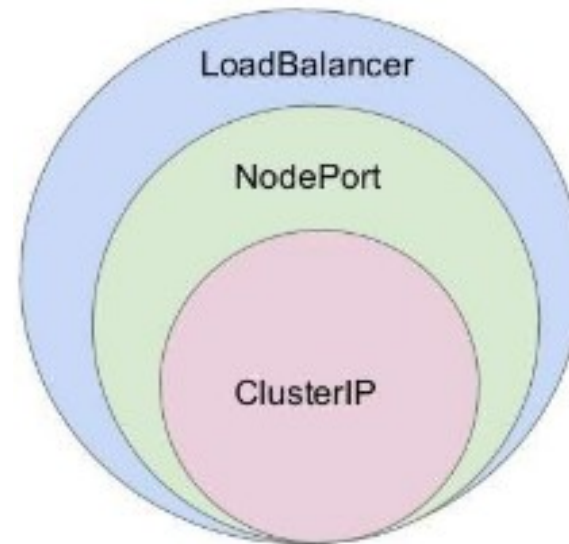
- Il est possible d'exposer le déploiement vote-deploy avec le service vote-service de type NodePort.

Lancement du Service

\$ kubectl create -f vote-service-nodePort.yaml

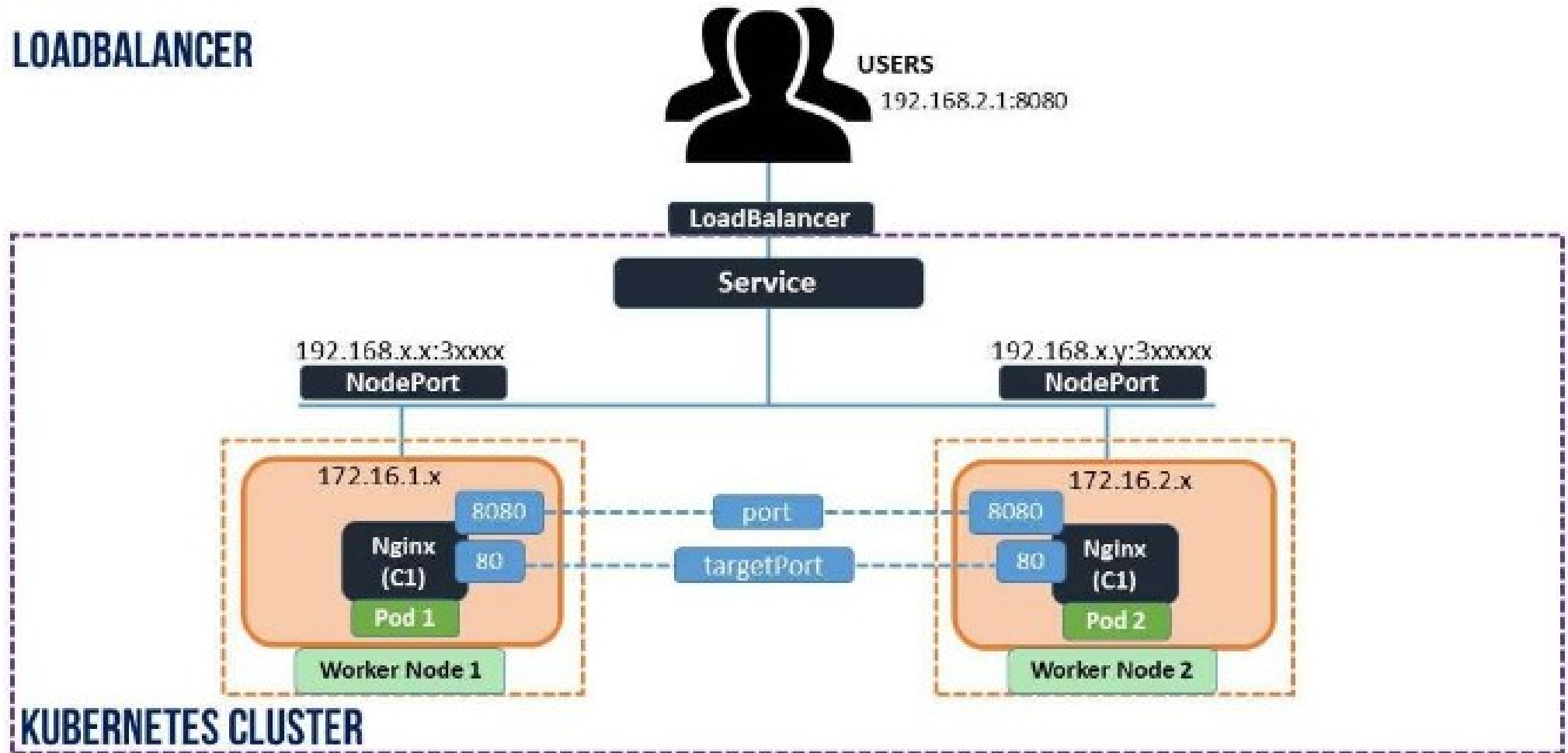


- Expose le service en externe à l'aide de l'équilibreur de charge d'un fournisseur de cloud.
- Ainsi, les services NodePort et ClusterIP sont créés automatiquement et sont acheminés par l'équilibreur de charge externe.
- Exemples de cloud provider:
 - AWS ELB/ALB/NLB
 - GCP LoadBalancer
 - Azure Balancer
 - OpenStack Octavia



<https://kubernetes.io/fr/docs/concepts/services-networking/service/#loadbalancer>

LOADBALANCER



- Il est possible d'exposer le déploiement vote-deploy avec le service vote-service de type LoadBalancer

\$ cat vote-service-LoadBalancer.yaml

```
apiVersion: v1
kind:
metadata:
  name: web-srv
spec:
  selector:
    app: web
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      nodePort: 32000
```

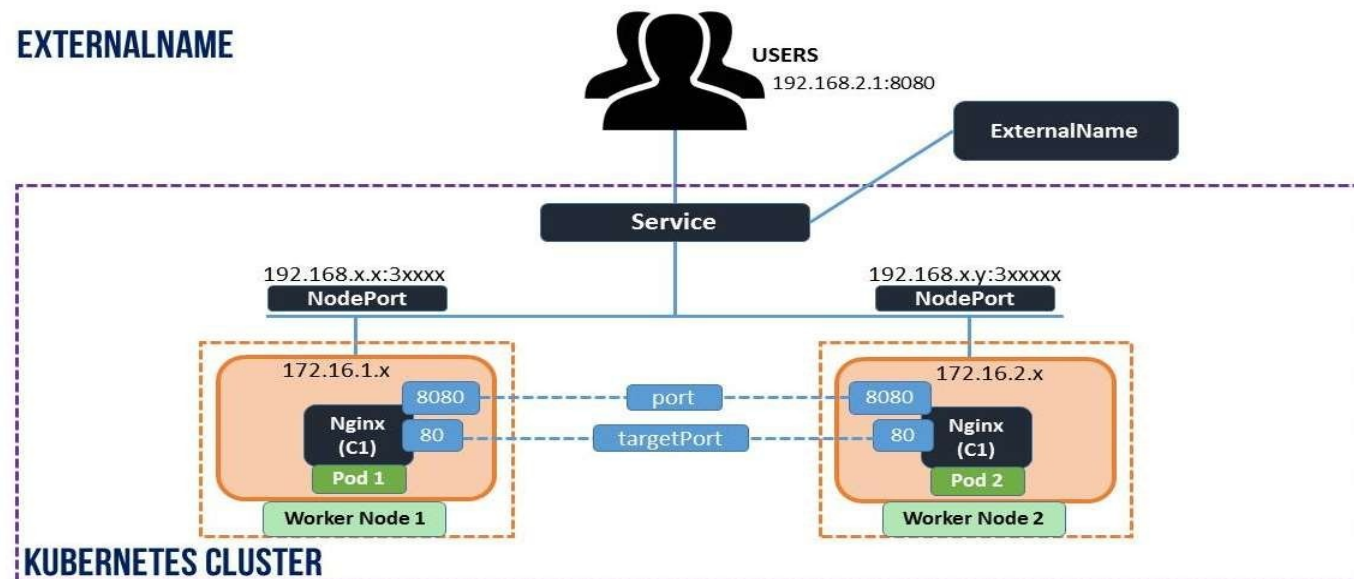
```
# Lancement du Deploy vote
$ kubectl create -f vote-deploment.yaml
# Lancement du Service de type LoadBalancer
$ kubectl create -f vote-service-LoadBalancer.yaml
# Affichage de service de type LoadBalancer
$ kubectl get svc
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP |
|--------------|--------------|-------------|-------------|
| kubernetes | ClusterIP | 10.96.0.1 | <none> |
| vote-service | LoadBalancer | 10.97.46.31 | <pending> |

- Notez le <pending> : cela veut dire que notre service n'a pas d'IP externe. (En gros : il ne sert à rien. Il lui faut une IP pour que nous puissions l'utiliser.)

- Les services de type ExternalName mappent un service à un nom DNS, et non à un sélecteur standard.
- Ce service effectue une simple redirection CNAME (par exemple rediriger le trafic vers le nom de domaine "example.com").

```
apiVersion: v1
kind:
metadata:
  name: web-srv
spec:
  type: ExternalName
  externalName: vote-service.com
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

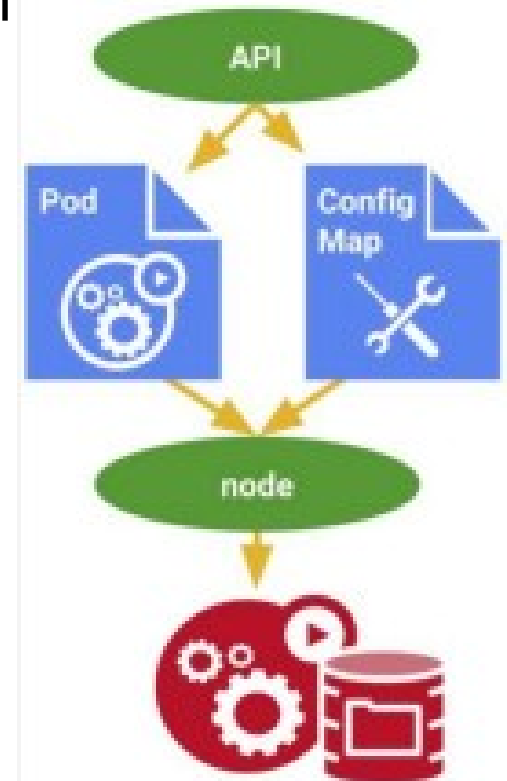


Les services : Commandes de base

```
# Afficher la liste des Services
kubectl get service [En option <SERVICE NAME>]
# Créer un Service depuis un template
kubectl create -f <template.yaml>
# Créer un Service depuis sans template
kubectl expose deployment <DEPLOYMENT NAME>
    --name : nom du service
    --type : type du service
    --protocol : protocole à utiliser (TCP/UDP)
    --port : port utilisé par le service
    --target-port : port utilisé utilisé par le Pod
    --selector='clé=valeur': le sélecteur utilisé par service

# Supprimer un Service
kubectl delete service <SERVICE NAME>
# Appliquer des nouveaux changements à votre Service sans le détruire
kubectl apply -f <template.yaml>
# Modifier et appliquer les changements de votre Service instantanément sans le détruire
kubectl edit service <SERVICE NAME>
# Afficher les détails d'un Service
kubectl describe service <SERVICE NAME>
```

- Objet Kubernetes permettant de stocker séparément les fichiers de configuration
- Un ConfigMap peut sollicité par plusieurs pods
- Les ConfigMaps sont utiles pour stocker et partager des informations de configuration non chiffrées et non sensibles
- Pour utiliser des informations sensibles dans les clusters, vous devez vous servir de secrets
- Les pods peuvent utiliser les ConfigMaps comme :
 - Fichier de configuration (création des volumes, configuration nginx par exemple)
 - Argument en ligne de commande
 - Variable d'environnement




```
$ cat cm-address.yml
```

```
# Création de ConfigMaps avec un fichier de configuration
$ kubectl create -f cm-address.yml
# Création de ConfigMaps en ligne de commande
$ kubectl create cm cm-address2 \
--from-literal="city=Ann Arbor" \
--from-literal=state=Michigan
# Lister les ConfigMaps
$ kubectl get configmap
$ kubectl describe cm/cm-address
```

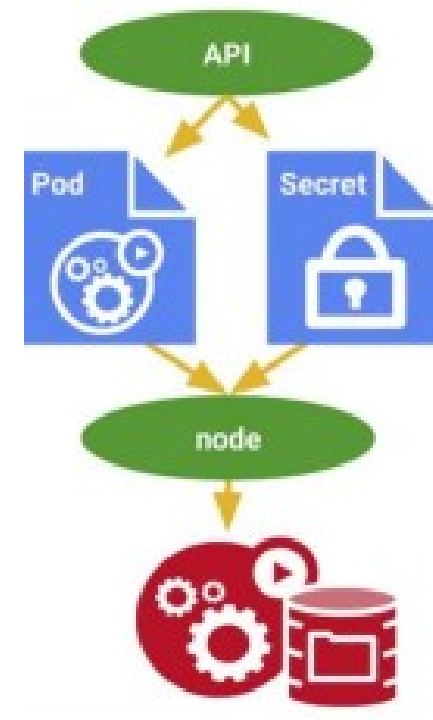
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-address
data:
  city: Paris
  state: Île-de-France
```

```
$ cat job-cm-address.yml
```

```
# Création du job
$ kubectl create -f job-cm-address.yml
# Listet les Pods
$ kubectl get pods
# Afficher la sortie du Pod.
$ kubectl logs job-cm-address-<pod-id>
Hello from Paris!
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cm-cmd-example
spec:
  selector:
    matchLabels :
      app: vote
  template:
    metadata:
      labels :
        app: vote
    spec:
      containers:
        - name: env
          image: nginx
          env:
            - name: CITY
              valueFrom:
                configMapKeyRef:
                  name: cm-address
                  key: city
```

- Objet Kubernetes de type secret utilisé pour stocker des informations sensibles comme les mots de passe, les tokens, les clés SSH...
- Similaire à un ConfigMap, à la seule différence que le contenu des entrées présentes dans le champ data sont encodés en base64.
- Kubernetes fournit plusieurs types intégrés pour certains scénarios d'utilisation courants. Parmi les quels:
 - **Generic (opaque)**: valeurs arbitraire comme dans une ConfigMap. Les valeurs doivent être encodées en base64
 - **tls**: certificat et clé pour utilisation avec un serveur web
 - **docker-registry**: utilisé en tant que **imagePullSecret** par un pod pour pouvoir pull les images d'une registry privée



```
$ cat my-secret.yml
```

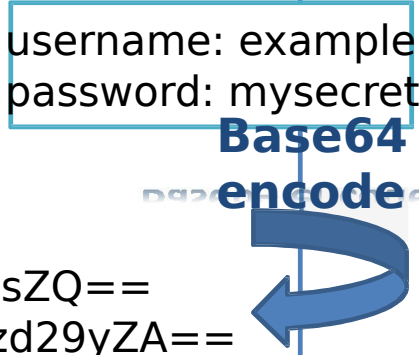
```
# Création d'un secret
$ kubectl create -f my-secret.yml
# Afficher le secret créé
$ kubectl get secret my-secret -o yaml
```

```
apiVersion: v1
data:
  password: bXlwYXNzd29yZA== username:
  ZXhhbXBsZQ==
kind: Secret
metadata:
  creationTimestamp: "2021-12-12T11:03:58Z"
  name: my-secret
  namespace: default
  resourceVersion: "17122"
  uid: 38618fd6-8870-4a5d-86d6-85dca26a9795
  type: Opaque
```

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: ZXhhbXBsZQ==
  password: bXlwYXNzd29yZA==
```

username: example
password: mysecret

Base64 encode



```
# Création d'un secret en ligne de commande
$ kubectl create secret generic my-secret2 \
  --from-literal=username=admin \
  --from-literal=password=password
$ kubectl get secret my-secret -o yaml
```

```
apiVersion: v1 data:
  password: cGFzc3dvcmQ=
  username: YWRtaW4=
kind: Secret
...
```

Secret : variables d'environnement



```
$ cat job-mysecret.yml
```

```
# Création du job
$ kubectl create -f job-mysecret.yml

# Listet les Pods
$ kubectl get pods

# Afficher la sortie du Pod.
$ kubectl logs job-mysecret-<pod-id>
example
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-mysecret
spec:
  template:
    spec:
      containers:
      - name: env
        image: alpine:latest
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: username
```

- Fournir du stockage persistant aux pods
- Sans les volumes persistants, il serait impossible de maintenir des services dont l'état de leurs données cohérent (une base de données par exemple).
- Chaque fois qu'un pod est remplacé, les données acquises pendant le cycle de vie de ce pod sont perdues.
- Kubernetes supporte plusieurs types de Volumes. Parmi lesquels:
 - Basé sur l'hôte (en local): emptyDir et hostPath
 - Volumes cloud : awsElasticBlockStore, gcePersistentDisk et azureDiskVolume
 - Volumes de partage de fichiers: le système de fichiers réseau (nfs)
 - Systèmes de fichiers distribués: cephfs, rbd et glusterfs

Les volumes : volume basé sur l'hôte



- Fonctionnent de la même façon que les volumes Docker pour les volumes hôte : EmptyDir ~= volumes docker et HostPath ~= volumes hôte

```
$ cat job-mysecret.yml
```

```
# Création du pod
$ kubectl create -f vol-redis.yml
# écoute des modifications du pod
$ kubectl get pod redis -watch

NAME READY STATUS      RESTARTS AGE
redis 1/1 Running         0         33s

# Dans un autre terminal, installez un shell sur le
# conteneur en cours d'exécution et créez un fichier :
$ kubectl exec -it redis -- /bin/bash # cd
/data/redis/
root@redis:/data/redis# echo Hello > test-file
root@redis:/data/redis# ls
test-file
```

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

```
# Dans le shell, listez les processus en cours
root@redis:/data/redis# apt-get update
root@redis:/data/redis# apt-get install procps
root@redis:/data/redis# ps aux
```

| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | START | TIME | COMMAND |
|-------|-----|------|------|-------|------|-------|-------|-------|------|-----------------|
| redis | 1 | 0.1 | 0.1 | 50288 | 3068 | ?Ssl | 19:27 | 0:00 | | redis-server *: |
| root | 15 | 0.0 | 0.1 | 18136 | 2100 | pts/0 | Ss | 19:29 | 0:00 | /bin/bash |
| root | 275 | 0.0 | 0.1 | 36636 | 2756 | pts/0 | R+ | 19:32 | 0:00 | ps aux |

```
# Tuez le processus Redis root@redis:/data/redis# kill 1 command terminated with
exit code 137
```

```
# Dans notre terminal d'origine, surveillez les modifications apportées
$ kubectl get pod redis -watch
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------|-------|-----------|----------|-------|
| redis | 1/1 | Running | 0 | 33s |
| redis | 1/1 | Running | 1 | 8m53s |
| redis | 0/1 | Completed | 0 | 8m50s |

Les volumes : volume basé sur l'hôte



```
# S'attacher au shell du conteneur redémarré kubectl exec -it redis -- /bin/bash  
root@redis:/data# ls /data/redis  
test-file
```

Nous pouvons voir que le fichier test-file, que nous avons créé avant le redémarrage du conteneur, existe toujours.

- **volumeMounts** : une liste spécifique au conteneur référençant les volumes du pod par nom, ainsi que leur **mountPath** souhaité.
- **volumes** : une liste d'objets de volume à attacher au pod. Chaque objet de la liste doit avoir son propre nom unique.

apiVersion: v1

kind: Pod

metadata:

name: redis

spec:

containers:

- **name:** redis

image: redis

volumeMounts:

- **name:** redis-storage

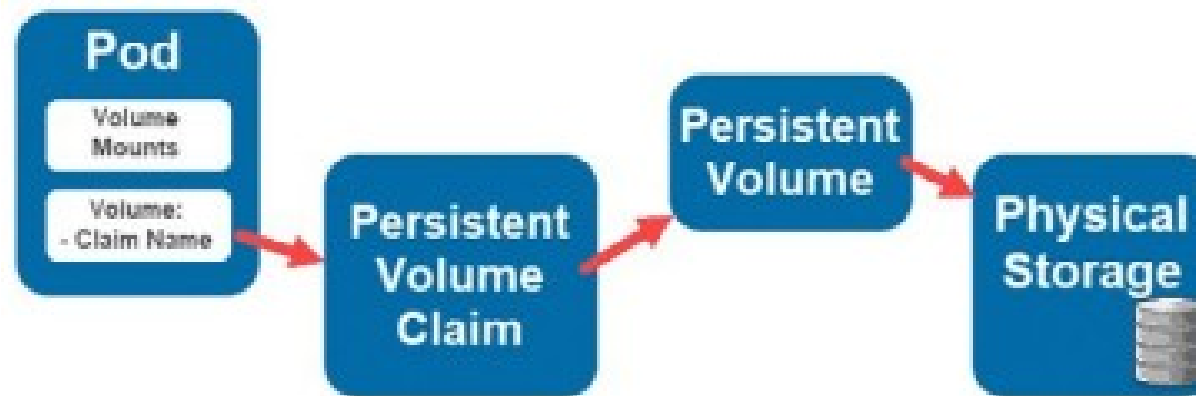
mountPath: /data/redis

volumes:

- **name:** redis-storage

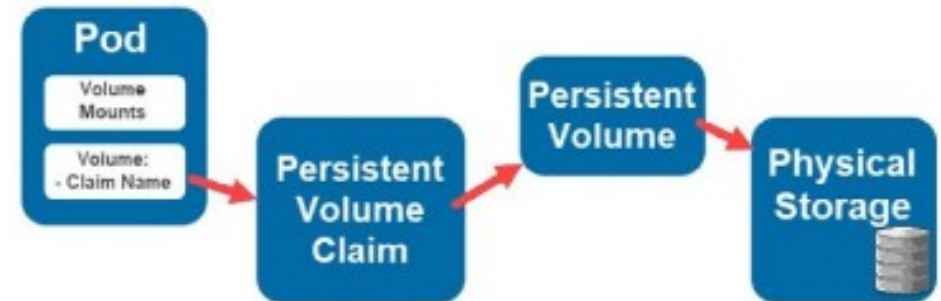
emptyDir: {}

- Un PersistentVolume (PV) représente une ressource de stockage.
- Les PV sont des ressources à l'échelle du cluster liée à un fournisseur de stockage : NFS, GCEPersistentDisk, RBD, etc.
- Généralement un PV est provisionné par un administrateur ou dynamiquement grâce à une StorageClass.
- Leur cycle de vie est géré indépendamment d'un pod.
- NE PEUT PAS être connecté directement à un pod.
- Un PV repose sur un PersistentVolumeClaim

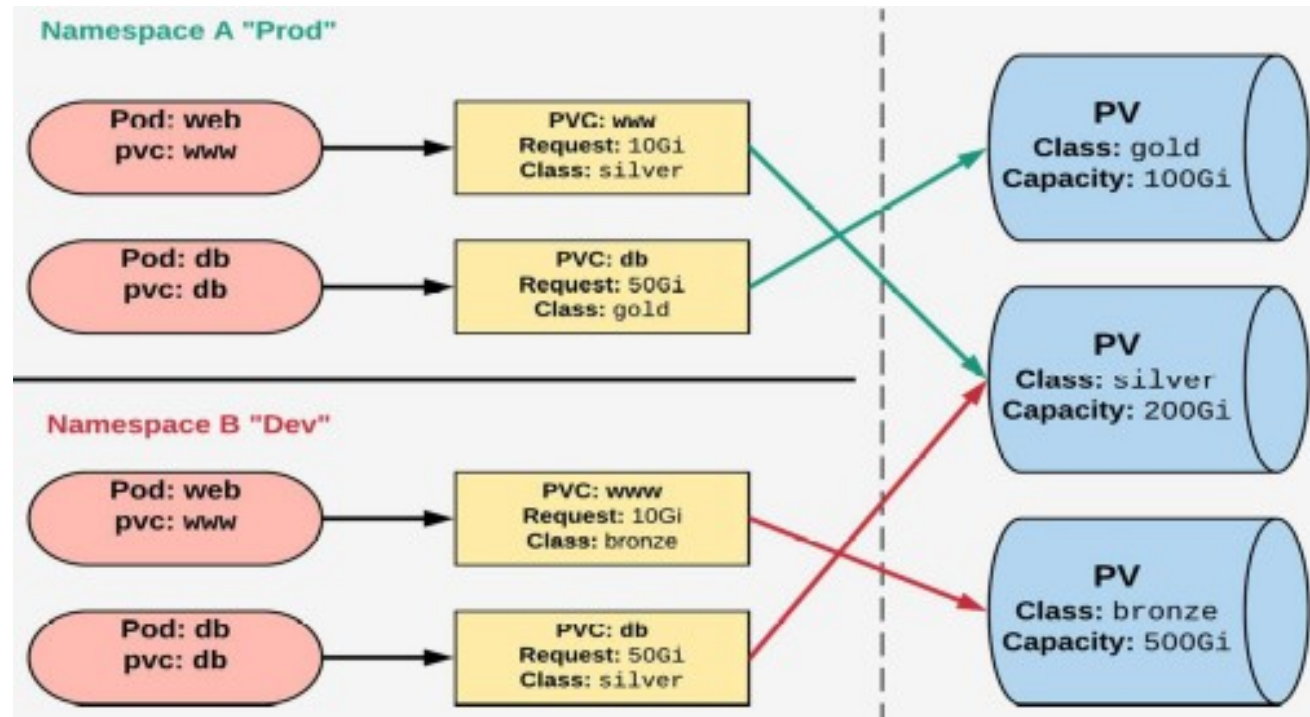


Les volumes : PersistentVolumeClaim

- Un **PersistentVolumeClaim** (PVC) est une **demande de stockage** avec espace de noms (namespace) par un **utilisateur**.
- Satisfait un **ensemble d'exigences** au lieu de mapper directement sur une ressource de stockage.
- Le principal avantage de PVC est de permettre aux développeurs de l'utiliser sans avoir à connaître trop de détails sur l'environnement cloud auquel il se connecte. L'administrateur configure la réclamation dans le PVC, mais le pod lui-même ne nécessite qu'un lien pour y accéder.

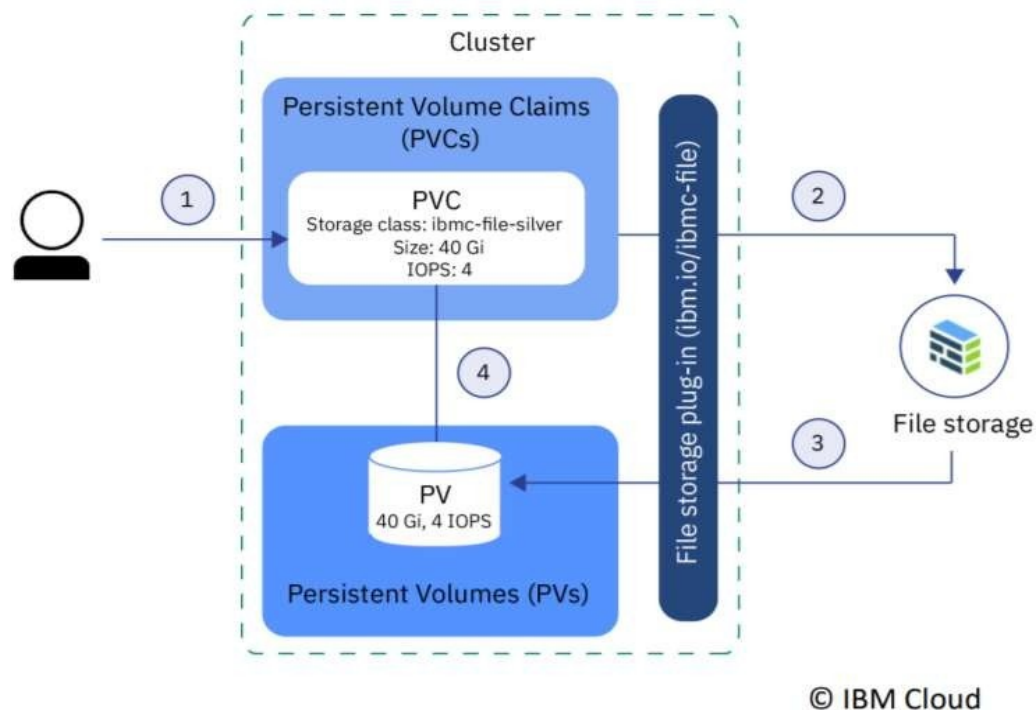


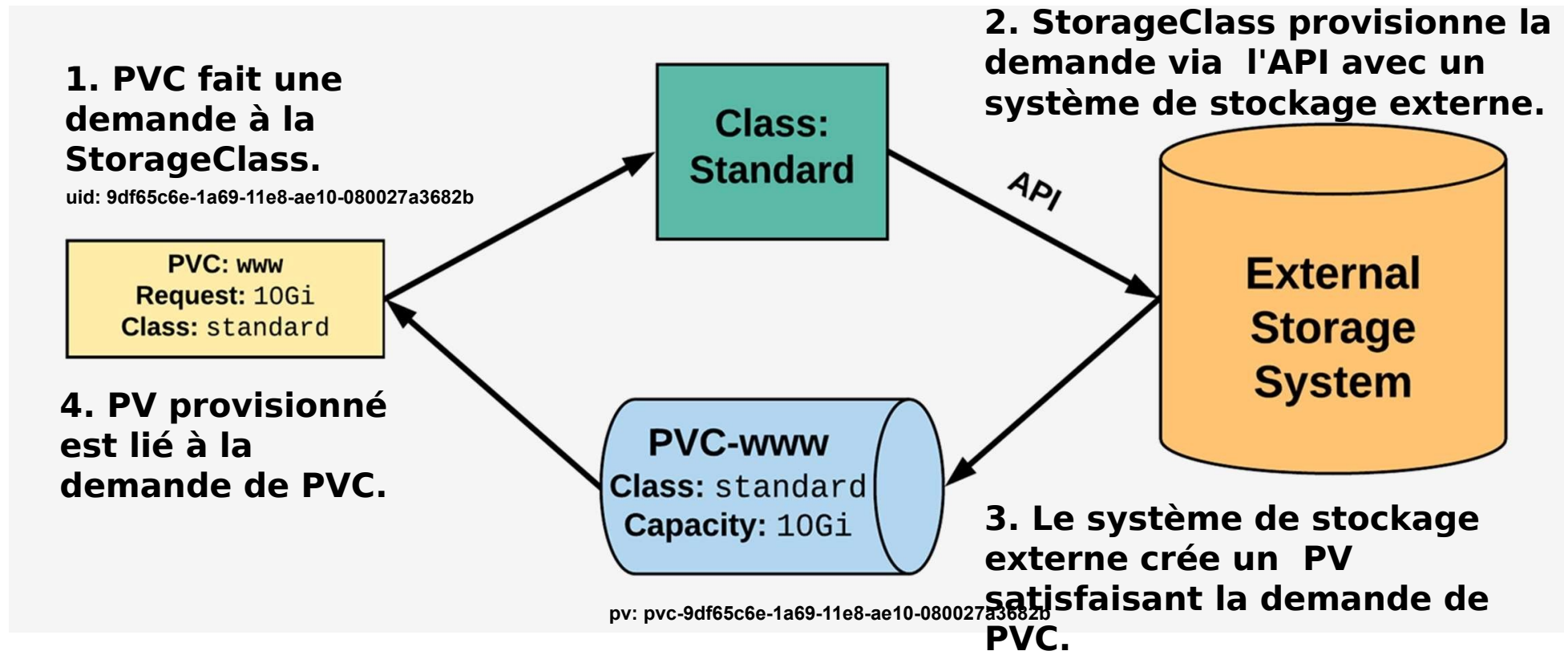
**Cluster
Users**



**Cluster
Admins**

- Les PV et les PVC fonctionnent très bien ensemble, mais nécessitent toujours un approvisionnement manuel. **Et si nous pouvions faire tout cela de manière dynamique ?**
- Les classes de stockage sont une **abstraction** des ressources de stockage externe
- Permet d'avoir un **provisionnement dynamique** du stockage
- Élimine le besoin pour l'administrateur provisionner un PV (généralement les fournisseurs de cloud, mais d'autres fonctionnent comme ceph)







- Les labels
- Pod avec plusieurs containers
- Le ReplicaSet
- Le Deployment
- Le DaemonSet
- Mise en réseau du cluster
- Les Services
- ConfigMap
- Secret
- Les Volumes

Kubernetes en production

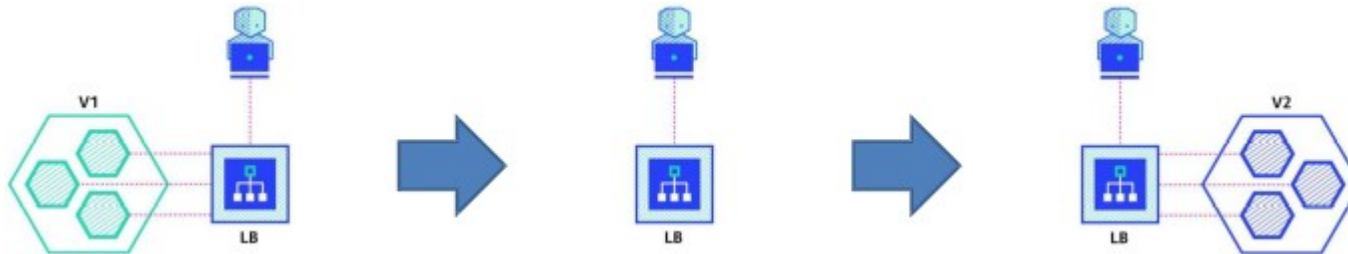
- Les stratégies de déploiement
- Les Namespaces
- Sécurité et contrôle d'accès
- Gestion des ressources
- Autoscaling
- Haute disponibilité et topologie etcd

- Il existe plusieurs stratégies de déploiement dans Kubernetes, les plus connues:
 - Recreate
 - RollingUpdate (Ramped)
 - Blue/Green: étape supplémentaire nécessaire
 - Canary: étape supplémentaire nécessaire

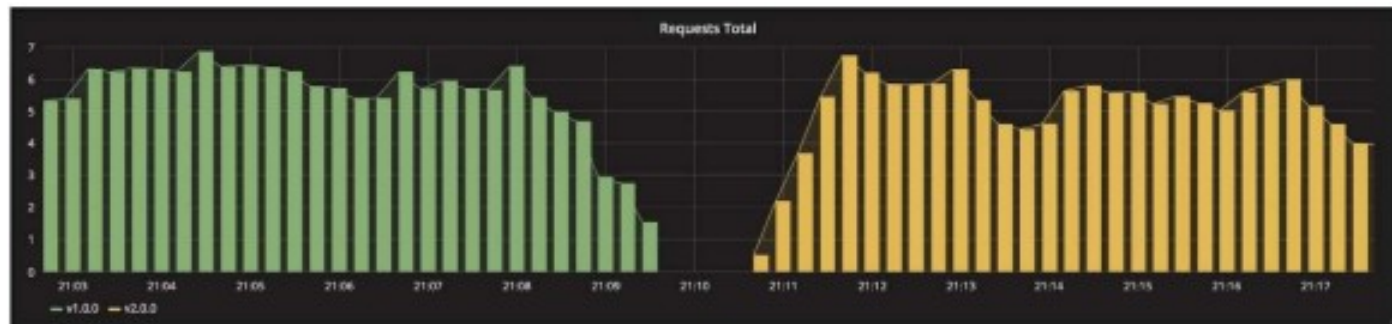
| Strategy | ZERO DOWNTIME | REAL TRAFFIC TESTING | TARGETED USERS | CLOUD COST | ROLLBACK DURATION | NEGATIVE IMPACT ON USER | COMPLEXITY OF SETUP |
|---|---------------|----------------------|----------------|------------|-------------------|-------------------------|---------------------|
| RECREATE version A is terminated then version B is rolled out | ✗ | ✗ | ✗ | ■□□ | ■■■ | ■■■ | □□□ |
| RAMPED version B is slowly rolled out and replacing version A | ✓ | ✗ | ✗ | ■□□ | ■■■ | ■□□ | ■□□ |
| BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B | ✓ | ✗ | ✗ | ■■■ | □□□ | ■■■ | ■■■ |
| CANARY version B is released to a subset of users, then proceed to a full rollout | ✓ | ✓ | ✗ | ■□□ | ■□□ | ■□□ | ■■■ |

Les stratégies de déploiement : Recreate

- Arrêter les Pods avec l'ancienne version en même temps et créer les nouveaux simultanément.
- Avantages: facile à installer
- Inconvénients: impact élevé sur l'utilisateur et un downtime qui dépend à la fois de la durée d'arrêt et de démarrage de l'application



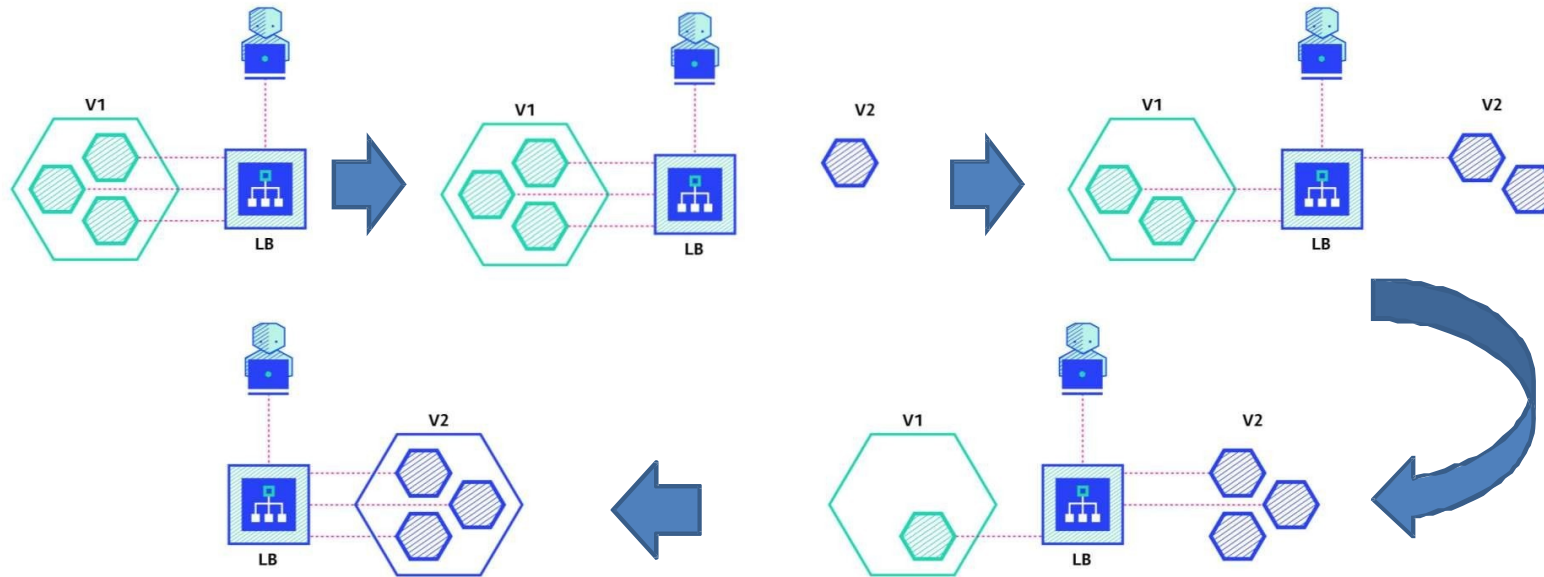
```
[...]
kind: Deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
[...]
```



Service unavailable

Les stratégies de déploiement : RollingUpdate

- Mise à jour continue, arrêt des anciens pods les uns après les autres et création des nouveaux au fur et à mesure (paramétrable). La stratégie par défaut dans Kubernetes.



```
[...]
kind: Deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
[...]
```

maxSurge: combien de pods pouvons-nous ajouter à la fois

maxUnavailable: définit le nombre de pods pouvant être indisponible pendant la mise à jour progressive

Les stratégies de déploiement : RollingUpdate

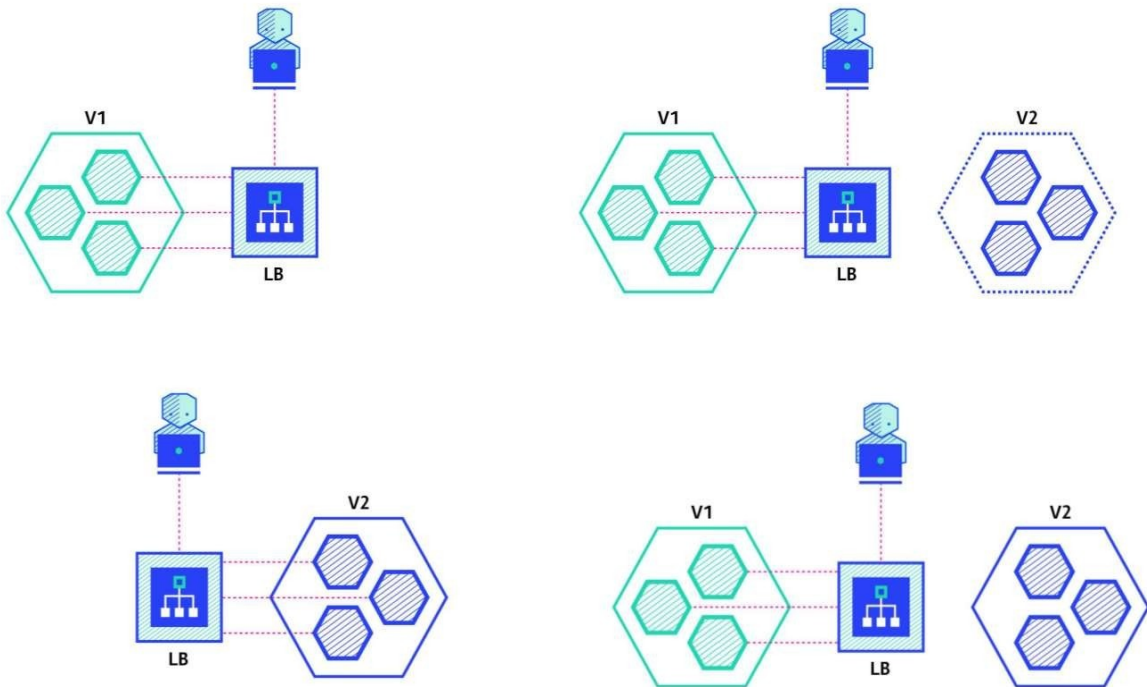


- Avantages:
 - Facile à utiliser
 - Pratique pour les applications avec état (stateful)
 - Rollout/rollback instantané
 - Les inconvénients:
- Inconvénients:
 - Le déploiement/restauration (rollout/rollback) peut prendre du temps
 - Aucun contrôle sur le trafic
 - cher car il nécessite le double des ressources



Les stratégies de déploiement : Blue/Green

- Publier une nouvelle version à côté de l'ancienne puis changer de trafic.



```
[...]
kind: Service
spec:
  selector:
    app: my-app
    version: v1.0.0
[...]
```

```
$ kubectl apply -f ./manifest-v2.yaml
$ kubectl patch service my-app -p \
'{"spec":{"selector":
{"version":"v2.0.0"}}}'
$ kubectl delete -f ./manifest-v1.yaml
```

Les stratégies de déploiement : Canary



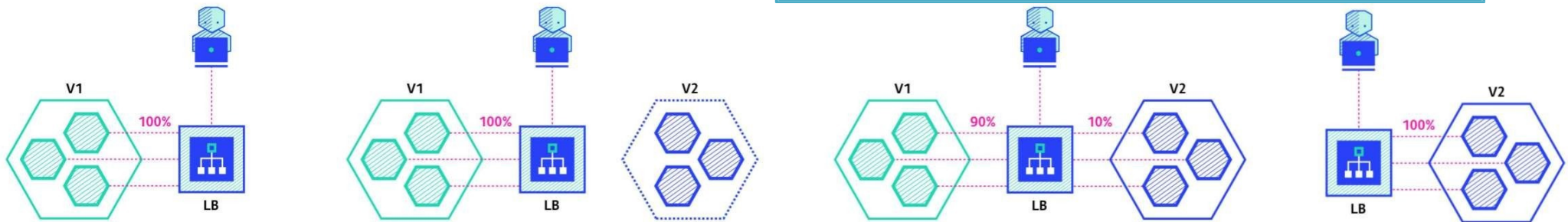
- Diffuser une nouvelle version à un sous-ensemble d'utilisateurs, puis procéder à un déploiement complet

```
[...]
kind: Deployment
metadata:
  name: my-app-v1
spec:
  replicas: 9
  template:
    labels:
      app: my-app
      version: v1.0.0
[...]
```

```
[...]
kind: Deployment
metadata:
  name: my-app-v2
spec:
  replicas: 1
  template:
    labels:
      app: my-app
      version: v2.0.0
[...]
```

```
[...]
kind: Service
metadata:
  name: my-app
spec:
  selector:
    app: my-app
[...]
```

```
$ kubectl apply -f ./manifest-v2.yaml
$ kubectl scale deploy/my-app-v2 --replicas=10
$ kubectl delete -f ./manifest-v1.yaml
```



Les stratégies de déploiement : Canary



- Avantages:
 - version publiée pour un sous-ensemble d'utilisateurs
 - pratique pour le taux d'erreur et la surveillance des performances
 - restauration rapide
- Les inconvénients:
 - déploiement lent
 - des sessions persistantes peuvent être nécessaires
 - un changement de trafic précis nécessiterait un outil supplémentaire comme Istio ou Linkerd



- Un namespace est un moyen de séparer logiquement un cluster physique en plusieurs clusters virtuels
- On peut ainsi faire un découpage par projet et procéder à une gestion des droits plus fine
- Les utilisateurs Kubernetes ne peuvent avoir accès qu'à un seul namespace , avec des quotas définis en terme de nombre de pods , de CPU, de mémoire...
- Un namespace par défaut existe et est utilisé quand on ne renseigne pas le namespace utilisé, il s'appelle « default »



```
# Création du namespaces development (option 1)
$ kubectl create namespace development
namespace "development" created

# Liste des Namespaces
$ kubectl get namespace

# Suppression du namespace
$ kubectl delete namespace/development
namespace "development" deleted

# Création du namespace development (option 2)
$ cat production-ns.yaml

$ kubectl create -f production-ns.yaml
namespace "development" created
```



```
apiVersion: v1
kind: Namespace
metadata:
  name: production
labels:
  name: production
```

```
$ cat nginx-pod-prod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: production
spec:
  containers:
    - name: www
      image: nginx:1.12.2
```

```
# Lancement d'un Pod dans le namespace development
```

```
$ kubectl create -f nginx-pod-prod.yaml
```

```
pod "nginx" created
```

```
# Liste des Pods dans le namespace default
```

```
$ kubectl get po
```

```
No resources found in default namespace.
```

```
# Liste des Pods dans le namespace production
```

```
$ kubectl get po --namespace=production
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------|-------|---------|----------|------|
| nginx | 1/1 | Running | 0 | 2m4s |

```
$ kubectl get po --all-namespaces
```

| NAMESPACE | NAME | READY | STATUS | RESTARTS | AGE |
|-------------|----------------------------|-------|---------|----------|-------|
| kube-system | calico-node-chjst | 1/1 | Running | 1 | 31h |
| kube-system | coredns-74ff55c5b-flgk8 | 1/1 | Running | 1 | 31h |
| kube-system | etcd-master-node | 1/1 | Running | 1 | 31h |
| kube-system | kube-apiserver-master-node | 1/1 | Running | 1 | 31h |
| kube-system | kube-proxy-dtr6t | 1/1 | Running | 1 | 31h |
| kube-system | kube-scheduler-master-node | 1/1 | Running | 1 | 31h |
| production | nginx | 1/1 | Running | 0 | 3m37s |

```
# Création d'un Deployment dans le namespace production
```

```
$ kubectl create deployment www --image nginx:1.12.2 --namespace production --replicas=2
```

```
deployment.apps/www created
```

```
# Liste des Deployments dans le namespace production
```

```
$ kubectl get deploy --namespace production
```

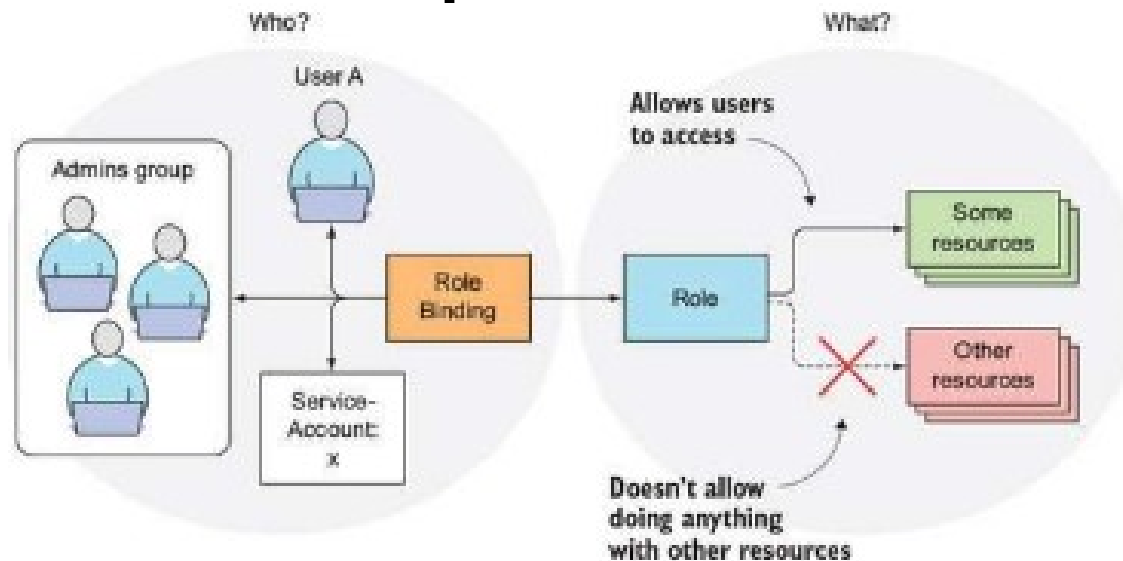
| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|------|-------|------------|-----------|-----|
| www | 2/2 | 2 | 2 | 82s |

```
$ kubectl get po --namespace production
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------|-------|---------|----------|-------|
| nginx | 1/1 | Running | 0 | 22m |
| www-744cd6f665-9fl7l | 1/1 | Running | 0 | 2m57s |
| www-744cd6f665-gxgln | 1/1 | Running | 0 | 2m57s |

```
$ kubectl delete all --all -n production
```

- Dans un environnement réel, il est nécessaire:
 - Avoir de nombreux utilisateurs avec des propriétés différentes qui fournissent le **mécanisme d'authentification souhaité**.
 - Avoir un **contrôle total sur les opérations** que chaque utilisateur ou groupe d'utilisateurs peut effectuer.
 - **Limitez la visibilité de certaines ressources** dans les espaces de noms.
- Kubernetes intègre **un système de permissions fines** sur les **ressources** et les **namespaces**.



Sécurité et contrôle d'accès : Role Based Access Control (RBAC)



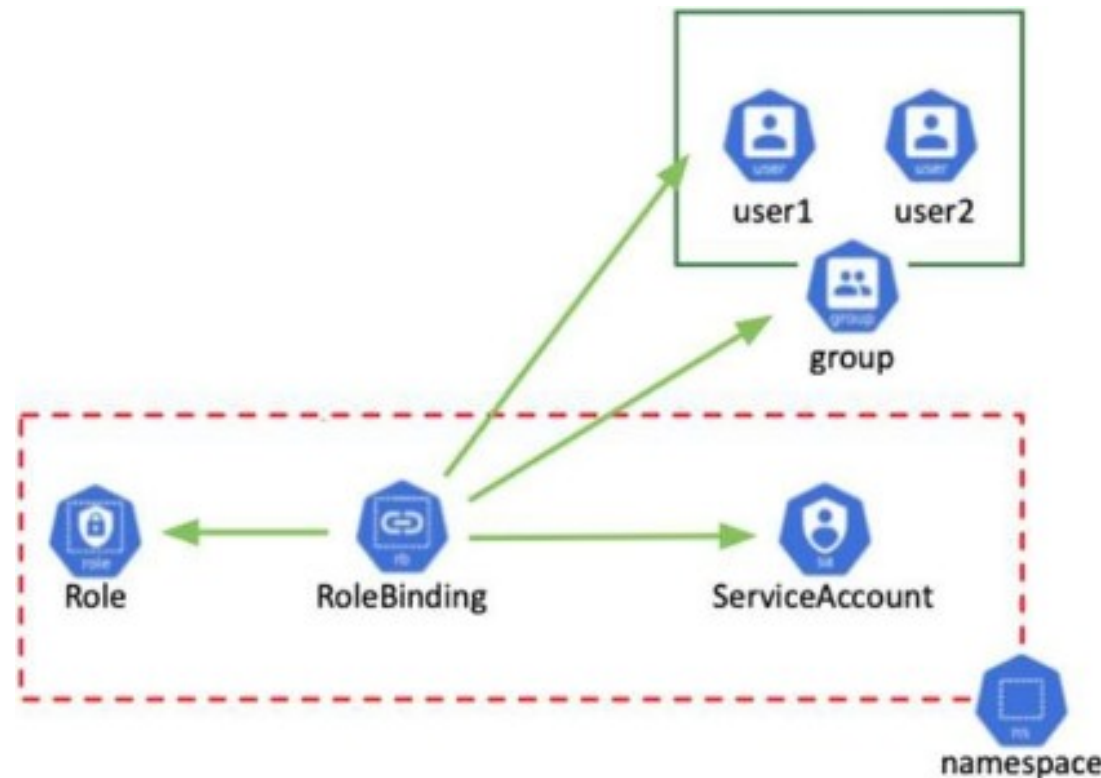
- Les RBAC **contrôlent les accès** dans Kubernetes, elles permettent de donner des droits à la fois à des **utilisateurs humains** mais aussi directement à des **pods**.
- **Trois entités** sont utilisées :
 - Utilisateurs représentés par les **Users** ou les **ServiceAccounts**
 - Ressources représentées par les **Deployments, Pods, Services**, etc.
 - Les différentes opérations possibles : **create, list, get, delete, watch, patch**



Sécurité et contrôle d'accès : Service Accounts, Roles et RoleBindings)



- Classiquement on crée des **Roles** comme admin qui désignent un ensemble de permission dans un **espace de nom**.
- La logique de ce système de permissions est d'associer un **objet** à un **verbe** (get, list, create, delete...)
- On crée ensuite des utilisateurs appelés **ServiceAccounts** dans k8s.
- On lie les **Roles** et **ServiceAccounts** à l'aide d'objets **RoleBindings**.



Sécurité et contrôle d'accès : Les rôles par défaut



- A côté des rôles créés pour les utilisateurs et processus du cluster, il existe des modèles de rôles prédéfinis qui sont affichables avec :

```
$ kubectl get clusterroles
```
- La plupart de ces rôles intégrés sont destinés au kube-system, c'est-à-dire aux processus internes du cluster.
- Cependant quatre rôles génériques existent aussi par défaut :
 - Le rôle **cluster-admin** fournit un accès complet à l'ensemble du cluster.
 - Le rôle **admin** fournit un accès complet à un espace de noms précis.
 - Le rôle **edit** permet à un utilisateur de modifier des choses dans un espace de noms.
 - Le rôle **view** permet l'accès en lecture seule à un espace de noms.

Sécurité et contrôle d'accès : Service Accounts



- Objet Kubernetes permettant d'identifier une application s'exécutant dans un pod.
- Par défaut, un **ServiceAccount** par **namespace**.
- Chaque Namespace possède une ressource ServiceAccount par défaut appelée **default**.

```
$ cat demo-sa.yml
```

```
apiVersion: v1
kind: ServiceAccount
Metadata:
  name: demo-sa
```

```
$ kubectl apply -y demo-sa.yml
```

```
$ kubectl get sa
```

| NAME | SECRETS | SAGE |
|---------|---------|-------|
| default | 1 | 3h48m |
| demo-sa | 1 | 6s |

- L'objet **Role** est un ensemble de règles permettant de définir quelle opération (ou verbe) peut être effectuée et sur quelle ressource
- Le **Role** ne s'applique qu'à un seul **namespace** et les ressources liées à ce **namespace**

```
$ cat demo-roles.yml
$ kubectl apply -f demo-roles.yml
$ kubectl get roles
```

| NAME | CREATED AT |
|-----------|---------------------|
| list-pods | XXXXXX-13T16:21:16Z |

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: list-pods
  namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - list
```

Sécurité et contrôle d'accès : RoleBinding



- L'objet **RoleBinding** va allouer à un **User**, **ServiceAccount** ou un **groupe** les permissions dans l'objet **Role** associé
- Un objet **RoleBinding** doit référencer un **Role** dans le même **namespace**.
- L'objet **roleRef** spécifié dans le **RoleBinding** est celui qui crée le lien

```
$ cat demo-rolebinding.yml
$ kubectl apply -y demo-rolebinding.yml
$ kubectl get rolebinding
```

| NAME | ROLE | AGE |
|-------------------|----------------|------|
| list-pods_demo-sa | Role/list-pods | 2m43 |

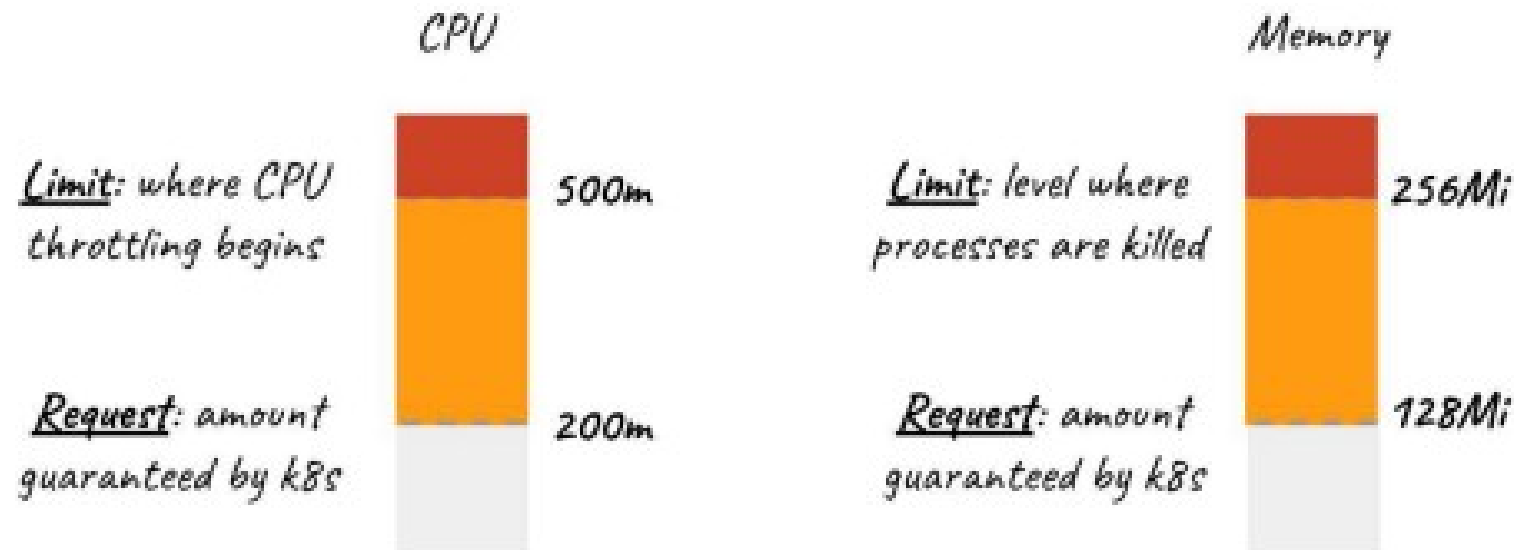
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: list-pods_demo-sa
  namespace: default
roleRef:
  kind: Role
  name: list-pods
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: demo-sa
  namespace: default
```

- La Scalabilité horizontale que permet Kubernetes est l'une de ses fonctionnalités les plus intéressantes mais scaler horizontalement implique plus de pods, ce qui veut dire aussi plus de processus et donc la demande et la consommation de plus de ressources
- Les ressources ne sont pas illimitées (Mémoire, Espace disque, Cpu..), si le nœud est surchargé, le Kernel n'aurait plus le choix que de tuer certains de ces processus pour libérer de l'espace (OOM Kill)
- Kubernetes nous permet d'avoir un niveau de **contrôle sur la consommation** de ressources, ainsi que de **prioriser certains pods par rapport à d'autres** afin d'éviter, en cas de surcharge, que le Kernel tue des pods qui peuvent être critiques, ceci se fait en définissant trois niveaux de QoS à l'aide des **limits** et **requests** ainsi que des **quotas** de ressources.0

Gestion des ressources : request et limits des ressources



- Permettent de gérer l'allocation de ressources au sein d'un cluster
- Par défaut, un pod/container sans request/limits est en **best effort**
- **Request**: allocation minimum garantie (réservation)
- **Limit**: allocation maximum (limite)
- Se base sur le CPU et la RAM



Gestion des ressources - Pods Ressources: CPU



- 1 CPU est globalement équivalent à **un cœur**
- L'allocation se fait par fraction de CPU:
 - 1 : 1 vCPU entier
 - 100m : 0.1 vCPU (un dixième de CPU)
 - 0.5 : 1/2 vCPU
- Lorsqu'un conteneur atteint la limite CPU, celui ci est **throttle**

- **Request:** réservation $\frac{1}{2}$ vcpu
- **Limit:** limite est de 1 vCPU entier

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "128Mi"
        cpu: 0.5
      limits:
        memory: "256Mi"
        cpu: 1
```

Gestion des ressources - Pods Ressources: RAM



- L'allocation se fait en unité de RAM:
 - M : en base 10
 - Mi : en base 2
- Lorsqu'un conteneur atteint la limite RAM, celui ci est **OOMKilled**

- **Request:** réservation est de 128Mi de RAM
- **Limit:** limite est de 256Mi de RAM

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "128Mi"
        cpu: 0.5
      limits:
        memory: "256Mi"
        cpu: 1
```

Gestion des ressources : Exemple



```
$ kubectl describe po web-app
```

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: nginx
    resources:
```

```
  requests:
    memory: "128Mi"
    cpu: 0.5
```

```
  limits:
    memory: "256Mi"
    cpu: 1
```

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl describe po web-app
Name:          web-app
Namespace:     default
Priority:       0
Node:          minikube/192.168.99.102
Start Time:    Thu, 15 Oct 2020 01:00:40 +0530
Labels:        <none>
Annotations:   <none>
Status:        Running
IP:            172.17.0.5
IPs:
  IP: 172.17.0.5
Containers:
  app:
    Container ID:  docker://187b5fb64cb6531638798075548f7e5517b50a08df0eb07a6d3ed229834b8e1f
    Image:         nginx
    Image ID:      docker-pullable://nginx@sha256:ed7f815851b5299f616220a63edac69a4cc200e7f536a56e421988da82e
    Port:          <none>
    Host Port:     <none>
    State:         Running
      Started:     Thu, 15 Oct 2020 01:00:47 +0530
    Ready:         True
    Restart Count: 0
    Limits:
      cpu:          1
      memory:       256Mi
    Requests:
      cpu:          500m
      memory:       128Mi
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7k7zj (ro)
Conditions:
  Type            Status
  Initialized      True
  Ready           True
```


- Permet de **scaler automatiquement** le nombre de pods d'un deployment
- **Métriques classiques (CPU/RAM):** En fonction d'un % de la request CPU/RAM
- **Métriques custom (personnalisées)**

```
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
- type: Resource
  resource:
    name: memory
    target:
      type: AverageValue
      averageValue: 500Mi
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
        - name: php-apache
          image: k8s.gcr.io/hpa-example
          ports:
            - containerPort: 80
          resources:
            limits:
              cpu: 500m
            requests:
              cpu: 200m
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
labels:
  run: php-apache
spec:
  ports:
    - port: 80
  selector:
    run: php-apache
```

```
$ kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
$ kubectl get deployment php-apache
$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
$ kubectl get hpa
# augmenter la charge dans un nouveau terminal
$ kubectl run -i --tty load-generator --rm --image=busybox \
--restart=Never -- /bin/sh -c \
"while sleep 0.01; do wget -q -O- http://php-apache; done"
$ kubectl get hpa php-apache --watch
```

| NAME | REFERENCE | TARGET | MINPODS | MAXPODS | REPLICAS | AGE |
|------------|-----------------------------|------------|---------|---------|----------|-----|
| php-apache | Deployment/php-apache/scale | 305% / 50% | 1 | 10 | 7 | 3m |

```
# afficher les détails de hpa
$ kubectl describe hpa php-apache

# supprimer le hpa php-apache
$ kubectl delete hpa php-apache

$ kubectl get hpa php-apache -o yaml > hpa-v2.yaml

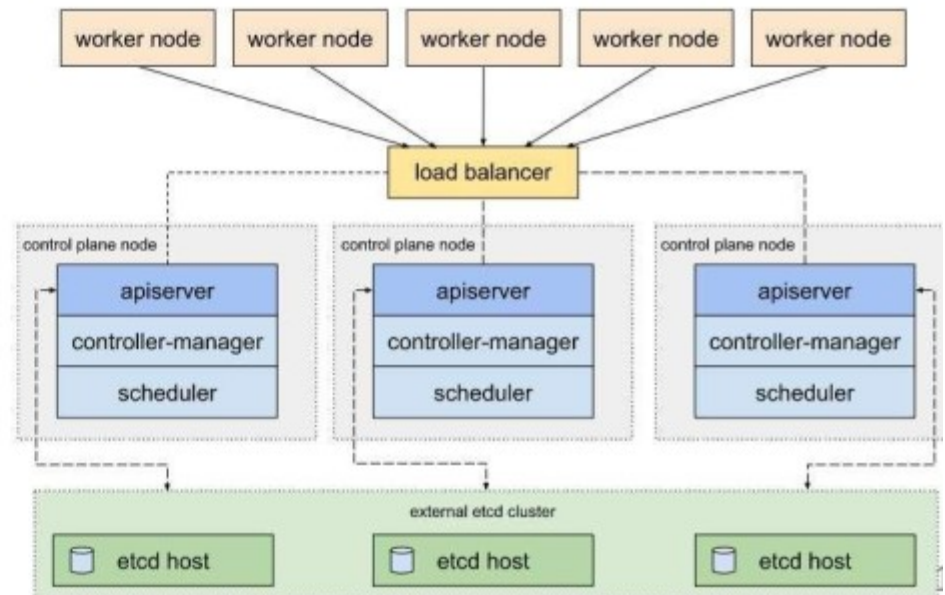
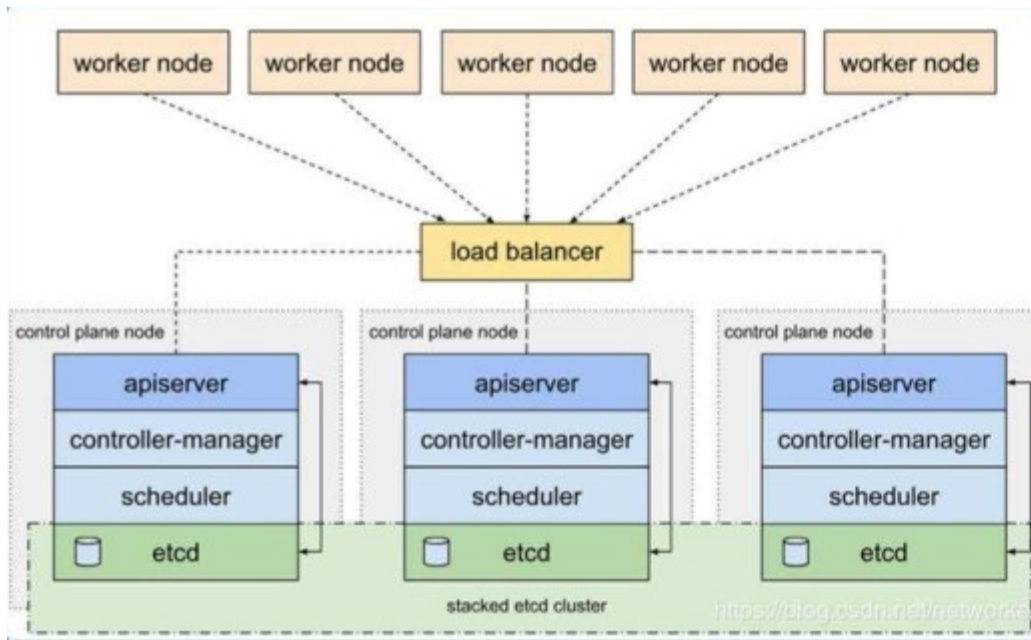
#changer le pourcentage CPU à 20 %
$ nano hpa-v2.yaml

$ kubectl apply -f hpa-v2.yaml
```

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  status:
    observedGeneration: 1
    lastScaleTime: <some-time>
    currentReplicas: 1
    desiredReplicas: 1
    currentMetrics:
    - type: Resource
      resource:
        name: cpu
        current:
          averageUtilization: 0
          averageValue: 0
```

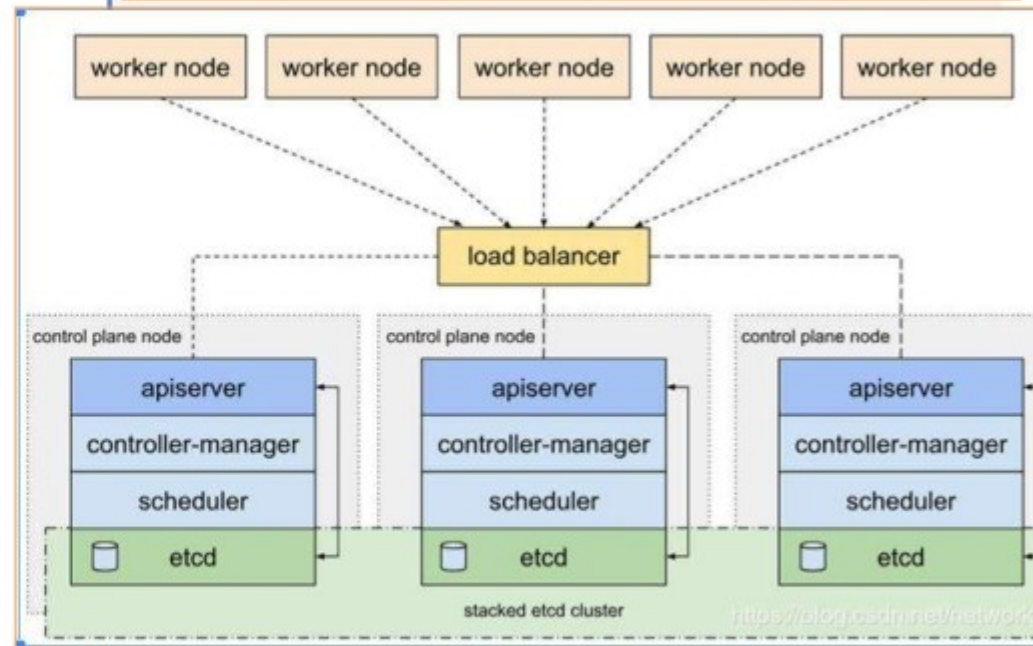
Haute disponibilité et mode maintenance

- Vous pouvez configurer un cluster k8s en haute disponibilité:
 - Avec **des nœuds du control plane empilés**, les nœuds etcd étant co-localisés avec des nœuds du control plane
 - Avec **des nœuds etcd externes**, où etcd s'exécute sur des nœuds distincts du control plane



- Un cluster HA empilé est une topologie réseau où le cluster de stockage de données distribuées est fourni par **etcd** et est superposé au cluster formé par les nœuds gérés par kubeadm qui exécute **les composants du control plane**.
- Chaque nœud du control plane exécute une instance de kube-apiserver, kube-scheduler et kube-controller-manager. Le kube-apiserver est exposé aux nœuds à l'aide d'un **loadbalancer**.
- Chaque nœud du control plane crée un membre etcd local et ce membre etcd communique uniquement avec le kube-apiserver de ce nœud. Il en va de même pour le kube-controller-manager local et les instances de kube-scheduler.

Cette topologie couple les control planes et les membres etcd sur les mêmes nœuds.



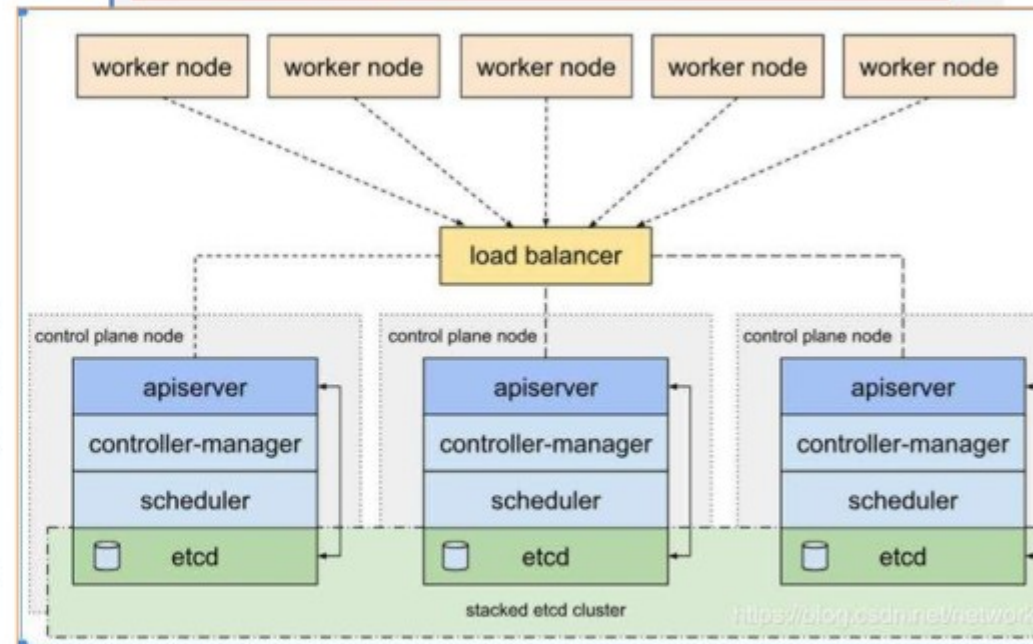
- **Avantages:**

- C'est plus simple à mettre en place qu'un cluster avec des nœuds etcd externes
- Plus simple à gérer pour la **réplication**.

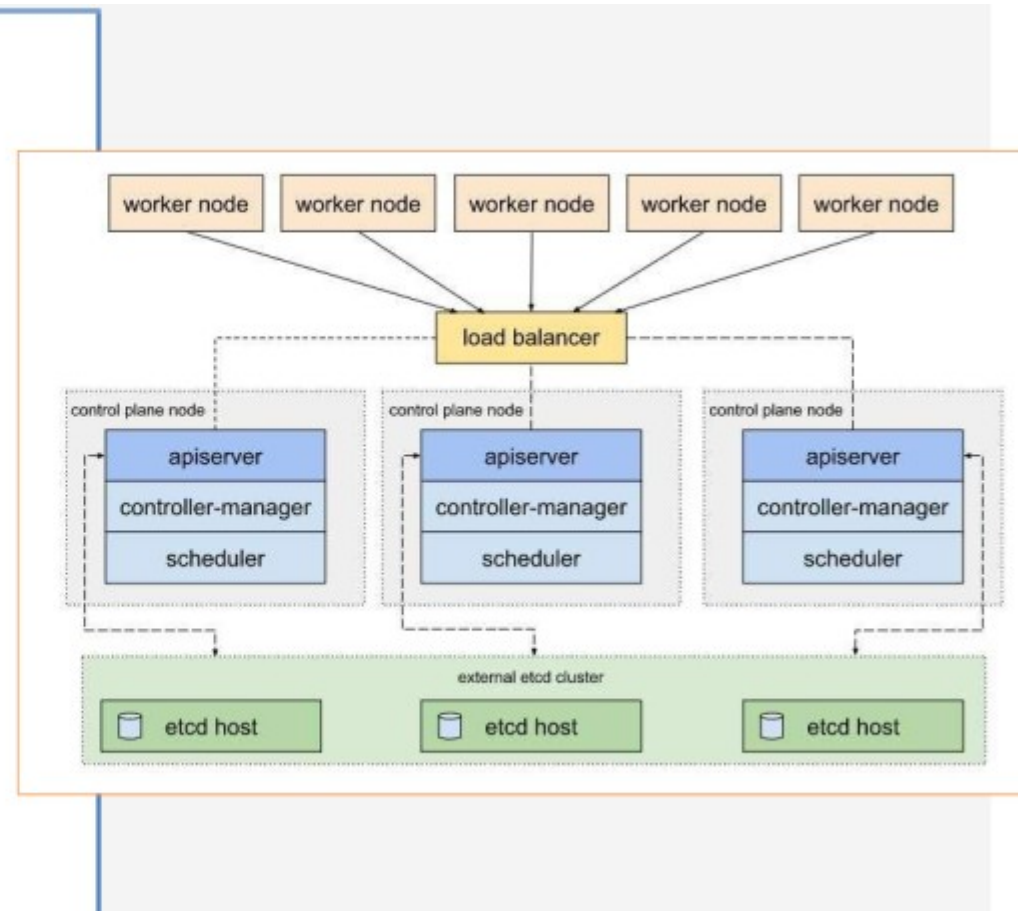
- **Inconvénients:**

- Risque d'échec du **couplage**: si un nœud tombe en panne, un membre etcd et une instance du control plane sont perdus et la redondance est compromise. Vous pouvez atténuer ce risque en ajoutant plus de nœuds au control plane.
- Par conséquent, vous devez exécuter au moins trois nœuds de control plane empilés pour un cluster en haute disponibilité.

C'est la topologie par défaut dans kubeadm.



- C'est une topologie réseau où le cluster de stockage de données distribuées fourni par **etcd** est **externe au cluster** formé par les nœuds qui exécutent les composants du control plane.
- Comme la topologie etcd empilée, chaque nœud du control plane d'une topologie etcd externe exécute une instance de kube-apiserver, kube-scheduler et kube-controller-manager. Et le kube-apiserver est exposé aux nœuds workers à l'aide d'un **load-balancer**.
- Cependant, les membres etcd s'exécutent sur des hôtes distincts et chaque hôte etcd communique avec le kube-apiserver de chaque nœud du control plane.

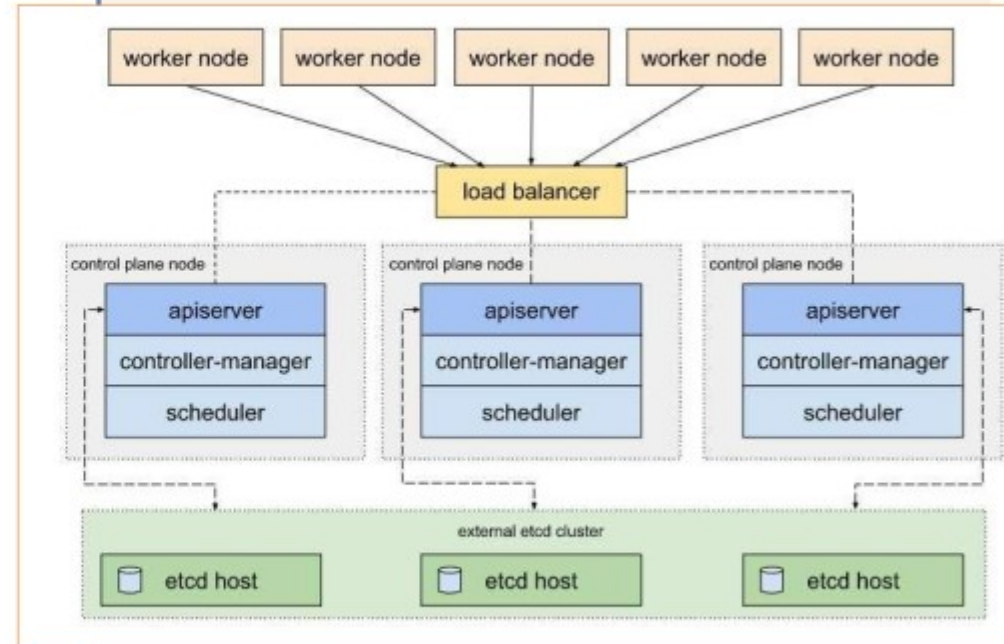


- **Avantages:**

- Cette topologie dissocie le **control plane** et le membre **etcd**: Elle fournit donc une configuration HA où perdre une instance de control plane ou un membre etcd a moins d'impact et n'affecte pas la redondance du cluster autant que la topologie HA empilée.

- **Inconvénients:**

- Cependant, cette topologie requiert le double du nombre d'hôtes de la topologie HA intégrée.
- Un minimum de trois machines pour les nœuds du control plane et de trois machines pour les nœuds etcd est requis pour un cluster HA avec cette topologie.



- Les stratégies de déploiement
- Les Namespaces
- Sécurité et contrôle d'accès
- Gestion des ressources
- Autoscaling
- Haute disponibilité et topologie etcd