



# Orchestration de micro-services Kubernetes

Ali Koudri

[ali.koudri@gmail.com](mailto:ali.koudri@gmail.com)

A cluster of decorative circles in shades of pink, purple, and orange in the top-left corner.

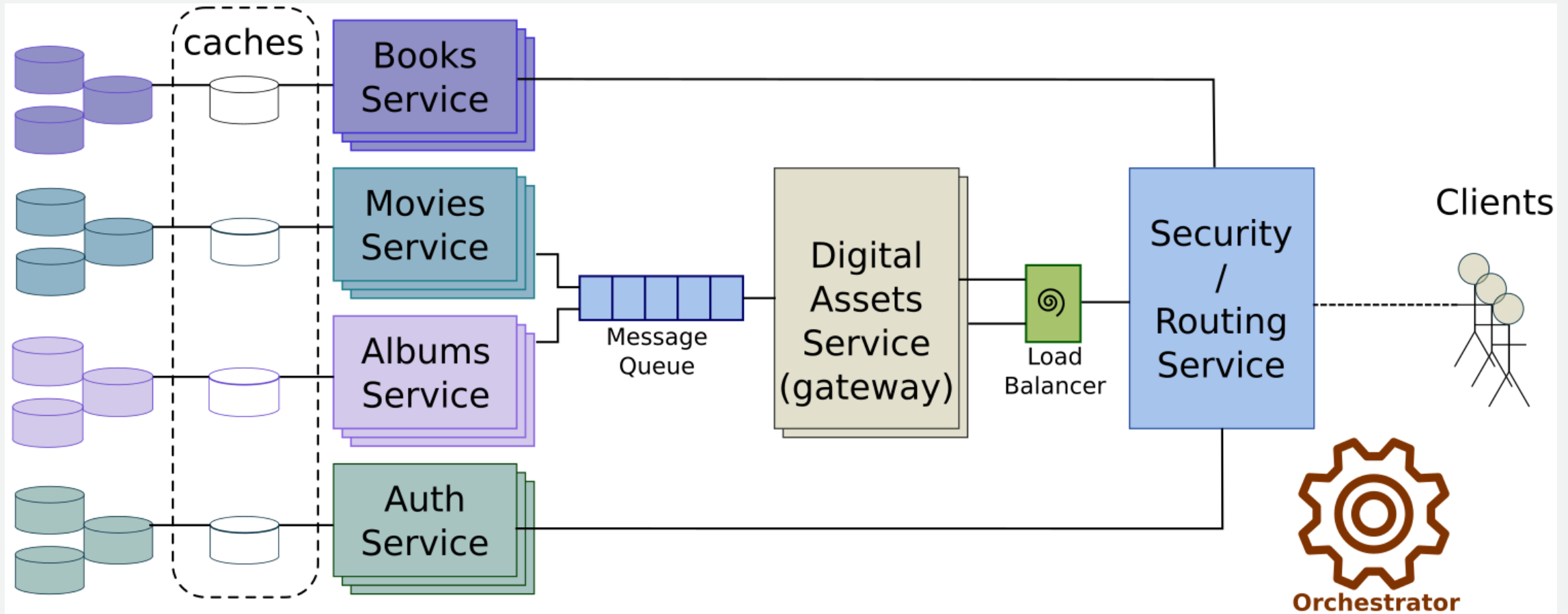
# Agenda

- Présentation
  - Architecture
  - Cycle de vie
  - Contrôleurs
  - Services
  - Gestion des ressources
  - Gestion des volumes
  - Sécurité
  - Monitoring
  - Kustomize
  - Helm
- 
- A collection of decorative circles in shades of blue, purple, pink, and teal in the bottom-right corner.

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner includes a small teal circle, a medium pink circle, a tiny dark purple dot, a medium purple circle, a large pink circle, and a large teal circle.

# Présentation

# Architecture en micro-services



- **Question:** qu'attend-on d'un service d'orchestration ?

# Motivations

- **Orchestration multi-hôtes**

Docker gère les conteneurs sur un seul hôte ou quelques hôtes, mais il ne fournit pas nativement une orchestration avancée sur un cluster complet. Kubernetes permet de gérer, déployer et faire évoluer des applications sur un cluster de nombreux nœuds, assurant ainsi la haute disponibilité et la répartition de charge automatique.

- **Mise à l'échelle automatique**

Docker ne propose pas de mécanisme natif pour augmenter ou réduire dynamiquement le nombre de conteneurs selon la charge. Kubernetes automatise la mise à l'échelle horizontale et verticale des applications en fonction de la demande.

- **Haute disponibilité et tolérance aux pannes**

Docker seul ne redémarre pas automatiquement les conteneurs ou ne les déplace pas en cas de panne d'un nœud. Kubernetes surveille l'état des applications et des nœuds, redémarre ou remplace automatiquement les conteneurs défectueux, et garantit la disponibilité du service.

# Motivations

- **Équilibrage de charge intégré**

Docker ne gère pas nativement la répartition du trafic réseau entre plusieurs conteneurs. Kubernetes fournit un équilibrage de charge automatique pour répartir le trafic entre les instances de service.

- **Déploiements et mises à jour automatisés**

Docker ne propose pas de stratégies sophistiquées pour les mises à jour continues ou les rollbacks. Kubernetes permet des déploiements progressifs, des mises à jour sans interruption et des retours arrière automatisés en cas d'échec.

- **Gestion du stockage persistant**

Docker ne gère pas nativement la persistance des données lors du redémarrage ou du déplacement des conteneurs. Kubernetes orchestre le stockage persistant et l'attache dynamiquement aux conteneurs où qu'ils soient exécutés.

- **Découverte de services**

Docker nécessite une configuration manuelle pour que les conteneurs se découvrent entre eux. Kubernetes fournit des mécanismes de découverte de services intégrés, facilitant la communication entre composants applicatifs.



# Présentation générale

- Kubernetes est né dans les laboratoires de Google avec l'objectif d'orchestrer des conteneurs sur des clusters de machines physiques ou virtuelle
  - Projet Borg
- Kubernetes est depuis devenu un projet Open-Source
  - Soutenue par une très large communauté
  - Utilisée par de nombreuses entreprises, dont de très grosses (amadeus, ebay, bla bla car, Yahoo!, ...)
  - Aujourd'hui maintenue par la Cloud Native Computing Foundation
- Kubernetes, également connu sous le nom de K8s, a été conçue pour automatiser le déploiement, la mise à l'échelle et le fonctionnement des conteneurs d'applications.
- Kubernetes fonctionne avec une série d'outils de conteneurs, dont Docker.

# Fonctionnalités

- **Gestion des nœuds et des pods**
  - Allocation dynamique des pods sur les nœuds
  - Gestion jusqu'à 5000 nœuds et 150 pods par nœud
  - Enregistrement dynamique de nouveaux nœuds
  - Ordonnancement des conteneurs sur différentes hôtes - kube-scheduler
- **Vérification de la qualité de service en temps réel et reconfiguration si besoin**
  - Utilisation des ressources
  - Disponibilité et temps de réponse
- **Persistance des données (volumes)**
  - Gestion des données sensibles (secret management)
- **Maintenance**
  - Mise à jour de pods (et roll-back en cas de problèmes)
  - Logging et monitoring
- **Kubernetes repose sur une architecture extensible par un mécanisme de plugins**
  - Pilotes réseau
  - Découverte de service
  - Visualisation



# Avantages

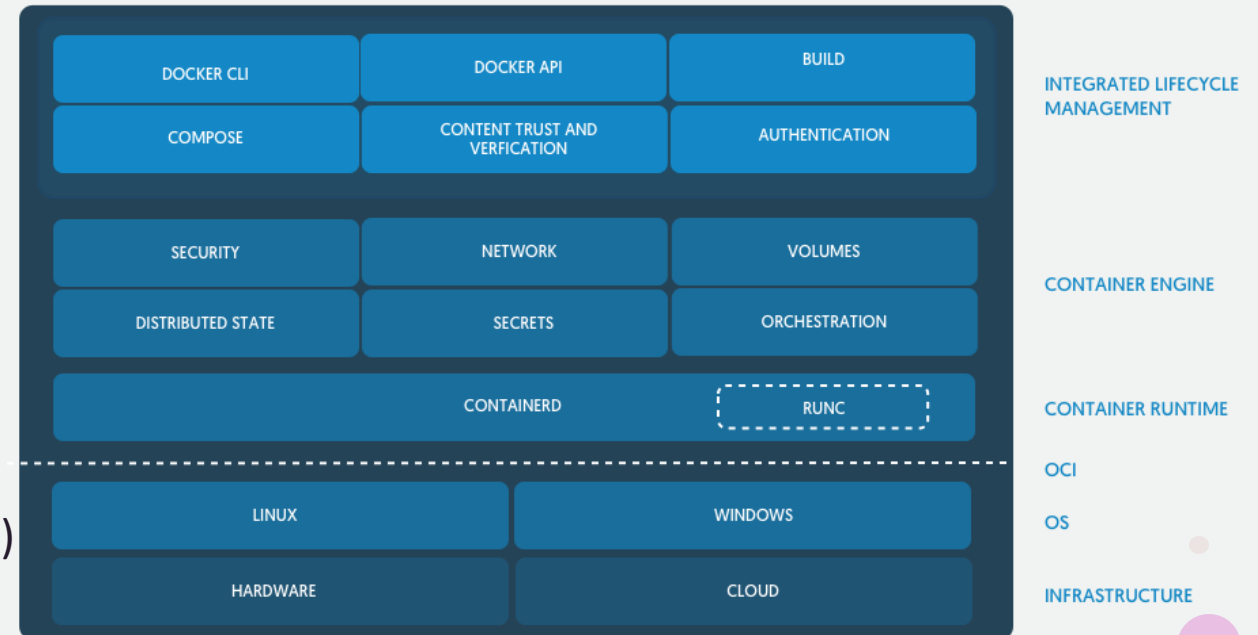
- **Découverte de services et équilibrage de charge**
  - Kubernetes peut exposer un conteneur à l'aide du nom DNS ou de sa propre adresse IP. Si le trafic vers un conteneur est élevé, Kubernetes est capable d'équilibrer la charge et de répartir le trafic réseau afin que le déploiement soit stable.
- **Orchestration du stockage Kubernetes**
  - Cela permet de monter automatiquement un système de stockage au choix, comme des stockages locaux, des fournisseurs de clouds publics, etc.
- **Déploiements et feedbacks automatiques**
  - Vous pouvez décrire l'état souhaité pour vos conteneurs déployés à l'aide de Kubernetes, et celui-ci peut faire évoluer l'état réel vers l'état souhaité à un rythme contrôlé.
  - Par exemple, vous pouvez automatiser Kubernetes pour créer de nouveaux conteneurs pour votre déploiement, supprimer les conteneurs existants et adopter toutes leurs ressources vers le nouveau conteneur.

# Avantages

- **Conditionnement automatique des conteneurs**
  - Vous fournissez à Kubernetes un cluster de nœuds qu'il peut utiliser pour exécuter des tâches conteneurisées. Vous indiquez à Kubernetes la quantité de CPU et de mémoire (RAM) dont chaque conteneur a besoin. Kubernetes peut adapter les conteneurs sur vos nœuds afin d'utiliser au mieux vos ressources.
- **Auto-réparation**
  - Kubernetes redémarre les conteneurs qui échouent, remplace les conteneurs, tue les conteneurs qui ne répondent pas à votre contrôle de santé défini par l'utilisateur, et ne les annonce pas aux clients avant qu'ils ne soient prêts à servir.
- **Gestion des secrets et des configurations**
  - Kubernetes vous permet de stocker et de gérer des informations sensibles, telles que des mots de passe, des jetons OAuth et des clés SSH. Vous pouvez déployer et mettre à jour les secrets et la configuration des applications sans reconstruire vos images de conteneurs et sans exposer les secrets dans la configuration de votre pile.

# Rappel: Containerd

- La notion de conteneur n'est pas un concept natif des OS
- Containerd est une couche d'abstraction entre les outils de gestion de conteneurs et le système d'exploitation
- Il implante tous les concepts liés aux conteneurs et à leur gestion en cachant les détails bas niveau liés à l'appel des primitives de l'OS
- Il offre des services de haut niveau pour gérer:
  - L'orchestration
  - La gestion des volumes
  - La sécurité
  - L'authentification
  - Les réseaux
  - ...
- Il est défini et utilisé par docker et kubernetes (entre autres)
- **Containerd ne gère pas l'orchestration multi-hôtes, l'auto-scaling, la haute disponibilité, la découverte de services, les déploiements progressifs ou l'équilibrage de charge. Ces fonctionnalités relèvent de Kubernetes.**



# Container Runtime Interface

- **Définition**

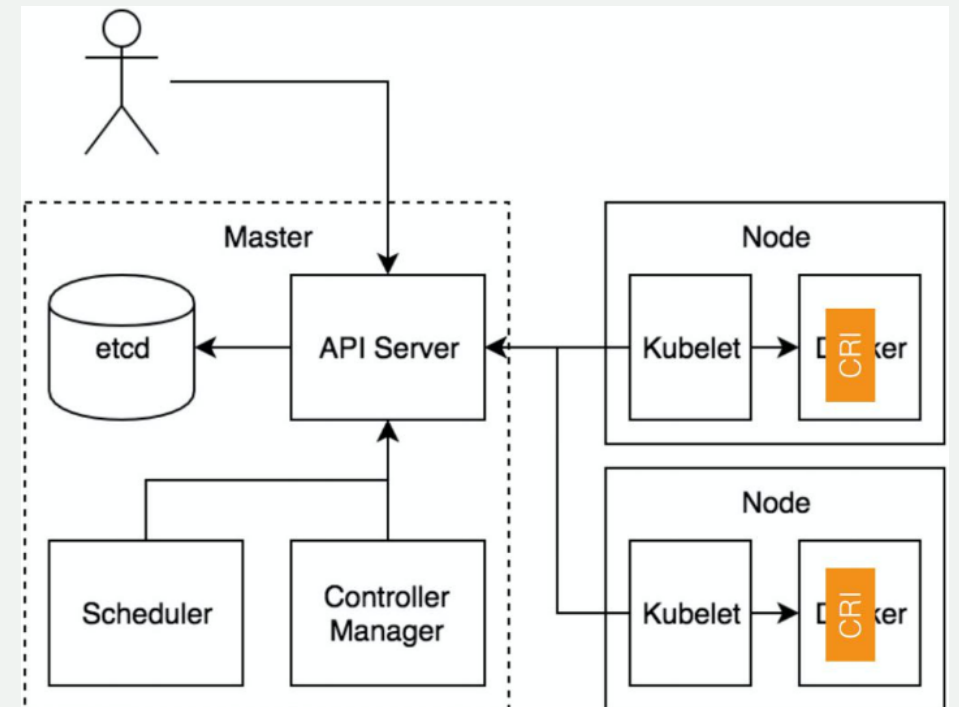
Le CRI est une interface (API) standard qui permet à Kubernetes (via le composant kubelet) de communiquer avec différents runtimes de conteneurs (comme containerd ou CRI-O), sans avoir à modifier le code de Kubernetes pour chaque runtime.

- **Fonctionnement**

Le kubelet utilise le CRI pour demander au runtime de lancer, arrêter, supprimer des conteneurs et gérer les images. Cette communication s'effectue via le protocole gRPC et des messages Protocol Buffers, assurant efficacité et extensibilité.

- **Découplage**

Avant le CRI, Kubernetes était fortement lié à Docker. Avec le CRI, il est possible d'utiliser n'importe quel runtime compatible, rendant Kubernetes plus flexible et évolutif.



# Container Runtime Interface

- **Composants principaux**

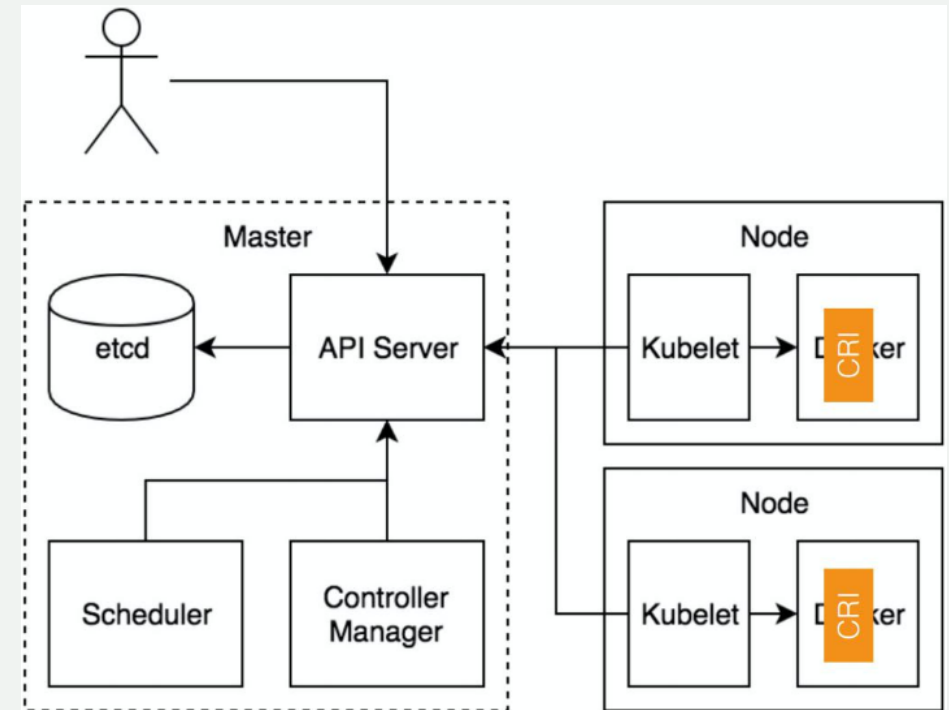
- Container Runtime Service : gère le cycle de vie des conteneurs (création, démarrage, arrêt, suppression).
- Image Service : gère le téléchargement, le stockage et la gestion des images de conteneurs.

- **Interopérabilité**

Grâce au CRI, il existe aujourd'hui plusieurs runtimes compatibles (containerd, CRI-O, etc.), ce qui favorise l'innovation et la spécialisation selon les besoins des utilisateurs.

- **Outils associés**

L'outil crictl permet d'interagir directement avec les runtimes compatibles CRI pour le débogage ou la gestion avancée des conteneurs.



# Installation

# Installation

- Pour faire des tests localement, il est recommandé d'utiliser microk8s
  - <https://ubuntu.com/tutorials/install-a-local-kubernetes-microk8s#1-overview>
- Vous pouvez également passer par un fournisseur
  - Amazon EKS: <https://aws.amazon.com/eks/>
  - Google GKE: <https://cloud.google.com/kubernetes-engine?hl=fr>
  - Azure AKS: <https://learn.microsoft.com/fr-fr/azure/aks/>
- Ou mettre en place votre propre stack avec
  - OpenShift
  - OpenStack
- Dans le cas où vous ne pouvez rien installer
  - <https://labs.play-k8s.com/>
  - <https://killercoda.com/>



# Installation locale avec microk8s

- `sudo snap install --classic microk8s`
- Pod to pod and pod-to-pod communication:
  - `sudo ufw allow in on cni0 && sudo ufw allow out on cni0`
  - `sudo ufw default allow routed`
- Addons:
  - `microk8s enable dns`
  - `microk8s enable dashboard`
  - `microk8s enable hostpath-storage`
  - `microk8s enable metrics-server`
- Accès au dashboard:
  - Génération d'un token de sécurité:
    - `token=$(kubectl -n kube-system get secrets | grep token | cut -d " " -f1)`
    - `microk8s kubectl -n kube-system describe secret $token`



# Exemples et exercices

- Clôner le repo git suivant:
  - <https://github.com/akoudri/microservices-training.git>

The slide features a light gray background with decorative elements in the corners. The top-left corner has a cluster of circles in shades of pink, purple, and orange. The top-right corner has circles in shades of blue, purple, and orange. The bottom-right corner has circles in shades of pink, purple, and teal. The main content is centered on the left side of the slide.

# Installation sur 4 VMs

Sur GCP

# Prérequis

- Compte GCP : Accès à un projet Google Cloud.
- VMs Créées : 4 machines virtuelles (VMs) sous Debian ou Ubuntu (l'historique semble utiliser Debian/Ubuntu).
  - Master (x1) :
    - Type de Machine : Minimum e2-standard-2 (2vCPU, 8Go RAM). n2d-standard-2 ou n2-standard-2 sont recommandés pour de meilleures performances.
    - Disque de Démarrage : Crucial : Choisir "Disque persistant SSD" (pd-ssd) lors de la création. Taille : 50 Go ou plus.
  - Workers (x3) :
    - Type de Machine : Minimum e2-standard-2. Adaptez selon les besoins de vos applications.
    - Disque de Démarrage : pd-balanced ou pd-ssd. Taille : 30-50 Go ou plus.
    - Réseau : Toutes les VMs doivent être dans le même réseau VPC et pouvoir communiquer entre elles (les règles de pare-feu GCP par défaut autorisent généralement le trafic interne sur le même réseau). Assurez-vous que les ports requis par Kubernetes sont ouverts si vous avez des règles de pare-feu personnalisées.
- Accès SSH : Pouvoir se connecter en SSH à toutes les VMs.
- Les commandes seront exécutées avec sudo

# Phase 1: Étapes communes

- Connexion SSH : Connectez-vous à chaque nœud.
- Mise à jour du système et installation des prérequis :
  - `sudo apt update`
  - `sudo apt install -y apt-transport-https ca-certificates curl gnupg lsb-release uidmap`
- Désactivation du Swap : Kubernetes requiert que le swap soit désactivé.
  - `sudo swapoff -a` # Désactiver pour la session actuelle
  - `sudo sed -i ' / swap / s/^(\.*\)$/#\1/g' /etc/fstab` # Désactiver de manière permanente
- Configuration du transfert IPv4 et `br_netfilter` : Nécessaire pour la mise en réseau des pods et la communication via les ponts réseau.

```
# Créer le fichier de configuration pour les
modules noyau
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
```

# Phase 1 - Étapes communes

- Charger les modules immédiatement
  - `sudo modprobe overlay`
  - `sudo modprobe br_netfilter`

```
cat << EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
```

- Appliquer les paramètres sysctl sans redémarrer
  - `sudo sysctl --system`
- Vérifier que `net.ipv4.ip_forward` est bien à 1
  - `sudo sysctl net.ipv4.ip_forward`

# Phase 1 - Étapes communes

- Installation de containerd

# Ajouter la clé GPG officielle de Docker (qui fournit containerd.io)

```
sudo install -m 0755 -d /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release && echo "$ID")/gpg | sudo gpg --dearmor  
-o /etc/apt/keyrings/docker.gpg # en une seule ligne
```

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

# Ajouter le dépôt Docker

```
echo "deb [arch=$(dpkg --print-architecture) signed-  
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/$(. /etc/os-release && echo "$ID")  
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" |
```

```
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null # en une seule ligne
```

# Mettre à jour et installer containerd

```
sudo apt update
```

```
sudo apt install -y containerd.io
```



# Phase 1 - Étapes communes

- Configuration de containerd

# Créer le répertoire de configuration s'il n'existe pas

```
sudo mkdir -p /etc/containerd
```

# Générer la configuration par défaut

```
sudo containerd config default | sudo tee /etc/containerd/config.toml
```

# Kubelet recommande d'utiliser le driver cgroup systemd

```
sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/g' /etc/containerd/config.toml
```

# Redémarrer containerd pour appliquer la configuration

```
sudo systemctl restart containerd
```

```
sudo systemctl enable containerd
```

# Phase 1 - Étapes communes

- Installation des outils Kubernetes (kubelet, kubeadm, kubectl) :

# Ajouter la clé GPG du dépôt Kubernetes (sur une ligne)

```
sudo curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.32/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
```

# Ajouter le dépôt Kubernetes (sur une ligne)

```
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.32/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

# Mettre à jour et installer les outils K8s

```
sudo apt-get update
```

```
sudo apt-get install -y kubelet kubeadm kubectl
```

# Empêcher les mises à jour automatiques des outils K8s

```
sudo apt-mark hold kubelet kubeadm kubectl
```

- Activation du service Kubelet :

```
sudo systemctl enable --now kubelet
```

# Phase 2 - Master

- Connexion SSH au nœud désigné comme master.
  - Initialiser le Control Plane :
    - Remplacez MASTER\_INTERNAL\_IP par l'adresse IP interne de votre VM master GCP (ex: 10.132.0.6).
    - Le CIDR 192.168.0.0/16 est souvent utilisé par Calico par défaut. Si vous choisissez un autre CNI ou une autre plage, adaptez cette valeur.
    - Utilisez l'IP interne de votre master GCP
- ```
export INTERNAL_IP="10.132.0.6"  
export POD_NETWORK_CIDR="192.168.0.0/16"  
export PUBLIC_IP="35.187.39.7"  
sudo kubeadm init --apiserver-advertise-address=${MASTER_INTERNAL_IP} \  
--pod-network-cidr=${POD_NETWORK_CIDR} --apiserver-cert-extra-sans=${PUBLIC_IP}
```
- **IMPORTANT** : À la fin de cette commande, kubeadm init affichera une commande kubeadm join ... avec un token et un hash.
    - Copiez cette commande complète et conservez-la précieusement. Vous en aurez besoin pour joindre les nœuds workers.

# Phase 2 - Master

- Configurer kubectl pour votre utilisateur
  - `mkdir -p $HOME/.kube`
  - `sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config`
  - `sudo chown $(id -u):$(id -g) $HOME/.kube/config`
- Installer le Plugin Réseau (CNI - Calico) :
  - Le cluster a besoin d'un CNI pour que les pods puissent communiquer.
  - `kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.27.3/manifests/calico.yaml`

# Phase 2 - Master

- Vérifier l'état du Master : Attendez quelques instants que les pods Calico et CoreDNS démarrent.
  - `kubectl get nodes` # Le master devrait passer à l'état "Ready" après un court instant
  - `kubectl get pods -A` # Vérifiez que les pods dans kube-system (et calico-system) sont Running ou Completed
- Configuration du firewall ( GCP Console -> VPC Network -> Firewall)
  - Ouvrir le Port 6443 dans le Pare-feu GCP : L'API Server Kubernetes écoute sur le port TCP 6443. Vous devez créer une règle de pare-feu dans GCP pour autoriser le trafic entrant sur ce port vers votre VM master, mais uniquement depuis votre adresse IP publique actuelle.
  - Ajouter au préalable un tag réseau au master

# Phase 3: Workers et vérification

- Connectez-vous en SSH à chaque nœud worker, un par un.
  - Utilisez la commande `kubeadm join ...` que vous avez copiée depuis la sortie de `kubeadm init` sur le master (en `sudo`)
- Connectez-vous en SSH au nœud master
  - Vérifier l'état de tous les nœuds: `kubectl get nodes -o wide`

# Phase 4 - Client

- Installation de kubectl sur le client
  - `curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"`
  - `curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"`
  - `echo "$(cat kubectl.sha256) kubectl" | sha256sum --check`
  - `sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl`
  - `kubectl version --client`
- Copier le fichier ~/.kube/config de la machine distante et adapter le contenu
  - `chmod 600 ~/.kube/config`



# Part 5: Load Balancer et Ingress

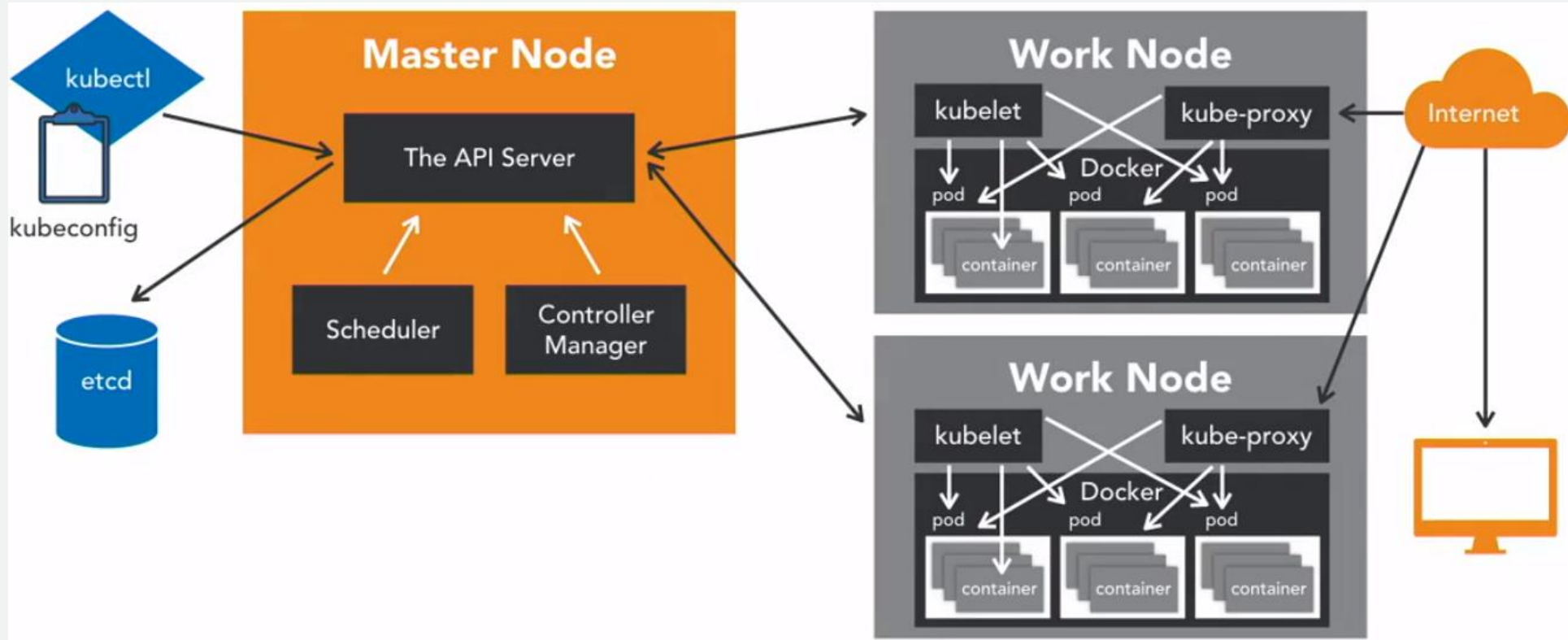
- MetalLB Fournit une implémentation de Load Balancer (L4) natif pour les clusters Kubernetes, en attribuant des IP externes aux services de type LoadBalancer
  - Installation: cf. Script `install-metallb.sh`
- Ingress-NGINX route le trafic HTTP/HTTPS vers les services internes via des règles déclaratives
  - Installation: cf. Script `install-ingress.sh`
- Sur les clusters gérés (GKE par exemple), cette étape n'est pas nécessaire, voire même inutile.

# Extra: réinitialisation

- Master:
  - `sudo kubeadm reset -f`
  - `sudo rm -rf /etc/cni/net.d ~/.kube /var/lib/etcd`
  - `sudo systemctl restart kubelet containerd`
  - Puis reprendre au début de la procédure
- Workers
  - `sudo kubeadm reset -f`
  - Puis reprendre la commande `kubeadm join`

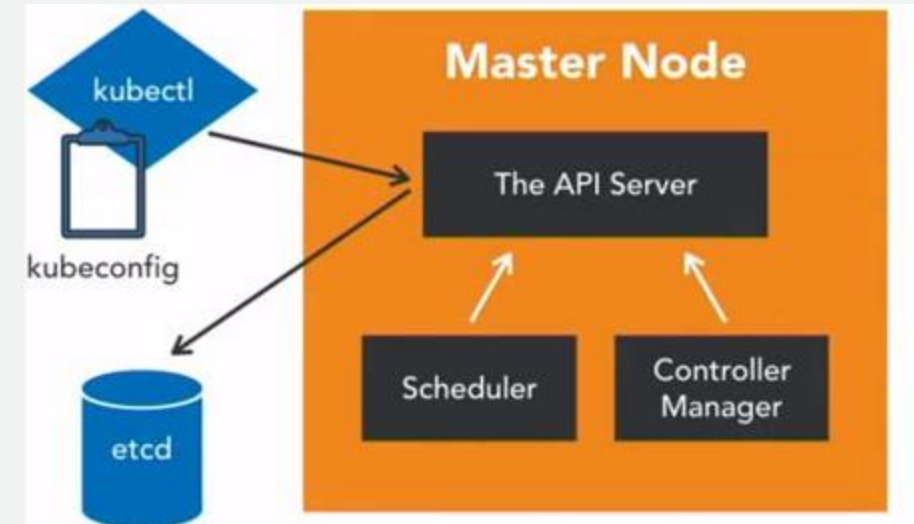
# Concepts

# Architecture – Vue d'ensemble



# Master

- Le nœud master d'un cluster Kubernetes est le composant en charge de gérer l'état du cluster. Il est responsable de toutes les fonctions globales au niveau du cluster, telles que la planification des pods, le maintien de l'état souhaité, la mise à l'échelle des pods et le déploiement des mises à jour.
- Les composants du master node travaillent ensemble pour traiter les commandes de l'utilisateur, gérer la persistance du cluster, planifier les charges de travail et gérer l'orchestration des processus (pods). Ils veillent à ce que le cluster fonctionne comme prévu et gèrent toute modification de son état.



# API Server

- **Point d'entrée central du cluster**

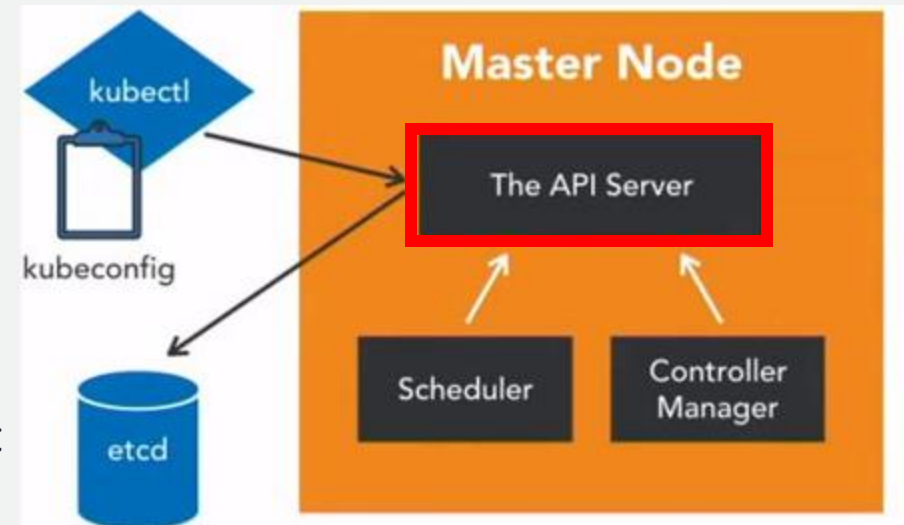
- L'API Server est le composant central du plan de contrôle Kubernetes.
- Il reçoit toutes les requêtes (création, modification, suppression, interrogation) concernant l'état du cluster, que ce soit depuis des utilisateurs, des outils comme kubectl, ou d'autres composants internes.

- **Interface de communication unique**

- Il expose une API HTTP/REST qui permet aux utilisateurs, aux composants du cluster et aux systèmes externes de communiquer avec Kubernetes.
- Toutes les interactions passent par lui, ce qui garantit la cohérence et la sécurité des opérations.

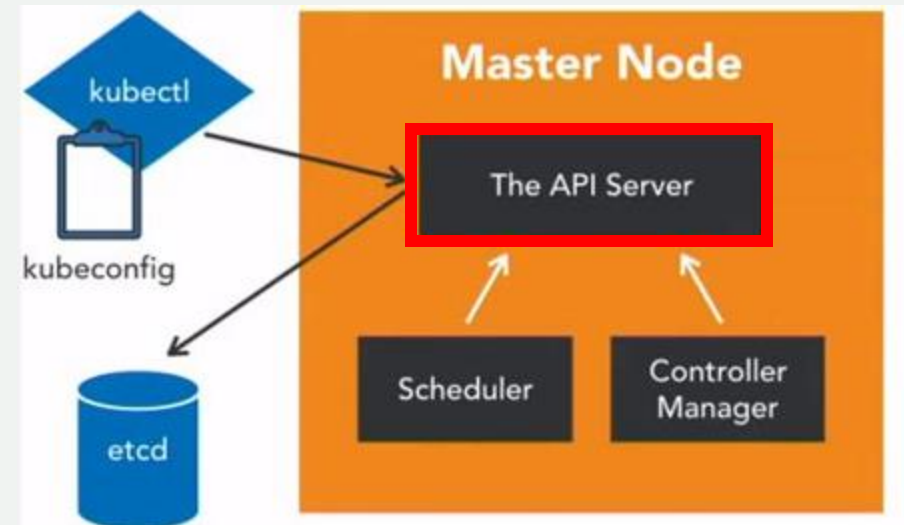
- **Validation, authentification et autorisation**

- Chaque requête reçue est authentifiée, validée et autorisée avant d'être traitée.
- L'API Server s'assure que seules les opérations valides et autorisées sont exécutées sur le cluster.



# API Server

- **Gestion de l'état du cluster via etcd**
  - Il est le seul composant à interagir directement avec la base de données etcd, qui stocke l'état souhaité du cluster.
  - L'API Server lit et écrit dans etcd pour enregistrer ou récupérer la configuration et l'état des ressources (Pods, Services, ConfigMaps, etc.).
- **Interaction avec les autres composants**
  - Après validation d'une requête, l'API Server notifie les autres composants du plan de contrôle (Scheduler, Controller Manager, Kubelet, etc.) pour qu'ils prennent les mesures nécessaires (déploiement de pods, gestion des ressources, etc.).
- **Versioning et groupes d'API**
  - Il gère plusieurs versions et **groupes d'API** pour permettre l'évolution de Kubernetes tout en maintenant la compatibilité avec les anciennes versions.





# Controller Manager

- **Rôle central d'automatisation**

Le Controller Manager est un composant clé du plan de contrôle de Kubernetes. Il orchestre l'automatisation du cluster en exécutant plusieurs "contrôleurs", qui sont des boucles logicielles surveillant en continu l'état du cluster pour le rapprocher de l'état désiré défini par l'utilisateur.

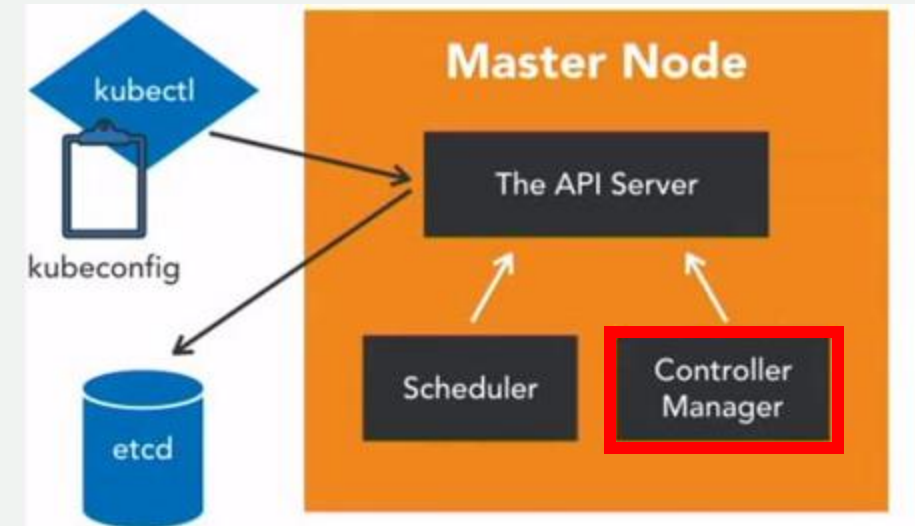
- **Réconciliation de l'état**

Chaque contrôleur compare l'état réel du cluster à l'état attendu (défini dans les manifestes YAML) : s'il détecte un écart (par exemple, un pod supprimé alors qu'il doit y avoir 3 répliques), il agit pour corriger la situation (ici, en recréant un pod).

- **Principaux contrôleurs intégrés**

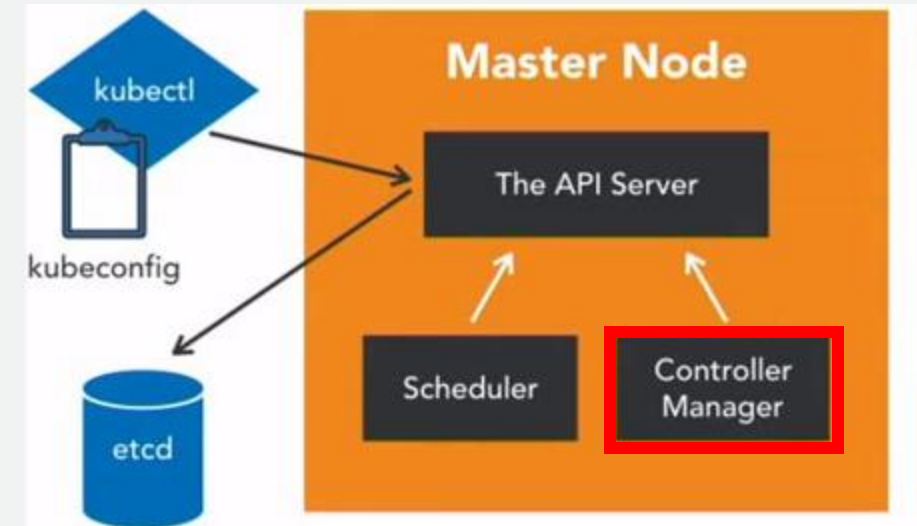
Parmi les contrôleurs les plus importants :

- **Node Controller** : surveille la santé des nœuds et gère leur inscription/suppression.
- **Replication Controller** : garantit le nombre souhaité de pods pour une application.
- **Endpoints Controller** : gère les endpoints des services pour assurer la connectivité.
- **Service Account & Token Controller** : gère les comptes de service et leurs jetons d'accès.
- **PersistentVolume Controller** : supervise l'attachement des volumes persistants.



# Controller Manager

- **Boucle de contrôle continue**  
Les contrôleurs fonctionnent en boucle : ils interrogent l'API Server, détectent les écarts, puis appliquent les actions nécessaires pour rétablir l'état désiré.
- **Haute disponibilité**  
Plusieurs instances du Controller Manager peuvent tourner, mais une seule est active à la fois grâce à un mécanisme d'élection de leader, pour éviter les conflits d'actions.
- **Extensibilité**  
Kubernetes permet d'ajouter des contrôleurs personnalisés pour automatiser des comportements spécifiques selon les besoins de l'organisation.



# Scheduler

- **Rôle principal**

Le scheduler est responsable d'assigner chaque Pod nouvellement créé (et non encore planifié) à un nœud du cluster où il pourra être exécuté.

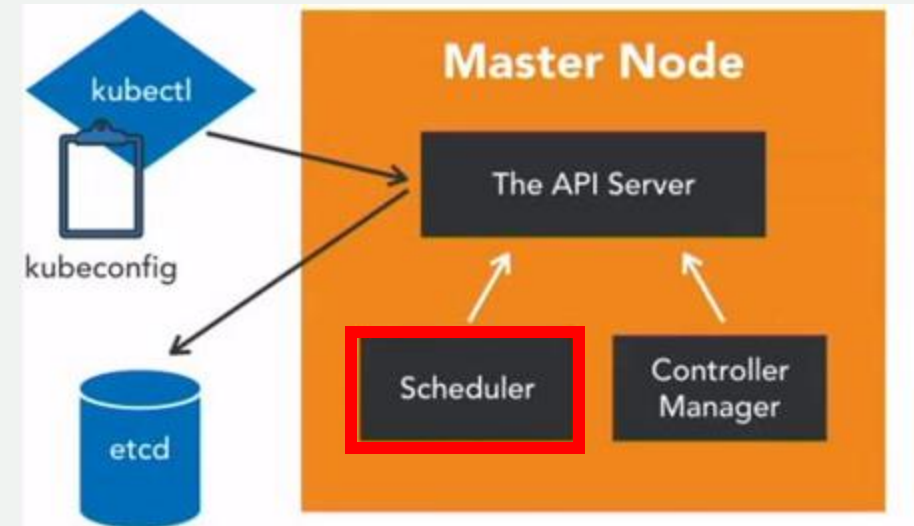
- **Processus de planification**

Pour chaque Pod à planifier, le scheduler suit deux grandes étapes :

- **Filtrage** : il identifie la liste des nœuds « faisables » qui répondent aux contraintes du Pod (ressources disponibles, labels, affinités, taints/tolerations, etc.).
- **Scoring** : il attribue un score à chaque nœud filtré selon différents critères (charge, proximité des données, affinités, etc.), puis sélectionne le nœud ayant le score le plus élevé pour y placer le Pod.

- **Décision et binding**

Une fois le meilleur nœud choisi, le scheduler notifie l'API Server de cette décision : c'est l'opération de binding. Le kubelet du nœud sélectionné prend alors le relais pour lancer le Pod.



# Scheduler

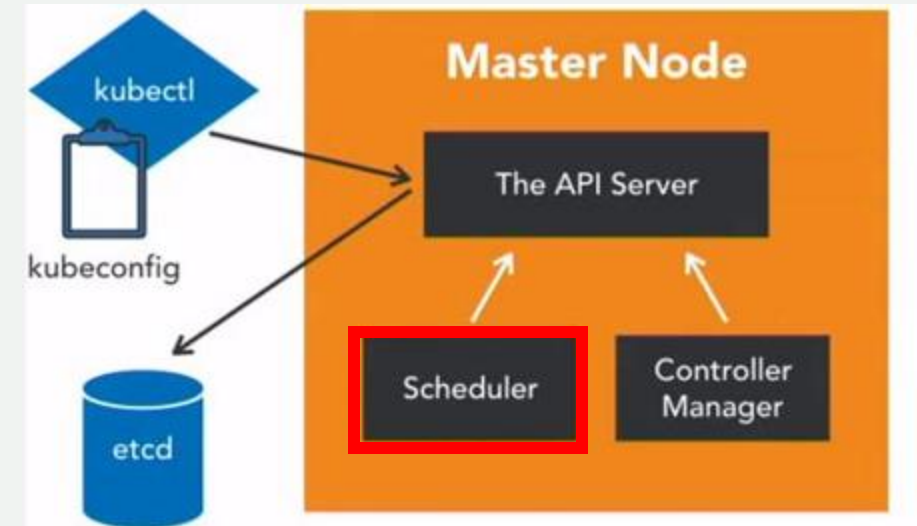
- **Prise en compte de multiples critères**

Le scheduler considère :

- Les ressources (CPU, mémoire, stockage) demandées et disponibles
- Les contraintes d'affinité/anti-affinité (préférences ou obligations de colocalisation/séparation de Pods)
- Les politiques de tolérance/rejet (taints/tolerations)
- La localisation des données ou des images
- Les contraintes personnalisées via des plugins ou des politiques spécifiques

- **Extensibilité**

Kubernetes permet de personnaliser ou d'étendre le scheduler via des plugins, des politiques de scheduling ou même de déployer plusieurs schedulers dans un même cluster pour des besoins spécifiques.



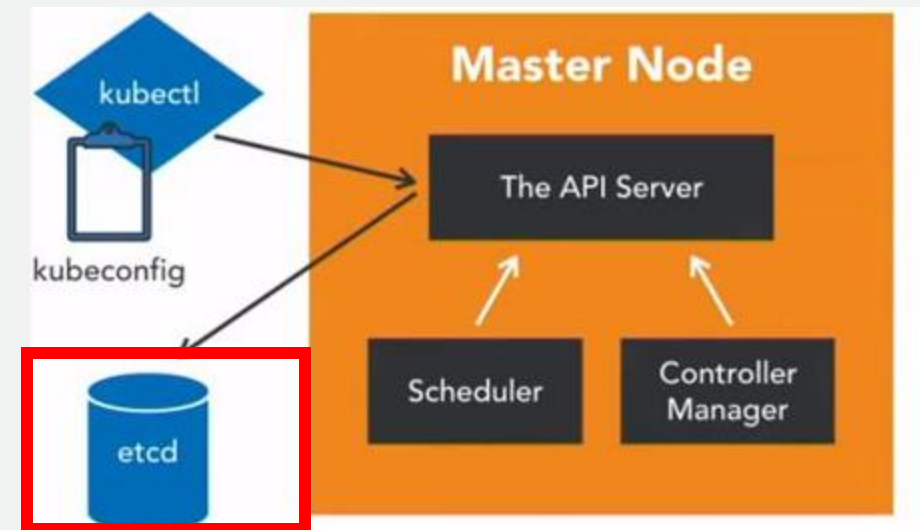
# ETCD

- **Stockage de l'état du cluster**

- **Source unique de vérité** : Etcd conserve toutes les données critiques du cluster, notamment les configurations, l'état des ressources (Pods, Services, Deployments), les secrets et les métadonnées.
- **Synchronisation** : Kubernetes utilise la fonction watch d'etcd pour détecter les écarts entre l'état réel et l'état désiré du cluster, déclenchant des actions correctives (ex. : redémarrage de Pods défectueux).

- **Haute disponibilité et cohérence**

- **Consensus via Raft** : Etcd utilise le protocole Raft pour maintenir la cohérence des données entre les nœuds, même en cas de panne. Un leader est élu pour gérer les écritures, et les données sont répliquées sur un nombre impair de nœuds pour éviter les pertes.
- **Tolérance aux pannes** : En cas de défaillance d'un nœud, etcd garantit la continuité du service grâce à son mécanisme d'élection automatique.



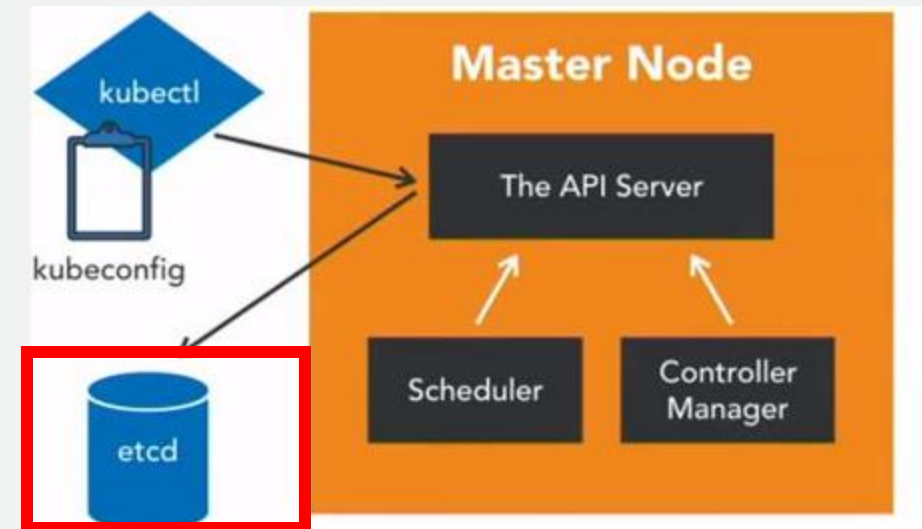
# ETCD

- **Intégration avec Kubernetes**

- **Interaction via l'API Server** : Toutes les opérations sur le cluster (création, modification, suppression de ressources) passent par l'API Server, qui persiste les données dans etcd.
- **Support des composants clés** : Le Controller Manager et le Scheduler s'appuient sur etcd pour piloter l'automatisation et la planification des ressources.

- **Sécurité et maintenance**

- **Chiffrement TLS** : Les communications avec etcd sont sécurisées par défaut, protégeant les données sensibles du cluster.
- **Gestion simplifiée** : L'opérateur etcd automatise les tâches complexes comme les sauvegardes, les restaurations ou la mise à l'échelle du cluster.



# ETCD

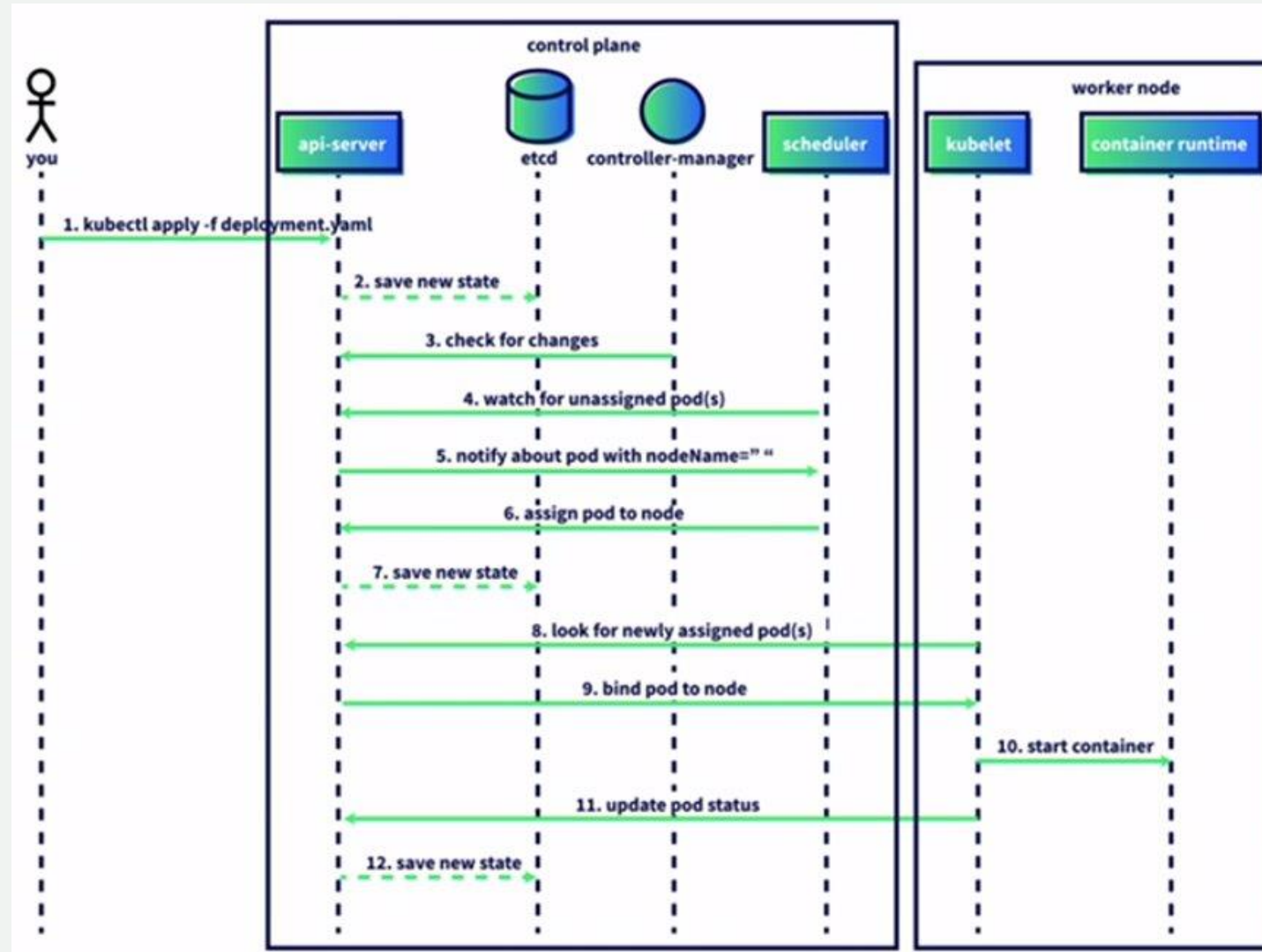


- etcd est une base de données clé-valeur **distribuée**, open source, conçue pour stocker et gérer les informations critiques nécessaires au fonctionnement des systèmes distribués, en particulier Kubernetes.
- Elle sert de socle de données principal pour de nombreux orchestrateurs de conteneurs et plateformes cloud-native, garantissant la cohérence, la haute disponibilité et la résilience des données de configuration, d'état et de métadonnées du cluster
- Page officielle: <https://etcd.io/>
- Exécution avec docker:
  - `docker run -it --name etcd --rm -p 2379:2379 -e ALLOW_NONE_AUTHENTICATION=yes -d bitnami/etcd`
  - `docker exec etcd etcdctl put hello "Hello World"`
  - `docker exec etcd etcdctl get hello`



# ETCD – Cas d'usage

- etcd est le cœur persistant de Kubernetes : sans lui, le cluster perdrait sa capacité à maintenir un état cohérent et à orchestrer les applications de manière fiable.
- Quand un Pod est créé via kubectl, l'API Server enregistre cette demande dans etcd.
- Le Scheduler consulte ensuite etcd pour planifier le Pod sur un nœud, et le Controller Manager vérifie en continu que le nombre de réplicas correspond à l'état désiré.
- TP** – Interrogation de la base





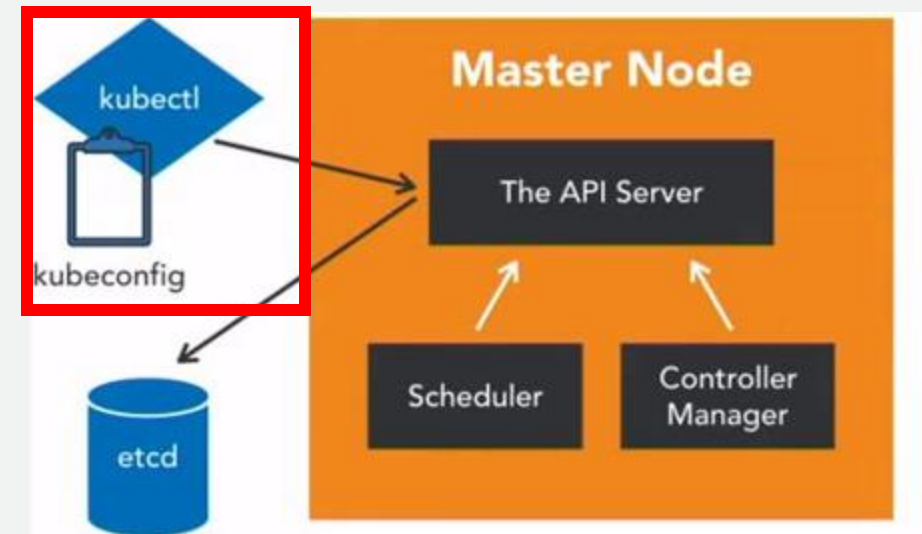
# Kubectl

- **Outil en ligne de commande officiel de Kubernetes**

Kubectl est l'interface principale pour administrer, déployer, surveiller et diagnostiquer les ressources d'un cluster Kubernetes directement depuis le terminal.

- **Communication avec l'API Server**

Toutes les commandes envoyées via kubectl transitent par l'API Server de Kubernetes, ce qui permet d'effectuer toutes les opérations (création, lecture, modification, suppression) sur les objets du cluster.



# Kubectl

- **Déploiement et gestion des ressources**

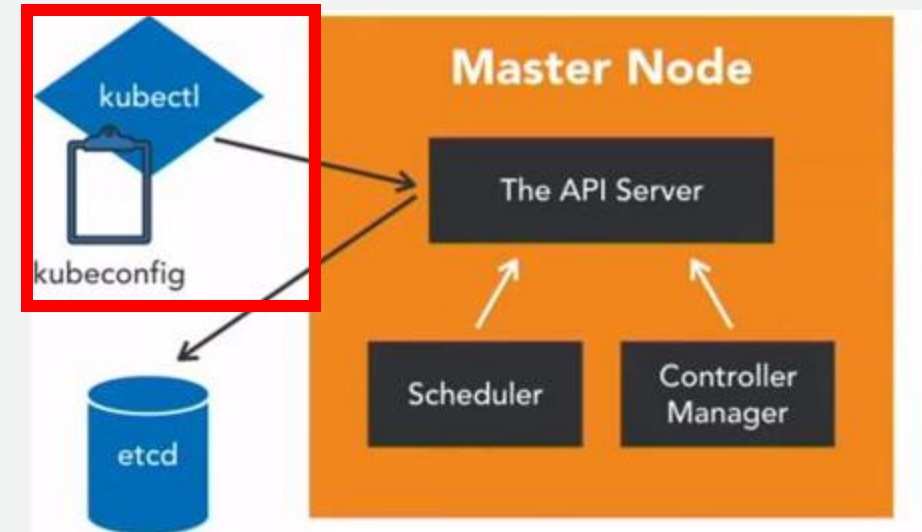
Permet de créer, mettre à jour, supprimer ou inspecter toutes les ressources Kubernetes : Pods, Services, Deployments, ConfigMaps, etc.

- **Supervision et diagnostic**

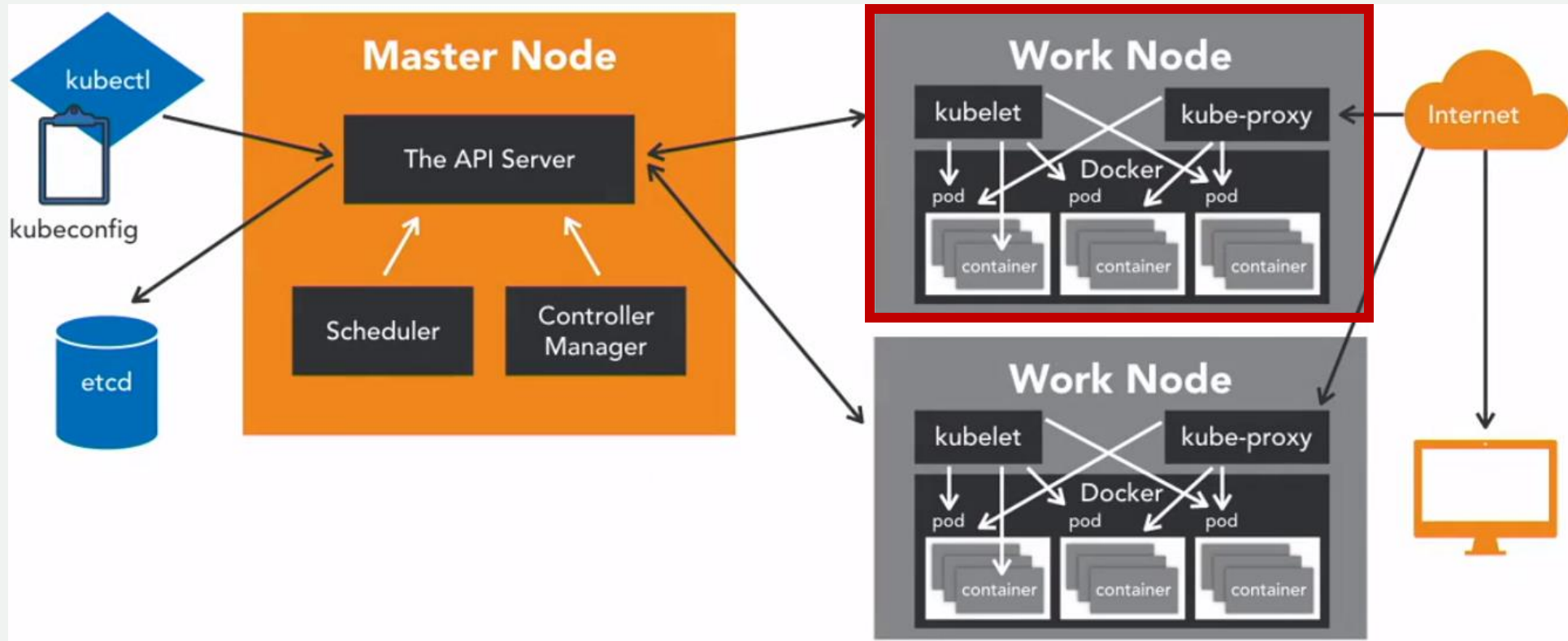
Offre des commandes pour visualiser l'état des ressources (kubectl get, kubectl describe), accéder aux logs (kubectl logs), exécuter des commandes dans les conteneurs (kubectl exec) et surveiller les performances (kubectl top).

- **Automatisation et scripts**

Kubectl s'intègre facilement dans des scripts d'automatisation pour déployer ou mettre à jour des applications de manière répétable.

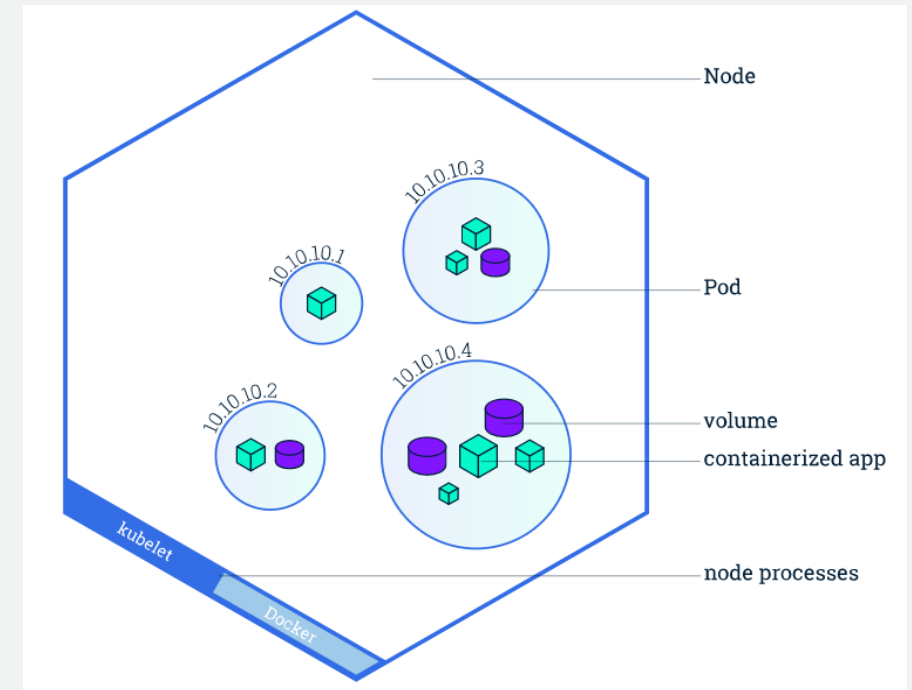


# Architecture – Vue d'ensemble



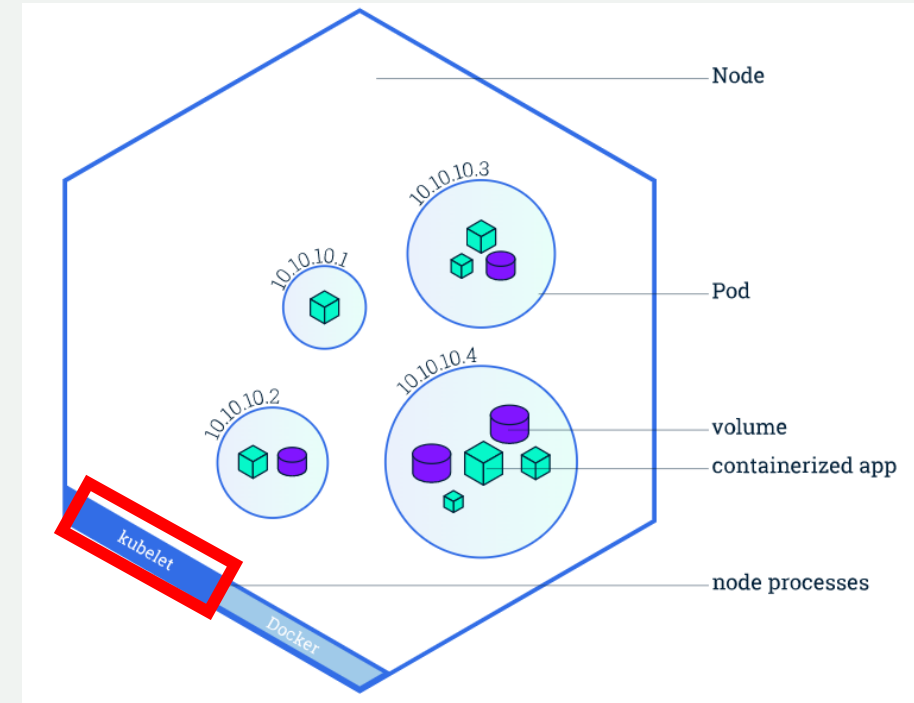
# Worker Node

- Un worker node est une machine (physique ou virtuelle) intégrée à un cluster Kubernetes, chargée d'exécuter les applications conteneurisées via les Pods.
- Il reçoit les tâches (Pods à exécuter) du plan de contrôle (control plane) et fournit les ressources nécessaires (CPU, mémoire, stockage, réseau) pour faire tourner ces applications.
- **Fonctionnement**
  - Le worker node exécute les Pods qui lui sont assignés, surveille leur état et redémarre automatiquement les conteneurs en cas de problème.
  - Il maintient la connectivité réseau entre les Pods du cluster et expose les applications vers l'extérieur si nécessaire.
  - Il communique en permanence avec le control plane pour recevoir de nouvelles instructions et reporter l'état des ressources locales.



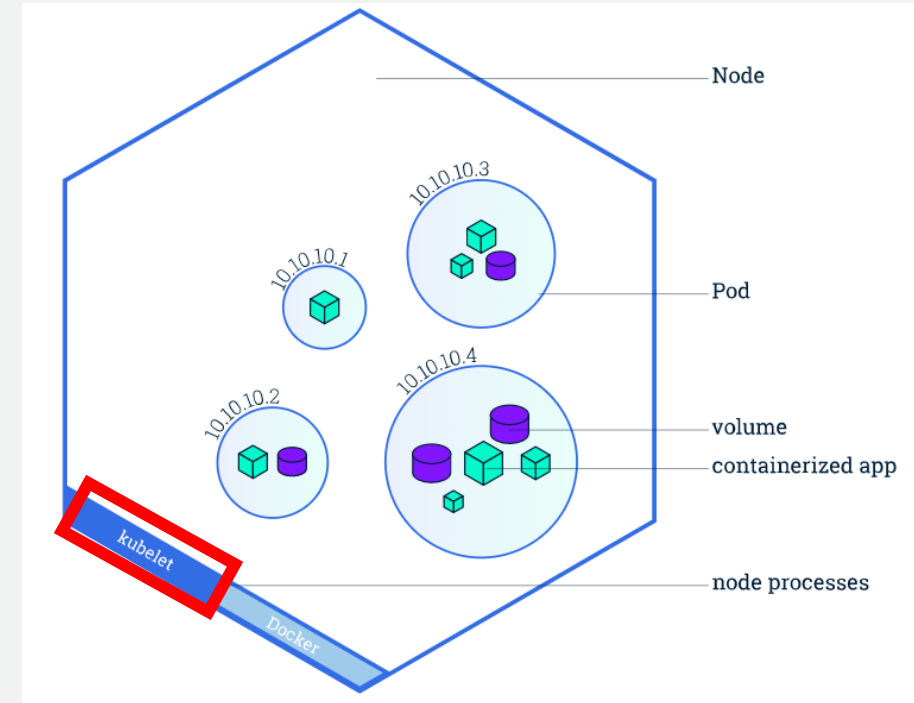
# Kubelet

- Kubelet est le **chef d'orchestre local** sur chaque nœud Kubernetes : il applique les ordres du control plane, supervise l'exécution des Pods, surveille leur état et garantit l'automatisation et la résilience des applications conteneurisées du cluster
- Il veille à ce que les Pods (et donc les conteneurs) soient créés, démarrés, maintenus et surveillés selon les spécifications reçues du plan de contrôle (control plane) via l'API Server.



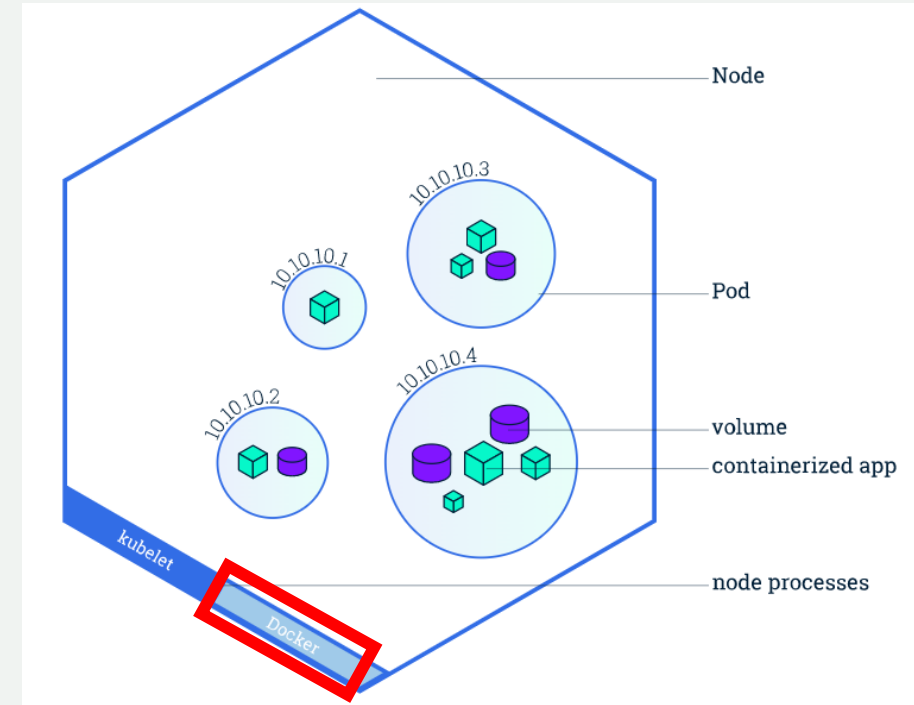
# Kubelet - Fonctionnement

- **Réception des instructions** : kubelet reçoit des instructions sous forme de PodSpecs (fichiers YAML ou JSON décrivant l'état désiré des Pods) depuis l'API Server.
- **Gestion du cycle de vie des Pods** : il s'assure que les conteneurs définis dans ces Pods sont bien démarrés, fonctionnent correctement et respectent l'état attendu (redémarrage automatique en cas de crash, arrêt, suppression...).
- **Surveillance et reporting** : kubelet surveille l'état des conteneurs et Pods, collecte des métriques (santé, ressources, disponibilité) et les remonte régulièrement au control plane.
- **Interaction avec le runtime de conteneur** : il délègue l'exécution concrète des conteneurs à un runtime compatible (containerd, CRI-O, etc.).



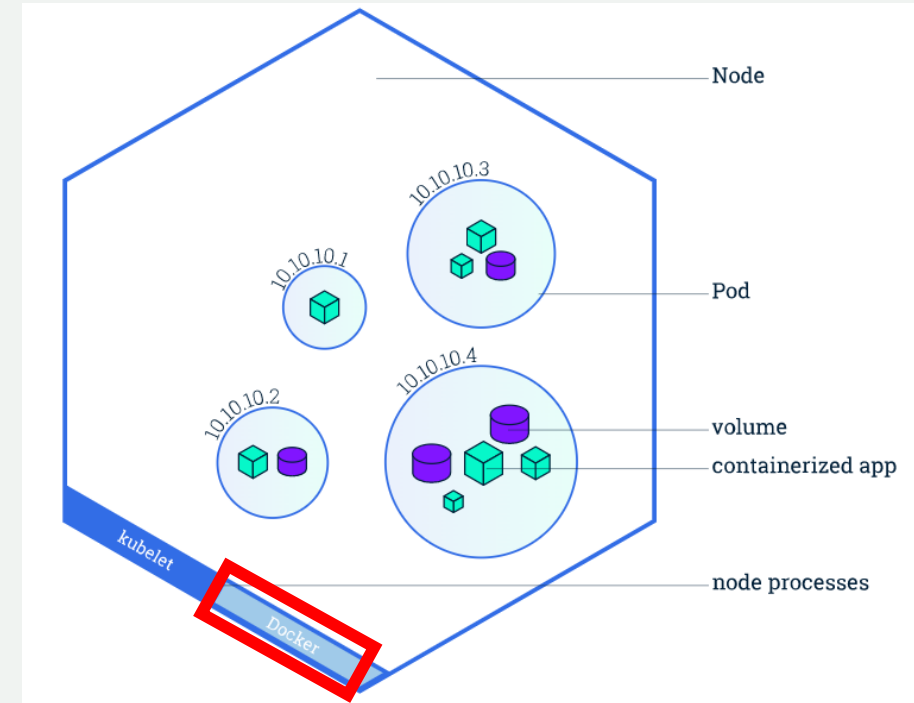
# Kube-proxy

- kube-proxy est un composant réseau essentiel qui s'exécute sur chaque nœud (worker node) d'un cluster Kubernetes.
- Il agit comme un proxy réseau et un load balancer : il met en place les règles de routage nécessaires pour que le trafic réseau à destination des services Kubernetes soit correctement redirigé vers les Pods correspondants.
- kube-proxy est le **chef d'orchestre réseau** de Kubernetes :
  - Il assure la connectivité entre les services et les Pods,
  - Met en place les règles réseau pour le routage et l'équilibrage de charge,
  - S'adapte dynamiquement aux changements dans le cluster,
  - Garantit l'accessibilité et la résilience des applications, aussi bien en interne qu'en externe



# Kube-proxy - Fonctionnement

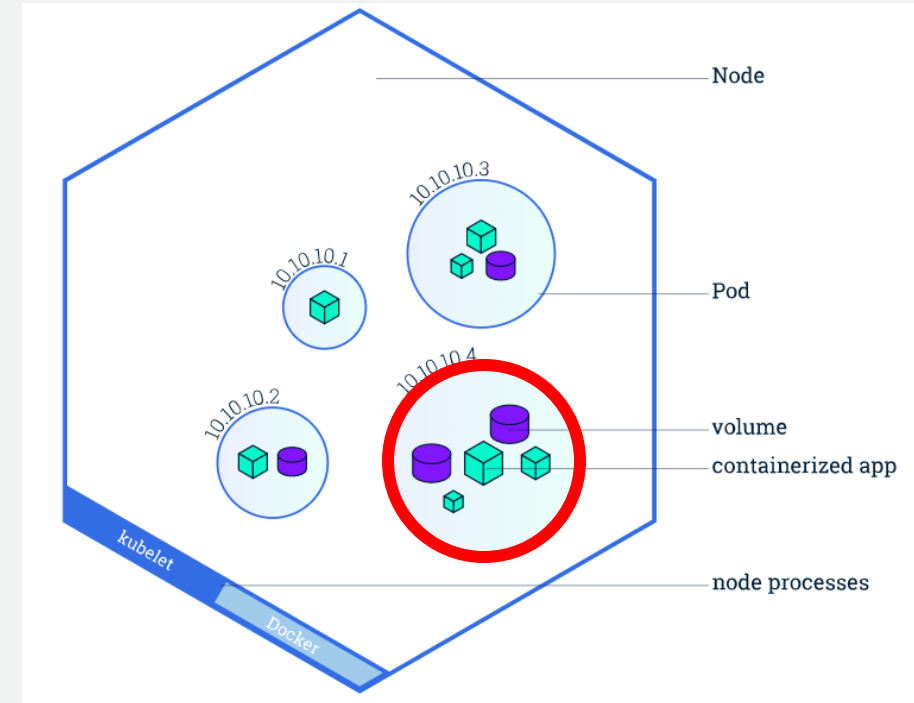
- **Surveillance des services et endpoints:** kube-proxy surveille en temps réel les objets Service et Endpoints via l'API Server. Lorsqu'un service ou un endpoint change (ajout, suppression, modification), il met à jour dynamiquement les règles réseau sur le nœud local.
- **Mise en place des règles réseau:** Il configure le système réseau du nœud (iptables, IPVS, ou plus récemment eBPF) pour intercepter les connexions à l'IP virtuelle du service (ClusterIP) et les rediriger vers l'un des Pods disponibles.
- **Équilibrage de charge:** kube-proxy répartit le trafic entrant entre les différents Pods d'un même service, selon des algorithmes comme le round-robin ou le hashing, assurant ainsi la haute disponibilité et l'équilibrage de charge.
- **Gestion des défaillances:** Si un Pod devient indisponible, kube-proxy met à jour ses règles pour que le trafic ne soit plus routé vers ce Pod, évitant ainsi toute interruption de service.
- **Sans kube-proxy, la communication entre services et Pods serait quasi-impossible (ou manuelle), ce qui rendrait le réseau Kubernetes inutilisable en production.**





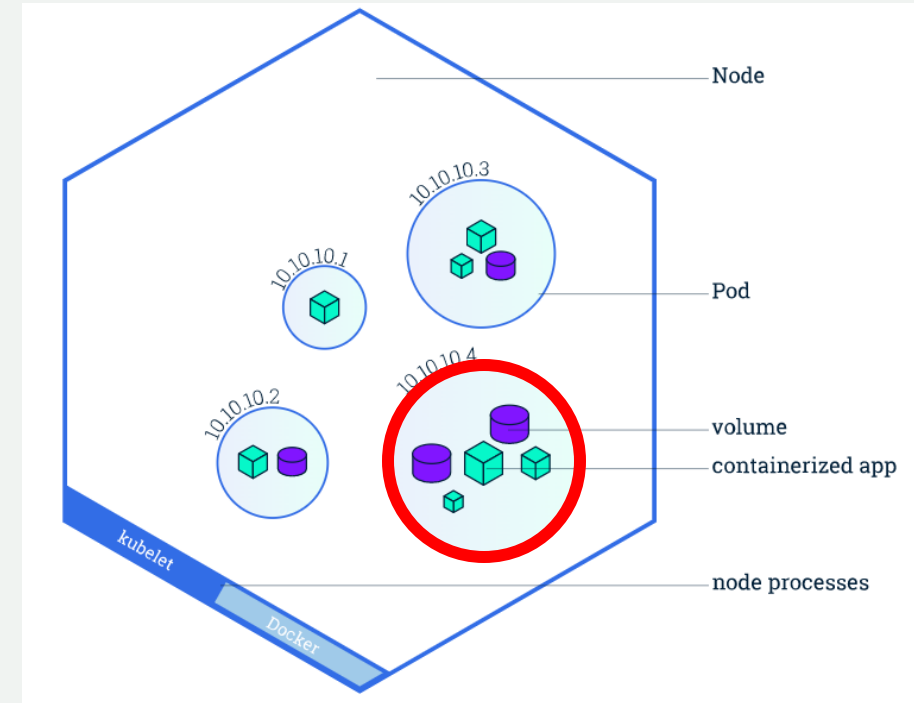
# Pod

- Le Pod est l'unité minimale d'exécution dans Kubernetes : il encapsule un ou plusieurs conteneurs qui partagent réseau et stockage.
- Il représente un ou plusieurs conteneurs (généralement un seul) qui partagent le même espace réseau, le même stockage et le même cycle de vie.
- Les conteneurs d'un même Pod sont toujours déployés ensemble sur le même nœud et peuvent communiquer facilement entre eux via localhost.
- Il facilite la gestion, le déploiement et l'automatisation des applications conteneurisées dans le cluster.
- **Les objets de plus haut niveau (Deployment, ReplicaSet, etc.) gèrent en pratique des groupes de Pods pour assurer la scalabilité et la résilience.**



# Pod - Caractéristiques

- **Partage des ressources.** Tous les conteneurs d'un Pod partagent :
  - L'adresse IP du Pod (et donc les ports réseau)
  - Les volumes de stockage attachés au Pod
  - Certaines configurations (variables d'environnement, secrets, etc.)
- **Utilisation typique**
  - Un Pod contient souvent un seul conteneur (cas le plus courant).
  - Plusieurs conteneurs dans un même Pod servent généralement à fournir des fonctionnalités complémentaires (ex : un conteneur principal + un sidecar pour la gestion des logs ou des proxys).
- **Cycle de vie**
  - Un Pod est éphémère : il peut être créé, détruit, remplacé automatiquement par Kubernetes selon l'état désiré.
  - Il n'est pas conçu pour être mis à jour : pour changer la configuration ou l'image d'un Pod, on le remplace par un nouveau.



# Node Controller

- Le contrôleur Node est un composant clé du plan de contrôle Kubernetes.
- Il est chargé de gérer l'état et le cycle de vie des nœuds (nodes) dans le cluster.
- Son objectif est de s'assurer que la liste des nœuds reflète la réalité de l'infrastructure, de surveiller leur santé, et de réagir automatiquement en cas de défaillance ou de changement d'état d'un nœud.
- Exemples de scénarii:
  - Nœud sain : Le kubelet envoie régulièrement des heartbeats à l'API Server. Le contrôleur Node maintient le nœud dans l'état "Ready".
  - Défaillance ou perte de connexion : Si le kubelet cesse d'envoyer des heartbeats (par exemple, à cause d'un crash ou d'une coupure réseau), le contrôleur Node marque le nœud comme "NotReady" ou "Unknown". Il applique alors des taints et peut déclencher l'expulsion des pods après un délai configurable (node-monitor-grace-period).
  - Suppression d'un nœud cloud : Si une VM est supprimée côté fournisseur cloud, le contrôleur Node détecte la disparition et supprime le nœud du cluster.

# Node Controller - Fonctionnement

- **Attribution du CIDR réseau** : Lorsqu'un nœud rejoint le cluster, le contrôleur Node peut lui attribuer automatiquement un bloc CIDR pour la gestion de son réseau interne (si l'option est activée).
- **Synchronisation avec le cloud** : Dans un environnement cloud, il maintient la liste des nœuds à jour en vérifiant régulièrement avec le fournisseur cloud si les machines associées aux nœuds Kubernetes existent toujours. Si une VM disparaît, le contrôleur Node supprime le nœud correspondant du cluster.
- **Surveillance de l'état des nœuds** : Il surveille la santé des nœuds via les signaux envoyés par le kubelet (heartbeat). Si un nœud devient injoignable ou cesse de répondre, le contrôleur Node met à jour son état et prend des mesures correctives, comme l'ajout de taints pour empêcher la planification de nouveaux pods, ou l'expulsion des pods existants après un certain délai.
- **Gestion des taints et tolérances** : Lorsqu'un nœud devient injoignable, il applique des "taints" (`node.kubernetes.io/unreachable`) pour contrôler la planification et l'expulsion des pods, en fonction des tolérances définies dans les manifestes des pods.

# Node Controller - Avantages

- **Haute disponibilité** : Permet de réagir automatiquement aux défaillances de nœuds, en réaffectant les pods sur des nœuds sains.
- **Synchronisation avec l'infrastructure réelle** : Garantit que l'état du cluster Kubernetes correspond à l'état réel des machines sous-jacentes.
- **Gestion automatisée** : Prend en charge l'attribution réseau, la surveillance de la santé et la gestion du cycle de vie des nœuds sans intervention manuelle.
- Exemples de commande:
  - *kubectrl get nodes* #Liste les noeuds du cluster
  - *kubectrl describe node my-node* #Affiche les détails d'un noeud
  - *kubectrl cordon my-node* #Rend le node non planifiable
  - *kubectrl drain my-node* #Évacue le node du cluster

# Garbage Collector Controller

- Le contrôleur Garbage Collector de Kubernetes est responsable du nettoyage automatique des ressources devenues inutiles ou orphelines dans le cluster.
- Il s'assure que les objets qui ne sont plus nécessaires (par exemple, des pods terminés, des jobs complétés, ou des ressources dépendantes d'un objet supprimé) sont supprimés pour éviter l'accumulation de ressources inutiles et préserver la santé du cluster.
- Points clés à retenir
  - Le Garbage Collector est essentiel pour la gestion automatisée du cycle de vie des ressources et la prévention des fuites de ressources dans le cluster.
  - Il s'appuie sur les OwnerReferences pour déterminer quelles ressources doivent être supprimées en cascade lors de la suppression d'un objet parent.
  - Il existe différentes politiques de suppression : foreground, background, orphan.
  - Le comportement peut être personnalisé via les finalizers et la configuration des OwnerReferences.

# GC Controller - Comportement

- **Suppression en cascade (OwnerReferences)** : Le Garbage Collector utilise le champ ownerReferences des objets Kubernetes pour déterminer les relations de dépendance. Lorsqu'un objet parent (owner) est supprimé, le contrôleur vérifie les objets enfants (dependents) qui lui sont liés. Selon la politique de suppression (foreground, background, orphan), il supprime les enfants ou les détache du parent.
  - **Foreground deletion** : Le parent est marqué comme "en suppression" (deletionTimestamp et finalizer), puis les enfants sont supprimés avant que le parent ne disparaisse complètement.
  - **Background deletion** : Le parent est supprimé immédiatement, et le Garbage Collector supprime ensuite les enfants de façon asynchrone.
  - **Orphaning** : Les enfants sont conservés, mais leur référence au parent est supprimée.
- **Nettoyage des ressources terminées** : Il supprime automatiquement les pods terminés, les jobs complétés, les volumes persistants dynamiquement provisionnés avec une politique de rétention Delete, les CertificateSigningRequests expirés, et d'autres ressources temporaires.
- **Gestion des containers et images** : Le kubelet effectue également un garbage collection local des conteneurs et images non utilisés pour libérer de l'espace disque.
- **Suppression des nœuds disparus** : Si un nœud est supprimé de l'infrastructure (cloud ou on-premises), le Garbage Collector veille à ce que l'objet Node soit également supprimé du cluster.

The image features a light gray background with several decorative circles of various colors (pink, purple, blue, orange, teal) scattered in the corners. The word "Kubectl" is centered in a dark purple font.

# Kubectl



# Anatomie d'une commande kubectl

- **kubectl [commande] [TYPE] [NOM] [options]**
  - **commande** : l'action à effectuer (ex : get, describe, apply, delete, etc.)
  - **TYPE** : le type de ressource ciblée (ex : pod, deployment, service, etc.), au singulier, pluriel ou abréviation (po pour pod, svc pour service...)
  - **NOM** : le nom de la ressource (optionnel, si vous voulez cibler une ressource précise)
  - **options** : des flags ou options supplémentaires (ex : -n pour le namespace, -o pour le format de sortie, etc.)
- **Quelques exemples**
  - *kubectl api-resources* : liste les services de l'API
  - *kubectl explain <type de ressource>* : expliquer un type de ressource (debug)
  - *kubectl get pods -n mynamespace* : liste tous les pods de l'espace de nommage
  - *kubectl get all -A* : liste toutes les ressources dans tous les espaces de nommage
  - *kubectl apply -f fichier.yaml* : applique une configuration depuis un fichier

# Avant de commencer

- Clôner le repo git <https://github.com/akoudri/microservices-training.git>

- Activer la complétion des commandes kubernetes dans bash

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

- Créer un espace de nom qui vous est dédié et en faire votre espace de nom par défaut

```
kubectl create ns <votre-nom>
```

```
kubectl config set-context --current --namespace=<votre-nom>
```

# Travaux Pratiques

- Création d'un pod nommé hw qui expose le port 80

```
kubectl run hw --image=karthequian/helloworld:black --port=80
```

- Création d'un pod nommé pg qui expose le port 6543 et qui spécifie des variables d'environnement

```
kubectl run pg --image=postgres --port=5432 --env="POSTGRES_DB=training" \  
--env="POSTGRES_USER=backend" --env="POSTGRES_PASSWORD=backend"
```

- Exposition du pod hw

```
kubectl expose pod hw --type=NodePort --port=80
```

- Lister les ressources de l'API Server

```
kubectl api-resources
```

**Exercice:** Afficher la page du service hw; quels problèmes voyez-vous ?

# Quelques liens utiles

- <https://kubernetes.io/fr/docs/reference/kubectl/quick-reference/>
- <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, purple, and orange circles. The bottom-right corner is decorated with pink, purple, and teal circles. The title 'ReplicaSet Controller' is centered in a dark purple font.

# ReplicaSet Controller

# Limites du déploiement manuel

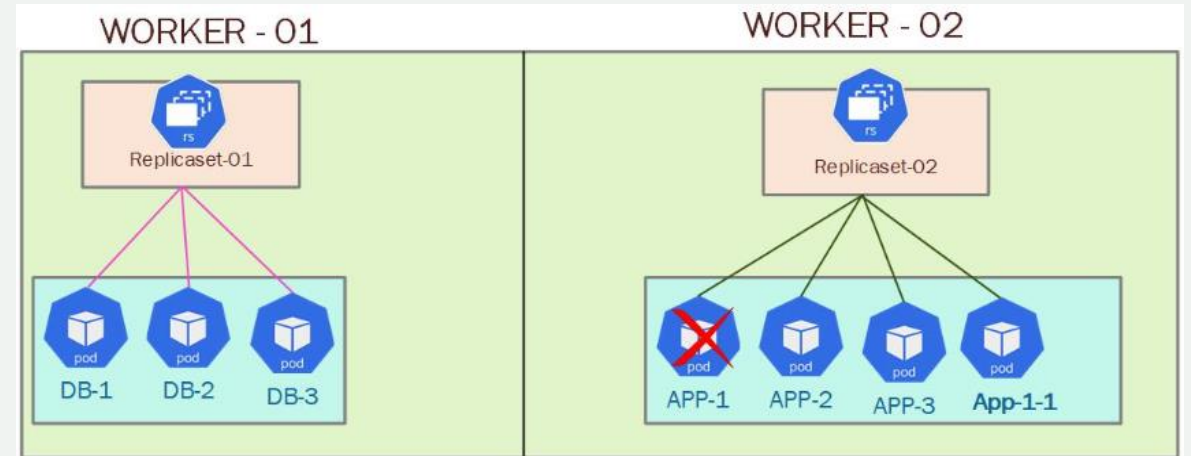
- **Absence d'auto-réparation:** Si le Pod crash, est supprimé accidentellement, ou si son nœud hôte tombe en panne, Kubernetes ne le redémarrera pas automatiquement. Vous devrez le recréer manuellement.
  - Exemple : Un Pod exécutant une base de données devient indisponible après une panne matérielle → Données inaccessibles jusqu'à intervention manuelle.
- **Impossible de scaler l'application:** Augmenter ou réduire le nombre d'instances de votre application nécessite de créer/supprimer manuellement des Pods, ce qui est chronophage et sujet aux erreurs.
- **Pas de haute disponibilité:** Un seul Pod signifie un "single point of failure". Aucune tolérance aux pannes ni équilibrage de charge.

# Limites du déploiement manuel

- **Gestion manuelle des mises à jour:** Mettre à jour l'image ou la configuration d'un Pod nécessite de le supprimer et de le recréer manuellement, entraînant un downtime.
- **Risque de dérive de configuration:** Si vous modifiez manuellement un Pod en cours d'exécution (ex : via `kubectl edit`), ces changements seront perdus si le Pod est recréé.
- **Gestion inefficace des ressources:** Vous risquez de surprovisionner (trop de Pods inactifs) ou sous-provisionner (trop peu de Pods pour la charge), car Kubernetes ne peut pas ajuster automatiquement le nombre de réplicas.

# ReplicaSet Controller

- Le *ReplicaSet Controller* est un composant du plan de contrôle de Kubernetes chargé de garantir qu'un nombre spécifié de Pods identiques (réplicas) sont toujours en cours d'exécution dans le cluster.
- Le *ReplicaSet Controller* veille en permanence à ce que le nombre voulu de Pods soit toujours disponible dans le cluster, en créant ou supprimant des Pods selon les besoins, pour garantir la stabilité et la haute disponibilité de tes applications Kubernetes.
- Le *ReplicaSet Controller* est essentiel pour assurer la tolérance aux pannes, la scalabilité et la gestion automatisée du cycle de vie des Pods dans Kubernetes.
- Bien que l'on puisse créer un ReplicaSet seul, il est souvent utilisé indirectement via un Deployment, qui ajoute la gestion des mises à jour et des rollbacks.





# ReplicaSet Controller - Fonctionnement

- **Boucle de réconciliation**

Le ReplicaSet Controller surveille en continu les objets ReplicaSet et les Pods associés. Si le nombre réel de Pods correspondant au sélecteur du ReplicaSet diffère du nombre désiré (spec.replicas), il agit automatiquement :

- S'il manque des Pods : il demande la création de nouveaux Pods à partir du template fourni.
- S'il y en a trop : il demande la suppression des Pods en excès.

- **Gestion par labels**

Le ReplicaSet utilise un sélecteur de labels pour déterminer quels Pods il doit gérer. Seuls les Pods dont les labels correspondent au sélecteur du ReplicaSet sont pris en charge.

- **Autoscaling et modification dynamique**

Le nombre de réplicas peut être modifié à la volée (via `kubectl scale` ou en éditant le ReplicaSet). Le controller ajuste alors automatiquement le nombre de Pods pour atteindre le nouvel objectif.

# ReplicaSet Controller - Exercices

- Observer et commenter le contenu du fichier hw-rs.yaml
- Exécuter ce fichier avec la commande suivante

```
kubectl apply -f hw-rs.yaml
```

- Lister les pods qui ont été créés; que remarquez-vous sur le nom des pods ?

```
kubectl get pods
```

- Supprimer un des pods

```
kubectl delete po/<nom-du-pod>
```

- Relister les pods; que remarquez-vous ?
- Scaler le replicaSet

```
kubectl scale rs hw-rs --replicas=5
```

- Relister les pods; que remarquez-vous ?
- Exposer le replicaSet et afficher le résultat

```
kubectl expose rs hw-rs --type=NodePort --port=80
```

# ReplicaSet Controller - Avantages

- **Haute disponibilité**

Si un Pod géré par le ReplicaSet échoue ou est supprimé, le controller le remplace immédiatement, assurant ainsi la disponibilité continue de l'application.

- **Scalabilité**

Il permet d'augmenter ou de réduire facilement le nombre d'instances d'une application pour répondre à la charge.

- **Gestion centralisée**

Toute modification du template du ReplicaSet s'applique à tous les nouveaux Pods créés.

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, orange, and purple circles. The bottom-right corner is decorated with pink, purple, and teal circles.

# Cycle de vie – Partie 1

## Gestion des ressources

# Impératif vs Déclaratif

- L'approche impérative décrit le "comment". On dit explicitement à kubernetes ce qu'il faut exécuter dans les moindres détails
- **Caractéristiques de l'approche impérative**
  - Rapide et simple pour des actions ponctuelles ou des tests.
  - Idéal pour l'expérimentation, le dépannage ou les modifications immédiates.
  - Ne nécessite pas de connaissances approfondies en YAML ou en structure d'objets Kubernetes.
- **Limites de l'approche impérative**
  - Pas de traçabilité ni de versioning : aucune trace de l'état désiré, ni d'historique des modifications.
  - Non reproductible : pour retrouver l'état du cluster, il faut se souvenir ou documenter toutes les commandes exécutées.
  - Dépend du contexte : chaque action doit être répétée à la main sur chaque environnement.
  - Maintenance difficile : impossible d'automatiser ou de revoir les changements facilement.
  - Limité au « comment » : vous guidez Kubernetes sur les actions à effectuer, mais il ne sait pas ce que vous voulez obtenir à la fin.

# Impératif vs Déclaratif

- L'approche déclarative décrit le "quoi". On dit explicitement à kubernetes quels sont les objectifs (états) à atteindre; kubernetes se charge ensuite d'exécuter les actions appropriées par atteindre ces objectifs.
- **Caractéristiques de l'approche déclarative**
  - Source de vérité : le fichier YAML décrit l'état attendu du cluster.
  - Versionnable : les fichiers peuvent être stockés dans Git, facilitant l'audit, la revue et la collaboration.
  - Reproductible : vous pouvez recréer l'infrastructure sur n'importe quel cluster à partir des mêmes fichiers.
  - Automatisable : idéal pour l'Infrastructure as Code, CI/CD et GitOps.
  - Idempotent : appliquer plusieurs fois le même fichier ne crée pas de doublons, Kubernetes gère la convergence.
  - Basé sur le « quoi » : vous exprimez le résultat attendu, Kubernetes se charge de trouver le chemin pour y arriver.
  - Permet de gérer des configurations complexes et de grande échelle.
  - Facilite la gestion multi-environnements (dev, test, prod).
  - Les changements sont visibles et réversibles via les outils de gestion de version.

# Gestion du cycle de vie

- **kubectl create**

- Permet de créer une ressource à partir d'un fichier YAML ou JSON décrivant l'état désiré de la ressource.
  - Crée la ressource si elle n'existe pas.
  - Non idempotent : échoue si la ressource existe déjà.
  - Idéal pour une première création ou des ressources ponctuelles.
- Exemple: *kubectl create -f ma-ressource.yaml*

- **kubectl replace**

- Remplace une ressource existante par une nouvelle définition fournie dans un fichier YAML ou JSON.
  - Nécessite que la ressource existe déjà.
  - Remplace l'intégralité de la définition : tout champ omis dans le YAML sera supprimé de la ressource.
  - Utile pour appliquer des modifications globales, mais attention à ne pas effacer des champs non précisés dans le fichier.
- Exemple: *kubectl replace -f ma-ressource.yaml*

# Gestion du cycle de vie

- **kubectl delete**

- Supprime une ou plusieurs ressources, soit par nom, par label, soit à partir d'un fichier YAML/JSON.
  - Libère les ressources du cluster.
  - Peut cibler précisément des ressources par nom, label ou fichier.
  - Permet de nettoyer ou de réinitialiser un environnement.
- Exemple: *kubectl delete -f ma-ressource.yaml*

- **kubectl apply**

- Elle permet de créer ou mettre à jour des objets (Pods, Deployments, Services, etc.) à partir de fichiers de configuration (YAML ou JSON) qui décrivent l'état désiré du cluster.
- Exemple: *kubectl apply -f ma-ressource.yaml*
- Avantages par rapport à create/replace
  - Souple et sécurisé : pas besoin de supprimer ou de remplacer l'objet en entier, seules les différences sont appliquées.
  - Recommandé pour l'Infrastructure as Code : facilite le versioning, la reproductibilité et la maintenance des configurations.
  - Moins de risques d'erreur : contrairement à replace, il ne supprime pas les champs non présents dans le fichier mais existants dans la ressource, sauf si explicitement demandé.



# kubectl apply - Fonctionnement

- **Création** : Si la ressource n'existe pas encore dans le cluster, kubectl apply la crée à partir du fichier fourni.
- **Mise à jour** : Si la ressource existe déjà, kubectl apply compare l'état actuel à l'état désiré décrit dans le fichier : il applique uniquement les changements nécessaires, sans supprimer ni réinitialiser les champs non modifiés.
- **Idempotence** : Vous pouvez appliquer plusieurs fois le même fichier sans créer de doublons ni provoquer d'erreurs.
- **Gestion de l'historique** : kubectl apply enregistre la dernière configuration appliquée dans une annotation de l'objet, ce qui permet de calculer les différences lors des prochaines applications.

# kubectl apply - Options

- **--recursive** : applique tous les fichiers d'un dossier et de ses sous-dossiers.
  - *kubectl apply -f dossier/*
- **-l** ou **--selector** : applique uniquement aux ressources correspondant à certains labels.
- **--prune** : supprime les ressources qui ne sont plus présentes dans les fichiers appliqués (fonctionnalité avancée).
- **--dry-run** : simule l'application sans modifier le cluster, utile pour valider les changements.

# Anatomie d'un fichier de ressources YAML

- Un fichier de ressource Kubernetes au format YAML repose sur quatre sections fondamentales : *apiVersion*, *kind*, *metadata* et *spec*.
- Ces sections sont obligatoires pour que Kubernetes puisse interpréter, créer et gérer correctement la ressource.
  - **apiVersion**: Indique la version de l'API Kubernetes à utiliser pour cette ressource; garantit la compatibilité entre la ressource décrite et le serveur Kubernetes.
  - **kind**: Spécifie le type d'objet Kubernetes à gérer; Permet à Kubernetes de savoir comment interpréter et gérer la ressource (Pod, Service, Deployment, ConfigMap, etc.).
  - **metadata**: Fournit les informations d'identification et de catégorisation de la ressource; permet de retrouver, organiser et référencer la ressource dans le cluster.
  - **spec**: Décrit l'état désiré de la ressource; c'est la section la plus détaillée : elle précise la configuration et le comportement attendus (conteneurs à lancer, ports, nombre de réplicas, stratégies de mise à jour, etc.).

# Exercices

- Analysez et exécutez le fichier hw-rs.yaml
- Éditez le runtime – Qu'observez-vous ?
- Comparez les fichiers hw-pod.yaml et hw-rs.yaml; qu'observez-vous ? **Quelles sont les limites des replicaSets** ?

```
apiVersion: v1
kind: Pod
metadata:
  name: hw
spec:
  containers:
  - name: helloworld
    image: karthequian/helloworld:black
    ports:
    - containerPort: 80
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: hw-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hw
  template:
    metadata:
      labels:
        app: hw
    spec:
      containers:
      - name: helloworld
        image: karthequian/helloworld:black
        ports:
        - containerPort: 80
```

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner is decorated with a small teal circle, a medium pink circle, a tiny dark purple dot, a medium purple circle, a large pink circle, and a large teal circle.

# Deployment Controller

# Limites des ReplicaSets

- **Pas de gestion des mises à jour:** Les ReplicaSets ne gèrent pas les rolling updates : pour modifier une application (ex : changer l'image d'un conteneur), vous devez créer manuellement un nouveau ReplicaSet et supprimer l'ancien, ce qui entraîne un temps d'arrêt (downtime).
- **Pas de rollback:** En cas de problème avec une nouvelle version, impossible de revenir à une configuration précédente automatiquement. Vous devez restaurer manuellement l'ancien ReplicaSet.
- **Pas de versioning:** Les ReplicaSets ne conservent pas d'historique des modifications, rendant le débogage et la traçabilité complexes.
- **Gestion manuelle de la scalabilité:** Bien que les ReplicaSets maintiennent le nombre de réplicas, ils n'intègrent pas de mécanisme d'autoscaling basé sur la charge CPU/mémoire (nécessite un HorizontalPodAutoscaler séparé).
- **Complexité opérationnelle:** Toute modification nécessite une intervention manuelle, ce qui augmente les risques d'erreurs en production.

# Deployment Controller

- Le Deployment Controller est un composant natif de Kubernetes chargé d'orchestrer le déploiement, la gestion, la mise à jour et la montée/descente en charge (scaling) des applications conteneurisées via l'objet Deployment.
- Il s'appuie sur les ReplicaSets pour garantir la haute disponibilité et la résilience des Pods, tout en ajoutant des fonctionnalités avancées de gestion du cycle de vie applicatif.
- Le Deployment Controller offre une gestion déclarative, fiable et automatisée du cycle de vie des applications, bien au-delà de ce que propose un ReplicaSet seul :
  - **Sécurité et robustesse** : auto-réparation, tolérance aux pannes, zéro downtime lors des mises à jour.
  - **Productivité** : simplifie la gestion des versions, des rollbacks et des montées en charge.
  - **Traçabilité** : chaque changement est historisé et réversible.
  - **Automatisation** : s'intègre parfaitement dans des pipelines CI/CD et des stratégies GitOps.

# Deployment Controller - Fonctionnement

- **Déclaration de l'état désiré**

L'utilisateur décrit l'état attendu de l'application (nombre de Pods, image, configuration...) dans un objet Deployment. Le Deployment Controller surveille en continu l'état réel du cluster et agit pour le faire converger vers l'état désiré.

- **Gestion des ReplicaSets et Pods**

Le Deployment Controller crée et gère automatiquement les ReplicaSets associés, qui eux-mêmes gèrent les Pods. Il ne faut pas modifier directement les ReplicaSets créés par un Deployment.

- **Mises à jour progressives (rolling updates)**

Lorsqu'une modification est apportée (nouvelle image, changement de configuration...), le Deployment Controller crée un nouveau ReplicaSet et remplace progressivement les anciens Pods par les nouveaux, sans interruption de service.

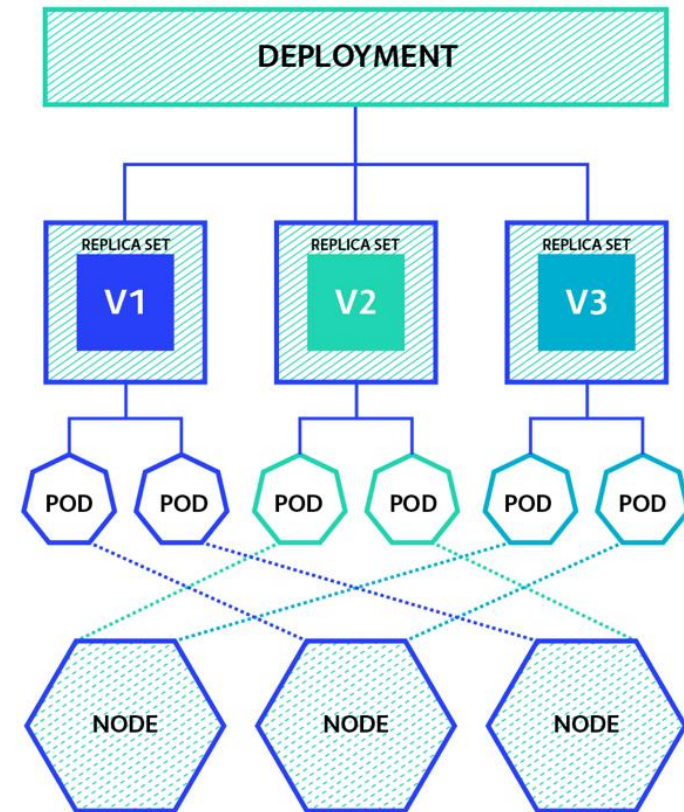
- **Rollbacks automatiques**

Si une mise à jour échoue ou provoque une instabilité, le Deployment Controller permet de revenir rapidement à une version antérieure stable (rollback).



# Deployment Controller - Fonctionnement

- **Stratégie de mise à jour** : Les déploiements vous permettent de définir une stratégie de mise à jour pour passer de l'état actuel au nouvel état
  - RollingUpdate: la stratégie par défaut, où les mises à jour sont déployées de manière incrémentale
  - Recreate (où tous les Pods existants sont tués avant que de nouveaux Pods ne soient créés).
- **Mise à l'échelle** : Les déploiements peuvent être facilement mis à l'échelle en modifiant le nombre de répliques dans la configuration.
- **Auto-réparation** : Si un pod tombe en panne, le déploiement le remplacera.



# Deployment Controller - Fonctionnement

- **Scaling**

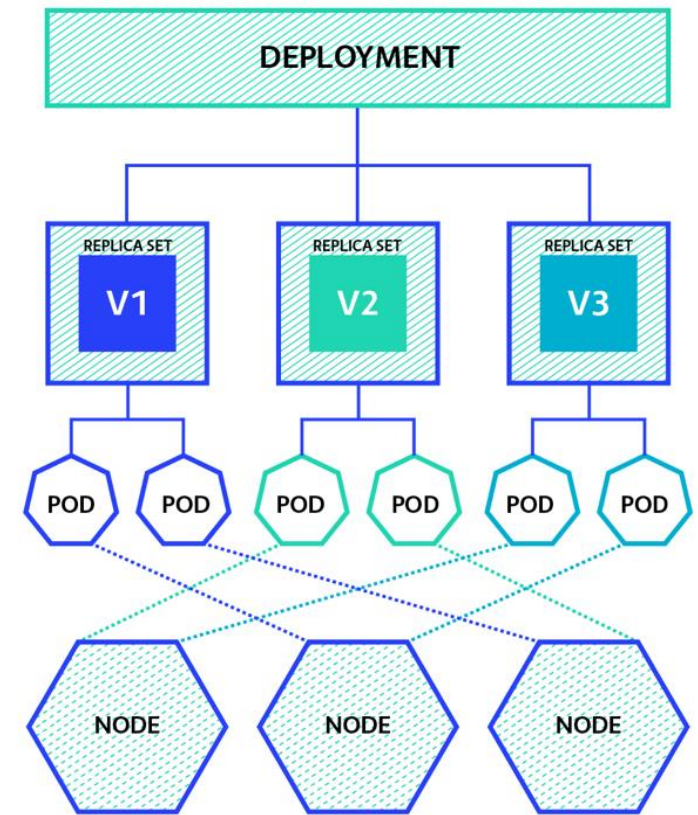
Il ajuste dynamiquement le nombre de Pods en fonction de la charge ou des besoins, soit manuellement, soit via des outils d'autoscaling.

- **Historique et traçabilité**

Chaque modification du Deployment crée une nouvelle révision, permettant de suivre l'historique des déploiements et de restaurer facilement un état antérieur.

- **Pause et reprise des déploiements**

Il est possible de mettre en pause un déploiement pour appliquer plusieurs changements, puis de le reprendre pour déclencher un nouveau rollout.



# Deployment Controller - Avantages

- Le Deployment Controller est le **chef d'orchestre de la gestion applicative** dans Kubernetes.
- Il automatise le déploiement, la mise à jour, le scaling et la résilience des applications, tout en assurant une expérience utilisateur fluide et sans interruption.
- C'est le composant recommandé pour toute application en production nécessitant robustesse, évolutivité et facilité de gestion.

# Déploiement: Cycle de vie

- **Création** : Le déploiement crée un ReplicaSet. Le ReplicaSet crée des Pods basés sur le modèle Pod (template).
- **Mise à jour** : si vous mettez à jour le modèle de pod dans le déploiement, celui-ci crée un nouvel ensemble de répliques basé sur le modèle mis à jour et commence à déplacer les pods de l'ancien ensemble de répliques vers le nouveau conformément à la stratégie de mise à jour définie.
- **Mise à l'échelle** : Vous pouvez faire évoluer un déploiement à la hausse ou à la baisse. Pour ce faire, il suffit de modifier le nombre de répliques dans la spécification.
- **Suppression** : Si vous supprimez le déploiement, il supprime les ensembles de répliques qu'il a créés, ce qui entraîne la suppression des pods.

# Deployment Controller - Commandes

- `kubectl create deployment --image=nginx nginx` # Crée un déploiement
- `kubectl scale deployment nginx --replicas=5` # Rescale un déploiement
- `kubectl set image deployment nginx nginx=nginx:alpine3.21` # Change l'image d'un conteneur
- `kubect expose deployment nginx --port=80` # Expose un déploiement
- `kubectl edit deployment nginx` # Edite un déploiement
- `kubectl rollout history deployment nginx` # Affiche les révisions
- `kubectl rollout undo deployment nginx` # Revenir à la version précédente

# Fichier YAML : En cas de doutes

- Si vous avez des doutes sur la syntaxe, vous avez la possibilité de générer rapidement un fichier que vous pourrez adapter par la suite
- *kubectl create deployment --image=nginx --replicas=2 --dry-run=client -o yaml nginx*
  - **--dry-run=client**  
Demande à kubectl de ne pas créer le Deployment dans le cluster, mais de générer uniquement la définition de la ressource côté client (sans contact avec l'API server).  
Cela permet de prévisualiser ou de générer le YAML sans rien déployer réellement.
  - **-o yaml**  
Demande à kubectl de formater la sortie au format YAML, plutôt qu'en JSON ou dans un format lisible par l'homme.

# Deployment Controller - Exercices

- Commenter et exécuter le fichier hw-deploy.yaml
- Lister l'ensemble des ressources; qu'observez-vous ?
- Modifier le nombre de répliques et appliquer de nouveau le fichier
- Lister l'ensemble des ressources; qu'observez-vous ?
- Exécuter la commande `kubectl get deploy hw -o yaml`
- Observer les annotations; que remarquez-vous ?

# Rollout

- Le rollout dans Kubernetes désigne le processus de déploiement, de mise à jour ou de restauration (rollback) d'une application gérée par un contrôleur tel que Deployment, DaemonSet ou StatefulSet.
- Le rollout permet d'appliquer de nouvelles versions du code ou de la configuration de façon contrôlée, tout en assurant la disponibilité et la continuité du service.
- Le rollout est déclenché à chaque modification du template de Pod d'un Deployment (changement d'image, de variables d'environnement, etc.).
- Kubernetes gère alors la création progressive des nouveaux Pods et la suppression des anciens, en s'appuyant sur des ReplicaSets pour orchestrer la transition.



# Rollout - Stratégie Rolling Update

- **Principe** : C'est la stratégie par défaut. Les Pods sont remplacés progressivement : Kubernetes lance de nouveaux Pods avec la nouvelle version, attend qu'ils soient prêts, puis supprime les anciens, jusqu'à ce que tous les Pods soient mis à jour.
- **Avantages** : Pas d'interruption de service, continuité assurée, possibilité de rollback rapide.
- **Paramètres clés** :
  - maxUnavailable : nombre maximal de Pods indisponibles pendant la mise à jour.
  - maxSurge : nombre maximal de nouveaux Pods créés au-delà du nombre de répliques souhaité.
- **Utilisation** : Adaptée à la majorité des applications nécessitant une haute disponibilité.

# Rollout - Stratégie Recreate

- **Principe** : Tous les anciens Pods sont supprimés avant de créer les nouveaux.
- **Avantages** : Évite les conflits entre versions (utile si l'application ne supporte pas plusieurs versions en parallèle).
- **Inconvénient** : Provoque une interruption de service pendant la transition.
- **Utilisation** : À privilégier pour les applications qui ne tolèrent pas la coexistence de plusieurs versions ou qui nécessitent un environnement propre.

# Rollout - Commandes

| Commande                                                     | Description                                                            |
|--------------------------------------------------------------|------------------------------------------------------------------------|
| <code>kubectl rollout status deployment/&lt;name&gt;</code>  | Affiche le statut du rollout en cours ou terminé.                      |
| <code>kubectl rollout history deployment/&lt;name&gt;</code> | Affiche l'historique des déploiements (révisions).                     |
| <code>kubectl rollout undo deployment/&lt;name&gt;</code>    | Annule le dernier rollout (rollback vers la version précédente).       |
| <code>kubectl rollout pause deployment/&lt;name&gt;</code>   | Met en pause le rollout en cours (utile pour valider étape par étape). |
| <code>kubectl rollout resume deployment/&lt;name&gt;</code>  | Reprend un rollout précédemment mis en pause.                          |
| <code>kubectl rollout restart deployment/&lt;name&gt;</code> | Redémarre les Pods du déploiement (déclenche un nouveau rollout).      |

# Rollout - Exercices

- Appliquer le fichier hw-service.yaml
- Modifier le fichier de configuration afin de changer l'image pour karthequian/helloworld:blue
- Appliquer la modification et vérifier qu'elle a été appliquée correctement
- Afficher l'historique du déploiement: *kubectl rollout history deploy/helloworld*
- Revenir à la version précédente: *kubectl rollout undo deploy/helloworld*
  - Vérifier l'état du changement: *kubectl rollout status deploy/helloworld*
  - Vérifier que le rollback a bien eu lieu

# Rollout – Exercices (Cont'd)

- Changer de nouveau l'image pour karthequian/helloworld:red
- Lister l'historique des révisions
- Revenir à la première révision: *kubectl rollout undo deploy/helloworld--to-revision=1*
  - Vérifier le résultat
- Faire un rollback en mode simulation:
  - *kubectl rollout undo deploy/helloworld --dry-run=server -o yaml*
  - Quel est l'intérêt de cette commande ?

# Rollout - Précisions

- En listant l'historique des révisions, vous avez pu constater qu'il y a une colonne "CHANGE-CAUSE" toujours vide; cette colonne sert à documenter les différentes révisions.
- Il y a deux manières de renseigner cette colonne:
  - Avec la commande annotate
    - *kubectl annotate deploy/hw kubernetes.io/change-cause="Mise à jour de l'image vers blue" --overwrite*
  - En éditant le fichier yaml

```
metadata:  
  annotations:  
    kubernetes.io/change-cause: "Mise à jour de l'image vers blue"
```

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, orange, and purple circles. The bottom-right corner is decorated with pink, purple, and teal circles. The title is centered in a dark purple font.

# Précisions sur les Pods

# Configuration d'un conteneur

- Pour passer des arguments à un conteneur lors de la création d'un Pod, il faut utiliser les champs `command` et `args` dans la spécification du conteneur du manifeste YAML.
  - `command` : définit la commande à exécuter (équivalent à `ENTRYPOINT` dans Docker).
  - `args` : définit la liste des arguments à passer à la commande (équivalent à `CMD` dans Docker).
- Si vous ne spécifiez que `args`, la commande par défaut de l'image sera utilisée avec les nouveaux arguments. Si vous spécifiez les deux, la commande et ses arguments remplaceront le comportement par défaut du conteneur.
- Vous pouvez, comme pour les conteneurs docker, passer des variables d'environnement avec la propriété `env` (tableau de clé / valeur)
- Exemple: `cmd-args-example.yaml`



# Plusieurs conteneurs dans un même pod

- L'utilisation de plusieurs conteneurs dans un même Pod est particulièrement utile lorsque les conteneurs doivent travailler en étroite collaboration.
- Il est essentiel de comprendre que les conteneurs d'un même Pod sont étroitement liés et partagent le même cycle de vie, le même réseau et le même stockage. Cette conception doit être utilisée lorsqu'il y a un besoin clair d'interaction étroite et d'interdépendance entre les conteneurs.
- **Avantages**
  - **Partage des ressources** : Les conteneurs d'un même Pod partagent le même espace réseau, ce qui signifie qu'ils peuvent communiquer entre eux via l'hôte local et qu'ils ont la même adresse IP. Ils peuvent également partager les mêmes volumes de stockage.
  - **Cycle de vie interdépendant** : Les conteneurs d'un pod sont démarrés, arrêtés et redémarrés ensemble.
  - **Communication simplifiée** : Les conteneurs peuvent facilement communiquer entre eux, ce qui simplifie la communication inter-processus.
  - **Efficacité des ressources** : L'exécution de conteneurs étroitement couplés dans le même module peut être plus efficace en termes de ressources que leur exécution dans des modules distincts.
- Exemple: multiple-containers.yaml

# Plusieurs pods dans un même conteneur

- Tous les conteneurs d'un pod partagent le même espace de noms réseau : ils ont la même adresse IP et le même espace de ports.
- Ils peuvent donc communiquer entre eux via localhost (127.0.0.1) et les ports exposés par chaque conteneur.
  - Exemple : si un conteneur écoute sur le port 8080, l'autre peut y accéder via localhost:8080
- De fait, deux conteneurs au sein d'un même pod ne peuvent pas écouter sur le même port:
  - En effet, tous les conteneurs d'un pod partagent exactement le même espace réseau : même adresse IP et, surtout, même espace de ports.
  - Cela signifie que si un conteneur écoute sur un port donné (par exemple le port 8080), aucun autre conteneur du même pod ne peut écouter sur ce même port, car il y aurait un conflit d'allocation de port au niveau du système d'exploitation

# Conteneur d'initialisation

- Les initContainers sont une fonctionnalité de Kubernetes qui vous permet d'avoir des conteneurs qui s'exécutent jusqu'à la fin avant que les conteneurs d'application principaux ne démarrent.
- Ces conteneurs font partie du même Pod que votre conteneur d'application principal, mais ils s'exécutent et se terminent dans l'ordre avant que les conteneurs principaux ne démarrent.
- Cas d'usage:
  - **Préparation d'un environnement de travail** : Mise en place des aspects de l'environnement dont votre application a besoin pour fonctionner, tels que les fichiers de configuration, la transformation des données, la récupération des dépendances, etc.
  - **Attente des dépendances** : S'assurer que les services, les bases de données ou d'autres ressources sont en place et prêts avant le démarrage de l'application.
  - **Sécurité et contrôle d'accès** : Mise en place du contexte de sécurité, des autorisations et d'autres conditions préalables qui doivent être remplies avant le démarrage de l'application principale.
- **Exercice**: Étudier l'exemple du fichier elasticsearch.yaml

# Pods statiques

- **Définition** : Un pod statique est un pod géré directement par le **kubelet** sur un nœud spécifique, indépendamment du plan de contrôle Kubernetes (API server, scheduler, etc.).
- **Gestion** : Le kubelet surveille un répertoire local (généralement **/etc/kubernetes/manifests**) sur chaque nœud ; tout fichier manifeste de pod déposé dans ce répertoire sera automatiquement créé et maintenu par le kubelet.
- **Visibilité** : Bien que non contrôlés par l'API server, les pods statiques sont visibles via kubectl car le kubelet crée un "mirror pod" sur l'API server à des fins de visibilité. Ce mirror pod est une simple représentation : il ne peut pas être géré (modifié/supprimé) via l'API server.
- **Portée** : Un pod statique n'est lié qu'au nœud où son manifeste est présent. Pour exécuter un pod statique sur plusieurs nœuds, il faut copier le manifeste sur chaque nœud concerné.

# Pods statiques : Cas d'usage

- **Bootstrapping du cluster:** Les composants critiques du plan de contrôle Kubernetes (kube-apiserver, kube-controller-manager, kube-scheduler, etcd) sont souvent lancés en tant que pods statiques lors du démarrage du cluster (notamment avec kubeadm).
- **Workloads spécifiques à un nœud:** Pour des agents de monitoring, de logging, ou des applications nécessitant un accès privilégié au système du nœud ou à du matériel spécifique.
- **Résilience:** Les pods statiques continuent de fonctionner même si l'API server est indisponible, ce qui garantit la disponibilité des composants critiques du cluster.
- **Cas d'urgence ou de maintenance:** Permet de déployer rapidement un pod sur un nœud, même si le plan de contrôle est partiellement ou totalement hors service.

# Pods statiques - Limitations

- **Pas de gestion centralisée** : Les pods statiques doivent être gérés manuellement sur chaque nœud (création, mise à jour, suppression).
- **Pas de scheduling** : Ils ne sont pas planifiés par le scheduler Kubernetes : ils tournent uniquement sur le nœud où le manifeste est présent.
- **Pas de support pour certaines fonctionnalités** : Impossible de référencer d'autres objets API (ServiceAccount, ConfigMap, Secret, etc.) dans le spec du pod statique.
- **Pas de montée en charge automatique** : Contrairement à un DaemonSet, il n'y a pas de gestion automatique du déploiement sur plusieurs nœuds.

# Pods Statiques de Kubernetes

- Le dossier **/etc/kubernetes/manifests** joue un rôle central dans l'architecture Kubernetes, notamment sur les nœuds de contrôle (control plane).
  - Ce répertoire contient les manifests YAML de type Pod dits "static pods".
  - Il est surveillé en continu par le processus kubelet local : tout fichier ajouté, modifié ou supprimé dans ce dossier entraîne la création, la mise à jour ou la suppression du Pod correspondant sur le nœud.
  - Le chemin par défaut /etc/kubernetes/manifests est utilisé par kubeadm et la plupart des distributions Kubernetes, mais il peut être personnalisé via l'option --pod-manifest-path du kubelet ou dans sa configuration
- Sur un nœud de contrôle, vous trouverez généralement :
  - **kube-apiserver.yaml**: Définit le Pod statique du serveur d'API Kubernetes, point d'entrée de toutes les requêtes et cœur du cluster.
  - **kube-controller-manager.yaml**: Définit le Pod statique du controller manager, responsable de l'automatisation et de la gestion de l'état du cluster.
  - **kube-scheduler.yaml**: Définit le Pod statique du scheduler, qui décide sur quel nœud chaque Pod doit être exécuté.
  - **etcd.yaml**: Définit le Pod statique de la base de données etcd, qui stocke l'état persistant du cluster.

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, and a small dark purple circle. The top-right corner features a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner has a small teal circle, a small pink circle, a small dark purple circle, a medium pink circle, and a large teal circle. The text is centered in the middle of the slide.

# Cycle de vie – Partie 2

## Édition des ressources



# Editer des ressources - Set

- La famille de commandes *kubectl set* permet de modifier rapidement et plus précisément certains aspects d'une ressource, sans éditer tout le manifest YAML.
  - *kubectl set image deployment nginx nginx=nginx:alpine3.21*
  - *kubectl set env deployment nginx ENV=production*
  - *kubectl set resources deployment nginx --limits=cpu=200m,memory=512Mi*
- **Avantages :**
  - Rapide, ciblé, idéal pour des modifications ponctuelles ou automatisées.
  - Pas besoin de manipuler de fichiers YAML.
- **Limites :**
  - Ne permet de modifier que certains champs spécifiques (image, env, ressources, etc.)

# Editer des ressources - Edit

- La commande `kubectl edit` ouvre la ressource à modifier dans votre éditeur de texte préféré (défini par `$EDITOR` ou `$KUBE_EDITOR`, sinon vi par défaut).
  - Récupère la ressource depuis l'API server.
  - L'ouvre en YAML (ou JSON avec `-o json`).
  - À la sauvegarde, applique les modifications directement dans le cluster.
- **Avantages :**
  - Permet de modifier n'importe quel champ de la ressource, y compris les champs avancés.
  - Pratique pour des corrections rapides ou des tests.
- **Limites :**
  - Modifications manuelles : risque d'erreur de syntaxe.
  - Les changements ne sont pas versionnés par défaut (sauf si vous utilisez `--save-config`).
  - Moins adapté pour l'automatisation ou la gestion d'infrastructure as code

# Editer des ressources - Apply

- **Approche déclarative**
- **Étapes**
  1. Exporter la ressource (facultatif): `kubectl get deploy/nginx -o yaml > nginx.yaml`
  2. Modifier le fichier localement
  3. Appliquer les modifications avec `kubectl apply`
- **Avantages :**
  - Permet de versionner et de suivre les modifications (Git).
  - Adapté à la gestion d'infrastructure as code (IaC), CI/CD, GitOps.
  - Idéal pour des modifications complexes ou répétées.
- **Limites :**
  - Plus de manipulations (export, édition, apply).
  - Risque de désynchronisation si plusieurs personnes modifient la même ressource.

# Éditer des ressources - Autres

- **kubectl patch**

- Permet de modifier une ressource en place via un patch JSON ou YAML (stratégique, merge, ou JSON patch).
- *kubectl patch deployment/nginx -p '{"spec":{"replicas":5}}'*
- Utile pour les modifications ciblées, notamment en automatisation ou scripting.

- **API Kubernetes**

- Vous pouvez modifier les ressources via des appels directs à l'API REST Kubernetes (avec curl, httpie, Postman, etc.), ou via des bibliothèques clientes (Python, Go...).

# Editer des ressources

## - Comparatif

| Méthode                 | Usage principal                      | Avantages                                   | Limites                                              |
|-------------------------|--------------------------------------|---------------------------------------------|------------------------------------------------------|
| kubectl set             | Modif. ciblées<br>(image, env, etc.) | Rapide, simple,<br>scripting                | Champs limités,<br>pas de versionning                |
| kubectl edit            | Modif. manuelle<br>interactive       | Flexible, immédiat                          | Risque d'erreur,<br>pas d'IaC, pas<br>versionné      |
| Fichier YAML +<br>apply | Déclaratif, IaC,<br>CI/CD            | Versionnable,<br>traçable,<br>reproductible | Plus de<br>manipulations,<br>gestion des<br>conflits |
| kubectl patch           | Patch ciblé<br>(scripting, API)      | Précis,<br>automatisable                    | Syntaxe parfois<br>complexe                          |
| API Kubernetes          | Automatisation<br>avancée            | Contrôle total                              | Plus technique,<br>gestion manuelle                  |

# Editer des ressources – En pratique

- Utiliser `kubectl set` ou `kubectl edit` pour des modifications ponctuelles ou en phase de test.
- Privilégier l'édition de fichiers YAML versionnés + `kubectl apply` pour la production, l'automatisation et la traçabilité.
- `kubectl patch` et l'API sont utiles pour des besoins très ciblés ou automatisés.

# Séparateur de documents

- Le séparateur --- dans un fichier YAML indique le début d'un nouveau document.
- Son rôle principal est de permettre la définition de plusieurs documents YAML dans un même fichier, chaque document étant séparé par ce marqueur.
  - --- signale explicitement le début d'un document YAML. Si le fichier contient plusieurs documents (par exemple, plusieurs ressources Kubernetes à appliquer en une seule fois), chaque document commence par ---
  - Il permet de placer plusieurs objets (Pods, Services, Deployments, etc.) dans un seul fichier YAML, ce qui facilite la gestion groupée de ressources dans Kubernetes.
  - Le marqueur ... existe aussi pour signaler explicitement la fin d'un document, mais il est rarement utilisé; --- suffit à séparer les documents dans la plupart des cas
- **Question:** quels sont les intérêts du séparateur de documents ?

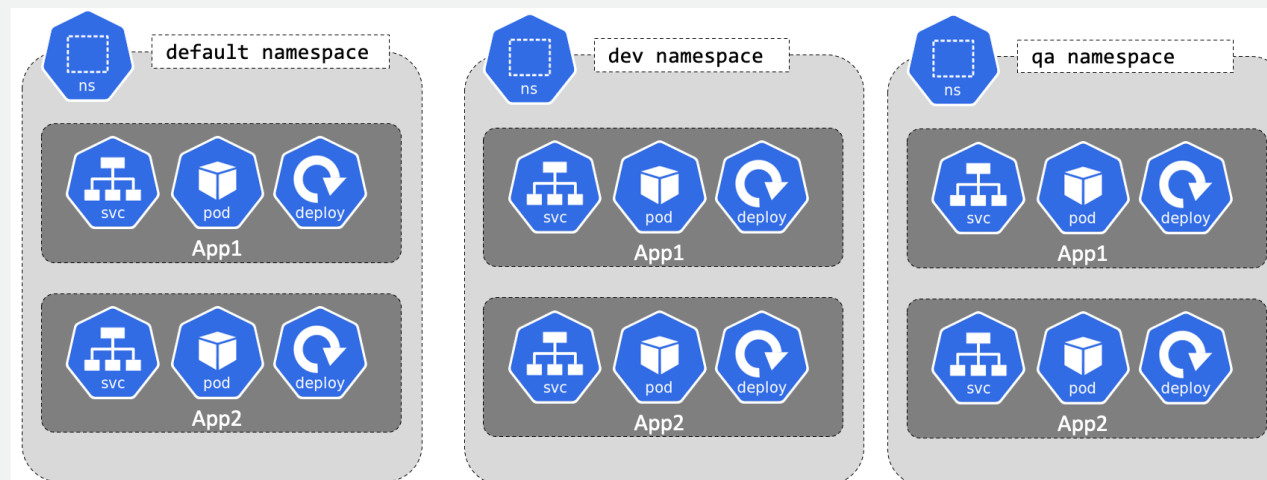
The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, orange, and purple circles. The bottom-right corner is decorated with pink, purple, and teal circles.

# Organisation des ressources



# Namespaces

- Un namespace est une division logique à l'intérieur d'un cluster Kubernetes : il permet de regrouper, isoler et organiser les ressources (Pods, Services, Deployments, etc.) dans des espaces distincts au sein du même cluster.
- Chaque ressource qui « vit » dans un namespace possède un nom unique dans cet espace, mais ce nom peut être réutilisé dans d'autres namespaces.
- Les namespaces sont essentiels pour organiser, isoler, sécuriser et gérer efficacement les ressources dans un cluster Kubernetes partagé ou complexe.



# Namespaces - Intérêts

- **Isolation des ressources** : Les namespaces permettent d'isoler les ressources entre différentes équipes, projets, environnements (développement, test, production), ou clients. Ainsi, les actions ou incidents dans un namespace n'impactent pas les autres.
- **Organisation** : Ils facilitent la gestion et la supervision des ressources dans des environnements complexes, en évitant la confusion et les conflits de noms
- **Contrôle d'accès (RBAC)** : Les namespaces permettent d'appliquer des politiques de sécurité et des droits d'accès spécifiques à chaque espace. Par exemple, une équipe de développement peut avoir des droits plus ouverts sur un namespace dev, tandis que l'équipe de production a des droits restreints sur le namespace prod.
- **Séparation multi-locataire** : Dans un cluster partagé, chaque équipe ou client peut disposer de son propre espace isolé, limitant les risques de fuite ou de conflit de ressources
- **Quotas et limites** : Il est possible de définir des quotas de ressources (CPU, mémoire, stockage) par namespace, évitant qu'un projet ne consomme toutes les ressources du cluster.

# Namespaces - Intérêts

- **Performance** : L'isolation par namespace réduit la charge sur l'API server lors de la gestion de grands clusters, car certaines opérations sont limitées à un namespace.
- **Gestion simplifiée** : Les namespaces permettent d'appliquer des politiques, des quotas, des stratégies réseau ou de sécurité à un groupe de ressources d'un seul coup, simplifiant la gestion quotidienne du cluster.
- **Nettoyage facilité** : Supprimer un namespace efface toutes les ressources qu'il contient, ce qui simplifie le nettoyage d'environnements temporaires ou de tests
- **Partage du cluster** : Un seul cluster Kubernetes peut héberger plusieurs environnements ou équipes, ce qui optimise l'utilisation des ressources et réduit les coûts d'infrastructure.
- **Évolutivité** : Les namespaces permettent de faire grandir le cluster en maintenant une organisation claire et des frontières entre les usages.

# Namespaces par défaut

- **default** : pour les objets sans namespace spécifié, espace de travail principal.
- **kube-system** : pour les composants internes de Kubernetes (DNS, contrôleurs...).
- **kube-public** : ressources accessibles publiquement, même sans authentification (usage rare).
- **kube-node-lease** : gestion des "leases" pour la détection rapide des nœuds défectueux.

# Namespaces - TP

- Analyser et exécuter le fichier de configuration hw-ns.yaml
- Lister l'ensemble des ressources – que remarquez-vous ?
- **Remarque:** La suppression d'un namespace entraîne la suppression de tout ce qui s'y trouve

# Labels

- Les labels (ou étiquettes) sont des paires clé/valeur attachées aux objets Kubernetes (Pods, Services, Deployments, Nodes, etc.).
- Ils servent à identifier, organiser, regrouper et sélectionner des ressources selon des critères arbitraires définis par l'utilisateur ou les outils.
- Syntaxe et règles
  - Clé/valeur :
    - La clé et la valeur sont sensibles à la casse.
    - La clé peut avoir un préfixe (optionnel) de type DNS, séparé par /. Ex : app.kubernetes.io/name.
    - La clé ne doit pas dépasser 63 caractères (253 avec préfixe) et doit commencer/finir par un caractère alphanumérique.
    - Les caractères autorisés sont : lettres, chiffres, tirets (-), underscores (\_), points (.).
  - Multiplicité :
    - Un objet peut avoir plusieurs labels.
    - Un même label peut être partagé par plusieurs objets.

# Labels - Intérêts

- **Organisation et regroupement:** Les labels permettent de regrouper logiquement des objets partageant une caractéristique (ex : tous les Pods d'une même application ou d'un même environnement).
  - Cela facilite la gestion, la supervision et l'automatisation dans des clusters complexes.
- **Filtrage et sélection:** Les labels sont utilisés dans les commandes kubectl et dans les sélecteurs (selectors) pour cibler précisément un ensemble d'objets.
  - Exemple: `kubectl get pods -l environment=production`
- **Opérations en masse:** Ils permettent d'appliquer des actions groupées (mise à jour, suppression, scaling...) sur tous les objets partageant un même label
  - Exemple: `kubectl delete pods -l app=web`
- **Gestion de l'infrastructure:** Les labels servent à piloter des comportements automatisés (scheduling, network policies, monitoring, etc.) ou à segmenter les ressources pour la sécurité et la conformité.
- **Interopérabilité et bonnes pratiques:** Un ensemble commun de labels facilite l'intégration avec des outils tiers et la standardisation des déploiements.

# Sélecteurs

- Les sélecteurs (selectors) sont des expressions qui permettent de cibler dynamiquement un ou plusieurs objets Kubernetes en fonction de leurs labels (étiquettes) ou de certains champs internes.
- Ils sont essentiels pour organiser, filtrer, automatiser et appliquer des opérations groupées sur les ressources du cluster.
- Utilisation dans les objets Kubernetes
  - Services : utilisent un label selector pour cibler les pods à inclure dans le service.
  - ReplicaSet/Deployment : le selector définit quels pods sont gérés par le contrôleur.
  - NetworkPolicy, Jobs, etc. : utilisent aussi des selectors pour cibler dynamiquement des groupes de ressources.



# Sélecteurs - Intérêts

- **Organisation dynamique** : Permet de regrouper ou d'isoler des ressources sans avoir à les nommer explicitement.
- **Automatisation** : Les controllers et services s'adaptent automatiquement aux changements de labels.
- **Opérations groupées** : Facilite les opérations de masse (mise à jour, suppression, monitoring...).
- **Flexibilité** : Les sélecteurs permettent des requêtes complexes et puissantes pour cibler précisément les ressources voulues.

# Sélecteurs de labels

- Ce sont les plus courants. Ils permettent de sélectionner des ressources en fonction de leurs paires clé/valeur de labels. Ils sont utilisés dans :
  - Les commandes kubectl (ex : lister des pods)
  - Les objets de type Service, ReplicaSet, Deployment, NetworkPolicy, etc.
- Sélecteurs basés sur l'égalité (Equality-based selectors)
  - *kubectl get pods -l app=nginx,env=production* #conjonction
- Sélecteurs basés sur un ensemble (Set-based selectors)
  - clé in (valeur1,valeur2)
  - clé notin (valeur1,valeur2)
  - clé (équivalent à exists : la clé existe)
  - !clé (la clé n'existe pas)
  - Exemple: *kubectl get pods -l 'env in (production,staging),tier notin (frontend)'*

# Sélecteurs de champs

- Ils permettent de filtrer les objets Kubernetes selon la valeur de certains champs internes, comme le nom, le namespace, ou l'état d'un pod.
- **Syntaxe :**
  - `metadata.name=mon-pod`
  - `status.phase=Running`
  - `metadata.namespace!=default`
- **Exemple:**
  - *`kubectrl get pods --field-selector status.phase=Running`*
  - *`kubectrl delete pod --field-selector=status.phase==Succeeded`*

# Labels & Selectors: Exercices

- Charger la configuration labels\_exercise.yaml
- Afficher l'ensemble des pods avec leur label
- Afficher les pods destinés à la production
- Afficher les pods dont le dev lead est karthik
- Afficher les pods dont le dev est karthik et destinés à un environnement staging
- Afficher les pods dont le dev n'est pas karthik et destinés à un environnement staging
- Afficher les pods dont la version release est comprise entre 1.0 et 2.0
- Afficher les pods dont la version release est comprise entre 1.0 et 2.0, et dont l'équipe est marketing ou ecommerce
- Afficher les pods dont la version release n'est pas comprise entre 1.0 et 2.0

# Namespace Controller

- Le contrôleur Namespace est un composant du control plane de Kubernetes chargé de gérer le cycle de vie des objets Namespace.
- Son rôle est de s'assurer que la création, la mise à jour et la suppression des namespaces se déroulent correctement dans le cluster.
- Lorsqu'un namespace est supprimé, il veille également à la suppression de toutes les ressources associées à ce namespace (pods, services, configmaps, etc.) afin de garantir une isolation et une cohérence des ressources au sein du cluster.
- Cas d'usage typiques
  - Isolation d'environnements (développement, test, production)
  - Séparation des ressources par équipe ou projet
  - Application de politiques de sécurité et de quotas spécifiques à chaque namespace

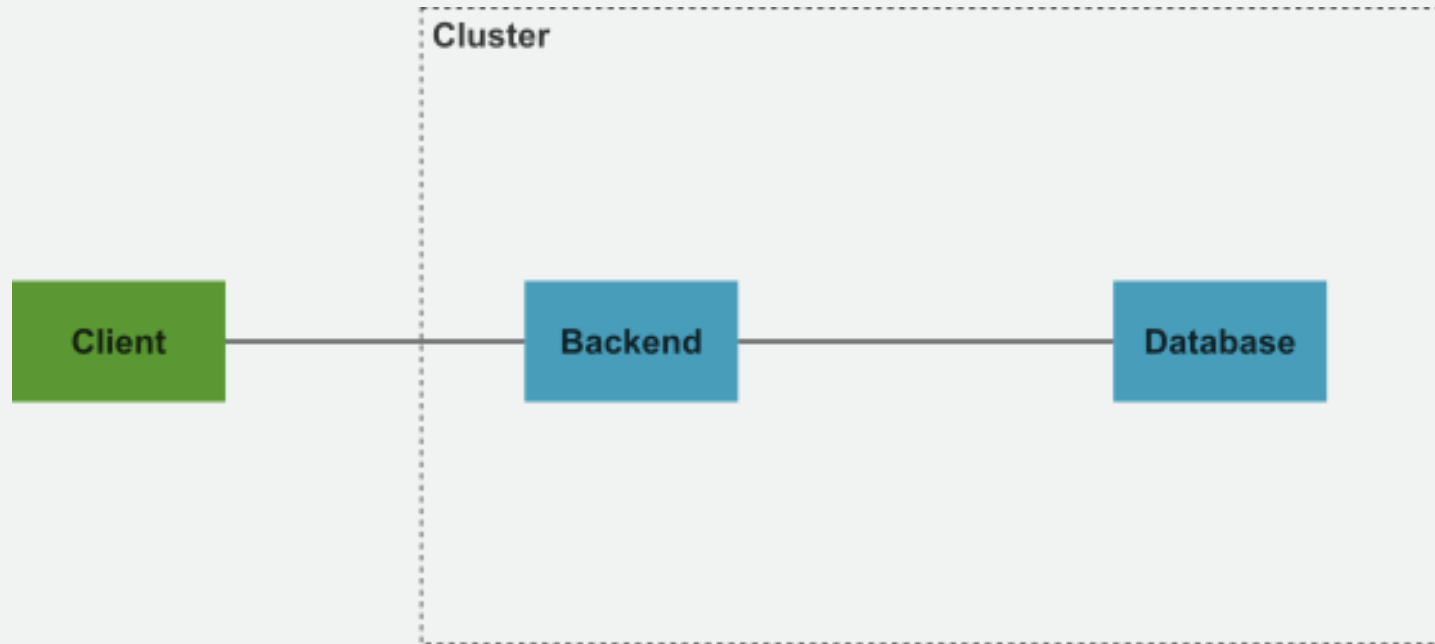
# Namespace Controller - Comportement

- **Création** : Lorsqu'un objet Namespace est créé, le contrôleur s'assure qu'il est bien enregistré dans le cluster et prêt à accueillir des ressources.
- **Suppression** : Quand un namespace est supprimé, le contrôleur Namespace marque d'abord le namespace comme "Terminating" et orchestre la suppression de toutes les ressources qui lui sont associées. Ce n'est qu'une fois toutes les ressources supprimées que le namespace disparaît définitivement du cluster.
- **Isolation** : Le contrôleur garantit que les ressources d'un namespace sont isolées de celles des autres namespaces. Les noms des ressources doivent être uniques à l'intérieur d'un namespace, mais peuvent être identiques dans des namespaces différents.
- **Gestion de quotas et de politiques** : Il permet l'application de quotas de ressources, de politiques de sécurité et de règles d'accès (RBAC) à l'échelle du namespace, facilitant ainsi la séparation des environnements (ex : dev, prod) ou des équipes.

# Services

# Interconnexion

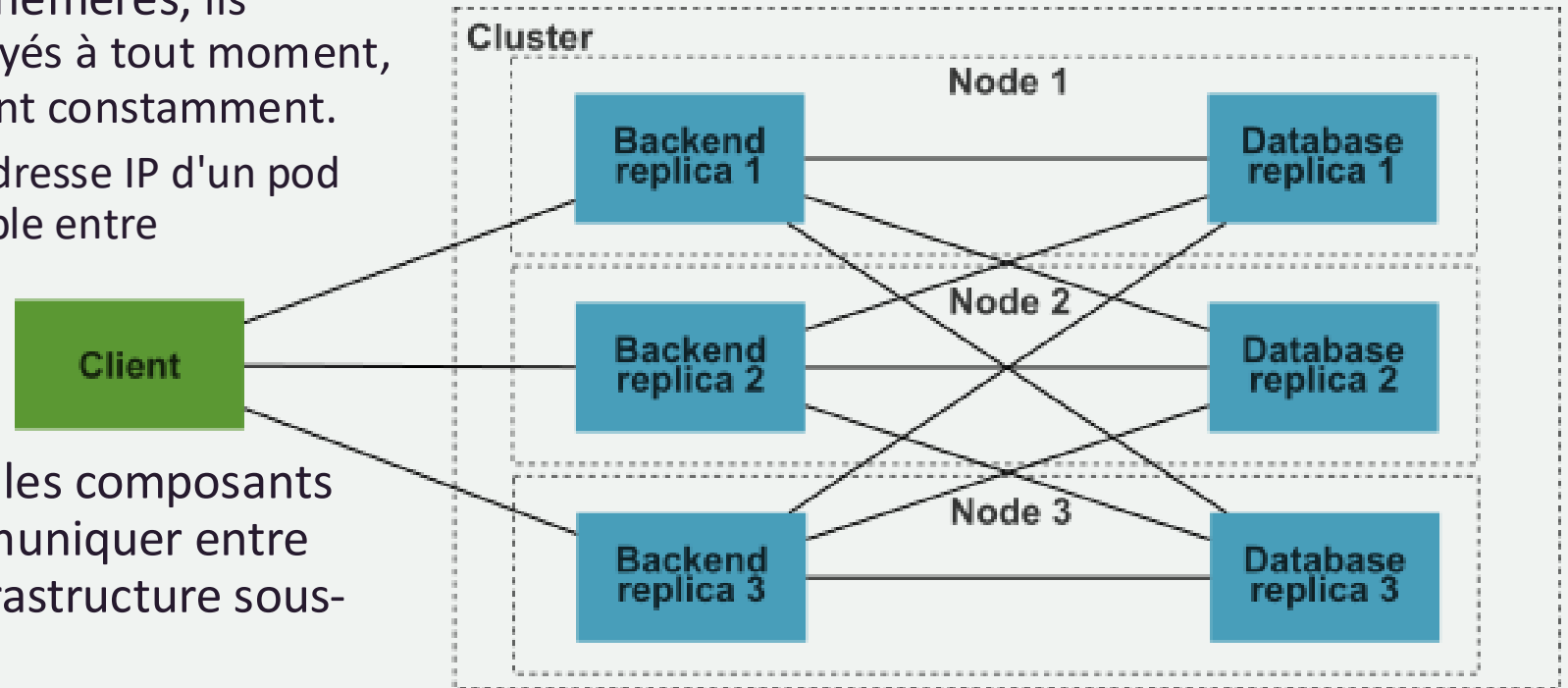
- À la lumière de ce que nous avons vu, quels sont les problèmes associés à cette architecture ?





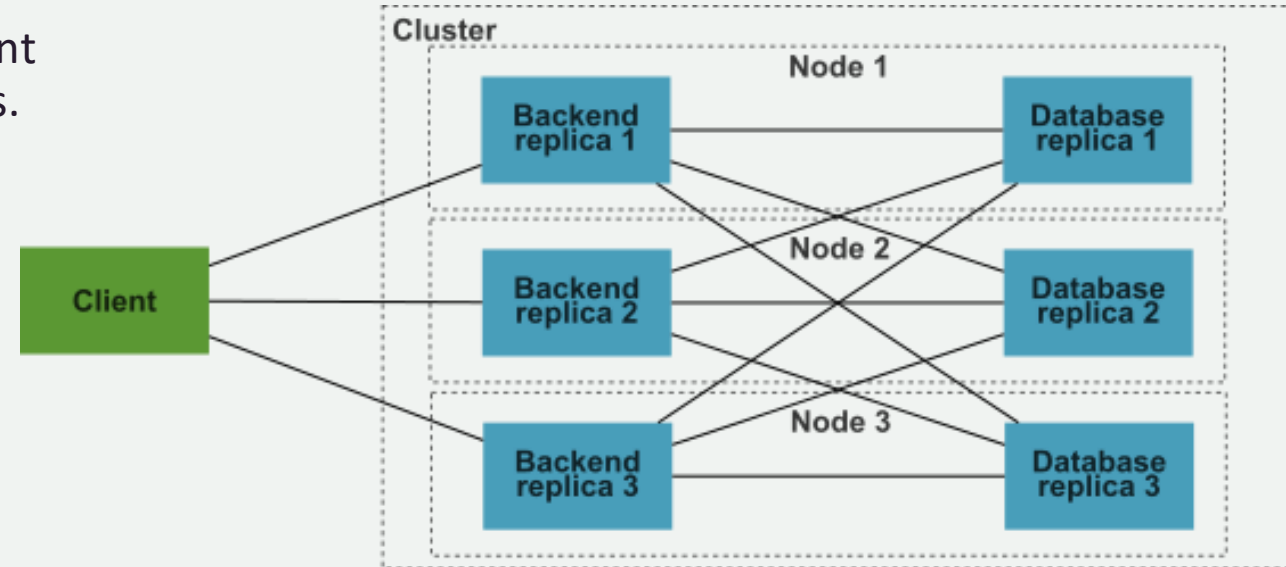
# Interconnexion : Problèmes

- Les ressources sont généralement éphémères, ils peuvent être créés, détruits ou redéployés à tout moment, ce qui fait que leurs adresses IP changent constamment.
  - Il est impossible de s'appuyer sur l'adresse IP d'un pod pour établir une communication stable entre composants d'une application
- Dans une architecture microservices, les composants doivent pouvoir se découvrir et communiquer entre eux sans connaître les détails de l'infrastructure sous-jacente (IP, nombre de Pods, etc.).
  - Sans abstraction, chaque composant devrait gérer lui-même la découverte des autres, ce qui est complexe et source d'erreurs.



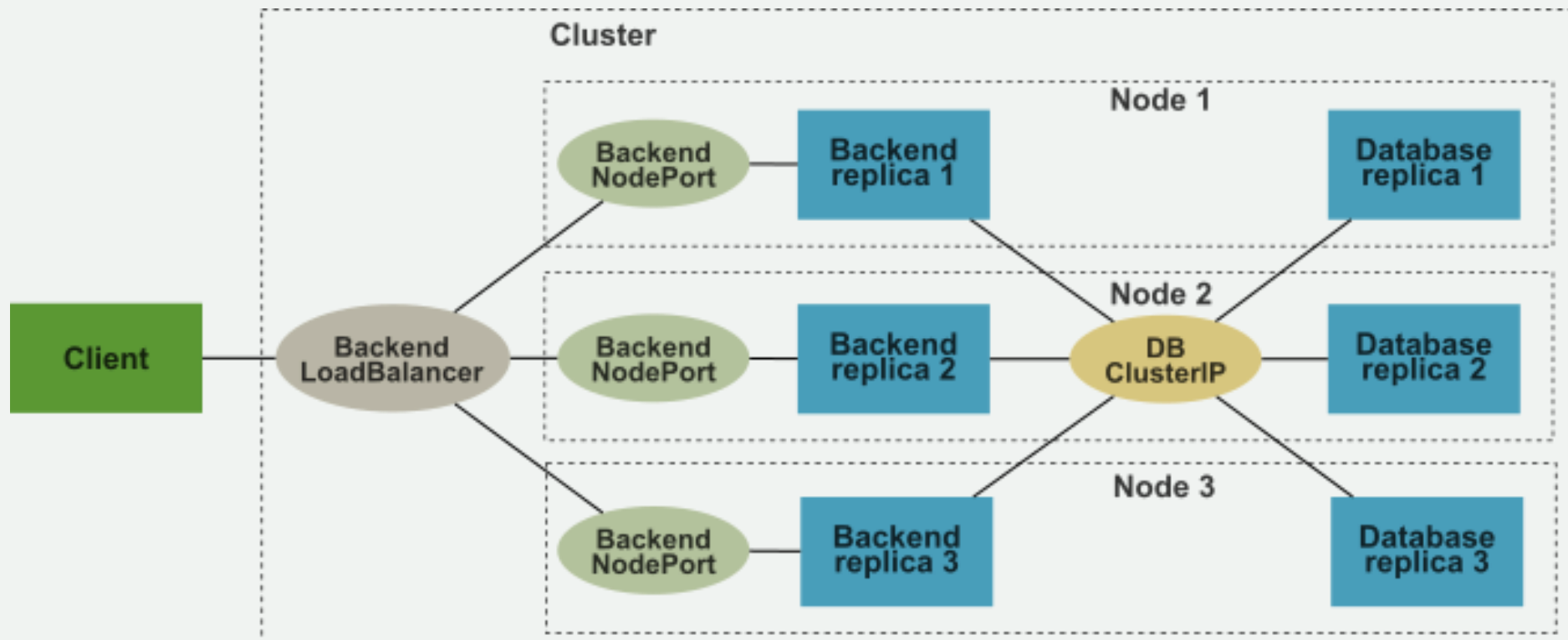
# Interconnexion : Problèmes

- Lorsqu'il existe plusieurs Pods pour un même service applicatif, il faut répartir le trafic entre eux pour optimiser la performance et la résilience.
  - Sans mécanisme d'équilibrage, certains Pods pourraient être surchargés tandis que d'autres resteraient inactifs.
- Par défaut, les Pods ne sont accessibles que depuis l'intérieur du cluster.
  - Impossible d'exposer une application (API, site web, etc.) à des clients externes sans solution dédiée.
- Les applications complexes reposent sur de multiples composants interdépendants (frontend, backend, base de données, etc.) dont la disponibilité et la connectivité doivent être garanties malgré la volatilité des Pods.



# Les services

- Les services apportent des solutions aux problèmes posés par les diapos précédentes
- Il existe différents types de services qui seront détaillés dans les diapos suivantes

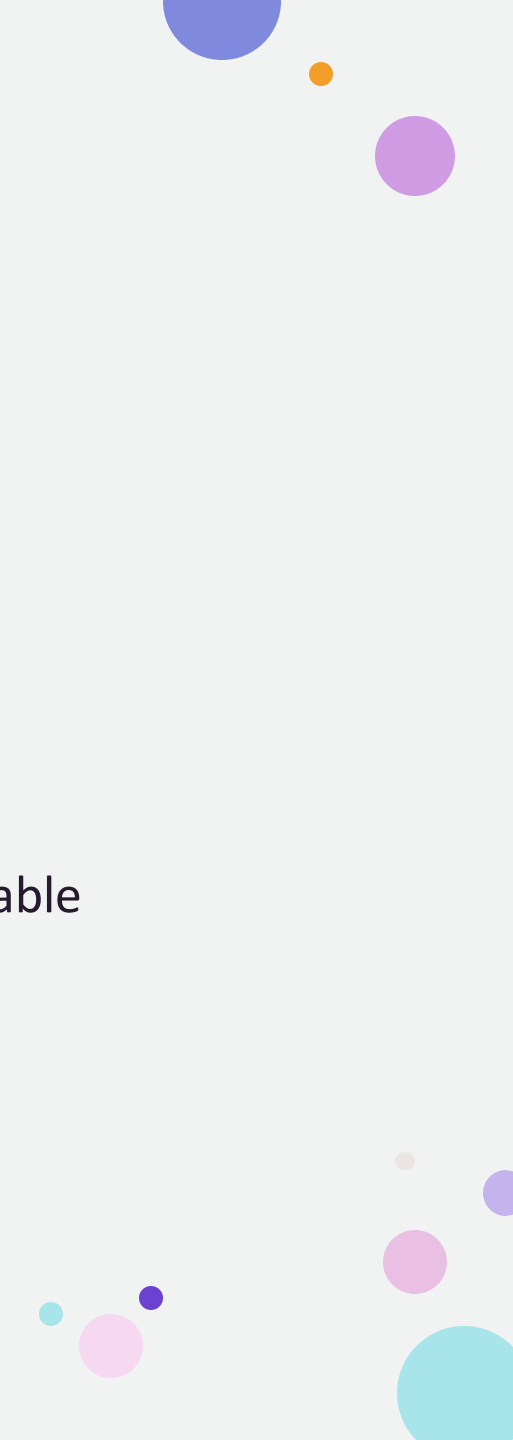


# Services: Présentation

- Un Service est une ressource Kubernetes qui fournit une abstraction réseau stable pour accéder à un ou plusieurs Pods, indépendamment de leur cycle de vie.
- Il agit comme un proxy ou un load balancer interne, permettant de communiquer avec des applications tournant dans des Pods sans se soucier de leurs adresses IP éphémères.
- Principaux rôles
  - **Adresse IP fixe et DNS stable:** Les Pods peuvent être créés, détruits ou déplacés : leur IP change. Un Service attribue une IP virtuelle et un nom DNS fixe, qui restent constants pour les clients.
  - **Load balancing:** Un Service répartit automatiquement le trafic entre tous les Pods correspondants (ciblés par un label selector), assurant disponibilité et scalabilité.
  - **Découverte et communication:** Les Services facilitent la découverte et la communication entre composants d'une application (frontend/backend, microservices, etc.) au sein du cluster.



# Services: Présentation

- Les Services Kubernetes sont indispensables pour :
    - Offrir une adresse IP et un nom DNS stables à des Pods dynamiques
    - Simplifier la découverte et la communication entre applications
    - Répartir la charge entre plusieurs Pods
    - Exposer des applications à l'extérieur du cluster
    - Garantir une connectivité fiable dans des architectures microservices
  - Sans services, la gestion réseau dans Kubernetes serait manuelle, fragile et non scalable
- 

# Services: Commandes de base

- Exposer un service:
  - `$kubectl expose deployment helloworld --type=NodePort`
  - `$kubectl expose deployment helloworld --type=LoadBalancer --name=hello-lb`
  - **TP:** lister les ressources après l'exécution de la première commande
  - **TP:** Accéder au service à partir du navigateur
- Quelques précisions :
  - Lorsque vous créez un service, une nouvelle entrée DNS permet au service d'être accessible via le serveur DNS de Kubernetes.
  - Vous pouvez définir un service pour cibler un ensemble spécifique de pods à l'aide de sélecteurs et d'étiquettes.
  - Les services surveillent en permanence les pods en cours d'exécution et disponibles pour recevoir du trafic, et ils dirigent le trafic vers ces pods uniquement.

# ClusterIP

- Un service de type ClusterIP est le type de service par défaut dans Kubernetes.
- Il attribue une adresse IP virtuelle interne au service, accessible uniquement depuis l'intérieur du cluster Kubernetes (jamais depuis l'extérieur).
- Il permet la **communication interne** entre différents composants de l'application (par exemple, entre un backend et une base de données, ou entre microservices).
- Sur l'exemple :
  - Le service DB ClusterIP est représenté au centre, accessible par les différents backends (Backend replica 1, 2, 3) répartis sur plusieurs nœuds du cluster.
  - Seuls les Pods à l'intérieur du cluster peuvent accéder à ce service via son ClusterIP ou son nom DNS interne (ex : db.default.svc.cluster.local).
  - Les clients extérieurs ne peuvent pas accéder directement à ce service : il n'est pas exposé hors du cluster.

# ClusterIP - Usages

- **Exposer des services internes :**

Sur l'exemple, la base de données (DB) n'est accessible que par les backends, jamais directement par les clients extérieurs.

- **Service discovery :**

Les applications internes peuvent communiquer entre elles de façon stable, même si les Pods changent d'IP, grâce à l'IP virtuelle et au nom DNS du service.

- **Load balancing interne :**

Le service ClusterIP répartit automatiquement le trafic entre tous les Pods cibles (par exemple, plusieurs réplicas d'une base de données ou d'un backend).

- Exemple: `kubectl expose deploy/db --port=5432`



# ClusterIP: Exercice

- **Objectif** : Créer un service ClusterIP simple pour exposer un déploiement au sein du cluster.
- Créer un déploiement :
  - Déployer une application comme nginx.
  - Nommez le déploiement nginx-deployment.
  - Veillez à ce qu'il ait une étiquette, par exemple app:nginx.
- Créer un service ClusterIP :
  - Créez un service de type ClusterIP pour exposer nginx-deployment.
  - Nommez le service nginx-service.
  - Le service doit cibler les pods avec le label app : nginx.
  - Exposer le port HTTP standard (80).
- Testez le service :
  - Depuis le cluster (en utilisant un Pod), accédez au service nginx sur le port 80.

# NodePort

- Un Service de type NodePort permet d'exposer une application Kubernetes à l'extérieur du cluster en ouvrant un port spécifique (dans la plage 30000–32767) **sur chaque nœud du cluster**.
  - Les clients externes peuvent accéder à l'application via l'adresse IP d'un nœud et le port NodePort attribué.
  - Le NodePort redirige ensuite le trafic vers les Pods cibles à l'intérieur du cluster, en s'appuyant sur la sélection par labels, comme pour un ClusterIP.
- Sur l'exemple
  - Le client externe envoie ses requêtes vers le Backend LoadBalancer ou directement vers un des Backend NodePort exposés sur chaque nœud.
  - Chaque Backend NodePort reçoit le trafic sur un port spécifique (par exemple, 31234) et le redirige vers un des Pods backend disponibles sur le cluster.
  - Le routage est assuré même si le Pod cible ne se trouve pas sur le nœud qui a reçu la requête : Kubernetes assure la redirection interne.
- Le service NodePort est accessible à la fois depuis l'intérieur du cluster (comme un ClusterIP) et depuis l'extérieur, à condition que le pare-feu autorise le port NodePort sur les nœuds.

# NodePort - Usages

- Exposer rapidement une application à l'extérieur (pour des tests, du débogage, ou des déploiements simples).
- Accès direct sans avoir besoin d'un LoadBalancer cloud ou d'un Ingress.
- Points importants
  - Plage de ports : 30000–32767 (modifiable dans la config du cluster).
  - Accès sur tous les nœuds : Le port est ouvert sur chaque nœud du cluster, quel que soit l'endroit où tourne le Pod.
  - Sécurité : Il faut ouvrir le port NodePort dans le pare-feu pour permettre l'accès externe.
  - Limites : Moins adapté à la production (ports non standards, gestion manuelle du firewall, pas d'équilibrage avancé).

# NodePort: Exercice

- **Objectif** : Exposer une application au trafic externe à l'aide d'un service NodePort.
- Créer ou utiliser un déploiement existant :
  - Utilisez le déploiement nginx de l'exercice précédent.
- Créer un service NodePort :
  - Exposez le déploiement nginx à l'aide d'un service NodePort.
    - § Exemple: `kubectl expose deploy/db --port=5432 --type=NodePort`
  - Nommez le service nginx-nodeport.
  - Assurez-vous qu'il cible les pods nginx.
  - Choisissez une plage de ports entre 30000 et 32767 pour l'accès externe.
- Accédez au service de manière externe :
  - Accédez à l'application nginx en utilisant le NodePort sur l'une des adresses IP de votre cluster Kubernetes.

# LoadBalancer

- Un Service de type LoadBalancer est le moyen le plus simple et le plus rapide d'exposer une application Kubernetes **à l'extérieur du cluster**, via une **IP publique** ou un nom **DNS externe**.
  - Ce type de service demande au cloud provider (GCP, AWS, Azure, etc.) de provisionner automatiquement un load balancer externe qui reçoit le trafic des clients et le redirige vers les nœuds du cluster.
  - Il fournit ainsi une entrée unique, stable et hautement disponible pour accéder à une application, sans avoir à gérer manuellement l'infrastructure réseau externe
- Sur l'exemple :
  - Le client externe envoie ses requêtes au Backend LoadBalancer.
  - Ce LoadBalancer est associé à une adresse IP publique ou un nom DNS, accessible depuis Internet.
  - Le LoadBalancer distribue le trafic vers les différents Backend NodePort exposés sur chaque nœud du cluster.
  - Les NodePorts redirigent ensuite le trafic vers les Pods backend disponibles, assurant la répartition de charge et la tolérance aux pannes.

# LoadBalancer – Fonctionnement et Usages

- **Fonctionnement**

- Lorsque vous créez un service avec type: LoadBalancer, Kubernetes crée d'abord un service NodePort, puis demande au cloud provider de provisionner un load balancer externe qui pointe vers ces NodePorts.
- Le load balancer externe gère la distribution du trafic, la surveillance de la santé des backends, et publie son adresse IP dans le champ `.status.loadBalancer` du service.

- **Usages**

- Exposer une application à Internet (API, site web, microservice) de manière sécurisée et scalable.
- Haute disponibilité : le load balancer répartit le trafic sur plusieurs Pods/nœuds, évitant tout point de défaillance unique.
- Simplicité : aucune configuration manuelle de l'infrastructure réseau externe n'est nécessaire, tout est automatisé par Kubernetes et le cloud provider.

# LoadBalancer – Points importants

- Nécessite un cloud provider compatible (GCP, AWS, Azure...). Sur un cluster **on-premise**, il faut utiliser une solution comme MetalLB.
- Coût : chaque service LoadBalancer crée une ressource facturée par le cloud provider.
- Limites : pour de nombreux services, il est préférable d'utiliser un **Ingress Controller** qui **mutualise l'accès externe**.

# LoadBalancer: Exercice

- **Objectif** : Créer un service LoadBalancer avec des annotations personnalisées, en supposant que le cluster est hébergé chez un fournisseur de cloud qui prend en charge les services LoadBalancer.
- Créer ou utiliser un déploiement existant :
  - Utiliser le déploiement nginx ou créer un déploiement similaire.
- Créer un service LoadBalancer avec des annotations :
  - Créer un service de type LoadBalancer pour exposer le déploiement nginx.
  - Nommez le service nginx-loadbalancer.
  - Ajoutez des annotations spécifiques à votre fournisseur de cloud pour personnaliser le comportement du LoadBalancer (comme la définition du type de load balancer ou des configurations de health check).
- Testez le service en externe :
  - Accédez à l'application nginx en utilisant l'IP publique fournie par le LoadBalancer.
- Défi supplémentaire :
  - Modifier les paramètres du LoadBalancer en utilisant des annotations pour atteindre des objectifs spécifiques (comme l'activation de l'affinité de session ou la terminaison SSL).



# ExternalName

- Un service de type ExternalName est un type spécial de Service Kubernetes qui ne pointe pas vers des Pods du cluster, mais agit comme un alias DNS vers une ressource externe (hors du cluster).
  - Il ne possède ni selector, ni ClusterIP, ni NodePort, ni LoadBalancer.
  - Il se contente de rediriger toute requête faite à son nom DNS interne vers un nom DNS externe que vous définissez.
- Fonctionnement
  - Lorsque vous créez un service ExternalName, Kubernetes enregistre un enregistrement DNS spécial (CNAME) dans le DNS interne du cluster.
  - Toute requête à ce service (ex : mydb.default.svc.cluster.local) sera résolue en une redirection DNS vers le nom externe spécifié (ex : my.database.example.com).
  - Il n'y a pas de routage réseau ou de proxy : seule la résolution DNS est affectée.

# ExternalName - Exemple

```
apiVersion: v1
kind: Service
metadata:
  name: my-external-db
spec:
  type: ExternalName
  externalName: my.database.example.com
```

- Ici, toute requête à my-external-db.default.svc.cluster.local sera redirigée (au niveau DNS) vers my.database.example.com

# ExternalName: Avantages et limites

- **Avantages**

- Simplicité : pas besoin de gérer un proxy ou un Pod intermédiaire.
- Uniformité : les applications accèdent à tous les services (internes ou externes) via le même schéma DNS Kubernetes.
- Centralisation : un seul point à modifier en cas de changement d'endpoint externe.

- **Limites**

- Pas de load balancing ou de proxying : c'est uniquement une redirection DNS, sans gestion de ports multiples ou de santé des endpoints.
- Le nom externe doit être résoluble et accessible depuis le cluster.
- Pas d'intégration avec les selectors ou les Endpoints Kubernetes.

# Endpoint Controller

- Le contrôleur Endpoint est un composant interne de Kubernetes qui gère dynamiquement les objets Endpoints associés à chaque Service.
- Son rôle principal est de maintenir la liste à jour des adresses IP et ports des pods qui sont sélectionnés par un **Service**, afin de permettre le routage du trafic réseau vers les bonnes instances d'application.
- Ce contrôleur fonctionne en arrière-plan :
  - Il surveille les changements dans les Services et les pods du cluster.
  - Lorsqu'un pod correspondant aux critères d'un Service (via le label selector) apparaît, disparaît ou change d'état, le contrôleur Endpoint met à jour l'objet Endpoints associé au Service.
- Comportement du contrôleur Endpoint
  - Ajout automatique : Dès qu'un pod "Ready" avec les bons labels est créé, son IP est ajoutée à l'objet Endpoints du Service concerné.
  - Suppression automatique : Si un pod disparaît, n'est plus "Ready", ou ne correspond plus au selector, son IP est retirée de l'objet Endpoints.
  - Mise à jour en temps réel : Toute modification (création, suppression, mise à jour de pods/services) est immédiatement répercutée dans l'objet Endpoints.
- Exemple: `kubectl get endpoints my-service -o yaml`

The slide features a light gray background with decorative elements in the corners. The top-left corner contains several circles in shades of pink, purple, and orange. The top-right corner has circles in blue, purple, and orange. The bottom-right corner is decorated with circles in teal, pink, and purple. The main text is centered in a dark purple font.

# Cycle de vie – Partie 3

## Autres contrôleurs liés aux pods

# StatefulSet

- Le contrôleur StatefulSet gère le déploiement et la gestion de pods stateful, c'est-à-dire des applications qui nécessitent :
  - Une identité réseau stable et unique pour chaque pod (nom DNS, hostname)
  - Un stockage persistant et stable, associé à chaque pod
  - Un ordre précis lors du déploiement, de la mise à l'échelle, des mises à jour et de la suppression des pods
- Contrairement à un Deployment (utilisé pour les applications stateless), le StatefulSet garantit que chaque pod conserve son identité et son volume persistant, même en cas de redémarrage ou de rescheduling sur un autre nœud
- Cas d'usage typiques
  - Bases de données distribuées : PostgreSQL, MySQL, MongoDB, Cassandra, etc.
  - Systèmes de messagerie : Kafka, RabbitMQ
  - Applications nécessitant un consensus ou une réplication, comme etcd ou ZooKeeper

# StatefulSet - Fonctionnement

- **Création et suppression ordonnée** : Les pods sont créés et supprimés un par un, dans l'ordre de leur index ordinal (par exemple, pod-0, pod-1, pod-2). Un nouveau pod n'est créé que lorsque le précédent est prêt et en fonctionnement.
- **Identité persistante** : Chaque pod reçoit un nom unique (ex : web-0, web-1), une adresse réseau stable, et son propre volume persistant (PVC). Si un pod est supprimé, son remplaçant reprend exactement le même nom et le même volume.
- **Mises à jour ordonnées** : Lors d'un rolling update, les pods sont mis à jour un par un, dans l'ordre inverse (du plus grand index vers le plus petit), pour garantir la stabilité et la cohérence de l'application.
- **Volumes persistants individuels** : Chaque pod a son propre PersistentVolumeClaim (PVC), créé à partir d'un template. Le volume reste associé au pod même après suppression ou recréation de celui-ci.
- Exercice: Étudier l'exemple du fichier mongo.yaml
  - Dans cet exemple, chaque service sera identifié avec ce nom par le DNS: mongo-X.**mongo**.default.svc.cluster.local
  - mongo représente un service "headless", sans IP, les requêtes étant directement dirigé vers les pods
  - Cf. Exemple headless-service.yaml

# DaemonSet

- Le contrôleur DaemonSet garantit qu'un pod spécifique est déployé et fonctionne sur chaque nœud d'un cluster Kubernetes, ou sur un sous-ensemble de nœuds si des sélecteurs sont utilisés. Ce comportement est essentiel pour les tâches qui doivent être présentes sur tous les nœuds, comme :
  - La collecte de logs (ex : Fluentd, Filebeat)
  - La surveillance (ex : Prometheus Node Exporter, Datadog)
  - Les agents réseau (ex : CNI comme Calico, Cilium)
  - Les outils de sécurité (ex : Falco, Sysdig)
- Comportement
  - Lorsqu'un DaemonSet est créé, Kubernetes déploie automatiquement un pod sur chaque nœud du cluster.
  - Si un nouveau nœud rejoint le cluster, le DaemonSet y déploie automatiquement un pod correspondant.
  - Si un nœud est supprimé, le pod associé est également supprimé.
  - Les mises à jour peuvent être manuelles (OnDelete) ou progressives (RollingUpdate), selon la configuration.
  - Il est possible de restreindre le déploiement à certains nœuds grâce à des labels et des nodeSelectors.
- Exercice: Étudier l'exemple du fichier fluentd.yaml



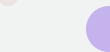
# Job

- Le contrôleur Job gère l'exécution de tâches ponctuelles ou batch, c'est-à-dire des traitements qui doivent s'exécuter jusqu'à leur complétion, puis s'arrêter.
- Contrairement aux Deployments ou StatefulSets qui maintiennent des pods en fonctionnement continu, le Job garantit qu'un ou plusieurs pods s'exécutent jusqu'à ce qu'ils terminent avec succès, même en cas de défaillance ou de suppression d'un pod : il relance automatiquement les pods jusqu'à atteindre le nombre de succès attendu.
- Cas d'usage typiques
  - Traitement de données batch
  - Scripts de migration de base de données
  - Génération de rapports ponctuels
  - Tâches de maintenance ou de nettoyage
- Exercice: Étudier l'exemple du fichier job-example.yaml



# Job - Fonctionnement

- **Exécution fiable** : Le Job crée un ou plusieurs pods pour exécuter une tâche. Si un pod échoue ou disparaît, le contrôleur en recrée un nouveau jusqu'à ce que le nombre de succès défini soit atteint.
- **Gestion de la parallélisation** : Il est possible d'exécuter plusieurs pods en parallèle pour accélérer le traitement (paramètres parallelism et completions).
- **Arrêt automatique** : Une fois la tâche terminée (nombre de succès atteint), le Job n'exécute plus de pods.
- **Nettoyage** : Supprimer le Job supprime aussi les pods associés.
- **Redémarrage contrôlé** : Le comportement de redémarrage des pods est défini par restartPolicy (souvent OnFailure ou Never).



# CronJob

- Le contrôleur CronJob permet de planifier et d'exécuter des tâches récurrentes dans Kubernetes, à la manière de la commande cron sous Linux.
- Il s'appuie sur le contrôleur Job pour lancer des pods selon un calendrier défini. Chaque exécution planifiée crée un Job indépendant, qui s'occupe d'exécuter la tâche jusqu'à son terme.
- Cas d'usage typiques :
  - Sauvegardes régulières de bases de données
  - Nettoyage périodique de ressources (logs, fichiers temporaires...)
  - Génération de rapports planifiés
  - Envoi automatique d'emails ou de notifications
- Exercice: Étudier l'exemple du fichier cronjob-example.yaml

# CronJob - Fonctionnement

- **Planification** : Utilise une syntaxe cron standard pour définir la fréquence d'exécution (schedule).
- **Gestion des exécutions** : À chaque occurrence, un nouveau Job est créé. Si une exécution précédente n'est pas terminée, le comportement dépend de la politique de concurrence (concurrencyPolicy).
- **Rétention** : Les anciens Jobs et pods peuvent être automatiquement supprimés selon les paramètres successfulJobsHistoryLimit et failedJobsHistoryLimit.
- **Gestion des erreurs** : Si un Job échoue, le CronJob peut le relancer selon la politique du Job (backoffLimit).

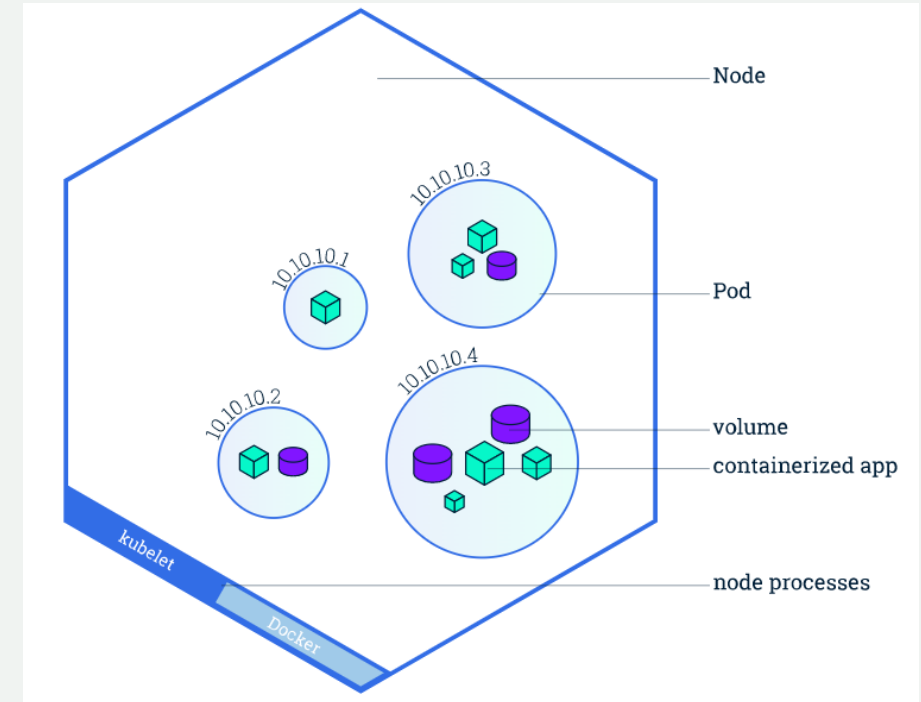
# Les volumes

# Présentation

- La gestion des volumes dans Kubernetes implique l'orchestration des ressources de stockage pour les conteneurs dans les Pods. Kubernetes prend en charge plusieurs types de volumes, chacun ayant ses propres caractéristiques et s'adaptant à différents cas d'utilisation.
- La gestion des volumes dans Kubernetes est polyvalente, prenant en charge une gamme de types de stockage allant du stockage temporaire spécifique à un pod au stockage persistant à l'échelle du cluster.
- En utilisant les **PV**, les **PVC** et les **classes de stockage**, Kubernetes fournit un cadre robuste et flexible pour gérer les besoins de stockage dans un environnement conteneurisé. Les applications peuvent ainsi utiliser les ressources de stockage de manière transparente, en fonction de leurs besoins spécifiques en matière de cycle de vie du stockage, de performances et d'évolutivité.
- Documentation: <https://kubernetes.io/fr/docs/concepts/storage/volumes/>

# Volumes et persistance

- De base, les volumes dans Kubernetes sont déclarés dans une spécification de Pod et sont montés dans les conteneurs au sein de ce Pod.
  - Ils peuvent être utilisés à diverses fins, comme le stockage de données d'application, la conservation de fichiers de configuration et la gestion de secrets.
- **Questions:**
  - Quels problèmes posent les volumes dans un environnement clusterisé ?
  - Que se passe-t-il quand un pod avec un volume est supprimé ? S'il est instancié sur un autre nœud en particulier ?



# Problèmes

- **Contention et performances**

- Lorsque plusieurs machines virtuelles ou applications partagent un même volume dans un cluster, il existe un risque de contention disque. Si trop de workloads sollicitent intensément le même volume, cela peut entraîner une dégradation significative des performances, notamment une augmentation de la latence des opérations d'E/S et une saturation du stockage.
- Il est donc crucial de bien dimensionner le nombre de volumes et leur répartition en fonction des charges de travail attendues, en concertation avec le fournisseur de stockage.

- **Organisation et gestion des volumes**

- La planification de l'organisation des volumes (nombre, taille, répartition des fichiers) est complexe. Une mauvaise organisation peut entraîner un sous-emploi de l'espace disque, des difficultés lors des migrations ou du basculement, et compliquer la maintenance.
- Il faut également tenir compte de la gestion des LUN (unités logiques) et de la compatibilité avec les différents systèmes de fichiers utilisés par les nœuds du cluster.



# Problèmes

- **Disponibilité et basculement**

- Lorsqu'un volume partagé devient indisponible (ex. : panne matérielle, problème réseau, corruption), cela peut impacter tous les nœuds qui y accèdent, provoquant l'arrêt ou le basculement des services associés.
- Des opérations comme la sauvegarde, la création de snapshots ou des mises à jour système peuvent provoquer la mise hors ligne temporaire des volumes partagés, entraînant un arrêt des machines virtuelles ou des applications dépendantes.

- **Problèmes liés aux mises à jour et à la compatibilité**

- Des mises à jour logicielles ou matérielles peuvent introduire des incompatibilités ou des bugs, rendant les volumes partagés inaccessibles ou instables après un redémarrage ou une modification de configuration.
- Il est donc recommandé de tester les mises à jour sur des environnements de préproduction avant de les appliquer en production.

# Problèmes

- **Sécurité et vulnérabilités**

- Les volumes locaux ou partagés peuvent être la cible de vulnérabilités, notamment des attaques par injection de commandes si la configuration n'est pas sécurisée ou si des fichiers malveillants sont introduits dans le cluster.
- Il est essentiel de limiter les droits d'accès et de surveiller les configurations pour éviter les failles de sécurité.

- **Gestion de la mémoire et du cache**

- L'activation du cache pour les volumes partagés peut améliorer les performances, mais nécessite une gestion fine de la mémoire sur chaque nœud. Une mauvaise configuration peut entraîner une contention mémoire ou des redémarrages nécessaires des nœuds pour prendre en compte les changements.

- **Dépannage et complexité opérationnelle**

- La résolution des problèmes liés aux volumes dans un cluster est souvent complexe, nécessitant l'analyse des journaux, la collaboration avec les fournisseurs de stockage et parfois la reconfiguration de l'infrastructure.
- Les incidents sur un volume peuvent avoir un effet domino sur l'ensemble du cluster, affectant la disponibilité globale des services.

# Apports de kubernetes

- **Abstraction et découplage du stockage**

- Kubernetes introduit les objets PersistentVolume (PV) et PersistentVolumeClaim (PVC), qui permettent d'abstraire la gestion du stockage. Les applications (Pods) ne sont plus liées à un volume physique spécifique : elles expriment simplement leurs besoins via une PVC, et Kubernetes se charge de trouver ou de provisionner dynamiquement un volume adapté.
- Cette abstraction facilite la portabilité des applications et simplifie la gestion du cycle de vie des volumes.

- **Provisionnement dynamique**

- Kubernetes supporte le provisionnement dynamique des volumes grâce au concept de **StorageClass** et aux provisioners (généralement via CSI). Lorsqu'un Pod réclame un volume via une PVC, Kubernetes peut automatiquement créer un volume sur le backend de stockage approprié, ce qui élimine la gestion manuelle et réduit les risques d'erreur ou de sous-provisionnement.
- Cela permet aussi d'optimiser les coûts et l'utilisation des ressources, car les volumes ne sont créés que lorsqu'ils sont nécessaires.

# Apports de Kubernetes

- **Gestion de la persistance et du cycle de vie**

- Les volumes persistants (PV) sont indépendants du cycle de vie des Pods : les données ne disparaissent pas lors du redémarrage ou du remplacement d'un Pod, ce qui résout le problème de perte de données inhérent aux conteneurs éphémères.
- Kubernetes permet aussi de définir des politiques de rétention et de recyclage des volumes, facilitant la gestion du stockage au fil du temps.

- **Contrôle d'accès et sécurité**

- Kubernetes offre des mécanismes natifs pour contrôler l'accès aux volumes : RBAC, namespaces, Network Policies, et gestion fine des droits sur les PV/PVC, ce qui limite les risques de fuite ou de corruption de données entre applications ou équipes.
- Les quotas et limites de stockage peuvent être appliqués pour éviter la saturation ou la contention des ressources.

# Apports de kubernetes

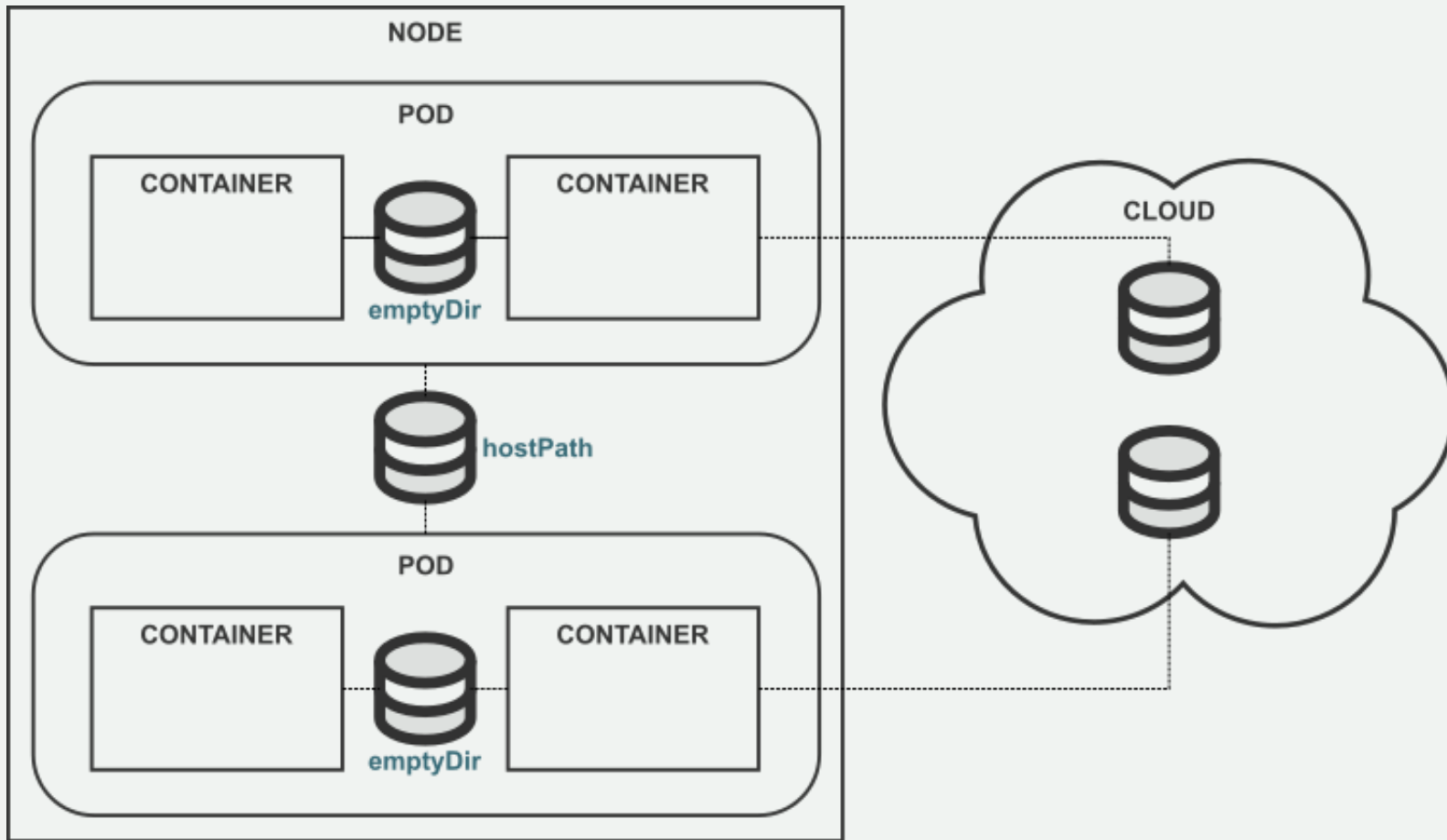
- **Optimisation des performances et de la disponibilité**

- Grâce aux différents modes d'accès (ReadWriteOnce, ReadOnlyMany, ReadWriteMany), Kubernetes permet d'adapter le type de volume à chaque besoin (accès exclusif, partagé, lecture seule, etc.), ce qui limite les risques de contention et optimise les performances selon le workload.
- L'intégration de solutions de stockage distribuées ou répliquées (GlusterFS, Ceph, EBS, etc.) via des drivers CSI permet de garantir la haute disponibilité et la résilience des données en cas de panne de nœud ou de volume.

- **Observabilité et troubleshooting**

- Kubernetes expose l'état des volumes, des PVC et des Pods via son API et ses outils (kubectl), ce qui facilite le diagnostic des problèmes de montage, de performance ou de capacité.
- Des solutions de monitoring comme Prometheus et Grafana peuvent être intégrées pour surveiller l'utilisation et la santé du stockage.

# Volumes - 1ère itération



- Analyser le contenu des fichiers suivants:
  - local-vol-examples.yaml
  - aws-vol-example.yaml
- Quels problèmes voyez-vous ?
- Inspirez-vous de ces exemples pour instancier un serveur postgresql persistant

# Types de volume

| Type de volume  | Description & Utilité                                                                                                                                                                                               | Persistance des données                                      | Cas d'usage principal                                                                                      |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <b>emptyDir</b> | Volume temporaire créé à l'assignation du Pod à un nœud. Vide au départ, <b>partagé</b> entre tous les conteneurs du Pod. Supprimé à la suppression du Pod. Peut être stocké sur disque ou en RAM (medium: Memory). | Durée de vie du Pod                                          | Espace temporaire, cache, traitement intermédiaire, partage de fichiers entre conteneurs d'un même Pod.    |
| <b>hostPath</b> | Monte un dossier ou fichier du système de fichiers du nœud hôte dans le Pod. Peut servir à accéder à des fichiers locaux du nœud, mais pose des risques de sécurité et de portabilité.                              | Durée de vie du Node (dépend du fichier/dossier sur le nœud) | Accès à des logs, sockets, ou stockage local pour tests/développement.                                     |
| <b>NFS</b>      | Monte un partage NFS (Network File System) distant dans le Pod. Permet de partager un même volume entre plusieurs Pods et nœuds.                                                                                    | Tant que le serveur NFS existe                               | Partage de données entre Pods, stockage persistant partagé, applications nécessitant un accès multi-nœuds. |

# Types de volume (Cont'd)

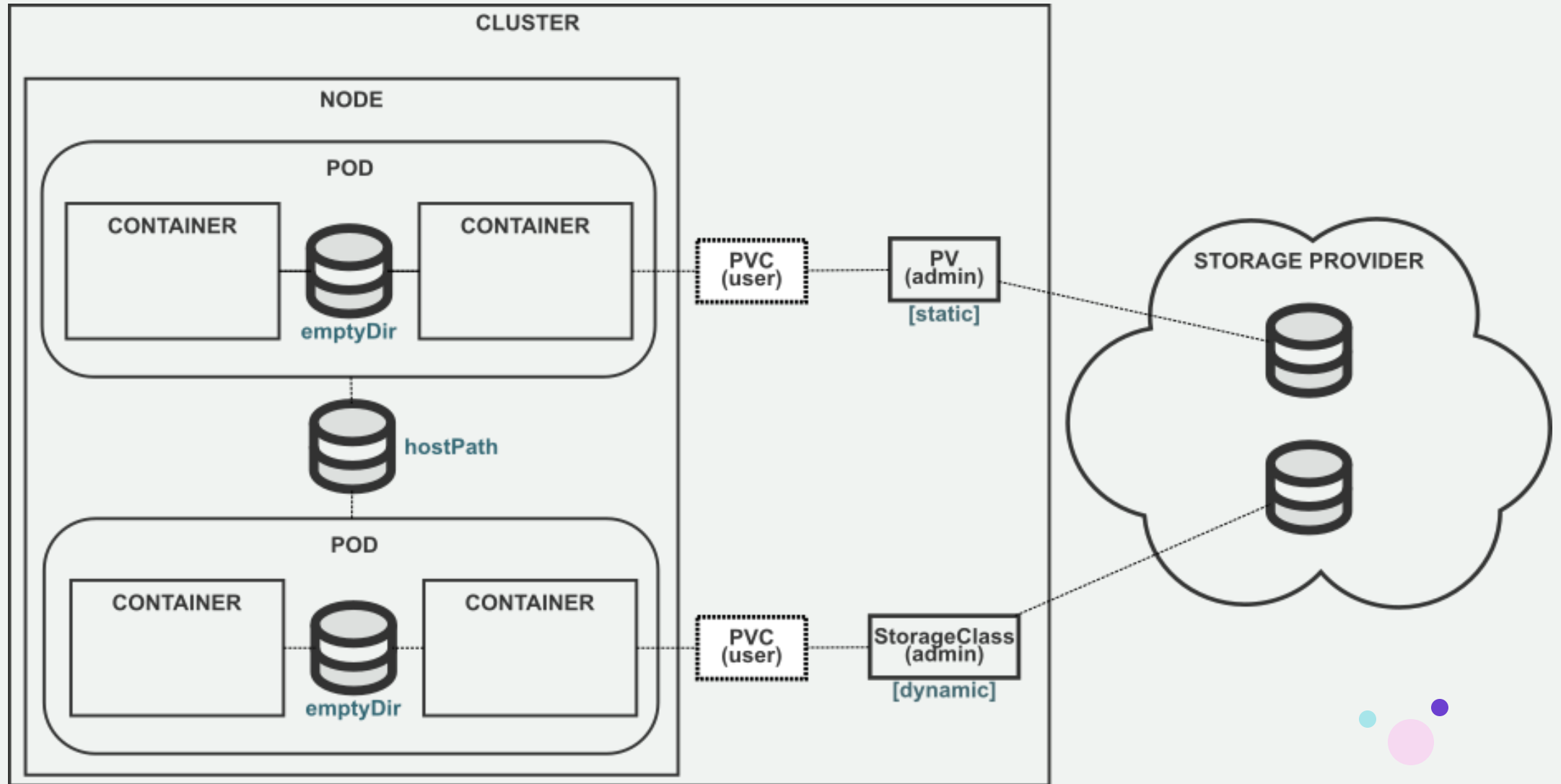
| Type de volume      | Description & Utilité                                                                                                                                                                                                         | Persistance des données                | Cas d'usage principal                                                                              |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|----------------------------------------------------------------------------------------------------|
| <b>Cloud volume</b> | Utilise des volumes fournis par le cloud (ex : AWS EBS, GCP Persistent Disk, Azure Disk). Provisionnement dynamique via StorageClass possible.                                                                                | Persistance au niveau du cluster/cloud | Stockage persistant fiable et scalable pour applications critiques, bases de données, etc.         |
| <b>PVC</b>          | Représente une demande de stockage persistante. Le PVC est lié à un PersistentVolume (PV), qui peut être de type local, NFS, cloud, etc. Le Pod utilise le PVC pour obtenir un stockage persistant, abstrait du backend réel. | Persistance au niveau du cluster       | Stockage persistant pour Pods, abstraction du stockage, gestion dynamique ou statique du stockage. |



# Pilotes de Volume

- Un pilote de volume (ou plugin de stockage) dans Kubernetes est un composant qui permet à un cluster d'interagir avec différents systèmes de stockage, qu'ils soient locaux, en réseau ou cloud.
- Les pilotes de volume sont essentiels pour abstraire la gestion du stockage, rendant possible l'utilisation de multiples solutions de stockage sans modifier la logique applicative.
- Le rôle principal d'un pilote de volume est de :
  - Provisonner dynamiquement ou statiquement des volumes pour les Pods.
  - Attacher et monter les volumes sur les nœuds où s'exécutent les Pods.
  - Gérer la persistance des données au-delà du cycle de vie des Pods.
  - Offrir des fonctionnalités avancées comme le snapshot, la réplication ou la gestion fine des accès.

# Volumes - 2nde itération



# Persistent Volume (PV)

- La 1ère itération laisse apparaître un problème de mélange de préoccupations: la provision de volumes (admin) et leur utilisation (user). En réponse à ce problème, kubernetes implante deux concepts essentiels: PV (moyen) et PVC (besoin).
- Un PersistentVolume (PV) est une ressource de stockage abstraite et indépendante dans un cluster Kubernetes.
- Il permet de fournir un espace de stockage persistant aux applications, c'est-à-dire un stockage qui subsiste même si les Pods qui l'utilisent sont arrêtés, supprimés ou recréés.
- Objectifs et utilité
  - **Abstraction du stockage** : Le PV masque les détails de la technologie de stockage sous-jacente (NFS, iSCSI, disque cloud, stockage local, etc.) pour les utilisateurs et les applications. Cela facilite la portabilité et la gestion du stockage dans le cluster.
  - **Persistance des données** : Contrairement aux volumes classiques liés au cycle de vie d'un Pod, un PV existe indépendamment et conserve les données au-delà de la durée de vie des Pods.
  - **Gestion centralisée** : Les administrateurs peuvent provisionner des PVs de façon statique (manuelle) ou dynamique (automatique via des StorageClass), permettant une gestion flexible et évolutive du stockage dans le cluster.

# PV - Fonctionnement

- Un PV est créé dans le cluster avec des spécifications précises (capacité, type de stockage, modes d'accès, politique de rétention, etc.).
- Les applications ne consomment pas directement un PV, mais passent par une PersistentVolumeClaim (PVC), qui est une demande de stockage.
  - Kubernetes associe alors automatiquement une PVC à un PV disponible qui satisfait les critères demandés.
- Une fois la liaison effectuée, le PV est réservé à la PVC et peut être monté dans un ou plusieurs Pods selon le mode d'accès choisi.

# Persistent Volume Claim (PVC)

- Un PersistentVolumeClaim (PVC) est une ressource Kubernetes qui représente une demande de stockage persistante, formulée par un utilisateur ou une application.
- Le PVC permet de spécifier la quantité de stockage souhaitée, le mode d'accès (lecture/écriture, lecture seule, etc.) et éventuellement la classe de stockage désirée, sans se soucier des détails techniques du volume sous-jacent.
- **Utilité**
  - Le PVC permet aux utilisateurs et aux Pods de consommer du stockage persistant de façon déclarative, sans connaître la technologie de stockage utilisée (NFS, disque cloud, etc.).
  - Il facilite la portabilité et l'automatisation : les développeurs décrivent seulement leurs besoins, et Kubernetes s'occupe de la gestion du stockage.

# PVC - Fonctionnement

- Principes de fonctionnement

- Le PVC agit comme une "réclamation" : il demande à Kubernetes de trouver ou de provisionner un PersistentVolume (PV) qui répond à ses critères (taille, mode d'accès, StorageClass).
- Kubernetes recherche alors un PV disponible correspondant, puis lie (bind) le PVC à ce PV. Une fois la liaison effectuée, le volume devient accessible au Pod qui utilise ce PVC.
- Si aucun PV existant ne correspond, et si une StorageClass est définie, Kubernetes peut automatiquement créer un PV adapté via le provisionnement dynamique.

- Principaux champs d'un PVC

- accessModes : définit les modes d'accès souhaités (ex : ReadWriteOnce, ReadOnlyMany, ReadWriteMany).
- resources.requests.storage : indique la quantité de stockage demandée.
- storageClassName : optionnel, précise la classe de stockage à utiliser (pour le provisionnement dynamique).
- selector : optionnel, permet de cibler des PV avec des labels spécifiques.

# Binding PV / PVC

- Lorsque plusieurs volumes persistants (PV) sont disponibles dans un cluster Kubernetes et qu'une réclamation de volume persistant (PVC) est créée, Kubernetes sélectionne un PV pour le PVC en fonction de plusieurs critères définis par les points suivants.
- **Exigences de dimensions** : La taille de stockage demandée par le PVC doit être inférieure ou égale à la capacité du PV. Kubernetes ne liera pas un PVC à un PV trop petit.
- **Modes d'accès** : Le PV et le PVC doivent avoir au moins un mode d'accès correspondant. Les modes d'accès courants sont les suivants :
  - ReadWriteOnce (RWO) : Le volume peut être monté en lecture-écriture par un seul nœud.
  - ReadOnlyMany (ROX) : Le volume peut être monté en lecture seule par plusieurs nœuds.
  - ReadWriteMany (RWX) : Le volume peut être monté en lecture-écriture par de nombreux nœuds.
- **Classe de stockage** : Si le PVC spécifie une StorageClass, Kubernetes ne prend en compte que les PV associés à cette StorageClass. Un PVC sans StorageClass spécifique ne peut se lier qu'à un PV sans StorageClass ou avec la StorageClass par défaut.

# Binding PV / PVC - Cont'd

- **Mode de volume:** Le mode de volume (système de fichiers ou bloc) du PVC et du PV doit correspondre.
- **Correspondance des sélecteurs:** Si le PVC spécifie un sélecteur, les étiquettes du PV doivent correspondre au sélecteur.
- **Correspondance ClaimRef:** Si un PV possède un champ claimRef, il est prélié au PVC mentionné dans claimRef. Dans ce cas, Kubernetes respecte cette liaison préalable et tente de lier ce PVC spécifique au PV.
- **Capacité et autres facteurs:** Kubernetes essaiera de trouver le plus petit PV approprié qui répond à tous les autres critères. Cependant, d'autres facteurs tels que les mécanismes de sélection interne peuvent influencer le choix.
- Note: Si aucune correspondance n'a été trouvée, le PVC passe en état **PENDING** jusqu'à ce qu'un PV lui corresponde.



# PV / PVC - Exercice

- Créer une demande de volume persistant (pvc-exercise.yaml)
  - Créer un PVC qui demande de stockage de 1 Go au cluster
  - Vérifier l'état de la demande
- Créer un volume persistant (pv-exercise.yaml)
  - Capacité: 500Mo
  - Mode: RWO
  - Montage sur l'hôte (/mnt/data)
  - Vérifier l'état du PVC
  - Quel problème voyez-vous ? Comment le résoudre ?
- Supprimer le PVC
  - Vérifier l'état du pv
  - Modifier la spécification du PV afin que celui-ci soit supprimé à sa libération
  - Vérifier que cela fonctionne

# Volume Reclaim Policy

- L'attribut persistant `VolumeReclaimPolicy` d'un objet PV dans Kubernetes définit ce que le cluster doit faire du volume une fois qu'il a été libéré de son PVC.
- **Exercice:** Étudier l'exemples du fichier `mongo.yaml`

| Valeur         | Comportement                                                                                                           | Remarques                                                                                                                                |
|----------------|------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Retain</b>  | Le PV reste existant après la suppression du PVC. Les données sont conservées : le volume passe en état "Released".    | L'administrateur doit manuellement nettoyer ou réaffecter le volume. Permet d'éviter la suppression accidentelle de données importantes. |
| <b>Delete</b>  | Le PV et la ressource de stockage associée (ex : disque cloud, volume NFS) sont supprimés automatiquement.             | Généralement utilisé pour les volumes provisionnés dynamiquement. Pratique pour les environnements éphémères ou automatisés.             |
| <b>Recycle</b> | Le volume est nettoyé (commande <code>rm -rf /thevolume/*</code> ) puis remis à disposition pour une nouvelle demande. | Peu utilisé aujourd'hui : la création dynamique de volumes via <code>StorageClass</code> est recommandée à la place.                     |

# Mode: FS vs Block - FS

- **Organisation** : Les données sont structurées en fichiers et dossiers, accessibles via des chemins hiérarchiques.
- **Gestion** : Le système de fichiers gère les droits d'accès, le partage, le verrouillage, les métadonnées (date, type, taille, etc.) et la structure des répertoires.
- **Utilisation** : Les applications accèdent aux données comme à des fichiers classiques. Ce mode est idéal pour les serveurs de fichiers, le stockage partagé (NAS), ou toute application qui manipule des fichiers et des dossiers.
- **Exemple dans Kubernetes** : Un volume de type Filesystem est monté dans un Pod, et l'application voit un répertoire prêt à l'emploi.

# Mode: FS vs Block - Block

- **Organisation** : Les données sont stockées dans des blocs de taille fixe, identifiés par des adresses uniques, sans structure hiérarchique ni métadonnées de fichiers.
- **Gestion** : L'application ou le système d'exploitation doit formater ce volume brut avec un système de fichiers ou y écrire directement des données binaires (cas des bases de données ou de certaines applications spécialisées).
- **Utilisation** : Convient aux bases de données, machines virtuelles, ou applications nécessitant des performances élevées et un accès direct au disque. Le stockage en bloc est privilégié pour sa rapidité, sa faible latence et sa flexibilité.
- **Exemple dans Kubernetes** : Un volume de type Block est exposé comme un périphérique brut dans le Pod, à formater ou à utiliser directement.

# Mode: FS vs Block

- **Filesystem** : prêt à l'emploi, structuré, idéal pour manipuler des fichiers.
- **Block** : brut, performant, flexible, nécessite une gestion supplémentaire côté application ou OS.

| Caractéristique      | Système de fichiers                       | Bloc                                         |
|----------------------|-------------------------------------------|----------------------------------------------|
| Structure            | Fichiers et dossiers hiérarchiques        | Blocs de données bruts                       |
| Gestion              | Par le système de fichiers                | Par l'application ou l'OS                    |
| Accès                | Par chemins de fichiers                   | Par adresses de blocs                        |
| Métadonnées          | Oui (date, type, droits, etc.)            | Non (juste un identifiant de bloc)           |
| Cas d'usage typiques | Partage de fichiers, serveurs de fichiers | Bases de données, VM, applications critiques |

# ClaimRef

- Le champ `claimRef` est un attribut de la ressource `PersistentVolume` (PV) dans Kubernetes. Il sert à indiquer explicitement qu'un PV est réservé pour un `PersistentVolumeClaim` (PVC) précis, en spécifiant le nom et le namespace de ce PVC. Cela permet de contrôler et de sécuriser l'appariement entre un volume et une demande de volume, en évitant qu'un autre PVC ne vienne réclamer ce volume.
- Fonctionnement
  - Lorsqu'un PVC est créé, Kubernetes tente d'associer automatiquement un PV disponible qui correspond à ses critères (capacité, modes d'accès, `StorageClass`, etc.).
  - Si vous souhaitez lier un PV à un PVC spécifique (pré-binding), vous pouvez :
    - Renseigner le champ `volumeName` dans le PVC pour cibler un PV précis.
    - Renseigner le champ `claimRef` dans le PV pour réserver ce volume à un PVC donné (nom et namespace).
  - Lorsque les deux sont utilisés ensemble (`volumeName` côté PVC et `claimRef` côté PV), cela garantit que seul le PVC ciblé pourra réclamer ce volume, et que ce PVC ne pourra pas être lié à un autre PV.

# Storage Class

- Une Storage Class dans Kubernetes est une ressource qui permet aux administrateurs de définir différentes "classes" de stockage, chacune correspondant à un type, une qualité de service, des performances ou des politiques spécifiques de stockage.
- Elle sert de modèle ou de profil pour le provisionnement dynamique des volumes persistants (PV) en fonction des besoins des applications.
- Utilité
  - **Provisionnement dynamique** : Grâce aux StorageClasses, les volumes persistants peuvent être créés automatiquement dès qu'un PersistentVolumeClaim (PVC) en fait la demande, sans intervention manuelle.
  - **Abstraction et flexibilité** : Les utilisateurs peuvent choisir la classe de stockage la plus adaptée à leur workload (par exemple : SSD pour la performance, HDD pour l'archivage, NFS pour le partage, etc.) simplement en spécifiant le nom de la StorageClass dans leur PVC.
  - **Gestion centralisée** : Les administrateurs définissent les types de stockage disponibles et leurs paramètres (provisionneur, options, politiques de rétention, etc.), ce qui simplifie la gestion et le respect des standards de l'entreprise.

# Storage Class - Propriétés

- **provisioner** : Indique le plugin ou driver utilisé pour créer les volumes (ex : kubernetes.io/aws-ebs, kubernetes.io/gce-pd, csi-driver.example-vendor.example).
- **parameters** : Options spécifiques au backend de stockage (type de disque, IOPS, réplication, zone, etc.).
- **reclaimPolicy** : Définit ce qu'il advient du volume après suppression du PVC (Delete, Retain, etc.).
- **allowVolumeExpansion** : Permet d'agrandir un volume après sa création (true ou false).
- **volumeBindingMode** : Contrôle le moment où le volume est effectivement provisionné (Immediate ou WaitForFirstConsumer).
- **Annotations** : Par exemple, pour définir la StorageClass par défaut du cluster (storageclass.kubernetes.io/is-default-class: "true")
- Exercice: Étudier les exemples des fichiers gce-pv-pvc.yaml et pod-gce-pvc.yaml



# Container Storage Interface (CSI)

- La Container Storage Interface (CSI) est une spécification standardisée, soutenue par la CNCF, qui définit comment un système de stockage doit exposer ses services à un orchestrateur de conteneurs tel que Kubernetes.
  - Son but principal est de permettre l'intégration flexible et indépendante de solutions de stockage (bloc ou fichier) dans les plateformes de gestion de conteneurs, sans nécessiter de modifications du code source de l'orchestrateur.
- Avant CSI, chaque orchestrateur (Kubernetes, Mesos, Cloud Foundry, etc.) possédait ses propres plugins de stockage, souvent intégrés ("in-tree") dans le code principal. Cela posait plusieurs problèmes :
  - Complexité et lenteur pour intégrer de nouveaux drivers ou corriger des bugs (dépendance au cycle de release de Kubernetes).
  - Sécurité et stabilité du core Kubernetes mises à mal par l'intégration de code tiers.
  - Fragmentation de l'écosystème, chaque fournisseur devant développer plusieurs plugins spécifiques à chaque orchestrateur.
- CSI a été conçu pour résoudre ces problèmes en permettant aux fournisseurs de stockage de développer un seul plugin compatible avec tous les orchestrateurs supportant la norme CSI.

# CSI - Fonctionnement

- Un driver CSI (ou plugin CSI) est composé de deux grandes parties :
  - **Controller Plugin** : déployé côté master, il gère la création, suppression, attachement, détachement et snapshot des volumes.
  - **Node Plugin** : déployé sur chaque nœud du cluster, il gère le montage et le démontage effectif des volumes sur les machines hôtes.
- Les communications entre Kubernetes et les plugins CSI se font via des appels gRPC standardisés définis par la spécification CSI.
- Principales fonctionnalités offertes par CSI:
  - Provisionnement dynamique et suppression de volumes persistants.
  - Attachement/détachement de volumes à des nœuds du cluster.
  - Montage/démontage de volumes sur les nœuds.
  - Gestion de snapshots et de clones de volumes.
  - Redimensionnement à chaud de volumes.

# Storage Class - Exercice

- Analyser comment la partie concernant postgresql a été transformée
  - Afin d'utiliser un stockage reparté basé sur microceph
  - Doc: <https://canonical-microceph.readthedocs-hosted.com/en/reef-stable/>
  - Cf. Fichier microceph.yaml
- À partir de l'exemple se trouvant dans le fichier docker-compose.yaml
  - Uniquement la partie concernant nginx
  - Écrire un fichier de configuration kubernetes qui fasse exactement la même chose
  - Quelles observations pouvez-vous faire ?

# StorageClass vs PV

- **Meilleure gestion du binding** : Pour les volumes locaux (local), Kubernetes recommande de créer une StorageClass avec `volumeBindingMode: WaitForFirstConsumer`. Ce mode retarde l'association du volume jusqu'à ce qu'un Pod soit programmé sur un nœud, ce qui permet de prendre en compte toutes les contraintes de placement du Pod (affinité, ressources, etc.) avant d'attacher le volume.
- **Cohérence et portabilité** : Utiliser une StorageClass rend les manifestes plus portables et cohérents, car le même mécanisme s'applique pour tous les types de stockage (cloud, réseau, local).
- **Abstraction et flexibilité** : Cela permet de centraliser la gestion des politiques de stockage (`reclaimPolicy`, options de montage, etc.) et d'offrir différents profils de stockage aux utilisateurs.
- **Compatibilité avec les outils et bonnes pratiques** : La plupart des outils et des guides Kubernetes s'appuient sur l'utilisation des StorageClass pour le provisionnement des volumes, ce qui facilite l'intégration et la maintenance.

# StorageClass pour les volumes locaux

- Pour les volumes de type local, il n'y a pas de provisionnement dynamique natif : les volumes doivent être créés statiquement. Cependant, Kubernetes recommande tout de même de définir une StorageClass avec `provisioner: kubernetes.io/no-provisioner` et `volumeBindingMode: WaitForFirstConsumer`.
- Cela permet d'éviter que le volume soit attaché à un Pod sur un nœud inapproprié, et garantit que le volume local sera utilisé sur le bon nœud, en tenant compte de l'affinité définie sur le PV.
- **Exercice:** Étudier l'exemple du fichier `local-sc-example.yaml`

# PV Controller

- Le contrôleur PersistentVolume (PV) gère le cycle de vie des volumes de stockage persistants dans un cluster Kubernetes.
- Son objectif principal est d'abstraire la gestion du stockage sous-jacent (NFS, iSCSI, disques cloud, etc.) afin que les utilisateurs puissent consommer du stockage sans se soucier de la technologie ou de l'implémentation réelle.
- Le contrôleur s'assure que les volumes sont provisionnés, disponibles, liés à des claims (PVC), utilisés par des pods, puis libérés ou recyclés selon la politique définie.
- Résumé des points clés
  - Le contrôleur PV gère le stockage persistant de façon indépendante du cycle de vie des pods.
  - Il supporte le provisionnement statique (manuel) et dynamique (automatique via StorageClass).
  - Le binding entre PV et PVC est exclusif : un PV ne peut être lié qu'à un seul PVC à la fois.
  - La politique de rétention (Retain, Recycle, Delete) définit le comportement du volume après suppression du claim.
  - Les pods consomment le stockage via les PVC, qui sont liés aux PV par le contrôleur.

# PV Controller - Fonctionnement

- **Provisionnement statique** : L'administrateur du cluster crée manuellement des objets PersistentVolume, qui décrivent les détails du stockage disponible (capacité, type, accès, etc.).
- **Provisionnement dynamique** : Si aucun PV statique ne correspond à une demande (PVC), Kubernetes peut automatiquement créer un PV grâce à un StorageClass, selon les besoins du claim.
- **Binding (liaison)** : Lorsqu'un utilisateur crée un PersistentVolumeClaim (PVC), le contrôleur recherche un PV compatible (taille, modes d'accès, StorageClass, etc.). Si un PV est trouvé ou créé dynamiquement, il est lié au PVC de façon exclusive.
- **Utilisation** : Un pod peut ensuite utiliser ce stockage en déclarant le PVC dans sa section volumes. Le volume sera monté dans le pod selon le mode d'accès demandé.
- **Libération et recyclage** : Lorsque le PVC est supprimé, le contrôleur applique la politique de rétention du PV (Retain, Recycle, ou Delete). Selon le cas, le volume peut être conservé, nettoyé ou supprimé automatiquement.

The page features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner is decorated with a small teal circle, a medium pink circle, a tiny dark purple dot, a medium pink circle, a large teal circle, and a small purple circle. The text 'Volumes spéciaux' is centered in a dark purple, sans-serif font.

# Volumes spéciaux



# ConfigMap

- Il peut être utile d'éviter de coder en dur certains paramètres dans le fichier de configuration
- Afin d'apporter plus de flexibilité, Kubernetes supporte la notion de configmap
  - Qui représente un ensemble de clef/valeur utilisé dans la configuration
  - *kubectl create configmap logger --from-literal=log\_level=debug*
  - *kubectl create configmap logger --from-file=<config-file>*
- Pour lister les configmaps
  - *kubectl get configmaps*
- Pour afficher une configmap
  - *kubectl get configmap/logger -o yaml*
- Pour éditer une configmap
  - *kubectl edit configmap/logger*
- Les valeurs peuvent être changées dynamiquement
- Cf. hw-configmap.yaml pour l'exemple

# Secret

- Les Secrets sont un type de ressource Kubernetes qui vous permet de stocker et de gérer des informations sensibles. Contrairement aux ConfigMaps qui sont destinés aux données non confidentielles, les Secrets sont spécifiquement destinés aux données sensibles.
- Les secrets sont stockés dans Kubernetes en tant qu'objets et peuvent être chiffrés dans le cluster, bénéficiant d'une couche de sécurité supplémentaire.
- Les volumes secrets sont montés dans les Pods de la même manière que les volumes ordinaires, mais leur contenu est constitué des données sensibles stockées dans l'objet Kubernetes Secret.
  - Cela permet aux conteneurs d'accéder à des informations sensibles sans les exposer dans la spécification du pod ou dans le code source.
- Pour créer un secret:
  - Commandes similaires à celles du configMap (juste remplacer le mot clef *configMap* par *secret*)
  - Cf. hw-secrets.yaml pour l'exemple

# Secret - Limitations

- Par défaut, les secrets sont stockés en base64 dans etcd, ce qui n'est pas du chiffrement mais un simple encodage réversible. Toute personne ayant accès à etcd ou à l'API Kubernetes peut facilement lire les secrets.
- Le chiffrement au repos est possible, mais il nécessite une configuration manuelle et la gestion séparée des clés de chiffrement. Une mauvaise configuration peut laisser les secrets exposés même si l'on pense être protégé.
- Même avec le chiffrement activé, les secrets sont déchiffrés en mémoire pour les applications, ce qui laisse une surface d'attaque résiduelle.
- Kubernetes utilise RBAC pour gérer l'accès aux secrets, mais ce contrôle est souvent trop large : un rôle peut donner accès à tous les secrets d'un namespace, ce qui ne permet pas une gestion fine ou granulaire des droits.
- Une mauvaise configuration RBAC peut exposer des secrets à des utilisateurs ou services non autorisés.

# Secret - Limitations

- Les secrets Kubernetes sont conçus comme immuables : ils ne peuvent pas être modifiés ou versionnés facilement une fois créés.
- La rotation (renouvellement régulier) des secrets est complexe : il faut supprimer et recréer les secrets, puis redémarrer les Pods pour qu'ils prennent en compte les nouvelles valeurs.
- Il n'existe pas de mécanisme natif pour générer ou distribuer des secrets dynamiques et éphémères.
- Kubernetes ne fournit pas de journalisation détaillée des accès ou modifications des secrets.
- Il est difficile de savoir qui a accédé à un secret, quand et pourquoi, ce qui pose problème pour la conformité et la détection d'incidents de sécurité.
- Les exigences de conformité (ex : data residency, souveraineté, audits) sont difficiles à satisfaire avec les secrets natifs Kubernetes, surtout dans des environnements multi-cloud ou réglementés.
- **À grande échelle ou dans des environnements sensibles, ces limitations deviennent critiques => Il faut envisager alors des solutions comme HashiCorp Vault.**

# Le Réseau

# Réseau Kubernetes

- Lors de l'exécution de Kubernetes dans un environnement avec des limites de réseau strictes, comme un centre de données sur site avec des pare-feu de réseau physique ou des réseaux virtuels dans le Cloud public, il est utile de connaître les ports et les protocoles utilisés par les composants de Kubernetes.

## Control plane

| Protocol | Direction | Port Range | Purpose                 | Used By              |
|----------|-----------|------------|-------------------------|----------------------|
| TCP      | Inbound   | 6443       | Kubernetes API server   | All                  |
| TCP      | Inbound   | 2379-2380  | etcd server client API  | kube-apiserver, etcd |
| TCP      | Inbound   | 10250      | Kubelet API             | Self, Control plane  |
| TCP      | Inbound   | 10259      | kube-scheduler          | Self                 |
| TCP      | Inbound   | 10257      | kube-controller-manager | Self                 |

## Worker node(s)

| Protocol | Direction | Port Range  | Purpose            | Used By              |
|----------|-----------|-------------|--------------------|----------------------|
| TCP      | Inbound   | 10250       | Kubelet API        | Self, Control plane  |
| TCP      | Inbound   | 10256       | kube-proxy         | Self, Load balancers |
| TCP      | Inbound   | 30000-32767 | NodePort Services† | All                  |

<https://kubernetes.io/docs/reference/networking/ports-and-protocols/>

# Container Network Interface

- CNI (Container Network Interface) est une norme open source qui définit une interface standardisée pour configurer les réseaux des conteneurs dans Kubernetes.
- Son rôle principal est de gérer la connectivité réseau des Pods en allouant des adresses IP, en créant des interfaces réseau et en configurant les règles de routage.
  - Il faut garder à l'esprit que la configuration manuelle des pods pour permettre la communication est très fastidieuse et sujette à erreurs
  - L'idéal serait de pouvoir déléguer cette responsabilité à un composant ayant un comportement standard => CNI
- CNI permet à Kubernetes de :
  - Faciliter la communication entre Pods (même sur des nœuds différents) et avec l'extérieur.
  - Isoler les réseaux via des namespaces réseau Linux.
  - Intégrer des solutions de réseau avancées (Calico, Cilium, Flannel, etc.) via des plugins.

# CNI – Ajout d'un Pod

- **Création d'un espace de noms réseau dédié** : Le CNI crée un "network namespace" pour le pod, isolant son environnement réseau de celui du nœud et des autres pods.
- **Attribution d'une adresse IP unique** : Grâce à un sous-plugin appelé IPAM (IP Address Management), le CNI alloue une adresse IP au pod, généralement unique dans le cluster ou le sous-réseau du nœud.
- **Création et configuration des interfaces réseau** : Le CNI crée une paire d'interfaces virtuelles (veth pair) : une extrémité dans le namespace du pod, l'autre sur le nœud. Il connecte ensuite le pod au réseau du cluster, configure les routes et les règles nécessaires pour permettre la communication.
- **Application des politiques réseau** : Si des plugins CNI additionnels sont utilisés (ex : pour la sécurité ou la gestion du trafic), ils appliquent les règles d'accès, de filtrage ou de routage conformément à la configuration du cluster.
- **Signalement de la disponibilité réseau** : Une fois la configuration terminée, le CNI informe le kubelet que le pod est prêt à communiquer sur le réseau du cluster.

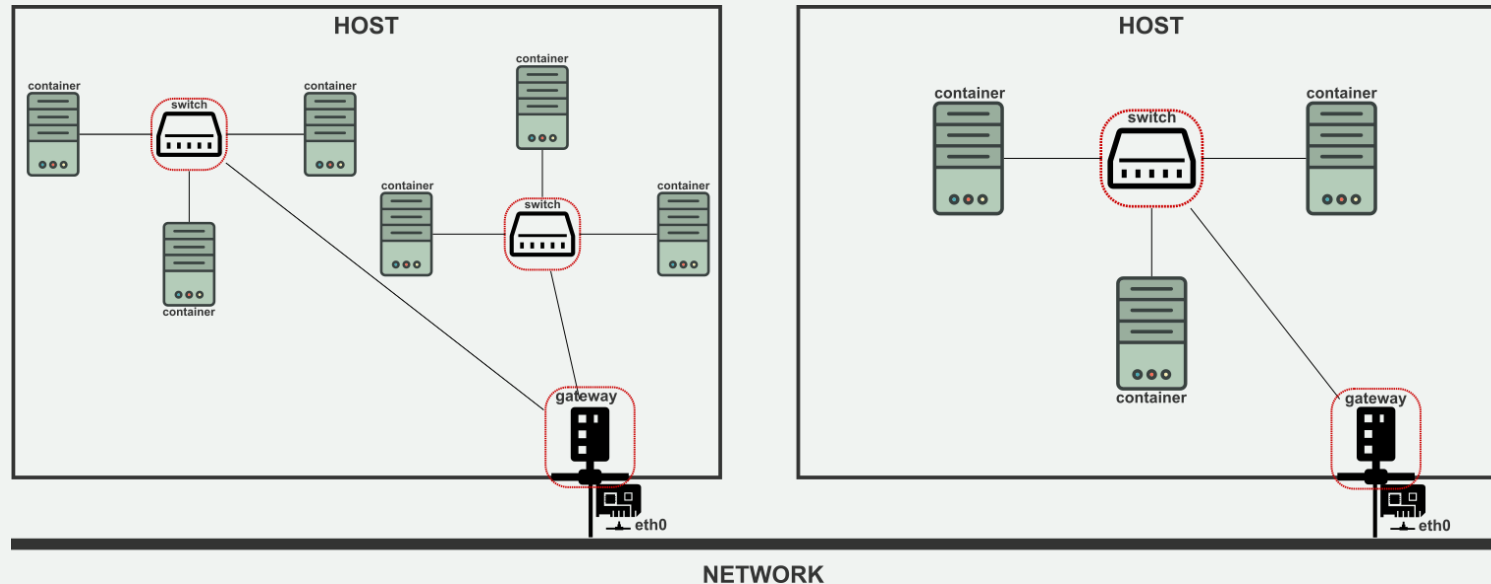


# CNI – Suppression d'un Pod

- **Suppression des interfaces réseau** : Le CNI supprime les interfaces virtuelles (veth pairs) qui reliaient le pod au réseau du cluster, retirant ainsi le pod du réseau.
- **Libération de l'adresse IP** : Le sous-plugin IPAM (IP Address Management) libère l'adresse IP qui avait été attribuée au pod, la rendant disponible pour de futurs pods.
- **Nettoyage des règles de routage et de filtrage** : Le CNI retire toutes les routes, règles iptables ou configurations réseau spécifiques qui avaient été appliquées pour permettre la communication du pod.
- **Suppression des configurations additionnelles** : Si des plugins CNI additionnels (comme port mapping, tuning, policies) avaient appliqué des réglages, ils sont également appelés pour supprimer ces paramètres.

# CNI - Illustration

- Les plugins CNI utilisent des technologies de type **Software-Defined Networking (SDN)** pour connecter les pods entre eux, gérer l'adressage IP, le routage et parfois l'isolation réseau.
- Certains plugins, comme Flannel ou Weave, créent des réseaux overlay qui encapsulent le trafic pour simuler un switch virtuel distribué entre les nœuds du cluster.
- D'autres, comme Calico, mettent en œuvre un routage de niveau 3 (Layer 3), distribuant dynamiquement les routes entre nœuds et pods, ce qui revient à fournir un routeur virtuel logiciel entre les workloads.
- Des solutions comme Open vSwitch (utilisé via certains plugins CNI) sont explicitement des switches virtuels capables de gérer le trafic, l'isolation et les règles de flux réseau au sein du cluster.



# CNI – Quelques implantations

| Type                 | Exemples       | Cas d'usage                                       |
|----------------------|----------------|---------------------------------------------------|
| <b>Overlay</b>       | Flannel, Weave | Réseau virtuel superposé (VXLAN, etc.).           |
| <b>Routage natif</b> | Calico, Cilium | Routage direct via BGP ou eBPF.                   |
| <b>Cloud intégré</b> | AWS VPC CNI    | Intégration native avec le cloud (AWS/GCP/Azure). |
| <b>Multi-réseaux</b> | Multus         | Attachement de multiples interfaces par Pod.      |

# CNI - Critères de sélection

- **Performance réseau**

- Latence, débit, consommation CPU : certains plugins (ex : Cilium, Calico) sont optimisés pour la performance, tandis que d'autres (ex : Flannel) privilégient la simplicité au détriment de la rapidité.
- Prise en charge de technologies modernes (ex : eBPF dans Cilium) pour améliorer le routage et la sécurité sans surcoût.

- **Sécurité et politiques réseau**

- Capacité à appliquer des politiques réseau avancées (filtrage, segmentation, micro-segmentation).
- Support des Network Policies Kubernetes et, pour certains plugins, de fonctionnalités avancées (audit, chiffrement, inspection applicative).

- **Scalabilité et résilience**

- Capacité à gérer des clusters de grande taille sans perte de performance ou de stabilité.
- Gestion efficace des adresses IP (overlay vs flat network).

- **Compatibilité et intégration**

- Intégration avec le cloud provider ou la plateforme Kubernetes utilisée (ex : AWS CNI pour EKS, Azure CNI pour AKS).
- Support multi-environnement (cloud, on-prem, hybride)

# CNI - Critères de sélection

- **Simplicité de déploiement et d'administration**

- Facilité d'installation, de configuration et de maintenance (ex : Flannel et Weave sont réputés pour leur simplicité).
- Documentation, outils de gestion, support communautaire ou entreprise.

- **Fonctionnalités avancées**

- Support du multi-réseau (Multus), du chaining de plugins, de l'isolation avancée, du monitoring réseau, etc.
- Possibilité de chaîner plusieurs plugins pour combiner les avantages de chacun (ex : performance + sécurité).

- **Support et écosystème**

- Disponibilité du support professionnel et de l'expertise, notamment pour les environnements critiques ou réglementés.
- Dynamisme de la communauté, fréquence des mises à jour et compatibilité avec les dernières versions de Kubernetes.

# CoreDNS

- CoreDNS est un serveur DNS open source, flexible et extensible, conçu pour répondre aux besoins des environnements cloud natifs comme Kubernetes.
- Il est écrit en Go, maintenu par la CNCF, et est devenu le serveur DNS par défaut dans Kubernetes depuis la version 1.13.
- Avantages pour Kubernetes
  - **Composant critique** : Il assure la découverte des services et la connectivité entre pods/applications au sein du cluster.
  - **Personnalisation facile** : Le fichier de configuration (Corefile) permet d'ajouter, retirer ou configurer des plugins selon les besoins (cache, logs, forwarding, etc.).
  - **Performance et fiabilité** : CoreDNS peut être surveillé via Prometheus et s'intègre avec des solutions de cache local (NodeLocal DNS) pour améliorer la rapidité et la résilience de la résolution DNS.
- <https://coredns.io/>

# CoreDNS - Fonctionnement

- **Résolution DNS interne au cluster** : CoreDNS gère la résolution des noms de services et de pods dans Kubernetes. Lorsqu'un pod cherche à contacter un service par son nom (ex : nginx-svc), la requête DNS est envoyée à CoreDNS, qui interroge l'API Kubernetes pour obtenir l'adresse IP correspondante et la retourne au client.
- **Extensible par plugins** : CoreDNS fonctionne grâce à une architecture modulaire basée sur des plugins. Chaque plugin apporte une fonctionnalité (cache, forwarding, metrics Prometheus, intégration Kubernetes, etc.), ce qui permet d'adapter le serveur à de nombreux cas d'usage.
- **Support de multiples protocoles** : CoreDNS peut répondre à des requêtes DNS via UDP, TCP, DNS sur TLS (DoT), DNS sur HTTPS (DoH), DNS sur QUIC, etc.
- **Intégration avec Kubernetes** : Le plugin kubernetes permet à CoreDNS de dialoguer avec l'API du cluster pour fournir la découverte de services et la résolution DNS interne.

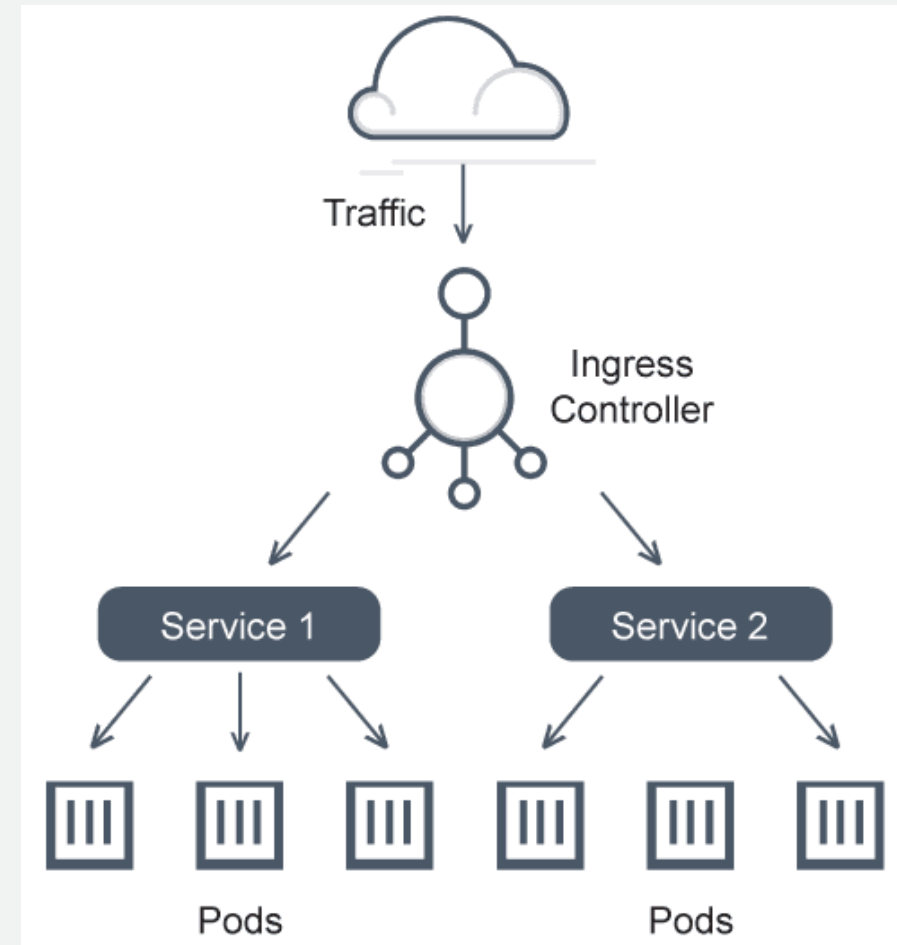
# CoreDNS - Nommage

- Le nom DNS complet (FQDN) d'un service suit ce schéma :
  - `<service-name>.<namespace>.svc.cluster.local`
  - `<service-name>` : nom du service (ex : db)
  - `<namespace>` : namespace du service (ex : default, prod, etc.)
  - `svc.cluster.local` : suffixe standard du cluster (personnalisable, mais c'est la valeur par défaut)
- **Exemple**
  - Les différents backends (Backend replica 1/2/3) pourraient être dans un namespace backend.
  - La base de données (DB ClusterIP) pourrait être dans un namespace *database*.
  - Pour qu'un backend accède à la base de données, il utiliserait l'adresse DNS :
    - `db.database.svc.cluster.local`
  - Chaque Pod reçoit un fichier `/etc/resolv.conf` configuré par le kubelet, contenant une liste de recherche DNS



# Ingress

- Un ingress est un service de type load-balancer particulier qui expose les routes HTTP et HTTPS de l'extérieur du cluster à des services au sein du cluster.
- Il agit comme un reverse-proxy et peut être configuré pour donner aux services des URLs accessibles de l'extérieur (routes)
- Quelques implantations:
  - NGINX
  - Traefik
  - AWS ALB
  - Google GCE



# Ingress: Rôle

- **Routage HTTP/HTTPS** : Il gère l'accès externe aux services d'un cluster, généralement HTTP/HTTPS, et peut assurer l'équilibrage de la charge, la terminaison SSL et l'hébergement virtuel basé sur le nom.
- **Gestion centralisée** : Au lieu de devoir gérer l'accès externe au sein de chaque service, le contrôleur d'entrée fournit un système de routage centralisé.
- **Efficacité** : Il vous permet de consolider vos règles de routage en une seule ressource, car il peut desservir le trafic de plusieurs services sous la même adresse IP.



# Ingress: Comportement

- **Écoute des ressources** : Le contrôleur d'entrée surveille en permanence les modifications apportées aux ressources d'entrée et met à jour sa configuration en conséquence.
- **Configuration de la passerelle** : Sur la base des spécifications d'entrée, le contrôleur d'entrée configure un équilibreur de charge ou d'un reverse proxy pour acheminer le trafic vers les services appropriés.
- **Terminaison SSL/TLS** : Il s'occupe souvent de la terminaison SSL/TLS pour le trafic crypté, déchargeant ainsi les services individuels de cette tâche.
- **Annotations** : Le contrôleur d'entrée peut être personnalisé à l'aide d'annotations dans le fichier Ingress. Ces annotations peuvent contrôler des comportements tels que la réécriture d'URL, la limitation de débit et la configuration SSL.
- **Routage basé sur le chemin d'accès** : Les règles d'entrée peuvent diriger le trafic en fonction du chemin d'accès à l'URL, en envoyant des chemins différents à des services différents.



# Ingress Controller

- L'Ingress Controller est un composant essentiel pour exposer les services Kubernetes vers l'extérieur du cluster, principalement pour le trafic HTTP et HTTPS.
  - Il est généralement installé par défaut chez tous les fournisseurs cloud
  - Il doit être configuré de la même manière que les autres ressources kubernetes
- Il agit comme un reverse proxy et un load balancer : il reçoit les requêtes provenant de l'extérieur, applique les règles de routage définies dans les ressources Ingress, puis achemine le trafic vers les bons services/pods internes.
- Sans Ingress Controller, la création d'une ressource Ingress n'a aucun effet : il faut impérativement déployer un contrôleur (par exemple NGINX, Traefik, HAProxy, AWS ALB, etc.) pour que le trafic soit effectivement routé.

# Ingress Controller - Fonctionnement

- **Surveillance des ressources Ingress** : Le contrôleur surveille en continu les objets Ingress du cluster. À chaque création, modification ou suppression d'un Ingress, il met à jour sa configuration interne pour refléter les nouvelles règles de routage.
- **Routage et équilibrage de charge** : Il agit comme point d'entrée unique pour le trafic HTTP/HTTPS, effectue l'équilibrage de charge entre les pods cibles, et permet d'utiliser un seul point d'accès (IP ou DNS) pour plusieurs services internes.
- **Fonctionnalités avancées** : Selon le contrôleur choisi, il peut offrir : terminaison SSL/TLS, redirections, gestion des headers, authentification, réécriture d'URL, etc..
- **Gestion des classes d'Ingress** : Plusieurs Ingress Controllers peuvent coexister dans un cluster. On utilise alors le champ `ingressClassName` dans les ressources Ingress pour indiquer quel contrôleur doit prendre en charge chaque Ingress.
- Exemples: `hw-ingress-path.yaml` et `hw-ingress-host.yaml`

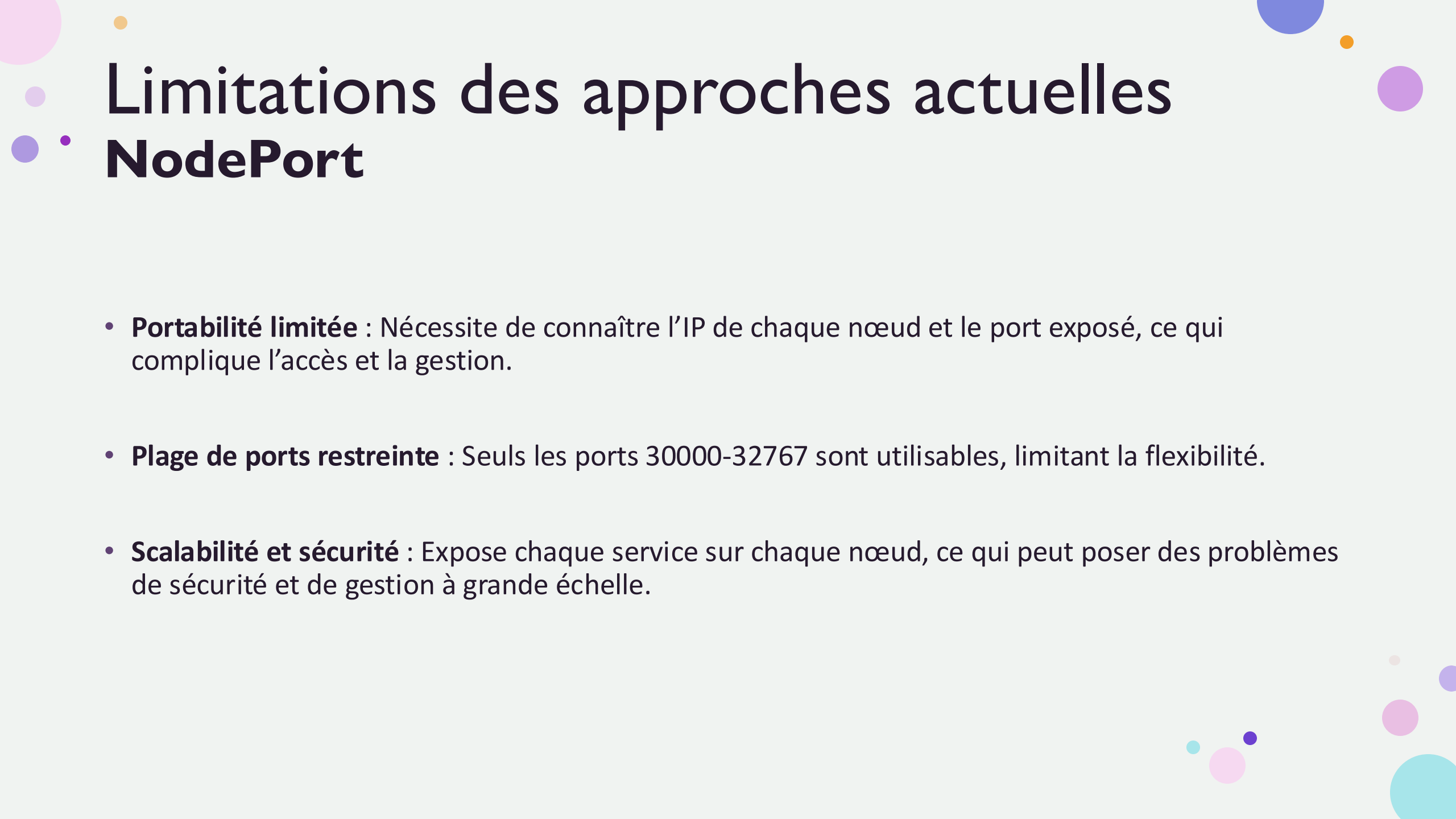
# Ingress: Exercice

- Étudier le contenu du fichier hw-ingress-path.yaml
  - Modifier le fichier hw-ingress.yaml de manière à associer un chemin à un service qui expose le déploiement hello2
  - Exécuter le fichier et tester
- Étudier le contenu du fichier hw-ingress-host.yaml
  - Quelle différence voyez-vous ?
  - Modifier le fichier pour exposer le déploiement hello2
  - Exécuter le fichier et tester

# Limitations des approches actuelles

## **Vue d'ensemble**

- Les approches classiques (NodePort, LoadBalancer, Ingress) sont suffisantes pour des besoins simples, mais montrent rapidement leurs limites pour des architectures modernes, sécurisées et évolutives.
- L'API Gateway répond à ces enjeux en offrant une gestion centralisée, riche et extensible du trafic entrant, adaptée aux environnements Kubernetes modernes et aux exigences des applications distribuées.



# Limitations des approches actuelles

## NodePort

- **Portabilité limitée** : Nécessite de connaître l'IP de chaque nœud et le port exposé, ce qui complique l'accès et la gestion.
- **Plage de ports restreinte** : Seuls les ports 30000-32767 sont utilisables, limitant la flexibilité.
- **Scalabilité et sécurité** : Expose chaque service sur chaque nœud, ce qui peut poser des problèmes de sécurité et de gestion à grande échelle.



# Limitations des approches actuelles

## LoadBalancer

- **Dépendance au cloud** : Fonctionne bien sur les clouds publics qui provisionnent automatiquement des load balancers, mais nécessite des solutions tierces (ex : MetalLB) en on-premise ou sur bare-metal.
- **Coût et gestion** : Attribue un load balancer par service, ce qui peut générer des coûts et une complexité inutiles pour des architectures microservices avec de nombreux services.
- **Peu de contrôle applicatif** : Opère principalement sur la couche réseau (L4), sans gestion fine des requêtes HTTP ou des politiques de sécurité avancées.

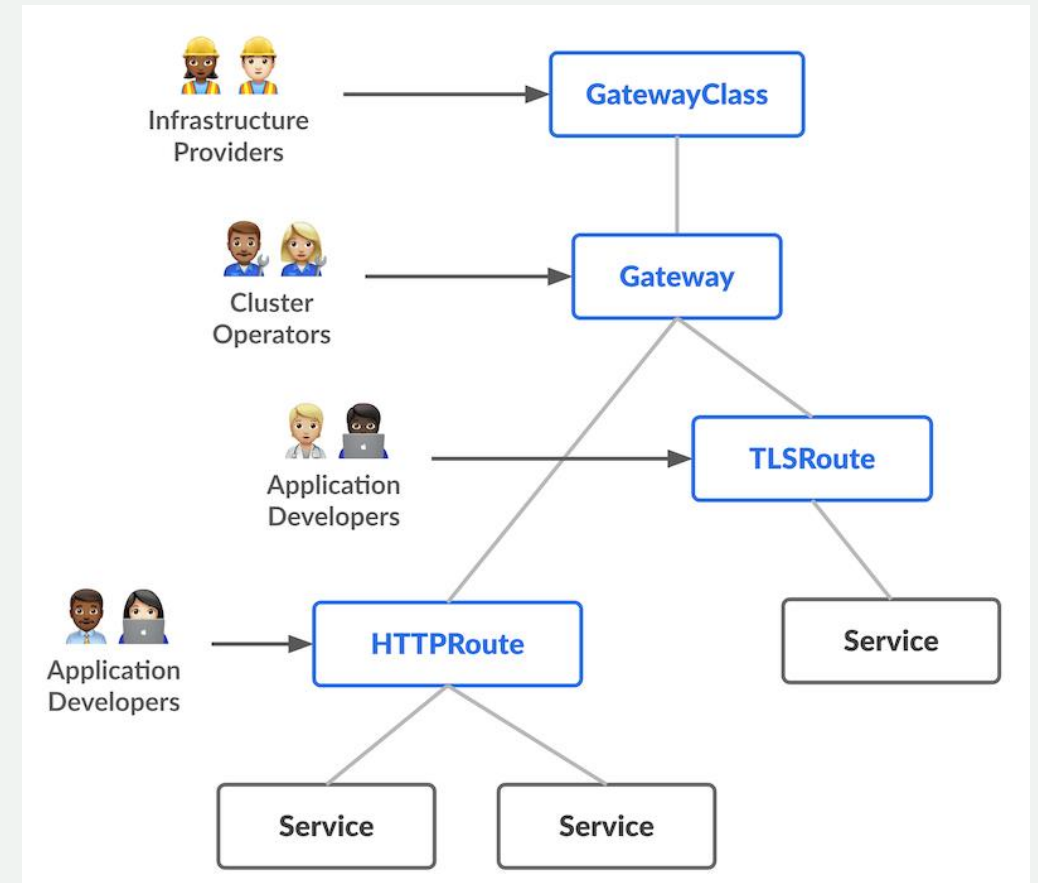
# Limitations des approches actuelles

## Ingress

- **Fonctionnalités limitées** : Principalement conçu pour le routage HTTP/S de base. Les fonctionnalités avancées (authentification, rate limiting, manipulation d'en-têtes, etc.) nécessitent des annotations spécifiques à chaque contrôleur, ce qui nuit à la portabilité et à la standardisation.
- **Manque d'extensibilité et de support multi-protocoles** : L'Ingress standard ne gère pas nativement d'autres protocoles (gRPC, WebSockets, TCP, etc.) et ne permet pas de scénarios de routage complexes (split, canary, etc.).
- **Gestion du cycle de vie et RBAC** : Difficulté à contrôler finement qui peut modifier les règles de routage, absence de séparation claire des responsabilités (pas de RBAC granulaire sur les routes).
- **Pas de support cross-namespace** : Impossible de référencer des services ou secrets dans d'autres namespaces de manière sécurisée.
- **Vendor lock-in** : Les annotations et comportements diffèrent selon les implémentations (NGINX, HAProxy, Traefik...), rendant les configurations peu portables.

# Nouvelle approche: API Gateway

- **Un point d'entrée unique** pour tout le trafic, centralisant la gestion et la sécurité des accès.
- **Des fonctionnalités avancées** : authentification, autorisation, rate limiting, agrégation de réponses, transformation des requêtes/réponses, monitoring, etc.
- **Support multi-protocoles** et routage applicatif avancé (split, canary, header-based, etc.).
- **Extensibilité** : plugins, intégration avec des outils de sécurité, observabilité, etc.
- **Séparation des rôles** et RBAC : gestion des routes, des politiques et des accès par équipe ou par application.
- **Portabilité et standardisation** : les nouveaux standards comme le Gateway API Kubernetes visent à uniformiser et enrichir ces usages pour tous les environnements (cloud, on-prem, hybride).

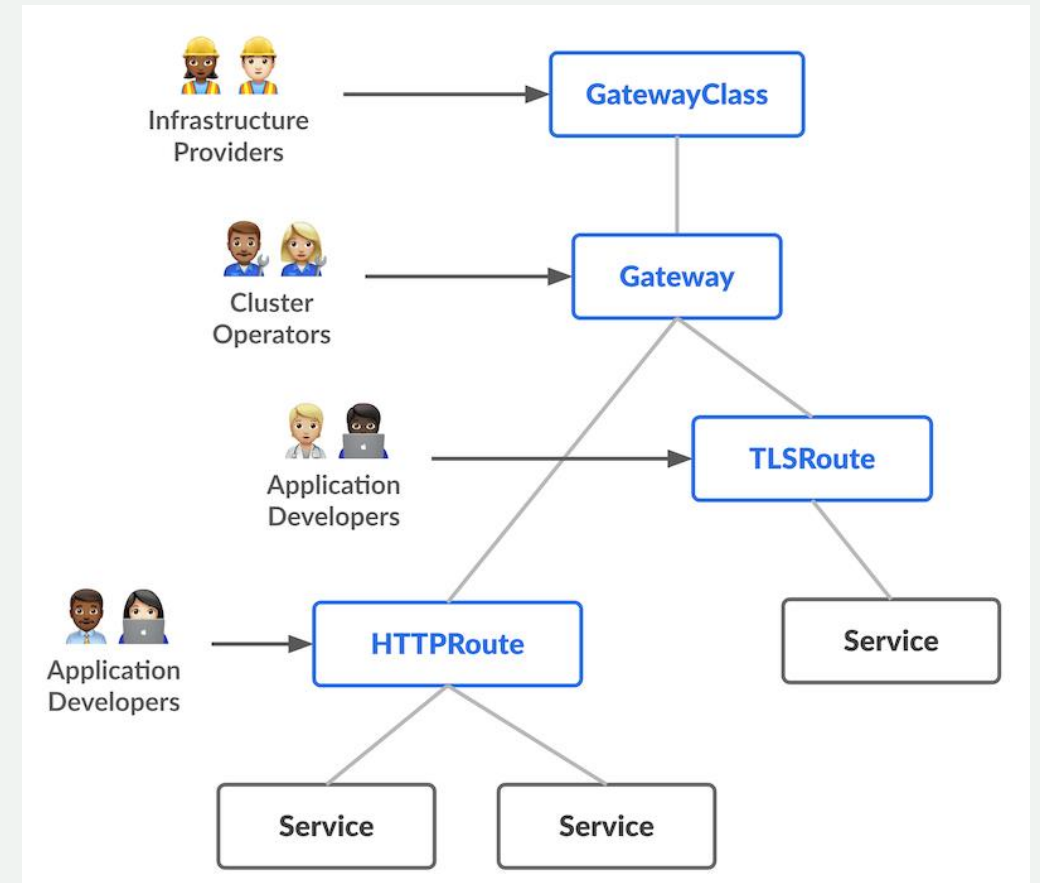


# API Gateway : Motivations

- **Centralisation** : Simplifie la communication client en offrant un point d'accès unique, au lieu d'exposer chaque service individuellement.
- **Sécurité** : Ajoute une couche de protection (authentification, filtrage, anti-DDoS) avant d'atteindre les services internes.
- **Optimisation** : Permet la mise en cache, la compression, l'agrégation de réponses, l'équilibrage de charge et la gestion fine du trafic.
- **Observabilité** : Centralise les logs, les métriques et la surveillance des accès API pour l'ensemble du système.
- **Facilite la gestion et l'évolution** : Permet de modifier les règles de routage, d'introduire de nouveaux services ou de faire du canary release sans changer la configuration des clients.

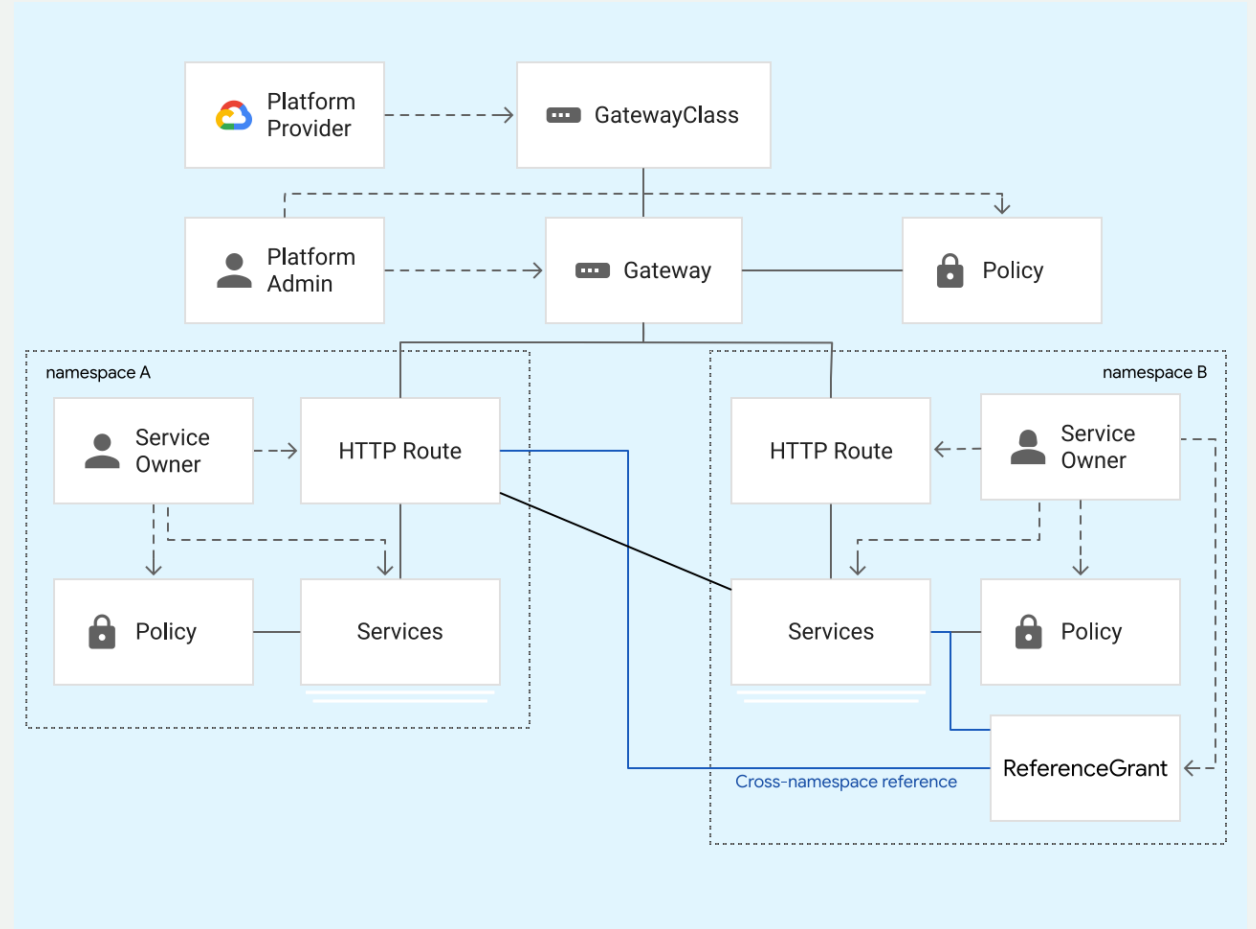
# API Gateway : Concepts

| Ressource           | Rôle/Description                                                                                                                 |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <b>GatewayClass</b> | Décrit une classe de gateways, gérée par un contrôleur spécifique (ex : Istio, NGINX, Envoy, etc.)                               |
| <b>Gateway</b>      | Instance concrète d'une gateway (point d'entrée réseau, ex : load balancer, proxy)                                               |
| <b>Route</b>        | Règles de routage (HTTPRoute, GRPCRoute, TCPRoute, etc.), attachées à une Gateway et pointant vers des backends (Services, etc.) |



# API Gateway : Exercice

- Étudier le contenu du fichier hw-gateway.yaml
- Modifier ce fichier de manière à associer un chemin à un service qui expose le déploiement hello2
- Exécuter le fichier et tester



# Les quotas

# Quotas

- Kubernetes permet de spécifier des quotas pour au niveau de:
  - Un cluster
  - Un déploiement
  - Un nœud
  - Un pod
- Cf. <https://kubernetes.io/docs/concepts/policy/resource-quotas/>
- Exemple de quotas appliqués un espace de nommage
  - \$k apply -f quotas.yaml -n training
- Exemple de quotas appliqués à un déploiement
  - \$k apply -f hw-quotas.yaml
- Exercice:
  - Exécuter cette dernière configuration
  - Qu'observez-vous ?

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.nvidia.com/gpu: 4
```



# Quotas: Exercice

- **Objectif** : Créer un espace de noms avec des quotas de ressources spécifiques et tester les quotas en déployant des pods qui dépassent les limites fixées.
- Étape 1: Créer un namespace 'quota-ns'
- Étape 2: Associer un quota à cet espace de nommage avec les caractéristiques suivantes
  - 1 CPU
  - 1 G de RAM
  - 1 limite de 2 CPU
  - 1 limite de 2G de RAM
- Étape 3 : Tester le quota
  - Essayez de déployer des Pods ou des Déploiements qui demandent et limitent les ressources dans le quota défini.
  - Par exemple, créez un déploiement qui demande plus de CPU ou de mémoire que le quota ne le permet

# Monitoring

# Monitoring des ressources

- Kubernetes permet de poser des sondes afin de surveiller l'usage des ressources
- Le kubelet utilise des sondes de disponibilité "readiness" pour savoir quand un conteneur est prêt à commencer à accepter du trafic.
  - Un Pod est considéré comme prêt lorsque tous ses conteneurs sont prêts. Lorsqu'un pod n'est pas prêt, il est retiré du load-balancing.
- De nombreuses applications fonctionnant pendant de longues périodes finissent par casser, et nécessitent d'être redémarrées. Kubernetes fournit des sondes de "liveness" pour détecter et remédier à de telles situations.
- **Exercice:**
  - Exécuter la configuration hw-probes.yaml et lister les déploiements
  - Modifier la configuration de manière à ce que l'observation du chargement (readiness) donne une erreur
  - Vérifier les informations du pod avec la commande suivante:
    - `$kubectl describe po/<pod-name>`
  - Faire de même avec la propriété de liveness

# Readiness Probe : Explications

- Propriétés:
  - **initialDelaySeconds : 5** : Ceci définit le délai initial pour la sonde d'état de préparation à 5 secondes. Cela signifie que le kubelet attendra 5 secondes après le démarrage du conteneur avant d'effectuer la première vérification.
  - **timeoutSeconds : 1** : Ceci définit le délai d'attente pour la sonde d'état de préparation à 1 seconde. Si la sonde prend plus d'une seconde, elle sera considérée comme un échec.
  - **httpGet** : Définit une requête HTTP GET utilisée comme sonde réelle. D'autres types de sondes incluent les sockets TCP et l'exécution d'une commande à l'intérieur du conteneur.
  - **path: /** : Ceci définit le chemin d'accès à l'adresse IP du conteneur pour effectuer la requête HTTP GET. Dans ce cas, il s'agit du chemin racine (/).
  - **port : 80** : Il s'agit du numéro de port sur lequel le conteneur écoute les requêtes HTTP. La sonde enverra une requête HTTP GET à ce port.

# Readiness Probe : Explications

- Fonctionnement:
  - Lorsque Kubernetes démarre un conteneur, le kubelet attend 5 secondes (comme spécifié par `initialDelaySeconds`). Il envoie ensuite une requête HTTP GET sur le chemin / du port 80 du conteneur. Si la sonde reçoit une réponse en moins d'une seconde (comme spécifié par `timeoutSeconds`) et que la réponse est un code de succès (HTTP 200-399), le conteneur est marqué comme prêt et commencera à recevoir du trafic des services.
  - En cas d'échec de la sonde (par exemple, en cas de dépassement de délai, si le chemin n'est pas trouvé ou si la sonde renvoie un code d'état HTTP de non-réussite), le kubelet réessaie périodiquement la sonde. Le conteneur ne recevra pas de trafic des services jusqu'à ce que la sonde de préparation réussisse.
- Importance:
  - Cette sonde est essentielle pour garantir que le trafic n'est envoyé qu'aux conteneurs entièrement démarrés et prêts à traiter les demandes. Cela est particulièrement important pour les applications dont le processus de démarrage prend un certain temps avant d'être pleinement opérationnel.

# Liveness Probe: Explications

- Propriétés:

- **initialDelaySeconds : 5** : Ce paramètre indique à Kubernetes d'attendre 5 secondes après le démarrage du conteneur avant d'effectuer la première vérification. Ce délai permet à l'application dans le conteneur de démarrer avant de commencer à vérifier sa santé.
- **periodSeconds : 5** : Il s'agit de la fréquence de l'analyse de l'état de santé. Elle indique que l'analyse doit être effectuée toutes les 5 secondes.
- **timeoutSeconds : 1** : Cette valeur définit le délai d'attente pour l'analyse de l'état de conservation à 1 seconde. Si l'analyse prend plus d'une seconde, elle est considérée comme un échec.
- **failureThreshold : 2** : C'est le nombre de fois que la sonde peut échouer avant que Kubernetes ne décide de redémarrer le conteneur. Dans ce cas, si la sonde d'intégrité échoue 2 fois de suite, Kubernetes redémarrera le conteneur.
- **httpGet** : Cette section configure la sonde comme une requête HTTP GET.
- **path : /** : Le chemin d'accès à l'adresse IP du conteneur par rapport auquel la requête HTTP GET sera effectuée. Dans ce cas, il s'agit du chemin racine (/).
- **port : 80** : Il s'agit du port du conteneur sur lequel le serveur HTTP écoute et sur lequel la requête HTTP GET pour la sonde sera envoyée.

# Liveness Probe: Explications

- Comportement:
  - Une fois que le conteneur démarre, Kubernetes attend 5 secondes (conformément à `initialDelaySeconds`), puis commence à exécuter la requête HTTP GET au chemin / sur le port 80 du conteneur toutes les 5 secondes (`periodSeconds`). Si la sonde n'obtient pas de réponse positive (HTTP 200-399) dans un délai d'une seconde (`timeoutSeconds`) ou si le point d'accès est inaccessible, on considère qu'il y a eu échec. Si 2 échecs consécutifs (`failureThreshold`) se produisent, Kubernetes redémarre le conteneur.
- Importance:
  - Les sondes d'intégrité sont essentielles pour détecter les conteneurs qui ne fonctionnent pas correctement mais qui sont toujours en cours d'exécution, un état parfois appelé "état de zombie". Dans ce cas, le redémarrage du conteneur peut résoudre le problème. La sonde de vivacité garantit que les conteneurs devant être redémarrés sont identifiés et gérés automatiquement, ce qui contribue à maintenir la santé globale des applications fonctionnant dans le cluster.

# Dashboard

- Kubernetes fournit des outils afin de gérer / surveiller les ressources d'un cluster
- Il faut pour cela d'abord installer quelques services supplémentaires (avec helm)

- Installation

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/  
helm repo update  
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard \\  
  --create-namespace --namespace kubernetes-dashboard
```

- Configuration

```
kubectl create serviceaccount admin-user -n kubernetes-dashboard  
kubectl create clusterrolebinding admin-user \  
  --clusterrole=cluster-admin \  
  --serviceaccount=kubernetes-dashboard:admin-user  
kubectl -n kubernetes-dashboard create token admin-user  
kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard 8443:443
```



# Monitoring Avancé

- Afin de monitorer l'ensemble du cluster, il est nécessaire d'installer 3 composants
  - CAdvisor: pour produire les métriques au niveau de chaque conteneur
  - Prometheus: pour collecter l'ensemble des métriques
  - Grafana: pour visualiser les métriques collectées par Prometheus
  - Opentelemetry: pour corréler les traces
- L'installation de ces outils peut être fastidieux
  - Par chance, helm nous facilite grandement la vie :-)
  - Opentelemetry s'installe à part

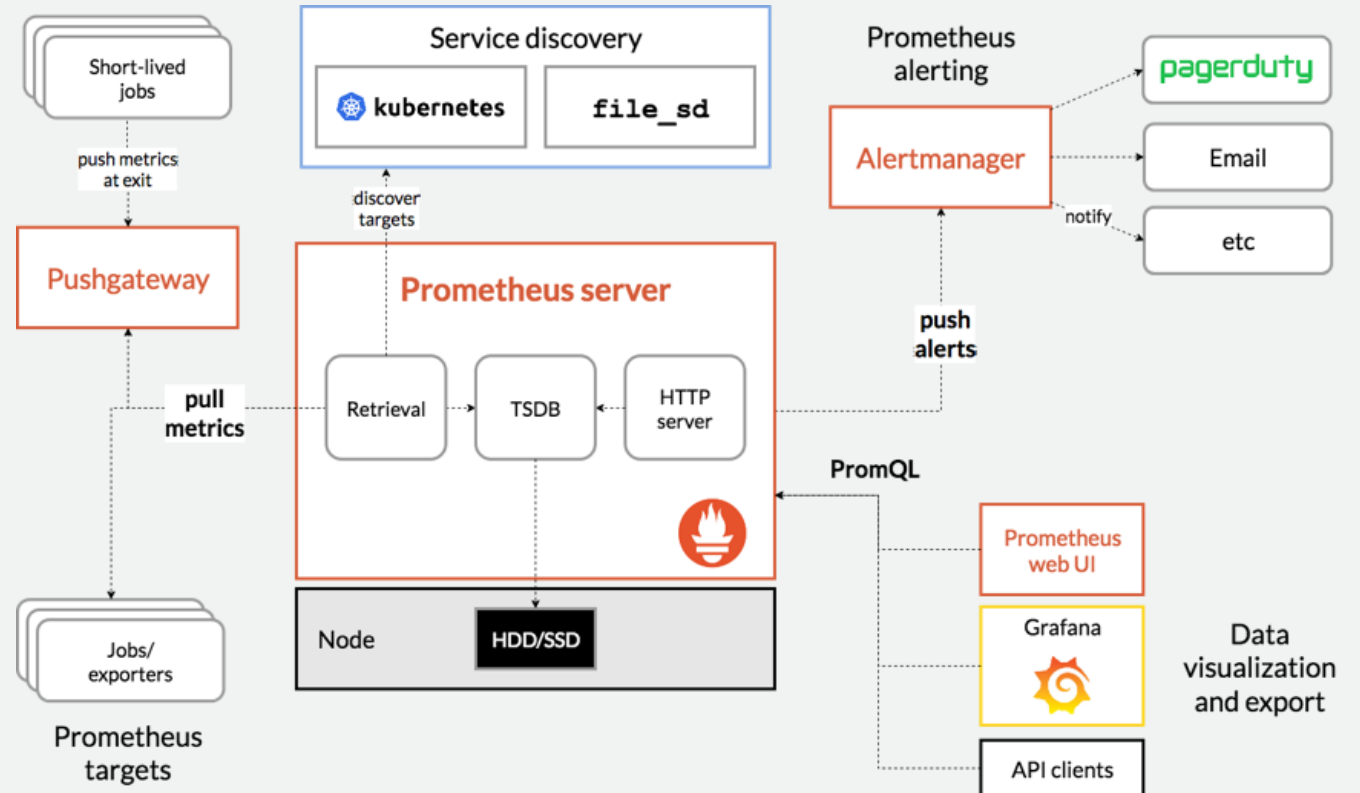


# cAdvisor

- Solution open-source fournie par Google pour monitorer les conteneurs
  - <https://github.com/google/cadvisor>
- cAdvisor est intégré aux kubelets
- Il permet aux administrateurs de suivre l'utilisation des ressources et leur performance
- Il s'agit d'un démon en cours d'exécution qui collecte, agrège, traite et exporte des informations sur les conteneurs en cours d'exécution
- Pour chaque conteneur, il exporte les paramètres d'isolation des ressources, l'utilisation historique des ressources, les histogrammes de l'utilisation historique complète des ressources et les statistiques du réseau
- cAdvisor supporte nativement les conteneurs Docker et devrait supporter à terme tout autre type de conteneur.

# Prometheus

- Prometheus est un outil de monitoring conçu à l'origine pour SoundCloud.
- Depuis sa création en 2012, de nombreuses entreprises et organisations ont adopté Prometheus
  - Le projet dispose d'une communauté de développeurs et d'utilisateurs très active.
- Il s'agit désormais d'un projet open source autonome et maintenu par une large communauté
- <https://prometheus.io/>



# Grafana

- Grafana est une solution open source permettant d'effectuer des analyses de données, d'extraire des métriques qui donnent un sens à l'énorme quantité de données et de surveiller les applications à l'aide de tableaux de bord personnalisables.
- Il permet d'étudier, analyser et surveiller les données sur une période de temps (séries temporelles)
- Il peut se connecter à toutes les sources de données possibles, communément appelées bases de données, telles que Graphite, Prometheus, Influx DB, ElasticSearch, MySQL, PostgreSQL, etc.
- Il possède un mécanisme de plugins qui le rend extensible
- <https://grafana.com/>

# P/G : Installation

- Installation

- *helm repo add prometheus-community <https://prometheus-community.github.io/helm-charts>*
- *helm repo update*
- *helm upgrade --install prometheus prometheus-community/kube-prometheus-stack \*
- *--namespace observability --create-namespace*
- *kubectl get secret --namespace observability prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo*
- *kubectl port-forward svc/prometheus-grafana -n observability 3000:80*

- Suppression

- *helm uninstall prometheus*

# Opentelemetry

- Opentelemetry est un outil capable de corréler les traces afin de détecter des chaînes causales pouvant mener à des pannes / erreurs
- C'est une solution "cloud native" qui s'intègre bien avec Kubernetes
- Installation:
  - `helm repo add open-telemetry https://open-telemetry.github.io/opentelemetry-helm-charts`
  - `helm repo update`
  - `helm install opentelemetry-operator open-telemetry/opentelemetry-operator --namespace observability`
  - `kubectl apply -f scripts/telemetry.yaml`

# Sécurité

# Introduction

- Pour sécuriser un cluster Kubernetes, il faut adopter une approche multidimensionnelle combinant contrôle des accès, protection des workloads, gestion des vulnérabilités et surveillance continue.
- Kubernetes, par sa nature dynamique et distribuée, expose plusieurs vecteurs de risque nécessitant des mesures spécifiques.
- La sécurisation d'un cluster Kubernetes se fait à plusieurs niveaux
  - Sécurité du réseau
  - Contrôle d'accès et privilèges
  - Sécurité des conteneurs
  - Surveillance et conformité



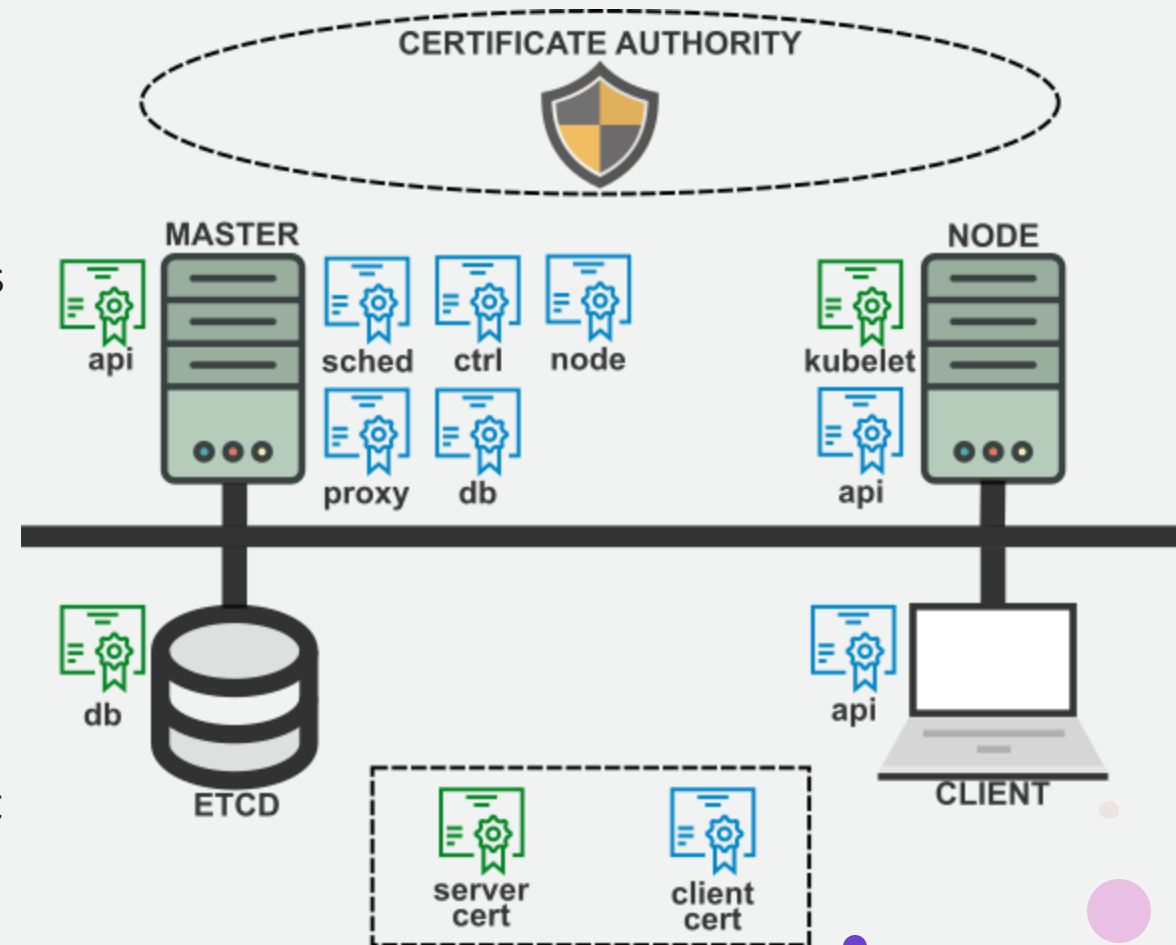
# Certificats – Vue d'ensemble

- **1. Certificats serveurs et clients**

- Certificats serveurs (server cert) : Utilisés par les services comme l'API Server, etcd, kubelet, etc. pour authentifier leur identité et chiffrer les connexions entrantes.
- Certificats clients (client cert) : Utilisés par les composants qui initient des connexions (scheduler, controller, proxy, kubelet, clients externes) pour prouver leur identité auprès des serveurs.

- **2. Chiffrement TLS généralisé**

- Toutes les communications critiques (API Server ↔ etcd, API Server ↔ kubelet, API Server ↔ clients, etc.) sont chiffrées via TLS, empêchant l'interception ou la falsification des données en transit.
- Authentification mutuelle (mTLS) : Les deux parties (client et serveur) présentent leur certificat signé par la CA, ce qui garantit que seuls les composants autorisés peuvent communiquer entre eux.



# Certificats – Vue d'ensemble

- **3. Gestion centralisée des identités**

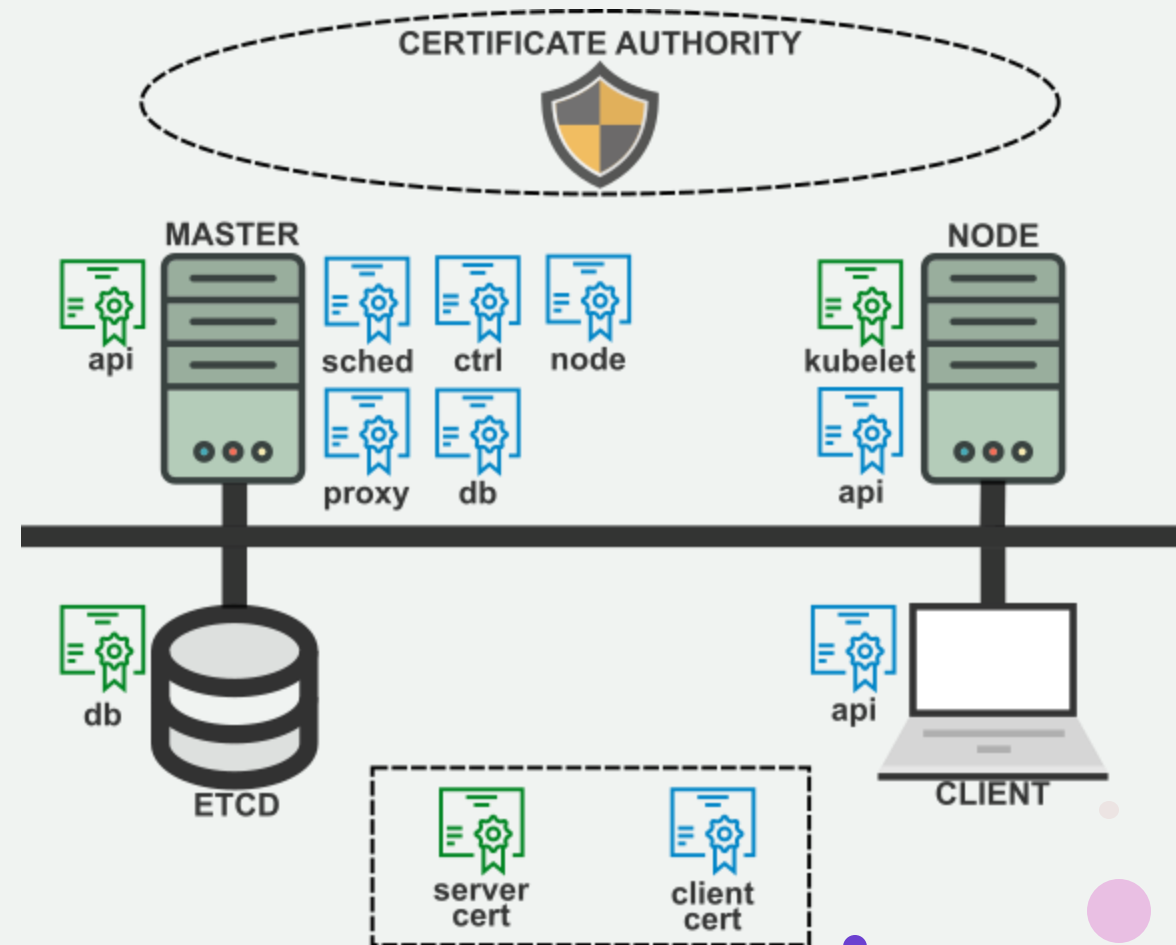
- CA unique : Tous les certificats sont émis par une autorité centrale, ce qui simplifie la gestion de la confiance et la révocation si besoin.
- Isolation des rôles : Chaque composant (scheduler, controller, proxy, kubelet, etc.) possède son propre certificat, limitant ainsi les privilèges à son périmètre fonctionnel.

- **4. Sécurisation des accès externes**

- Clients externes (administrateurs, outils d'automatisation) doivent également utiliser un certificat client signé par la CA pour accéder à l'API Server, ce qui protège l'accès au cluster contre les connexions non autorisées.

- **5. Sécurisation d'etcd**

- etcd (la base de données clé/valeur du cluster) utilise également des certificats pour chiffrer les échanges et authentifier les composants qui y accèdent, car il contient des données sensibles (secrets, configurations, etc.).



# Certificats - Détails

- **Sécurité des communications dans le cluster :**

- Communication avec le serveur API : Kubernetes utilise TLS (Transport Layer Security) pour sécuriser la communication entre le serveur API et d'autres composants tels que les kubelets, les contrôleurs et les nœuds. Le serveur API présente un certificat TLS à ces composants, qui doivent faire confiance à l'autorité de certification (CA) qui a signé le certificat du serveur API.
- Certificats TLS des kubelets : Chaque kubelet dispose d'un certificat TLS pour une communication sécurisée avec le serveur API. Ces certificats sont utilisés pour l'authentification et le chiffrement du trafic.
- Sécurité etcd : etcd, la base de données backend de Kubernetes, utilise également TLS pour sécuriser la communication entre les membres d'etcd et les clients (comme le serveur API).

- **Communication de service à service :**

- Kubernetes ne gère pas nativement le chiffrement de service à service. Cependant, il est possible d'y parvenir en utilisant des maillages de services comme Istio ou Linkerd, qui gèrent des certificats TLS pour chiffrer le trafic entre les services.

# Certificats - Détails

- **Authentification et autorisation :**

- Authentification de l'utilisateur : Pour l'authentification des utilisateurs, Kubernetes prend en charge les certificats clients comme l'un des mécanismes d'authentification. Les utilisateurs ou les systèmes externes peuvent présenter un certificat que le serveur API valide par rapport à son autorité de certification.
- Comptes de service : Pour l'authentification des services au sein du cluster, Kubernetes utilise des jetons JWT émis par son serveur API. Ces jetons sont montés dans des pods à l'aide de ressources ServiceAccount et sont utilisés pour s'authentifier auprès du serveur API.

- **Rotation et expiration :**

- Kubernetes prend en charge la rotation automatique des certificats pour certains composants, comme les certificats de kubelet. Cela permet de maintenir la sécurité en renouvelant régulièrement les certificats avant qu'ils n'expirent.

# Certificats - Détails

- **Gestion des certificats :**

- Gestion manuelle : Les administrateurs peuvent créer et distribuer des certificats manuellement, ce qui est possible dans les environnements plus petits ou plus statiques.
- Gestion automatisée : Kubernetes dispose d'un contrôleur intégré, `certificates.k8s.io`, qui peut être utilisé pour émettre des certificats. Il permet de créer des demandes de signature de certificat (CSR) que le contrôleur peut approuver automatiquement ou qu'un administrateur peut approuver manuellement.
- Outils externes : Des outils comme `cert-manager` étendent les capacités natives de Kubernetes et fournissent une gestion automatisée des certificats, y compris l'intégration avec des autorités de certification externes comme Let's Encrypt.

- **Ingress TLS :**

- Pour sécuriser le trafic externe vers le cluster, les ressources Kubernetes Ingress peuvent être configurées avec des certificats TLS, permettant le HTTPS pour les services exposés via Ingress.

# Certificats - Illustration

- Exemples de génération:
  - Cf. create-server-cert.sh
  - Cf. create-client-cert.sh
  - Utilisation (après installation) : `curl https://api-server:6443/api/v1/pods --key admin.key --cert admin.crt --cacert ca.crt`
- Dans une installation Kubernetes standard, les certificats sont gérés par défaut par le système
  - Kubernetes génère automatiquement les certificats nécessaires pour sécuriser la communication entre ses composants principaux, comme l'API Server, etcd, le kubelet, le scheduler, le controller-manager, etc.
  - Cf. Configuration des pods statiques
- Certificats par l'exemple
  - `openssl x509 -in apiserver.crt -text -noout` #Afficher le contenu d'un certificat
  - `cat ~/.kube/config` #Afficher les certificats pour communiquer avec l'API Server
  - `kubectl proxy` #Lancer un proxy local pour requêter l'API Server en utilisant la configuration locale
    - Exemple: `curl http://localhost:8001 -k`

# Certificats - Automatisation

- **Génération automatique** : Lors de l'initialisation du cluster (par exemple avec `kubeadm init`), Kubernetes crée une autorité de certification (CA) et émet tous les certificats clients et serveurs requis pour sécuriser les communications internes.
- **Emplacement des certificats** : Les certificats sont stockés, par défaut, dans le répertoire `/etc/kubernetes/pki` sur le nœud maître.
- **Gestion du cycle de vie** : Les certificats générés ont généralement une durée de validité d'un an. Kubernetes propose des mécanismes pour renouveler automatiquement certains certificats, comme ceux du kubelet, via des requêtes de signature de certificats (CSR) approuvées automatiquement par le controller-manager.
- **Renouvellement manuel ou automatique** : Vous pouvez renouveler les certificats manuellement avec des commandes comme `kubeadm certs renew`, ou laisser Kubernetes gérer automatiquement la rotation des certificats pour certains composants.

# Certificats - API

- Kubernetes propose des mécanismes natifs pour émettre, renouveler et distribuer des certificats X.509, essentiels à la sécurité interne et externe du cluster. Cette gestion s'appuie principalement sur deux API : `certificates.k8s.io` et, plus récemment, les `ClusterTrustBundles` (en alpha).
- L'API `certificates.k8s.io` permet de demander, d'approuver et d'obtenir des certificats TLS/X.509 signés par une autorité de certification (CA) reconnue par le cluster. Elle fonctionne via la ressource `CertificateSigningRequest` (CSR).



# API Certificats - Fonctionnement

- **Création d'une demande de signature de certificat (CSR)**
  - Un utilisateur ou une application génère une paire de clés privée/publique.
  - Il crée un fichier de demande de signature de certificat (CSR) contenant l'identité souhaitée.
  - Il soumet un objet `CertificateSigningRequest` à l'API Kubernetes, avec le CSR encodé en base64 dans le champ `spec.request`.
  - L'objet CSR doit préciser un `signerName` (par exemple : `kubernetes.io/kube-apiserver-client` pour un client de l'API Server).
- **Approbation du CSR**
  - Un administrateur (ou un contrôleur automatisé) vérifie la légitimité de la demande et l'approuve via la commande `kubectl certificate approve`.
  - Le CSR peut aussi être refusé (`kubectl certificate deny`).
- **Signature et distribution du certificat**
  - Une fois approuvé, un contrôleur (généralement intégré au kube-controller-manager) signe le certificat à l'aide de la CA du cluster.
  - Le certificat signé est stocké dans le champ `status.certificate` de l'objet CSR.
  - Le demandeur récupère le certificat et peut l'utiliser pour s'authentifier auprès du cluster.

# Sécurité réseau : Bonnes pratiques

- **Network Policies** : Bloquez par défaut les communications entre pods, puis autorisez explicitement les flux nécessaires via des règles basées sur les labels ou namespaces.
- **Micro-segmentation** : Isolez les environnements (prod, staging) et appliquez des zones de confiance distinctes avec des politiques strictes.
- **Chiffrement des flux** : Protégez les données sensibles avec TLS/SSL, notamment pour etcd et l'API server.

# Contrôles d'accès

- La gestion du contrôle d'accès dans Kubernetes est cruciale pour garantir que les utilisateurs et les applications disposent des autorisations correctes pour interagir avec les ressources du cluster Kubernetes.
- Kubernetes fournit plusieurs mécanismes pour gérer le contrôle d'accès:
  - **RBAC**: Le RBAC est la principale méthode de régulation de l'accès aux ressources au sein d'un cluster Kubernetes. Il permet aux administrateurs de définir des rôles et d'attribuer des autorisations à ces rôles.
  - **Compte de service**: Les comptes de service sont utilisés pour fournir une identité aux processus qui s'exécutent dans un pod. Ils sont utilisés par les applications de votre cluster pour interagir avec l'API Kubernetes.
  - **Politiques réseau**: Les politiques de réseau sont utilisées pour contrôler l'accès au réseau depuis et vers les pods.
  - **Contrôleurs d'admission**: Les contrôleurs d'admission sont des plugins qui interceptent les demandes adressées au serveur API de Kubernetes avant la persistance de l'objet, mais après l'authentification et l'autorisation de la demande.

# Méthodes d'authentification

## Certificats clients (X.509)

- **Description** : Utilisation de certificats TLS signés par une autorité de certification reconnue par le cluster. Le champ "subject" du certificat définit l'identité de l'utilisateur.
- **Cas d'usage** : Privilégiée pour les administrateurs ou utilisateurs humains, ainsi que pour les scripts automatisés hors cluster.
- **Avantages** : Sécurisée, bien supportée, idéale pour les petits/équipes techniques.
- **Inconvénients** : Gestion manuelle des certificats, rotation laborieuse, moins adaptée aux grandes équipes.
- **Exemple** : Le fichier `.kube/config` contient le certificat client pour l'authentification.

# Méthodes d'authentification

## Bearer Token

- **Description** : Utilisation de tokens (généralement des JWT) transmis dans l'en-tête HTTP Authorization: Bearer <token>.
- **Cas d'usage** : Utilisateurs ou applications externes.
- **Avantages** : Simple à mettre en œuvre, compatible avec de nombreux outils.
- **Inconvénients** : Nécessite une gestion sécurisée des tokens (rotation, expiration).
- **Exemple** : Les tokens statiques ou issus de services externes (OAuth2, OpenID Connect, etc.).

# Méthodes d'authentification

## Service Account

- **Description** : Les Pods Kubernetes sont associés à un compte de service, dont le token est monté dans le Pod (`/var/run/secrets/kubernetes.io/serviceaccount/token`). Le token est utilisé pour authentifier les requêtes vers l'API Server.
- **Cas d'usage** : Applications et workloads à l'intérieur du cluster.
- **Avantages** : Automatique, intégré à Kubernetes, idéal pour l'accès interne.
- **Inconvénients** : Les tokens ne sont pas expirés par défaut, nécessitent une gestion attentive.
- **Exemple** : Un Pod utilise le token de son compte de service pour communiquer avec l'API Server.

# Méthodes d'authentification

## Authentication Proxy

- **Description** : L'API Server délègue l'authentification à un proxy qui ajoute des en-têtes HTTP spécifiques (par exemple, X-Remote-User). Le proxy doit s'authentifier auprès de l'API Server via un certificat client.
- **Cas d'usage** : Intégration avec des systèmes d'authentification externes (LDAP, SAML, etc.).
- **Avantages** : Permet d'utiliser des systèmes d'authentification existants.
- **Inconvénients** : Configuration complexe, nécessite un proxy dédié, généralement non supporté dans les clusters managés.
- **Exemple** : Un proxy d'entreprise injecte l'identité de l'utilisateur dans les requêtes vers l'API Server.

# Méthodes d'authentification

## Webhook

- **Description** : L'API Server vérifie les tokens auprès d'un service externe (webhook), qui valide le token et retourne les informations d'identité.
- **Cas d'usage** : Intégration avec des systèmes d'authentification externes (OIDC, Keycloak, etc.).
- **Avantages** : Extensible, permet de centraliser la gestion des identités.
- **Inconvénients** : Nécessite un service externe fonctionnel, complexité de mise en œuvre.
- **Exemple** : Configuration avec `--authentication-token-webhook-config-file` pour pointer vers le service de validation



# Méthodes d'authentification

## OIDC

- **Description** : Utilisation d'un fournisseur d'identité externe (Google, Azure AD, Keycloak, etc.) pour authentifier les utilisateurs via des tokens JWT.
- **Cas d'usage** : Utilisateurs humains, intégration avec des systèmes d'identité d'entreprise.
- **Avantages** : Centralisation de l'authentification, support multi-cluster, gestion des accès simplifiée.
- **Inconvénients** : Nécessite un fournisseur OIDC, configuration spécifique.
- **Exemple** : L'API Server est configuré pour accepter les tokens émis par un fournisseur OIDC.

# Méthodes d'authentification

## Récapitulatif

| Méthode                                   | Utilisation principale     | Avantages                            | Inconvénients                       |
|-------------------------------------------|----------------------------|--------------------------------------|-------------------------------------|
| <b>Certificats clients (X.509)</b>        | Admins, scripts            | Sécurisé, simple pour petits groupes | Gestion manuelle, peu scalable      |
| <b>Tokens (Bearer)</b>                    | Utilisateurs, applications | Simple, compatible                   | Gestion des tokens à surveiller     |
| <b>Compte de service (ServiceAccount)</b> | Applications/Pods          | Automatique, intégré                 | Tokens non expirés, gestion requise |
| <b>Proxy d'authentification</b>           | Intégration externe        | Utilise l'existant                   | Complexe, proxy dédié               |
| <b>Webhook</b>                            | Intégration externe        | Extensible, centralisé               | Service externe, complexité         |
| <b>OpenID Connect (OIDC)</b>              | Utilisateurs humains       | Centralisé, multi-cluster            | Fournisseur OIDC, configuration     |

# Méthodes d'authentification

## Bonnes pratiques

- **Utiliser au moins deux méthodes** : Par exemple, comptes de service pour les workloads et une méthode externe pour les utilisateurs.
- **Principe du moindre privilège** : Limiter les accès au strict nécessaire.
- **Rotation régulière** : Des certificats et tokens pour limiter les risques en cas de compromission.
- **Intégrer RBAC** : Après l'authentification, utiliser le RBAC pour gérer les autorisations.

# RBAC - Introduction

- Le RBAC (contrôle d'accès basé sur les rôles) est un système de gestion des permissions qui permet de définir finement qui peut effectuer quelles actions sur quelles ressources au sein d'un cluster Kubernetes.
- Il s'agit d'un mécanisme essentiel pour assurer la sécurité et la gouvernance de l'infrastructure, particulièrement dans les environnements multi-utilisateurs ou multi-applications
- Les principaux concepts de RBAC
  - Role : Définit un ensemble de permissions dans un namespace spécifique.
  - ClusterRole : Définit un ensemble de permissions valables au niveau du cluster entier.
  - RoleBinding : Associe un Role à un ou plusieurs sujets (utilisateurs, groupes, comptes de service) dans un namespace.
  - ClusterRoleBinding : Associe un ClusterRole à un ou plusieurs sujets au niveau du cluster entier.

# Organisation des groupes API

- Kubernetes structure ses API selon deux grands types de groupes : le groupe API core et les groupes API nommés.
  - Cette organisation facilite la gestion, l'extensibilité et la découverte des ressources.
- **1. Groupe API core** (Core API Group)
  - Endpoint : /api/v1
  - Ressources : Pods, Services, Namespaces, ConfigMaps, Secrets, Nodes, etc.
  - Caractéristiques : Ce sont les ressources fondamentales nécessaires au fonctionnement du cluster.
- **2. Groupes API nommés** (Named API Groups)
  - Endpoint : /apis/<group>/<version>
  - Ressources : Déploiements, ReplicaSets, StatefulSets, NetworkPolicies, CronJobs, StorageClasses, etc.
  - Caractéristiques : Ces groupes organisent les ressources par thématique (applications, networking, stockage, etc.) et permettent d'étendre Kubernetes.

# Groupes et verbes

- Les droits d'accès dans Kubernetes (notamment via RBAC) sont définis en fonction de trois éléments principaux : le groupe API, le verbe (action) et la ressource concernée.
- Kubernetes distingue deux types de rôle:
  - **Roles** : Définissent des permissions dans un namespace spécifique.
  - **ClusterRoles** : Définissent des permissions à l'échelle du cluster.
- Dans un rôle, on précise :
  - Le **groupe API** : Par exemple, apps, networking.k8s.io, ou laisser vide pour le core.
  - La **ressource** : Par exemple, pods, deployments, secrets.
  - Le **verbe** : Par exemple, get, list, create, update, delete.
  - Cf. rbac-example.yaml

# RBAC – Sujets et permissions

- Les sujets d'un RBAC peuvent être :
  - Des utilisateurs (authentifiés via certificats, tokens, etc.)
  - Des groupes (groupes d'utilisateurs)
  - Des comptes de service (ServiceAccount), utilisés principalement pour les Pods et applications internes.
- Les permissions RBAC précisent :
  - Les actions (verbs) : get, list, create, update, delete, etc.
  - Les ressources : pods, services, deployments, secrets, etc.
  - Les namespaces : La portée des permissions, soit à l'échelle d'un namespace, soit du cluster entier.
- Pour activer le RBAC, il faut démarrer l'API server avec l'option `--authorization-mode=Node,RBAC`

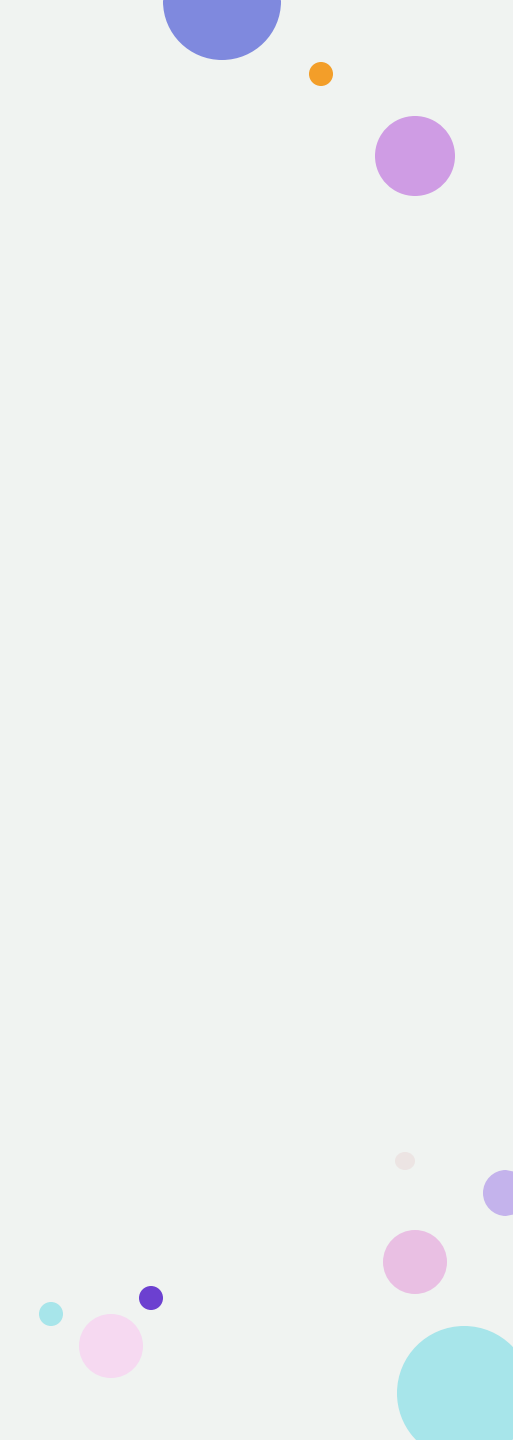
# RBAC - Rôles par défaut

- **view** : Lecture seule (sauf les secrets)
- **edit** : Lecture et modification (sauf rôles et role bindings)
- **admin** : Administration d'un namespace (gestion des rôles et role bindings)
- **cluster-admin** : Accès total au cluster





# RBAC – Bonnes pratiques

- Principe du moindre privilège : Ne donner que les permissions nécessaires.
  - Utiliser les rôles par défaut quand possible pour simplifier la gestion.
  - Limiter l'utilisation de ClusterRoleBinding aux cas strictement nécessaires.
  - Auditer régulièrement les permissions pour éviter les accès indésirables.
- 

# Authentication - Exemple

- Dans l'exemple ci-dessous, chaque ligne contient:
  - Le token d'authentification.
  - Le nom de l'utilisateur.
  - L'identifiant unique de l'utilisateur.
  - Les groupes auxquels appartient l'utilisateur, séparés par des virgules et encadrés par des guillemets s'il y en a plusieurs. (Optionnel)
- Il faut ensuite lancer le serveur avec l'option suivante:
  - `--token-auth-file=/chemin/vers/static-tokens.csv`
- Pour vérifier si on a des droits particuliers:
  - `kubectl auth can-I create deployments`
- **Questions:** Quelles sont les limites de cette approche ?

```
token1,alice,1000,"dev,ops"  
token2,bob,1001,"dev"  
token3,charlie,1002
```

# Service Account & Token Controller

- Le contrôleur Service Account & Token englobe deux composants principaux dans Kubernetes :
  - **ServiceAccount Controller** : Il s'assure que chaque namespace possède un objet ServiceAccount nommé default et gère la création/suppression des objets ServiceAccount dans le cluster.
  - **Token Controller** : Il gère la création, la distribution, la rotation et la suppression des tokens d'authentification associés aux ServiceAccounts. Ces tokens sont utilisés par les pods et les applications pour s'authentifier auprès de l'API Kubernetes
- Points clés
  - **ServiceAccount** : Identité pour les pods et **processus non humains** dans Kubernetes, associée à des permissions via RBAC.
  - **Token** : Jeton JWT signé, utilisé pour authentifier le pod auprès de l'API Kubernetes. Automatiquement injecté et géré par le contrôleur.
  - **Sécurité** : Les permissions du ServiceAccount sont définies par des rôles RBAC. Les tokens courts-vivants limitent le risque en cas de fuite.
  - **Rotation et révocation** : Les tokens sont automatiquement renouvelés et invalidés à la suppression du pod ou du ServiceAccount. Pour révoquer un token, il suffit de supprimer le pod ou le ServiceAccount associé.
- Pour lister les "service accounts"
  - *kubectl get serviceaccounts -A*

# Service A&T Controller : Comportement

- **Création automatique** : Lorsqu'un namespace est créé, le contrôleur crée automatiquement un ServiceAccount default dans ce namespace.
- **Gestion des tokens** :
  - À chaque création de ServiceAccount, le contrôleur Token génère un token d'authentification (JWT) et le rend disponible via un Secret ou, dans les versions récentes de Kubernetes, via un volume projeté dans les pods.
  - Les tokens sont signés par une clé privée spécifiée dans le kube-controller-manager et vérifiés par l'API server avec la clé publique correspondante.
- **Rotation et expiration** :
  - Depuis Kubernetes v1.22, les tokens ont une durée limitée (par défaut : 1 heure) et automatiquement renouvelés par le kubelet via le mécanisme TokenRequest API. Ils sont projetés dans les pods via un volume projeté et sont automatiquement invalidés à la suppression du pod ou à l'expiration du token.
- **Suppression** :
  - Si un ServiceAccount est supprimé, le contrôleur Token supprime aussi tous les Secrets associés contenant les tokens.
  - Si un Secret contenant un token est supprimé, le contrôleur met à jour le ServiceAccount pour retirer la référence à ce Secret1.
- **Assignment à un pod** :
  - Lorsqu'un pod est créé, Kubernetes injecte automatiquement le token du ServiceAccount assigné dans le pod (sauf si automountServiceAccountToken: false est spécifié).
  - Le pod utilise ce token pour s'authentifier auprès de l'API Kubernetes ou d'autres services qui valident l'identité du ServiceAccount.

# Admission Controller

- Le contrôleur Admission (ou admission controller) est un composant fondamental du plan de contrôle Kubernetes.
  - Il s'agit d'un morceau de code qui intercepte toutes les requêtes adressées à l'API server après l'authentification et l'autorisation, mais avant que la ressource ne soit persistée dans etcd.
- Son rôle est de valider ou modifier ces requêtes selon des politiques, des règles de sécurité, ou des besoins d'organisation du cluster.
  - Si la requête ne respecte pas les critères définis, elle peut être rejetée.
- Exemples d'usage
  - **Sécurité** : Refuser les pods sans certaines annotations, empêcher la création de ressources dans des namespaces protégés, imposer des images signées, etc.
  - **Gouvernance** : Appliquer des quotas de ressources, forcer la présence de labels, injecter automatiquement des sidecars (ex : Istio), etc.
  - **Personnalisation** : Ajouter des règles métiers spécifiques via des webhooks dynamiques.

# Admission Controller - Comportement

- **Phase de mutation** (mutating)
  - Les contrôleurs de type mutating peuvent modifier la ressource avant son enregistrement (ajout de labels, injection de sidecar, définition de valeurs par défaut, etc.).
  - Exemples : MutatingAdmissionWebhook, AlwaysPullImages, LimitRanger (pour l'ajout de limites par défaut).
- **Phase de validation** (validating)
  - Les contrôleurs de type validating vérifient que la ressource respecte bien les politiques du cluster (schéma, quotas, sécurité, etc.) et peuvent rejeter la requête si besoin.
  - Exemples : ValidatingAdmissionWebhook, ResourceQuota, PodSecurityPolicy, NamespaceLifecycle.

# Admission Controller - Types

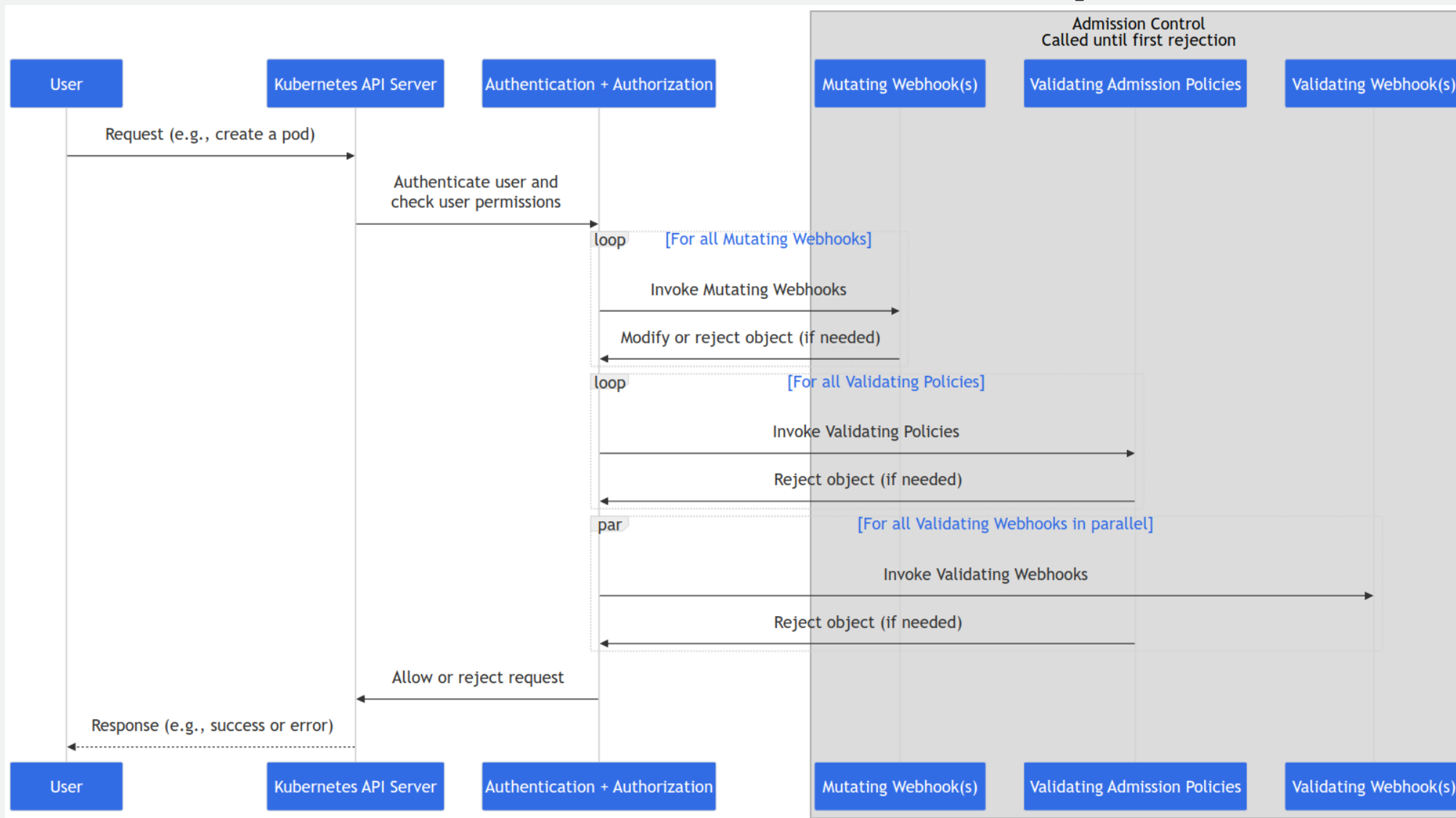
- **Contrôleurs intégrés (built-in)**

- Inclus dans le binaire kube-apiserver, activables ou désactivables par l'administrateur.
- Exemples : NamespaceLifecycle, ResourceQuota, ServiceAccount, DefaultStorageClass.
- Pour lister les contrôleurs disponibles (sur le master):
  - `kubectl exec kube-apiserver-master -n kube-system -- kube-apiserver -h | grep enable-admission-plugins`
- Pour modifier les contrôleurs actifs, modifier la configuration du pod statique de l'API Server
- <https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/>

- **Contrôleurs dynamiques (webhooks)**

- Permettent d'étendre le comportement d'admission via des **webhooks HTTP externes**, configurés dynamiquement grâce aux ressources suivantes:
  - **MutatingAdmissionWebhook** : Modifie les requêtes.
  - **ValidatingAdmissionWebhook** : Valide ou rejette les requêtes.
- Lister les webhooks modifiant les requêtes:
  - `kubectl get mutatingwebhookconfigurations`

# Admission Controller - Comportement





# Surveillance et conformité

- **Profils de sécurité** : Appliquez des profils seccomp/AppArmor pour restreindre les appels système.
- **Pods non privilégiés** : Désactivez `privileged: true` et `allowPrivilegeEscalation: false`.
- **Mises à jour régulières** : Maintenez les composants Kubernetes (kubelet, etcd) à jour.
- **Journalisation centralisée** : Collectez les logs d'audit de l'API et des conteneurs avec des outils comme Falco ou Sysdig.
- **Benchmarks CIS** : Utilisez kube-bench pour vérifier la conformité aux bonnes pratiques.
- **Politiques d'admission** : Déployez des contrôleurs comme OPA/Gatekeeper pour valider les configurations avant déploiement.

# Sécurité des conteneurs

- **Images minimales** : Privilégiez des images "distroless" ou Alpine pour réduire la surface d'attaque.
- **Analyse des vulnérabilités** : Intégrez des outils comme Trivy ou Snyk dans la CI/CD pour scanner les images avant déploiement.
- **Contextes de sécurité** :
  - *runAsNonRoot: true* pour empêcher l'exécution en root
  - *readOnlyRootFilesystem: true* sauf nécessité
  - Suppression des capacités Linux inutiles (*capabilities.drop*)

# Sécurité: Bonnes pratiques

- **Contrôle d'accès basé sur les rôles (RBAC)** : Définissez et appliquez des politiques d'accès strictes.
- **Network policies** : Limitez la communication entre les pods pour réduire la surface d'attaque.
- **Chiffrement des données** : Utilisez le chiffrement pour protéger les données sensibles, en transit et au repos.

# Exemples

- Afin de se prémunir contre d'éventuelles attaques, il est nécessaire de mettre en place certaines stratégies
- Au niveau du pod, on peut restreindre les droits
  - Cf. Exemples hw-security.yaml et rbac-example.yaml
- Vous pouvez également utiliser un outil comme snyk (<https://snyk.io/>) pour vérifier votre configuration et obtenir des recommandations
  - \$snyk iac test deployment.yaml
- Gérer les contrôles d'accès
  - <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
- Consulter régulièrement les ressources suivantes
  - <https://www.opencve.io/cve?cvss=critical&search=kubernetes>
  - [https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR\\_KUBERNETES\\_HARDENING\\_GUIDANCE\\_1.2\\_20220829.PDF](https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF)

```
- name: helloworld
  image: karthequian/helloworld:latest
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
  securityContext:
    allowPrivilegeEscalation: false
    runAsNonRoot: true
    capabilities:
      drop:
        - ALL
      readOnlyRootFilesystem: true
  ports:
    - containerPort: 80
```

## Medium Severity Issues: 3

### [Medium] Container is running without root user control

Info: Container is running without root user control. Container could be running with full administrative privileges

Rule: <https://snyk.io/security-rules/SNYK-CC-K8S-10>

Path: [DocId: 0] > input > spec > template > spec > containers[pod-info-container] > securityContext > runAsNonRoot

File: deployment.yaml

Resolve: Set `securityContext.runAsNonRoot` to `true`

The image features a light gray background with several decorative circles of various colors (pink, purple, blue, orange, teal) scattered in the corners. The word "Debug" is centered in a dark purple font.

# Debug

# Problèmes courants

- **Problèmes de ressources**

- Surutilisation ou sous-utilisation du CPU/mémoire : provoque des pods en attente ou des ralentissements.
- CrashLoopBackOff : redémarrages en boucle des pods, souvent liés à des erreurs de configuration, de ressources insuffisantes ou de dépendances manquantes.
- ErrImagePull : échec d'extraction d'une image de conteneur, souvent à cause d'un nom d'image incorrect ou d'un problème d'accès au registre.

- **Problèmes de connectivité réseau**

- Pods incapables de communiquer entre eux ou avec l'extérieur (ex. : port VXLAN en conflit, politiques réseau trop restrictives).
- Services inaccessibles malgré des pods fonctionnels (erreurs de mapping de ports, problèmes d'Ingress, DNS interne mal configuré).

# Problèmes courants

- **Problèmes de plan de contrôle**

- Nœuds indisponibles ou défaillants.
- Services du plan de contrôle (API server, etcd, scheduler) en panne ou instables.
- Problèmes de certificats ou de configuration empêchant la communication avec l'API server.

- **Problèmes de déploiement**

- Pods bloqués à l'état « ContainerCreating » (souvent lié à l'image de pause ou à la configuration réseau).
- Déploiements échoués après une mise à jour (rollback nécessaire, erreurs dans les logs, sondes de disponibilité non satisfaites).

# Méthodes

- **Vérifier l'état global du cluster**

- *kubectl cluster-info* : informations sur le plan de contrôle.
- *kubectl get componentstatus* : santé des composants principaux (API server, etcd, scheduler).

- **Inspecter les nœuds**

- *kubectl get nodes -o wide*
- *kubectl describe node <node-name>* : détails sur la santé, les ressources, les taints, etc.

- **Analyser les pods**

- *kubectl get pods -A -o wide --field-selector status.phase!=Running* : liste des pods non sains.
- *kubectl describe pod <pod-name>* : événements récents, raisons d'échec, détails sur les conteneurs.
- *kubectl logs <pod-name>* : logs applicatifs (stdout/stderr).
- *kubectl exec -it <pod-name> -- /bin/sh* : shell interactif pour diagnostiquer de l'intérieur .



# Méthodes

- **Examiner les services et la connectivité**

- `kubectl get services`, `kubectl describe service <service-name>`
- `kubectl describe ingress <ingress-name>`
- Vérification des network policies : `kubectl get networkpolicies`, `kubectl describe networkpolicy <policy-name>`.

- **Surveillance et événements**

- `kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>` : chronologie des événements récents.
- `kubectl top pods`, `kubectl top nodes` : consommation de ressources.

# Techniques avancées

| Outil/Commande                   | Usage principal                                                      |
|----------------------------------|----------------------------------------------------------------------|
| <b>kubectl-debug</b>             | Lancer un conteneur éphémère pour diagnostiquer un pod/nœud          |
| <b>kubectl-neat</b>              | Rendre la sortie YAML/JSON plus lisible                              |
| <b>kubectl-tree</b>              | Visualiser la hiérarchie des ressources (ownership)                  |
| <b>kubectl-sniff</b>             | Capturer le trafic réseau d'un pod                                   |
| <b>kubectl-node-shell</b>        | Ouvrir un shell directement sur un nœud du cluster                   |
| <b>k9s, Kubernetes Dashboard</b> | UI pour navigation et visualisation interactive                      |
| <b>Outils d'observabilité</b>    | Prometheus, Grafana, Jaeger pour logs, métriques, traces distribuées |



# Bonnes pratiques

- Toujours commencer par l'état global du cluster avant d'aller dans le détail d'un pod ou service.
- Utiliser les événements Kubernetes pour comprendre la chronologie d'un incident.
- S'appuyer sur la centralisation des logs et la surveillance des métriques pour anticiper les problèmes.
- Documenter les incidents et les solutions pour améliorer la résilience du cluster.

# crictl - TP

- Installation: <https://github.com/kubernetes-sigs/cri-tools>
- Éditer le fichier /etc/crictl.yaml

```
runtime-endpoint: unix:///var/run/containerd/containerd.sock  
image-endpoint: unix:///var/run/containerd/containerd.sock  
timeout: 10  
debug: false
```

- Documentation: <https://kubernetes.io/docs/tasks/debug/debug-cluster/crictl>

The image features a light gray background with several decorative circles of various colors (pink, purple, blue, orange, teal) scattered in the corners. The word "Kustomize" is centered in a dark purple, sans-serif font.

# Kustomize

# Présentation Rapide

- Kustomize est un outil natif de gestion de configuration pour Kubernetes, intégré directement à kubectl depuis la version 1.14.
- Il permet de personnaliser et de gérer les fichiers YAML de ressources Kubernetes sans recourir à des templates ou à la duplication de fichiers, contrairement à des outils comme Helm qui utilisent un système de templates avec des valeurs dynamiques.
- L'objectif principal de Kustomize est d'appliquer le principe DRY (Don't Repeat Yourself) aux manifests Kubernetes.
  - Plutôt que de dupliquer des fichiers YAML pour chaque environnement (développement, staging, production), Kustomize propose de définir une base commune et d'y appliquer des modifications ciblées via des overlays ou des patches.
  - Cela simplifie la maintenance et l'évolution des configurations, tout en gardant les fichiers d'origine intacts et réutilisables.

# Avantages

- **Natif et simple** : Utilisable directement avec kubectl, sans installation supplémentaire depuis la v1.14.
- **Pas de templating** : Utilise une approche déclarative et sans logique de template, ce qui facilite la validation et la compréhension des manifests.
- **Gestion multi-environnements** : Permet de gérer facilement plusieurs variantes de configuration pour différents environnements, équipes ou clients.
- **Sécurité** : Facilite la gestion des secrets et des ConfigMaps sans les inclure directement dans les manifests versionnés.

# Fonctionnement

- **Fichier kustomization.yaml** : Au cœur de Kustomize, ce fichier décrit comment assembler et personnaliser les ressources Kubernetes. Il référence les ressources de base, les patches à appliquer, les générateurs de ConfigMaps et de Secrets, ainsi que les labels ou annotations à ajouter.
- **Bases et overlays** - La structure d'un projet Kustomize repose sur une hiérarchie :
  - Base : Ensemble de ressources communes à tous les environnements.
  - Overlay : Personnalisation spécifique à un environnement, qui s'appuie sur la base (ex : changer le nombre de réplicas, ajouter un label, modifier une image Docker).
- **Générateurs** : Permettent de créer des ConfigMaps et des Secrets à partir de fichiers ou de valeurs littérales, sans avoir à écrire les ressources à la main.
- **Transformateurs** : Appliquent des modifications globales sur les ressources (ajout de préfixes/suffixes, labels, annotations, etc.).
- **Patches** : Modifient de façon ciblée des champs spécifiques dans les ressources YAML existantes, sans dupliquer l'intégralité du fichier.



# Transformers communs

| Transformer              | Rôle principal                            | Exemple d'utilisation            |
|--------------------------|-------------------------------------------|----------------------------------|
| <b>commonLabels</b>      | Ajouter/modifier des labels               | app: my-app, env: prod           |
| <b>commonAnnotations</b> | Ajouter/modifier des annotations          | version: "1.0.0", branch: master |
| <b>namespace</b>         | Définir le namespace commun               | staging, production              |
| <b>namePrefix</b>        | Ajouter un préfixe aux noms de ressources | dev-, test-                      |
| <b>nameSuffix</b>        | Ajouter un suffixe aux noms de ressources | -v1, -prod                       |
| <b>images</b>            | Modifier nom/tag/digest d'images Docker   | nginx:1.25, my-app:2.0           |

**Cohérence** : Toutes les ressources reçoivent les mêmes labels, annotations, etc.

**Centralisation** : Les modifications se font à un seul endroit (kustomization.yaml).

**Gain de temps** : Plus besoin de modifier chaque fichier YAML individuellement.

**Évolutivité** : Idéal pour gérer plusieurs environnements ou équipes

# Patches

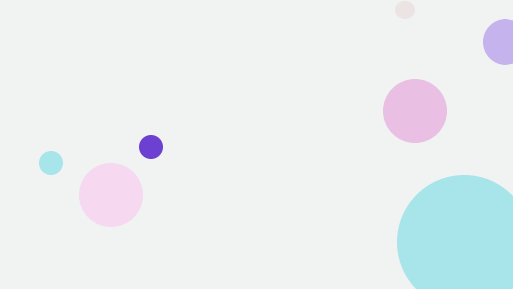
- Kustomize propose un système de patches permettant de modifier les fichiers de ressources Kubernetes sans toucher aux fichiers de base.
  - Cette approche favorise la réutilisation, la maintenabilité et la séparation des préoccupations entre la configuration commune et les personnalisations spécifiques à un environnement ou à un besoin particulier.
- **Opérations principales :**
  - **Ajout ou modification de champs :** Vous pouvez ajouter de nouveaux champs ou modifier la valeur de champs existants en ne spécifiant que les champs concernés dans le patch.
  - **Fusion intelligente :** Pour certains objets (comme les listes de conteneurs dans un Deployment), la fusion se fait en fonction de la clé (par exemple, le nom du conteneur), ce qui permet d'ajouter ou de modifier des éléments sans écraser l'ensemble de la liste.
  - **Suppression d'éléments :** Possible via la directive spéciale `$patch: delete` dans le patch YAML, mais ce n'est pas aussi flexible que le JSON Patch pour la suppression de champs ou d'éléments de tableau.
- **Limitations :**
  - Ne permet pas de manipuler précisément les tableaux (par exemple, supprimer un élément à un index donné).
  - Ne permet pas d'opérations conditionnelles ou de tests.

# Components

- Les components (composants) sont une fonctionnalité avancée de Kustomize permettant de factoriser et de réutiliser des morceaux de configuration dans différents overlays, sans dupliquer le code.
- Ils sont conçus pour modulariser les personnalisations, notamment lorsque certaines fonctionnalités (monitoring, debug, dashboards, etc.) doivent être activées ou désactivées selon l'environnement ou le besoin.
  - Un component est un répertoire avec un kustomization.yaml spécial, dont le type est kind: Component (et non Kustomization).
  - Il peut contenir des ressources, des patches, des générateurs, des transformers, etc., comme un overlay classique.
  - Contrairement à un overlay, un component n'est pas destiné à être appliqué seul, mais à être inclus dans d'autres overlays via le champ components.
  - Un component s'applique après l'accumulation des ressources de l'overlay parent, ce qui permet de transformer ou d'étendre les ressources déjà présentes, et de composer plusieurs components séquentiellement.



# Components - Avantages

- **Réutilisabilité** : Centraliser des fonctionnalités optionnelles (debug, monitoring, RBAC, dashboards, etc.) pour les activer facilement dans un ou plusieurs overlays sans duplication.
  - **Modularité** : Construire des overlays complexes à partir de briques élémentaires, activables à la demande.
  - **Simplicité** : Réduire la complexité et la maintenance des overlays, en évitant la répétition de patches ou de ressources partagées.
- 

# Différences avec Helm

| Critère                 | Kustomize               | Helm                              |
|-------------------------|-------------------------|-----------------------------------|
| Approche                | Patch des fichiers YAML | Templates avec valeurs dynamiques |
| Complexité              | Simple et natif         | Plus puissant mais plus complexe  |
| Gestion des dépendances | Non intégrée            | Support des dépendances de charts |
| Intégration kubectl     | Oui (intégré)           | Non (nécessite Helm)              |
| Apprentissage           | Facile pour débutants   | Peut être complexe                |

# Exemples

- Cf. Contenu du **dossier kustomize**
- Preview
  - *kubectl kustomize overlays/development*
  -
- Application
  - *kubectl apply -k overlays/development*
  -
- Suppression
  - *kubectl delete -k overlays/development*
- Validation
  - *kubectl kustomize overlays/production | kubectl apply --dry-run=client -f -*

The image features a light gray background with several decorative circles of various colors (pink, purple, blue, orange, teal) scattered in the corners. The word "Helm" is centered in a dark gray, sans-serif font.

# Helm

# Présentation Rapide

- Helm est à Kubernetes ce que apt est à Debian
- Il offre une capacité de versionnement pour une application basée sur des micro-services
- Il facilite le déploiement
  - Il évite notamment les nombreux `$kubectl apply -f ressource.yaml`
  - Il centralise les versions des micro-services

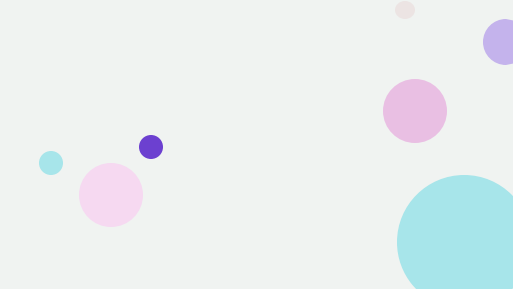
```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

- Artifact Hub est pour Kubernetes ce que Docker Hub est pour docker
    - <https://artifacthub.io/>
  - Helm tire toutes les dépendances requises lors de l'installation d'un chart
- Demo: installation de Wordpress par Helm
  - Ajout du repo
  - Installation





# Introduction

- Helm est le gestionnaire de paquets officiel pour Kubernetes, conçu pour simplifier et automatiser le déploiement, la gestion et la mise à jour d'applications dans des environnements Kubernetes.
  - Il joue un rôle similaire à celui d'outils comme apt ou yum dans le monde des distributions Linux, mais appliqué à l'écosystème Kubernetes, en permettant de gérer des applications complexes sous forme de paquets appelés charts.
- 

# Motivations

- Déployer une application sur Kubernetes nécessite généralement de manipuler de nombreux fichiers YAML, souvent redondants et difficiles à maintenir à grande échelle.
- Helm répond à cette problématique en proposant :
  - Un système de packaging (chart) qui regroupe tous les fichiers nécessaires au déploiement d'une application, ainsi que ses dépendances et sa configuration.
  - Un moteur de templating qui permet de paramétrer dynamiquement les ressources Kubernetes, évitant la duplication des fichiers et facilitant la gestion multi-environnements.
  - La gestion du cycle de vie des applications : installation, mise à jour, rollback, suppression, le tout de façon versionnée et traçable.

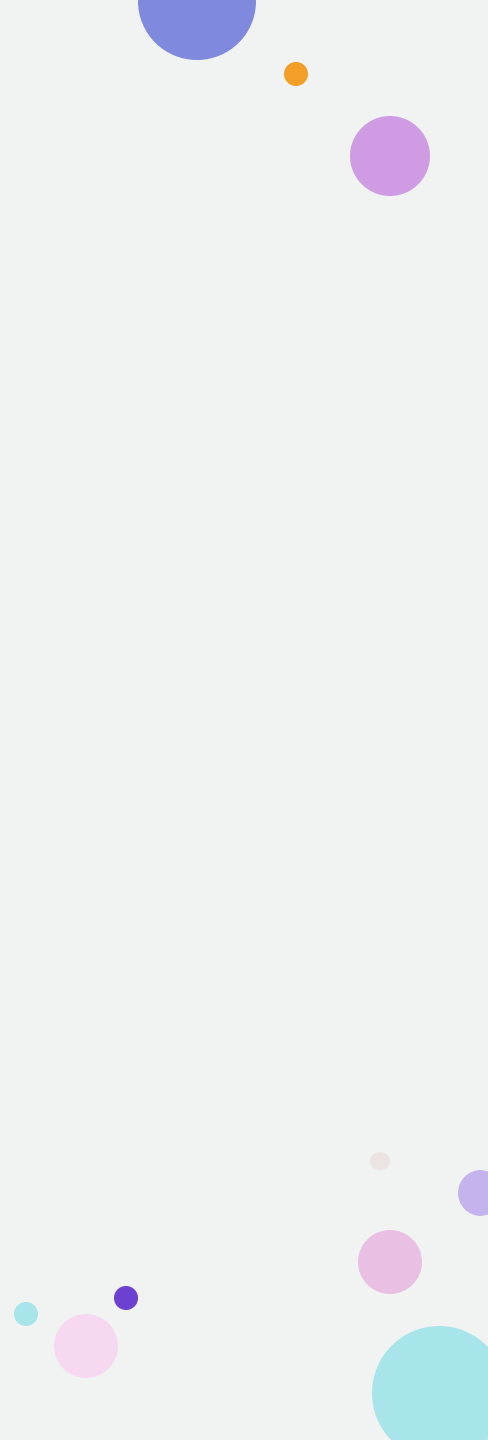
# Historique

- 2015 : Helm est lancé par l'équipe Deis pour simplifier le déploiement sur Kubernetes.
- Helm v2 : Introduction du composant serveur Tiller pour gérer les releases dans le cluster, mais avec des limitations de sécurité.
- 2019 – Helm v3 : Suppression de Tiller, Helm interagit directement avec l'API Kubernetes, rendant l'outil plus sécurisé et simple à utiliser.
- Helm fait désormais partie de la Cloud Native Computing Foundation (CNCF), garantissant son évolution et son adoption dans l'écosystème Kubernetes.



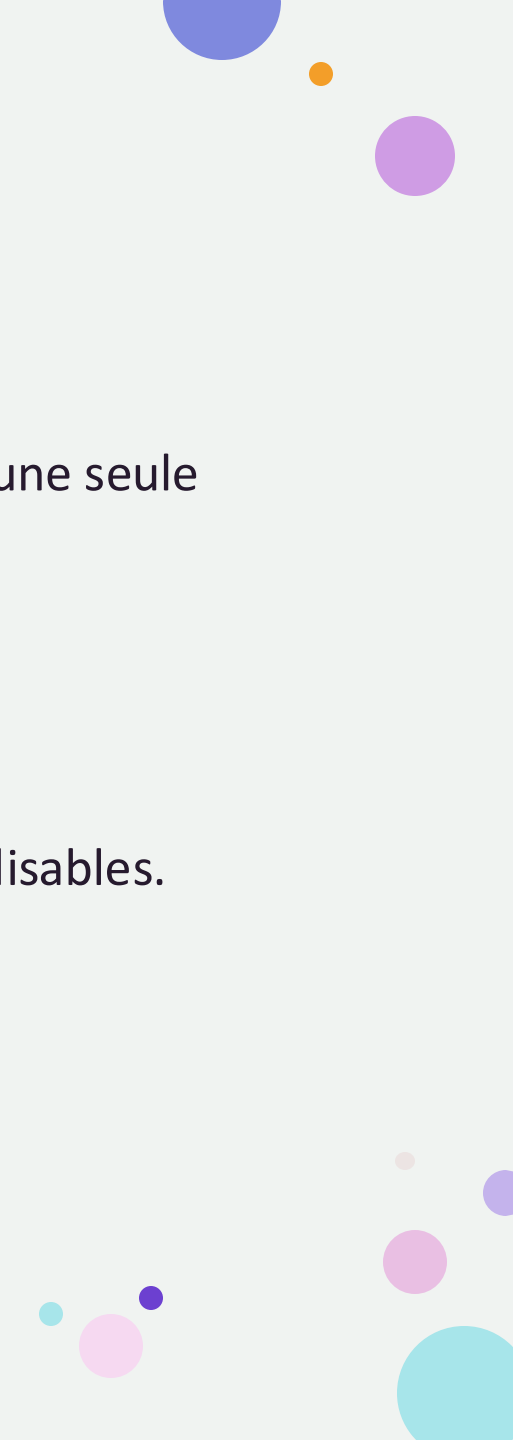
# Avantages

- Déploiement rapide et standardisé d'applications complexes.
- Gestion centralisée des configurations et des dépendances.
- Possibilité de versionner, mettre à jour et restaurer des applications facilement.
- Forte adoption communautaire et nombreux charts disponibles en open source.



A cluster of circles in shades of pink, purple, and orange in the top-left corner.

# Cas d'usage typiques

- Déploiement d'applications multi-tiers (bases de données, backends, frontends) en une seule commande.
  - Automatisation des déploiements CI/CD dans des environnements Kubernetes.
  - Standardisation et partage de bonnes pratiques de déploiement via des charts réutilisables.
- 
- A collection of circles in shades of blue, purple, pink, and teal in the bottom-right corner.

# Concepts Essentiels

- **Chart** : Paquet contenant toutes les ressources Kubernetes nécessaires à une application (déploiements, services, configurations, etc.).
- **Release** : Instance d'un chart déployée sur un cluster Kubernetes, associée à une configuration spécifique.
  - Les releases peuvent avoir différentes versions
- **Repository** : Emplacement (distant ou local) où sont stockés et partagés les charts Helm.
- **Values** : Fichier de valeurs permettant de personnaliser la configuration d'un chart lors de son déploiement.

# Installation - Linux

- Par un script (recommandé)
  - `curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3`
  - `chmod +x get_helm.sh`
  - `./get_helm.sh`
- Par le gestionnaire de paquets
  - `sudo apt-get install helm`
  - `sudo yum install helm`
  - `sudo snap install helm --classic`

# Installation – Windows & Mac

- Par script – Comme sur linux
- Windows: Par chocolatey
  - *choco install kubernetes-helm*
- Mac: Par Homebrew
  - *brew install helm*



# Commandes Essentielles

## Gestion des releases

- *helm install <release> <chart>*  
Installe une nouvelle release à partir d'un chart.
- *helm upgrade <release> <chart>*  
Met à jour une release existante avec une nouvelle version du chart ou de ses valeurs.
- *helm rollback <release> [révision]*  
Revient à une version précédente d'une release.
- *helm uninstall <release>*  
Supprime une release du cluster.
- *helm list*  
Liste toutes les releases déployées dans le cluster (ajoutez -A pour toutes les namespaces).

# Commandes Essentielles

## Exploration des ressources

- *helm get all <release>*  
Affiche toutes les informations sur une release (manifests, valeurs, hooks, etc.).
- *helm get values <release>*  
Affiche les valeurs utilisées pour une release.
- *helm get manifest <release>*  
Affiche le manifeste Kubernetes généré par Helm pour une release.
- *helm history <release>*  
Affiche l'historique des révisions d'une release.
- *helm status <release>*  
Donne l'état actuel d'une release.

# Commandes Essentielles

## Gestion des dépôts

- *helm repo add <nom> <url>*  
Ajoute un dépôt de charts Helm.
- *helm repo update*  
Met à jour la liste locale des charts disponibles dans les dépôts.
- *helm search repo <mot-clé>*  
Recherche un chart dans les dépôts configurés.
- *helm repo list*  
Liste les dépôts ajoutés localement.
- *helm pull <chart> [--untar]*  
Télécharge un chart localement (optionnellement le décompresse).
- *helm show values <chart>*  
Affiche les valeurs de configuration par défaut d'un chart.

# Commandes Essentielles

## Création des charts

- *helm create <nom-du-chart>*  
Génère la structure de base d'un nouveau chart Helm.
- *helm package <dossier-du-chart>*  
Crée une archive du chart pour distribution.
- *helm lint <dossier-du-chart>*  
Vérifie la syntaxe et la cohérence d'un chart.

# Commandes Essentielles

## Personnalisation des déploiements

- *helm install ... --values <fichier.yaml>*  
Utilise un fichier de valeurs personnalisé pour configurer le chart.
- *helm install ... --set clé=valeur*  
Surcharge une ou plusieurs valeurs à la volée via la ligne de commande.
- L'ordre de priorité lors de l'application des valeurs est :  
values.yaml < fichier passé avec -f < valeurs passées avec --set.

# Commandes Essentielles

## Tests

- *helm template <chart>*  
Génère les manifests Kubernetes sans les appliquer, pour inspection locale.
- *helm test <release>*  
Lance les tests définis dans le chart pour vérifier le bon fonctionnement de la release.

# Templates

- Helm utilise le moteur de templates de Go (Go templating engine) enrichi par la bibliothèque de fonctions Sprig, ce qui permet de générer dynamiquement des fichiers manifestes Kubernetes à partir de modèles paramétrables.
- **Syntaxe Go Template** : Les fichiers dans le dossier templates/ d'un chart Helm sont principalement des fichiers YAML dans lesquels on insère des expressions entre doubles accolades `{{ ... }}`.
- **Variables et objets intégrés** : Helm expose plusieurs objets spéciaux dans le contexte du template, comme `.Values` (valeurs utilisateur), `.Release` (informations sur la release), `.Chart` (métadonnées du chart), etc.
- **Fonctions Sprig** : Helm intègre la librairie Sprig, offrant de nombreuses fonctions utilitaires pour manipuler des chaînes, des listes, des dates, etc. Exemples :
  - `{{ upper .Values.env }}` pour mettre en majuscule une variable.
  - `{{ default "valeurParDefaut" .Values.option }}` pour définir une valeur par défaut

# Templates

- **Contrôle de flux** : On peut utiliser des structures de contrôle comme les conditions (**if**, **else**) et les boucles (range) pour générer dynamiquement des parties du YAML.
- **Templates nommés et helpers** : Il est possible de factoriser du code réutilisable dans des fichiers comme `_helpers.tpl` grâce à la directive `define/template`.

```
{{- if .Values.autoscaling.enabled }}  
# YAML block for autoscaling  
{{- end }}
```

```
{{- range .Values.ports }}  
- containerPort: {{ . }}  
{{- end }}
```

```
{{- define "monchart.fullname" -}}  
{{ printf "%s-%s" .Release.Name .Chart.Name }}  
{{- end }}
```



# TP: Créer un Chart

- Créer un dossier
- Exécuter la commande `$helm create my-chart`
- Les fichiers présents dans le Chart sont des templates
  - Ce qui apporte énormément de flexibilité dans la définition du Chart
- Arborescence:
  - Le dossier charts contient les dépendances aux autres charts
  - Le dossier templates contient tous les templates des ressources kubernetes
  - Le fichier Notes.txt représente la documentation du chart
  - Le fichier `_helpers.tpl` contient des fonctions utiles pour s'assurer du bon formatage des templates
  - Le fichier values.yaml centralise les valeurs utilisées par les templates

# Conclusion

# Conclusion

- Les architectures en micro-services ont beaucoup gagné en popularité ces dernières années
- Ils apportent réactivité et flexibilité
- Les conteneurs permettent de rapprocher les métiers du développement et ceux de l'opérationnalisation et de la maintenance
- Chaines CI/CD plus fluides, plus efficaces
- Cependant, le déploiement dans le cloud peut poser bien des soucis en termes de qualité de service ou tolérance aux pannes
- Kubernetes permet aux développeurs de s'affranchir d'une grande partie de ces problèmes
- Mais nécessite une bonne compréhension de son architecture
- Pour aller plus loin:
  - Comme nous l'avons vu, le versionnement d'une application à base de micro-services n'est pas aisée
  - Helm est une surcouche à Kubernetes permettant de gérer une architecture comme un tout cohérent

# Conclusion

- **Flexibilité et Portabilité** (Docker vs VMs) : Docker offre une grande flexibilité et portabilité par rapport aux machines virtuelles traditionnelles. Les conteneurs Docker sont plus légers, démarrent plus rapidement et nécessitent moins de ressources, ce qui les rend idéaux pour les déploiements rapides et scalables.
- **Orchestration** (Kubernetes vs Docker Swarm) : Kubernetes est souvent comparé à Docker Swarm. Bien que Docker Swarm soit plus simple à utiliser et à configurer, Kubernetes offre une meilleure scalabilité, une gestion plus robuste des conteneurs et une communauté plus large et plus active.
- **Écosystème et Communauté** : Kubernetes bénéficie d'un large écosystème et d'une communauté dynamique, offrant une vaste gamme de plugins et d'outils pour la gestion de cluster, la surveillance et la sécurité. Cette richesse de ressources peut être un avantage significatif par rapport à d'autres solutions moins matures ou moins soutenues.
- **Complexité et Courbe d'Apprentissage** : Kubernetes est plus complexe et a une courbe d'apprentissage plus raide par rapport à d'autres solutions comme Docker Swarm ou Mesos. Cela peut être un facteur à considérer pour les équipes avec des ressources ou des compétences limitées.