Mise en place d'un cluster haute disponibilité avec Docker et Kubernetes

Avertissement

Cette présentation porte uniquement sur kubernetes et la haute disponibilité. **Bien que Docker soit utilisé, aucun détail ne sera donné sur son utilisation.**

La situation à l'Enssib :

- Plusieurs serveurs physiques différents en debian 7 et 8.
- Utilisation de LXC pour créer des machines virtuelles.
- Seul le mail est en haute disponibilité.
- Forte dépendance de l'admin système en cas de problème physique sur un serveur.
- Documentation propre à chaque serveur et complexe à maintenir.

Les objectifs

- Mise en place d'un cluster permettant de virtualiser différents services (mail, web, ldap, sso...)
- Haute disponibilité : La défaillance d'une machine du cluster doit être transparente.
- Légèreté : Utilisation d'un moteur de virtualisation qui ne virtualise pas tout un OS.
- Simplicité : Tout le Système d'Information doit pouvoir être décrit dans des fichiers simples.
- Performant et évolutif : Possibilité d'ajouter des nœuds dans le cluster.
- Libre, gratuit et documenté.

Docker

Docker est un moteur de virtualisation basé sur les cgroups du noyau Linux.

- On déploie des « containers ».
- Les images de ces containers sont stockées dans un registry.
- On ne virtualise que l'espace utilisateur, toute les VM utilisent le noyau de l'hôte.
- Les containers sont sans état. Toute modification faite sera perdue à l'arrêt du container.
- La construction d'un container est entièrement décrite dans un fichier Dockerfile.

Docker ne gère pas la partie cluster ou haute disponibilité. Il nous faut un « orchestrateur » qui va, au travers d'un cluster, lancer des containers avec docker.

- Swarm : L'outil de docker. Très incomplet et trop jeune au moment du choix.
- CoreOS : OS trop minimaliste, pas de gestion des volumes iSCSI par exemple.
- Mesos : Utilisation trop complexe.

Kubernetes a été choisi.

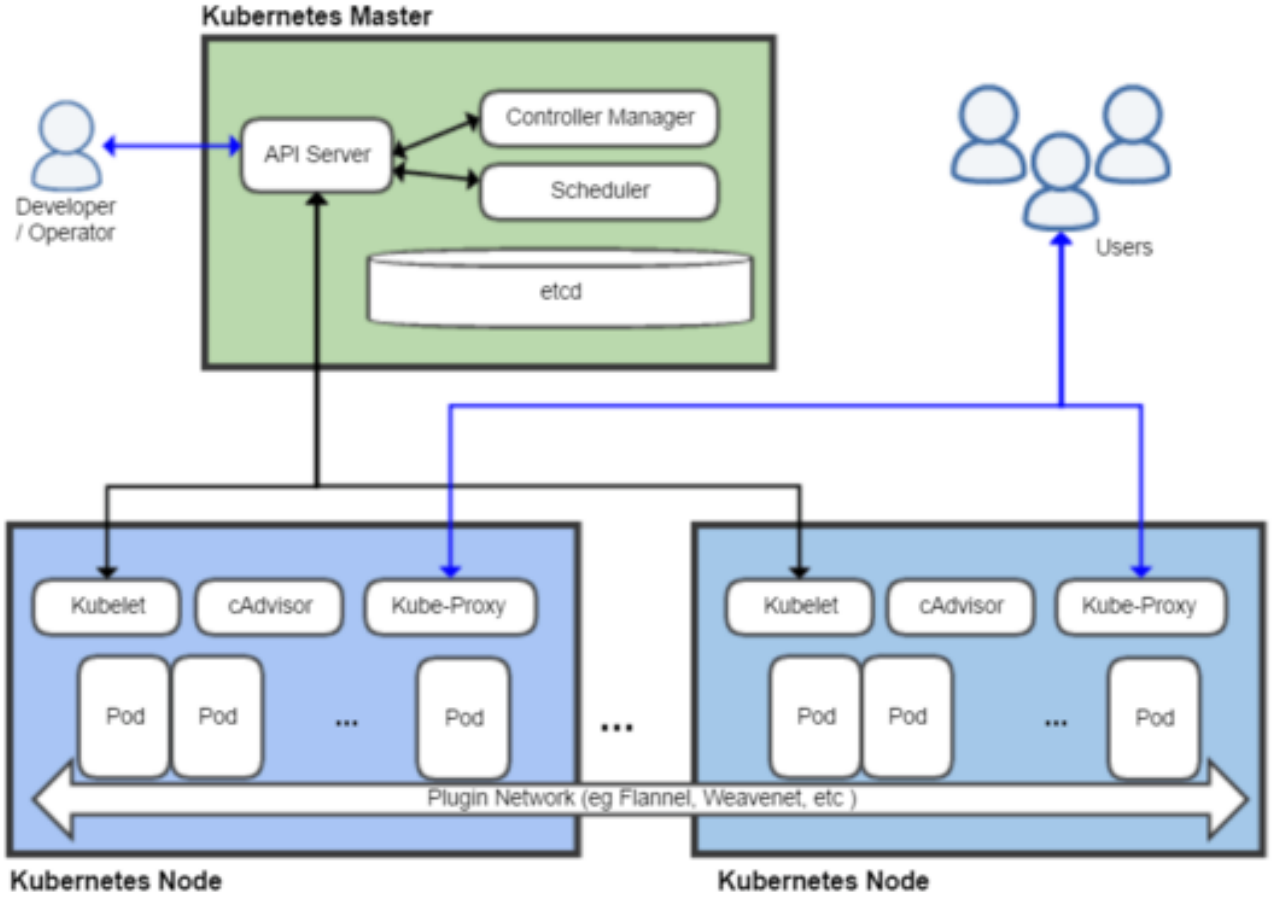
- Créé par Google, est open source et dispose d'une communauté active.
- Utilisation de fichiers yaml pour la configuration et de etcd (backend de stockage clé/valeur) pour stocker ces configurations.
- Pas de contrainte de distribution ou d'OS spécifique.
- Toute la gestion se fait via une unique commande kubectl.

Architecture

Un peu de vocabulaire

- Master est une machine qui va faire tourner les processus d'orchestration du cluster
- Worker est une machine qui va exécuter les ordres du master et donc faire tourner les VM.
- Un pod est une VM docker. On n'y accède pas directement, il tourne sur un des workers mais il est inutile de connaître son IP ou sur quelle machine il est exécuté.
- Un service est une IP sur un réseau privée dédié (réseau du cluster) qui sera contactée pour accéder au service demandé. Le service est directement relié au(x) pod(s) exécutant ce service. Le DNS du cluster associe un service à son IP.

Architecture (image wikipedia)



Architecture

- Etcd: Démon permettant de stocker des couples clés/valeurs. Ne fait pas partie de kubernetes.
- API serveur : Interface centrale du cluster, tous les éléments du cluster communiquent avec en HTTPS (JSON over HTTP). Il stocke sa configuration dans etcd.
- Controller-manager : Démon chargé de surveiller les pods à lancer ou à effacer.
- Scheduler : démon chargé de trouver le bon worker sur lequel lancer un pod
- Kubelet : démon chargé le lancer les pods qui lui sont assignés.
- Kube-proxy : démon permettant de rendre accessible le réseau des services sur chaque worker.

Contraintes

- Il faut au moins 2 masters pour assurer la haute disponibilité.
- Il faut que etcd tournent sur 3 machines au moins (nombre impair de machines)
- Chaque worker fait tourner ses pods sur son propre réseau privé et tous les pods du cluster doivent communiquer entre eux et accéder aux services.
- Les données doivent être accessibles sur chaque worker.

Installation à l'Enssib

- 2 masters data1 et data2 en debian 8 qui assureront :
  - les services du cluster (API server, DNS, scheduler, ...)
  - Le stockage de données
  - L'accès réseau
- 3 workers node01, node02 et node03 en debian 8 qui exécuteront les VM. Ces workers sont interchangeables, ne stockent aucune données et ne sont pas accessibles par les utilisateurs.
- Utilisation de systemd sur chaque machine pour lancer les différents démons et les relancer en cas de crash.

Le Stockage

Le stockage

Les VM étant sans états, il faut monter des volumes pour les données persistantes (base de données, LDAP, mail…).

Ces volumes ne peuvent être stockés localement sur l'hôte car par défaut, une VM n'est pas liée à un hôte.

Ces volumes doivent être répliqués en temps réel en cas de défaillance du serveur de stockage.

Le SAN du pauvre

- Disques durs identiques sur data1 et data2 et répliqués avec DRBD (bonding avec liens directs entre les 2 serveurs)
- Utilisation de LVM pour créé des « disques »
- Certains volumes étant montés à distance, on utilise des snapshots LVM pour la sauvegarde. Celle-ci se faisant uniquement sur le serveur de stockage. Inutile de sauvegarder les nœuds.

Export des disques

Plusieurs solutions pour exporter des disques :

- NFS : Pas de haute disponibilité par défaut. Pas très performant sur de multiples petits fichiers.
- OCFS2 : peu de différence de performance avec NFS et très punitif quand quelque chose ne lui plaît pas: blocage complet de l'hôte.
- iSCSI : Ne permet pas de monter plusieurs fois un volume en écriture.
- GlusterFS/Ceph : Non testé mais que 2 serveurs de stockage donc non pertinent.

Export des disques

Solutions choisies :

- iSCSI : Utilisation de scst. Haute disponibilité assurée par multipathd sur les noeuds. Chaque volume utilise le lien vers data1 et data2 (ATTENTION : nv_cache à 0 et write_through à 1 dans scst.conf. Lire la doc de scst pour éviter la corruption de données !!)
- NFS : Haute disponibilité assurée par une IP virtuelle et keepalived. Pas de TCP à cause du timeout de session en cas de défaillance. Kubernetes ne gère pas les options de montage donc on force NFS en version 3 et en UDP en utilisant un fichier nfsmount.conf (ATTENTION : Il faut un MTU de 9000 pour éviter la corruption sur un lien gigabit. Man nfs pour des détails).

Différents types de stockage

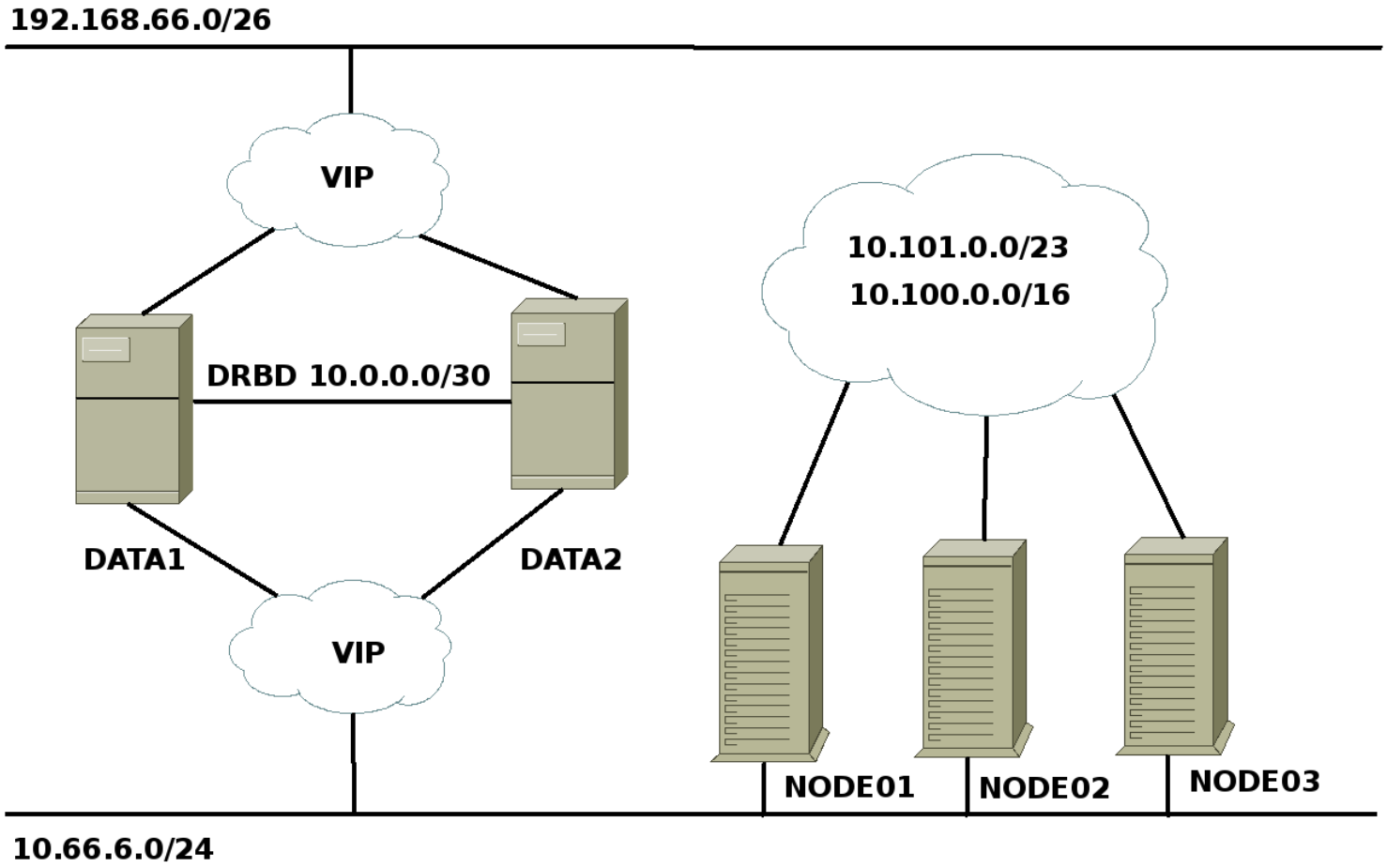
- Prise en charge native de nombreux protocoles comme iSCSI, NFS, GlusterFS, CephFS…
- Gestion des « secrets ». Ce qui permet de stocker les données sensibles (certificats, mot de passe…) dans l'API server. Ces données seront accessibles par le montage d'un volume de type secret.

Le Réseau

Les réseaux

- Réseau 10.100.0.0/16 : Réseau virtuel pour les pods
- Réseau 10.101.0.0/23 : Réseau virtuel pour les services
- Réseau 10.66.6.0/24 : Réseau non routé pour les machines du cluster
- Réseau 192.168.66.0/26 : Réseau routé sur lequel les utilisateurs se connectent.
- Attribution d'un sous-réseau fixe pour chaque worker (10.100.1.0/24 pour node01, 10.100.2.0/24 pour node02…) dans le réseau des pods et mise en place des routes sur chaque worker. (Utilisation possible de sous-réseaux attribués par le cluster et de flannel pour les faire communiquer)

Le réseau


192.168.66.0/26
VIP
10.101.0.0/23
10.100.0.0/16
DRBD 10.0.0.0/30
DATA1
DATA2
VIP
NODE01
NODE02
NODE03
10.66.6.0/24

La catastrophe de l'IP source

- On ne peut pas savoir sur quel worker va s'exécuter un pod ni son IP. Seule l'IP (externe ou interne) du service est connue.
- Il est possible de fixer à un service une IP sur le réseau routable afin que les utilisateurs puissent y accéder. Le service s'occupe de router les paquet au(x) bon(s) pods mais on perd l'information de l'IP source qui est remplacée par l'IP du worker qui a reçu le paquet.

Solutions pour l'IP source

- Déclaration sur un nœud d'une IP routable associée à un service. Installation de nginx qui va envoyer les paquets sur l'IP interne non routable du service. On a ainsi l'IP source dans les logs de nginx et on peut filtrer. Mais cela ne marche pas pour un radius qui base sa réponse sur l'IP source.
- Installation d'un load balancer externe (assez technique à mettre en place mais faisable)
- Installation d'un keepalived capable de lire les infos du cluster dans l'API server.

Keepalived-vip à la rescousse

Keepalived-vip fait partie du dépôt contrib de kubernetes. Il fait office de load balancer.

- Lecture des IP des pods d'un service dans l'API server.
- Génération d'un keepalived.conf suivant un template.
- Mise à jour en temps réel.
- Pas besoin de machine externe.
- Utilisation de LVS NAT pour router les paquets directement vers les pods sans passer par les services.

Contrainte : tous les workers doivent avoir comme route par défaut l'IP de la machine faisant tourner le keepalived (afin d'être sûr d'intercepter les paquets retour et remettre les bonnes IP)

=> Tout le trafic réseau « utilisateur ↔ cluster » passe à travers une seule machine.

Le filtrage

Par défaut, tous les pods communiquent entre eux et vers l'extérieur.

On distingue 2 types de filtrage. Ingress qui est le trafic entrant vers un pod et egress qui est le trafic sortant d'un pod. Kubernetes gère le trafic ingress avec les network policies mais ne gère pas le filtrage par CIDR ni le trafic egress.

Il faut donc s'appuyer sur un outil tiers : Calico.

- Installation de Calico sur chaque nœud.
- Installation de Calico policy controller.

Le filtrage

Le filtrage se fait par port, par CIDR ou par label.

- Le filtrage par port et par CIDR est une base mais n'est pas suffisant car on ne connaît pas les IP des différents pods.
- Le filtrage par label permet de s'affranchir des IP. Chaque pod a un ou plusieurs label et on peut maintenant dire que les pods avec le label A ne peuvent communiquer sur le port X qu'avec les pods ayant le label B.

Exemple de filtrage

```
apiVersion: v1
kind: policy
metadata:
  name: www
spec:
  selector: name == 'apache'
  order: 100
  ingress:
  - action: allow
    protocol: tcp
    destination:
      ports: [80,443]
  egress:
  - action: allow
    protocol: tcp
    destination:
      ports: [3306]
      selector: unit == 'sql'
  - action: allow
    protocol: tcp
    destination:
      ports: [3128]
      net: 10.66.6.10/32
```

Mise en place à l'Enssib

Personnalisation

- Mise en place d'un registry privé qui va servir à accueillir toutes les images docker créées.
- Sécurisation de tous les démons kubernetes, etcd et du registry avec des certificats.
- Écriture de scripts pour automatiser les tâches de création d'image, dépôt dans le registre et création/mise à jour des pods
- Écriture de scripts pour mettre à jour tout le cluster en cas d'ajout de volumes iSCSI.
- Personnalisation du template keepalived.
- Mise en place de bonding sur les liens réseau des masters.
- MTU fixé à 9000 sur toutes les interface en 10.0.0.0/8.
- Modification de keepalived-vip pour avoir plusieurs services par IP.

Serveur MySQL en cluster :

- Sql1 sur node01
- Sql2 sur node02
- Sql3 sur node03

Le /var/lib/mysql de chaque worker est monté dans le pod faisant tourner mysql pour des raisons de performance (seules données présentes sur les workers)

Utilisation

- Toute l'administration se fait à travers la commande kubectl et des fichiers de configuration en JSON ou YAML.

- Exemples :
  - Kubectl create -f monfichier.yaml
  - Kubectl delete deployment monservice
  - Kubectl get po -o=wide

Exemple fichier service

```
kind: Service
apiVersion: v1
metadata:
  name: ldap
spec:
  selector:
    name: ldap
  ports:
  - name: ldap
    protocol: "TCP"
    port: 389
    targetPort: 389
  - name: ldaps
    protocol: "TCP"
    port: 636
    targetPort: 636
```

Fichier de déploiement 1/3

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ldap
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: ldap
```

Fichier de déploiement 2/3

```
spec:
      restartPolicy: "Always"
      containers:
      - name: ldap
        imagePullPolicy: Always
        image: registry.cluster.enssib.fr:5000/ldap
        ports:
        - containerPort: 389
          containerPort: 636
        volumeMounts:
        - mountPath: "/var/lib/ldap"
          name: ldap-iscsi
        - mountPath: "/etc/ldap/certs"
          name: "certificate"
```

Fichier de déploiement 3/3

```
volumes:
    - name: ldap-iscsi
      iscsi:
        targetPortal: 10.66.6.1:3260
        iqn: iqn.2016-04.fr.enssib:ldap
        lun: 0
        fsType: ext4
        readOnly: false
    - name: "certificate"
      secret:
        secretName: "cert-ldap"
```

Bénéfices

- Architecture redondée et serveurs interchangeables (maintenance matérielle inutile)
- Tous les pods sont construits de zéro à chaque fois à partir du fichier Dockerfile et de scripts => Tout le SI peut être mis dans un dépôt git.
- Évolutif, en cas de besoin, on peut ajouter des serveurs en tant que workers.
- Il est possible de taguer des serveurs pour n'autoriser le lancement de certains pods que sur ces serveurs (sur des serveurs puissants par exemple)
- Avec les bons scripts, l'administration devient très simple.

Le matériel

Pour les masters :

- 8 Go de RAM
- Carte contrôleur RAID avec batterie et mémoire
- Disques durs en fonction des besoins (SSD, gros disque magnétique…)
- 8 ports réseau

Pour les workers :

- 32 Go de RAM
- CPU puissant
- 2 ports réseau
- Disque système en SSD

Retour d'expérience

Difficultés rencontrées

- Kubernetes reste un produit très jeune avec des lacunes (pas de gestion native de la haute disponibilité de kubernetes lui-même, pas de filtrage egress, pas d'option de montage des volumes...) mais qui évolue très vite.
- Kubernetes bien conçu pour s'interfacer avec un cloud Google, Amazon ou Azure mais beaucoup moins dans un environnement sans cloud.
- Nombreux essais, de débugage, de tests de performance, de destruction/reconstruction de système.
- Travail conséquent d'écriture de scripts de gestion.
- La « Dockerisation » de service n'est pas toujours évidente et il faut ajouter à cela l'intégration dans le cluster.

Services dockerisés

Quelques services de base:

- DNS: Plusieurs réplicats
- LDAP: Un en écriture et plusieurs réplicats en lecture seule.
- Radius : Plusieurs réplicats prenant leurs infos dans l'Active Directory.
- Registry : Outil de stockage local d'image docker

Services dockerisés

Quelques services plus évolués :

- Mail : container postfix/dovecot
- MySQL : Utilisation de Percona MySQL qui permet d'avoir 3 serveurs MySQL répliqués.
- Apache : serveur web et proxypass d'autres services du cluster (sso, alfresco...)
- SSO : Serveur tomcat faisant tourner CAS.
- GED Alfresco : Serveur Tomcat.
- SIGB Koha : Serveur apache avec démons en Perl.
- Elasticsearch : 3 instances en cluster.

Points à améliorer

La gestion de la DMZ n'est pas encore prise en charge. Si on veut de l'agrégation de lien, il faut ajouter 2 ports réseaux sur data1 et data2 pour être sur la DMZ, soit on met 2 machines en load balancer.

Actuellement, le stockage se fait sur un gros disque de 6 To répliqué. Envisager de mettre un SSD de 1 To répliqué en plus pour le stockage ayant besoin de beaucoup d'IOPS (mail, code php des sites web, fichiers alfresco...)

Quelques indications de temps

- Recherche préalable sur du virtualbox : Environ 15 jours. (essai de coreos puis debian)
- Installation simple de kubernetes : 2 jours. (L'outil kubeadm qui permet de déployer un cluster rapidement ne gère pas encore la haute disponibilité)
- Mise en place du stockage : Environ 20 jours de recherche et d'essais (ocfs2, nfs avec pacemaker, nfs avec keepalived, LVM, multipathd, recompilations de scst et nfs...)
- Le réseau : Environ 30 jours de recherche et d'essais (solution haproxy, nginx, flannel, LVS, keepalived-vip modifié…)
- Le filtrage réseau : Environ 30 jours de recherche et d'essais. (kubenet avec Network Policies, trireme, calico sans etcd puis avec etcd…)
- « Dockerisation » de services et écriture des fichiers yaml d'intégration dans le cluster : Environ 30 jours.

Mise en place ailleurs

- Achat de machines. Pour l'enssib, environ 12.000 euros TTC pour les 5 machines. Possibilité de récupérer des machines en fin de vie pour les nœuds.
- Possibilité de simplifier en utilisant un vrai SAN et des attachements iSCSI.
- Avec la doc et les scripts, un cluster fonctionnel peut être mis en route en 3-4 jours.
- Il reste la « Dockerisation » de service et les fichiers yaml pour intégrer dans le cluster (déclaration du service, du deployment, du filtrage…).

Questions