Kubernetes

Jean-marc Pouchoulon
Septembre 2022

Généralités

Pré-requis

Connaissance des :

- containers applicatifs (Docker).
- bases de Linux.
- bases de git.

NB : dans ce document hl est un alias pour la commande highlight pour magnifier les sorties yaml => Aucun rapport avec la syntaxe CLI de K8S.

Orchestrateur de containers

L’hébergement d’application doit répondre aux problématiques de :

- Scalabilité (capacité de l’application à absorber une montée en charge par une augmentation linéaire des ressources en infrastructure sous jacentes)
- Disponibilité (inutile de vous faire un dessin)

Reprendre l’héritage de la virtualisation

- Problème : une VM ou un ensemble de VM par application
- De la scalabilité sur une granulométrie à l’échelle d’une VM (ex 1 serveur web de plus pour absorber les connections)
- De la scalabilité verticale par augmentation des ressources n’est pas toujours efficace et pas sans limites.

=> la virtualisation c’est lourd et cher.

Les containers seuls ne répondent pas à la problématique

- Un hôte peut se « crasher ». Pas de H.A. !
- Scaler sur le même hôte n'a pas de sens.

  => « alors vint K8S »

  (8 lettres entre le K et le S de Kubernetes)

Kubernetes - K8S κυβερνήτης (*kubernḗtēs*).

- Orchestrateur issu de Google (Borg).
- Signifie le « pilote » de bateau en grec ancien.

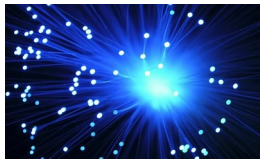
  => le mot « cybernétique »

  => Les cybernautes de « chapeau melon et bottes de cuir »

  (aucun rapport avec K8S mais cool non ?)

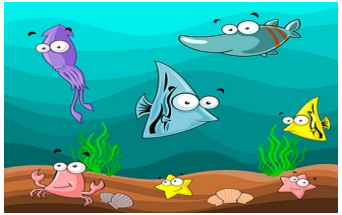
Points marquants de K8S

- K8S peut être vu comme :
  - un système d'exploitation répartie ou réseau.
  - une boucle disciplinée et obstinée.



- K8S a comme caractéristiques d'avoir:
  - Un couplage faible avec l'infrastructure.

  - Une même API pour tous les env.
  - Un fantastique écosystème.



Ce que fait Kubernetes

- Équilibrer dynamiquement de la charge en fonction de métriques (CPU, mémoire...).
- Gérer des applications.
- Mettre à jour sans interruption de services avec différentes stratégies de déploiements.
- « Scheduler » des tâches.
- Découvrir automatiquement des services.

L’avenir (radieux) de Kubernetes

- Kubernetes est devenu un moteur partagé universellement pas les éditeurs de solutions logicielles.
- Seul Mesos pour les gros clusters le concurrence vraiment. Swarm de Docker qui utilise aussi K8s et Nomad (Hashicorp) existent néanmoins...
- L’enjeu pour les D.S.I. est d’intégrer K8S dans le cycle de vie applicatif du développement à la production dans le CLOUD ou « on premise »
- On peut citer des distributions Kubernetes comme :

OPENSHIFT

RANCHER

VMware Tanzu

K3S

L’essentiel de l’infrastructure de Kubernetes

« control plane » & « data plane »

Le « control plane » de Kubernetes

Le « control plane » est en charge de mettre en œuvre ce qui lui est demandé par un utilisateur du cluster. Il est constitué de :

- L'API (REST) server qui est en charge de recevoir les requêtes et est le seul élément qui « discute » avec la couche de persistance du cluster (ETCD).
- ETCD est (en général) la couche de persistance qui contient l'état du cluster. Élément central du cluster, elle se plie au « *RAFT consensus* » et utilise plusieurs nœuds. Les informations sont stockées sous forme d'ensembles clefs/valeurs. Elle valide aussi l'authentification et l'accès (RBAC)
- Du « kube controller manager » qui monitore l'état des nœuds du cluster, s'occuper du « garbage collector » des nœuds et de PODS, de la création des NameSpaces pour le cluster. Il y a plusieurs types de « controllers ». C'est lui qui « boucle » et maintient l'état du cluster.
- Le « scheduler » alimente les nœuds afin de mettre en œuvre les ordres donnés en particulier le scheduling de POD, le respect des affinités et des labels, ainsi que les limites hardware imposées aux PODS.

E

Architecture de K8S

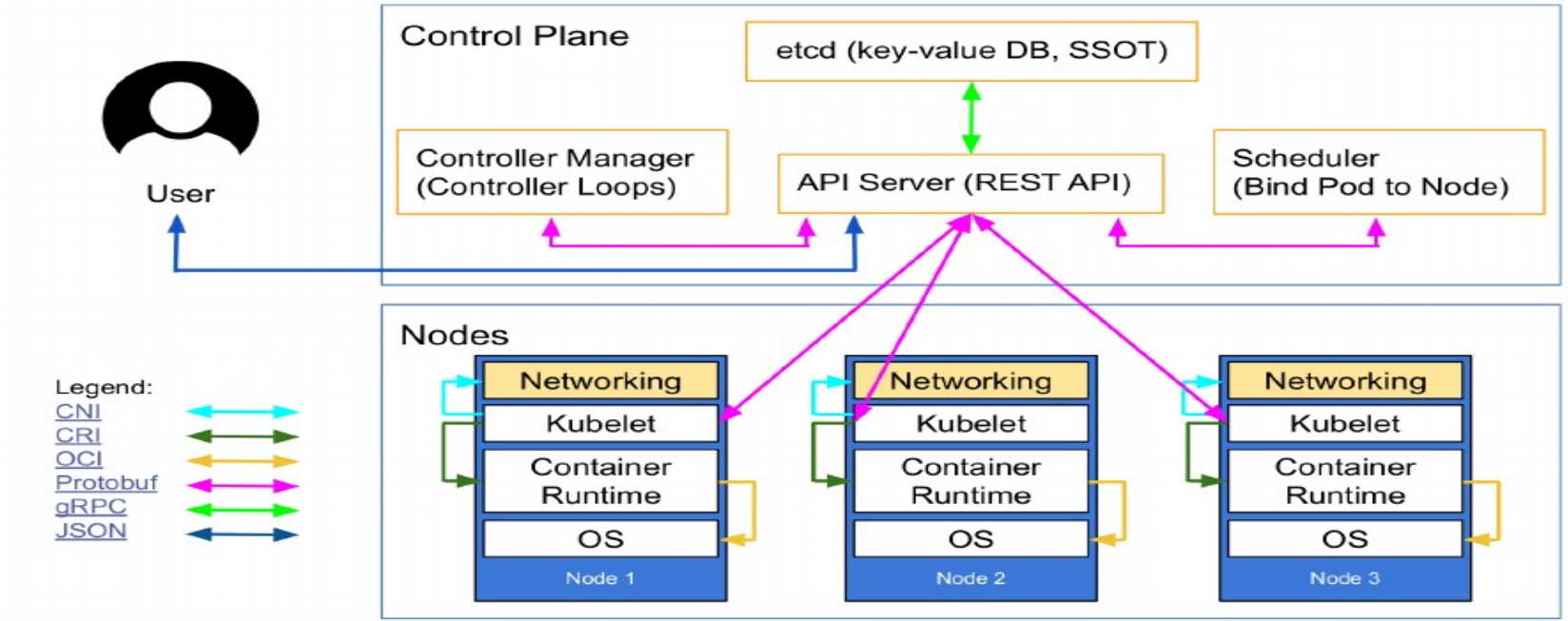


FIGURE 1 – Eléments du "control plane" et du "data plane" de Kubernetes[a]

a. extrait de https://github.com/cloud-native-nordics/workshop-infra avec l'aimable autorisation de son auteur Lucas Käldström.

Le « data plane » Kubernetes

Le « data plane » exécute les ordres du « control plane ». Sur chaque nœud du cluster on trouvera ses composants :

- « **kube proxy** » qui s'occupe de mettre en œuvre les ordres réseaux.Il est généralement basé sur la génération de règles par « iptables » (mais « LVS » , « kube router » sont aussi des solutions pour les « gros » clusters).
- « **kubelet** » qui reçoit les ordres du « control plane » et les met en œuvre.(met en œuvre les ordres du « controlplane » sur le cycle de vie des PODS.
- Un « **runtime** » parmi Docker , containerd, cri-o, katacontainers…. qui gère le cycle de vie des containers ou des machines virtuelles du cluster. Il est appelé par le « kubelet » du nœud. Les runtime implémente l'interface « **C**ontainer **R**untime **I**nterface ».

Les autres composants optionnels

- « kube DNS » qui s’occuper de la résolution des noms dans le cluster.
- « metrics API server » qui monitore les nœuds. Interrogeable par la commande « kubectl top nodes ».
- « cloud-controller-manager » qui permet de gérer les aspects spécifiques des « cloud providers ».

Le réseau avec Kubernetes

Le réseau avec Kubernetes

- La couche réseau de Kubernetes est fournie par un « plugin » d'une tierce-partie.
- Ce « plugin » implémente la « **C**ontainer **N**etwork **I**nterface ».
- Les plugins réseaux sont nombreux. *Flannel* est le plus générique , le plus simple mais limité en fonctionnalités .Il y a en a bien d'autres *Calico* , *Weave*… qui supportent des « Network Policies »

Caractéristiques des réseaux K8S

- Les réseaux sont propres au cluster.
- Tous les containers d’un même « POD » peuvent dialoguer ensemble.
- Tous les « PODS » communiquent avec les autres sans NAT.
- Tous les nœuds et tous les POD communiquent entre eux sans NAT.
- On peut filtrer (Network Policies) la communication entre POD mais pas avec tous les plugins.

Objets fondamentaux de Kubernetes

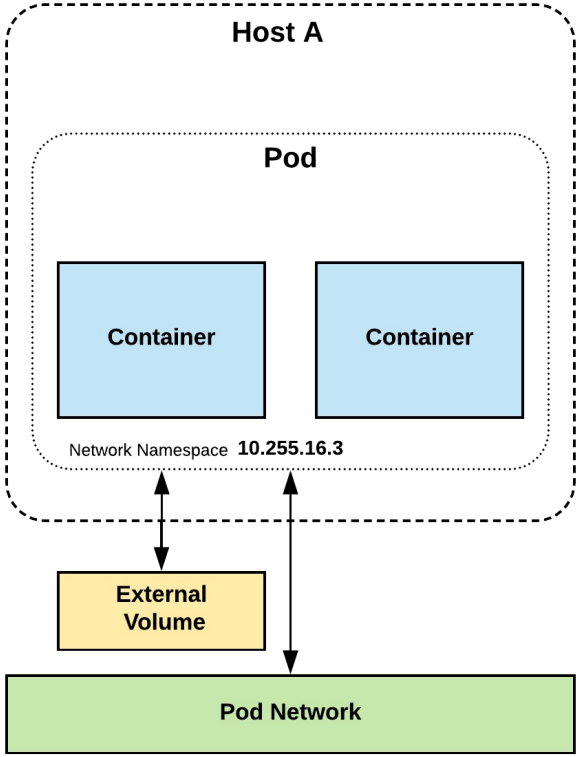
Les « Manifests » en YAML

Un « manifest » est un fichier Yaml qui décrit une ressource ou un ensemble de ressources d’un cluster Kubernetes.

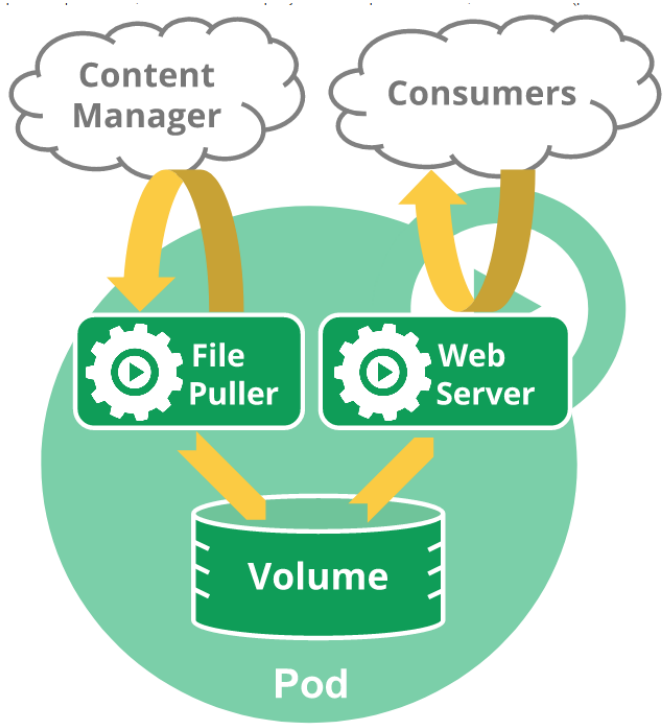
« POD » de Kubernetes

- Le pod est le plus petit objet déployable d'un cluster Kubernetes. Il est éphémère ("cattle" et pas "pet" !).
- Il est constitué par un ou plusieurs containers partageant les mêmes NameSpaces (réseaux, IPC).
- Les containers d'un POD sont aussi capables de partager entre eux des données au travers d'un partage sur l'hôte sur lequel ils s'exécutent.

POD



Source :https://github.com/mrbobbytables/k8s-intro-tutorials



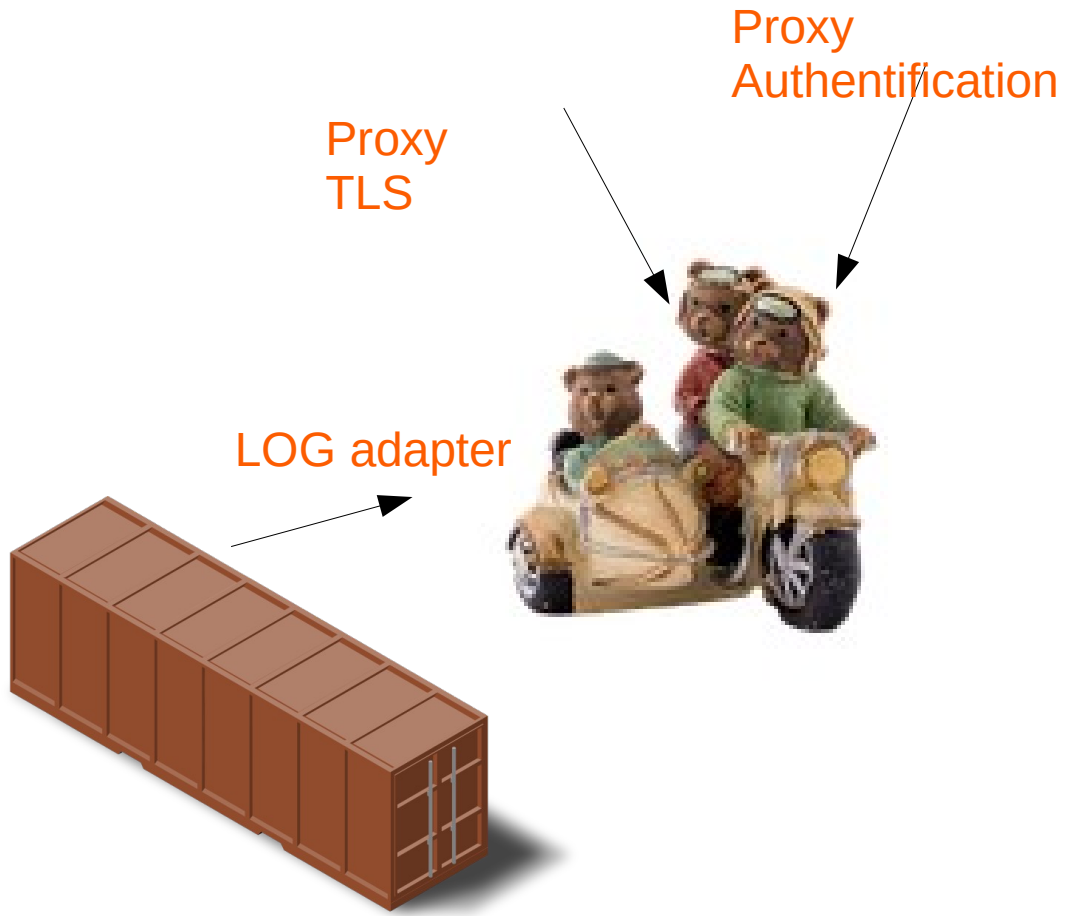
Source :https://kubernetes.io/fr/docs/concepts/workloads/pods/pod/

POD’s patterns

- sidecar
- proxy
- adapter

  => Proxy est

  inclu dans side-car



Name Spaces

- Notion partagée avec les containers (Network NameSpace, User NameSpace …)

```
❯ k create --dry-run=client namespace monAppli -o yaml |hl
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: monAppli
spec: {}
status: {}
```

NameSpaces génériques

- « **default** » ~= VLAN defaut
- « **kube-system** » contient les objets systèmes ~= VLAN admin
- « **kube-public** » NS en RO ~= VLAN invité

```
> kubectl get ns --show-labels|hl
NAME                   STATUS   AGE   LABELS
default                Active   29h   <none>
kube-node-lease        Active   29h   <none>
kube-public            Active   29h   <none>
kube-system            Active   29h   <none>
local-path-storage     Active   29h   <none>
metallb-system         Active   28h   app=metallb
```

Exemple de manifeste de POD

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: debianpod
  name: debianpod
spec:
  containers:
  - image: registry.iutbeziers.fr/debianiut:latest
    name: debianpod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Exemples de commandes relatives aux « PODS »

```
# un alias indispensable
alias kubectl=k
# générer le manifest d'un POD
k run --dry-run=client  debianpod --image=registry.iutbeziers.fr/debianiut:latest -o yaml > debianpod.yml
k create -f ./debianpod.yml # créer le pod à partir du manifeste
k get pod debianpod -o wide --show-labels # voir le noeud où s'exécute le pod
k describe pod/debianpod # dumper la configuration du pod
k exec -it debianpod -- bash # lancer un processus bash dans le pod
apt update && apt -y update && apt install apache2 && service apache2 start # Lancer apache dans le pod
k port-forward pod/debianpod  8002:80 & # nattez le port pour y accéder depuis l'hôte.
http -h  localhost:8002 # accéder au serveur apache
k logs pod/debianpod # voir les logs
k delete -f ./debianpod.yml # detruire le POD
```

REQUESTS / LIMITS

Requests

- Affect Scheduling Decision
- Priority (CPU, OOM adjust)

Limits

- Limit maximum container usage

```
resources:
  requests:
    cpu: 100m
    memory: 300Mi
  limits:
    cpu: 1
    memory: 300Mi
```


zalando

« Mode Management »

Mode impératif

L’utilisateur donne précisément les ordres nécessaires pour atteindre un but sans passer par un fichier yaml. Mode direct mais limité. Il n’est pas dans la philosophie DevOps/GitOps

```
1 kubectl run nginx-pod --image nginx
2 kubectl exec -it nginx-pod -- sh
3 kubectl create deployment hello-nginx --image nginx
4 kubectl scale deployment hello-nginx --replicas 2
5 kubectl expose deployment hello-nginx --type=LoadBalancer --port 80 --target-port 80
```

le « **mode impératif** » en utilisant des fichiers de configuration.

Ce mode permet d'avoir de la persistence mais il n'est pas *idempotent*. Deux personnes peuvent appliquées des changements et écraser les modifications de l'autre.

Exemples :

kubectl create -f config-objet.yaml

kubectl replace -f config-objet.yaml

kubectl delete -f config-objet.yaml

Mode impératif fichier vers mode déclaratif pour générer du yaml

kubectl create --dry-run=client -o yaml « objet » -output ...

objet = service, deployment, pods, secret,cronjob…

output = yaml, json...

exemple :

k run –dry-run=client pod1 –image=nginx -o yaml

Mode déclaratif

Le « **mode déclaratif »** indique l'état souhaité. Il s'agit d'une directory « $DIR », versionnée avec git dans laquelle on déclare les « manifests » des objets (NameSpace, deployments, services...). La commande « kustomize » intégrée à kubectl est pleinement déclarative et permet de générer des configurations à partir de l'existant sans le modifier.

« kubectl diff -f ${DIR}/ » permet d'afficher les contenus

« kubectl apply -f ${DIR}/ » permet d'appliquer les contenus

apply = create + replace » si les objets n'existent pas dans le cluster

« kubectl kustomize ou kubectl apply -k fichier » permet de générer des configurations en assemblant des fichiers yaml existants et en en générant des combinaisons.

Mode impératif fichier vers mode déclaratif pour générer du yaml

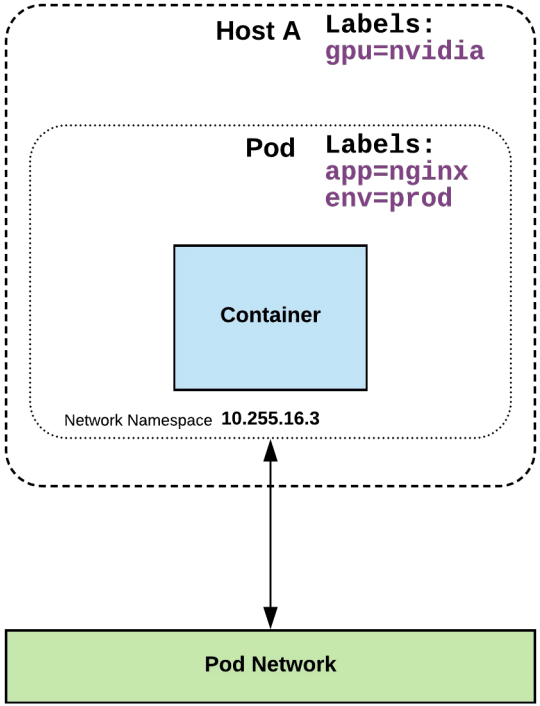
```
❯ cat ./create-deploy.sh|hl
alias k=kubectl
# génération d'un "deployment" dans un fichier
k create deploy deploy1 -oyaml --image=busybox --dry-run=client > deploy1.yaml
# ajout de "requests" et "limits" dans le même fichier
k run deployment \
    -oyaml \
    --dry-run=client \
    --image=busybox \
    --requests "cpu=100m,memory=256Mi" \
    --limits "cpu=200m,memory=512Mi" \
    --command \
    -- sh -c "sleep 1d" >> mondeploiement.yaml
# application
k apply -f mondeploiement.yaml
```

Labels & Selecteurs

Labels

Ce sont des paires clefs-values utilisées pour selectionner un ensemble d'objets Kubernetes.

- Un déployment retrouve « ses pods » au travers d'un label app=monapplication
- On tagge les « pods » pour la production ou le dev
- On taggue les nœud en fonction du hardware pour y « sticker » certains PODS (gpu,ssd..)

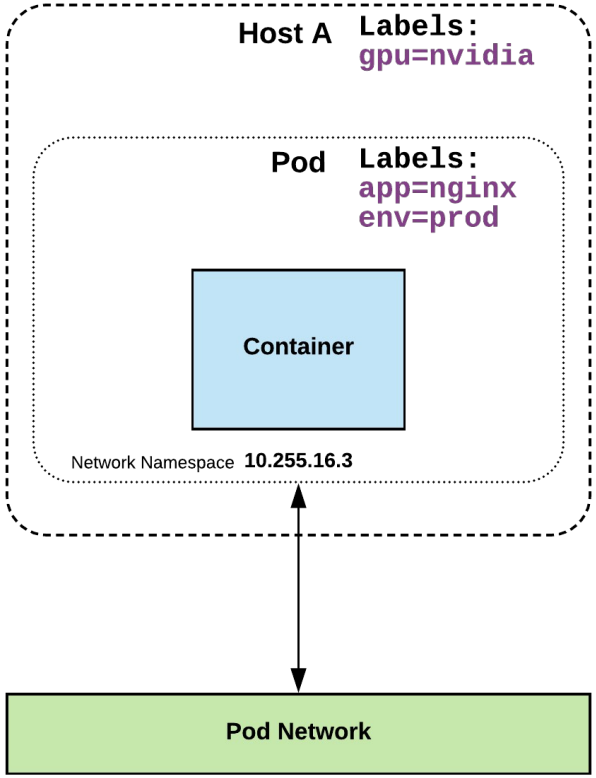


Source :https://github.com/mrbobbytables/k8s-intro-tutorials

Exemple de label

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-exemple
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

Source :Bob Killen
https://github.com/mrbobbytables/k8s-intro-tutorials



Sélectionner un nœud via un label

Un selecteur utilise

un label pour filtrer

et sélectionner un objet

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-exemple
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
  nodeSelector:
    gpu: nvidia
```

configmap et secret

Configmap et secret

- Configmap est un objet qui permet de stocker des configurations sous forme clefs/valeurs (par exemple un fichier de configuration Nginx)
- Secret permet de stocker des mots de passe. La sécurité repose sur les RBAC. Des outils comme Hashicorp/Vault peuvent être utilisés en complément.

Les « deployments »

« Deployments » et « DaemonSets »

- Un POD ne sait pas se multiplier pour absorber la charge.

  => Il faut donc d'autres objets pour assurer la montée en charge.

  => Les « Deployments » et les « DaemonSets » répondent à cette problématique :

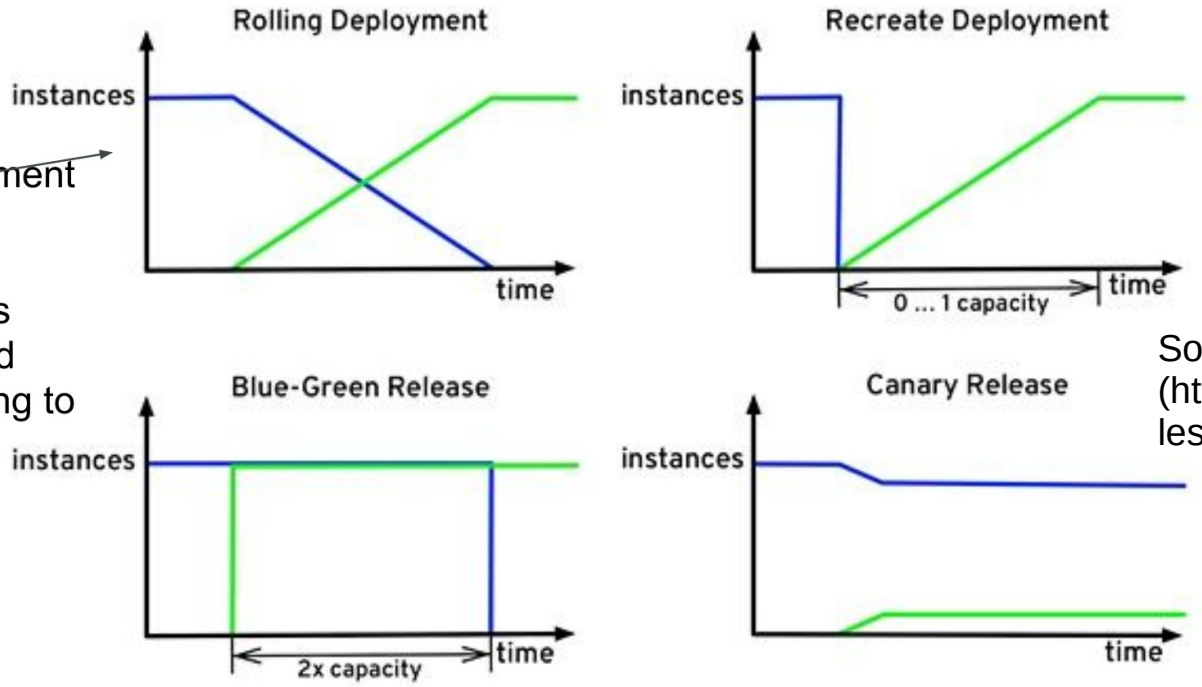
  Un DaemonSet réplique un « POD » par Noeud et s'assure qu'en cas de créations d'un nouveau nœud un pod est créé.

  Un « Deployment » lui va « scaler » le nombre de « PODS » au nombre demandé.

Style de déploiement

The built-in Deployment behavior

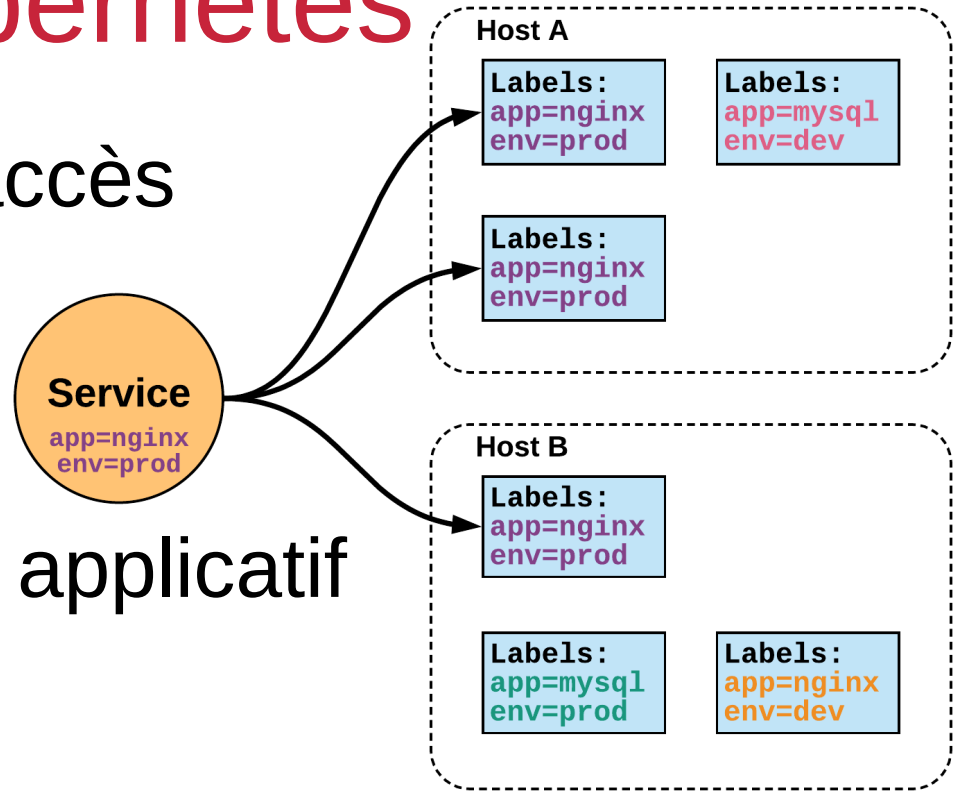
The other strategies can be implemented fairly easily by talking to the API.



Source (https://www.infoq.com/articles/kubernetes-effect/)

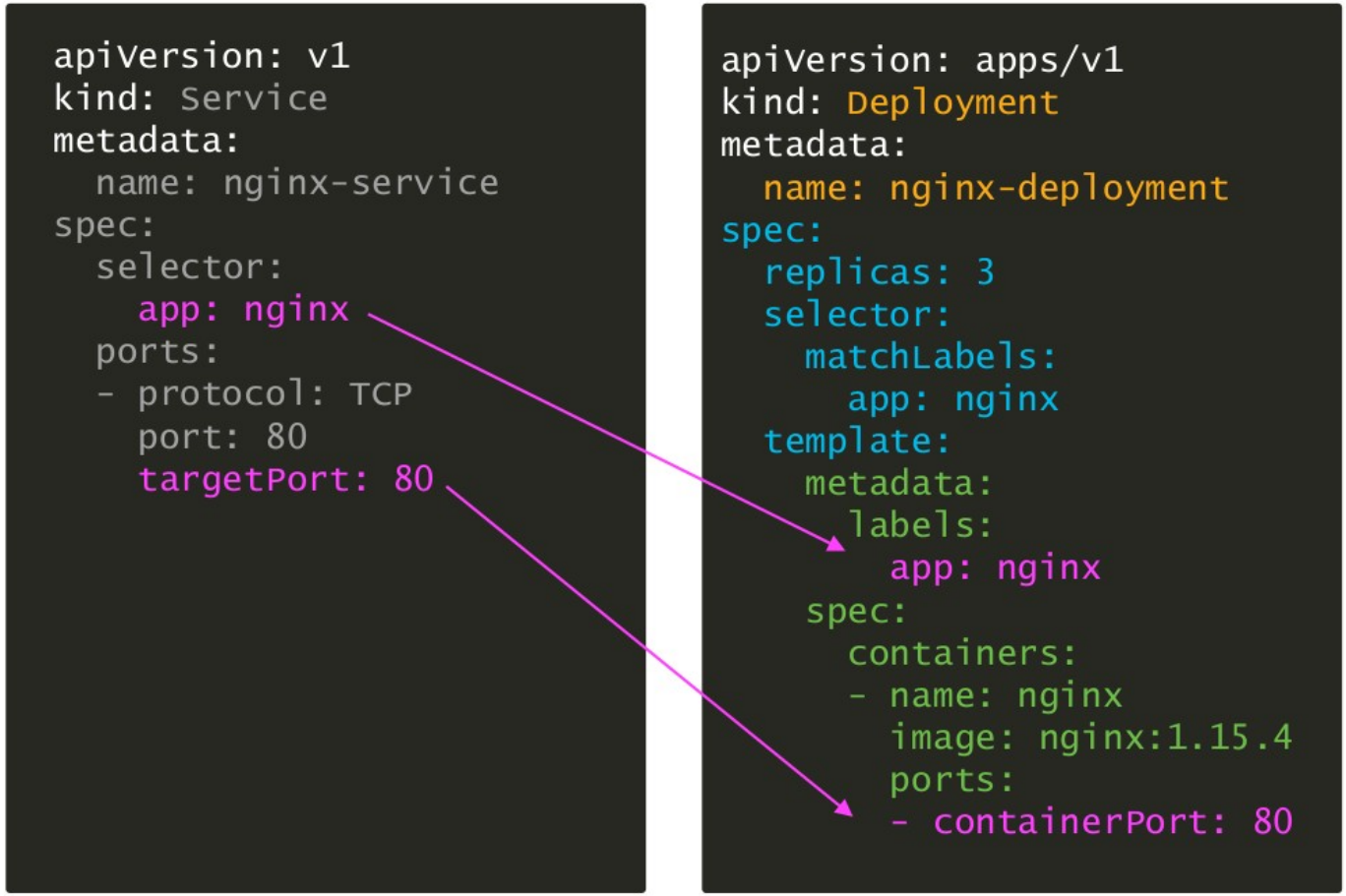
Les services Kubernetes

- Un service permet l'accès à une application via TCP ou UDP.
- Il réunit un ensemble applicatif au travers de labels.



Source :https://github.com/mrbobbytables/k8s-intro-tutorials

Connecting Applications to Services



Service

Deployment

Les services Kubernetes

Les services

- Un POD meurt et se recréé sur un autre nœud avec une nouvelle adresse IP

  => **on ne peut donc pas se fier à l'IP du POD pour construire une architecture applicative**.

  => La notion de service est la pour pallier ce manque.
- Ce sont les kube-proxy de chaque noeud qui vont se charger de faire un reverse-proxy vers les Pods derrière le service.
- Un service est « niveau 4 » TCP ou UDP.

4 différents types de services

- ClusterIP (service par défaut)
- LoadBalancer
- NodePort
- ExternalName

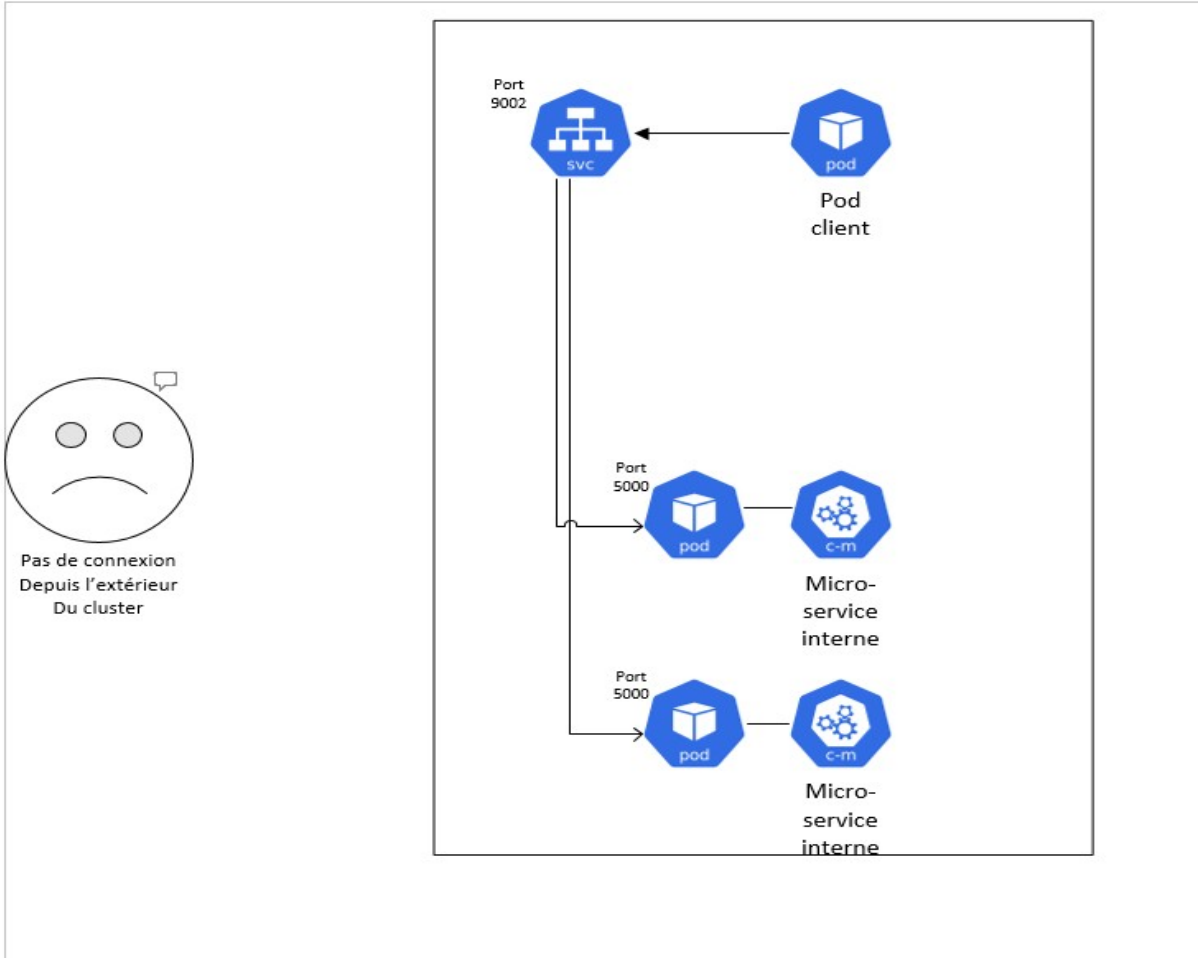
Service ClusterIP

Une « Virtual IP Address » est allouée pour le service dans le cluster.

- Elle est interne au cluster.
- Elle est accessible depuis les nœuds et les « PODS » du cluster.

=> cas d'utilisation : communications internes au cluster si les services n'ont pas besoin d'être accéder depuis l'extérieur du cluster.

ClusterIP

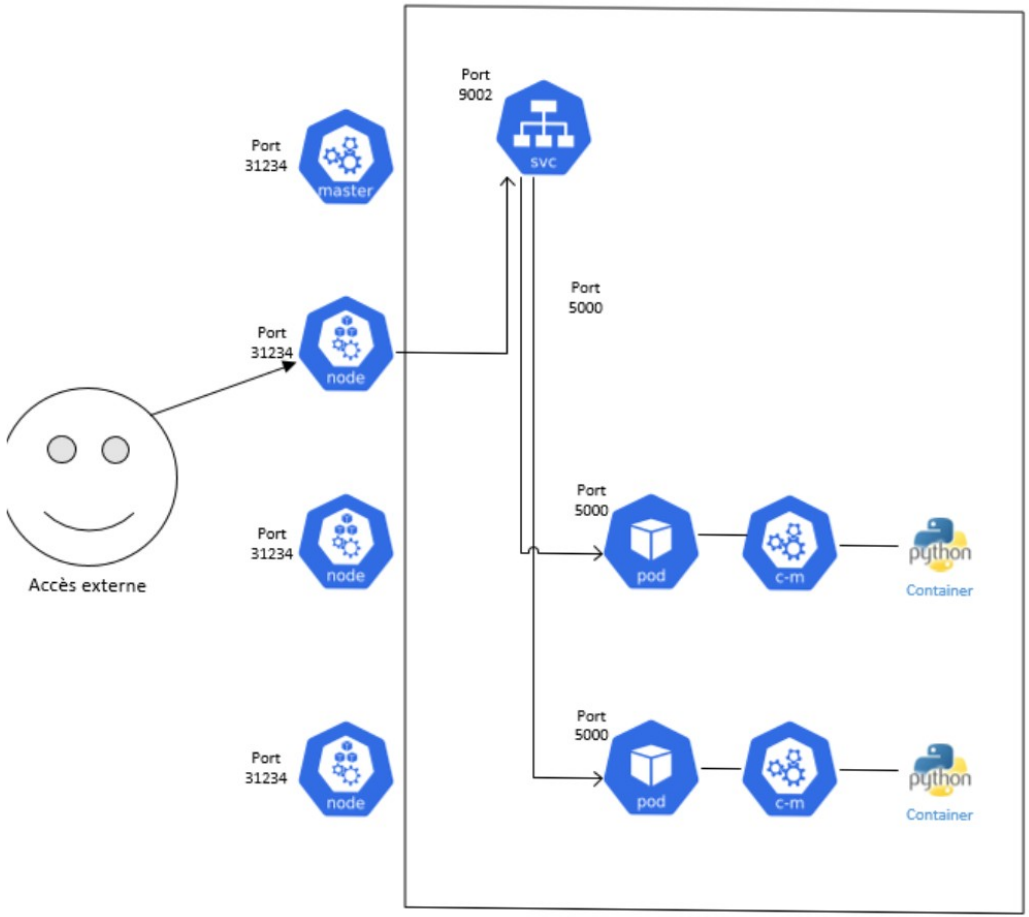

Port
9002
svc
pod
Pod
client
Pas de connexion
Depuis l'extérieur
Du cluster
Port
5000
pod
c-m
Micro-
service
interne
Port
5000
pod
c-m
Micro-
service
interne

Service NodePort

- Les Services de type **NodePort** permettent d'exposer le service sur un port aléatoire choisi entre 30000 et 32767 sur chaque noeud du cluster.
- Ce port sera joignable sur tous les noeuds du Cluster qu'il héberge un Pod éligible ou non.
- Les nœuds d’un cluster K8S dans le CLOUD n’ont pas toujours une IP publique.

  => On peut mettre en frontal un équilibreur de charge traditionnel mais il faut maintenir le lien ou que l’équilibreur dialogue avec Kubernetes.

NodePort


Port
9002
svc
Port
31234
master
Port
31234
node
Port
5000
Accès externe
Port
31234
node
Port
5000
pod
c-m
python
Container
Port
31234
node
Port
5000
pod
c-m
python
Container

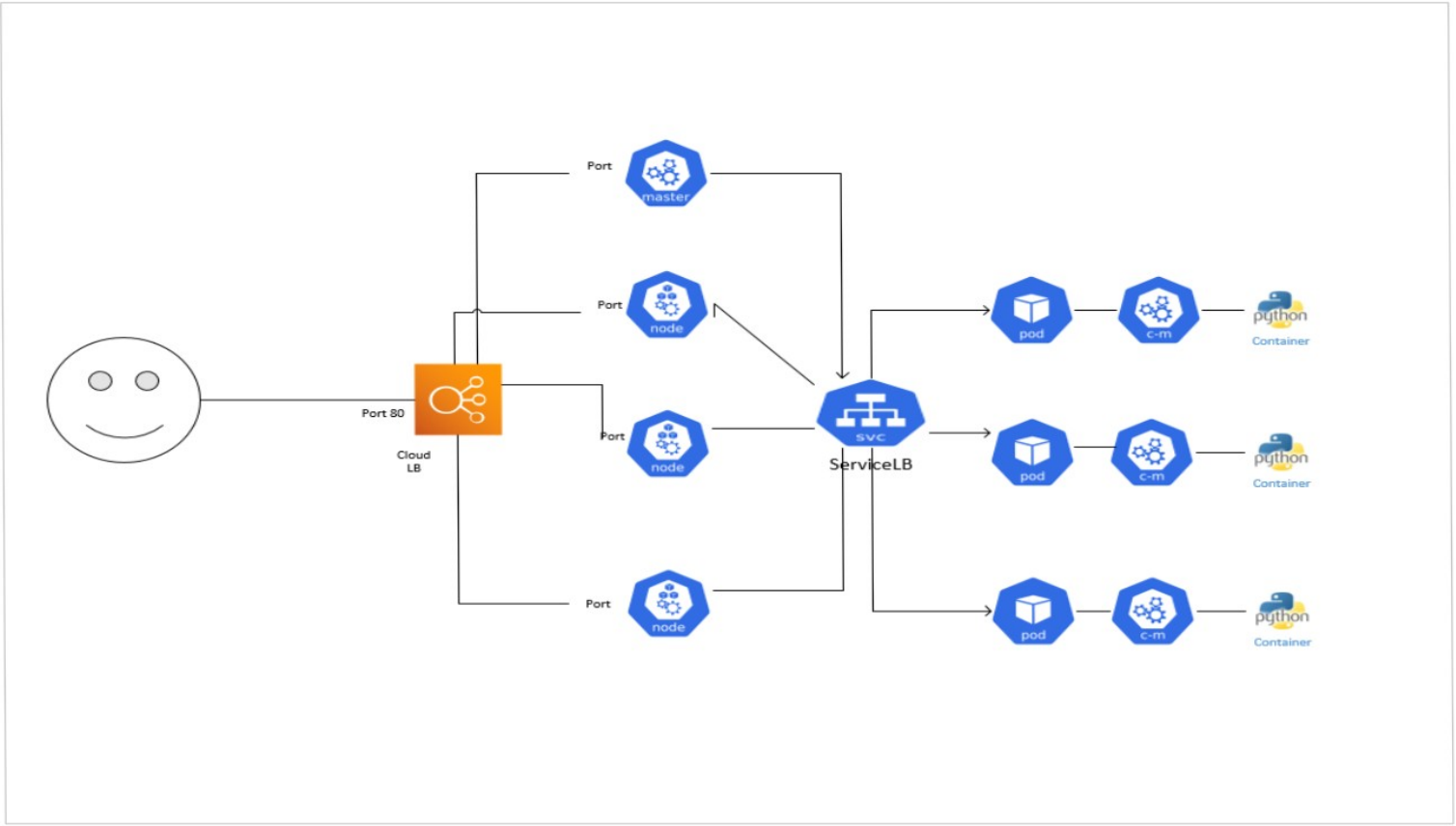
LoadBalancer

- Les Services de type **LoadBalancer** sont utilisés quand le cluster est hébergé chez un « Cloud Provider ».
- Le cluster va demander la création du load balancer en le faisant pointer sur tous les noeuds du cluster sur un port attribué aléatoirement sur chaque noeud (comme pour un NodePort).

  => cher

  => « On premise » si on veut de l'équilibrage de charge il faut « tricher » avec une solution comme metallb.

Load Balancer



Service ExternalName

- Configuration d’un ALIAS vers une IP ou un service extérieur au cluster

  => intégration de service « legacy »

Service Headless

- Le POD client recoit du DNS non pas l'IP d'un service mais une liste d'IP de PODS constituant le service.
- Charge au client de se connecter directement à l'IP du POD de son choix sans passer par une VIP.

Autres solutions que les services

HostPort : c’est le modèle Docker avec l’option « -p ». Chaque nœud K8S a une version du POD (Déployé via un « daemonSet ») et on fait du RoundRobin DNS sur tous les nœuds du cluster.

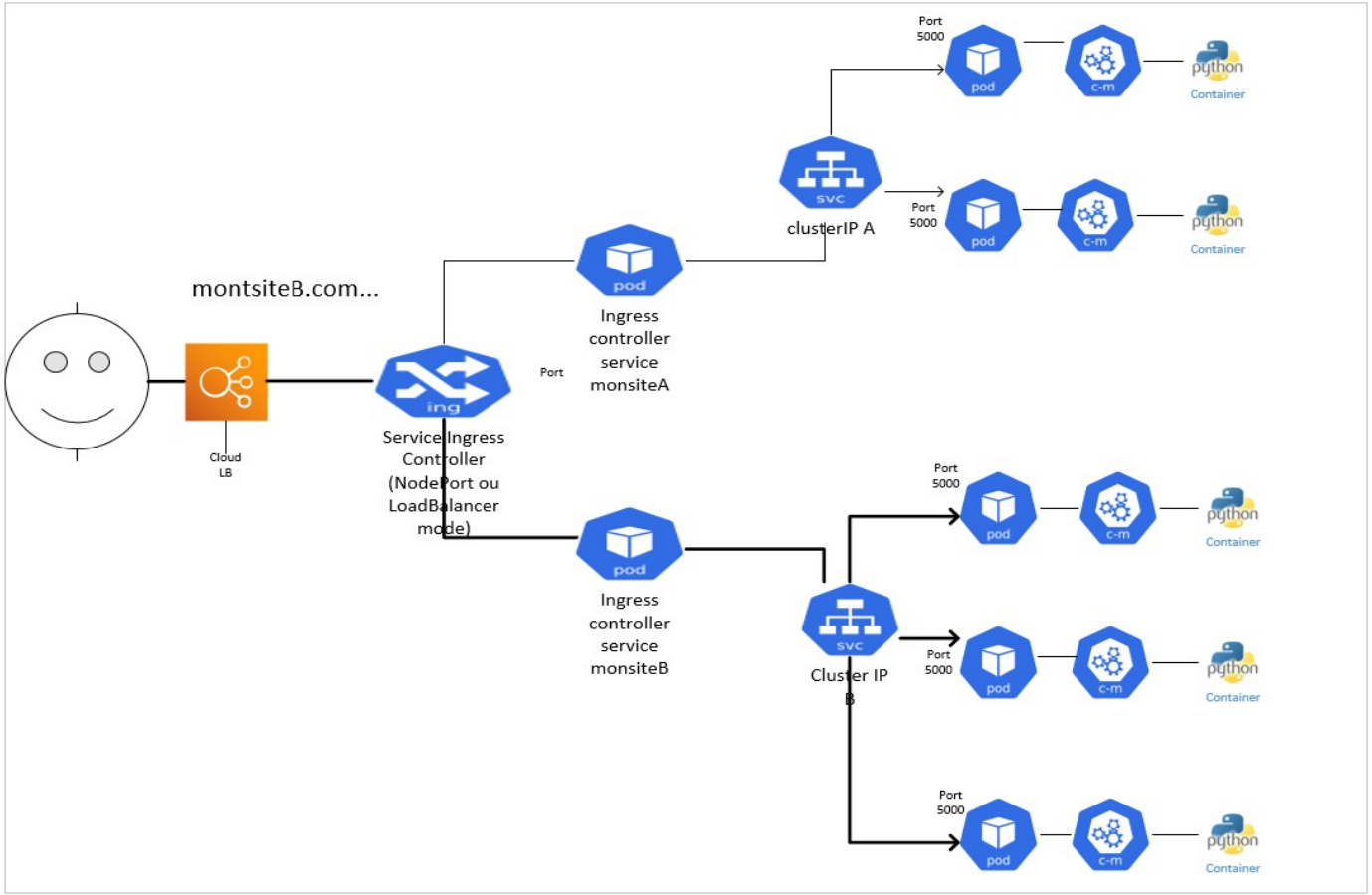
« Ingress Controller »

- Il faut autant de services de type « LoadBalancer » que d'applications hébergées

  => solution chère.

- « L'**ingress controller** » implémente le bon vieux schéma du ***reverse proxy L7***. En fonction d'un header http on dirige le flux vers le bon service « clusterIP »

Ingress Controller



Construire et gérer son cluster

Gérer son cluster

- Kubeadm est l’outil de référence pour créer et upgrader les composants de son cluster
- krew permet d’installer des plugins pour kubectl

Storage avec Kubernetes

Objets stockages

- Persistent Volume, Persitent Volume Claim, Storage class

Autres objets Kubernetes

Configmaps

Cronjob

Infrastructure Kubernetes en HA

Secret