Introduction au DevOps

Kubernetes

Jolan PHILIPPE

April 8, 2026

Université d'Orléans

Les rôles des outils d'IaC

Préparation d'application

Customiser, configurer
tester l'application
et la conteneuriser

Provisionnement d'infrastructure

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

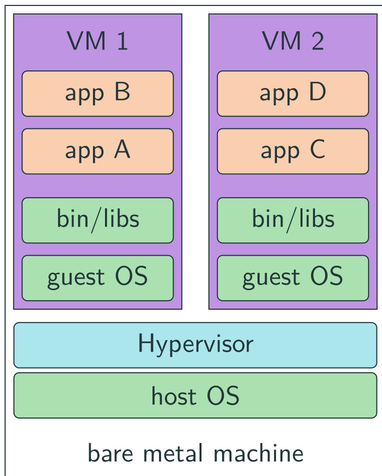
Installation et Configuration

Installer les services
(app + deps)
configurer les services;
et les intégrer

Orchestration de cycle de vie

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

Rappel - Conteneurs



VMs avec un hyperviseur

container 1
app A
bin/libs
minimal OS

container 2
app B
bin/libs
minimal OS

container 3
app C
bin/libs
minimal OS

container 4
app D
bin/libs
minimal OS

container engine

host OS

bare metal machine

Containers

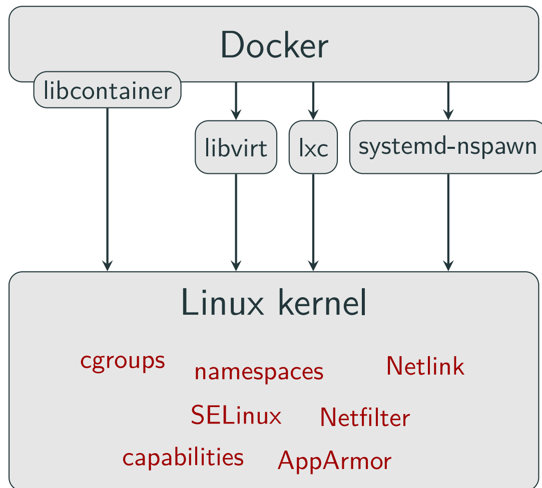
Rappel - Conteneurs

Tirer parti des fonctionnalités du kernel

- `Cgroups`: qu'est ce que mon processus peut consommer ?
- `Namespaces`: qu'est ce que mon processus peut voir ? Et donc faire ?

Et d'autres choses ...

- `Capabilities`: gestion fine des permissions
- `SELinux/AppArmor`: tout contrôler
- `Netlink/Netfilter`: communication breakdown



Kubernetes

Kubernetes

A brief history

- **2014** : Annonce initiale du projet par Google (développé en Go)
- **2017** : Devient le standard *de facto* face à ses concurrents (Docker Swarm, Apache Mesos)
- **2018** : Premier projet de la CNCF (Cloud Native Computing Foundation) à obtenir le statut de maturité maximale ("Graduated")
- **Aujourd'hui** : Adopté massivement et proposé en service managé par tous les géants du Cloud (AWS, Azure, GCP)

Vue d'ensemble de Kubernetes

Les différents éléments de Kubernetes

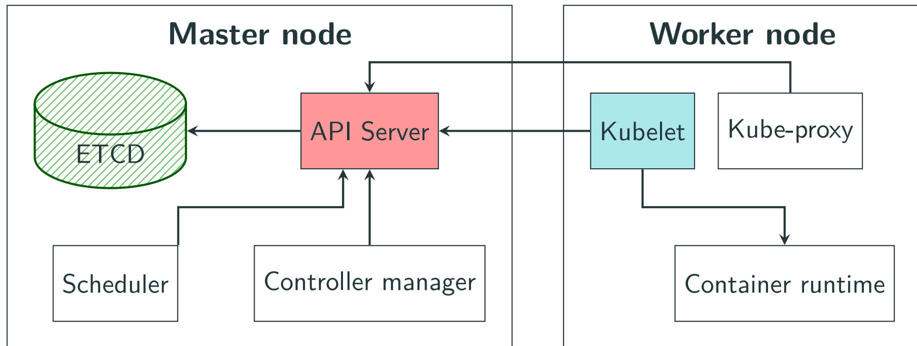
- kubectl (Command Line Interface)
- Le plan de contrôle (Control Plane)
- Les noeuds (Nodes)
- Les pods

Les rôles du Control Plane

- Planifier (*scheduling*) les pods sur les noeuds
- Gérer l'état désiré (*desired state*)
- Superviser (*monitoring*) les déploiements
- Gérer le réseau et les services
- Assurer la scalabilité et la haute disponibilité
- etc.

Vue d'ensemble de Kubernetes

Un cluster Kubernetes est composé de plusieurs noeuds: un noeud ***master***, en charge du *Control plane* et des noeuds ***workers***.



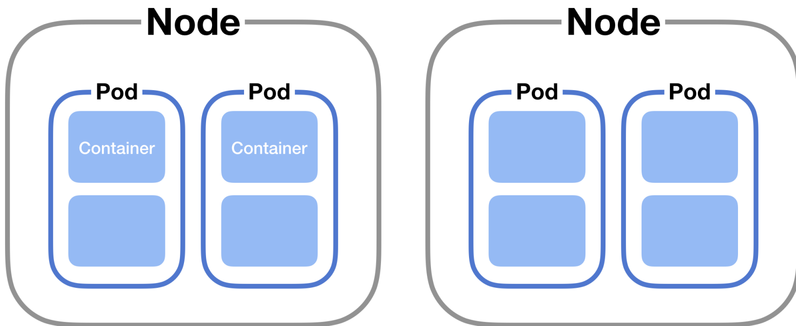
Différents types de noeuds

Master node - Control plane

- Kunernetes API server: Un utilisateur peut communiquer avec cette API via **kubectl**. Les services internes peuvent également communiquer avec cette API.
- Scheduler: planifie les applications sur les workers
- Controller manager: controle toutes les ressources d'un cluster Kubernetes (y compris les ressources intégrées telles que les pods)
- ETCD (`etc/` distribué): système de stockage de données distribué et fiable utilisé par Kubernetes pour stocker l'état des ressources et la configuration du cluster.

Worker nodes

- Container runtime: Docker ou un autre
- Kubelet: communique avec l'API server et gère les conteneurs sur le noeud
- Kube-proxy: répartit la charge du trafic réseau entre les composants de l'application


Cluster
Node
Node
Pod
Pod
Pod
Pod
Container
Container

Les pods dans Kubernetes

La plus petite unité de Kubernetes

- Un Pod contient **un ou plusieurs conteneurs** (comme Docker) fortement couplés.
- Tous les conteneurs d'un même Pod partagent la même adresse IP (ils se parlent sur *localhost*) et le même stockage.
- Les conteneurs d'un Pod sont toujours planifiés et exécutés ensemble sur le même serveur (*Worker node*).
- Les pods sont ephemeres. S'il plante ou si le serveur tombe en panne, Kubernetes le détruit et en recrée un nouveau (nouvelle IP).

Dimensionner un Pod

Pour éviter qu'un Pod ne consomme toutes les ressources d'une machine au détriment des autres, on configure deux seuils (CPU et Mémoire) :

Requests (Le minimum garanti)

- C'est ce dont le Pod a **strictement besoin** pour fonctionner.
- Le *Scheduler* utilise cette valeur pour trouver un Noeud ayant assez de place disponible pour l'accueillir.

Limits (Le plafond maximum)

- C'est la limite **à ne pas dépasser**.
- Protège le Noeud et les autres Pods contre une application qui s'emballe (ex: fuite de mémoire).

Dimensionner un Pod

Que se passe-t-il si un Pod dépasse ses limites ?

- **Pour le CPU** : Le conteneur est “bridé” (*Throttling*). L’application ralentit, mais elle ne plante pas.
- **Pour la Mémoire (RAM)** : Le conteneur est tué et redémarré brutalement (**Erreur OOMKilled** - *Out Of Memory*).

Les règles d’or

- **Ne jamais deviner** : Mesurez d’abord la consommation réelle de l’application en charge avant de fixer vos valeurs.
- **Toujours définir des limites** : Un Pod sans limite est un danger pour l’ensemble du cluster Kubernetes.
- **Unités** : Le CPU se mesure en *millicores* (ex: 500m = 0.5 CPU) et la RAM en Mébioctets/Gibioctets (ex: 256Mi, 1Gi).

Les manifestes

Manifeste Kubernetes

Infrastructure as Code (IaC)

Un manifeste est un fichier YAML qui décrit l'**état désiré** de votre application. Vous donnez ce fichier à Kubernetes, et il se débrouille pour que la réalité corresponde à ce fichier.

Les 4 piliers obligatoires

Tout manifeste Kubernetes valide (sans exception) possède ces 4 champs principaux à sa racine :

1. **`apiVersion`** : La version de l'API Kubernetes à utiliser.
2. **`kind`** : Le type d'objet que l'on veut créer (Pod, Service, etc.).
3. **`metadata`** : Les données pour identifier l'objet.
4. **`spec`** : Les spécifications techniques (le cœur du sujet).

Étape 1 : Quoi et Qui ?

L'en-tête (apiVersion et kind)

On commence par dire à Kubernetes à quelle partie de son cerveau s'adresser, et quel type d'objet on souhaite construire.

```yaml
apiVersion: v1
kind: Pod
```

L'identité (metadata)

Ensuite, on donne un nom unique à notre objet pour pouvoir le retrouver plus tard. On peut aussi lui coller des étiquettes (*labels*).

```yaml
metadata:
  name: mon-premier-pod
  labels:
    environnement: dev
```

Étape 2 : Le comportement (Spec)

Le bloc `spec`

C'est ici que l'on décrit ce que notre ressource doit faire concrètement. Le contenu de ce bloc change complètement selon le `kind` choisi.

Exemple pour un Pod

Pour un Pod, la seule chose vraiment obligatoire est de fournir la liste des conteneurs à lancer (souvent un seul), avec leur nom et l'image Docker à utiliser.

```yaml
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

Étape 2 : Le comportement (Spec)

Le bloc `spec`

C'est ici que l'on décrit ce que notre ressource doit faire concrètement. Le contenu de ce bloc change complètement selon le `kind` choisi.

Exemple pour un Pod

Pour un Pod, la seule chose vraiment obligatoire est de fournir la liste des conteneurs à lancer (souvent un seul), avec leur nom et l'image Docker à utiliser.

```yaml
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

Le manifeste final complet

Le puzzle assemblé (`pod.yaml`)

En collant nos 3 morceaux en respectant l'indentation de 2 espaces, nous obtenons notre manifeste finalisé :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mon-premier-pod
  labels:
    environnement: dev
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

kubectl (CLI de Kubernetes)

Une commande pour les gouverner toutes : kubectl

`kubectl [ACTION] [RESSOURCE] [NOM] [OPTIONS]`

- **[ACTION]** : `get`, `describe`, `create`, `delete`, `apply`
- **[RESSOURCE]** : Le nom de l'objet ciblé. (`pods`, `services`, `deployments`, `nodes`...)
- **[NOM]** : L'identifiant précis (optionnel, sinon l'action s'applique à tous les objets du même type).
- **[OPTIONS]** : (ex: `-n` pour cibler un namespace, `-o yaml` pour changer le format de sortie).

kubectl

Observer et comprendre

- `kubectl get pods` : Liste les pods avec leur statut résumé (e.g., *Running*).
- `kubectl describe pod <nom>` : Affiche l'historique complet d'un pod. Utile pour lire la section Events et debugger
- `kubectl logs <nom>` : Affiche les logs générés par l'application à l'intérieur du conteneur.
- `kubectl exec -it <nom> - /bin/sh` : Ouvre un terminal à l'interieur du pod (l'équivalent d'une connexion SSH).

Run - approche imperative

- `$ kubectl run mon-serveur -image=nginx`: Ordre direct à Kubernetes

Apply - approche déclarative

- `$ kubectl apply -f application.yaml`: Reconcilier avec le manifeste YAML