L'orchestration de conteneurs avec Kubernetes

Virtualisation avancée - CM

Maxime Lambert

2025 / 2026

BUT Informatique - Semestre 5 - Ressource R5.09

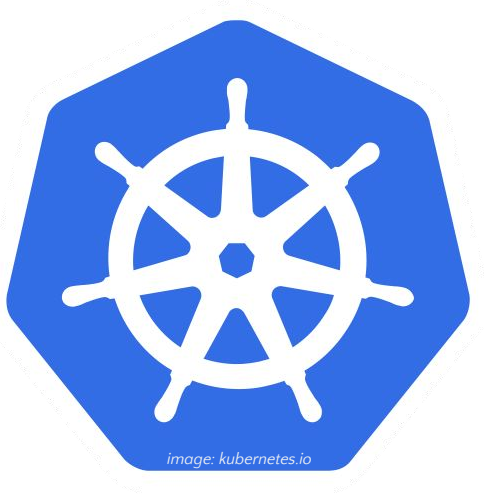
Plan

1. Introduction à Kubernetes
2. Les principes fondamentaux
3. Déploiement d'applications
4. Gestion de la scalabilité
5. Gestion du stockage
6. Défis de l'adoption de Kubernetes

Avant-propos

- Au semestre 4, au cours du module de virtualisation, vous avez étudié les conteneurs sous Linux en pratiquant avec la plateforme Docker.
- Vous avez déployé des conteneurs, réseaux et volumes en utilisant l'interface en ligne de commande de Docker. Vous avez également été initié à l'orchestration grâce à l'outil Docker Compose.
- Bien qu'acceptable dans un contexte de prototypage ou de développement, ces méthodes ne conviennent pas pour exploiter une application complexe dans un environnement de production exigeant.
- Kubernetes est un outil largement adopté par l'industrie qui offre une approche normalisée pour la gestion des applications conteneurisées. C'est aussi un outil open source ce qui aide à éviter de se retrouver dans une situation de verrouillage du fournisseur (*vendor lock-in)*. C'est pour cette raison qu'il a été retenu dans la cadre de ce cours.
- Veuillez noter que nous nous placerons plutôt dans le rôle d'utilisateur de Kubernetes, l'administration d'un cluster de Kubernetes pour satisfaire aux exigences de la production est encore un autre sujet… Et il existe des alternatives que nous détaillerons plus tard.

1. Introduction à Kubernetes



Introduction à Kubernetes

Présentation de la plateforme

- Kubernetes est **une plateforme open-source** extensible et portable pour la gestion de charges de travail et de services conteneurisés.
- Le projet a été **rendu open-source par Google en 2014**, première version stable en 2015
- API standard pour la création d'applications ***cloud native***
  - Vous êtes invité à consulter la méthodologie *The Twelve-factor App*[1] qui dispense plusieurs bonnes pratiques qu'il est important de connaître.
- Permet de construire et déployer des **systèmes distribués fiables et évolutifs**

[1] https://12factor.net/fr

Introduction à Kubernetes

Intégration dans l'écosystème *Cloud*

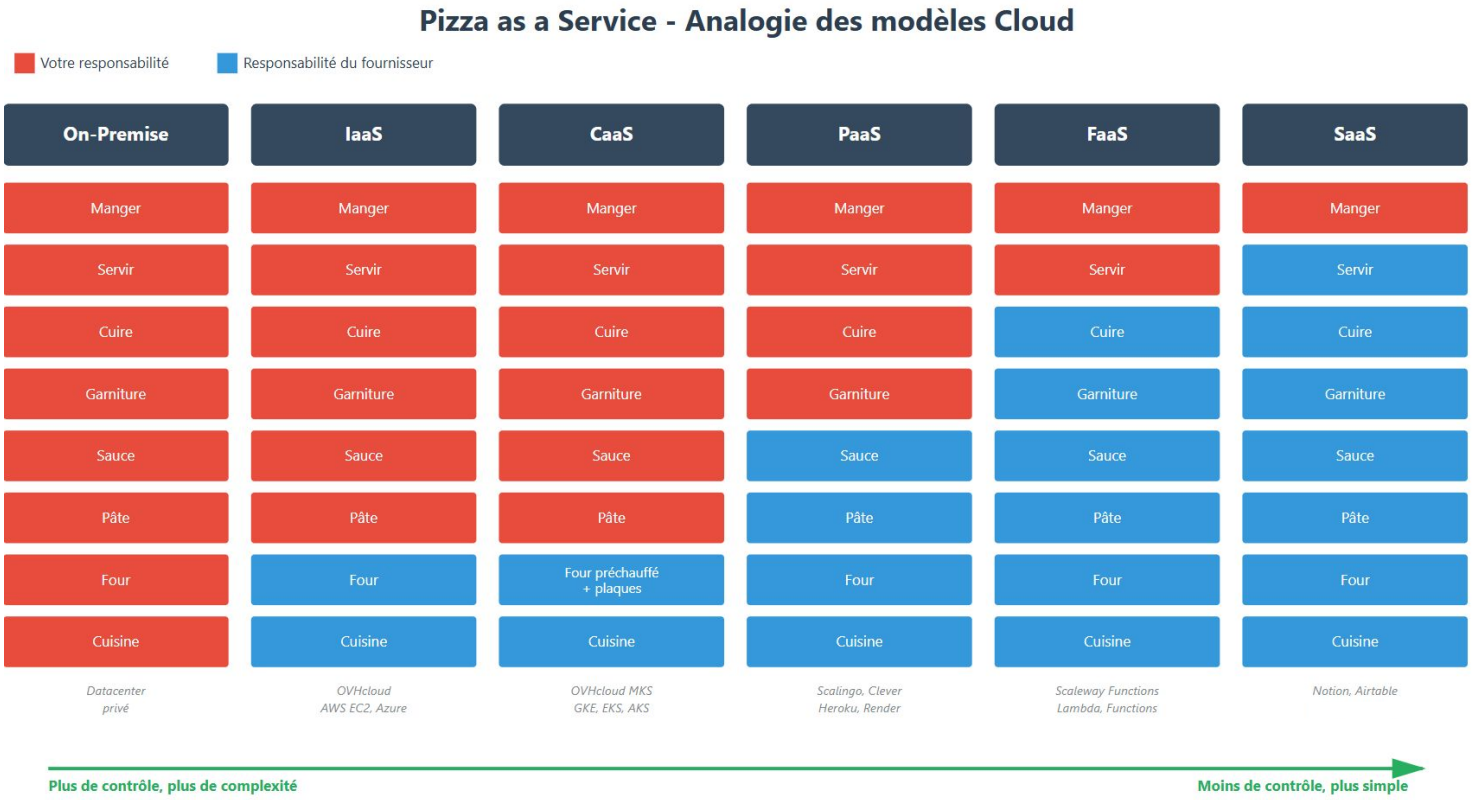
- Les services *Cloud* mettent à disposition à travers le réseau des ressources et fonctionnalités tout **en rendant abstrait les détails de l'infrastructure sous-jacente** pour les utilisateurs.

- Kubernetes fournit un **environnement de gestion focalisé sur le conteneur**. Il **orchestre** les ressources machines, la mise en réseau et l'infrastructure de stockage. On qualifie ce type de plateforme de **CaaS** (*Container as a Service*). Du fait de sa situation de monopole, vous rencontrerez également le terme KaaS (*Kubernetes as a Service*).

- Il est nécessaire de provisionner en amont une infrastructure pour y déployer Kubernetes. Ce qu'il est possible de faire manuellement, de manière automatisée par exemple grâce à Terraform que nous abordé au semestre 4, ou alors vous pouvez déléguer cette tâche à un tiers.

Introduction à Kubernetes

Comprendre le modèle “as a Service”



Introduction à Kubernetes

Comprendre le modèle “as a Service”

- ***On-Premise***: Contrôle totale sur l’infrastructure. Système informatique hébergés, gérés et maintenus directement par l’organisation.
- ***IaaS*** *(Infrastructure as a Service)*: Virtualisation des principaux composants d’infrastructure (calcul, stockage, réseau, sécurité et répartition de charge).
- ***CaaS*** *(Container as a Service)*: Mise à disposition par un fournisseur *cloud* d’un orchestrateur de conteneurs managé. Permet d’être utilisateur de Kubernetes et non administrateur de Kubernetes.

Introduction à Kubernetes

Comprendre le modèle “as a Service”

- ***PaaS*** *(Platform as a Service)*: Téléversement du code chez un fournisseur *cloud* qui disposera de l’outillage nécessaire pour construire et déployer l’application web, tout en proposant des fonctionnalités supplémentaires (bases de données, stockage objet…)

- ***FaaS*** *(Function as a Service)*: Déploiement d’un morceau de code qui réagit à un évènement (par exemple un appel HTTP)

- **SaaS**: Le logiciel n’est pas installé. Il est consommé au travers d’une interface (généralement web). Les données manipulées sont stockées sur les serveurs de l’éditeur.

Introduction à Kubernetes

Comprendre le modèle “as a Service”

Le positionnement dépend de la perspective, voici un exemple avec Heroku.:

- Pour Heroku: IaaS (il semblerait, location de serveurs AWS EC2)
- Pour un développeur: PaaS (git push pour déployer son application)
- Pour l’utilisateur final: SaaS (utilise l’application web sans connaître Heroku)

Chaque modèle dispose de son unité de déploiement:

- On-Prem: machines physiques (serveurs, switches, baies de stockage)
- IaaS: machines virtuelles (Vms avec OS de votre choix)
- Caas: conteneurs (par exemple des images Docker dans des Pods Kubernetes)
- FaaS: fonctions (morceaux de code exécutés à la demande)
- SaaS: configuration (données métier via interface, paramètres, voir du code)

Plus vous montez dans la stack, plus l’unité de déploiement est abstraite. Vous passez du matériel physique tangible à de simples paramètres de configuration. Le time-to-deployment diminue drastiquement : plusieurs semaines pour du matériel physique, quelques secondes pour une fonction serverless.

Introduction à Kubernetes

Comment K8s aide à déployer rapidement et de manière fiable

- **Kubernetes encourage le concept d'immutabilité** en incitant les utilisateurs à créer de nouveaux déploiements au lieu de mettre à jour des conteneurs en cours d'exécution. Cette approche garantit que chaque modification est effectuée à travers une nouvelle version, préservant ainsi la stabilité de l'application. L'immutabilité facilite également le suivi des changements, la gestion des versions et la récupération en cas d'incident.

- **Kubernetes adopte un modèle de configuration déclarative**, où les utilisateurs décrivent l'état souhaité de leur application via des fichiers de configuration YAML. L'objet de configuration détermine la manière dont les ressources, telles que les pods, les services et les volumes, doivent être créées et gérées. Cette approche simplifie la gestion, la reproductibilité et le contrôle des ressources, tout en évitant la configuration manuelle.

- **Kubernetes est conçu pour être auto-guérisseur**, ce qui signifie qu'il surveille en permanence l'état des ressources et prend des mesures pour maintenir la santé de l'application. En cas de défaillance d'un pod ou d'une machine, Kubernetes redémarre automatiquement les pods ou planifie leur réparation sur d'autres nœuds sains. Cette capacité d'auto-guérison contribue à maintenir la disponibilité et la fiabilité des applications dans un environnement dynamique.

Introduction à Kubernetes

Gérer l'évolution d'une organisation avec K8s

- **Kubernetes favorise une architecture découplée en utilisant des API et des équilibreurs de charge** pour isoler chaque composant du système. Les API agissent comme des tampons entre les composants, permettant l'évolutivité des programmes sans ajuster les autres couches du service. Ce découplage simplifie également la gestion et la communication entre les équipes de développement, favorisant ainsi l'évolutivité des équipes.

- **L'évolutivité des applications est simple grâce à la nature immuable et déclarative de K8s.** Sous réserve de bénéficier des ressources disponibles dans le cluster; la modification d'un nombre dans un fichier de configuration suffit pour mettre à l'échelle un service.

- L'évolution des besoins humains peut compliquer l'organisation du travail en équipes. **Maintenir des équipes de petite taille est essentiel pour une communication efficace. L'architecture orientée services (SOA) offre une solution** en découpant le système en composants indépendants, permettant à chaque équipe de se concentrer sur un service spécifique. Cela favorise l'évolutivité des équipes et la flexibilité pour gérer des besoins en constante évolution. Kubernetes offre des abstractions et des API qui facilitent ce type d'architecture..

- Kubernetes offre une cohérence accrue de l'infrastructure en séparant les problèmes et en découplant les conteneurs des machines. Cette séparation permet à une petite équipe de gérer de nombreuses machines de manière efficace. De plus, **K8s crée un contrat clair grâce à son API d'orchestration de conteneurs, délimitant clairement les rôles et responsabilités entre l'opérateur d'applications et l'opérateur d'orchestration de clusters.** Cela simplifie la gestion et l'évolutivité de l'infrastructure.

Introduction à Kubernetes

Une application cloud-native, qu'est-ce que c'est ?

“Une application cloud-native se compose de services plus petits, indépendants et faiblement couplés. Elle est conçue de façon à apporter une valeur métier incontestée, comme la capacité à prendre en compte rapidement l'avis des utilisateurs dans un effort d'amélioration continue. En d'autres termes, le développement d'applications cloud-native permet d'accélérer la création des nouvelles applications, d'optimiser les anciennes et de les connecter les unes aux autres. L'objectif est double : fournir aux utilisateurs les applications dont ils ont besoin tout en suivant le rythme imposé par leur activité.

Quel est le sens du mot « cloud » dans ce contexte ? **Lorsque l'on dit d'une application qu'elle est « native pour le cloud », cela signifie qu'elle a été conçue spécialement pour offrir une expérience cohérente de développement et de gestion automatisée dans les clouds privés, publics et hybrides.** Aujourd'hui, les entreprises adoptent le cloud computing pour améliorer l'évolutivité et la disponibilité de leurs applications, grâce à l'approvisionnement en libre-service et à la demande des ressources ainsi qu'à l'automatisation du cycle de vie des applications (de la phase de développement jusqu'à la production).

Toutefois, pour véritablement bénéficier de ces avantages, elles doivent mettre en place une nouvelle stratégie de développement des applications.

C'est tout l'enjeu du développement d'applications cloud-native : créer et mettre à jour rapidement des applications et, dans le même temps, améliorer la qualité et réduire les risques. Plus précisément, l'approche permet de développer et d'exécuter des applications réactives, évolutives et résistantes aux pannes dans toute architecture, que ce soit un cloud public, privé ou hybride.“

***source: redhat.com**/fr/topics/cloud-native-apps*

Introduction à Kubernetes

Les caractéristiques essentielles d'une application cloud native

Une application cloud native est conçue pour tirer parti des environnements cloud et s'exécuter de manière optimale dans des infrastructures distribuées.

Elle doit disposer à minima de ces caractéristiques :

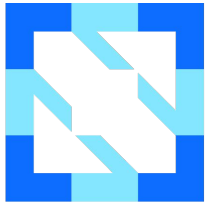
- **Stateless** : sans état local
- **Configuration via variables d'environnement** : aucun paramètre en dur dans le code
- **Conteneurisée** : packagée dans une image avec toutes ses dépendances
- **Résiliente aux pannes** : gère les défaillances (retry automatique, circuit breaker, timeouts, health checks)
- **Observabilité intégrée** : métriques exposées, logs structurés, tracing distribué
- **API-first et découplée** : communication avec contrats versionnés, documentation OpenApi ou similaire
- **Déploiement automatisé** : commit git déclenche CI/CD, IaC, rolling update
- **Scalabilité horizontale** : pas de limite de scaling vertical d'un serveur unique

Introduction à Kubernetes

La Cloud Native Computing Foundation (CNCF)

- **La CNCF est une organisation à but non lucratif qui a pour mission de favoriser l'adoption de technologies cloud natives et de soutenir les projets open source liés à ces technologies.** Elle a été créée pour répondre aux besoins croissants des entreprises en matière de déploiement et de gestion d'applications dans des environnements cloud.

- La CNCF dispose de trois niveaux de maturité pour guider dans l'adoption de projets cloud native :
  - **Bac à sable:** La majorité des projets sont à ce statut. Il indique un projet encore dans un stade de développement précoce et il n'est pas recommandé de l'adopter.
  - **Incubation**: Projet ayant démontré sa viabilité, adopter par la communauté et conforme aux principes cloud natives.
  - **Graduation**: Seuls quelques projets ayant atteint un haut niveau de maturité, stabilité et d'adoption appartiennent à cette catégorie.



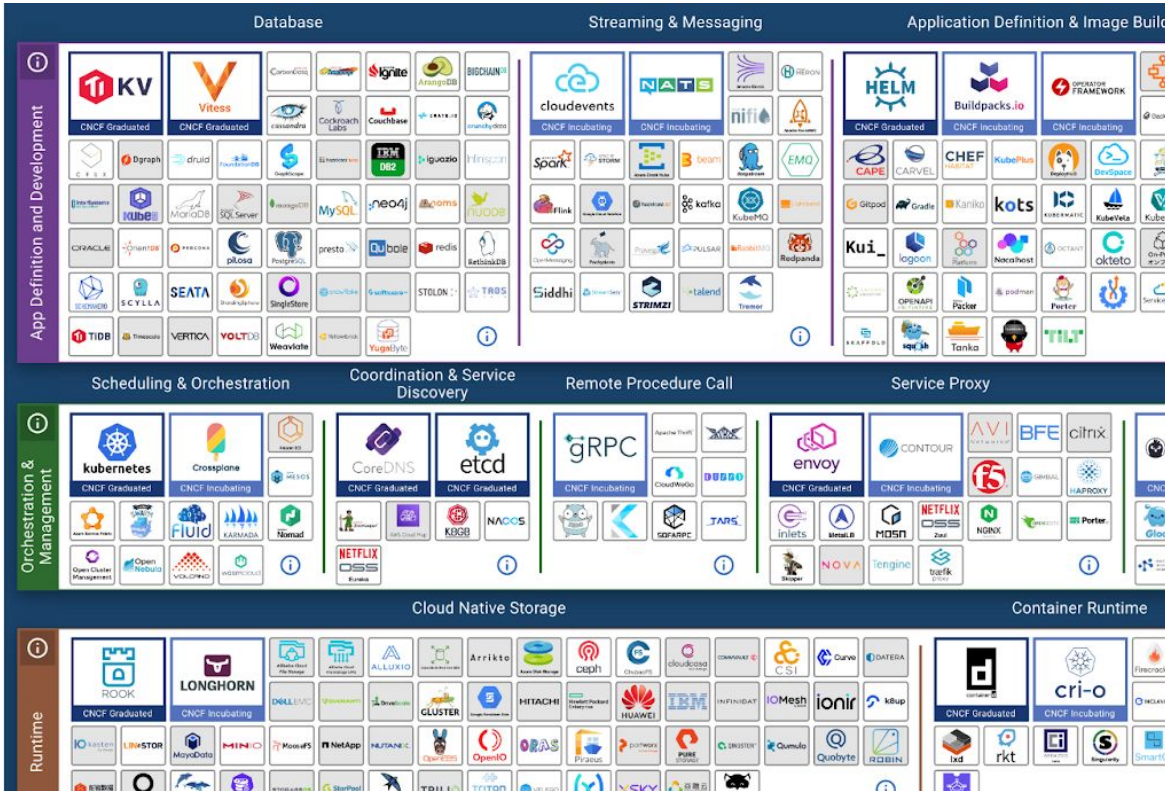
CLOUD NATIVE
COMPUTING FOUNDATION

Introduction à Kubernetes

La CNCF Landscape

- Après la sortie de Kubernetes, de nombreux outils ont été développés pour une variété de tâches, de l'apprentissage automatique aux modèles de programmation sans serveur.

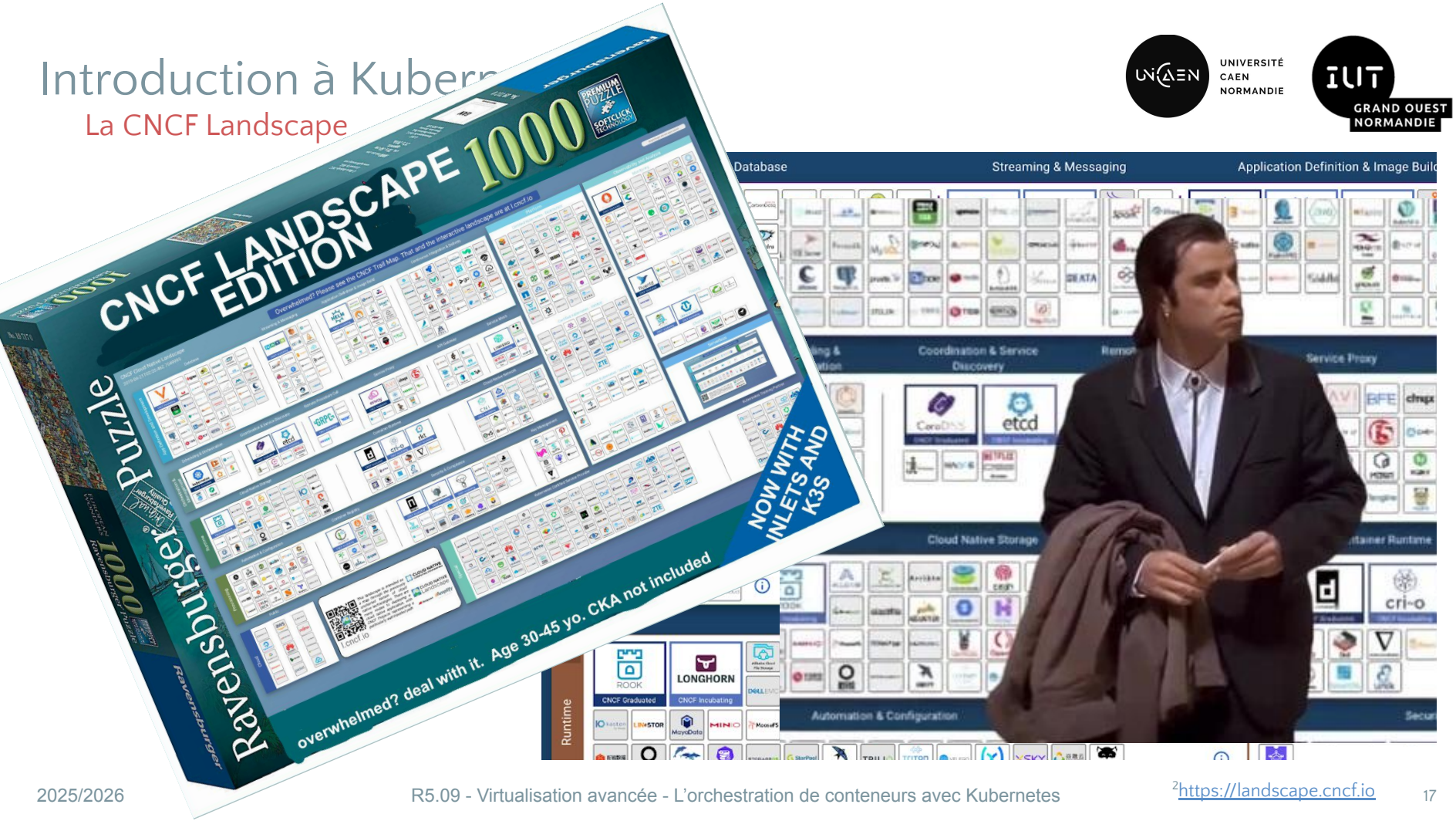
- Cependant, la difficulté réside souvent dans le choix de la meilleure solution parmi les nombreuses disponibles. Pour vous aider, vous avez à votre disposition la CNCF Landscape[2]...



[2] https://landscape.cncf.io

Introduction à Kuberr

La CNCF Landscape



[2] https://landscape.cncf.io

Introduction à Kubernetes

Les outils de base

- **kubectl** : interface en ligne de commandes officiel de Kubernetes. Avec cet outil open source vous pouvez déployer des applications, inspecter et gérer les ressources clusters ou encore afficher les journaux. Il est disponible sous Linux, macOS et Windows.

- **OpenLens** : interface graphique open source relativement conviviale qui facilite la visualisation et la gestion des ressources dans un cluster. Disponible sous Linux, macOS et Windows.

- **kind et minikube** : Ces deux outils permettent la création d'environnement de développement Kubernetes locaux mais présentent quelques différences. La principale est que le premier instancie un cluster à partir de conteneurs Docker alors que le second utilise une machine virtuelle.

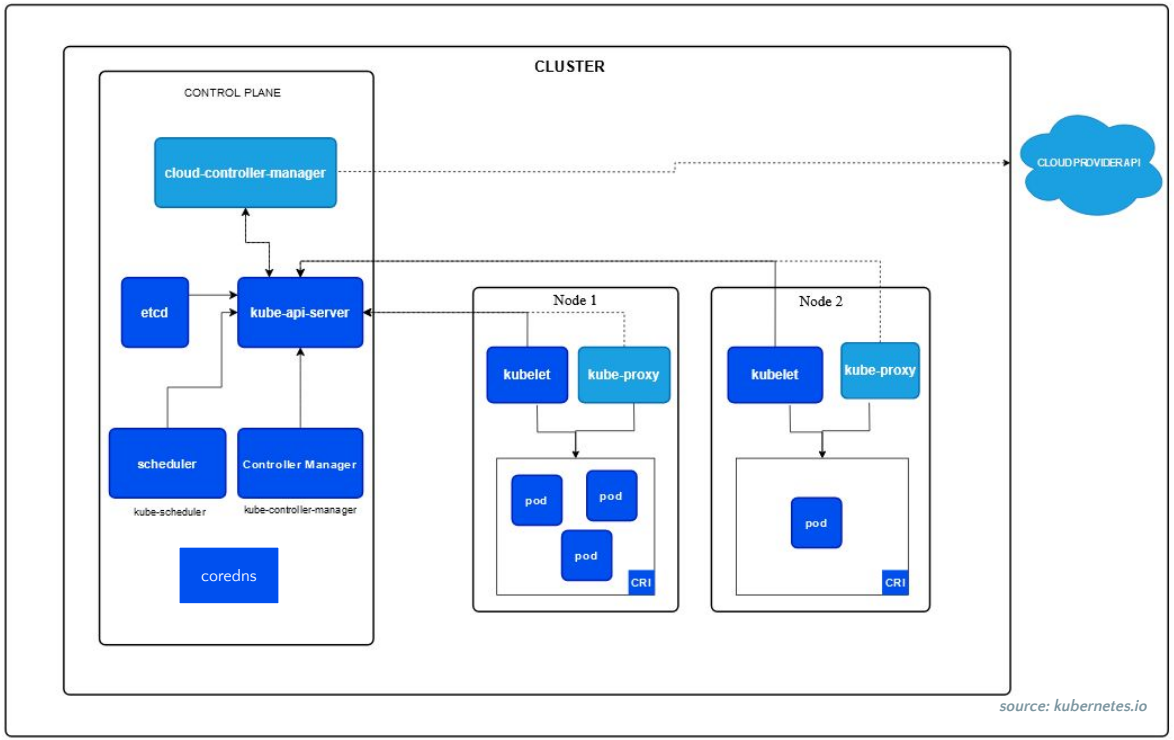
- **kubeadm** : outil destiné à l'initialisation de clusters kubernetes de production. Cet outil n'est donc pas nécessaire dans le cadre de ce module du fait de l'utilisation de kind en TP.

2. Les principes fondamentaux

Les principes fondamentaux

L'architecture d'un cluster

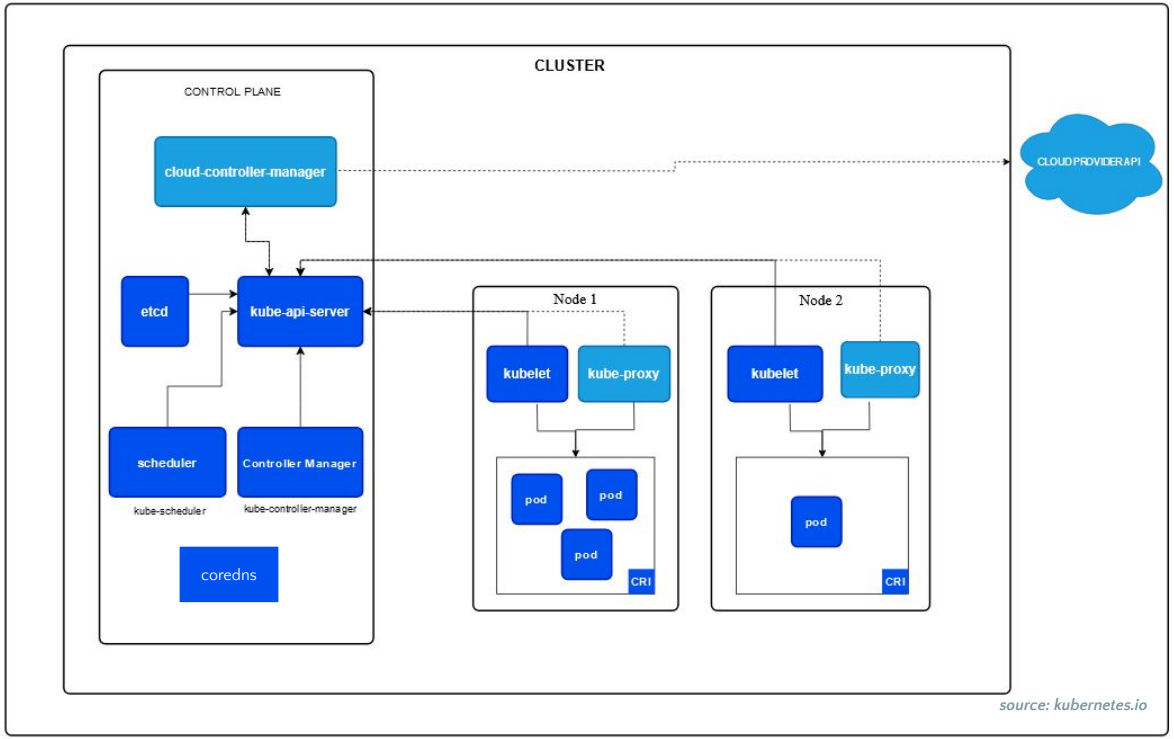
- ***Control plane***: Noeud portant la couche d'orchestration des conteneurs qui expose l'API et les interfaces pour définir, déployer et gérer le cycle de vie des conteneurs.
  - ***API server***: point d'entrée pour les commandes et opérations sur le cluster. Il expose l'API Kubernetes
  - ***Scheduler***: chargé de planifier les pods sur les noeuds disponibles en fonctions des exigences et des contraintes
  - **Etcd**: Magasin de données clé-valeur cohérent et distribué qui stocke les données du cluster
  - ***Controller Manager***: Surveille l'état du cluster et prend des mesures pour garantir l'état demandé



Les principes fondamentaux

L'architecture d'un cluster

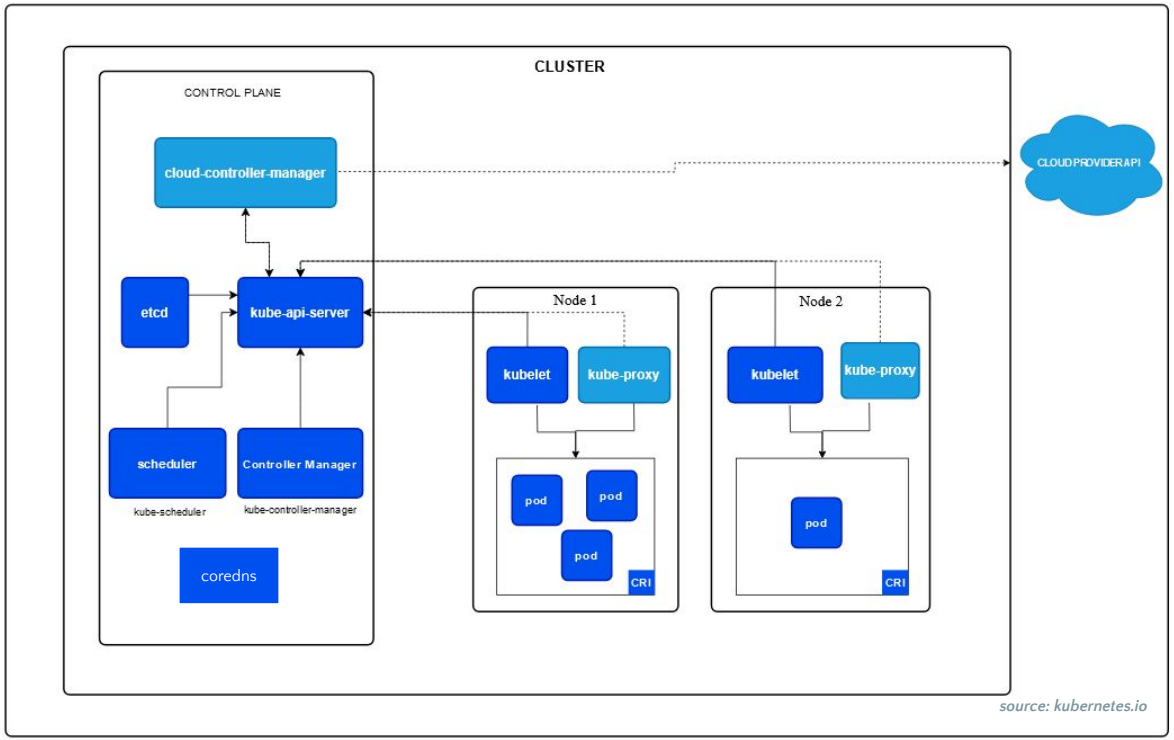
- ***Control plane***:
  - ***Cloud controller manager***: C'est un composant d'extension, distinct du controller manager. Son but est de gérer les ressources cloud spécifiques à un fournisseur de services cloud. Comme par exemple les volumes de stockage type S3 sur AWS
  - **Coredns**: serveur DNS de k8s. Il fournit un nommage et une fonctionnalité de découverte des services dans le cluster. Si vous vous connectez dans un conteneur, vous verrez que l'ip du serveur dns a été injectée dans `/etc/resolv.conf`



Les principes fondamentaux

L'architecture d'un cluster

- ***Node***: Noeud travailleur pouvant être un conteneur, une VM ou une machine physique. Il dispose des démons locaux ou services nécessaires à l'exécution des Pods.
    - ***Kubelet***: Communique avec le *control plane* et gère les conteneurs sur le noeud pour s'assurer qu'ils sont en cours d'exécution
    - ***Kube proxy***: Gère la configuration réseau et la connectivité réseau entre les pods
    - ***Container runtime***: Logiciel responsable de l'exécution des conteneurs implémentant l'interface CRI.



Les principes fondamentaux

Container Runtime Interface (CRI)

- Comment est-il possible d'exécuter des conteneurs Docker (ou autre) sous Kubernetes ?
- Pour rappel, les conteneurs Docker sont conformes à la spécification "OCI Runtime Specification", qui garantie la standardisation du format d'exécution du conteneur et donc l'interchangeabilité des *runtimes*. Podman qui est un outil open source conforme à la spécification OCI peut être un *runtime* de substitution à celui de Docker pour exécuter des conteneurs à partir d'images construites avec Docker.
- La *Container Runtime Interface* est une interface standardisée de plugin qui permet au kubelet d'utiliser différents *runtimes* de conteneurs, sans avoir besoin de recompiler les composants du cluster. Il est nécessaire de disposer d'un exécuteur de conteneur par noeud du cluster afin que le kubelet puisse lancer les pods et leurs conteneurs.
- CRI-O, un *runtime* de conteneurs pour k8s, implémente CRI. Il est open source et conforme aux spécifications de l'OCI. Cela permet à Kubernetes de gérer des conteneurs conformes à OCI.

Les principes fondamentaux

Les espaces de noms (*namespaces*)

- **Les espaces de noms sont un concept qui permet d'organiser des objets dans le cluster. Cela permet de diviser un cluster Kubernetes en plusieurs espaces logiques et isolés.**
- Généralement on isole les ressources de manière à les rendre inaccessible entre plusieurs espaces de noms.
- Les espaces de noms facilitent également la gestion des ressources au sein du cluster. Ils permettent de regrouper les ressources liées à une application ou à un service particulier dans un espace de noms spécifique. Cela simplifie la gestion, la surveillance, et la maintenance, car les administrateurs et les développeurs peuvent se concentrer sur un espace de noms à la fois, sans avoir à s'inquiéter des autres parties du cluster.
- Les espaces de noms sont particulièrement utiles dans les environnements multi-utilisateurs, où plusieurs équipes ou utilisateurs partagent un même cluster Kubernetes.
- Attention à ne pas confondre les espaces de noms Linux et Kubernetes.

Les principes fondamentaux

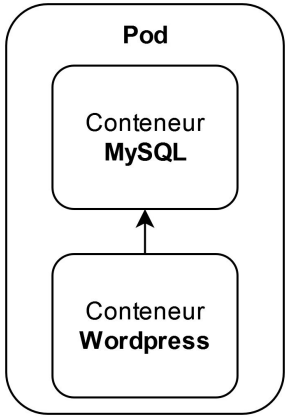
Les pods

- **Un pod est un ensemble atomique de conteneurs et volumes fonctionnant dans le même environnement d'exécution.**
- **Il s'agit de la plus petite unité déployable dans Kubernetes.** Les conteneurs d'un même pod partagent plusieurs espaces de noms Linux. Concrètement, ils partagent le même espace réseau, le même espace de stockage et les mêmes spécifications de ressources (RAM/CPU). Toutefois, ils s'exécutent dans leur propre cgroup.
- Il n'est pas possible de diviser un pod en parties plus petites et tous les conteneurs d'un pod sont situés sur le même noeud.

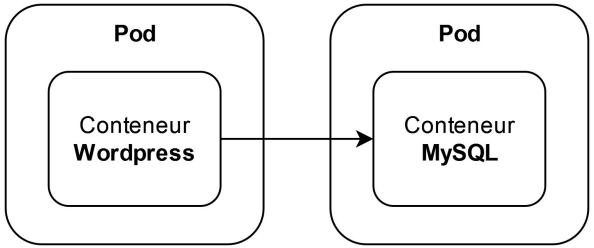
Les principes fondamentaux

Que mettre dans un pod ?

- Admettons que vous ayez deux conteneurs à déployer. Un pour site Wordpress et un second pour une base de données MySQL. Le conteneur Wordpress doit stocker les informations de son site dans le conteneur MySQL.
- Quelle solution choisiriez-vous entre les deux propositions suivantes ?



Solution A

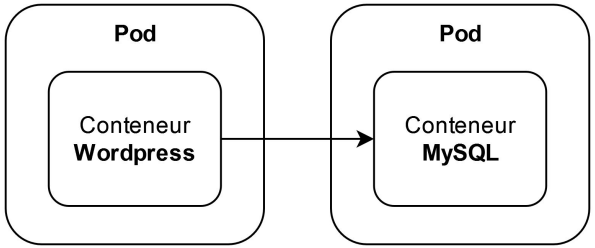


Solution B

Les principes fondamentaux

Que mettre dans un pod ?

- Lors de la conception des Pods, il est essentiel de se poser la question suivante : est-ce que les conteneurs peuvent opérer (de manière efficace et fiable) s'ils sont exécutés sur des machines distinctes ?
- Si la réponse est non, alors l'utilisation d'un seul Pod pour regrouper ces conteneurs est la solution appropriée.
- Cependant, si la réponse est oui, il serait plus judicieux d'envisager l'utilisation de plusieurs Pods.
- Dans notre exemple:
  - La base MySQL n'est pas fortement couplée au site Wordpress. En réalité, nous pourrions avoir plusieurs services consommant notre base de données.
  - Les deux conteneurs communiquent par le biais d'une interface réseau, ce n'est pas un obstacle qu'ils se trouvent sur deux machines différentes.
  - Notre site Wordpress est une application sans état, en cas de pics de charge, il est plus facile de créer plusieurs réplicats d'un Pod Wordpress que d'augmenter les ressources CPU/RAM d'un unique pod Wordpress/MySQL (le dimensionnement d'une base de données est plus délicat).



Solution B

Les principes fondamentaux

Les services

- **Un service dans Kubernetes est une ressource qui permet de fournir une abstraction réseau stable et constante pour les Pods, quel que soit l'endroit où ils sont déployés dans un cluster**.

- **Un équilibrage de charge interne est assuré par le service. Cela permet à un client de s'adresser à un groupe de Pods en utilisant un point d'entrée unique (l'ip du service) au lieu de pointer individuellement vers chaque Pod (l'ip du pod).** Cela facilite grandement la gestion des applications, car les Pods peuvent être créés et détruits dynamiquement sans perturber la communication des clients avec l'application.

- Les Services peuvent être de plusieurs types :
    - **ClusterIP**: service accessible uniquement à l'intérieur du cluster. Il est généralement utilisé pour exposer les services back-end ou des bases de données
    - **NodePort**: service exposant un port sur tous les nœuds du cluster
    - **LoadBalancer**: service utilisant un équilibreur de charge externe
    - **ExternalName**: service redirigeant vers un nom de domaine externe

Les principes fondamentaux

L'ingress

- **L'ingress est une ressource Kubernetes qui définit les règles d'acheminement du trafic HTTP et HTTPS entrant vers des services. Pour qu'il fonctionne, un contrôleur d'Ingress doit être déployé dans le cluster** comme par exemple Nginx Ingress Controller. Un contrôleur agit comme un gestionnaire de trafic qui interprète les règles Ingress et dirigent le trafic vers les services en conséquence.

- **L'ingress permet un routage basé sur des critères tels que les noms de domaine, les chemins d'URL, ou d'autres en-têtes HTTP.** Cela signifie que vous pouvez configurer des règles pour diriger le trafic vers des services spécifiques en fonction de l'URL demandée.

- Par exemple, vous pouvez rediriger le trafic vers un service de site Web, un service d'API ou d'autres applications en fonction de l'URL ou du nom de domaine. Cette flexibilité permet de gérer efficacement l'accès à plusieurs services à partir d'une seule passerelle d'entrée.

Les principes fondamentaux

Les étiquettes (*labels*)

- **Les étiquettes sont des métadonnées clés-valeurs que vous pouvez attacher à des objets Kubernetes tels que des pods, des services… Elles permettent d'associer des informations spécifiques à une ressource, facilitant ainsi la catégorisation, l'organisation et la recherche des ressources au sein d'un cluster.** C'est un élément omniprésent dans Kubernetes.

- Ces étiquettes permettent aux utilisateurs de faire correspondre leurs propres structures organisationnelles aux objets du système de manière souple, sans que les clients n'aient à stocker ces correspondances.

- **Dans un cluster k8s, il n'y a pas de notion de hiérarchie entre les composants.** Mais les objets doivent être reliés entre eux, et ces relations sont déterminés par les étiquettes et sélecteurs d'étiquettes.

- Les sélecteurs d'étiquettes permettent à un objet de faire référence à un ensemble d'autres objets Kubernetes.

- Plusieurs contraintes de nommage s'applique sur les étiquettes, vous les retrouverez dans la documentation:
  - https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Les principes fondamentaux

Les annotations

- **Les annotations Kubernetes sont utilisées pour attacher des métadonnées arbitraires et non identifiantes aux objets.** Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.
- **A la différence des étiquettes, il n'est pas possible de sélectionner des objets Kubernetes à l'aide d'une annotation.**
- Si vous avez un doute entre annotation ou étiquette, il est conseillé de créer une annotation. Si vous avez besoin de requêter votre objet à partir de cette méta-donnée, il suffira de modifier son type en étiquette.
- Quelques exemples d'utilisation:
    - Suivre l'état d'un déploiement (il s'agit d'une ressource Kubernetes que nous détaillerons plus tard)
    - Indiquer une information sur le build, la *release*, ou l'image
- Les mêmes contraintes de nommage que les étiquettes s'appliquent sur une annotation.

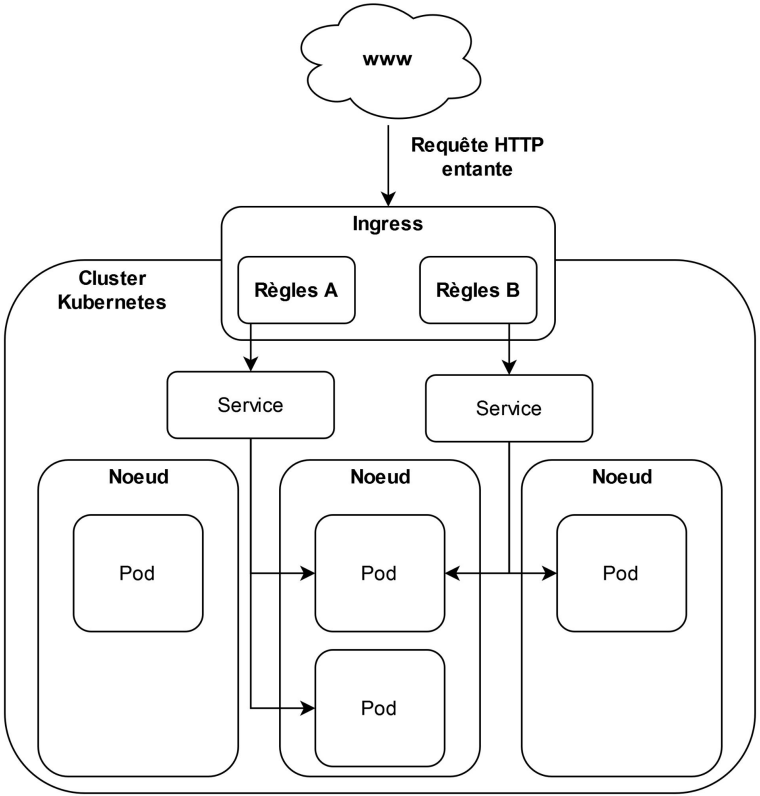
Les principes fondamentaux

Synthèse de la relation entre l'ingress, les services et les pods

- **Pods** : Les pods sont les unités de base de déploiement dans Kubernetes. Ils contiennent une ou plusieurs applications et sont les plus petits composants déployables. Les pods sont souvent exposés via des services pour permettre l'accès externe à leurs applications.

- **Services** : Les services Kubernetes fournissent une abstraction pour accéder aux pods. Ils définissent un point d'accès réseau stable pour un groupe de pods en utilisant des sélecteurs.

- **Ingress** : L'ingress est un contrôleur de gestion du trafic entrant qui permet de configurer des règles d'acheminement pour le trafic HTTP et HTTPS. Il agit comme une passerelle d'entrée pour plusieurs services, en fonction de critères tels que les noms de domaine ou les chemins d'URL. L'ingress dirige le trafic vers les services appropriés, qui, à leur tour, redirigent le trafic vers les pods correspondants. Ainsi, l'ingress permet de gérer de manière centralisée l'accès externe à plusieurs services Kubernetes.



Les principes fondamentaux

Les configMaps

- **Le configMap est une ressource Kubernetes qui permet de stocker des données de configuration sous forme de paires clé-valeur.**

- **Il est utilisé pour stocker des données de configuration qui sont nécessaires aux applications déployées dans un cluster Kubernetes.** Cela permet de séparer les données de configuration de l'application elle-même, ce qui rend l'application plus portable et plus facile à gérer.

- Les configMaps sont souvent utilisés pour stocker des paramètres de configuration tels que les variables d'environnement, les fichiers de configuration, les URL, les adresses IP, etc.

- Une fois qu'un configMap est créé, il peut être référencé dans les spécifications des pods Kubernetes. Les pods peuvent utiliser ces références pour injecter les données de configuration stockées dans le configMap directement dans leurs conteneurs. Cela permet aux applications de récupérer dynamiquement les données de configuration dont elles ont besoin à partir du configMap.

Les principes fondamentaux

Les secrets

- Le secret est une ressource Kubernetes qui permet de stocker des données sensibles sous forme de paires clé-valeur. Il est relativement similaire au ConfigMap qui concerne, lui, les données de configuration.

- Si vous devez stocker une clé d'authentification, un mot de passe, un certificat ou une autre information confidentielle, vous devez passer par un secret et non par un ConfigMap !

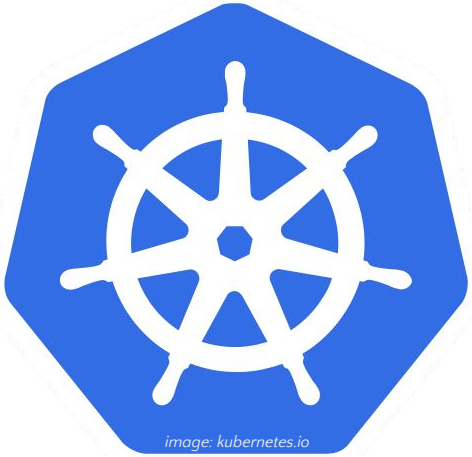
- **Attention :** Par défaut, les données dans un secret sont seulement **encodées en base64**, pas chiffrées. L'utilisation de secrets permet principalement de séparer les données sensibles de la configuration applicative et d'appliquer des politiques d'accès spécifiques via RBAC.

- Les secrets peuvent être référencés dans les spécifications des pods Kubernetes, ce qui permet aux applications d'accéder aux données sensibles. Par exemple, un pod peut référencer un secret pour obtenir un mot de passe de base de données.

- Pour bénéficier d'un vrai chiffrement, il faut activer l'**Encryption at Rest** (dans etcd). Le secret transite en TLS entre l'API server et les nœuds, mais il reste en clair et monté en tmpfs sur le nœud. Attention si le nœud est compromis.

- Solutions plus poussées : Vault, Sealed Secrets, External Secrets Operator.

3. Déploiement d'applications



Déploiement d'applications

Le manifeste de Pod

- **Un manifeste de Pod décrit la configuration d'un seul pod.**
- Le format du fichier peut-être le JSON ou YAML.
- Il est utilisé pour définir les caractéristiques spécifiques d'un pod, telles que :
  - les conteneurs
  - les volumes
  - les variables d'environnement…
- Les pods créés à partir de manifestes individuels n'offrent pas de mécanisme de mise à l'échelle automatique ni de gestion avancée des mises à jour.
- Vous utiliserez les commandes `kubectl` pour charger ce manifeste sur Kubernetes.

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod
spec:
  containers:
    - name: conteneur-1
    image: mon-image:latest
    ports:
    - containerPort: 80
  restartPolicy: Always
```

Déploiement d'applications

Le déploiement (*Deployment*)

- Un déploiement permet de gérer la création, la mise à l'échelle et la mise à jour de Pods.
- Il utilise un modèle de Pod comme base, mais il encapsule la gestion des répliques, de la mise à jour et du contrôle de l'état.
- Ils sont conçus pour gérer des applications dont les Pods doivent être répliqués pour une haute disponibilité.
- Ils permettent également de réaliser des mises à jour et des rollbacks des applications de manière contrôlée.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mon-app
  template:
    metadata:
      labels:
        app: mon-app
    spec:
      containers:
        - name: conteneur-1
          image: mon-image:latest
          ports:
            - containerPort: 80
```

Déploiement d'applications

Exposition dans le cluster d'un déploiement

- Vous trouverez ci-contre un manifeste de service nommé "mon-service", qui expose à l'intérieur du cluster les pods du déploiement précédent.
- Notez bien que nous référençons le déploiement par l' étiquette "app" ayant la valeur "mon-app".
- Dans cet exemple, le service écoute sur le port 80 et redirige le trafic vers le port 80 du pod.

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

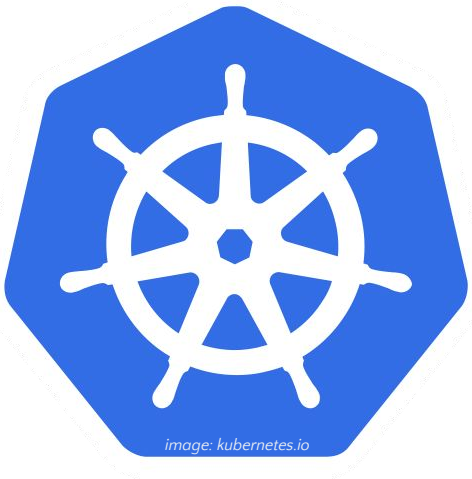
Déploiement d'applications

Exposition d'une application en dehors du cluster

- Vous trouverez ci-contre un manifeste d'ingress nommé "mon-ingress".
- Elle expose notre application en dehors du cluster au travers d'une règle pour le domaine "`mon-app.example.com`".
- Toutes les requêtes HTTP reçues sur ce domaine sont acheminées sur le port 80 de notre service, et donc transmis à un des pods exécutant notre application dans un conteneur.
- Notez qu'il est nécessaire d'avoir déployé en amont un ingress controller dans le cluster pour que cette règle soit applicable.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: mon-ingress
spec:
  rules:
    - host: mon-app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: mon-service
                port:
                  number: 80
```

4. Gestion de la scalabilité



Gestion de la scalabilité

The Great Escape

- Pouvez-vous indiquer ce que fait ce morceau de code ?
- Quel est le risque si l'on exécutait ce code dans un pod ?

```java
import java.util.ArrayList;
import java.util.List;

public class MemoryLeakExample {
    public static void main(String[] args) {
        List<byte[]> memoryList = new ArrayList<>();

        while (true) {
             // Alloue 1 Mo de mémoire
            byte[] memoryChunk = new byte[1024 * 1024];
            memoryList.add(memoryChunk);

            try {
                Thread.sleep(100); // Pause de 100 millisecondes
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Gestion de la scalabilité

La gestion des ressources CPU et mémoire

- **La gestion des ressources CPU et mémoire est un aspect à ne pas négliger dans Kubernetes afin d'éviter de faire s'effondrer l'ensemble d'un cluster.**
- Au sein d'un conteneur, il est possible d'appliquer des limites. Par exemple, pour une application java, vous vous pouvez indiquer une taille maximale pour le tas. Si cette dernière est dépassée, votre application sera tuée.
- Toutefois, rien n'empêche un autre processus de consommer énormément de ressources CPU ou mémoire. La manière la plus saine est de venir appliquer des quotas de ressources sur les objets Kubernetes.
- **Vous pouvez dans le manifeste d'un pod ou dans un déploiement, appliquer des requêtes et limites de ressources pour un pod.** Cela signifie que ces limites s'appliqueront à l'ensemble des conteneurs sur le pod.
- Deux notions sont importantes dans le contexte de gestion des ressources :
    - **Les requêtes** : Il s'agit de la quantité minimale de CPU et mémoire dont un pod a besoin pour s'exécuter correctement. Cela permet de gérer la planification du pod sur un noeud disposant des ressources nécessaires.
    - **Les limites** : Lorsqu'un pod dépasse les limites affectées, il peut subir un arrêt forcé ou une restriction sur l'utilisation des ressources. Ce phénomène peut avoir un impact sur les performances de l'application et la rendre instable.

Gestion de la scalabilité

Placer des requêtes et limites de ressources sur un pod

- L'exemple ci-contre, indique comment nous pouvons placer dans un manifeste des requêtes et limites de ressources sur un pod.
- Notez que vous devrez analyser le fonctionnement de votre application pour affiner ces valeurs. Nous vous présenterons ultérieurement des outils répondant à ce besoin.
- **Par défaut, il n'y a pas de requête ou limite sur pod. C'est-à-dire qu'il va consommer autant de ressources qu'il lui est nécessaire.**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-app-image:latest
      resources:
        requests:
          memory: "256Mi"  # Exemple de requête de mémoire
          cpu: "100m"      # Exemple de requête de CPU (100
milli-CPU)
        limits:
          memory: "512Mi"  # Exemple de limite de mémoire
          cpu: "250m"      # Exemple de limite de CPU (250
milli-CPU)
```

Gestion de la scalabilité

Placer des requêtes et limites de ressources sur un namespace

- Vous pouvez également placer des requêtes et limites de ressources sur un espace de noms Kubernetes grâce à l'objet "ResourceQuota".
- Il est possible d'utiliser cette mesure conjointement avec celles sur les pods. C'est même une méthode recommandée.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    requests.cpu: "2"          # Limite de demande CPU
    requests.memory: 4Gi       # Limite de demande mémoire
    limits.cpu: "4"            # Limite de limite CPU
    limits.memory: 8Gi         # Limite de limite mémoire
```

Gestion de la scalabilité

Minimiser l'interruption de service

- **Lorsque vous effectuez la mise à jour d'un des services de votre application, vous devez chercher à éviter le plus possible l'interruption de service.**
- Kubernetes propose trois mécanismes qu'il est possible d'utiliser lors de la mise à jour d'un de vos déploiements:
  - **Mise à jour incrémentielle (*rolling update*)** : Comportement par défaut. Création d'un nouveau pod avec la nouvelle version, puis suppression de l'ancien. Remplacement des pods un par un.
  - **Re-création (*recreate*)** : possible interruption de service car l'on supprime tous les pods de l'ancienne version avant de créer les pods avec la nouvelle version.
  - **Bleu-vert (blue-green)** : Deux environnements coexistent. Le bleu est l'environnement en cours d'exploitation, l'environnement vert contient la nouvelle version. La mise à jour s'effectue en redirigeant tout le trafic vers le nouvel environnement.

Gestion de la scalabilité

Comment savoir qu'un service est prêt à traiter des requêtes ?

- Pouvez-vous indiquer ce que fait ce morceau de code ?
- Admettons que nous encapsulons ce code dans un conteneur et que nous le déployons dans un pod sur notre cluster.
- Si nous avons choisi une stratégie de mise à jour *Rolling update*. Comment savoir que mon nouveau pod est prêt à traiter les requêtes ?

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MySpringBootApplication {

    public static void main(String[] args) {
        SpringApplication.run(MySpringBootApplication.class, args);
    }

    @Bean
    public CommandLineRunner delayBeforeStartup() {
        return args -> {
            Thread.sleep(60000); // Délai de 1 minute (en millisecondes)
            System.out.println("Le délai est écoulé. Démarrage du
serveur...");
        };
    }
}
```

Gestion de la scalabilité

Les principales sondes

- Il existe **deux types de sondes** jouant un rôle clé dans la scalabilité des pods de Kubernetes :
    - **Sonde de vivacité (*liveness probe*)** : Permet de vérifier si un pod est en cours d'exécution et fonctionne correctement. Si un pod ne répond pas, K8s le considère comme non fonctionnel et va tenter de le redémarrer.
    - **Sonde de disponibilité (*readiness probe*)** : Indique si un pod est prêt à servir le trafic. Si il ne l'est pas, il ne figurera pas dans l'équilibrage de charge du service. Ce qui signifie qu'il ne recevra pas de requêtes.
- **En combinant ces deux types de sonde, cela permet d'optimiser la scalabilité d'un cluster en s'assurant que seuls les pods prêts (*ready*) et en bon état (*healthy*) répondent au trafic.**

Gestion de la scalabilité

A quoi ressemble les sondes dans un déploiement ?

- Ici vous avez un exemple de sonde pour un serveur java spring-boot classique.
- La plupart des sondes utilise le protocole HTTP. Mais il est possible d'utiliser TCP ou d'exécuter une commande arbitraire à l'intérieur du conteneur.

```
… // Code omis par soucis de lisibilité
spec:
      containers:
        - name: conteneur-1
          image: mon-image:latest
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
              path: /actuator/health/readiness
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
          livenessProbe:
            httpGet:
              path: /actuator/health/liveness
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 15
```

https://spring.io/blog/2020/03/25/liveness-and-readiness-probes-with-spring-boot

Gestion de la scalabilité

Les *statefulsets*

- Jusqu'à maintenant nous avons essentiellement parlé du **déploiement qui est un objet kubernetes conçu pour gérer des applications sans état.**

- **Certaines applications nécessite une certaine stabilité, un ordonnancement et parfois une identité unique sur le réseau et/ou du stockage persistant. Le *statefulset* est l'objet kubernetes dédié à ces besoins:**
    - **Identité unique** : Chaque pod géré par un *statefulSet* reçoit un nom unique et persistant, tel que "web-0", "web-1", "web-2", etc. Cela garantit que les pods conservent leur identité même en cas de redémarrage ou de mise à jour.
    - **Ordonnancement prévisible** : Les pods sont déployés et démarrés dans un ordre spécifique, ce qui garantit que chaque pod attend que son prédécesseur soit prêt avant de commencer. Cela est essentiel pour les applications qui dépendent de la cohérence des données entre les pods.
    - **Stockage persistant** : Les *statefulSets* sont souvent utilisés pour des applications qui nécessitent un stockage persistant, comme des bases de données. Chaque pod peut être associé à un volume persistant, ce qui garantit que les données sont conservées même en cas de redémarrage ou de remplacement du pod.
    - **Service Headless** : Par défaut, un service *headless* est créé avec un *statefulSet*. Ce type de service permet de découvrir et de communiquer avec les pods individuels du *statefulSet* à l'aide de leur nom de réseau.

Gestion de la scalabilité

Le concept de Horizontal Pod Autoscaling (HPA)

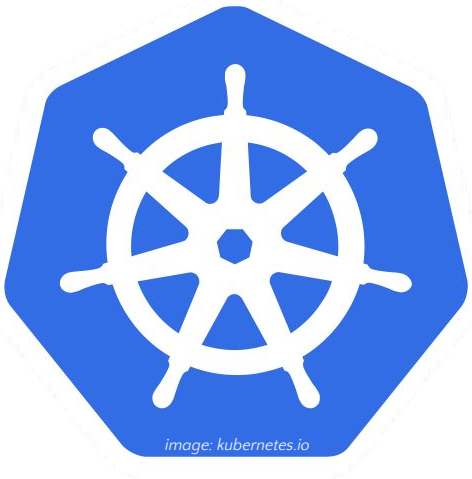
- **Le fonctionnement de mise à l'échelle horizontale des noeuds repose sur des métriques telles que l'utilisation du CPU ou de la mémoire, qui sont collectées sur les pods d'un déploiement.**
- Lorsque les métriques dépassent ou descendent en dessous d'un seuil configuré, l'HPA peut automatiquement réduire ou augmenter le nombre de pods.
- **L'objet kubernetes *HorizontalPodAutoscaler* est particulièrement utile pour maintenir une application stable lors des pics de charge tout en optimisant l'utilisation des ressources.**

Gestion de la scalabilité

Scalabilité verticale et évolutivité des noeuds

- **La scalabilité verticale (*scale up*) consiste à augmenter les ressources (CPU, RAM, stockage ou autre) d'une machine virtuelle ou d'un serveur physique existant.**
- Il s'agit d'une approche qui convient pour les charges de travail pouvant être traitée par un seul serveur puissant mais il est possible que cela ne suffise pas si la demande continue de croître.
- C'est la qu'entre en jeux le concept d'**évolutivité des noeuds (*scale out*) dont le but est simplement de rajouter davantage de noeuds à un cluster.**
- Le *scale out* convient pour les charges de travail qui peuvent être distribuées.
- Ce sont les besoins de vos applications et vos contraintes en matière d'infrastructure qui vous orienterons vers l'une ou l'autre solution.

5. Gestion du stockage



Gestion du stockage

Les volumes

- **Pour rappel, lorsqu'un conteneur est supprimé ou redémarré, toutes les données de son système de fichier qui se trouvent sur la couche conteneurs sont perdues.** Dans Kubernetes cela correspond à la suppression ou au redémarrage d'un pod.
- Voici un exemple de création d'un volume qui monte le répertoire `/opt` du noeud hôte. Ce volume sera monté dans le conteneur dans le répertoire `/data`.
- On utilise les volumes pour de multiples raisons:
  - Communication/synchronisation entre des conteneurs
  - Amélioration des performances en cachant des données
  - Montage du système de fichiers hôte pour accéder, par exemple à des périphériques réseaux
  - Stockage de données persistantes

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod
spec:
  volumes:
    - name: "my-vol"
    hostPath:
      path: "/opt"
  containers:
    - name: conteneur-1
    image: mon-image:latest
    volumeMounts:
      - mountPath: "/data"
      name: "my-vol"
    ports:
    - containerPort: 80
  restartPolicy: Always
```

Gestion du stockage

A quel endroit stocker des données persistantes ?

- Deux alternatives s'offrent à vous lorsque souhaitez stocker des données persistantes, indépendamment du noeud sur lequel se trouve votre pod:
    - **Volume de stockage réseau distant** : il s'agit d'un stockage hébergé chez un fournisseurs cloud tel quel Azure file (Microsoft) ou encore Elastic Block Store (Amazon) pour n'en citer que deux. Kubernetes prends en charge plusieurs protocoles comme NFS ou iSCSI.
    - **Volume de stockage persistant** : Kubernetes dispose d'une solution de stockage persistante intégrée à l'intérieur du cluster que nous allons détailler dans la prochaine slide.

Gestion du stockage

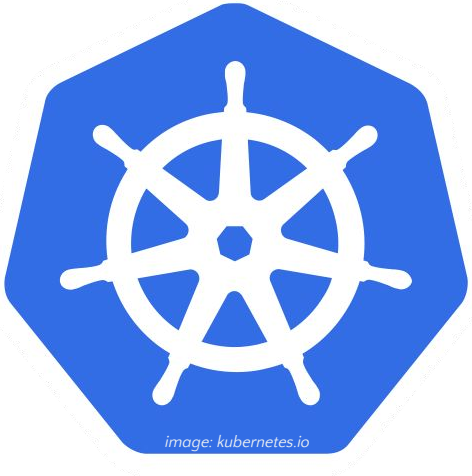
Volume Persistant (*PV*) et Demande de Volume Persistant (*PVC*)

- **Un volume persistant représente une partie physique ou une ressource de stockage dans le cluster, comme par exemple un disque sur un noeud.** Ce sont les administrateurs d'un cluster Kubernetes qui crées les *PV* en associant des ressources de stockage du cluster.

- **Une demande de volume persistant est un objet Kubernetes créé par les développeurs pour demander de l'espace de stockage. Un *PVC* est une demande de quantité pour une classe de stockage précise.** Cela ne définit pas où se trouvent ces ressources dans le cluster.

- **Les classes de stockage sont une abstraction de l'infrastructure sous-jacente permettant de définir des stockages avec différentes propriétés.** Concrètement, en fonction du types de vos disques et des technologies associées vous obtiendrez des différences sur la vitesse en lecture/écriture , la latence, la durée de vie, la fiabilité ou encore le coût. Les utilisateurs d'un cluster Kubernetes n'ont pas besoin de connaître ces détails spécifiques. Ils peuvent simplement choisir une *storageClass* répondant à leur exigence.

6. Défis de l’adoption de Kubernetes



Défis de l'adoption de Kubernetes

La complexité, un frein majeur à l'adoption

- Kubernetes introduit une complexité technique significative qui freine son adoption dans de nombreuses organisations.
- **46% des organisations qui débutent leur parcours cloud native identifient la formation insuffisante comme leur plus grand défi. La sécurité arrive en second avec 40% des organisations qui citent ce problème lors de l'utilisation de conteneurs en production.**
- Kubernetes impose de maîtriser une multiplicité de concepts. Pods, ReplicaSets, Deployments, StatefulSets, DaemonSets, Jobs, CronJobs, Services, Ingress, NetworkPolicies. Chaque concept a son rôle mais augmente la charge cognitive.
- L'écosystème d'outils est fragmenté. kubectl, helm, kustomize, kubectx, stern, k9s. Chacun avec sa propre syntaxe et sa propre logique.
- La courbe d'apprentissage est longue. 6 à 12 mois pour maîtriser les bases opérationnelles. Cette période ne couvre que les fondamentaux nécessaires pour déployer et maintenir des applications simples.

Défis de l'adoption de Kubernetes

Les coûts organisationnels

- Le déploiement de Kubernetes implique **des coûts organisationnels significatifs au-delà de l'infrastructure technique.**
- Former les équipes existantes demande du temps et des ressources. **L'adoption de Kubernetes nécessite souvent une restructuration organisationnelle.**
- **La maintenance est continue et exigeante.** Les mises à jour du cluster sont régulières. Kubernetes ne supporte que trois versions mineures simultanément, avec un cycle de sortie d'environ un an. **Chaque mise à jour requiert des tests de compatibilité des applications.**
- **Les services managés comme GKE, EKS ou OVHcloud Managed Kubernetes réduisent significativement la charge opérationnelle mais ne l'éliminent pas.** Le fournisseur prend en charge la gestion du control plane. Les mises à jour des composants critiques sont automatisées. La haute disponibilité du control plane est garantie sans intervention. Le support technique du cluster est inclus.
- Les coûts humains restent présents mais réduits. Il faut toujours des compétences Kubernetes pour concevoir et déployer les applications. La configuration de l'observabilité reste à la charge de l'équipe. La sécurité applicative et la gestion des secrets demeurent nécessaires. La définition des stratégies réseau et RBAC est toujours requise.

Défis de l'adoption de Kubernetes

Le syndrome de l'over-engineering

Kubernetes n'est pas une solution universelle. **Son adoption doit répondre à des besoins réels et mesurables.**

Quand Kubernetes est pertinent

- Produit avec plus d'une dizaine de services. En dessous, la complexité de Kubernetes dépasse ses bénéfices.
- Besoin de scalabilité automatique forte. Charge variable avec des pics importants nécessitant un autoscaling horizontal.
- Haute disponibilité critique. Le self-healing et les rolling updates assurent la continuité de service.
- Abstraction de l'infrastructure sous-jacente pour migrer entre cloud providers
- Ressources suffisantes pour maintenir l'infrastructure.

Défis de l'adoption de Kubernetes

Le syndrome de l'over-engineering

Alternatives plus adaptées selon le contexte

- Application simple ou monolithe : VMs classiques avec CI/CD. Un monolithe sur VPC OVH avec Nginx et systemd suffit largement.
- Petite équipe de développeurs : Docker Compose. Apprentissage en quelques heures contre plusieurs mois pour Kubernetes.
- Prototypage rapide ou side projects : PaaS comme Heroku, Render ou Fly.io. Déploiement par simple git push sans gérer l'infrastructure.
- Architecture event-driven légère : Serverless avec Lambda ou Cloud Functions. Facturation à l'usage, scaling automatique sans serveur à maintenir.

Défis de l'adoption de Kubernetes

Le syndrome de l'over-engineering

Le principe du monolithe d'abord

- Martin Fowler recommande de commencer par un monolithe avant les microservices. Un monolithe permet de comprendre le domaine métier et d'itérer rapidement.
- Les frontières entre microservices sont difficiles à définir initialement. Les découper trop tôt conduit à des services mal dimensionnés nécessitant des refactorings coûteux.
- Refactorer un monolithe reste plus simple que refondre une architecture microservices mal conçue.
- Migrer vers les microservices devient justifié quand le monolithe atteint ses limites de scalabilité. À ce moment, l'équipe a grandi et les compétences DevOps sont présentes.

Défis de l'adoption de Kubernetes

Le syndrome de l'over-engineering

Évaluer objectivement son contexte

- Quel est le vrai besoin de scalabilité ? Une VM moderne 32 cores sert des milliers de requêtes par seconde.
- Quelle est la criticité réelle ? 99% de disponibilité représente 7 heures d'indisponibilité par mois, souvent suffisant.
- Combien de temps consacré à l'infrastructure ? Au-delà de 30% du temps de développement, il est probable que votre solution soit trop complexe pour vos besoins réels.
- **L'objectif est de créer de la valeur métier, pas de maintenir une infrastructure complexe pour le plaisir de la technique.**

Pour aller plus loin

Ce cours a couvert les **fondamentaux de Kubernetes**. Les domaines suivants constituent des axes d'approfondissement indispensables pour appréhender pleinement les enjeux d'un environnement de production :

- **Observabilité et traçabilité** : Métriques (ex: Prometheus, Grafana), logs (ex: ELK), traces (ex: Jaeger)
- **Stratégies de déploiement avancées** : Au-delà du rolling update, voir les principes Blue/green, Canaray, Feature Flags
- **Ecosystème Kubernetes étendu**: GitOps (ex: ArgoCD, synchronisation Git -> Cluster), service mesh (ex: istio, gestion du trafic inter-services), sécurités (RBAC, Network policies, pod security standards, secrets management)

Kubernetes est un écosystème vaste et en constante évolution. **Ne chercher pas à tout maîtriser d'un coup ! Kubernetes s'apprend par la pratique et l'expérience.**

Bibliographie

BURNS Brendan, BEDA Joe, HIGHTOWER Kelsey, EVENSON Lachlan. Traduit de l'anglais par MANIEZ Dominique. Kubernetes Maîtriser l'orchestrateur des infrastructures du futur. 3e édition. DUNOD, 2023. 308p.

MERCIER Pierre-Olivier. *Conteneurs et technologies du DevOps.* Gentilly: Editions ALPO, 2022. 209 p.

CLOUX Pierre-Yves, GARLOT Thomas, KOHLER Johann. *Docker et conteneurs: architectures, développement, usages et outils.* 3e édition. Malakoff: Dunod, 2022. 317 p.

LANDURE Julien. Pizza As A Service : les différents modèles du Cloud [en ligne]. [Consulté le 02/11/2023]. Disponible à l'adresse: https://medium.zenika.com/pizza-as-a-service-les-differents-modeles-du-cloud-b18ffaa6906c

CNCF 2023 Annual survey [en ligne]. [Consulté le 12/10/2025]. Disponible à l'adresse: https://www.cncf.io/reports/cncf-annual-survey-2023/

Monolith First [en ligne]. Martin Fowler. [Consulté le 12/10/2025]. Disponible à l'adresse: https://martinfowler.com/bliki/MonolithFirst.html

Merci de votre attention

Maxime Lambert

maxime.lambert@unicaen.fr