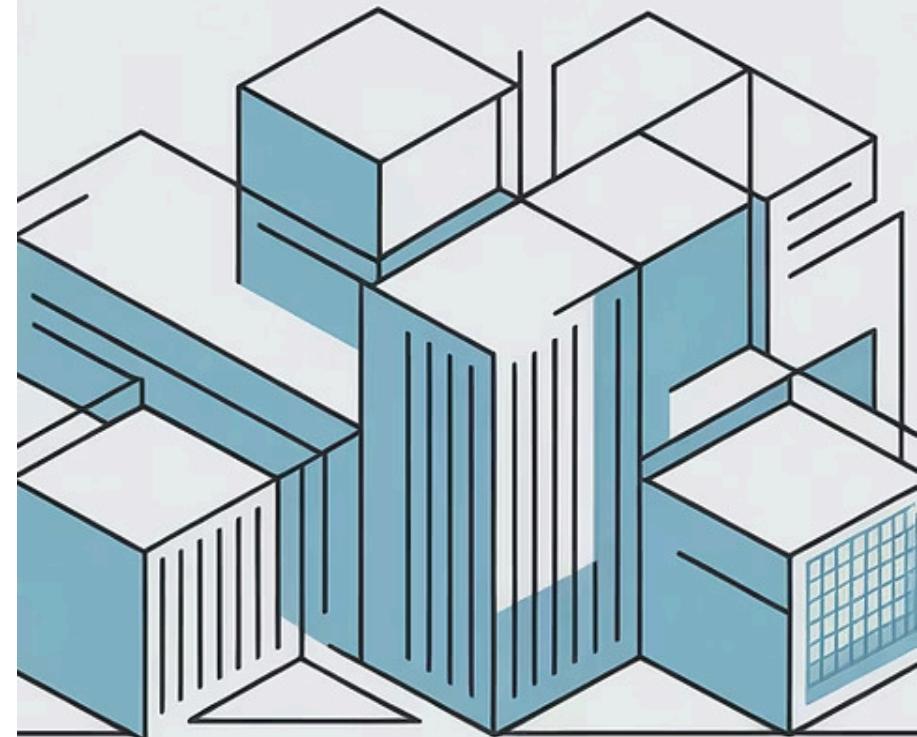


Introduction à Kubernetes

Objectifs du chapitre

- Comprendre ce qu'est Kubernetes et pourquoi il est utilisé.
- Situer Kubernetes par rapport à Docker et aux runtimes de containers.
- Connaître l'écosystème CRI/CNI et ses rôles.
- Découvrir les principales options d'installation (avec un focus sur Minikube).
- Savoir accéder à un cluster via kubectl, un dashboard et l'API.
- Réaliser un premier déploiement et publier une application.
- Apprendre à détailler et à introspecter un déploiement.



Pourquoi Kubernetes ?

Kubernetes est une plateforme d'orchestration de containers qui automatise:

Déploiement

le déploiement des applications

Scaling

la montée en charge (scaling)

Résilience

la tolérance aux pannes et l'auto-réparation

Mises à jour

la mise à jour progressive (rolling update) et le retour arrière (rollback)

Réseau

la gestion réseau et la découverte de services

Il permet de décrire l'état désiré d'une application sous forme de manifestes déclaratifs (YAML). Kubernetes se charge ensuite de faire converger l'état réel vers l'état voulu.

Évolution des relations Docker/Kubernetes

Docker

- Docker a popularisé les containers et l'image OCI (Open Container Initiative). Docker inclut divers composants (construction d'images, runtime, CLI).
- Docker reste très utile pour construire et tester des images.

Kubernetes

- Kubernetes orchestre des containers et repose sur un runtime conforme à la spécification CRI (Container Runtime Interface).
- Historiquement, Docker Engine pouvait être utilisé comme runtime. Aujourd'hui, Kubernetes s'appuie généralement sur des runtimes CRI (ex. containerd, CRI-O).

❑ **En résumé:** Docker (outil de build et de packaging) + Kubernetes (orchestrateur) + un runtime CRI (exécution) = pipeline moderne de déploiement de containers.

L'ensemble CRI / CNI / Kubernetes



CRI (Container Runtime Interface)

Abstraction utilisée par kubelet pour piloter l'exécution des containers.

Exemples: containerd, CRI-O.



CNI (Container Network Interface)

Standard pour brancher des plugins réseau au cluster.

Exemples: Calico, Cilium, Flannel.

Garantit que chaque Pod dispose d'une IP et que la connectivité Pod ↔ Pod et Pod ↔ Service fonctionne selon les règles.



Kubernetes (API, scheduler, contrôleurs)

Expose des objets (Pod, Deployment, Service, ConfigMap, Secret, etc.).

Contrôleurs réconcilient l'état désiré (YAML) avec l'état courant.

Schéma simplifié:

```
[Manifeste YAML] --> [API Server] --> [Controllers/Scheduler] --> [Kubelet]  
|--> [CRI: containerd/CRI-O]  
|--> [CNI: Calico/Cilium/...]
```

Solutions d'installation

Plusieurs approches existent selon le besoin (apprentissage, CI, environnements):

Minikube

Solution locale simple et pédagogique pour lancer un cluster sur une machine personnelle.

Supporte divers drivers (Docker, Hyperkit, KVM, etc.).

Idéal pour expérimenter rapidement les commandes kubectl et les manifestes.

Distributions managées

Fournissent un plan de contrôle managé, simplifiant l'exploitation.

Généralement utilisées pour des environnements de production.

On-Premise / Bare-metal

Déploiement sur des serveurs internes.

Contrôle total, mais nécessite la gestion du plan de contrôle, du réseau CNI, du stockage, de l'observabilité, etc.

Outils d'installation automatisés

kubeadm (référence pour bootstraper un cluster), kubespray, rancher RKE, kOps (selon les cibles et contraintes).

- ❑ **Note:** pour faciliter la mise en pratique des concepts, les exemples feront référence à Minikube.

Démarrer avec Minikube

Installation

Exemple générique:

- Installer kubectl.
- Installer minikube.

Lancer un cluster local

```
minikube start --driver=docker
```

Vérifier l'accès

```
kubectl get nodes  
kubectl cluster-info
```

Minikube fournit aussi des utilitaires pratiques (ex. tunnel, addons) pour simuler des comportements proches d'un cluster complet.

Accéder au cluster Kubernetes

1

Via la CLI: kubectl

Configuration du contexte (généralement créée par Minikube):

```
kubectl config get-contexts  
kubectl config use-context minikube
```

Commandes de base:

```
kubectl get nodes  
kubectl get pods -A  
kubectl describe node  
kubectl api-resources
```

Utilisation des espaces de noms:

```
kubectl get ns  
kubectl -n kube-system get pods
```

2

Via un Dashboard (interface web)

Minikube peut activer un dashboard:

```
minikube dashboard
```

Le dashboard permet de visualiser les objets (Pods, Deployments, Services), d'inspecter les événements et de surveiller l'état du cluster.

3

Via l'API Kubernetes

L'API Server expose toutes les opérations CRUD sur les objets.

kubectl est un client officiel de cette API.

L'API est accessible avec authentification/autorisation et peut être interrogée directement:

```
kubectl proxy  
curl http://127.0.0.1:8001/apis
```

Déployer une application

Un déploiement standard se fait avec un objet Deployment qui gère un ReplicaSet de Pods.

Exemple minimal d'un Deployment NGINX:

```
# fichier: deploy-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-nginx
  labels:
    app: web-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-nginx
  template:
    metadata:
      labels:
        app: web-nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
```

Appliquer et vérifier le déploiement

Contenu pratique avec les commandes pour appliquer le manifeste et vérifier le déploiement:

Appliquer le manifeste

```
kubectl apply -f deploy-nginx.yaml
```

Vérifier le déploiement

```
kubectl get deploy  
kubectl get rs  
kubectl get pods -l app=web-nginx  
kubectl describe deploy web-nginx
```

Publication manuelle d'une application

Plusieurs options existent pour exposer un service:



Port-forward

Pour un accès rapide local:

```
kubectl port-forward deploy/web-nginx 8080:80
```

Puis accéder à <http://localhost:8080>

Service ClusterIP

Service de type ClusterIP + port-forward ponctuel:

```
# fichier: svc-nginx.yaml
apiVersion: v1
kind: Service
metadata:
  name: web-nginx
spec:
  selector:
    app: web-nginx
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
```

```
kubectl apply -f svc-nginx.yaml
kubectl port-forward svc/web-nginx 8080:80
```

Service NodePort

Exposition sur un port du nœud:

```
apiVersion: v1
kind: Service
metadata:
  name: web-nginx-nodeport
spec:
  type: NodePort
  selector:
    app: web-nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
```

```
kubectl apply -f svc-nodeport.yaml
```

Sur Minikube:

```
minikube service web-nginx-nodeport --url
```

□ **Remarque:** l'usage d'un Ingress Controller est courant pour une exposition plus flexible. Sur Minikube, un addon Ingress peut être activé si nécessaire.

Détail et introspection d'un déploiement

Savoir lire l'état et diagnostiquer est essentiel.

Objets et labels

```
kubectl get deploy,rs,pods -l app=web-nginx -o wide  
kubectl get all -l app=web-nginx  
kubectl get events --sort-by=.lastTimestamp
```

Décrire et inspecter:

```
kubectl describe deploy web-nginx  
kubectl describe rs -l app=web-nginx  
kubectl describe pod <pod-name>
```

Journaux applicatifs:

```
kubectl logs deploy/web-nginx  
kubectl logs pod/<pod-name> -c nginx --previous
```

Exécuter une commande dans un Pod:

```
kubectl exec -it deploy/web-nginx -- sh
```

Problèmes fréquents et dépannage

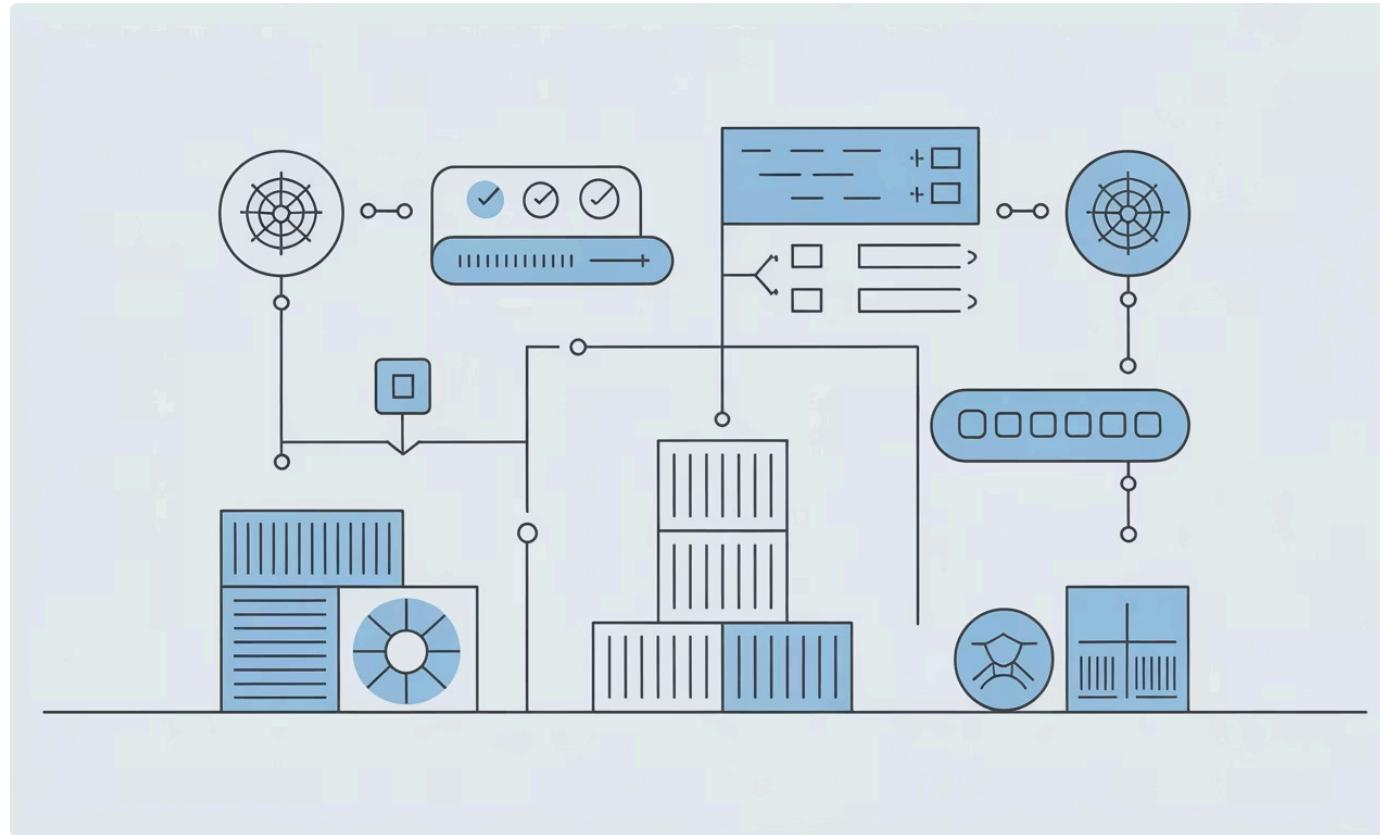
Problèmes fréquents à repérer:

- **Image introuvable ou pull rate-limited**
 - événements sur le Pod / état des containers
- **Affinité/ressources inadéquates**
 - scheduler ne place pas le Pod
- **Probes qui échouent (readiness/liveness)**
 - vérifier spec.containers, routes, ports
- **Port déjà utilisé sur NodePort**
 - choisir un autre nodePort

Stratégies de mise à jour et historique:

```
kubectl rollout status deploy/web-nginx  
kubectl rollout history deploy/web-nginx  
kubectl rollout undo deploy/web-nginx
```

Configuration des probes et ressources



Exemple de configuration des probes et ressources pour un déploiement robuste :

```
spec:  
  containers:  
    - name: nginx  
      image: nginx:1.25  
      ports:  
        - containerPort: 80  
      readinessProbe:  
        httpGet:  
          path: /  
          port: 80  
        initialDelaySeconds: 3  
        periodSeconds: 5  
      resources:  
        requests:  
          cpu: "100m"  
          memory: "128Mi"  
        limits:  
          cpu: "500m"  
          memory: "256Mi"
```

Les probes permettent à Kubernetes de vérifier l'état de santé de l'application, tandis que les ressources garantissent un placement et une utilisation appropriés.

Bonnes pratiques initiales



Labels cohérents

Déclarer des labels cohérents (app, tier, part-of, version) pour faciliter la sélection et l'observabilité.



Versionner les images

Versionner et paramétriser les images: toujours préciser un tag d'image stable.



Définir des probes

Définir des probes pour la disponibilité applicative.



Ressources

Définir des ressources (requests/limits) pour une planification fiable.



Namespaces

Utiliser des namespaces pour séparer les environnements logiques.



Configuration externe

Centraliser la configuration avec ConfigMap/Secret (même si l'usage détaillé sera approfondi plus tard).



Référentiel Git

Tenir les manifestes dans un référentiel (Git) et appliquer des revues.

Récapitatif



Kubernetes orchestre des containers et s'appuie sur des standards: CRI pour l'exécution, CNI pour le réseau.



Minikube est un moyen simple d'exécuter un cluster local et d'apprendre kubectl.



Un Deployment gère la réPLICATION et les mises à jour de Pods, un Service expose le trafic.



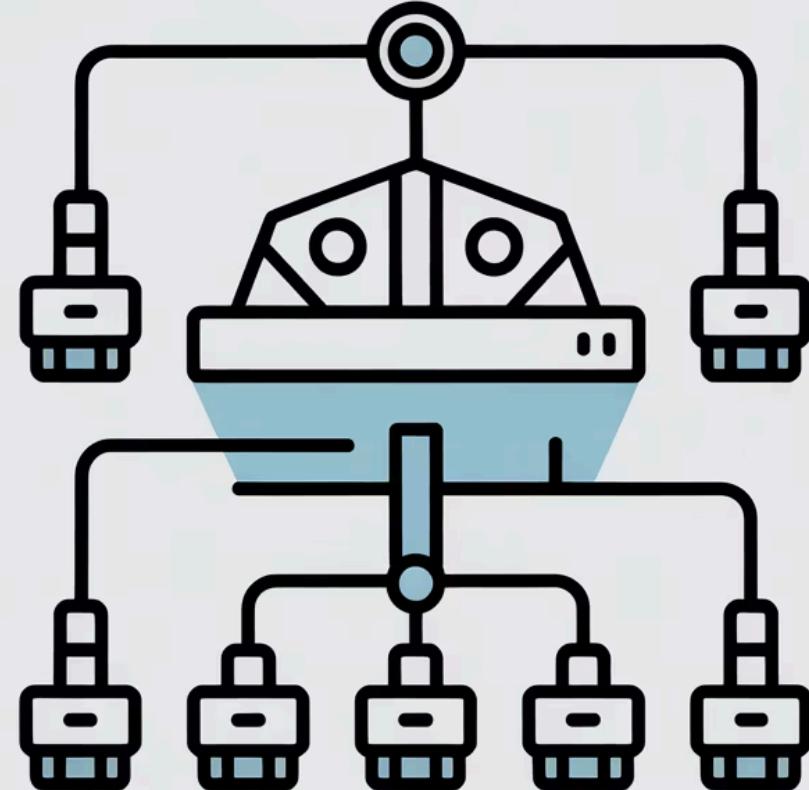
Les outils d'introspection (get/describe/logs/events/rollout) sont indispensables pour diagnostiquer et piloter un déploiement.

Ce chapitre pose le vocabulaire et les réflexes de base. Les chapitres suivants approfondiront l'architecture, l'exploitation, la production et le déploiement d'un cluster complet.

Architecture Kubernetes

Objectifs du chapitre

- Comprendre les composants du plan de contrôle (control plane) et leur rôle.
- Identifier les composants d'un nœud et la chaîne d'exécution des Pods.
- Connaître les principaux objets Kubernetes et leur articulation.
- Distinguer les applications stateless et stateful et choisir l'objet adapté.
- Introduire la solution de déploiement standard avec Deployment.



Vue d'ensemble

Kubernetes met en œuvre un modèle déclaratif: on décrit un état désiré via l'API; des contrôleurs font converger l'état réel vers cet état. L'architecture se compose de deux ensembles:

Plan de contrôle (control plane)

Expose l'API et orchestre les décisions.

Nœuds (workers)

Exécutent les pods (unités d'exécution de containers).

Schéma simplifié:

```
[Client: kubectl/CI]
--> [API Server]
--> [etcd]
|
+--> [Scheduler]
+--> [Controller Manager(s)]
|
[Node]
kubelet -- CRI(containerd/CRI-O) -- CNI -- kube-proxy -- Pods
```

Composants du plan de contrôle



Contrôleurs et gestionnaires



Controller Manager

Ensemble de contrôleurs supervisant des boucles de réconciliation:

- Deployment/ReplicaSet/ReplicationController
- StatefulSet
- Job/CronJob
- Node, EndpointSlice, Namespace, ServiceAccount, etc.

Compare en continu l'état courant à l'état désiré et agit en conséquence.



Cloud Controller Manager

Intègre des fonctionnalités spécifiques à un fournisseur (ex: LoadBalancer, nœuds, disques).

Sépare la logique cloud-specific du noyau Kubernetes.

Add-ons et points clés

Add-ons critiques:

CoreDNS

service DNS interne pour la découverte des services

Ingress/Gateway Controller

route le trafic HTTP/HTTPS vers les Services

Points clés à retenir:

 Tous les accès passent par l'API Server

 Les contrôleurs ne modifient jamais directement etcd; ils parlent à l'API

 Les décisions de placement appartiennent au Scheduler; l'exécution au kubelet

Architecture d'un nœud



Kubelet

Agent local au nœud; reçoit les Pods à exécuter via l'API.

Interagit avec le runtime de containers via la CRI.

Gère le cycle de vie des pods/containers (start/stop, probes, logs, volumes).



Runtime de containers (CRI)

Interface standard: Container Runtime Interface.

Implementations courantes: containerd, CRI-O.

Télécharge et exécute les images, gère les sandboxes (netns), montages, etc.



kube-proxy

Programme les règles de dataplane pour les Services (ClusterIP/NodePort).

Modes iptables/ipvs; avec certains CNI modernes, peut être remplacé/optimisé par eBPF.

Interfaces réseau et stockage



CNI (réseau)

Branche les pods au réseau, assigne des adresses IP, définit le routage/policies.

Exemples: Calico, Cilium, Flannel.

Garantit le modèle réseau Kubernetes: chaque pod a sa propre IP; la communication Pod ↔ Pod et Pod ↔ Service est routable.



CSI (stockage)

Interface de stockage pour provisionner/attacher des volumes via un driver CSI.

Intervient lors de la création des PersistentVolumes/PersistentVolumeClaims.



Autres composants utiles

- Kubelet plugins pour volumes (CSI Node plugin)
- Outils d'observabilité (agents DaemonSet, logs, métriques) déployés comme Pods

Chaîne d'exécution d'un Pod

Résumé du processus d'exécution :



Création / Actualisation de l'objet

Un contrôleur crée/actualise un objet (ex: ReplicaSet) → un Pod non assigné apparaît.



Assignation par le Scheduler

Le Scheduler assigne le Pod à un nœud.



Exécution par le Kubelet

Le kubelet du nœud crée la sandbox réseau via CNI, demande au runtime CRI de lancer les containers, monte les volumes via CSI si nécessaire.



Prise en charge du trafic

Les probes déterminent readiness/liveness; le Service peut router le trafic vers le Pod prêt.

Objets Kubernetes essentiels



Namespace

Cloison logique des ressources (isolation logique, quotas, RBAC).



Pod

Unité minimale d'exécution (un ou plusieurs containers co-localisés).

Partage de réseau (IP/ports) et de volumes au sein du Pod.
Éphémère par nature; remplacé plutôt que réparé.



ReplicaSet

Assure un nombre désiré de répliques de Pods identiques.
Souvent géré indirectement par un Deployment.



Deployment

Gère la stratégie de mise à jour (rolling update), l'historique, les rollbacks.

Modifie les ReplicaSets sous-jacents.



StatefulSet

Pour les workloads avec état nécessitant une identité stable (nom/pod ordinal), un stockage persistant lié, un démarrage/arrêt ordonné.

Souvent utilisé avec un Service headless.



DaemonSet

Garantit l'exécution d'un Pod par nœud (ou par sous-ensemble de nœuds).

Typique pour agents système, logging, monitoring, CNI/CSI nodes.



Job / CronJob

Exécutions finies (batch) et planifiées.



Service

Abstraction réseau qui expose un ensemble de Pods via une IP/port stable.

Types: ClusterIP (interne), NodePort, LoadBalancer.
Sélectionne les Pods via labels.



Ingress / Gateway

Route HTTP(S) vers des Services selon nom d'hôte/chemins.

Requiert un contrôleur déployé dans le cluster.



ConfigMap / Secret

Configuration externe à l'image (texte) et données sensibles (Secrets) montées en volume ou injectées en variables d'environnement.



Volumes

Attachement de stockage à un Pod. Classes:

- éphémères (emptyDir, configMap, secret, projected)
- persistantes via PVC/PV/StorageClass.

Volumes et stockage

PersistentVolume (PV)

Ressource de stockage fournie au cluster (statiquement ou dynamiquement).

PersistentVolumeClaim (PVC)

Demande de stockage par une application (capacité, mode d'accès, StorageClass).

StorageClass

Définit la politique de provisionnement dynamique (paramètres, type de disque, réplication, mode de binding).

Modes d'accès

ReadWriteOnce (RWO), ReadOnlyMany (ROX), ReadWriteMany (RWX) selon le backend.

Volumes éphémères

emptyDir (lié au cycle de vie du Pod), configMap/secret, downwardAPI.
À privilégier pour caches/temporaire, jamais pour données critiques.

Bonnes pratiques:

- Utiliser PVC/PV/StorageClass pour les données persistantes.
- Définir des stratégies de sauvegarde/restauration au niveau du backend.
- Préciser les modes d'accès requis par l'application.

Réseau et Services (aperçu architectural)

Modèle réseau plat Chaque Pod a une IP routable dans le cluster.	Service (ClusterIP) IP virtuelle stable exposant un ensemble d'Endpoints/EndpointSlices. kube-proxy ou dataplane eBPF réalise le load-balancing au niveau nœud.	NodePort / LoadBalancer NodePort ouvre un port sur tous les nœuds; LoadBalancer s'appuie sur une intégration externe.
DNS CoreDNS résout les noms des Services/Pods; les apps utilisent le DNS interne.	NetworkPolicy Règles d'autorisation L3/L4 entre pods (implémentées par le CNI).	

Stateless vs Stateful

Stateless (sans état persistant)

- Horizontalement scalable; répliques interchangeables.
- Pannes gérées par simple remplacement de Pods.
- Exemple d'objets: Deployment + Service.

Stateful (avec état persistant)

- Chaque instance peut nécessiter un identifiant stable, un volume dédié, un ordre de démarrage/arrêt.
- Exemple d'objets: StatefulSet + PVC + Service headless.
- Cas typiques: bases de données, systèmes de files, clusters distribués.

Critères de choix:

- Besoin d'une identité stable (pod-0, pod-1...) → StatefulSet.
- Volumes dédiés par instance (un PVC par Pod) → StatefulSet.
- Instances interchangeables, trafic stateless → Deployment.

Anti-patterns:

- Stocker des données critiques dans emptyDir/hostPath.
- Utiliser Deployment pour des bases de données nécessitant des identités/volumes stables.

Solution du Deployment (déploiement standard)

Le Deployment est l'objet de haut niveau recommandé pour les applications stateless.

Structure minimale

- labels cohérents (app, version), selector aligné sur template.metadata.labels
- stratégie de rolling update (maxSurge, maxUnavailable)
- probes (readiness/liveness/startup) pour signaler l'état
- resources requests/limits pour un placement fiable
- variables d'environnement, volumes/configuration externes

Exemple de Deployment complet

Voici un exemple pratique de fichier de configuration YAML pour un Deployment Kubernetes, démontrant les meilleures pratiques et les paramètres essentiels :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-demo
  labels:
    app: api-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-demo
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: api-demo
    spec:
      containers:
        - name: app
          image: ghcr.io/org/api-demo:1.0.0
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
              periodSeconds: 5
          livenessProbe:
            httpGet:
              path: /livez
              port: 8080
              initialDelaySeconds: 10
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "256Mi"
```



Service associé et points d'attention

Service associé :

```
apiVersion: v1
kind: Service
metadata:
  name: api-demo
spec:
  selector:
    app: api-demo
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

Points d'attention :

- Le selector du Service doit matcher les labels des Pods
- Les probes conditionnent l'inclusion des Pods derrière le Service
- L'historique des ReplicaSets permet rollback/rollout control

Labels, selectors et organisation

Labels

Paires clé/valeur attachées aux objets; base de la sélection.

Selectors

Expressions logiques pour cibler des ensembles d'objets (matchLabels, matchExpressions).

Recommandations:

- app, part-of, version, tier (frontend/backend/db), environment.
- Cohérence labels/selector entre Deployment/Service/NetworkPolicy.

Admission et sécurité (survol)

Admission Controllers

Mutating/Validating webhooks pour imposer des normes (images, limites, annotations).

RBAC

Rôles et liaisons pour contrôler l'accès aux ressources.

Policies

Pod Security (modes baseline/restricted selon la configuration), SecurityContext.

Images

Politique d'extraction (imagePullPolicy), registries privés (Secrets), signatures.

Récapitatif

Le plan de contrôle centralise l'API, la persistance (etcd), la planification (scheduler) et la réconciliation (controllers).

Un nœud exécute les Pods via kubelet, un runtime CRI et un CNI; kube-proxy/eBPF assurent le dataplane des Services.

Les objets forment un graphe logique: Deployment/ReplicaSet/Pod pour stateless; StatefulSet/PVC pour stateful; Service/Ingress pour l'exposition; ConfigMap/Secret/Volume pour la configuration et les données.

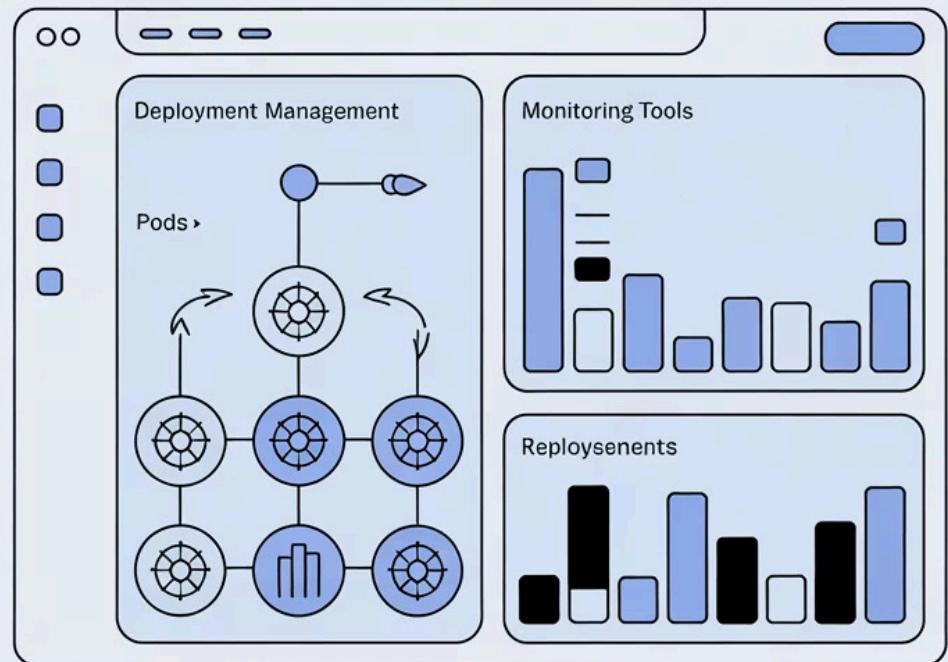
Le Deployment est la solution de déploiement standard pour les workloads stateless, avec rolling update, probes et ressources définies.

Exploiter Kubernetes



Objectifs du chapitre

- Piloter le cycle de vie d'un Deployment (révisions, mises à jour, retours arrière).
- Choisir et configurer le bon type de Service pour l'exposition réseau.
- Organiser l'ordonnancement via labels, sélection de nœuds et contraintes.
- Mettre en œuvre affinités et anti-affinités entre Pods et avec les nœuds.
- Déployer des DaemonSets, définir des probes de santé.
- Gérer la configuration et les secrets avec ConfigMap et Secret.
- Comprendre et utiliser StorageClass, PV et PVC pour la persistance.



Gérer les révisions d'un Deployment

Un Deployment gère les ReplicaSets sous-jacents et maintient un historique de révisions. Chaque modification du template de Pod crée une nouvelle révision.

Voir l'état d'un déploiement:

```
kubectl get deploy myapp  
kubectl describe deploy myapp
```

Suivre un déploiement:

```
kubectl rollout status deploy/myapp
```

Lister l'historique des révisions:

```
kubectl rollout history deploy/myapp
```

Détails d'une révision précise:

```
kubectl rollout history deploy/myapp --revision=3
```

Stratégie de mise à jour (exemple):

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: myapp  
spec:  
  replicas: 4  
  strategy:  
    type: RollingUpdate  
    rollingUpdate:  
      maxSurge: 1 # nombre max de Pods supplémentaires pendant l'update  
      maxUnavailable: 0 # nombre max de Pods indisponibles pendant l'update  
  revisionHistoryLimit: 10  
  selector:  
    matchLabels:  
      app: myapp  
  template:  
    metadata:  
      labels:  
        app: myapp  
    spec:  
      containers:  
      - name: app  
        image: registry.example.com/myapp:1.2.3  
      ports:  
      - containerPort: 8080
```

Bonnes pratiques:

- Définir `revisionHistoryLimit` pour maîtriser l'historique et éviter d'accumuler trop de ReplicaSets.
- Utiliser des tags d'images immuables (ex: 1.2.3) plutôt que `latest`.

Retour arrière (undo) vers la révision précédente:

```
kubectl rollout undo deploy/myapp
```

Retour arrière vers une révision spécifique:

```
kubectl rollout undo deploy/myapp --to-revision=3
```

Types de Services

Un Service fournit une identité réseau stable et un équilibrage vers des Pods sélectionnés par labels.

ClusterIP (par défaut)

Exposé uniquement à l'intérieur du cluster.

Usage: communication inter-services interne.

```
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
    - port: 80
      targetPort: 8080
      protocol: TCP
```

NodePort

Ouvre un port sur tous les nœuds et redirige vers le Service.

Usage: tests rapides, environnements sans LoadBalancer natif.

```
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  type: NodePort
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 32080 # optionnel
```

LoadBalancer

Demande un équilibrEUR externe via l'intégration cloud ou un contrôleur.

Usage: exposition externe simple d'un service TCP/UDP.

```
apiVersion: v1
kind: Service
metadata:
  name: public-api
spec:
  type: LoadBalancer
  selector:
    app: api
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

Headless (clusterIP: None)

Ne fournit pas d'IP virtuelle; résout directement les Endpoints/EndpointSlices.

Usage: découverte de pairs, StatefulSet.

```
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  clusterIP: None
  selector:
    app: db
  ports:
    - port: 5432
      targetPort: 5432
```

ExternalName

Mappe un nom interne vers un FQDN externe via un enregistrement DNS CNAME.

Usage: pointer vers un service géré hors cluster.

Labels et choix d'un nœud pour le déploiement

Les labels sont des paires clé/valeur utilisés pour:

- sélectionner des Pods (Services, NetworkPolicies),
- organiser et filtrer les objets,
- poser des contraintes d'ordonnancement.

Recommandations de labels:

app, part-of, version, tier, environment.

Cohérence entre Deployment, Service, NetworkPolicy, etc.

Choisir un nœud:

nodeSelector

Sélection simple par labels de nœuds.

```
spec:  
  nodeSelector:  
    nodepool: high-cpu
```

nodeName

Impose un nœud précis (fort couplage; à éviter sauf cas particuliers).

```
spec:  
  nodeName: worker-02
```

Taints & tolerations

Un taint repousse les Pods sauf s'ils déclarent une tolérance correspondante.

Utile pour réserver des nœuds à des workloads spécifiques.

```
spec:  
  tolerations:  
    - key: "dedicated"  
      operator: "Equal"  
      value: "ml"  
      effect: "NoSchedule"
```

Affinité et anti-affinité

Affiner l'ordonnancement pour gérer la colocalisation ou la dispersion:

Node Affinity

Contraintes basées sur les labels de nœuds.

- requiredDuringSchedulingIgnoredDuringExecution: dur et strict.
- preferredDuringSchedulingIgnoredDuringExecution: préférence pondérée.

Pod Affinity / Anti-Affinity

Basées sur les labels des Pods déjà planifiés.

- **Pod Affinity:** rapproche des Pods correspondants (ex: mettre frontend proche du cache).
- **Pod Anti-Affinity:** éloigne des Pods correspondants (ex: répartir des réplicas).

Exemple combiné:

```
spec:  
  affinity:  
    nodeAffinity:  
      requiredDuringSchedulingIgnoredDuringExecution:  
        nodeSelectorTerms:  
          - matchExpressions:  
            - key: nodepool  
              operator: In  
              values: ["high-mem", "general"]  
      preferredDuringSchedulingIgnoredDuringExecution:  
        - weight: 50  
          preference:  
            matchExpressions:  
              - key: topology.kubernetes.io/zone  
                operator: In  
                values: ["zone-a"]  
    podAntiAffinity:  
      preferredDuringSchedulingIgnoredDuringExecution:  
        - weight: 100  
          podAffinityTerm:  
            labelSelector:  
              matchLabels:  
                app: myapp  
            topologyKey: "kubernetes.io/hostname"
```

Points clés:

- topologyKey contrôle le domaine de dispersion (hostname, zone, région).
- Utiliser anti-affinité pour éviter la co-localisation de réplicas sur un même nœud.

DaemonSet, probes de santé, ConfigMap et Secret

DaemonSet:

Garantit au moins un Pod par nœud (ou sous-ensemble via nodeSelector/affinity/taints).

Cas d'usage: agents de logs, monitoring, CNI/CSI, sidecars système.

Exemple DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
    metadata:
      labels:
        app: node-exporter
    spec:
      containers:
        - name: exporter
          image: prom/node-exporter:v1.7.0
          ports:
            - containerPort: 9100
```

Probes de santé:

- **livenessProbe:** redémarre le container si l'app est bloquée.
- **readinessProbe:** détermine si le Pod peut recevoir du trafic.
- **startupProbe:** laisse du temps au démarrage avant d'activer liveness/readiness.

Exemple probes HTTP:

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  periodSeconds: 5
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 10
  failureThreshold: 30
  periodSeconds: 5
  startupProbe:
    httpGet:
      path: /startup
      port: 8080
```

ConfigMap:

Stockez la configuration non sensible, injectée via variables d'environnement ou volumes.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: "production"
  LOG_LEVEL: "info"
```

Injection via env:

```
containers:
- name: app
  image: ghcr.io/org/app:1.0.0
  envFrom:
  - configMapRef:
    name: app-config
```

Montage en volume:

```
volumes:
- name: cfg
configMap:
  name: app-config
containers:
- name: app
  volumeMounts:
  - name: cfg
    mountPath: /etc/app
```

Secret:

Données sensibles encodées en base64 (clé API, mot de passe, certificat).

Opaque:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: ZGJ1c2Vy
  password: c2VjdXJlUGFzcw==
```

Référence en env:

```
containers:
- name: app
  env:
  - name: DB_USER
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: username
  - name: DB_PASS
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: password
```

Montage en volume:

```
volumes:
- name: db-cred
secret:
  secretName: db-secret
```

Bonnes pratiques:

- Ne pas commettre de secrets en clair.
- Contrôler l'accès via RBAC.
- Préférer des mécanismes de rotation et, si possible, des solutions de chiffrement côté serveur.

StorageClass, PersistentVolume et PersistentVolumeClaim

Objectifs:

- Déléguer la gestion du stockage au cluster.
- Distinguer la demande (PVC) de la ressource (PV).
- Automatiser la création via StorageClass (provisionnement dynamique).

StorageClass

Définit le provisioner (driver CSI), les paramètres et la politique de binding/reclaim.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: csi.example.com
parameters:
  type: "ssd"
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
```

PersistentVolumeClaim (PVC)

Requête de capacité et mode d'accès par l'application.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-claim
spec:
  storageClassName: fast-ssd
  accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 20Gi
```

Utilisation d'un PVC dans un Pod/Deployment:

```
volumes:
- name: app-data
  persistentVolumeClaim:
    claimName: data-claim
containers:
- name: app
  image: ghcr.io/org/app:1.0.0
  volumeMounts:
    - name: app-data
      mountPath: /var/lib/app
```

Modes d'accès:

- **ReadWriteOnce (RWO):** lecture/écriture par un seul nœud.
- **ReadOnlyMany (ROX):** lecture seule par plusieurs nœuds.
- **ReadWriteMany (RWX):** lecture/écriture par plusieurs nœuds (selon le backend).

Politiques:

- **reclaimPolicy:** Retain (conserver), Delete (supprimer), Recycle (déprécié).
- **volumeBindingMode:**
 - Immediate: provision dès la création.
 - WaitForFirstConsumer: provision lors de la planification du premier Pod (optimise la localisation).

Bonnes pratiques:

- Choisir une StorageClass adaptée aux besoins (latence, IOPS, coûts).
- Activer l'expansion de volume si supportée (allowVolumeExpansion).
- Vérifier les modes d'accès requis par l'application.
- Mettre en place sauvegarde/restauration au niveau du backend.

Récapitulatif

Les Deployments maintiennent des révisions; rollout/undo permettent de contrôler les mises à jour.

Les Services offrent plusieurs modes d'exposition: ClusterIP, NodePort, LoadBalancer, Headless, ExternalName.

Les labels sont le socle de la sélection; nodeSelector et (survol) taints/tolerations orientent le placement.

Les affinités/anti-affinités affinent la colocalisation ou la dispersion des Pods.

Les DaemonSets assurent un Pod par nœud; les probes garantissent la santé applicative.

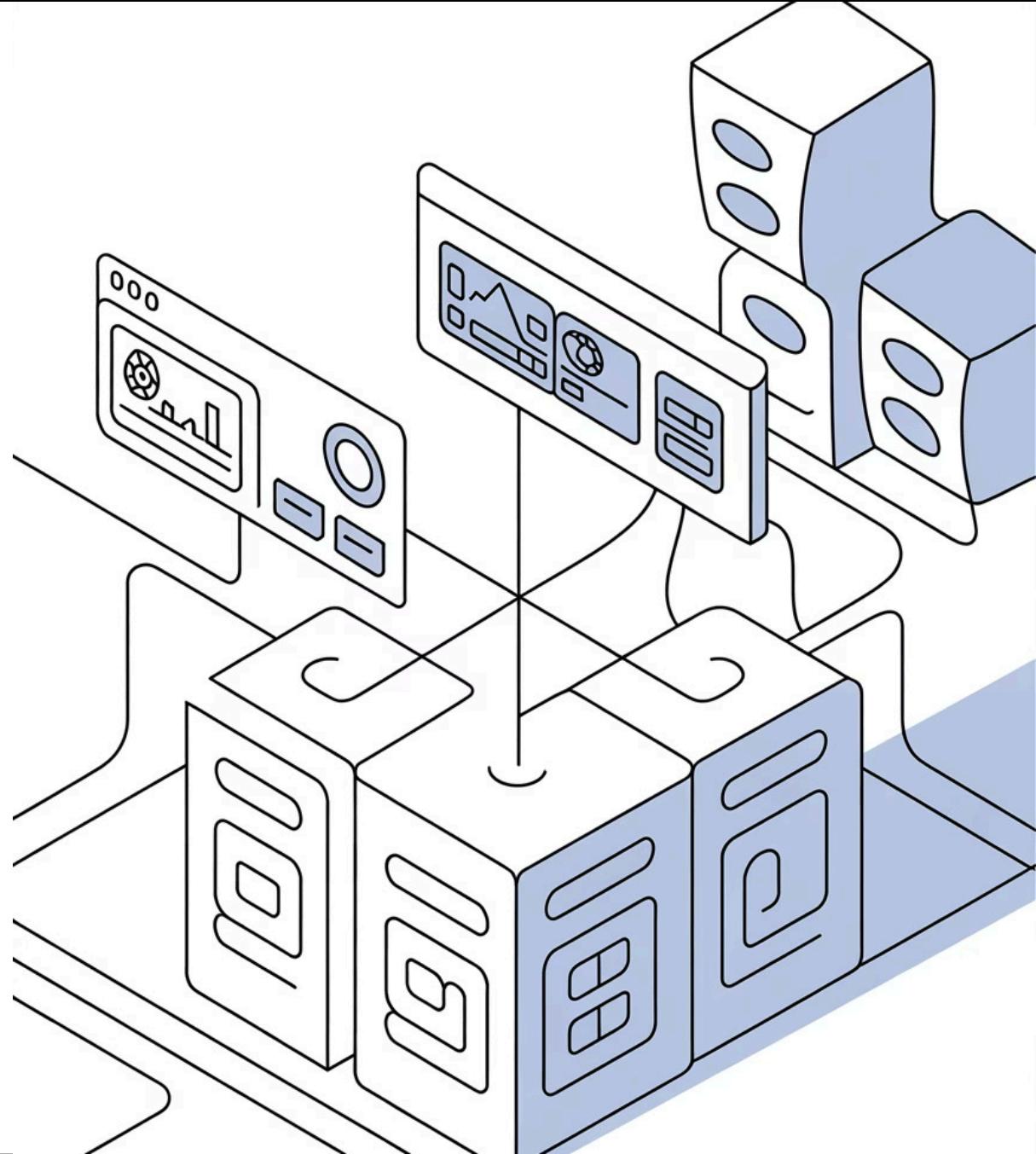
ConfigMap et Secret externalisent la configuration et les données sensibles.

StorageClass, PV et PVC rendent la persistance déclarative et portable.

Kubernetes en production

Objectifs du chapitre

- Mettre en place un reverse proxy et le routage HTTP/HTTPS via Ingress (exemple Traefik).
- Dimensionner correctement les ressources avec requests/limits et comprendre les classes QoS.
- Mettre en œuvre l'autoscaling horizontal.
- Utiliser la découverte de services via variables d'environnement et DNS interne.
- Organiser les isolations par namespaces, quotas et limites par défaut.
- Gérer les accès avec RBAC et ServiceAccounts.
- Concevoir pour la haute disponibilité et appliquer un mode maintenance contrôlé.



Reverse proxy et routage Ingress (Traefik)

Un Ingress expose des Services HTTP/HTTPS au trafic entrant. Traefik peut agir comme contrôleur d'Ingress et reverse proxy.

Principes:

- Un IngressController observe les objets Ingress et configure dynamiquement le proxy.
- Les Services cibles restent internes (ClusterIP); l'Ingress s'occupe du routage L7.
- TLS peut être terminé au niveau du proxy.

Avec Minikube:

- Déployer Traefik via un manifeste Helm ou YAML.
- Créer des Services ClusterIP et un Ingress pointant vers ces Services.

Exemple de Service et Ingress

Exemple de Service + Ingress standard (compatible Traefik) :

```
apiVersion: v1
kind: Service
metadata:
  name: web
  namespace: prod
spec:
  selector:
    app: web
  ports:
    - name: http
      port: 80
      targetPort: 8080
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web
  namespace: prod
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints: web
spec:
  rules:
    - host: web.example.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
                port:
                  number: 80
```

Configuration TLS et points d'attention

La terminaison TLS peut être gérée au niveau de l'Ingress Controller, comme Traefik, déchargeant les services backend de cette tâche.

TLS (terminaison au proxy) :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-tls
  namespace: prod
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints: websecure
spec:
  tls:
    - hosts: ["web.example.local"]
      secretName: web-cert
  rules:
    - host: web.example.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
                port:
                  number: 80
```

Points d'attention :

- Choisir un contrôleur d'Ingress unique pour éviter des chevauchements.
- Standard Ingress couvre la majorité des cas; Traefik propose aussi des CRD (IngressRoute) pour des scénarios avancés.
- Gérer les headers (X-Forwarded-*) et les timeouts via annotations.

Requests, Limits et classes de Qualité de Service (QoS)

Définitions

- **Requests:** ressources garanties pour la planification (scheduling).
- **Limits:** plafond d'utilisation autorisé par le runtime.
- **Unité CPU:** millicores (500m = 0,5 vCPU). Mémoire: octets (Mi, Gi).

Exemple:

```
resources:  
  requests:  
    cpu: "200m"  
    memory: "256Mi"  
  limits:  
    cpu: "500m"  
    memory: "512Mi"
```

Classes QoS

Impactent les priorités d'éviction en cas de pression mémoire:

- **Guaranteed:** requests = limits pour CPU et RAM sur tous les containers.
- **Burstable:** requests définis, limits absents ou supérieurs.
- **BestEffort:** aucun requests/limits; évicté en premier.

Bonnes pratiques:

- Définir requests réalistes (basés sur l'observation) pour une planification fiable.
- Définir limits mémoire pour prévenir les débordements; CPU limit selon besoin (le throttling peut nuire aux latences).
- Harmoniser les requests avec l'autoscaling CPU/mémoire.

Autoscaling d'une application

HPA (HorizontalPodAutoscaler) ajuste le nombre de réplicas d'un Deployment/ReplicaSet/StatefulSet selon des métriques.

Mécanismes courants:

- Métriques CPU/Memory (resources).
- Métriques personnalisées (Prometheus Adapter, etc.).

Exemple de configuration HPA

Exemple HPA v2 (CPU target 70%) :

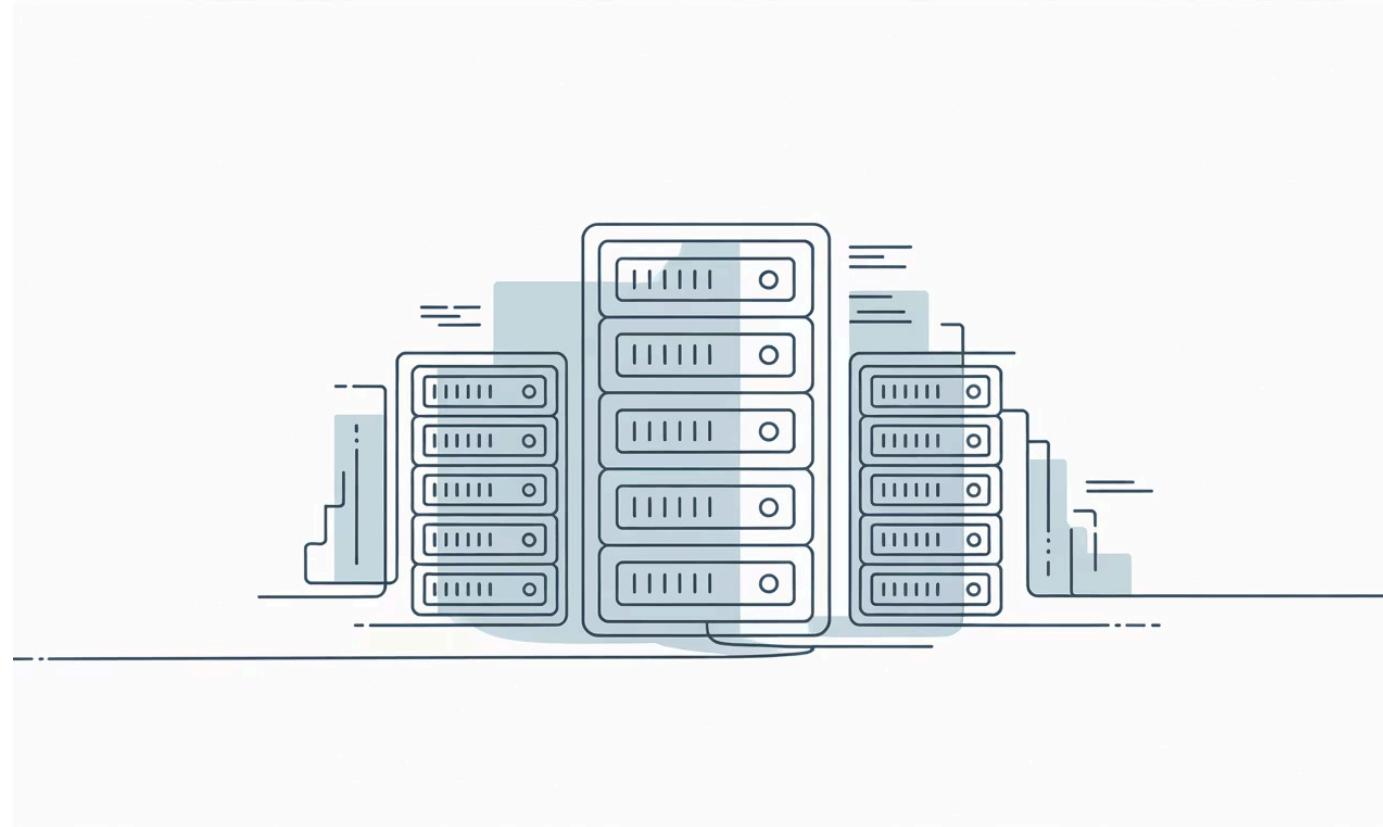
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-hpa
  namespace: prod
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
    target:
      type: Utilization
      averageUtilization: 70
```

Cette configuration surveille l'utilisation CPU du Deployment "web" et ajuste automatiquement le nombre de réplicas entre 2 et 10 pour maintenir une utilisation moyenne de 70%.

Points clés pour l'autoscaling

Contenu sur les bonnes pratiques et considérations importantes pour l'autoscaling de vos applications Kubernetes.

- Le HPA se base par défaut sur la moyenne des Pods cibles
- Les requests influencent le calcul d'utilisation CPU (% de request)
- Coupler HPA avec des probes, des budgets de perturbation et des contraintes topologiques pour la stabilité
- Définir des requests CPU appropriées pour un calcul précis
- Utiliser des PodDisruptionBudgets pour éviter les interruptions
- Surveiller les métriques pour ajuster les seuils
- Tester le comportement lors des pics de charge



Service Discovery (variables d'environnement, DNS)

Variables d'environnement:

Kubernetes injecte pour chaque Service des variables de type `NOM_SERVICE_SERVICE_HOST` et `NOM_SERVICE_SERVICE_PORT` dans les Pods créés après le Service.

Adapté à des applications simples; dépend de l'ordre de création.

DNS interne (recommandé):

Chaque Service est résolu par CoreDNS.

Nommage: `service.namespace.svc.cluster.local`

Ex: `api.prod.svc.cluster.local:80`

Headless Service (`clusterIP: None`) résout directement vers les Pod IPs (utile pour clients stateful).

Découverte entre namespaces:

- Toujours préciser le namespace ou utiliser un FQDN.
- Éviter les couplages durs en exposant des noms stables.

Namespaces, ResourceQuotas et LimitRanges



Namespaces

Isolation logique pour séparer environnements, équipes ou produits.



ResourceQuota

Plafonds de consommation et de nombres d'objets par namespace.



LimitRange

Valeurs par défaut et maximums pour requests/limits des Pods/containers.

Exemple de ResourceQuota

Exemple ResourceQuota :

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: prod
spec:
  hard:
    requests.cpu: "8"
    requests.memory: "16Gi"
    limits.cpu: "16"
    limits.memory: "32Gi"
    pods: "200"
    persistentvolumeclaims: "50"
    services.loadbalancers: "5"
```

Explication des limites ResourceQuota

Contenu explicatif sur les limites définies :

Cette configuration limite les ressources dans le namespace "prod" :

1	2	3
<p>Ressources CPU et mémoire</p> <ul style="list-style-type: none">• 8 CPU et 16Gi de mémoire en requests (garantis)• 16 CPU et 32Gi en limits (maximum autorisé)	<p>Objets Kubernetes</p> <ul style="list-style-type: none">• Maximum 200 pods• Maximum 50 PersistentVolumeClaims (PVC)• Maximum 5 LoadBalancers	<p>Impact</p> <ul style="list-style-type: none">• Empêche l'épuisement des ressources du cluster• Force les équipes à dimensionner correctement leurs applications• Permet une allocation équitable entre les namespaces



Exemple de LimitRange

Voici un exemple de configuration LimitRange :

```
apiVersion: v1
kind: LimitRange
metadata:
  name: defaults
  namespace: prod
spec:
  limits:
    - type: Container
      default:
        cpu: "500m"
        memory: "512Mi"
      defaultRequest:
        cpu: "200m"
        memory: "256Mi"
      max:
        cpu: "2"
        memory: "2Gi"
      min:
        cpu: "100m"
        memory: "128Mi"
```

Bonnes pratiques pour la gestion des ressources

Contenu sur les recommandations importantes :

Bonnes pratiques essentielles :

Appliquer un LimitRange pour garantir des valeurs par défaut cohérentes

Définir des quotas pour prévenir l'épuisement des ressources partagées

Segmenter par namespaces afin de limiter la portée RBAC et les effets de bord

Recommandations supplémentaires :

- Surveiller régulièrement l'utilisation des ressources
- Documenter les politiques de ressources pour les équipes de développement
- Ajuster les limites en fonction des besoins réels des applications
- Mettre en place des alertes en cas de dépassement des quotas

Impact positif :

Stabilité

Meilleure stabilité du cluster

Allocation

Allocation équitable des ressources

Prévention

Prévention des applications "gourmandes"

Troubleshooting

Facilitation du troubleshooting



RBAC (Role-Based Access Control)

RBAC (Role-Based Access Control):

- **Role:** permissions au niveau d'un namespace.
- **ClusterRole:** permissions au niveau cluster ou réutilisables dans plusieurs namespaces.
- **RoleBinding/ClusterRoleBinding:** lient des rôles à des sujets (users, groups, ServiceAccounts).

Exemple Role + Binding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: dev-read
  namespace: prod
rules:
- apiGroups: [""]
  resources: ["pods","services","endpoints"]
  verbs: ["get","list","watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-read-bind
  namespace: prod
subjects:
- kind: User
  name: alice
roleRef:
  kind: Role
  name: dev-read
  apiGroup: rbac.authorization.k8s.io
```

ServiceAccounts

ServiceAccounts:

Identité des Pods vis-à-vis de l'API.

Associer une SA dédiée par application; restreindre ses permissions via RBAC.

Exemple:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: web-sa
  namespace: prod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: prod
spec:
  template:
    spec:
      serviceAccountName: web-sa
```

Pod Security

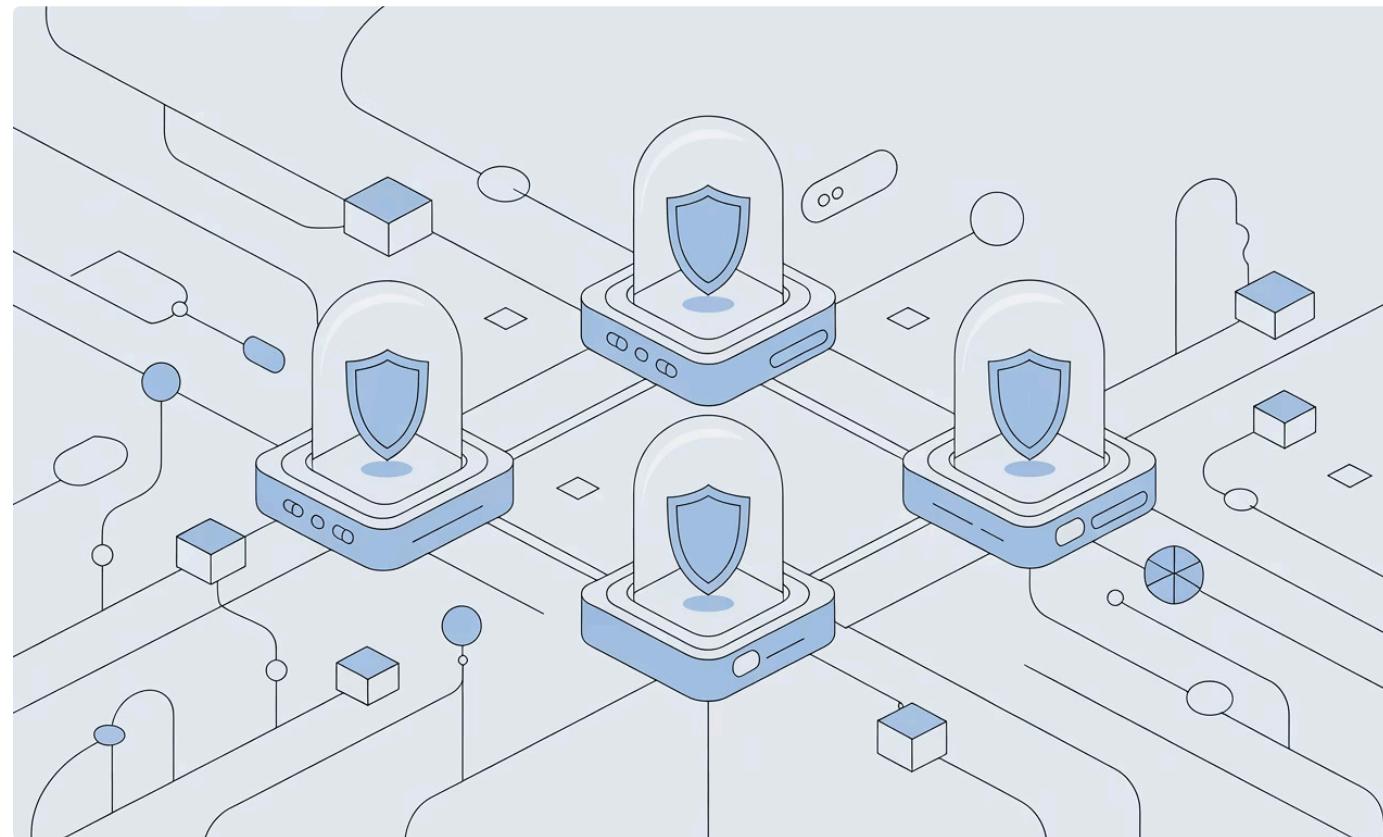
Pod Security (niveau de sécurité des Pods):

Via labels de namespace et securityContext:

Réduire les privilèges: runAsNonRoot, readOnlyRootFilesystem, drop CAPs, éviter hostPID/hostNetwork.

Exemple securityContext (survol):

```
securityContext:  
  runAsNonRoot: true  
  allowPrivilegeEscalation: false  
  capabilities:  
    drop: ["ALL"]
```



Haute disponibilité (HA) applicative

HA applicative:

Réplicas multiples

Réplicas ≥ 2 pour éviter le point unique de défaillance.

Répartition

Répartition via anti-affinité ou TopologySpreadConstraints pour éviter la co-localisation.

PodDisruptionBudget

PodDisruptionBudget (PDB) pour limiter les indisponibilités lors de perturbations contrôlées.

Exemple PDB:

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: web-pdb
  namespace: prod
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: web
```

TopologySpreadConstraints (dispersion):

```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: "kubernetes.io/hostname"
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      app: web
```

Haute disponibilité (HA) cluster

HA cluster (survol):

RéPLICATION DU PLAN DE CONTRÔLE

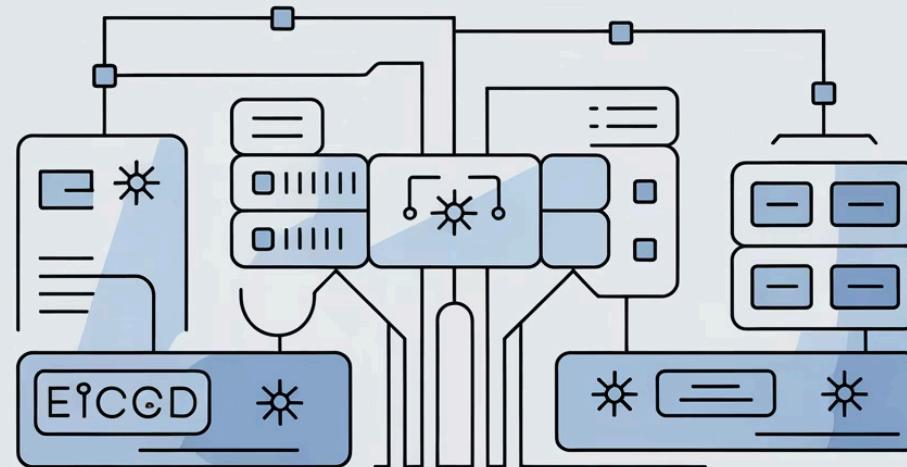
Répliquer le plan de contrôle et etcd, répartir sur plusieurs nœuds/zones.

STOCKAGE ET CNI RÉSILIENTS

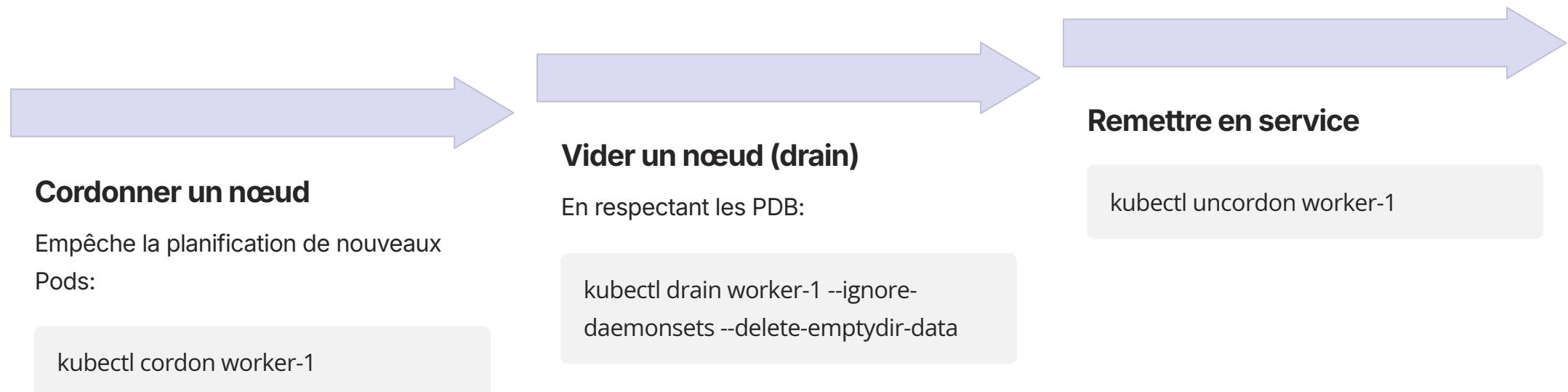
Utiliser un stockage et un CNI tolérants aux pannes.

SURVEILLANCE ACTIVE

Surveiller l'état du cluster (metrics, logs, alertes).



Mode maintenance et opérations sûres



Gestion avancée des opérations

Budgets de perturbation:

Définir des PDB pour vos workloads critiques afin que drain/updates ne réduisent pas trop la capacité disponible.

Rollout contrôlé:

Pauser un rollout:

```
kubectl rollout pause deploy/web
```

Reprendre:

```
kubectl rollout resume deploy/web
```

Revenir en arrière:

```
kubectl rollout undo deploy/web
```

Gestion des arrêts et isolement

Gestion des interruptions involontaires:

1

Probes robustes et timeouts raisonnables.

2

Readiness avant liveness pour éviter des redémarrages prématuress.

3

Utiliser terminationGracePeriodSeconds et preStop hooks pour un arrêt propre.

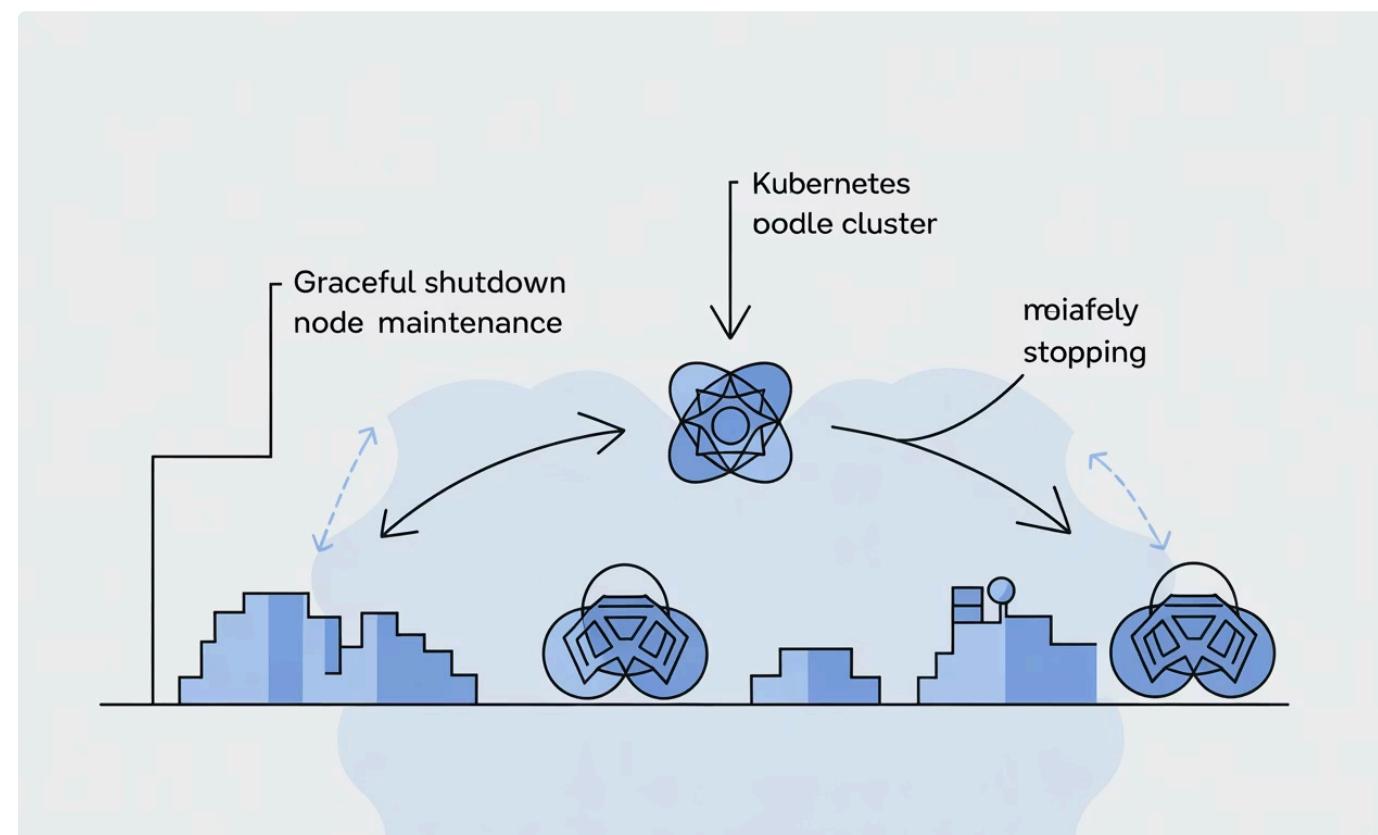
Exemple preStop + délai:

```
spec:  
terminationGracePeriodSeconds: 30  
containers:  
- name: web  
image: ghcr.io/org/web:1.0.0  
lifecycle:  
preStop:  
exec:  
command: ["/bin/sh","-c","sleep 10"]
```

Taints/Tolerations pour évacuer/segmenter:

Appliquer temporairement un taint pour isoler un nœud en maintenance.

Seuls les Pods tolérants continueront d'y être planifiés.



Observabilité et fiabilité opérationnelle (survol)



Journaux et événements

kubectl logs, kubectl describe, kubectl get events -A.



Métriques

Metrics Server pour HPA; solution de monitoring pour métriques d'application.



Traces

Intégrer un traçage distribué si nécessaire pour diagnostiquer les latences.



Budgets d'erreur et SLO

Définir des objectifs de disponibilité et surveiller les écarts.

Récapitulatif

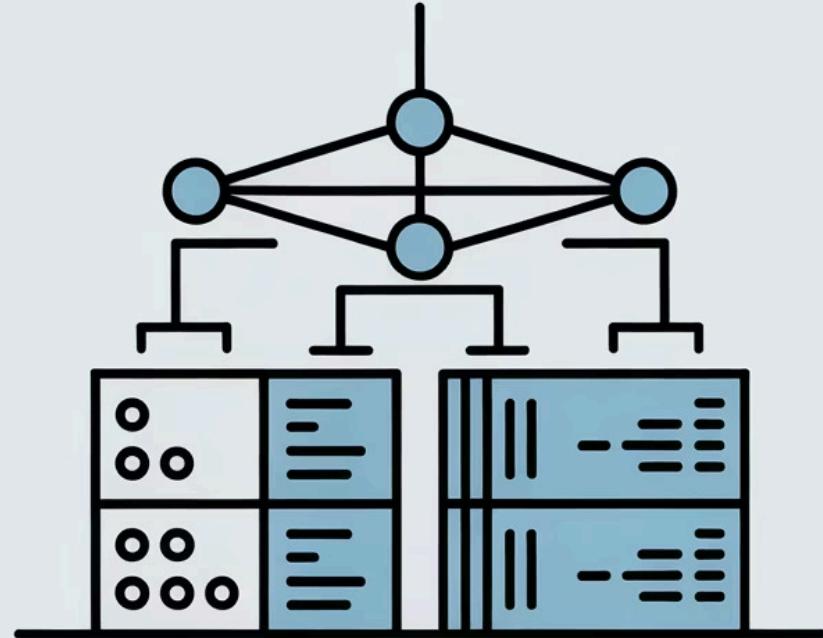
L'Ingress (ex. Traefik) permet un routage HTTP/HTTPS souple au-dessus des Services.		Requests/limits structurent la planification et déterminent la QoS; des valeurs par défaut et des quotas protègent le cluster.		Le HPA ajuste dynamiquement les réplicas selon les métriques.
La découverte de services s'appuie sur des variables d'environnement et surtout le DNS interne.		Namespaces, ResourceQuota et LimitRange organisent l'isolation et la gouvernance des ressources.		RBAC et ServiceAccounts contrôlent précisément les permissions.
La haute disponibilité repose sur la redondance, la dispersion topologique et les PDB; le mode maintenance s'appuie sur cordon/drain/uncordon et des rollouts maîtrisés.				

Déployer un cluster Kubernetes

Objectifs du chapitre

- Préparer proprement des nœuds Linux pour Kubernetes (runtime, modules noyau, sysctl, réseau).
- Déployer un cluster minimal "production-like" avec kubeadm et containerd.
- Installer un addon réseau (CNI) moderne.
- Lier des nœuds au cluster et configurer kubectl pour les administrateurs.
- Poser les bases d'administration: sauvegardes etcd, upgrades, certificats, surveillance.

Astuce: adaptez les versions à votre contexte. Dans les exemples, remplacez vX.Y.Z par la version validée par votre équipe.



Préparation des nœuds

Pré-requis (tous les nœuds: control-plane et workers):

- OS: distributions courantes (Ubuntu 22.04 LTS, Debian 12, RHEL 9...). Mêmes versions/patchlevels si possible.
- CPU/RAM: min. 2 vCPU/4 GiB pour un control-plane d'atelier; 1 vCPU/2 GiB sur un worker léger.
- Résolution DNS et NTP correctes entre nœuds; adresses IP stables.
- Ports ouverts (voir tableau plus bas) et MTU cohérente sur tout le chemin réseau.

Étapes typiques (root ou sudo):

1. Désactiver le swap et le rendre persistant:

```
swapoff -a  
sed -ri 's/^\\s*([^#].*\$swap\\s)/#\\1/' /etc/fstab
```

2. Modules noyau et paramètres réseau:

```
modprobe overlay  
modprobe br_netfilter  
  
cat <<EOF >/etc/modules-load.d/k8s.conf  
overlay  
br_netfilter  
EOF  
  
cat <<EOF >/etc/sysctl.d/99-kubernetes-cri.conf  
net.bridge.bridge-nf-call-iptables=1  
net.bridge.bridge-nf-call-ip6tables=1  
net.ipv4.ip_forward=1  
EOF  
  
sysctl --system
```

Installation des composants Kubernetes

3. Installer containerd (CRI):

Ubuntu/Debian (ex. via apt):

```
apt-get update && apt-get install -y containerd
```

Configurer containerd avec le cgroup driver systemd:

```
containerd config default >/etc/containerd/config.toml  
sed -ri 's/SystemdCgroup = false/SystemdCgroup = true/' /etc/containerd/config.toml  
systemctl enable --now containerd
```

4. Installer kubeadm, kubelet, kubectl:

Ubuntu/Debian:

```
apt-get update && apt-get install -y apt-transport-https ca-certificates curl  
curl -fsSL https://pkgs.k8s.io/core:/stable:/vX.Y/deb/Release.key | gpg --dearmor -o /etc/apt/trusted.gpg.d/kubernetes.gpg  
echo "deb https://pkgs.k8s.io/core:/stable:/vX.Y/deb/ /" >/etc/apt/sources.list.d/kubernetes.list  
apt-get update && apt-get install -y kubelet kubeadm kubectl  
systemctl enable kubelet
```

RHEL/CentOS/Alma/Rocky: utilisez le dépôt pkgs.k8s.io équivalent (rpm) et dnf/yum.

5. Horloge et DNS:

- Synchronisation NTP active (chrony/systemd-timesyncd).
- /etc/hosts ou DNS d'entreprise renseigné pour tous les nœuds.

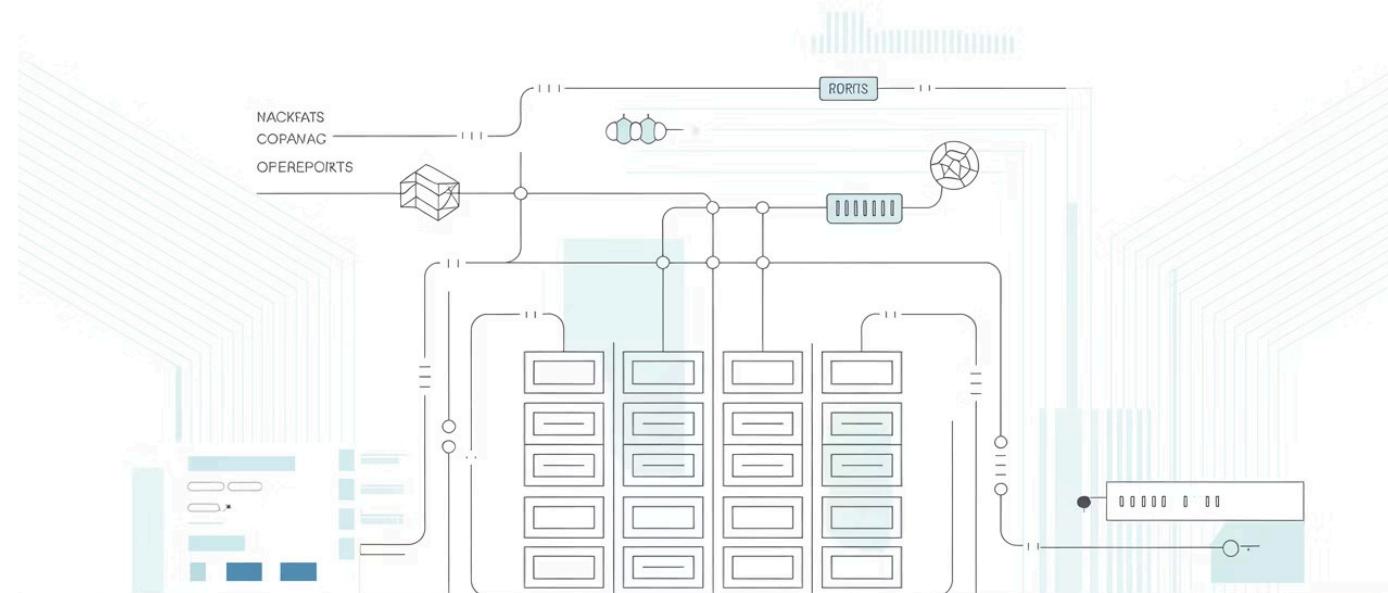
Configuration réseau et sécurité

6. Pare-feu/ports nécessaires (à adapter si firewalld/iptables actifs):

Composant	split the slide	Rôle
API Server	split the slide	Entrée API
etcd (CP)	split the slide	Quorum/Client
Kubelet	split the slide	API kubelet
Scheduler	split the slide	Webhook interne
Controller-Manager	split the slide	Webhook interne
NodePort	split the slide	Exposition L4
CNI (ex Cilium/Calico)	split the slide	Dataplane

Points importants:

- Ouvrir ces ports entre les nœuds du cluster
- Adapter la configuration selon votre pare-feu (firewalld, iptables, ufw)
- Vérifier la MTU cohérente sur tout le chemin réseau
- S'assurer que les nœuds peuvent communiquer entre eux



Configuration système et modules noyau

Configuration des modules noyau et paramètres système requis:

Modules noyau nécessaires:

```
# Charger les modules au démarrage
cat <<EOF | tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter
```

Paramètres sysctl pour Kubernetes:

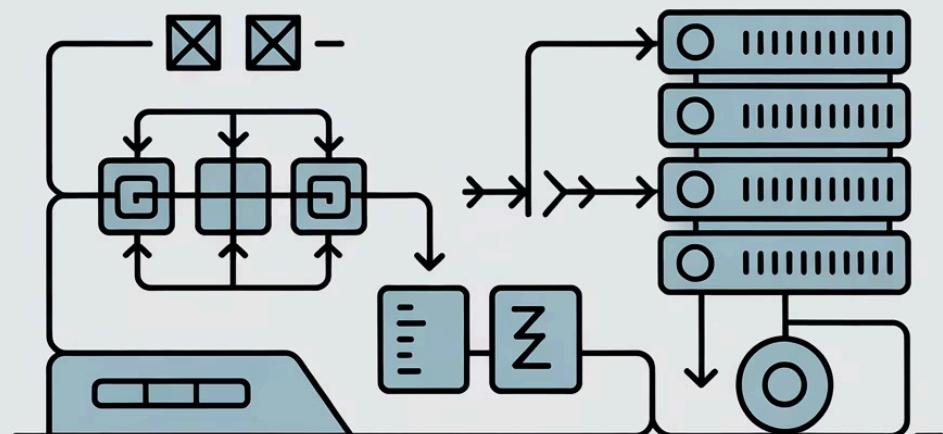
```
# Configuration réseau pour Kubernetes
cat <<EOF | tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward      = 1
EOF

sysctl --system
```

Vérification:

```
# Vérifier que les modules sont chargés
lsmod | grep br_netfilter
lsmod | grep overlay

# Vérifier les paramètres sysctl
sysctl net.bridge.bridge-nf-call-iptables net.bridge.bridge-nf-call-ip6tables net.ipv4.ip_forward
```



Déploiement d'un cluster "minimum" conforme aux bonnes pratiques

Nous utilisons kubeadm pour initialiser un control-plane et définir des sous-réseaux explicites pour Pods et Services. On force le cgroup driver systemd et le CRI containerd.

Fichier kubeadm-config.yaml (sur le nœud control-plane principal):

```
apiVersion: kubeadm.k8s.io/v1beta4
kind: InitConfiguration
nodeRegistration:
  criSocket: unix:///run/containerd/containerd.sock
  kubeletExtraArgs:
    cloud-provider: "none"
---
apiVersion: kubeadm.k8s.io/v1beta4
kind: ClusterConfiguration
kubernetesVersion: "vX.Y.Z"
clusterName: "uoc-cluster"
controlPlaneEndpoint: "cp.example.local:6443" # ou IP virtuelle en HA
networking:
  podSubnet: "10.244.0.0/16" # adapté au CNI choisi
  serviceSubnet: "10.96.0.0/12"
  dnsDomain: "cluster.local"
controllerManager:
  extraArgs:
    bind-address: "0.0.0.0"
scheduler:
  extraArgs:
    bind-address: "0.0.0.0"
---
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "ipvs"
```

Initialisation du cluster

Initialiser le cluster avec kubeadm:

Commande d'initialisation:

```
kubeadm init --config=kubeadm-config.yaml
```

Configuration kubectl pour l'utilisateur:

```
mkdir -p $HOME/.kube  
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Récupérer la commande join:

```
# La commande join est affichée à la fin de kubeadm init  
# Ou la régénérer avec:  
kubeadm token create --print-join-command
```

Vérification initiale:

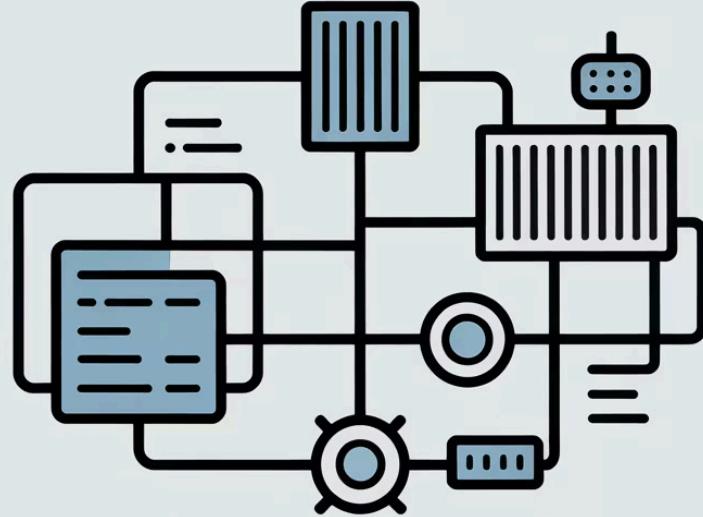
```
kubectl get nodes  
kubectl get pods -n kube-system
```

Points importants:

- Sauvegarder le fichier admin.conf
- Noter la commande join pour les workers
- Le cluster n'est pas encore fonctionnel sans CNI

Déploiement du CNI et validation

Choisissez un CNI adapté à vos exigences (NetworkPolicy, eBPF, performance, encryption). Trois options populaires:



Cilium (eBPF):

Prérequis: noyau récent; pas de kube-proxy si vous activez kube-proxy replacement.

Installation rapide via CLI:

```
curl -L --remote-name https://github.com/cilium/cilium-cll/releases/latest/download/cilium-linux-amd64.tar.gz  
tar xzf cilium-linux-amd64.tar.gz -C /usr/local/bin  
cilium install --version vX.Y.Z  
cilium status --wait
```

Calico (BGP/felix):

Manifest standard (podSubnet 192.168.0.0/16 ou adaptez au vôtre):

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Flannel (simple):

Manifest:

```
kubectl apply -f https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml
```

Important:

- Le podSubnet défini dans kubeadm doit correspondre au CNI (ou inversement).
- Attendez que kube-dns/CoreDNS et le CNI soient "Ready" avant d'ajouter des workloads.

Validation rapide:

```
kubectl get nodes -o wide  
kubectl get pods -A -o wide
```

Ajout de nœuds au cluster

Lier des nœuds au cluster

Sur chaque worker (et sur d'autres control-planes si HA):

1. Assurez la même préparation (swap, modules, containerd, kubelet).
2. Exécutez la commande join fournie par kubeadm, par ex.:

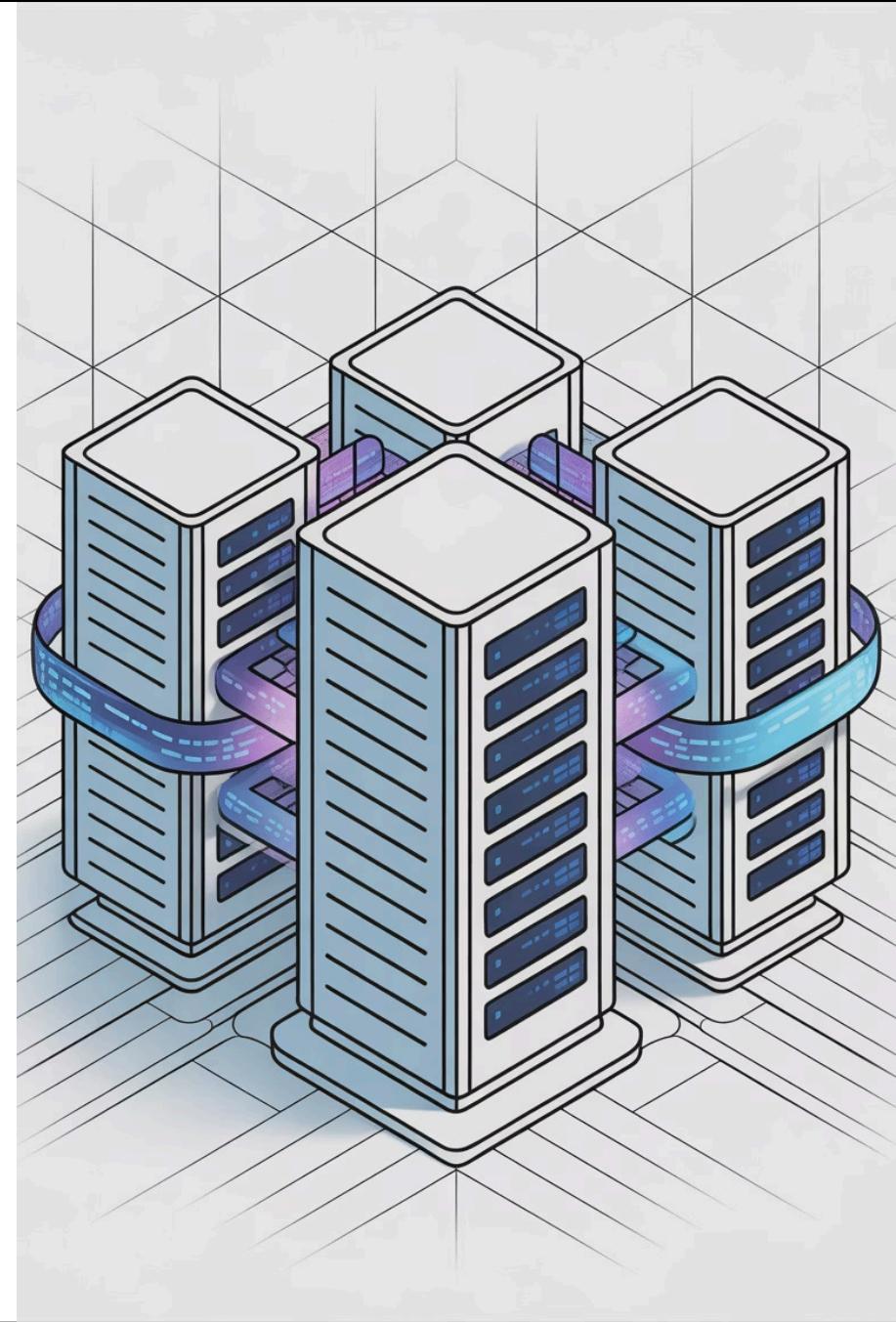
```
kubeadm join cp.example.local:6443 --token <token> \  
--discovery-token-ca-cert-hash sha256:<hash>
```

Pour joindre un nœud en tant que control-plane supplémentaire:

```
kubeadm join cp.example.local:6443 --token <token> \  
--discovery-token-ca-cert-hash sha256:<hash> --control-plane
```

Régénérer un token si expiré (sur un control-plane):

```
kubeadm token create --print-join-command
```



Vérification du cluster

Vérifier l'état du cluster après l'ajout des nœuds:

Commandes de vérification:

```
kubectl get nodes -o wide  
kubectl get pods -A  
kubectl cluster-info
```

Points à vérifier:

- Tous les nœuds sont en état "Ready"
- Les pods système (kube-system) sont en cours d'exécution
- Le CNI est correctement déployé

Test de connectivité et validation réseau

Valider le bon fonctionnement du réseau du cluster:

Tests de connectivité essentiels:

Test de résolution DNS:

```
kubectl run test-dns --image=busybox --rm -it -- nslookup kubernetes.default.svc.cluster.local
```

Test de communication inter-pods:

Créer un pod de test

```
kubectl run test-pod --image=nginx --port=80
```

Tester depuis un autre pod

```
kubectl run test-client --image=busybox --rm -it -- wget -qO- http://test-pod
```

Validation du CNI:

Vérifier que chaque pod a une IP unique

```
kubectl get pods -o wide --all-namespaces
```

Tester la connectivité entre nœuds

```
kubectl get nodes -o wide  
ping <node-ip>
```

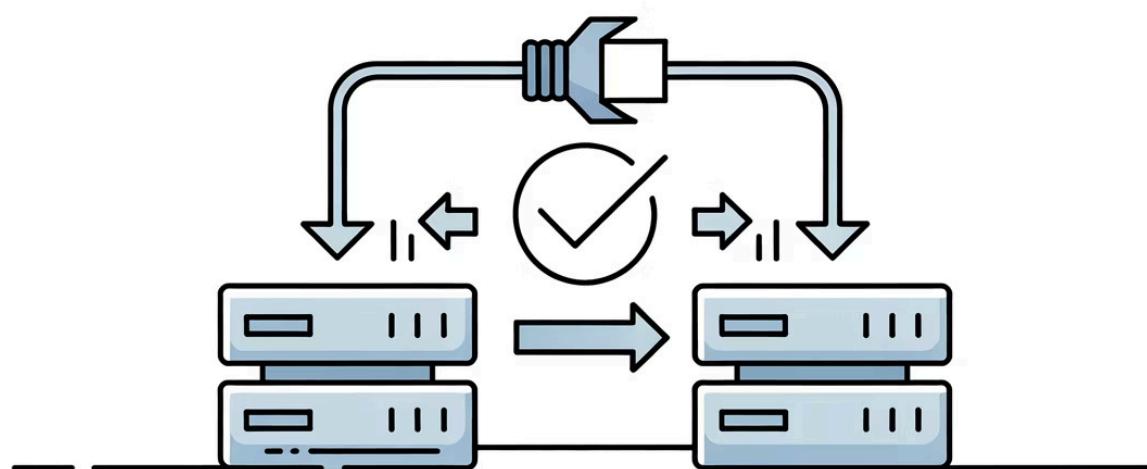
Test des Services:

Créer un service de test

```
kubectl expose pod test-pod --port=80 --target-port=80
```

Tester l'accès au service

```
kubectl run test-service --image=busybox --rm -it -- wget -qO- http://test-pod:80
```



Dépannage et résolution de problèmes

Résoudre les problèmes courants lors du déploiement:

Problèmes fréquents et solutions:

Nœuds en état NotReady:

```
# Vérifier l'état kubelet  
systemctl status kubelet  
journalctl -u kubelet -f
```

```
# Vérifier la configuration CNI  
kubectl get pods -n kube-system
```

Pods en état Pending:

```
# Vérifier les événements  
kubectl describe pod  
kubectl get events --sort-by=.lastTimestamp
```

```
# Vérifier les ressources disponibles  
kubectl top nodes
```

Problèmes réseau:

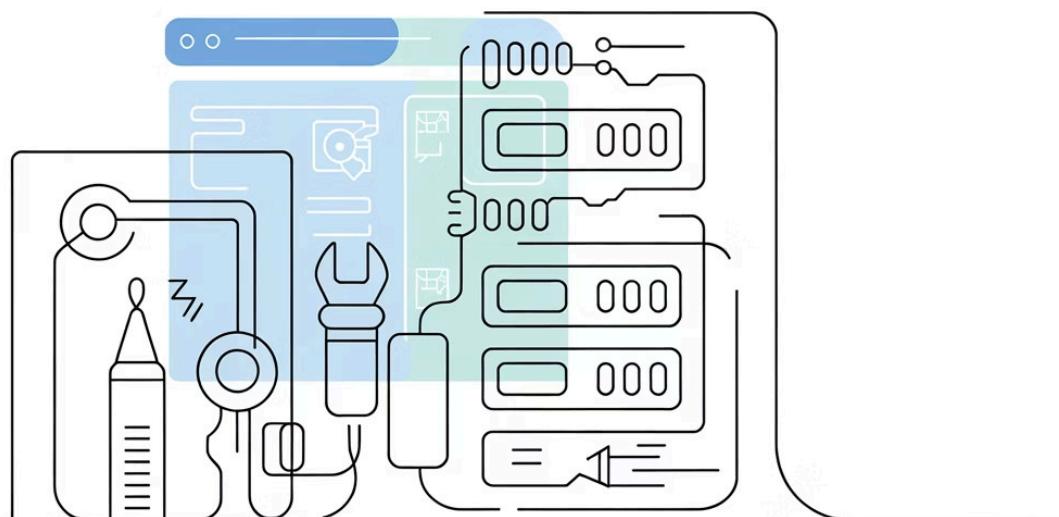
```
# Tester la connectivité DNS  
nslookup kubernetes.default.svc.cluster.local
```

```
# Vérifier les routes réseau  
ip route show
```

Certificats expirés:

```
# Vérifier l'expiration des certificats  
kubeadm certs check-expiration
```

```
# Renouveler les certificats  
kubeadm certs renew all
```



Récapitulatif



Une préparation homogène des nœuds (swap off, cgroups systemd, sysctl, containerd) est la base de la stabilité.



kubeadm permet d'initialiser un cluster propre avec des sous-réseaux explicites, kube-proxy en IPVS et un endpoint HA optionnel.



Le choix du CNI (Cilium/Calico/Flannel) doit s'aligner sur vos besoins en performances et politiques réseau.



Les nœuds rejoignent via kubeadm join; kubectl est distribué via kubeconfig adaptés (humains/CI).



L'administration implique sauvegardes etcd, gestion des certificats, upgrades séquencés et observabilité.



Pour la production: pensez HA du control-plane, encryption des secrets et politiques de sécurité strictes.