

Formation aux outils de versioning et d'automatisation d'intégration et de déploiement

Helm et ArgoCD

Sylvain Allemand

Git → CI + Trivy → Helm → ArgoCD → Kubernetes

- ▶ Trivy : détecter vulnérabilités et mauvaises configurations.
- ▶ Helm : empaqueter et paramétrer une application Kubernetes.
- ▶ ArgoCD : faire du déploiement GitOps à partir d'une source de vérité Git.

Kubernetes en 10 minutes

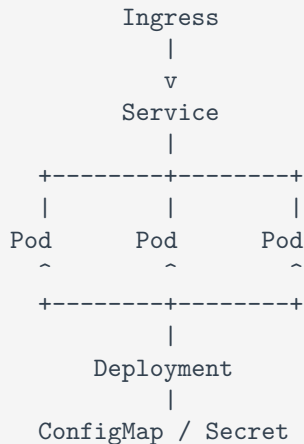
Kubernetes est un orchestrateur de conteneurs : **décrire l'état souhaité** → Kubernetes maintient cet état.

- ▶ Le cluster exécute des applications conteneurisées.
- ▶ Les ressources Kubernetes sont décrites en YAML.
- ▶ L'API Kubernetes est la source d'interaction avec le cluster.
- ▶ 'kubectl' est l'outil en ligne de commande le plus courant. Dans la PLM, c'est plutôt 'oc'.

Le vocabulaire minimal

Objet	Rôle principal
Pod	Plus petite unité d'exécution ; contient un ou plusieurs conteneurs
Deployment	Maintient un nombre de Pods et gère les mises à jour
Service	Fournit un point d'accès vers des Pods
ConfigMap	Stocke de la configuration non sensible
Secret	Stocke des données sensibles
Ingress	Décrit l'exposition HTTP/HTTPS vers des Services
Namespace	Permet d'isoler logiquement les ressources

Comment les objets s'articulent ?



Le Deployment gère les Pods ; le Service sélectionne les Pods ; les autres objets fournissent configuration et exposition.

Un objet Kubernetes est surtout du YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: nginx:1.27
```

Les commandes kubectl à connaître

```
kubectl get pods  
kubectl get deployments  
kubectl get svc  
kubectl describe pod <nom>  
kubectl logs <pod>  
kubectl apply -f manifest.yaml  
kubectl delete -f manifest.yaml
```

Lien avec Helm : Helm va générer des manifests Kubernetes similaires à ceux que vous pourriez écrire à la main.

Anatomie d'un chart

```
my-app/  
|-- Chart.yaml  
|-- values.yaml  
'-- templates/  
    |-- _helpers.tpl  
    |-- deployment.yaml  
    |-- service.yaml  
'-- ingress.yaml
```

- ▶ 'Chart.yaml' : métadonnées et version du chart.
- ▶ 'values.yaml' : paramètres par défaut.
- ▶ 'templates/' : manifests Kubernetes paramétrés.

Templates et values

Template :

```
replicas: {{ .Values.replicaCount }}  
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

Valeurs :

```
replicaCount: 2  
image:  
  repository: nginx  
  tag: "1.27.1"
```

Le même template peut être réutilisé pour différents environnements (dev, staging et production).

Pourquoi Helm ?

Sans Helm

- ▶ beaucoup de YAML ;
- ▶ duplication entre environnements ;
- ▶ gestion manuelle des versions ;
- ▶ paramètres dispersés.

Avec Helm

- ▶ un chart versionné ;
- ▶ templates réutilisables ;
- ▶ values séparées ;
- ▶ charts traçables.

Commandes Helm essentielles

```
helm create my-app  
helm lint my-app  
helm template my-app ./my-app  
helm install my-app ./my-app  
helm upgrade my-app ./my-app  
helm upgrade --install my-app ./my-app  
helm history my-app  
helm rollback my-app 1  
helm uninstall my-app
```

Helm : ce qu'il faut comprendre

- ▶ Une **chart** est le package source.
- ▶ Une **release** est une instance installée dans Kubernetes.
- ▶ 'values' contrôlent le comportement de la chart.
- ▶ Le rendu final reste du YAML Kubernetes.
- ▶ Helm prépare et déploie les ressources.
- ▶ Helm utilise le contexte kubernetes disponible (commande kubectl/oc).

Avant de passer au TP Helm

Helm permet de déployer une application. Mais comment savoir si l'image que nous sommes en train de déployer contient des vulnérabilités ?

Pourquoi intégrer Trivy ?

Une application peut fonctionner correctement tout en embarquant des vulnérabilités.

Trivy permet notamment de rechercher :

- ▶ vulnérabilités dans les images ;
- ▶ secrets détectables ;
- ▶ erreurs ou mauvaises configurations selon le mode de scan ;
- ▶ problèmes dans un filesystem / repository.

Trivy : premiers scans

```
# Image container
trivy image nginx:1.27

# Dpt / filesystem
trivy fs .

# Configuration / manifests selon le besoin
trivy config ./helm/my-app
```

Idée importante : scanner tôt, avant que le déploiement ne devienne le problème de quelqu'un d'autre.

Le security gate dans une pipeline CI/CD

```
trivy image \  
  --severity HIGH,CRITICAL \  
  --exit-code 1 \  
  myimage:1.0
```

- ▶ '–severity' cible les niveaux à traiter.
- ▶ '–exit-code 1' peut faire échouer la CI.
- ▶ Le pipeline peut ainsi empêcher la promotion d'une image jugée non conforme.

TP 01 — Helm + Trivy

`https://plmlab.math.cnrs.fr/anf-mathrice-2026/helm`

Le problème du déploiement manuel

Développer → 'helm upgrade' → Kubernetes

Questions :

- ▶ Qui a effectué le changement ?
- ▶ Comment détecter une modification manuelle du cluster ?
- ▶ Quelle est la source de vérité ?

GitOps avec ArgoCD

Git repository → ArgoCD → Kubernetes

ArgoCD compare Git et Kubernetes puis réconcilie.

- ▶ Git = **desired state**
- ▶ Git garde l'historique.
- ▶ ArgoCD détecte le drift.
- ▶ La synchronisation peut être manuelle ou automatisée.

Les objets d'ArgoCD

ArgoCD s'appuie notamment sur :

- ▶ **Application** : décrit un déploiement ;
- ▶ **AppProject** : définit son périmètre et ses permissions ;
- ▶ **ApplicationSet** : génère plusieurs Applications ;
- ▶ **Repository / Cluster** : définit les sources et destinations accessibles.

Application est l'objet central du déploiement.

L'objet Application

Une Application décrit le lien entre une source Git et une destination Kubernetes.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app
spec:
  project: default
  source:
    repoURL: https://plmlab.math/projet.git
    path: helm/my-app
  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
  syncPolicy:
    automated: {}
```

- ▶ source : **quoi déployer ?**
- ▶ destination : **où déployer ?**
- ▶ syncPolicy : **comment synchroniser ?**
- ▶ project : **dans quel périmètre ?**

Application : le cycle de réconciliation

1. ArgoCD lit la source définie dans l'Application.
2. Les manifests sont générés, par exemple avec Helm.
3. ArgoCD compare l'état désiré avec l'état réel.
4. Un écart correspond à un **drift**.
5. ArgoCD peut effectuer la synchronisation automatiquement ou attendre une action manuelle.

Application ne contient pas les ressources Kubernetes elles-mêmes : elle décrit **comment les obtenir et où les appliquer.**

L'objet AppProject

Un AppProject permet de définir un **périmètre de confiance** pour les Applications.

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: platform
spec:
  sourceRepos:
    - 'https://plmlab.math.cnrs.fr/plmteam/*'
  destinations:
    - namespace: my-app
      server: https://kubernetes.default.svc
  clusterResourceWhitelist:
    - group: ''
      kind: Namespace
```

Il permet notamment de contrôler :

- ▶ les repositories Git autorisés ;
- ▶ les clusters et namespaces de destination ;
- ▶ les types de ressources Kubernetes autorisés ;
- ▶ les permissions associées aux Applications.

AppProject : pourquoi ?

AppProject = frontière de sécurité

Sans projet :

- ▶ une Application pourrait avoir un périmètre trop large ;
- ▶ les sources et destinations sont moins facilement séparées.

Avec des projets :

- ▶ projet dev → namespaces de développement ;
- ▶ projet prod → namespaces de production ;
- ▶ repositories autorisés explicitement ;
- ▶ ressources Kubernetes autorisées explicitement.

Application → “déployer quoi et où ?”

AppProject → “dans quel périmètre est-ce autorisé ?”

ApplicationSet

ApplicationSet = générateur d'Applications

Au lieu de créer manuellement :

```
Application-dev  
Application-staging  
Application-prod  
Application-client-A  
Application-client-B
```

Un ApplicationSet peut générer ces Applications à partir d'un **template** et d'un **generator**.

$$\text{ApplicationSet} \rightarrow \begin{cases} \text{Application A} \\ \text{Application B} \\ \text{Application C} \end{cases}$$

Comment les objets s'articulent



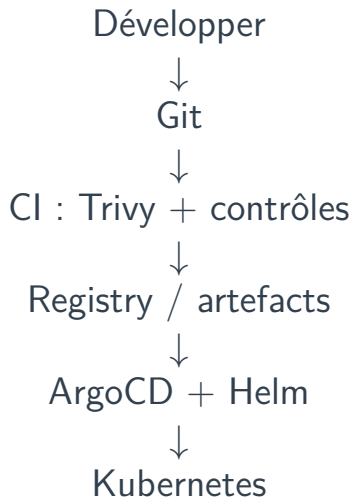
ArgoCD peut exploiter le chart Helm sans que l'utilisateur exécute lui-même 'helm upgrade'.

À retenir

Outil	Rôle
Git	Versionner la source de vérité
Helm	Packager et paramétrer Kubernetes
Trivy	Contrôler les risques de sécurité
ArgoCD	Réconcilier Git et Kubernetes
Kubernetes	Exécuter l'application

Ces outils sont complémentaires.

Architecture finale



TP 02 — GitOps avec ArgoCD

`https://plmlab.math.cnrs.fr/anf-mathrice-2026/argocd`