Projet de Migration DevOps - 40 Microservices

Contexte du Projet

Migration de 40 microservices avec ArgoCD, Jenkins et déploiement dans Kubernetes. Équipes impliquées : Dev, DevOps, Cloud Engineers.

🎯 Organisation de l'Équipe DevOps (4 personnes)

Team Leader DevOps - Rôle & Responsabilités

Missions principales :

- **Architecture technique** : Définir la stratégie de migration CI/CD
- **Coordination équipes** : Interface avec Dev, Cloud Engineers, Management
- **Gouvernance** : Standards, politiques de sécurité, RBAC global
- **Pilotage projet** : Planning, risques, reporting

Responsabilités techniques :

- Design des pipelines Jenkins (templates, shared libraries)
- Architecture ArgoCD (projets, ApplicationSets, sync policies)
- Définition des environnements (dev/staging/prod)
- Stratégie de rollback et disaster recovery

💼 Répartition des Tâches par DevOps Engineer

DevOps Engineer #1 - "Jenkins Specialist"

```yaml
```

Focus: CI/CD Pipelines & Build Automation

Responsabilités:

```
├── Jenkins Configuration
│   ├── Pipeline templates pour les 40 microservices
│   ├── Shared Libraries (build, test, package)
│   ├── Multi-branch pipelines
│   └── Jenkins agents (Docker, K8s)
├── Build & Test
│   ├── Dockerfiles standards par tech stack
│   ├── Scripts de test automatisés
│   ├── Security scanning (SAST/DAST)
│   └── Artifact management (Nexus/Harbor)
└── Intégration
    ├── Webhooks Git → Jenkins
    ├── Jenkins → ArgoCD (GitOps trigger)
    └── Notifications (Slack, Teams)
```

DevOps Engineer #2 - "ArgoCD & GitOps Specialist"

yaml

Focus: Deployment & GitOps

Responsabilités:

```
├── ArgoCD Setup
│   ├── Projets par équipe/domaine
│   ├── ApplicationSets pour automation
│   ├── Sync policies & health checks
│   └── Multi-cluster management
├── GitOps Repository
│   ├── Structure des repos (app configs)
│   ├── Helm charts standardisés
│   ├── Kustomize overlays par environnement
│   └── Secret management (Sealed Secrets)
└── Deployment Strategy
    ├── Progressive delivery (Canary, Blue/Green)
    ├── Rollback automatique
    └── Environment promotion
```

DevOps Engineer #3 - "Platform & RBAC Specialist"

yaml

Focus: Security, RBAC & Monitoring

Responsabilités:
├── RBAC & Security
│ ├── Jenkins: rôles par équipe/projet
│ ├── ArgoCD: projets + permissions granulaires
│ ├── Kubernetes: namespaces + NetworkPolicies
│ └── Secret rotation & vault integration
├── Monitoring & Observability
│ ├── Prometheus metrics (Jenkins + ArgoCD)
│ ├── Grafana dashboards
│ ├── AlertManager rules
│ └── Logging centralisé (ELK)
└── Platform Services
├── Service mesh (Istio/Linkerd)
├── Ingress controllers
└── Certificate management

📋 Planning de Migration (Exemple 12 semaines)

Phase 1 - Setup Infrastructure (Semaines 1-3)

Team Leader :

- Architecture globale et standards
- Coordination avec Cloud Engineers

Engineer #1 : Jenkins master + agents setup **Engineer #2 :** ArgoCD installation + configuration **Engineer #3 :** RBAC baseline + monitoring

Phase 2 - Templates & Automation (Semaines 4-6)

Team Leader :

- Validation des templates
- Formation équipes Dev

Engineer #1 :

- Pipeline templates par stack (Java, Node.js, Python)
- Shared libraries Jenkins

Engineer #2 :

- Helm charts standards
- ApplicationSets templates

Engineer #3 :

- Policies de sécurité
- Dashboards monitoring

Phase 3 - Migration par Vagues (Semaines 7-12)

Vague 1 (5 services critiques) :

- Team Leader : Coordination + support
- Engineer #1 : Migration pipelines Jenkins
- Engineer #2 : Configuration ArgoCD apps
- Engineer #3 : RBAC + monitoring par service

Vague 2-4 (35 services restants) :

- Réplication du pattern validé
- Support et troubleshooting

Stratégie RBAC Détaillée

Jenkins RBAC :

```groovy
// Par Engineer DevOps
Jenkins Admin: Team Leader + Engineer #1
Project Admin: Engineer #2, Engineer #3
Developer: Équipes Dev (read-only jobs, trigger builds)

// Par Microservice
Folder-based security:
├── payment-services/ (Payment team access)
├── user-services/ (User team access)
└── shared-libs/ (DevOps team only)
```

ArgoCD RBAC :

```yaml
```

 # Par Engineer DevOps
platform-admin: Team Leader (tous projets)
team-payment-admin: Engineers selon assignation
team-user-admin: Engineers selon assignation

 # Par Équipe Dev
team-payment-dev: sync, get sur projet payment
team-user-dev: sync, get sur projet user

🎯 Métriques & KPI par Engineer

Engineer #1 (Jenkins):

- Build success rate par microservice
- Temps de build moyen
- Pipeline failures et MTTR

Engineer #2 (ArgoCD):

- Deployment success rate
- Sync time par application
- Drift détection et correction

Engineer #3 (Platform):

- Security vulnerabilities détectées
- RBAC violations
- Platform uptime

🚀 Workflow Quotidien Type

Daily Standup (15min) :

- Team Leader : Blockers et priorités
- Chaque Engineer : Progress sur ses microservices assignés
- Sync inter-équipes (Dev feedback)

Weekly Review :

- Migration progress (combien de services migrés)
- Performance metrics
- Lessons learned et améliorations

Cette organisation permet de migrer efficacement 40 microservices en parallélisant les expertises tout en maintenant la cohérence et la qualité !

EXPLICATION DE SHARED LIBRARY POUR LES PARTIES PIPELINE CI/CD

🎯 Oui, il y aura 40 pipelines Jenkins !

MAIS pas 40 pipelines complexes écrits à la main. Voici comment ça fonctionne :

📋 Répartition Réelle

Avec Shared Libraries :

```
jenkins-shared-library/     # 1 seul repo avec la logique
├── vars/buildMicroservice.groovy
├── vars/testApplication.groovy
├── vars/packageDocker.groovy
└── vars/deployToK8s.groovy

microservice-1/          # 40 repos de microservices
├── src/                # Code applicatif
├── Dockerfile
└── Jenkinsfile         # ← 5 lignes qui appellent la shared library

microservice-2/
├── src/
└── Jenkinsfile         # ← 5 lignes identiques

... (38 autres)
```

🔍 Exemple Concret

Dans chaque microservice (40 fois) :

groovy

```groovy
// microservice-payment/Jenkinsfile (5 lignes seulement)
@Library('company-shared-library@main') _

microservicePipeline([
    serviceName: 'payment-service',
    language: 'java',
    buildTool: 'maven'
])
```

groovy

```groovy
// microservice-user/Jenkinsfile (5 lignes seulement)
@Library('company-shared-library@main') _

microservicePipeline([
    serviceName: 'user-service',
    language: 'nodejs',
    buildTool: 'npm'
])
```

La logique complexe (1 seule fois) :

```groovy
// jenkins-shared-library/vars/microservicePipeline.groovy
def call(Map params) {
    pipeline {
        agent any
        stages {
            stage('Build') {
                steps {
                    buildMicroservice(params)
                }
            }
            stage('Test') {
                steps {
                    testApplication(params)
                }
            }
            stage('Package') {
                steps {
                    packageDocker(params)
                }
            }
            stage('Deploy') {
                steps {
                    deployToK8s(params)
                }
            }
        }
    }
}
```

📊 Comparaison

❌ Sans Shared Library :

40 microservices × 100 lignes de Jenkinsfile = 4000 lignes à maintenir

✅ Avec Shared Library :

1 shared library × 200 lignes + 40 microservices × 5 lignes = 400 lignes à maintenir

🏗️ Comment Jenkins voit ça

Dans l'interface Jenkins :

```
Jenkins Dashboard:
├── 📁 payment-team/
│   ├── payment-service     # Pipeline 1
│   ├── billing-service     # Pipeline 2
│   └── fraud-service       # Pipeline 3
├── 📁 user-team/
│   ├── user-service        # Pipeline 4
│   ├── auth-service        # Pipeline 5
│   └── profile-service     # Pipeline 6
└── 📁 inventory-team/
    ├── product-service     # Pipeline 7
    ├── stock-service       # Pipeline 8
    └── catalog-service     # Pipeline 9
... (31 autres pipelines)
```

⚙️ Création Automatique

Option 1: Multibranch Pipeline par équipe

```groovy
// Scanne automatiquement tous les repos d'une équipe
payment-team-multibranch:
├── Source: GitHub Organization
├── Owner: company
├── Pattern: payment-*
└── Découvre automatiquement: payment-service, billing-service, etc.
```

Option 2: Pipeline par service

```groovy
```

```
// Un pipeline Jenkins par microservice
40 jobs Jenkins créés manuellement ou via JobDSL:
├── payment-service-pipeline
├── user-service-pipeline
├── product-service-pipeline
└── ... (37 autres)
```

🎯 Travail du DevOps Engineer #1

Il va créer :

1. **1 Shared Library** avec toute la logique complexe
2. **40 Jenkinsfiles simples** (5 lignes chacun)
3. **40 jobs Jenkins** qui pointent vers ces repos

Résultat :

- **40 pipelines Jenkins** dans l'interface
- **1 endroit** pour maintenir la logique
- **Cohérence** garantie sur tous les microservices

En gros : 40 pipelines qui UTILISENT le même code, pas 40 pipelines différents à maintenir !

BUILD ET TEST POUR DEVOPS ENGINEER #1

🐳 Dockerfiles standards par tech stack

Le DevOps Engineer #1 va créer des **templates Docker** réutilisables :

dockerfile

```
# templates/Dockerfile.java
FROM openjdk:17-jre-slim
WORKDIR /app
COPY target/*.jar app.jar
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]
```

dockerfile

```
# templates/Dockerfile.nodejs
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

dockerfile

```
# templates/Dockerfile.python
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

Pourquoi c'est important ?

- **Cohérence** : Tous les services Java utilisent la même base
- **Sécurité** : Images validées et sécurisées
- **Performance** : Optimisées pour la production

🧪 Scripts de test automatisés

Des scripts standardisés pour chaque type de test :

bash

```
# scripts/test-java.sh
#!/bin/bash
echo "🧪 Running Java tests..."

# Unit tests
mvn test -Dtest.profile=unit

# Integration tests
mvn test -Dtest.profile=integration

# Coverage report
mvn jacoco:report

# Quality gates
if [ $(cat target/site/jacoco/index.html | grep -o '[0-9]\+%' | head -1 | sed 's/%//') -lt 80 ]; then
    echo "❌ Coverage below 80%"
    exit 1
fi
```

bash

```
# scripts/test-nodejs.sh
#!/bin/bash
echo "🧪 Running Node.js tests..."

# Unit tests
npm run test:unit

# Integration tests
npm run test:integration

# E2E tests (optionnel)
npm run test:e2e

# Linting
npm run lint
```

🔒 Security scanning (SAST/DAST)

SAST (Static Analysis Security Testing)

groovy

```groovy
// Dans la pipeline
stage('Security Scan - SAST') {
    steps {
        script {
            // SonarQube pour analyse statique
            sh 'sonar-scanner -Dsonar.projectKey=${serviceName}'

            // Snyk pour vulnérabilités dépendances
            sh 'snyk test --severity-threshold=high'

            // Checkmarx ou Veracode
            sh 'checkmarx-cli scan --project ${serviceName}'
        }
    }
}
```

DAST (Dynamic Analysis Security Testing)

```groovy
stage('Security Scan - DAST') {
    steps {
        script {
            // OWASP ZAP pour tests dynamiques
            sh '''
                docker run -t owasp/zap2docker-stable zap-baseline.py \
                -t http://staging.${serviceName}.company.com \
                -r zap-report.html
            '''

            // Publish security report
            publishHTML([
                allowMissing: false,
                reportDir: '.',
                reportFiles: 'zap-report.html',
                reportName: 'Security Report'
            ])
        }
    }
}
```

📦 Artifact management (Nexus/Harbor)

Harbor (Registry Docker)

```groovy
```

```
stage('Push Docker Image') {
    steps {
        script {
            def image = docker.build("${serviceName}:${env.BUILD_NUMBER}")

            docker.withRegistry('https://harbor.company.com', 'harbor-credentials') {
                image.push()
                image.push('latest')

                // Scan de sécurité intégré Harbor
                sh "harbor-cli scan harbor.company.com/${serviceName}:${env.BUILD_NUMBER}"
            }
        }
    }
}
```

Nexus (Artifacts binaires)

```
groovy

stage('Publish Artifacts') {
    steps {
        script {
            // Pour Java - publish JAR
            sh '''
                mvn deploy \
                -DrepositoryId=nexus-releases \
                -Durl=https://nexus.company.com/repository/maven-releases/
            '''

            // Pour Node.js - publish NPM package (si library)
            sh '''
                npm config set registry https://nexus.company.com/repository/npm-internal/
                npm publish
            '''
        }
    }
}
```

🎯 Organisation concrète du DevOps Engineer #1

Structure de ses livrables :

```
jenkins-templates/
├── dockerfiles/
│   ├── Dockerfile.java
│   ├── Dockerfile.nodejs
│   └── Dockerfile.python
├── test-scripts/
│   ├── test-java.sh
│   ├── test-nodejs.sh
│   └── test-python.sh
├── security/
│   ├── sonar-quality-gates.xml
│   ├── snyk-policy.yml
│   └── zap-baseline.conf
└── shared-library/
    ├── vars/buildMicroservice.groovy
    ├── vars/testApplication.groovy
    ├── vars/securityScan.groovy
    └── vars/publishArtifacts.groovy
```

🔧 Pipeline intégrée finale

groovy

```
// Utilisation dans chaque microservice
pipeline {
    stages {
        stage('Build') {
            steps {
                buildMicroservice(language: 'java')
            }
        }
        stage('Test') {
            steps {
                testApplication(language: 'java', coverage: 80)
            }
        }
        stage('Security') {
            steps {
                securityScan(sast: true, dast: true)
            }
        }
        stage('Package') {
            steps {
                publishArtifacts(
                    dockerRegistry: 'harbor.company.com',
                    nexusRepo: 'maven-releases'
                )
            }
        }
    }
}
```

En résumé : Le DevOps Engineer #1 créé tous les "outils" standardisés que les 40 microservices vont utiliser automatiquement via les pipelines !

COMMENT LES 40 PIPELINES SONT LANCÉES ET ORGANISATION GITOPS

🚀 Comment les 40 Pipelines sont Lancées

Déclenchement des Pipelines :

1. Git Webhooks (Le plus courant)

```
Developer push code → GitHub/GitLab → Webhook → Jenkins → Lance la pipeline du service
```

2. Triggers Automatiques

```
groovy
```

```groovy
// Dans chaque Jenkinsfile
pipeline {
    triggers {
        // Déclenchée à chaque push sur main
        githubPush()

        // Ou déclenchée par timer
        cron('H/15 * * * *')  // Toutes les 15 minutes

        // Ou déclenchée par une autre pipeline
        upstream(upstreamProjects: 'shared-library-pipeline', threshold: hudson.model.Result.SUCCESS)
    }
}
```

Flow Complet :

1. Developer push → microservice-payment/
2. GitHub webhook → Jenkins
3. Jenkins déclenche "payment-service-pipeline"
4. Pipeline charge @Library('shared-library')
5. Pipeline exécute buildMicroservice(), testApplication(), etc.
6. Pipeline update le repo GitOps
7. ArgoCD détecte le changement → Déploie

📂 Structure GitOps - Réponse Directe

Oui, généralement 1 SEUL repo GitOps pour tous :

```
k8s-gitops-repo/
├── applications/
│   ├── payment-service/        # App ArgoCD #1
│   │   ├── dev/
│   │   ├── staging/
│   │   └── prod/
│   ├── user-service/           # App ArgoCD #2
│   │   ├── dev/
│   │   ├── staging/
│   │   └── prod/
│   ├── billing-service/        # App ArgoCD #3
│   │   └── ...
│   └── ... (37 autres services)
├── shared-infrastructure/
├── applicationsets/
└── projects/
```

🎯 ArgoCD Apps - Réponse Directe

OUI, il y aura 40+ Applications ArgoCD !

Mais organisées intelligemment :

yaml

```
# ArgoCD UI montrera :
ArgoCD Applications:
├── 📁 core-services (ApplicationSet)
│   ├── 🟢 payment-service-dev
│   ├── 🟢 payment-service-staging
│   ├── 🟢 payment-service-prod
│   ├── 🟢 user-service-dev
│   ├── 🟢 user-service-staging
│   └── 🟢 user-service-prod
├── 📁 business-services (ApplicationSet)
│   ├── 🟢 billing-service-dev
│   └── ... (autres services)
└── 📁 shared-infrastructure
    ├── 🟢 ingress-controller
    ├── 🟢 monitoring
    └── 🟢 logging
```

Total = 40 services × 3 environments = 120 Applications ArgoCD

⚙️ Comment C'est Automatisé

ApplicationSet pour Générer Automatiquement :

yaml

```yaml
# applicationset/microservices.yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: microservices-platform
spec:
  generators:
  - matrix:
      generators:
      - git:
          repoURL: https://github.com/company/k8s-gitops
          directories:
          - path: applications/*
      - list:
          elements:
          - env: dev
            cluster: https://kubernetes.default.svc
          - env: staging
            cluster: https://staging-cluster
          - env: prod
            cluster: https://prod-cluster
  template:
    metadata:
      name: '{{path.basename}}-{{env}}'
    spec:
      project: microservices
      source:
        repoURL: https://github.com/company/k8s-gitops
        path: '{{path}}/{{env}}'
        targetRevision: HEAD
      destination:
        server: '{{cluster}}'
        namespace: '{{path.basename}}-{{env}}'
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
```

🔄 Workflow Complet

Pipeline Jenkins → ArgoCD :

```groovy
```

```
// Dans la Shared Library
def deployToK8s(Map params) {
    stage('Update GitOps') {
        script {
            sh """
                # Clone GitOps repo
                git clone https://github.com/company/k8s-gitops.git
                cd k8s-gitops

                # Update image tag pour chaque environnement
                for env in dev staging prod; do
                    sed -i 's|image: .*|image: harbor.company.com/${params.serviceName}:${env.BUILD_NUMBER}|' \
                        applications/${params.serviceName}/\$env/values.yaml
                done

                # Commit et push
                git add .
                git commit -m "🚀 Deploy ${params.serviceName}:${env.BUILD_NUMBER}"
                git push origin main
            """
        }
    }
}
```

ArgoCD Détection :

```
yaml

# ArgoCD surveille le repo GitOps
applications/payment-service/dev/values.yaml:
  image: harbor.company.com/payment-service:123  # ← Changé par Jenkins

# ArgoCD détecte → Déploie automatiquement
```

📊 Organisation Pratique

Dans ArgoCD UI tu verras :

Folders par équipe :

- **payment-team/** (12 apps = 4 services × 3 envs)
- **user-team/** (9 apps = 3 services × 3 envs)
- **inventory-team/** (15 apps = 5 services × 3 envs)

Ou par environnement :

- **dev/** (40 apps)
- **staging/** (40 apps)
- **prod/** (40 apps)

🎯 Réponses Directes

1. Comment sont lancées les pipelines ?

- Git webhooks + Jenkins multibranch pipelines
- Chaque push sur un microservice déclenche SA pipeline
- La pipeline charge la shared library automatiquement

2. Un seul repo GitOps ?

- **OUI**, un seul repo GitOps avec structure par service
- Plus facile à gérer et gouverner

3. 40 apps ArgoCD ?

- **OUI** minimum 40 (ou 120 avec les environnements)
- Mais générées automatiquement via ApplicationSets
- Organisées en folders pour la lisibilité

En gros : 1 repo GitOps + ApplicationSets = 40+ apps ArgoCD créées automatiquement !

JE PEUX DIRE QUE J'AI GERE POUR DEVOPS 1 ET 2 ET SUR DEVOPS 3 JUSTE LA PARTIE DE MONITORING AVEC PROMETHEUS ET GRAFANA