GitOps : L'évolution du déploiement continu

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

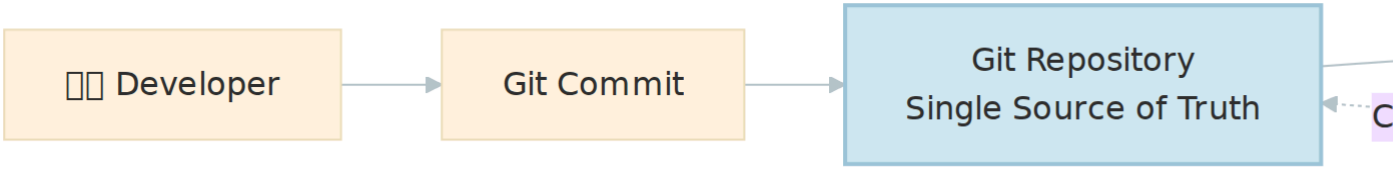
2026-10-17

Table des matières

1 Qu'est-ce que GitOps ? 1
2 L'évolution : De CI/CD traditionnel à GitOps 1
2.1 ⚙ Génération 2 : CI/CD Push 2
2.2 ☒ Génération 3 : GitOps 3
3 Exemple concret 3
4 Les 4 principes de GitOps 3
5 Séparation Application Code vs Configuration Code 4
6 Progressive Delivery avec GitOps 5
7 Outils GitOps principaux 5
8 Avantages de GitOps 6
8.1 Avantages : Rollback & Fiabilité 6
8.2 Avantages : Productivité & Collaboration 6
9 Quand adopter GitOps ? 6

1 Qu'est-ce que GitOps ?

GitOps = Git + Operations : Git comme interface de contrôle pour l'infrastructure

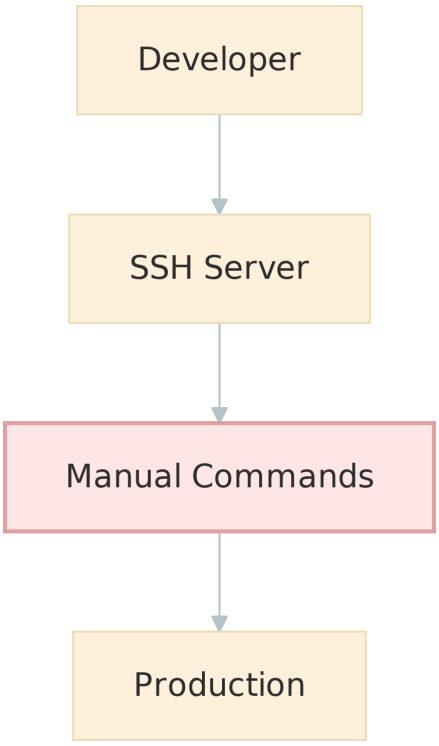


GitOps est une méthodologie de déploiement où :

1. **Git = source de vérité**
   - Tout est versionné dans Git
   - Infrastructure, apps, config
2. **Déclaratif**
   - On décrit l'état désiré
   - Pas de scripts impératifs
3. **Automatique**
   - Réconciliation continue
   - Self-healing
4. **Observable**
   - État actuel = Git
   - Audit trail complet

2 L'évolution : De CI/CD traditionnel à GitOps

Génération 1 : Manuel

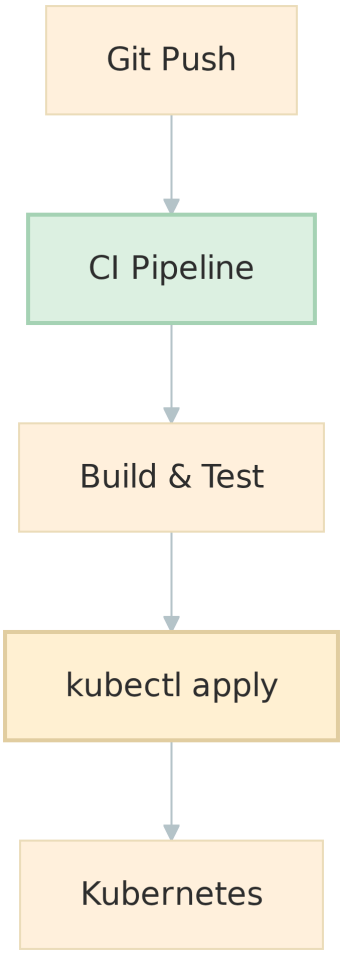


- Scripts shell
- SSH manuelle
- Restart services
- ✕ Erreurs humaines
- ✕ Pas reproductible
- ✕ Pas d'historique

Problèmes :

- Erreur humaine (mauvais fichier, mauvais serveur)
- Pas de rollback facile
- Pas d'historique des changements
- Documentation outdated
- « Snowflake servers » (chaque serveur différent)

2.1 ⚙ Génération 2 : CI/CD Push



- Automatisé
- Tests intégrés
- Pipeline as code
- ⚠ Push = one-way
- ⚠ Drift possible
- ⚠ Credentials dans CI

Mais encore des problèmes :

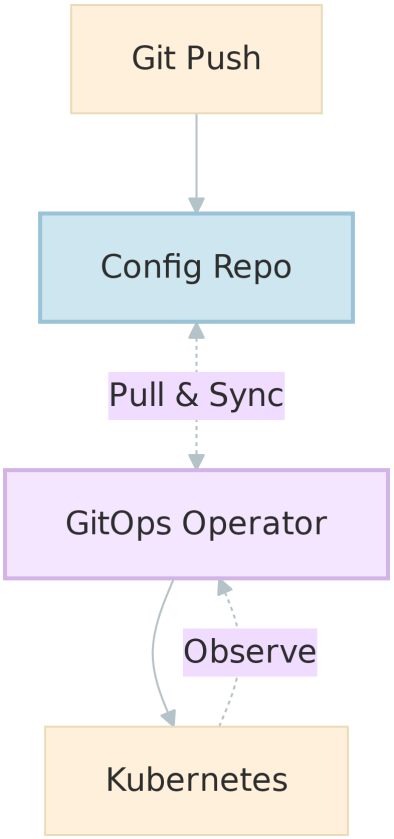
1. **Configuration drift** : changements manuels non détectés, état réel ≠ état désiré
2. **Security** : credentials dans CI/CD, risque si CI compromis

3. **One-way sync** : CI pousse vers K8s, pas de feedback loop
4. **Disaster Recovery** : cluster détruit → comment reconstruire ? — pas de garantie Git = Production

Améliorations :

- Automatisation complète
- Tests avant déploiement
- Infrastructure as Code
- Reproductible

2.2 □ Génération 3 : GitOps



- □ Déclaratif
- □ Convergence auto
- □ Secure by design

Principes clés :

1. **Git = Single Source of Truth**:
   - Toute la config dans Git
2. **Pull model** :
   - GitOps operator pull config
3. **Continuous reconciliation** :
   - Compare état réel vs désiré
   - Applique changements automatiquement
4. **Cryptographically signed commits** :
   - Sécurité renforcée

3 Exemple concret

Quelqu'un fait un changement manuel :

```
kubectl scale deployment myapp --
replicas=10
```

CI/CD traditionnel : Rien ne se passe, drift permanent

GitOps :

- ArgoCD détecte : Git dit 3 replicas, K8s a 10
- Status « OutOfSync »
- Auto-sync (si activé) → Revient à 3 replicas
- Notification « Manual change detected and corrected »

4 Les 4 principes de GitOps

OpenGitOps : Les principes fondamentaux

□□□ Déclaratif

```
# Pas "comment faire"
# Mais "quel état je veux"
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 3  # État désiré
  template:
    spec:
```

```
      containers:
      - image: myapp:v1.2.3
```

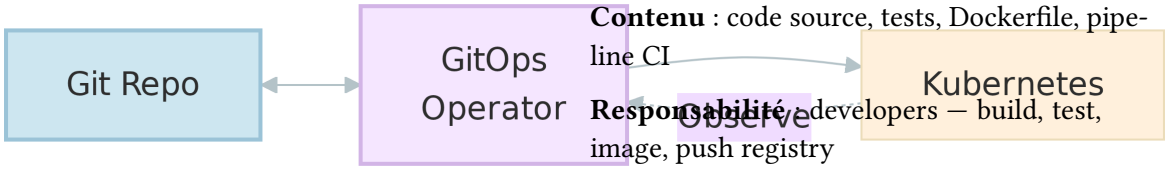
 Reproductible Idempotent Versionnable

Versionné et Immutable

```
git log --oneline
abc123 Update to v1.2.3
def456 Scale to 5 replicas
789abc Add health check
```

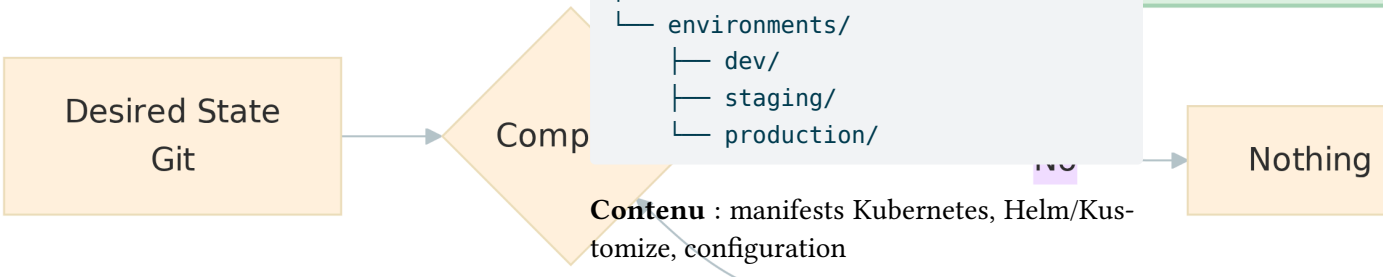
 Historique complet Rollback facile Audit trail

Pulled Automatiquement



 No push access needed Continuous sync Self-healing

Réconciliation Continue



 Auto-correction Drift detection Guaranteed convergence

5 Séparation Application Code vs Configuration Code

Deux repositories distincts : Une bonne pratique GitOps

Application Repository

```
my-app-repo/
├── src/
│   └── main/
│       └── java/
├── tests/
├── pom.xml
├── Dockerfile
└── .github/
    └── workflows/
        └── ci.yml
```

Contenu : code source, tests, Dockerfile, pipeline CI

Responsabilité : developers — build, test, image, push registry

GitOps Repository

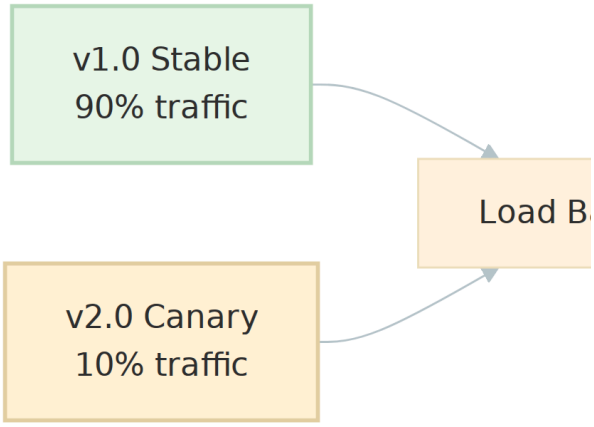
```
gitops-repo/
├── apps/my-app/
│   ├── deployment.yaml
│   ├── service.yaml
│   ├── ingress.yaml
│   └── configmap.yaml
├── infrastructure/
└── environments/
    ├── dev/
    ├── staging/
    └── production/
```

Contenu : manifests Kubernetes, Helm/Kustomize, configuration

Responsabilité : platform team/SRE — config & infrastructure ; ArgoCD/Flux surveille ce dépôt

6 Progressive Delivery avec GitOps

Stratégies de déploiement avancées : Canary et Blue/Green

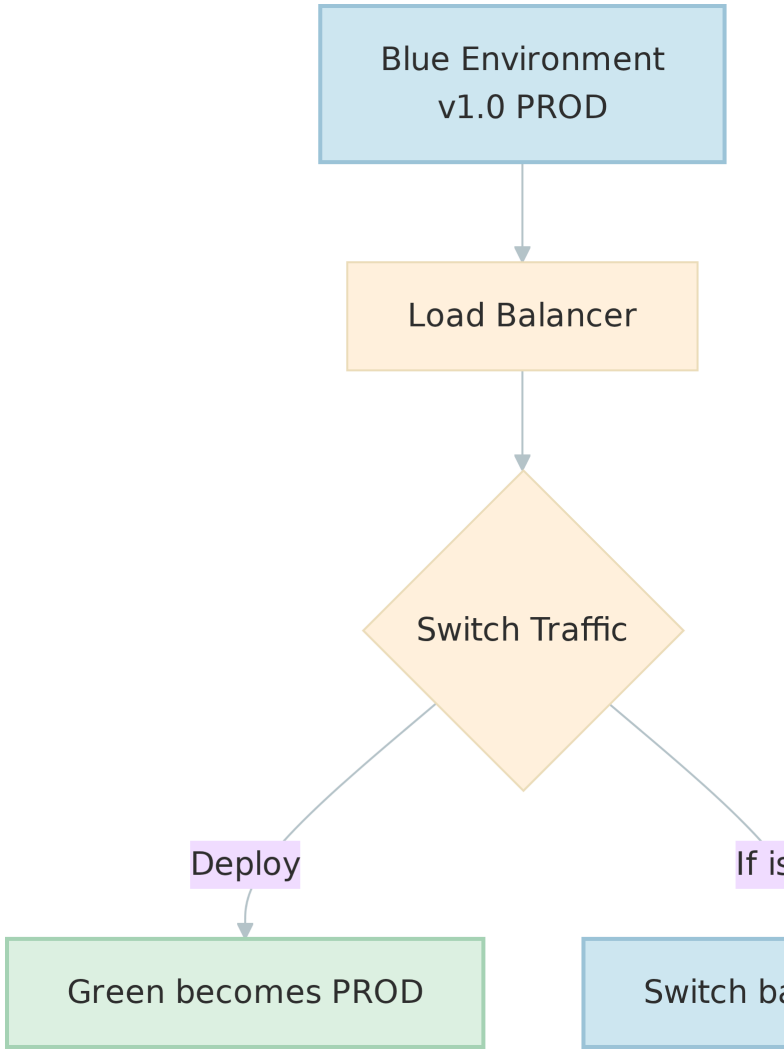


Canary : nouvelle version exposée à un petit % d'utilisateurs avant extension

1. v2.0 déployé pour 10% users
2. Monitoring (erreurs, latence)
3. Si OK ⟶ 25%, 50%, 100%
4. Si KO ⟶ Rollback immédiat

GitOps : Changer replicas progressivement dans Git

Blue/Green Deployment



Principe : Deux environnements identiques

1. Blue (v1.0) = Production actuelle
2. Green (v2.0) = Nouvelle version testée
3. Switch load balancer Blue ⟶ Green
4. Rollback = Switch Green ⟶ Blue

GitOps : Basculer Ingress/Service dans Git

7 Outils GitOps principaux

ArgoCD et Flux : Les deux leaders

ArgoCD

- UI/Dashboard riche
- Créé par Intuit (2018)

- CNCF Graduated
- Multi-cluster natif
- CLI puissant
- SSO intégré

Avantages : Interface graphique Facile à démarrer Écosystème plugins

Usage : Équipes préférant UI, multi-tenancy

Flux

- GitOps Toolkit modulaire
- Créé par Weaveworks (2016)
- CNCF Graduated
- Approche « toolkit »
- CLI `flux`
- Notifications avancées

Avantages : Léger et composable Helm natif Progressive delivery

Usage : Équipes cloud-native, automation

8 Avantages de GitOps

Les bénéfices concrets de l'approche GitOps

Sécurité

- Pas de credentials externes
- Audit trail complet
- Commits signés
- RBAC via Git

Traçabilité

- Historique complet dans Git
- Qui, quoi, quand, pourquoi
- Compliance (SOC2, ISO27001)

8.1 Avantages : Rollback & Fiabilité

Rollback instantané

- `git revert` = rollback production
- Retour à n'importe quelle version
- Pas de procédure complexe

Fiabilité

- Self-healing automatique
- Drift detection
- État convergent garanti

8.2 Avantages : Productivité & Collaboration

Productivité

- Déploiements reproductibles
- Multi-environnements facile
- Disaster recovery simplifié

Collaboration

- Pull Request (PR) = Review process
- GitOps Repo = Documentation vivante
- Changements atomiques et versionnés

9 Quand adopter GitOps ?

GitOps est idéal si…

Vous devriez utiliser GitOps si :

- Déploiements sur Kubernetes
- Besoin d'audit trail complet
- Multi-environnements (dev/staging/prod)
- Équipe cloud-native
- Besoin de self-healing
- Compliance stricte (finance, santé)
- Multi-cluster management

Cas d'usage typiques : - Microservices sur K8s - SaaS multi-tenant - Infra as Code (IaC) - Platform engineering

⚠ **GitOps peut être overkill si :**

- Applications monolithiques
- Déploiement sur VMs traditionnelles
- Pas de Kubernetes
- Très petite équipe (1-2 devs)
- Déploiements rares
- Environnement unique

Alternatives : - CI/CD classique (GitHub Actions, GitLab CI) - Scripts Ansible - Docker

Compose pour dev/staging - Cloud-native solutions (AWS ECS, Cloud Run)