Engineer

GitOps

exploiter l'infrastructure selon un état désiré dans Git

Ce que c’est

GitOps est une manière d’exploiter des systèmes logiciels où l’état désiré d’un système est consigné sous forme de déclarations dans Git et où des agents logiciels maintiennent le système en fonctionnement en conformité avec elles. Personne ne modifie la production à la main ou depuis un script sur un ordinateur portable. Un changement est un commit : un agent dans l’environnement remarque le nouveau commit et rend le système conforme. Le projet OpenGitOps formule cette idée en quatre principes, dont la version 1.0.0 a été publiée le 8 octobre 2021 par le GitOps Working Group, qui supervise le projet.

Ces principes disent que l’état désiré doit être déclaratif, versionné et immuable, récupéré automatiquement et réconcilié en continu. Déclaratif signifie que les fichiers décrivent ce que le système doit être, pas les commandes pour y parvenir. Versionné et immuable signifie qu’une version conservée n’est jamais modifiée, seulement remplacée par une autre, et que tout l’historique est gardé. Récupéré automatiquement signifie que les agents logiciels vont chercher eux-mêmes les déclarations à la source. Réconcilié en continu signifie qu’ils comparent régulièrement le système en fonctionnement à sa déclaration et travaillent à les faire concorder ; le glossaire du projet précise que continu signifie que le travail se poursuit, pas qu’il est instantané.

Deux limites de ce glossaire passent facilement inaperçues. Git, à l’origine du nom, est le stockage canonique de l’état, mais tout stockage convient s’il conserve des versions immuables et assure le contrôle d’accès et l’audit des modifications de l’état désiré. Et cet état est la configuration nécessaire pour recréer le système : il exclut généralement les données persistantes, comme le contenu d’une base. GitOps peut reconstruire les machines qui hébergent une base, pas les lignes qu’elle contient.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que GitOps ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des changements testés, publiés et réversibles, pour livrer aux clients un mardi après-midi plutôt qu'un vendredi soir tard.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Déploiement SaaS GIS automatisé : GitHub Actions — déploiement, données et reporting automatisés

Déploiement SaaS GIS automatisé : GitHub Actions Contactez-nous →

Pourquoi c’est important

La différence avec un pipeline de déploiement ordinaire tient à ce qui lance le travail. Le glossaire OpenGitOps oppose les tâches CI/CD lancées par des déclencheurs configurés à l’avance à GitOps, où la réconciliation se déclenche dès que l’état réel diverge de l’état déclaré. Cette divergence peut venir des deux côtés : un nouveau commit change le résultat souhaité, ou le système en fonctionnement s’écarte de sa déclaration. Un pipeline ayant exécuté `kubectl apply` une fois ne voit pas ce qui s’est passé ensuite. La documentation de Kubernetes note que l’application déclarative conserve les écritures faites directement sur les objets actifs sans les réintégrer dans les fichiers de configuration : les fichiers et le cluster peuvent donc diverger discrètement. Un agent de réconciliation remarque les deux sortes de divergence.

Le deuxième avantage concerne l’endroit où les droits d’accès sont nécessaires. La documentation d’Argo CD souligne qu’avec la synchronisation automatique, un pipeline de déploiement peut fonctionner sans identifiants pour le serveur d’API d’Argo CD : le pipeline crée un commit et le pousse. Il lui faut l’accès en écriture à un dépôt plutôt qu’au cluster, et c’est déjà dans le dépôt que les changements sont revus.

Le troisième est l’historique. L’état désiré étant versionné et immuable, le journal du dépôt consigne chaque changement voulu : ce qui a changé, qui l’a changé et quand. Revenir en arrière revient à choisir une version antérieure, et Argo CD cite parmi ses fonctions le retour à toute configuration d’application committée dans Git. Après un incident, cet historique répond à la première question, « qu’est-ce qui a changé ? », sans que personne doive le reconstruire à partir de l’historique du shell et de sa mémoire.

À quoi cela ressemble en pratique

Flux et Argo CD sont deux agents qui mettent cela en pratique sur Kubernetes. Argo CD se présente comme un outil déclaratif de livraison continue GitOps pour Kubernetes ; Flux appelle son ensemble de contrôleurs et d'API le GitOps Toolkit. Prenons une équipe dont les manifestes de production sont dans un dépôt appelé fleet. Deux objets Flux le relient au cluster : le premier indique au contrôleur de sources de Flux où se trouve le dépôt et à quelle fréquence le récupérer ; le second indique au contrôleur kustomize quel répertoire appliquer et à quelle fréquence vérifier le cluster par rapport à celui-ci.

```
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: fleet
  namespace: flux-system
spec:
  interval: 5m
  url: https://git.example.com/platform/fleet.git
  ref:
    branch: main
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: production
  namespace: flux-system
spec:
  interval: 10m
  sourceRef:
    kind: GitRepository
    name: fleet
  path: "./production"
  prune: true
  timeout: 1m
```

Suivons un changement. Un ingénieur fusionne une demande d'intégration qui passe un service à une nouvelle étiquette d'image. Le contrôleur de sources récupère la branche toutes les cinq minutes et découvre une nouvelle révision à sa prochaine récupération. Une nouvelle révision de source est traitée immédiatement plutôt qu'au prochain intervalle : le contrôleur kustomize construit donc les manifestes sous `./production` et les applique. La documentation Flux décrit la suite pour exactement cette paire d'objets. Toutes les dix minutes, selon l'intervalle fixé, la `Kustomization` effectue une simulation de l'opération d'application côté serveur pour détecter les divergences, puis les

réconcilie. Un collègue modifiant le même objet avec `kubectl edit` voit donc sa modification annulée. Comme `prune` est activé, un objet dont le manifeste a été retiré du répertoire est supprimé du cluster à la révision suivante. Revenir en arrière consiste à faire un `git revert` puis une fusion, par le même chemin que tout autre changement.

Argo CD accomplit le même travail avec une ressource `Application`, selon d'autres réglages prédéfinis. Une page de sa documentation indique une interrogation des dépôts toutes les trois minutes ; une autre, sur la synchronisation automatique, fixe l'intervalle à 120 secondes plus jusqu'à 60 secondes de variation aléatoire, soit trois minutes au maximum. Lorsque l'état actif diffère de la cible dans Git, l'application est marquée OutOfSync et, si la synchronisation automatique est activée, Argo CD la synchronise. Sans réglage contraire, cependant, elle ne supprime rien de ce qui a disparu de Git et ne réagit pas aux changements faits directement dans le cluster. La suppression des ressources retirées de Git et l'autoréparation ont deux interrupteurs distincts, `prune` et `selfHeal`, désactivés au départ.

Le même schéma fonctionne hors de Kubernetes. La réconciliation d'une zone DNS de 20 enregistrements avec l'API du fournisseur, réalisée en août 2026, a transformé les enregistrements d'une console web de fournisseur en déclaration dans le dépôt. Un point d'entrée réconcilie la déclaration avec le fournisseur ; un autre, qui exige une activation explicite, se limite à l'audit et signale la différence sans rien changer. La page indique que l'audit est la partie la plus utilisée, car on souhaite plus souvent connaître la différence que la supprimer. Au regard des quatre principes, cela correspond aux deux premiers : la zone est déclarée et versionnée. La page ne décrit pas d'agent récupérant la déclaration et l'appliquant seul ; les deux derniers ne sont donc pas revendiqués. C'est aussi un argument raisonnable pour commencer comme Argo CD dans sa configuration standard : montrer la dérive avant de laisser quoi que ce soit la corriger.

Où les problèmes surviennent

Des secrets en clair dans Git. Un dépôt contenant tout l'état désiré incite à y mettre aussi les mots de passe, et son historique les conserve après leur suppression. La documentation de Flux qualifie de dangereux le stockage des secrets Kubernetes en clair ou en base64 dans Git, même dans un dépôt privé à accès restreint. Chiffrez-les avant de les committer. Flux déchiffre avec SOPS selon la configuration de `.spec.decryption`, au moyen d'age, d'OpenPGP ou d'un service cloud de gestion de clés ; ne chiffrez que les champs `data` et `stringData`, car `metadata`, `kind` et `apiVersion` doivent rester lisibles. La

documentation d'Argo CD recommande fortement une autre voie : résoudre les secrets sur le cluster de destination avec un outil comme Sealed Secrets, External Secrets Operator ou Secrets Store CSI Driver, plutôt que de les injecter pendant la génération des manifestes par Argo CD. Elle déconseille cette injection notamment parce que son cache Redis conserve une copie non chiffrée des manifestes produits.

Un correctif urgent qui ne tient pas, ou une dérive qui ne disparaît jamais. Les deux outils se comportent différemment avec leurs réglages initiaux. Avec Flux, un changement fait par `kubectl edit`, `patch` ou `delete` est rapidement annulé. C'est souhaitable jusqu'au jour où quelqu'un applique à la main un correctif urgent pendant une panne et le voit disparaître. Avec Argo CD, une modification directe du cluster ne déclenche pas de synchronisation automatique sans autoréparation activée : la dérive peut rester là indéfiniment, marquée OutOfSync. La solution est la même : faire le changement dans Git ou suspendre ouvertement la réconciliation. `.spec.suspend` de Flux arrête l'application des nouvelles révisions et suspend la correction des dérives jusqu'à son retour à false. Pour un champ appartenant à un autre contrôleur, tel que le nombre de réplicas d'un `Deployment` géré par un autoscaler, `.spec.ignore` de Flux accepte des chemins JSON Pointer comme `/spec/replicas` afin que les deux contrôleurs cessent d'écraser leurs modifications respectives. Revenez aussi en arrière par Git : Argo CD refuse un retour arrière sur une application dont la synchronisation automatique est activée.

Supprimé dans Git, toujours en fonctionnement. Retirer un manifeste retire la déclaration, pas l'objet. La synchronisation automatique d'Argo CD ne supprime les ressources qui ne sont plus définies dans Git que si la suppression est activée. Flux ne les supprime que lorsque `.spec.prune`, un champ que chaque `Kustomization` doit définir, vaut true. Laissez-le désactivé et le cluster se remplit lentement d'objets que personne ne déclare. Activez-le et marquez les quelques objets qui doivent survivre à leur manifeste avec le label ou l'annotation `kustomize.toolkit.fluxcd.io/prune: disabled`. Argo CD ajoute un garde-fou à conserver : la synchronisation automatique avec suppression ne laisse pas une application sans aucune ressource, sauf si `allowEmpty` est défini. Ce garde-fou empêche ainsi un commit défectueux qui vide un répertoire de vider le cluster.

Une fusion qui attend l'interrogation. Un agent qui récupère la source à intervalles réguliers échange de la vitesse contre de l'indépendance : Argo CD donne trois minutes comme période d'interrogation, et Flux voit un commit à la prochaine récupération de la source. Les deux acceptent un webhook du serveur Git pour réduire l'attente. Argo CD le reçoit au point d'entrée `/api/webhook` ; Flux

utilise une ressource `Receiver` qui transforme un événement push en réconciliation des sources qu'elle énumère. Le secret partagé du webhook d'Argo CD est facultatif, car une charge utile ne déclenche qu'une actualisation, mais sa documentation en recommande un quand Argo CD est publiquement accessible, pour prévenir les attaques distribuées par déni de service. Flux génère un chemin de webhook unique à partir du nom du récepteur, de son espace de noms et d'un jeton secret ; pour GitHub, il vérifie la signature HMAC de la charge utile.

Git devient une dépendance de la production. Si l'agent récupère ses déclarations depuis le serveur Git, une panne de Git arrête tous les changements, correctifs compris. La variante de Flux fondée sur OCI, appelée Gitless GitOps, garde Git comme interface pour les personnes, mais fait récupérer au cluster des artefacts versionnés dans un registre de conteneurs. Le cluster peut ainsi obtenir ses artefacts de production sans contacter le serveur Git. Quelle que soit la source lue par l'agent, donnez-lui la disponibilité attendue de la production : pendant une panne, c'est la seule voie qu'un changement peut emprunter.

Pour aller plus loin

- OpenGitOps — les quatre principes GitOps, version 1.0.0, du GitOps Working Group.
- Glossaire GitOps v1.0.0 — les définitions derrière les principes : état désiré, dérive, réconciliation et stockage de l'état.
- Politique de synchronisation automatique — la synchronisation automatique d'Argo CD, les interrupteurs de suppression, d'autoréparation et d'application vide, ainsi que le moment où chaque synchronisation s'exécute.
- Kustomization — l'API Flux pour appliquer un répertoire de manifestes : intervalle, suppression, suspension, exclusion et déchiffrement SOPS.
- Gestion des secrets — les arguments d'Argo CD pour résoudre les secrets sur le cluster de destination plutôt que pendant la génération des manifestes.
- Gestion déclarative des objets Kubernetes par fichiers de configuration — le fonctionnement de `kubectl apply` et `kubectl diff` sur un répertoire de fichiers de configuration, et le traitement par apply des modifications faites sur des objets actifs.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que GitOps ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/gitops/