Engineer

GitOps

running infrastructure from a desired state kept in Git

What it is

GitOps is a way of operating software systems in which the desired state of a system is written down as declarations in Git, and software agents keep the running system in line with them. Nobody changes production by hand or from a script on a laptop. A change is a commit, and an agent inside the environment notices the new commit and makes the system match it. The OpenGitOps project spells the idea out as four principles, and version 1.0.0 of those principles was published on 8 October 2021 by the GitOps Working Group that oversees the project.

The principles say the desired state must be declarative, versioned and immutable, pulled automatically and continuously reconciled. Declarative means the files describe what the system should look like, not the commands that get it there. Versioned and immutable means a stored version is never edited, only superseded, and the complete history is kept. Pulled automatically means software agents fetch the declarations from the source themselves. Continuously reconciled means agents repeatedly compare the running system with its declaration and work to bring the two together; the project's glossary adds that continuous means the work keeps happening, not that it is instantaneous.

Two boundaries in that glossary are easy to miss. Git, from which the name comes, is the canonical state store, but any store qualifies that keeps immutable versions and provides access control and auditing on changes to the desired state. And the desired state is the configuration needed to recreate the system, which generally excludes persistent data such as the contents of a database. GitOps can rebuild the machines a database runs on; it cannot rebuild the rows inside it.

Use the concept in a decision

Put “What is GitOps?” to work in a real decision. We can help relate the concept to the system you are building or buying. Tested, released and reversible, so changes reach customers on a Tuesday afternoon rather than late on a Friday night.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Automated GIS SaaS deployment with GitHub Actions — deployment, data and reporting, all off the hands

Automated GIS SaaS deployment with GitHub Actions Get in touch →

Why it matters

The difference from an ordinary deployment pipeline is what starts the work. The OpenGitOps glossary contrasts CI/CD jobs started by previously configured triggers, with GitOps, where reconciliation is triggered whenever the actual state diverges from the declared one. That divergence can come from either side: a new commit changes what is wanted, or the running system drifts away from what was declared. A pipeline that ran `kubectl apply` once has no view of what happened afterwards. Kubernetes’ own documentation notes that declarative apply keeps writes made directly to live objects without merging them back into the configuration files, so the files and the cluster can quietly disagree. An agent that reconciles notices both kinds of divergence.

The second gain is where access lives. Argo CD’s documentation points out that with automated sync, a deployment pipeline can work without credentials for Argo CD’s API server: the pipeline makes a commit and pushes it. What the pipeline needs is write access to a repository rather than to the cluster, and the repository is where changes are reviewed anyway.

The third is the history. Because the desired state is versioned and immutable, the log of the repository is a record of every intended change: what changed, who changed it and when. Going back means choosing an earlier version, and Argo CD lists rollback to any application configuration committed in Git among its features. After an incident, that history answers the first question, what changed, without anyone reconstructing it from shell history and memory.

What it looks like in practice

Flux and Argo CD are two agents that put this into practice on Kubernetes. Argo CD describes itself as a declarative, GitOps continuous delivery tool for Kubernetes; Flux calls its set of controllers and APIs the GitOps Toolkit. Take a

team that keeps its production manifests in a repository called fleet. Two Flux objects connect the cluster to it: the first tells Flux's source controller where the repository is and how often to fetch it, and the second tells the kustomize controller which directory to apply and how often to check the cluster against it.

```
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
```

Now follow one change. An engineer merges a pull request that moves a service to a new image tag. The source controller fetches the branch every five minutes, and at its next fetch it sees a new revision. A new source revision is handled at once rather than at the next interval, so the kustomize controller builds the manifests under `./production` and applies them. Flux's documentation describes the rest for exactly this pair of objects. Every ten minutes, as the interval sets, the `Kustomization` uses a dry run of server-side apply to discover divergence, then reconciles it, so a colleague who changes the same object with `kubectl edit` sees the change undone. Because `prune` is on, an object whose manifest has been removed from the directory is deleted from the cluster with the next revision. Rolling back is a `git revert` and a merge, which travels the same path as any other change.

Argo CD does the same job through an `Application` resource, with different defaults. One page of its documentation says it polls repositories every three minutes; another, on automated sync, puts the interval at 120 seconds plus up to 60 seconds of jitter, for a maximum of three minutes. When the live state differs from the target state in Git, the application is marked OutOfSync, and with automated sync on, Argo CD syncs it. By default, though, automated sync deletes nothing that has disappeared from Git and does not react to changes made directly in the cluster. Pruning and self-healing are separate switches, `prune` and `selfHeal`, and both start off.

The same pattern works outside Kubernetes. A 20-record DNS zone reconciled against the provider's API, carried out in August 2026, turned records that lived in a provider's web console into a declaration in the repository. One entry point reconciles the declaration against the provider; another, behind an explicit opt-in, only audits, reporting the difference without changing anything. The page names the audit as the part used most, because knowing the difference is more often what is wanted than closing it. Measured against the four principles, that is the first two: the zone is declared and versioned. The page does not describe an agent fetching the declaration and applying it on its own, so the last two are not claimed. It is also a fair argument for starting where Argo CD starts by default: show the drift before letting anything correct it.

Where it goes wrong

Secrets in plain text in Git. A repository that holds the whole desired state tempts people to put the passwords there too, and its history keeps them after they are deleted. Flux's documentation calls storing Kubernetes secrets in Git as plain text or base64 unsafe, even in a private repository with restricted access. Encrypt them before they are committed. Flux decrypts with SOPS, using age, OpenPGP or a cloud key management service, configured through `.spec.decryption`; encrypt only the `data` and `stringData` fields, because `metadata`, `kind` and `apiVersion` must stay readable. Argo CD's documentation strongly recommends a different route: resolve secrets on the destination cluster with a tool such as Sealed Secrets, the External Secrets Operator or the Secrets Store CSI Driver, rather than injecting them while Argo CD generates manifests, which it cautions against partly because its Redis cache holds an unencrypted copy of the resulting manifests.

A hotfix that will not stay, or drift that never goes. The two tools behave differently here by default. Under Flux, a change made with `kubectl edit`, `patch` or `delete` is promptly reverted, which is right until someone applies an

urgent fix by hand during an outage and watches it disappear. Under Argo CD, a change made directly in the cluster does not trigger an automated sync unless self-healing is on, so drift can sit there, marked OutOfSync, indefinitely. The fix is the same for both: make the change in Git, or pause reconciliation openly. Flux's `.spec.suspend` stops new revisions from being applied and pauses drift correction until it is set back to false. For a field another controller owns, such as the replica count of a `Deployment` that an autoscaler manages, Flux's `.spec.ignore` takes JSON Pointer paths like `/spec/replicas`, so the two controllers stop overwriting each other. Roll back through Git as well: Argo CD refuses a rollback on an application with automated sync enabled.

Deleted in Git, still running. Removing a manifest removes the declaration, not the object. Argo CD's automated sync does not delete resources that are no longer defined in Git unless pruning is enabled, and Flux deletes them only when `.spec.prune`, a field every `Kustomization` must set, is true. Leave it off and the cluster slowly fills with objects that nobody declares. Turn it on, and mark the few objects that must outlive their manifest with the `kustomize.toolkit.fluxcd.io/prune: disabled` label or annotation. Argo CD adds a guard worth keeping: automated sync with pruning will not leave an application with no resources at all unless `allowEmpty` is set, which stops a broken commit that empties a directory from emptying the cluster.

A merge that waits for the poll. An agent that fetches on a timer trades speed for independence: Argo CD gives three minutes as its polling period, and Flux sees a commit at the next fetch of the source. Both accept a webhook from the Git server to cut the wait. Argo CD takes it at its `/api/webhook` endpoint; Flux uses a `Receiver` resource that turns a push event into a reconciliation of the sources it lists. Argo CD's shared webhook secret is optional, because a payload only triggers a refresh, but its documentation recommends one when Argo CD is publicly reachable, to prevent distributed denial-of-service attacks. Flux generates a unique webhook path from the receiver's name, namespace and a secret token and, for GitHub, verifies the payload's HMAC signature.

Git becomes a production dependency. If the agent fetches from the Git server, a Git outage stops every change, fixes included. Flux's OCI-based variant, which it calls Gitless GitOps, keeps Git as the interface people use but has the cluster pull versioned artifacts from a container registry, allowing the cluster to obtain its production artifacts without contacting the Git server. Whichever source the agent reads, give it the availability you expect of production, because during an outage it is the only route a change can take.

Further reading

- OpenGitOps — the four GitOps principles, version 1.0.0, from the GitOps Working Group.
- GitOps Glossary v1.0.0 — the definitions behind the principles: desired state, drift, reconciliation and the state store.
- Automated Sync Policy — Argo CD's automated sync, with the prune, self-heal and allow-empty switches and when each sync runs.
- Kustomization — Flux's API for applying a directory of manifests: interval, prune, suspend, ignore and SOPS decryption.
- Secret Management — Argo CD's case for resolving secrets on the destination cluster rather than during manifest generation.
- Declarative Management of Kubernetes Objects Using Configuration Files — how `kubectl apply` and `kubectl diff` work on a directory of configuration files, and what apply does with changes made to live objects.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is GitOps?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/gitops/