The Essentials of GitOps

CONTENTS

BY WILLIAM CHIA
SR. PRODUCT MARKETING MANAGER, GITLAB

THE NEED FOR A NEW OPERATIONAL MODEL

With increasing user demand for always-on applications, available on any device, the complexity and sophistication of software development and delivery amplifies. Traditional, monolithic apps are being decomposed into cloud services that are built and run by a distributed workforce, collaborating from around the globe. Operations teams need a new paradigm to support the upsurge of deployment frequency across multiplying services and in multiple regions, all while reducing risk, increasing uptime, and staying secure.

GitOps is an operational framework that takes DevOps best practices used for application development, such as version control, collaboration, compliance, and CI/CD, and applies them to infrastructure automation.

While GitOps practices work well within any software environment, Kubernetes is particularly well suited due to its use of declarative infrastructure definitions kept alongside application code in a Git repository. Keeping your system definition in Git means your engineers can use familiar, Git-based tooling and workflows to manage both application and infrastructure changes.

With the entire state of your cluster kept under source control, you gain the ability to automate changes for less risk, recreate historical states, review a robust audit log, manage compliance, and more.

INFRASTRUCTURE AUTOMATION FOR MODERN APPLICATIONS

GitOps practices aren't dependent on any specific technology. While logically GitOps is simply managing operations by Git, there are three core practices that make up a mature GitOps practice: Infrastructure-as-code (IaC), merge requests (MRs) as the agent of change, and continuous integration/continuous delivery (CI/CD).

GitOps = IaC + MRs + CI/CD

CORE PRACTICE: INFRASTRUCTURE-AS-CODE

Infrastructure-as-Code (IaC) is the practice of keeping all infrastructure configuration stored as code. GitOps uses a Git repository as the single source of truth for the definition of

GitLab

A complete DevOps platform, delivered as a single application.

GitOps

Why Choose GitLab for GitOps?

Drive infrastructure automation and collaboration for cloud native, multicloud, and legacy environments. GitLab is a DevOps platform with built-in agile planning, source code management, and CI/CD delivered as a single application.

Foster collaboration between your infrastructure, operations, and development teams. Deploy more frequently with greater confidence while also increasing the stability, reliability, and security of your software environments.

How GitLab enables GitOps:

- **Environments as code** stored in GitLab version control as a single source of truth.
- **Teams collaborate** using GitLab's agile planning and code review.
- **The same tool** used to plan, version, and deploy your application code works for your operations code as well.
- **CI/CD for infrastructure automation** reconciles your environments with your single source of truth in version control
- **World-class integrations** with Terraform and Kubernetes ensure all your GitOps tooling works seamlessly.

The core of your GitOps collaboration starts with GitLab

Try GitLab Free Now

your infrastructure environments. By shifting your environment definitions from manual configuration to configuration by code, you gain access to an array of benefits such as version control, code collaboration, and auditability. You also unlock Git as the user interface for your infrastructure, allowing you to leverage all of the developer tooling, training, and knowledge associated with Git for your infrastructure operations.

Although IaC is a popular and well-known practice, GitOps isn't relegated to simply infrastructure. Any operations that can be defined as code (e.g., network, policy, security) are also benefits of GitOps. In some cases, the term "X-as-Code" (XaC) can be used to encompass operations beyond infrastructure. This Refcard will use the more established term "IaC" with the understanding that we are including the entirety of the operational environment, and not simply the infrastructure.

DECLARATIVE VS. IMPERATIVE ENVIRONMENTS

Many modern infrastructure tools such as Kubernetes, Terraform, and AWS Cloud Formation work off of a declarative model. An operations engineer declares the desired state as code, and the system then changes itself to conform to that state via automation. For example, a Kubernetes manifest can declare the number of pods desired for a particular service. The engineer doesn't need to write an imperative script to bring these pods up or down until the right number is achieved because Kubernetes handles this itself.

It's the difference between saying, "I have three servers but want six, so I need to write a script to create three more servers," and simply telling the system, "There should be six servers. If there comes a point when there are too few or too many, change the state of the environment until we have the correct number." Using declarative patterns can be very powerful within a GitOps operational model, but they aren't a strict requirement. You can still benefit from GitOps practices if your environments are imperatively defined.

CORE PRACTICE: MERGE REQUESTS AS THE AGENT OF CHANGE

It may be surprising to learn that the underlying Git version control system used to power tools such as GitLab, GitHub, and Bitbucket doesn't include a way to request your branch be merged back into the branch it was created or forked from. This was a later advancement introduced by Git management tools.

GitLab uses the term merge request (MR), while GitHub and Bitbucket use the term pull request (PR), but functionally, they perform in a similar way. The MR is the central point of developer collaboration for code review and change orchestration. Without a proper version control and branching strategy, collaboration on new changes is a frustrating endeavor.

When anyone can modify a file without a way to track who made which change, it can be almost impossible to ensure the correct version is being used. A common application development workflow uses a main branch as a centralized collaboration point. Feature branches are created from the main branch, where new work is developed, and then merged back into the main branch via a merge request. Leveraging this best practice for all of your infrastructure code nets you the same benefits that developers enjoy.

In an infrastructure model, the main branch represents a particular environment (e.g., dev or production), as well as the state running in that environment. Changes are proposed on a feature branch, and an MR is made to merge the changes into the main branch. This MR allows for collaboration between operations engineers for peer review, along with the development teams, security teams, and other stakeholders. This powerful model for collaboration permits anyone to propose a change, while also allowing you to maintain compliance by limiting the number of people who can merge the changes.

CORE PRACTICE: CI/CD AUTOMATION

The final component of a robust GitOps strategy is automating all changes made to environments via CI/CD. In an ideal scenario, no manual changes are made to a GitOps-managed environment. Instead, CI/CD serves as a type of reconciliation loop. Each time a change is made, the automation tool compares the state of the environment to the source of truth defined in the Git repository. If the Git repository shows a change, the automation tool reconciles this difference by configuring the environment to match the canonical desired state.

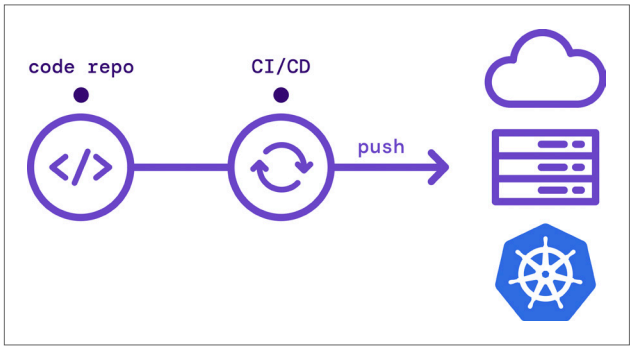
This type of automation serves as a powerful protection against configuration drift. There are many reasons configurations can fall out of sync. Whether due to a component failure or inadvertent manual change, each time the automation runs, it overwrites the existing state with Git source of truth.

AGENT VS. AGENTLESS GITOPS

A couple of different models for GitOps automation have emerged, namely Agent-based and Agentless, each with their own pros and cons. ***Agentless GitOps*** is a traditional model, also known as push-based GitOps, in which your CI/CD tool reads from your Git repository and pushes changes into your environment.

- **Pro** – It's simpler and more flexible as it can be used with any type of infrastructure, from physical servers and VMs to Kubernetes clusters.
- **Con** – You must give your CI/CD tool access to make writes to your environment. Requiring your environment to be open to writes from the external internet can cause security and compliance issues.

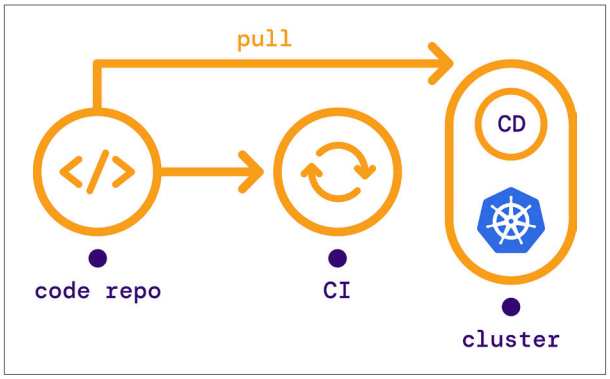
Figure 1: Agentless GitOps model



Agent-based GitOps, also known as pull-based GitOps, makes use of an agent that runs inside of your infrastructure. This agent pulls changes in from an external Git repository when it detects that the state of the environment is out of sync with the source of truth.

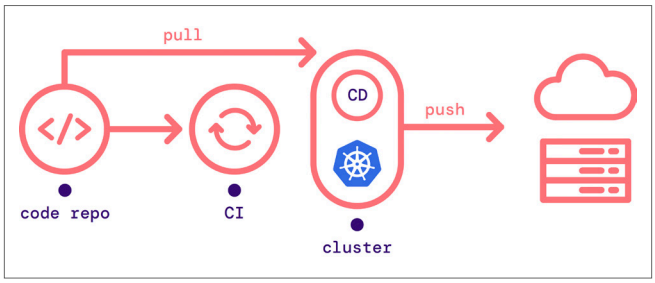
- **Pro** – CD agents can allow you to operate in a more secure and compliant way without the need to open inbound ports in your firewall.
- **Con** – Agents must be custom designed to the type of infrastructure you want to use.

Figure 2: Agent-based GitOps model



Today, most GitOps agents, such as the GitOps Engine, are designed specifically for Kubernetes. For example, these agents won't run inside of a VM-based cluster. A workaround requires setting up a Kubernetes cluster dedicated to orchestration. The CD agent runs inside the orchestration cluster and enacts changes to external infrastructure.

Figure 3: GitOps Engine model



GITOPS BENEFITS

GitOps best practices are far-reaching and can provide the following benefits.

TIGHTER COLLABORATION

Using the merge requests as a central point of collaboration enables teams across the organization to work together in a fast, automated, and asynchronous way. Best practices from one team can be easily shared and consumed across the organization because they are documented as code.

INCREASED DEPLOYMENT FREQUENCY

With automated application deployment and environment provisioning, you can do many small, frequent deployments rather than occasional large, risk-prone deployments. Smaller changes are easier to reason about when troubleshooting and easier to roll back. Additionally, deploying features to your users sooner leads to happier users, more actionable feedback, and ultimately improved software.

REDUCED MEAN TIME TO RECOVERY

A core benefit of keeping environment state in version control is that rolling back to the last known good configuration when you experience problems is straightforward. This can dramatically reduce your mean time to recovery (MTTR) because you can fix issues quickly during an incident by rolling back, and then after your system is operating normally, you can troubleshoot.

IMPROVED POST-INCIDENT RESPONSE

During a firefight, the goal is simply to get everything operational and within acceptable limits. There's not always time to document enough details about what went wrong so that it can be improved. But a GitOps system defined as code can be replicated at any version. The faulty production version can be replicated after an incident in a test environment to do forensics and a root cause analysis.

GREATER RELIABILITY AND UPTIME

Manually configured infrastructure can be brittle and unreliable. With GitOps automation, human error is greatly mitigated, keeping your infrastructure uptime stable and allowing your IT Ops team to sleep at night.

SIMPLIFIED COMPLIANCE AND AUDITING

Too often organizations must tradeoff between moving fast and staying compliant. With GitOps, compliance and approvals can be automated, distributed, and conducted asynchronously so the pace of innovation keeps moving forward. With manual tools, auditing is highly painful. Pulling data from multiple places and trying to normalize it is extremely time consuming. Having all operations in Git gives you a one-stop shop for audit logs of every change so auditing can be effortless.

ENHANCED SECURITY

Leveraging Git's robust permission model makes it simple to grant, revoke, and track permissions for each environment. Beyond naive read/write access, the Git merge request workflow unlocks an additional ability to grant proposal access. Many users can be granted the ability to propose changes, while simultaneously keeping the pool of people who can enact those changes small.

GITOPS TOOLING AND TECHNOLOGIES

As a methodology, GitOps doesn't require any specific technology to implement. The following table lists examples of the types of tooling that you'll generally want to implement when adopting GitOps, along with some examples of each.

Table 1: Examples of GitOps tooling

TOOLING TYPE	EXAMPLE
Git code repository	Git
Git management tool	GitLab, GitHub, Bitbucket
Continuous integration tool	GitLab, Jenkins, CircleCI
Continuous delivery tool	GitLab, Spinnaker, Flux
Container registry	GitLab, Docker Hub
Configuration manager	Ansible, Chef, Puppet
Infrastructure provisioning	Terraform, Pulumi, AWS CloudFormation
Container orchestration	Kubernetes, Nomad

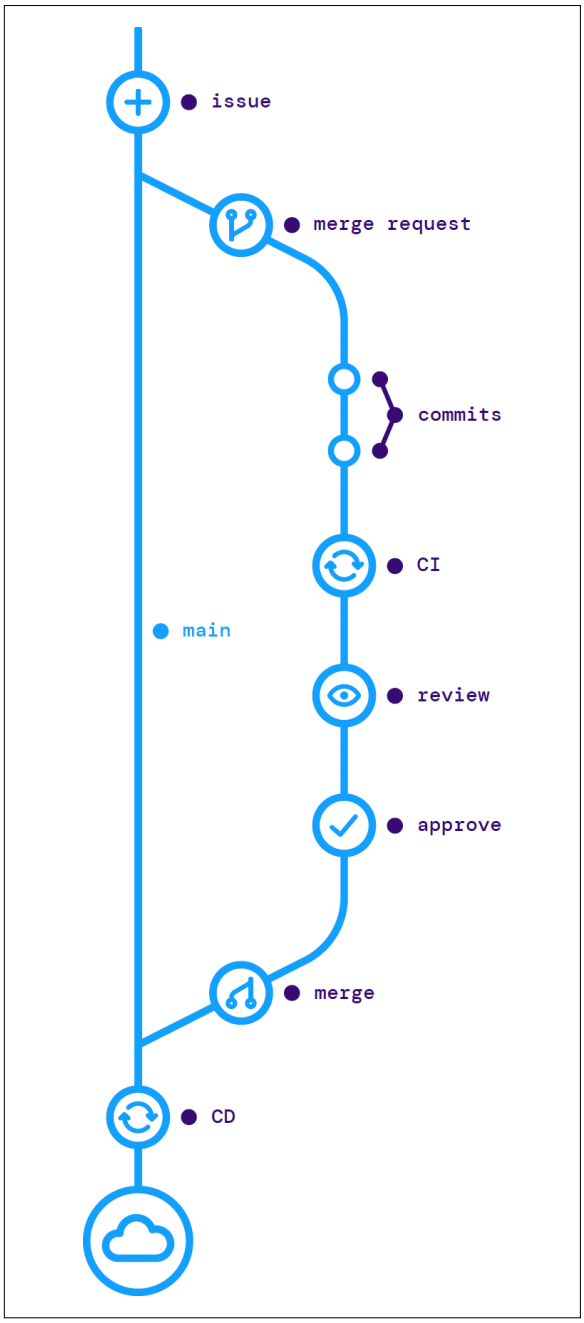
A COMMON GITOPS WORKFLOW

The power of GitOps is that the same workflows used to add features to a service or application can be used to configure and provision the environments where the software runs. This shared understanding of workflows and tooling across the organization drives many of the benefits that GitOps brings.

WORKFLOW FOR DEPLOYING A NEW FEATURE

1. A ticket is logged in the product issue tracker for the new feature.
2. A corresponding ticket is logged in the platform operations issue tracker to provision the infrastructure needed to support the new feature.
3. A branch is created in each respective repository to work on the feature and infrastructure changes.
4. Code is committed to each branch.
5. A merge request is created for each branch to perform code review and testing.
6. The CI/CD merge request pipeline runs automated tests and checks against the branch. Sophisticated tooling will display the results within the MR to aid collaboration and troubleshooting.
7. The MR serves as the central point of collaboration for peers and stakeholders. Reviewers comment on the general approach, as well as on specific lines of code.
8. The code is iterated on until it meets the standards to be merged. This typically means all automated checks have passed, all comments are resolved, and any needed approvals have been added.
9. The Git merge to the main branch triggers the CI/CD deployment pipeline, where an additional set of automated tests can be run.
10. The infrastructure pipeline configures and provisions the environment, while the product pipeline deploys the application code.

Figure 4: GitOps workflow



Workflows can vary depending on the size of the team, the repository setup, and if you are using Agent or Agentless tooling. The workflow above can be adapted and extended to meet the specific requirements of your organization.

CONCLUSION

GitOps is an operational model that leverages DevOps best practices used in application development for infrastructure automation. Using Infrastructure-as-Code, the environment definition is stored in a Git repository as the single source of truth. The merge request workflow is used for collaboration and compliance, while CI/CD automation not only deploys the application code, but also configures and provisions the underlying environments in which the code runs.

Adopting GitOps best practices means that developers and operations engineers can use familiar Git tooling to manage updates to software environments. Automation, along with asynchronous collaboration at scale, speeds up the pace of innovation to decrease lead times and increase deployment frequency. This increased agility allows business to respond to customer and market demands to build and maintain a competitive advantage. Whether teams are working side by side in the same office or distributed around the world, GitOps increases collaboration between development, operations, security, and all business stakeholders.

WRITTEN BY WILLIAM CHIA,
SR. PRODUCT MARKETING MANAGER, GITLAB

William Chia is a product-minded storyteller who's been crafting both code and copy since the late 90s. At GitLab, William leads product marketing for GitLab's Cloud Native and GitOps use cases. Prior to joining GitLab, William's career focused on serving technical users and buyers at companies like Twilio and Digium, the sponsor of open source Asterisk. Outside of work, you can find William cooking, practicing Taekwondo, and playing video games with his wife and three children.

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300