GitOps for Kubernetes

Essentials to Applications Hosted in Cloud-Native Ecosystems

CONTENTS

NINAD DESAI
STAFF ENGINEER, INFRACLOUD

Git is the most widely used version control system, with more than 80 percent of its market share in today's software industry. GitOps is a methodology which revolves around procedure and practices with Git at the center. It provides a fast and secure method for developers to maintain and update complex applications running in Kubernetes.

In this Refcard, we will dive into what GitOps means in the Kubernetes world, key principles, and the advantages for cloud-native ecosystems.

ABOUT GITOPS FOR KUBERNETES

Thanks to the DevOps movement, the focus on and responsibilities of developers has increased, and the focus has shifted rightly so to improving the developer experience. In the last few years, the number of tools and systems needed to manage version control, configuration management, Infrastructure as Code, CI/CD, and observability have drastically increased. This puts an additional burden on the developer as they now must oversee more and more components of the application.

Ideally, the developer focuses mainly on delivering business value via code. Thus, to improve developer experience, **GitOps** emerged where infrastructure and application configuration changes revolve around Git, i.e., the version control ecosystem. The focus is entirely on keeping Git as a single source of truth for infrastructure and application configuration changes. In an ideal world, this approach allows the developer to focus on working and pushing code to Git, and then, further deployment and tweaks would be taken care of by an automated mechanism.

Since Kubernetes and many other cloud-native technologies are almost entirely declarative, infrastructure definitions can be kept alongside application code in Git. Keeping your entire system in Git means that your development team uses familiar Git-based workflows and pull requests to deploy both application and infrastructure changes to Kubernetes. With the entire state of your cluster kept under source control, diff tools and synchronization agents can compare what's running in production with what's under source control — and when a divergence is detected between the two, an alert can be sent, effectively creating a feedback and control loop for managing your cluster.

PRINCIPLES OF GITOPS

GitOps is based on several core tenets:

DECLARATIVE DESCRIPTION

Thanks to modern Infrastructure as Code (IaC) and configuration management tools like Terraform, you can define, configure, and spin up your entire application declaratively via code. These declarative configurations will be stored ideally in your version control systems (VCS) like Git, GitLab, or GitHub.

Fully Automated, Zero-Effort K8s Deployments with Rafay's GitOps Service

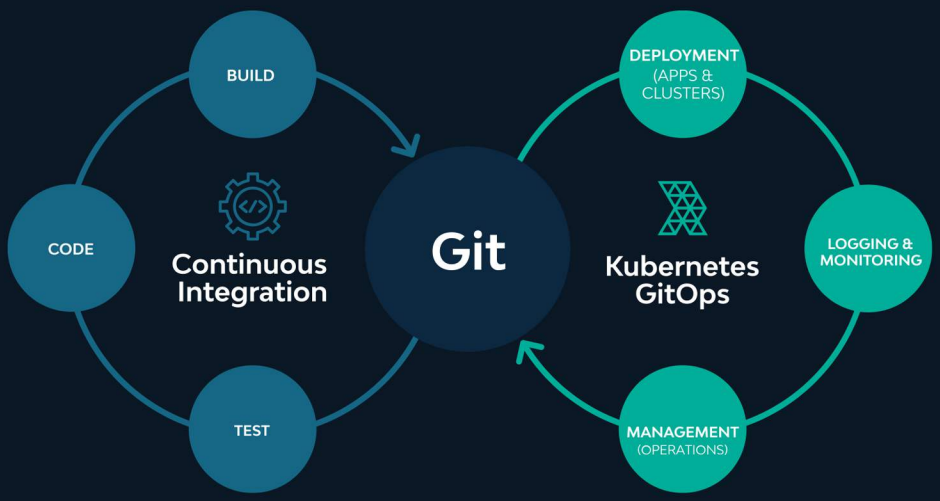
Rafay's GitOps Service provides infrastructure orchestration and application deployment through multi-stage, git-triggered pipelines. With Rafay, enterprises increase the speed of deployments and improve the consistency of their K8s infrastructure.

Multi-stage pipelines applying a variety of features, including stages, dependencies, conditions and templates.

Kubernetes and non-Kubernetes resources governed as part of your deployment pipeline.

A Fully managed service (SaaS) mitigating risk of repeating cycle of cloud native infrastructure and tool evolution.

Rafay GitOps Service



"By using Rafay's declarative GitOps model to power our IaC initiative, we've eliminated application and cluster configuration drift and accelerated deployments by 4x."

Vice President of Architecture,
Fortune 500 FinServ Company

TRY IT FOR FREE

RAFAY

SINGLE SOURCE OF TRUTH

In the GitOps world, version control systems (VCS) like Git should be the only source of truth. Whatever changes one wants to perform in the system should be declared and always committed first in Git or the GitLab repo alone. This helps to identify configuration drift as well as implement auditing to determine any possible changes or misconfigurations conducted outside VCS (i.e., Git) and restore back to normal if needed.

CANONICALLY VERSIONED DESIRED SYSTEM STATE

Now, by being able to declaratively store your desired system state in Git, you can easily apply versioning — as one does to application code packages — to your system configuration state, i.e., infra config code present in Git. This makes rollbacks easier in case your desired state in Git results in system instability for your actual state present in the cluster.

For example, let's say you have stored your current system state with version number 1.0.0 (semantic versioning) in Git. Now you want to apply a minor patch to your system state, i.e., application configuration. So you can commit that change to Git with version number 1.0.1 and push those changes. In case 1.0.1 does not bring the expected results, then you can just use "Git revert" to rollback those changes from Git. Then your actual state from the cluster will also get changed back to 1.0.0 as it was previously.

AUTOMATICALLY APPROVED CHANGES

Once you have the declared state kept in Git, the next step is to have the ability to automatically apply any state changes to your system. What's significant about this is that you don't need specific cluster credentials to make a change to your system. With GitOps, there is a segregated environment, and the state definition lives outside of it. This allows your team to separate what they actually do from how they are going to do it.

CONFIGURATION DRIFT DETECTION

A GitOps approach suggests that your declarative configuration files and application code should be stored together in the Git repository. This is what we call the "desired state of your system". Once you deploy all of this into the cluster and set up your application ecosystem, we call it an "actual state of your system". The "Configuration drift" is a term that commonly describes the gradual changes that we unknowingly and unnoticeably bring to the actual state of the system, which brings inconsistency and unpredictability between the actual and desired state. Our GitOps system of choice should ideally be able to detect, notify, and possibly auto-correct the actual state to make it consistent and match the desired state again.

BENEFITS OF GITOPS FOR KUBERNETES

Kubernetes, also known as K8s, is an open-source system for automating the deployment, scaling, and management of containerized applications. From 2013 to 2014, a revolution in software infrastructure emerged with tools such as Docker that demonstrated the capabilities of containers, thus exposing a new way of packaging and isolating application services.

This led to the birth of designing distributed systems. Containers took the world by storm and introduced new ways to orchestrate applications. When Google open-sourced Kubernetes in 2014, it grabbed the attention of developers and quickly became the de facto standard by 2016 — thanks to the many open-source communities that made it what it is today.

Kubernetes brings much-needed container orchestration capabilities like:

- Automated rollouts and rollbacks of deployments
- Service discovery and load balancing capabilities
  - As Kubernetes gives its own IP address to Pods and services, it can cross load-balance them
- Storage orchestration capability
  - I.e., mounting the storage system of choice
- Secret and configuration management
  - Note: *Without* rebuilding Docker images or exposing secrets
- Self-healing abilities
  - I.e., restarting of the failed containers, replacing, or rescheduling Pods in case of underlying nodes issues
- Horizontal scaling capabilities to scale up and down based on CPU usage
- Declarative in nature

This list continues to grow with every release. Let's now explore the key benefits GitOps brings to Kubernetes.

INCREASED SPEED AND PRODUCTIVITY

Continuous deployment automation with integrated feedback and control loops speeds up your deployment frequency. Declarative definitions kept in Git allow developers to use familiar workflows, reducing the time it takes to spin up new development, test environments, or deploy new features to a cluster. Teams can ship more changes per day, and this translates into a faster turnaround for new features and functionality for the customer.

SELF-SERVICE

With the existing CI/CD approach, the development team remains dependent on the operations team to deploy new changes in production. GitOps empowers teams to become more efficient and strategic. It enables development teams to be more self-service in nature as they would be less dependent on platform and operations teams to deploy and handle their code and configuration changes in production. All they have to do now is commit their changes to the Git repo and merge PR — the rest will be taken care of by GitOps tools.

DECLARATIVE APPROACH

Kubernetes itself is declarative in nature. A declarative approach suggests that you should only declare what you want to achieve vs. how it should be achieved, and then the automated system should take care of it ahead of time. The opposite of declarative is an imperative approach where you try to define the sequence of steps, which unfortunately is an additional overhead on the operator. GitOps promotes a declarative approach. Every object you use in GitOps can be coded and declared in your Git repo; then your GitOps system will create those for you.

OBSERVABILITY

Observability from a GitOps perspective is the ability to constantly have access to monitoring the actual state present in the cluster and the ability to compare it to what was desired. GitOps helps to measure and monitor what's running on the cluster and notify teams if it`s desirable or undesirable. Every GitOps tool nowadays comes up with the ability to monitor and notify us in this way.

COMPLIANCE AND AUDITING CAPABILITIES

GitOps helps to follow the compliance and auditing capabilities as needed and regulated in different domains. In GitOps, whatever is stored in Git is the only source of truth. So, this makes the audit process simple: The auditor can analyze the desired state by observing and focusing on the source code repository. They can also determine the current state of the system by reviewing the underlying infrastructure provider as well as the Kubernetes cluster state.

MULTI-CLUSTER CONFIGURATION MANAGEMENT

Being declarative in nature, Kubernetes helps most larger and enterprise organizations use multiple clusters for different business and team needs. Managing these clusters and keeping configuration and organizational security and other policies consistent across these clusters is a big ask for the Ops team. GitOps takes this burden away by letting the GitOps agent running across these clusters handle everything on your behalf, and so, it is a great fit from a multi-cluster configuration management perspective.

INFRASTRUCTURE AS CODE

In the Kubernetes world, GitOps is getting used to overall automate the process of infrastructure provisionings like creating deployments, services, and every possible Kubernetes object. DevOps and SRE teams are adopting GitOps so that you can store all your infrastructure configuration files as code. GitOps innately has Infrastructure as code as one of its characteristics.

STRONGER SECURITY GUARANTEES

Git's firm correctness and security guarantees — backed by the strong cryptography used to track and manage changes, as well as the ability to sign the changes to prove authorship and origin — are key to a correct and secure definition of the cluster's desired state. If a security breach does occur, the immutable and auditable source of truth can be used to recreate a new system independent of the compromised one, reducing downtime and allowing for better incident response.

Also, separating responsibility between packaging software and releasing it to a production environment embodies the security principle of least privilege, reducing the impact of compromise and providing a smaller attack surface.

DISASTER RECOVERY

Disaster recovery (DR) is a practice that every organization must have in place to recover from an event that negatively affects business operations. The goal of DR methods is to enable the organization to regain the use of critical systems and IT infrastructure as soon as possible after a disaster occurs.

Now how does GitOps help achieve the same? Well, it helps in the recovery of infrastructure environments as the entire environment and components we would have present are declaratively defined in our Git repository. So disaster recovery in case of such events becomes as easy as simply reapplying the configuration field present in Git to quickly restore your ecosystem.

CONFIGURATION MANAGEMENT IN KUBERNETES

The way Kubernetes handles deployments lends itself very well to GitOps workflows. For example, when a group of configuration updates are made by a human operator, the Kubernetes orchestrator will keep applying those changes until the cluster's state is converged to the updated configuration made by the human. The same is true for any type of Kubernetes resource.

Kubernetes deployments have the following properties that make them perfect for GitOps-style deployment workflows:

- **Automation**: Kubernetes provides a built-in mechanism for automating the deployments. A Kubernetes cluster applies a set of changes in the correct order and in a timely manner.
- **Convergence**: Kubernetes will keep trying to make the update until it eventually succeeds.
- **Idempotence**: Multiple and simultaneous convergence instances will all have the same outcome.
- **Determinism**: Assuming that the cluster has adequate resources available, the updated cluster state will depend only on the desired state.

Kubernetes deployments can also be extended and automated using Kubernetes Custom Resource Definitions (CRDs) with the operator pattern. These agents can then be used to automatically detect and apply configuration changes from outside of the system when you need them, essentially creating feedback and control loops.

Managing existing application and infrastructure configurations, making changes across them in different release cycles, and repeating

similar structures for all microservices — all these tasks can be error-prone and laborious. To solve these different application configuration conundrums, there are Kubernetes configuration management and packaging tools available. Let's take a look at a few popular ones below:

HELM

Helm is based on the parameterized templating approach where all resource definition files of applications are templated to make them customizable based on the requirement. Nowadays, it is viewed as the de facto package manager for the Kubernetes ecosystem. It acts just like APT/YUM/RPM packet manager but for Kubernetes.

It runs on the concept of Helm Charts and Templates, which are shown below. When the Templates are combined with the values, these templates will help to generate valid Kubernetes manifests based on values/variables. You can find more about them here.

HELM FILE STRUCTURE

test-chart Example

```
```
test-chart/
├── charts   #Required
├── Chart.yaml   #Required
├── templates   #Required
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests
│   └── test-connection.yaml
└── values.yaml   #Required
```
```

KUSTOMIZE

Apart from the Parameterized Templating, another approach to manage is Overlay Configuration. Kustomize is a configuration management tool based on this approach one.

Kustomize works off a concept of "where, what, and how" to refactor specific Kubernetes manifests. The "where" to refactor/change are the base manifests, e.g., a deployment.yaml. The "what" to change are the overlays or small snippets of YAML to change, e.g., a replica count, volume mounts, etc. The "how" to change are the kustomization/config files.

Kustomize file structure example:

```
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
└── overlays
    ├── dev
    │   ├── replica-count.yaml
    │   └── kustomization.yaml
    ├── production
    │   ├── replica-count.yaml
    │   ├── kustomization.yaml
    └── staging
        ├── kustomization.yaml
        └── replica-count.yaml
```

JSONNET

Jsonnet is a programming language open-sourced by Google that provides configuration management as one of its main features. Its use is not specifically limited to Kubernetes alone (although, it's been popularized by Kubernetes). You can consider Jsonnet as a combination of JSON plus templating. Jsonnet lets you leverage the best of what you could do with JSON. It's, again, declarative in nature.

Unfortunately, Jsonnet is not widely adopted in the Kubernetes community as love for YAML across the cloud-native community has dominated over the last few years.

GITOPS FOR KUBERNETES ESSENTIALS

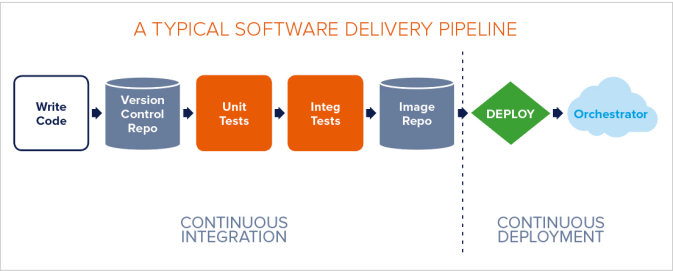
Let's deep dive into how GitOps workflow is different from our traditional CI-CD flow. We will also review core benefits over the traditional CI/CD approach.

TRADITIONAL CI/CD WORKFLOW OVERVIEW

Before diving further, let's first understand how traditional CI/CD works as most organizations that set off on their journey to continuous delivery normally start by automating a CI/CD pipeline.

In this simplified example, let's say there is a single microservice repository that bundles the microservice's code with its deployment YAML manifest files. YAML files, if you recall, are what define or declare how the microservice runs in the cluster. When the developer pushes the code to Git, a continuous integration tool kicks off unit tests that eventually build the Docker container image that gets pushed to the container registry.

Figure 1



With this typical CI/CD push-based pipeline, Docker container images are then deployed to the actual cluster using some sort of bespoke bash scripts or through some other method that talks directly to the Kubernetes' API.

COMMON CI/CD CHALLENGES

The typical CI/CD approach brings some of its own challenges as listed below:

SECURITY

With this approach, your CI tooling pushes and deploys images to the cluster. For the CI system to apply the changes to a cluster, you have to share your API credentials with the CI tooling. That means your CI tool becomes a high-value target. If someone breaks into your CI tool, they will have total control over your production cluster, even if your production cluster is highly secure.

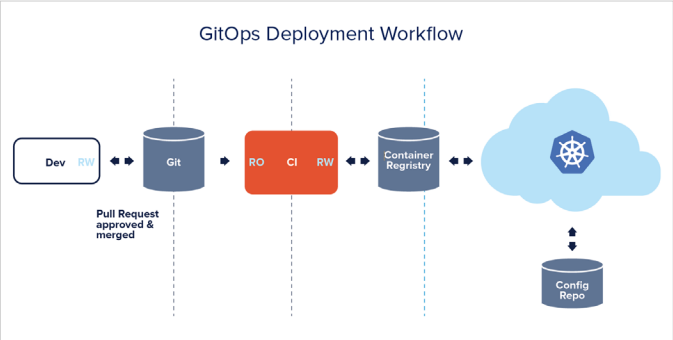
DISASTER RECOVERY

What happens when you need to recreate your cluster in the case of a total meltdown? How do you restore the previous state of your application? You would have to run all of your CI jobs to rebuild everything, and then re-apply all of the workloads to the new cluster. The typical CI pipeline doesn't have its state easily recorded like when you're using GitOps.

GITOPS DEPLOYMENT WORKFLOW OVERVIEW

The GitOps core machinery is in its CI/CD tooling, with the critical piece being continuous deployment (CD) that supports Git-cluster synchronization. It is designed specifically for version-controlled systems and declarative application stacks. Every developer on your team is familiar with Git and can make pull requests. Now, they can use Git to accelerate and simplify application deployments to Kubernetes as well.

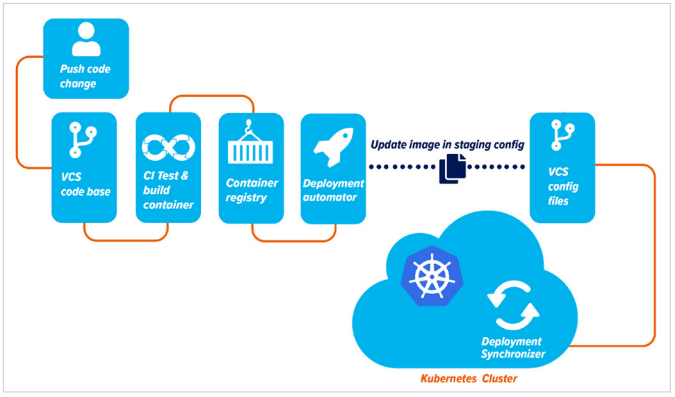
Figure 2



Here is a typical developer workflow for creating or updating a new feature:

1. A pull request for a new feature is pushed to GitHub for review.
2. The code is reviewed and approved by a colleague.
3. After the code is revised and re-approved, it is merged to Git.
4. The Git merge triggers the CI and build pipeline, runs a series of tests, and then eventually builds a new image and deposits the new image to a registry.
5. The Deployment Automator watches the image registry, notices the image, pulls the new image from the registry, and updates its YAML in the config repo.
6. The Deployment Synchronizer detects that the cluster is out of date, pulls the changed manifests from the config repo, and deploys the new feature to production.

Figure 3: GitOps workflow in Kubernetes



GITOPS SECURITY

The image is pulled using read-only access to the container registry. The CI tool is not granted cluster privileges and, therefore, is not introducing significant security risks to your pipeline.

SEPARATION OF PRIVILEGES

GitOps separates CI from CD. This is one reason as to why it is a more secure method for deploying applications to Kubernetes. The table below shows how GitOps separates read/write privileges among the cluster, CI and CD tooling, and the container repository, providing your team with a secure method for creating updates.

CI TOOLING: TEST, BUILD, SCAN, PUBLISH	CD TOOLING: RECONCILIATION BETWEEN GIT AND THE CLUSTER
Runs outside the production cluster	Runs inside the production cluster
Read access to the code repository	Read/write access to configuration repo
Read/write access to the continuous integration environment	Read/write access to the production cluster
Read/write access to container repository	Read access to image repo

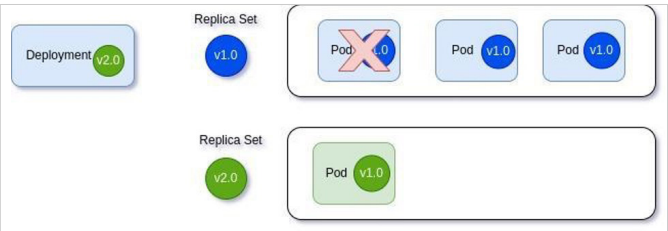
DEPLOYMENT STRATEGIES

When you're planning to define the overall application deployment and upgrade process, it's important to first decide which deployment strategy fits well for your application and for your end users. Broadly, in GitOps, the below Kubernetes deployment strategies are well supported and can be benefited from adoption:

ROLLING UPDATE

By default, Kubernetes comes up with a basic rolling update strategy, which is generally adopted and used across the industry. It basically creates a new replica set. While the new replica set is creating the Kubernetes pod with the new version, the old replica set scales down the old one. There are some challenges with this approach like less control over at what speed the rollout will happen, the ability to control traffic flow to the new version, and access to external monitoring tool-based metrics that verify if the new rollout is successful or not.

Figure 4



BLUE-GREEN DEPLOYMENT

Blue-green deployments allow teams to run the old version alongside the new version of your application; microservices run at the same time and then switch user traffic from the old to the new version. Once the new version is perfectly stable, then the old one will be taken down. There are tools from the GitOps family like Flagger and Argo-Rollout, which help to achieve and implement this approach. This strategy is fairly simple, has excellent rollback, and requires minimal downtime in best-case scenarios.

CANARY DEPLOYMENT

For canary deployments, the new version reduces risk by slowly rolling out changes to an initially small subset of end users before rolling it out to all users. The canary deployment strategy ties together the best of both blue-green and rolling update strategies, allowing fast rollback, minimal disruption, and optimized compute cost over blue-green.

Both blue-green and canary strategies can be also considered as various approaches to progressive delivery.

OBSERVABILITY AS A DEPLOYMENT CATALYST

An essential component of GitOps is feedback and control. But what is meant exactly by this? In order to have control so that developers can go faster, they need observability built into their deployment workflows. Built-in observability allows engineers to make informed decisions on real-time data. For example, when a deployment is being rolled out, a final health check can be made against your running cluster before committing to that update. Or an update that didn't go as planned can be easily rolled back to a previously stable state.

With a feedback control loop, you can effectively answer the following questions:

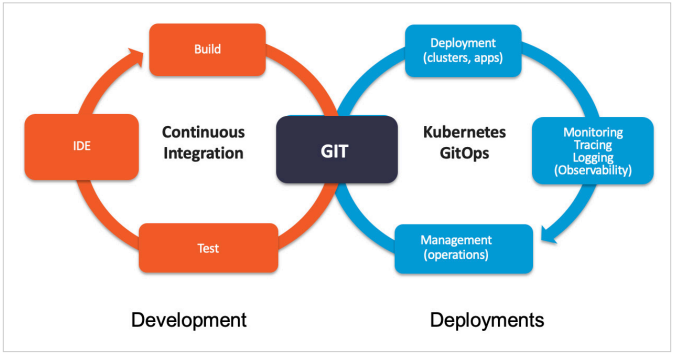
- How do I know if my deployment will succeed?
- How do I know if the live system has converged to the desired state?
- Can I be notified when this differs?
- Can I trigger a convergence between the cluster and source control?

While Git is the source of truth for the desired state of the system, observability provides a benchmark for the actual production state of the running system. GitOps takes advantage of both to manage applications.

With GitOps, divergence and convergence are achieved with a set of "diff" and "sync" tools (e.g., kubediff, terradiff, and ansiblediff) that compare the intended state with the actual state. Diff tools are also used to alert developers when the deployment is out of sync.

A feedback loop with diffs and built-in observability looks something like this, where observability provides feedback for developers and operators to make key decisions about deployments and the system:

Figure 5: Feedback and control loop



Because about-to-be-released services or updates can be observed in real-time within the running cluster, you can deploy with confidence and deliver higher-quality features.

Observability is a principal driver of the continuous delivery cycle for Kubernetes since it describes the actual running state of the system at any given time. After new features and fixes are merged to Git, the deployment pipeline is triggered, and once the image is ready to be released, it can be observed in real-time against the running cluster. At this point, the developer may return to the beginning of the pipeline based on this feedback, or they may deploy and release the image to the production cluster.

CONCLUSION

To summarize, we learned how GitOps assists continuous delivery and implements Git as a single source of truth for declarative infrastructure and applications. As an operating model for Kubernetes and other cloud-native technologies, GitOps provides a set of best practices that unify deployment, management, and monitoring for containerized clusters and applications.

With Git as the central hub for your automated CI/CD pipelines, developers can make pull requests to accelerate and simplify application and infrastructure deployments to Kubernetes. In the coming years, the cloud-native world is going to quickly adopt GitOps as the role of CI/CD brings better developer experience and faster feature releases, resulting in improved developer efficiency, a lean set of tools to maintain, visibility across the infrastructure to internal stakeholders, and enhanced collaboration amongst teams.

WRITTEN BY NINAD DESAI,
STAFF ENGINEER, INFRACLOUD

Ninad is Certified in Kubernetes Administration (CKA) as well in Kubernetes development (CKAD). In his current role, he helps customers to adopt cloud-native technologies. At InfraCloud, Ninad is responsible for designing/scoping any new solution and sharing plans/timelines. This involves helping to make technical directions on different DevOps and SRE project teams, overlooking client engagement health, ensuring there are no technical blockers for teams, and mentoring and coaching new team members. He is currently leading solution engineering, platform engineering, and managed services teams from a DevOps and SRE perspective. On a day-to-day basis, he works and has a specific interest in progressive delivery, GitOps, Kubernetes, and the observability stack.

DZone
600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.