

GitOps

Does delivering software at a rapid pace make you uneasy? There is so much in the stack, such as Docker, Kubernetes, Jenkins, CircleCi, TravisCI, FluxCD, and ArgoCD.

Many leaders in the IT world are still using lagging systems because they fear failed releases which are very common in IT.

However, this does not have to be the case as we are now going to cover GitOps, which has become very famous due to its ability to deliver the microservices to Kubernetes Declaratively.

First, let us remind ourselves of the differences between imperative and declarative techniques of deploying microservices in the Kubernetes cluster.

The imperative is where we define and execute the deployment of the microservices in the cluster using the Command-line interface (CLI).

This means that we use commands that we run in the terminal to determine the desired state and control the running state in the cluster.

Example,


In our previous sessions, to execute the desired state in the native Kubernetes cluster, we have always used the CLI and command such as

Kubectl create namespace nathilda

Kubectl create -f deployment.yml (name of the deployment)

Kubectl create -f pvc.yml

```
$ kubectl create namespace nathilda
namespace/nathilda created
$ kubectl create -f deployment.yml
```



To accomplish this, we use the CLI to clone the repository containing the manifests files and then manually run the kubectl commands.

Doing so means that we are deploying to the cluster imperatively, and this is very RISKY, SLOW, AND NOT CONVENIENT FOR BIG COMPANIES like Apple, Google, and American Express.

Below find examples of how to deploy microservices in the cluster imperatively.

```
Terminal +
$
$ git clone https://github.com/joshking1/gitops-foundations-env-2892009.git
Cloning into 'gitops-foundations-env-2892009'...
remote: Enumerating objects: 383, done.
remote: Counting objects: 100% (67/67), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 383 (delta 48), reused 46 (delta 43), pack-reused 31
Receiving objects: 100% (383/383), 75.83 KiB | 5.05 MiB/s, done.
Resolving deltas: 100% (209/209), done.
$ ls -l
total 4
drwxr-xr-x 9 root root 4096 Jun  7 14:23 gitops-foundations-env-2892009
$ ls
gitops-foundations-env-2892009
$ cd gitops-foundations-env-2892009/
$ ls
arc    CONTRIBUTING.md  flux    NOTICE    setup.sh
argo  flagger           LICENSE README.md  terraform
$ cd flux/
$ ls
deployment.yaml namespace.yaml README.md service.yaml
```

```
Terminal +
$ ls
deployment.yaml namespace.yaml README.md service.yaml
$ kubectl create -f deployment.yaml
Error from server (NotFound): error when creating "deployment.yaml"
: namespaces "flux-exercise" not found
$ kubectl create namespace flux-exercise
namespace/flux-exercise created
$ kubectl create -f deployment.yaml
deployment.apps/gitops-foundations created
```

Alternatively, we can create manifests manually on the CLI (terminal) and use the VI or NANO editors to edit and save the codes.

Example

```
--
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment # name of the manifest
metadata:
  name: huguette-dev # This is the name of the APP/pods
  namespace: josh
spec:
  selector:
    matchLabels:
      app: huguette-dev # The name of APP/pods
  replicas: 1 # tells us the number of pods matching the template. In this case 1 pod/container
  strategy:
    type: RollingUpdate # bluegreen Canary deployment method
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1

  template:
    metadata:
      labels:
        app: huguette-dev
    spec:
      containers:
      - name: huguette-dev
        image: matttrayner/lamp # This image will be given by the company
        imagePullPolicy: Always
        ports:
        - containerPort: 80 # This is the container port
---
```



Terminal



```
$ touch deployment.yml
$ vi deployment.yml
$ vi deployment.yml
```

After saving the content, we can deploy the desired state to the Kubernetes cluster imperatively.

Let us deploy the desired state imperatively.

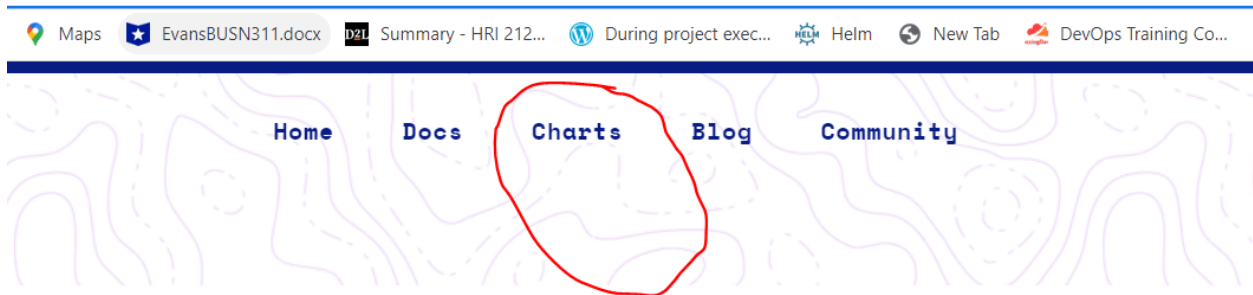
```
$
$
$ touch deployment.yml
$ vi deployment.yml
$ kubectl create namespace josh
namespace/josh created
$ kubectl create -f deployment.yml
```

Let deploy the desired state to Kubernetes using HELM

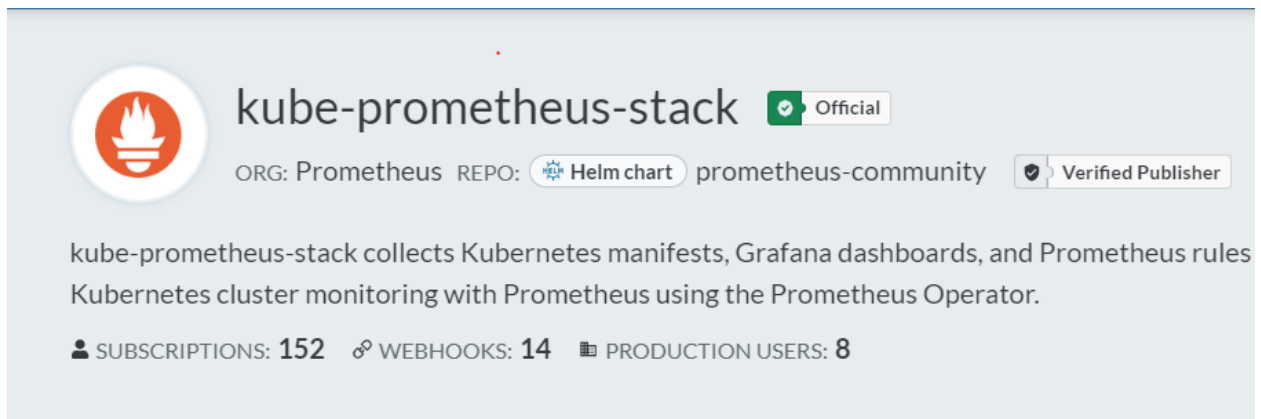
Use this link to go to Helm Chart

<https://helm.sh/>

Click on the Charts



Search and select Kube-Prometheus-stack



Imperative Commands to run

Get Helm Repository Info

```
helm repo add Prometheus-community https://prometheus-community.github.io/helm-charts
```

```
helm repo update
```

```
$  
$ helm repo add prometheus-community https://prometheus-community.  
ithub.io/helm-charts  
  
Command 'helm' not found, but can be installed with:  
  
snap install helm  
  
$  
$ helm repo update  
  
Command 'helm' not found, but can be installed with:  
  
snap install helm
```

Install Helm Chart

```
helm install [RELEASE_NAME] Prometheus-community/Kube-Prometheus-stack
```

```
$  
$  
$ helm install josh-wahome prometheus-community/kube-prometheus-sta  
ck
```

We have deployed the desired state to the cluster **Imperatively**, which is VERY



WRONG. 

Disadvantages of deploying the desired state imperatively to the Native Kubernetes Cluster

Security



Deploying imperatively goes against the Kubernetes security practices.

For you to deploy the desired state, you must access the CLI of the cluster, and this can pose high-security issues to the production environment, especially if everybody in the team must access the CLI to deploy THE DESIRED STATE.

No Source of Truth

Deploying imperatively means we cannot track the changes being deployed in our native Kubernetes cluster. This means anybody can run the commands and drift the cluster from the Desired State.

Production Outage Surprises

When we cannot track the changes being deployed in the cluster, it is normal to expect Production outage surprises because deploying imperatively leaves the company with no knowledge of what is running in the cluster. When this happens, it is difficult to know what is causing the outage because there is no source of truth that engineers can refer to. This makes it difficult for the IT team to quickly fix the production issues and return the services to normal faster.

Causes of Production Outage in Most Companies

Human Error

The most significant and frequent cause for IT downtime is human error. 49 percent of respondents in the ITIC survey reported that their server went out because of human error. While human error is not completely avoidable, errors often happen because employees choose not to follow set protocols or standards.

If your company wants to avoid network downtime due to human error, the following steps can help:

- Document each task step-by-step to ensure that a standard procedure is always followed.
- Ongoing assessment for IT employees can help them stay abreast of the latest updates, device configurations, and security challenges.
- Having secure access policies can ensure that the network does not fall into the wrong hands.
- If your company is BYOD, ensure that your entire team's policies are standard and well-understood.

Human Error: Whether accidental or due to negligence, human error is one of the most common causes of unplanned downtime.

An employee unintentionally deleting data, accidentally unplugging a cable, or not following standard protocols can lead to costly downtime.

Human error is unavoidable, but with regular training and a well-documented IT checklist or policies, its frequency can be reduced.

How can we reduce the production outages in most companies?

- The future of how we manage our configurations is not the hamster wheel of “rip and replace the old one.”
- However, creating a single record of configuration truth where operators can go when issues arise.
- Where engineers who are on-call can use as the first place to understand who did what when.
- A place where engineers can quickly identify which team or person might have implemented a recent change and resolve issues without wasting time on troubleshooting.
- A place where security and compliance teams can ensure only the right people are making appropriate changes.
- A single record of change is what most companies need to understand what is happening across their systems.

That single record/source of truth is the concept behind the famous GitOps Concept.

What is GitOps



Principles

Practices

Tools

DevOps

that DevOps is the foundation of GitOps.

Let us first remind ourselves what DevOps is.

DevOps is a Software Development Lifecycle (SDLC) that allows the company to deliver the software at a faster rate because its core principles are centered around automation, testing, monitoring, collaboration, and communication using different tools.

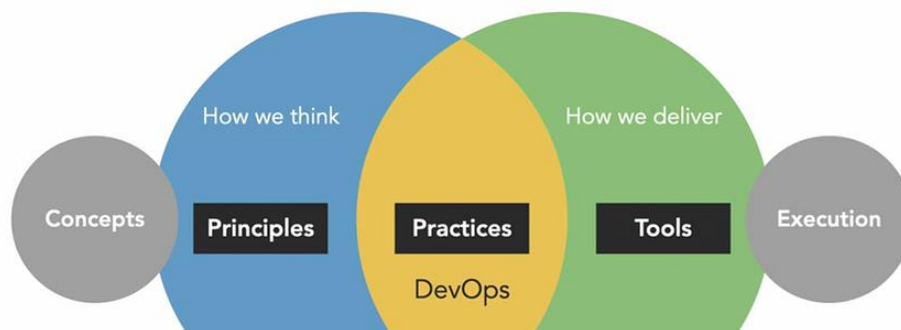
DevOps is more of a culture/methodology rather than a technical concept. Continuous planning, automation, integration, delivery, testing, and monitoring allow the company to deliver software faster. The software delivered is reliable and quality and usually experiences fewer maintenance issues.

As a culture, DevOps influences everybody in the company by impacting how they think about the final product to be delivered.

This is why a DevOps Engineer must be able to work with different members of different teams to help them understand the main concept behind DevOps culture and practices.



To simply do this, we can think of DevOps as a concept and an execution.



What about GitOps

GitOps concept usually focuses on Git as the only source of truth. This means Git is at the center of culture and practices in the company.

It is important to know that GitOps is not a tool. It is a concept just like DevOps.

This means that GitOps will apply git centric approach to some common DevOps principles, Practices and Tools.

It is the same thing we have been doing; however, all our configurations will be stored in Git, and any change that needs to be implemented in the company must be executed through Git.

This makes Git the only source of Truth and any changes made to the infrastructures in the company are easy to track.

Not only can we track the changes made in the company, but we can also track the people making the changes to the configuration files to be deployed in the cluster or cloud.

Why Git?

Through Git, it is impossible to merge the changes to the main branch (let us avoid the word master branch - this is obsolete) without creating a pull request.

Also, the company's main branch will be protected all the time. It will require approval from different key team leaders, status checks to pass, and in some cases, deployment/provisioning to be successfully completed before merging can happen.

This means that unless the Code you are merging has passed all the checks, reviews, and approval, you cannot merge.

This is a great security feature that is helping the company to control how changes are being implemented, reducing the production outage surprises.

The concept behind GitOps is that all changes in the company are made through Git (Version Control System), and the changes made are automatically deployed.

Developers have been writing codes using editors such as Visual Studio Code and pushing the changes to the Source Code Management such as GitHub for over a decade.

GitOps just shift the practice to the operation side. Any change made to the configuration files in the main is automatically deployed to the operation side.


The Big Idea in GitOps


- System changes captured in version control (Git)
- Git stores desired system state
- System runtime mirrors the desired state stored in Git

You can look at the changes I made a few days ago.




● Merge pull request #5 from joshking1/revert-4-feature3

Revert "Changing output.tf"

 master (#5)

 joshking1 committed 6 days ago Verified

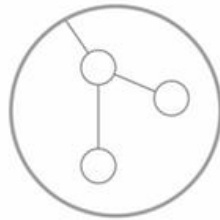
Showing 1 changed file with 1 addition and 1 deletion.

▼ 2   output.tf 

...	...	@@ -1,3 +1,3 @@
1	1	output "jenkins_ip_address" {
2	-	value = "\${aws_instance.jenkins-instance.public_ip}"
	2	+ value = "\${aws_instance.jenkins-instance.public_dns}"
3	3	}

For the GitOps concept to fully work, this equation must balance.

The GitOps Equation



Desired State

=



Running System

The Desired state must be equal to the Running state in the production or AWS or any other cloud.

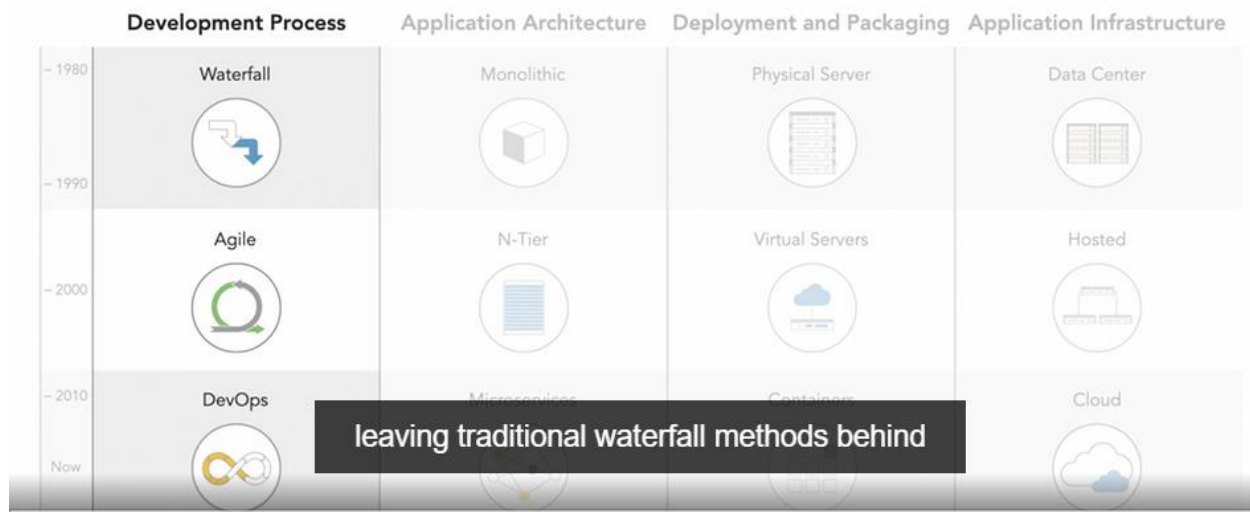
This makes it very easy for all the team to know where to refer to in case of any production outage. Everybody goes to Git because it is the only source of truth, and any changes made to the configuration files can easily be traced in Git.

Simply commit all your changes to Git, which runs in the system.

Where are we heading with GitOps?

Companies always look for better ways to deliver software faster because nobody likes to wait.

Evolution of Development and Deployment

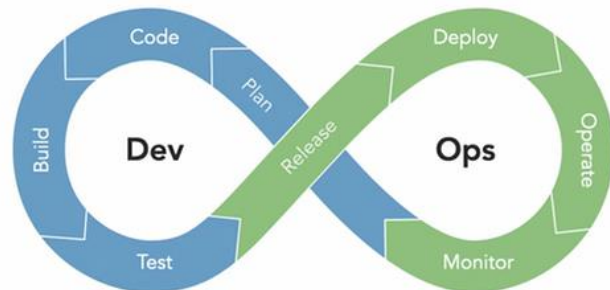


Teams using DevOps, Microservices application Architecture, and Containers/Application images to package their builds and application infrastructure like the cloud can deliver software much faster than teams still in the 2000 era.

Why GitOps?

- **Stabilizes** systems experiencing a high rate of change
- Applies **consistent** development workflow to operations
- Continuous deployment automation

without sacrificing system stability.



GitOps is focused entirely on production and plays a vital role in stabilizing systems that experience a high rate of change.

Using GitOps, which focuses entirely on the operation side, teams manage operation activities through the declarative infrastructure code stored in Git.

This means, in GitOps, there is no space for scripted Code.

Continuous deployment Automation is the key and is done by Declarative tools, not human beings.

GitOps allows the team to rapidly yet safely deliver the necessary changes to the system running in production without sacrificing system stability.

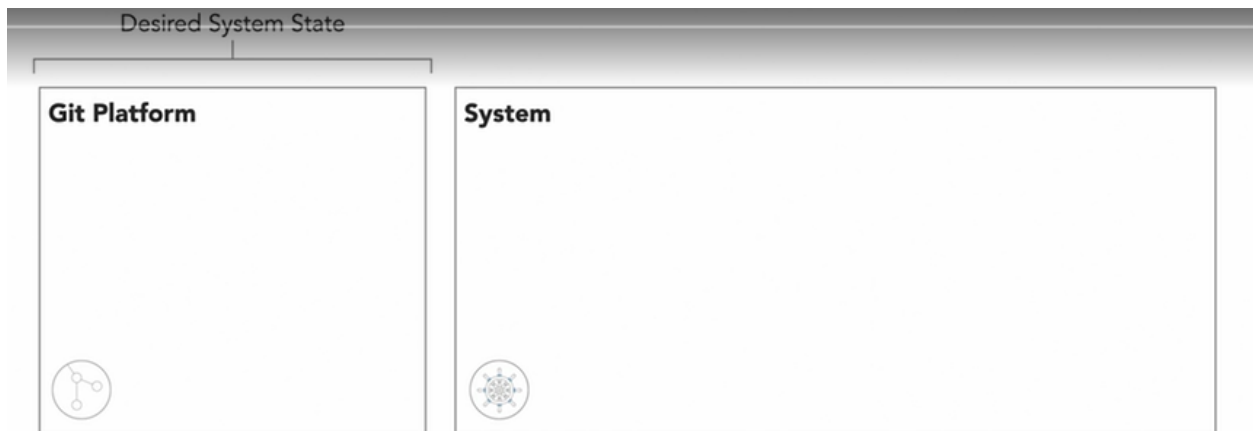
The concept of GitOps is new and was introduced in year 2017 through a blog that was entitled "Operation by Poll Request"

A poll is a quick request, while a pull is a slow demand.

One may poll asking if the information is immediately available, which can be pulled.

The distinction is not that the answer to a poll must be boolean, but that the answer to a poll is quick and readily available, or the answer will be denied.

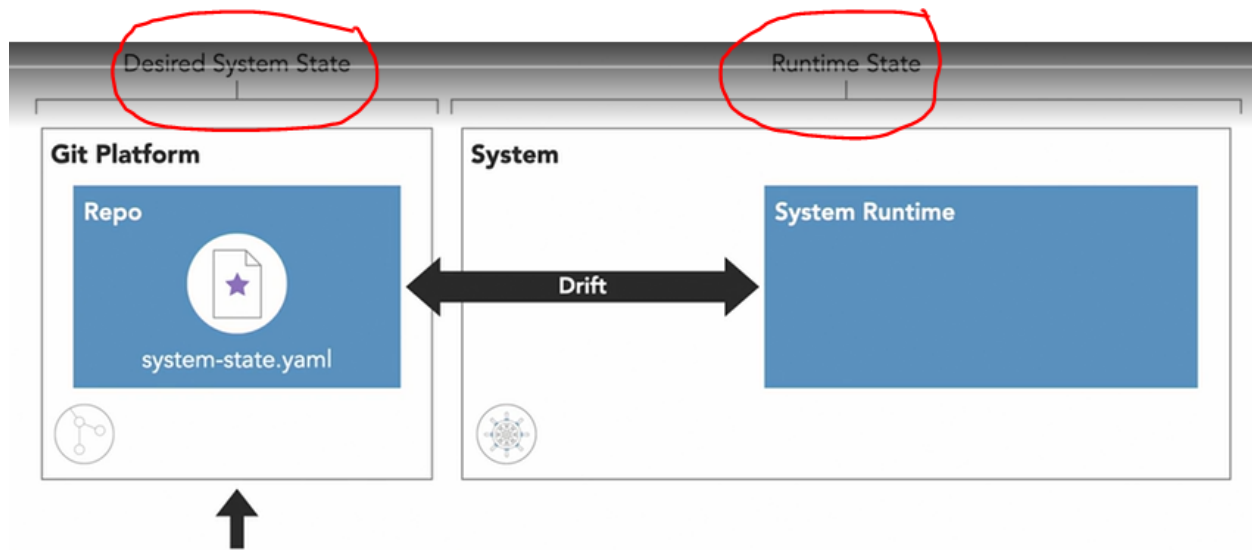
The future of GitOps is promising, so it is important to grasp the concept early.



Two words that are very important: Desired state and Running state.

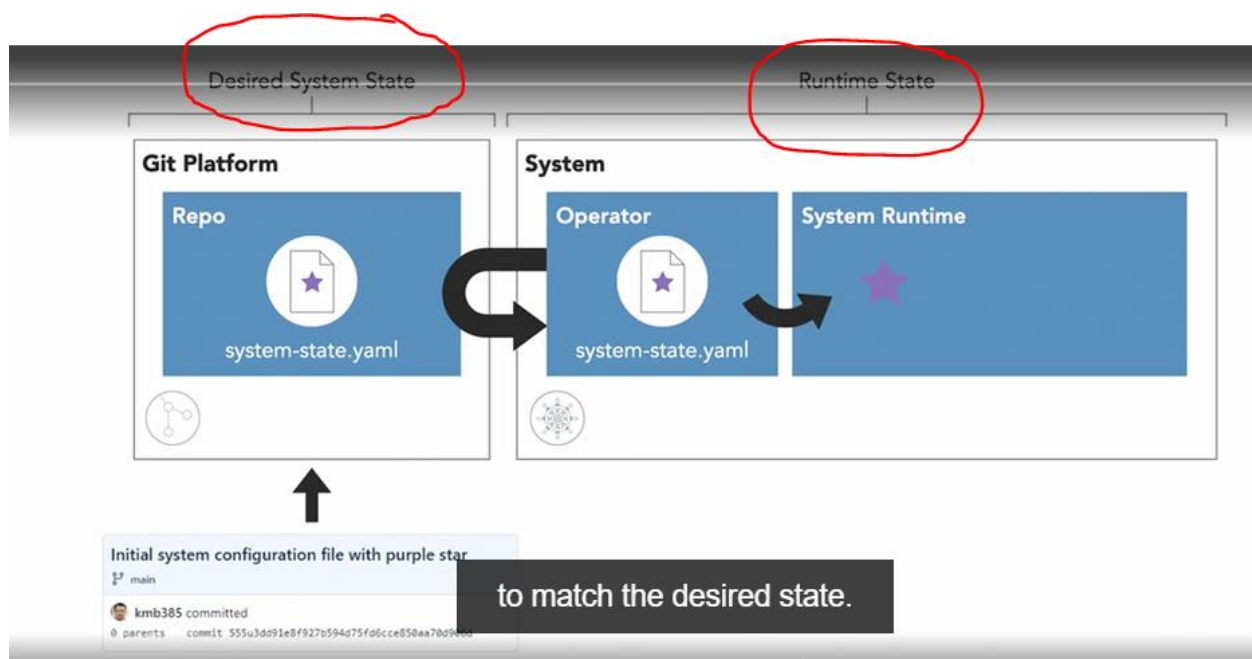
The desired state is always stored in the Git platform, while the running state is what runs the system.

Both desired and running states must be the same.

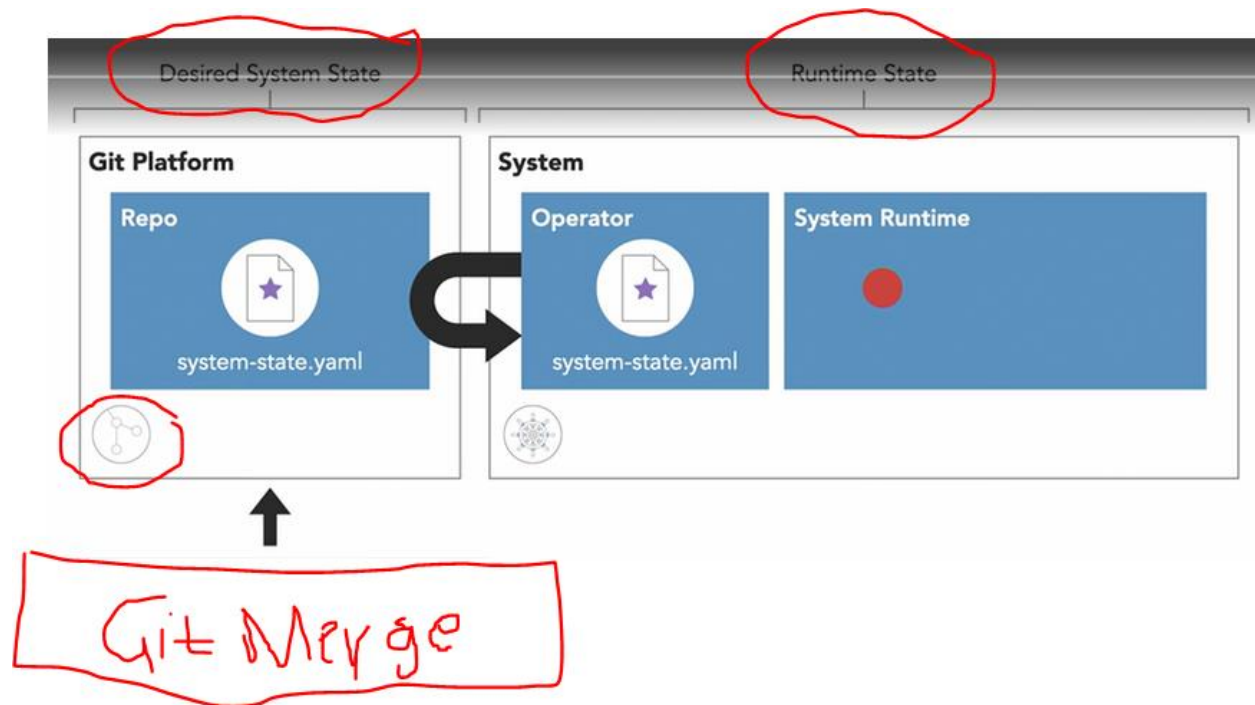


In the example, our desired state has changed after committing the infrastructure code, causing a drift in the system runtime. For the GitOps principle to work, both the Desired and the Runtime must be equal.

In GitOps, deployment is automatically done by the operator that runs inside the cluster/system. The system admin usually installs these operators.



To solve the drift, the operator will pull the Code describing the desired state inside the YAML file and deploy it to the Runtime of the system to match the desired state.



Over the weekend, something happens, and our system runtime drifts. Let us say we now have a red circle instead of a purple star. This is a major crisis that can cause a production outage.

However, in GitOps, you do not need to worry because the operator continually monitors the system.

The operator will notice the drift between the Runtime and the desired state in Git and will automatically correct the drift between them using the **ONLY SOURCE OF TRUTH**, which is the **DESIRED STATE STORED IN GIT**.

This monitoring and correction process continually running by the operator is known as the control loop or the reconciliation loop.

Let us summarize

Key Takeaways

- Desired state vs. runtime state
- Git stores code describing the desired state
- Automated change deployment
- Control or reconciliation loop corrects drift

Benefits of GitOps

Productivity – When you adopt the GitOps concept, every deployment starts with Git. Any change one wants to make starts with Git, and no other way can those changes be made.

This standardizes the workflow and change process, which is vital to the security and productivity of the company.

Operation of the cluster or systems is done through Git Commands.



This means that an engineer/developer to deploy or release the changes only needs to know a few Git commands. That all.

Imagine your delivery process looks like this, and only a few team members understand the whole process entirely.

It would not be a surprise that the team will deliver software slowly because nobody understands all the processes.



In GitOps, you need to know a few commands to release changes.

This allows everybody to release changes that enhance high team collaboration. On the other hand, frequent releases allow the software to be delivered faster.

Workflow is automated with tools, which means it is repeatable and consistent, making the system more predictable and less prone to human errors. It also increases the rate of deployment.

In case of an issue with the release changes, all it takes is a git commit that returns the system to the previous working state, undoing the changes that caused the system to fail.

If your system has a complete meltdown, GitOps allows you to rebuild the entire system using the infrastructure code stored in Git (THE ONLY SOURCE OF TRUTH).

In simple sentences, GitOps standardizes the operational processes to remove risk and avoid surprises.

This is not a birthday party or breakfast in bed kind of surprise.

It goes like this. It is on Friday, and we release some changes to be deployed to the system, and suddenly, the system starts to thrash. We are still imperatively, so we SSH into the system and discover that somebody made some changes 4 years ago that nobody knows about. That causes you to work over the whole weekend.

GitOps can help you avoid these kinds of surprises.

With GitOps, operations become transparent because Git is the only way to make a change. After all, the entire system is described in Git as infrastructure code.

Silos between Dev, Sec, and Ops can be broken down using Git as a collaboration tool.

With Git, members can review, discuss, and approve being made to the system using a pull request. Anybody with access can inspect the Code before it is merged into the main branch.



GitSecOps or DevSecOps

Same thing

GitOps provides a strong balance between the strong security control and the transparency the engineers and developers need to accomplish their tasks.

The workflow audit log and the team's changes are automatically available, making it easy to audit the system and know who made changes, the kind of changes they made, and to which infrastructure code precisely.

GitOps restrict how changes are applied to the runtime systems. Changes are automatically deployed through agents installed in the system and no other way.

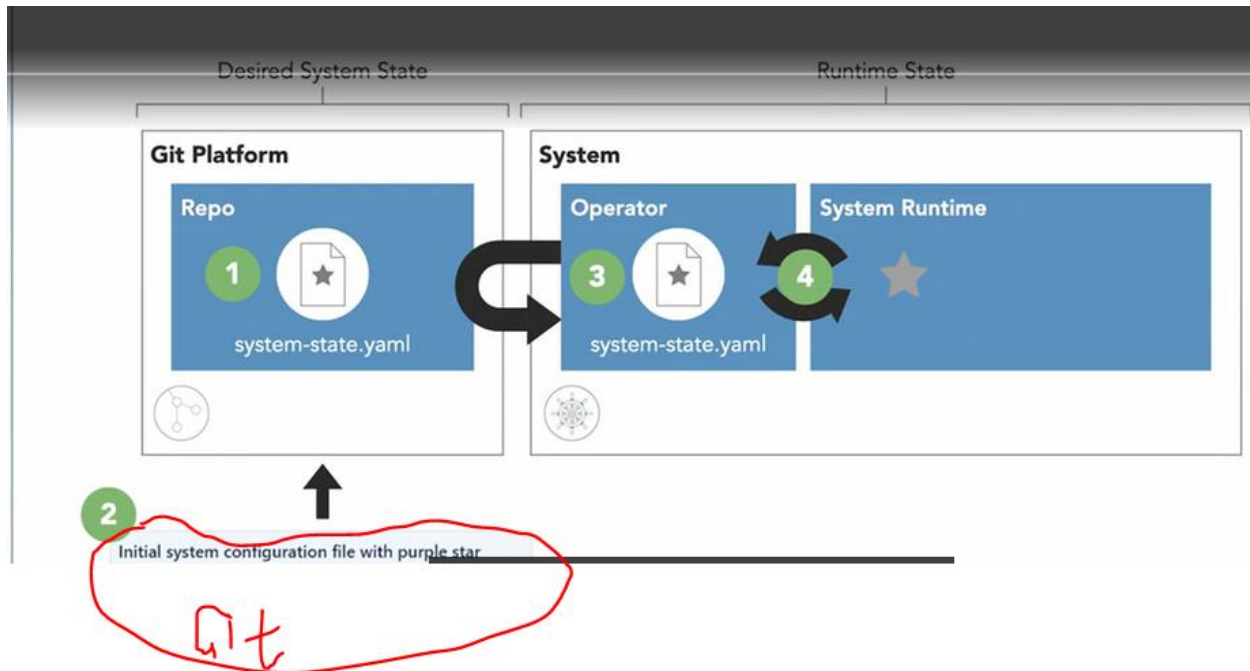
There is no need for more SSH as changes can be made in Git.

Also, no need for SSH to see the changes made in the system as everything is auditable and available in Git.

This is an added security feature that is vital to the runtime systems in the company.

Repository permissions control the change delivery.

GitOps Principles



Four key principles drive the Gitops workflow.

These are,

Key Principles

Declarative System Description

The whole system is described using data that identifies the desired end state.

Single Source of Truth

The desired state of the system is stored in Git. Any change to the desired state must be made through Git.

Automated Change Delivery

Changes to the desired state are automatically applied to the system.

Automated State Control

Software agents ensure the running state of the system does not deviate from the desired state.

Declarative System description

This is the blueprint of our infrastructure, usually available as configuration files containing the infrastructure code. This Code is the blueprint used to build our infrastructure.

Single Source of Truth

The section principle requires that we store the blueprint in Git. Also, the principle requires that all changes to the blueprint must be implemented through Git.

Automated Change delivery

This principle requires that all changes made to the blueprint stored in Git be deployed Automatically using GitOps operators. There are no imperative or manual changes in GitOps.

Automated State Control

The fourth principle requires that the running state stay aligned with the desired state through automation. If the desired state drifts away from what we have described in the Blueprint stores in Git, the operator within the system will heel it and return the running state in alignment with the desired state.

Adapting GitOps in AWS is the ideal way to manage infrastructure at scale in a cloud-based development process. By implementing a CI/CD pipeline for GitOps,

users can introduce the same automated and agile development to handle infrastructure configurations.