International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V73I9P102

Volume 73 Issue 9, 10-18, September 2025

Original Article

Extending GitOps Principles to Terraform Deployments

Ravindra Agrawal[1], Saurabh Verma[2]

[1]*Amazon Web Services Inc, Charlotte, North Carolina, USA.*
[2]*Amazon Web Services Inc, Columbus, Indiana, USA.*

[1]*Corresponding Author : raviagrawal86@gmail.com*

Received: 15 July 2025 Revised: 17 August 2025 Accepted: 04 September 2025 Published: 29 September 2025

Abstract *- GitOps has completely changed how we deploy and manage Kubernetes apps. It's made Git repos the single source of truth for Infrastructure, using the declarative approach. But when it comes to Infrastructure as Code deployments and especially Terraform, GitOps is pretty underutilized. This article digs into extending GitOps beyond just Kubernetes to build reliable, auditable Terraform workflows. Traditional CI/CD has some limitations when dealing with Infrastructure, and we are proposing a controller-based approach that brings all those GitOps benefits to Terraform operations. Through analyzing existing patterns and tools, organizations can get way better deployment reliability, security, and operational visibility. The article shows GitOps-driven Terraform deployments are superior for drift detection, rollbacks, and compliance management. GitOps has the potential to be the next big step in infrastructure automation.*

Keywords *- Devops, Gitops, Kubernetes, Argocd, Terraform.*

1. Introduction

Software deployment has come a long way, and for the most part, we have been looking for a combination of reliability, speed, and just getting things right operationally. With GitOps, our approach to deploying apps and managing Infrastructure has changed a lot. The idea of making Git repositories a single source of truth has proven to be a brilliant approach. Combined with declarative configs, GitOps has proven itself in Kubernetes land, especially with tools like Argo CD and Flux.

The infrastructure layer has mostly stayed stuck with traditional CI/CD approaches, and these approaches tend to be missing a lot of what makes GitOps so appealing, namely auditability, rollback capabilities and drift detection. While Terraform has completely changed the Infrastructure as Code game with declarative infrastructure definitions, deployment patterns have not caught up with the GitOps philosophy, which works so well for applications. Even organizations that've gone all-in on GitOps for their app deployments often end up managing Infrastructure through completely separate, way less integrated processes. The result is a hybrid approach that does not really capture the full benefits of unified, Git-centric operations. Extending GitOps principles to infrastructure deployments would provide a cohesive operational model covering both applications and Infrastructure. We will see how GitOps principles can be adapted and extended to Terraform deployments. We will also explore how controller-driven workflows can bring the same reliability and operational benefits to infrastructure management. We will analyze existing patterns, emerging tools, and practical implementation strategies to show how organizations can achieve truly unified GitOps operations across their entire tech stack.

2. GitOps Fundamentals

First, let us understand GitOps and why it has become popular. At its core, GitOps is basically a way of managing Infrastructure and applications where Git becomes the single source of truth for everything. The whole philosophy revolves around three key principles that are incredibly powerful when you put them together. First is declarative configuration. Instead of writing imperative scripts that outline step-by-step instructions, you are describing what you want your system to look like. It is the difference between giving someone directions, just showing them a picture of the destination, and letting them figure out how to get there. With declarative configs, you are essentially saying this is the end state and letting the system figure out how to get there.

The second principle is version control as the source of truth. Everything lives in Git. If it is not in Git, it does not exist—application code, infrastructure definitions, configuration files, and even policies that control deployments.

When something needs to change, it needs to be reflected directly into Git, not into a tool, not on a server. Then you make a pull request to integrate the changes. Git at that point becomes the single source of truth with its well-established auditability and traceability.

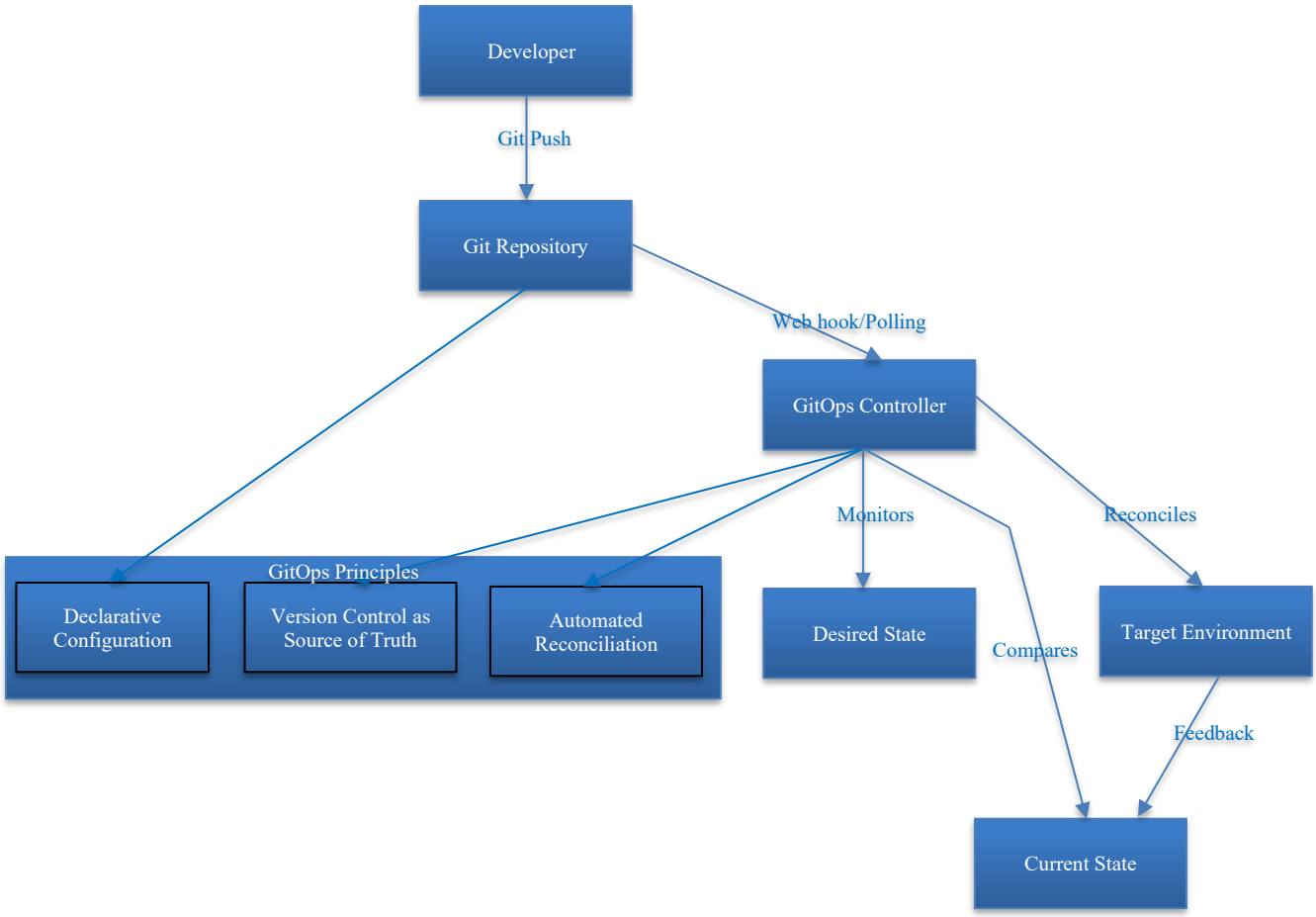

Fig. 1 GitOps Principles

Finally, there is automated reconciliation, where the magic happens. You have these controllers running continuously, comparing what is deployed with what's defined in Git. When it spots a difference in the declared state vs the actual state, it starts the reconciliation process to bring the system back to the desired state.

This is a big step-up from the traditional CI/CD approaches, where, in initial deployments, there is no way to check for drift, much less for reconciliation, unless the deployment pipeline is triggered again.

Now this is where things get really interesting. Kubernetes was a perfect platform for adoption by GitOps. The platform's declarative nature meant you can describe your entire application stack in YAML files. The controller pattern was pretty much built right into the core of Kubernetes. Tools like Argo CD and Flux did not have to go against the platform but complemented the ongoing architectural pattern [1][2].

Before GitOps, deploying to Kubernetes often meant running a bunch of kubectl commands wrapped in a CI/CD script. Credentials were managed manually; rollbacks were manual with no to very little auditability. With GitOps, the deployment process becomes as simple as merging a pull request. The controller applies changes, and if something breaks, a rollback can be performed by merely reverting a commit.

This approach has solved some fundamental problems that plagued traditional deployment methods. Drift detection becomes automatic; the controller will either fix it or generate an alert. Rollbacks become easier as every change is captured in Git history. The same applies to compliance; as every change goes through a Git workflow, a complete audit trail is available by default.

The ecosystem around Kubernetes GitOps has exploded. Argo CD has become the main tool for many organizations, offering a UI for visualizing deployments and managing applications across multiple clusters.

Flux takes a more toolkit approach by providing composable components that can be adapted to suit the needs. Both have their strengths and are solving the same fundamental problem of making Git the operational interface for your Infrastructure [1][2]

3. Traditional CI/CD vs GitOps for Infrastructure

Traditional CI/CD pipelines for application deployments have been proven to be effective over the last couple of decades. However, with Infrastructure, things have become more complex. Push-based deployments, in which an infrastructure code commit initiates a pipeline, are still widely used by enterprises.

The pipeline then invokes Terraform commands such as plan and apply to apply the changes. A human review process to ensure the generated plan meets the expectations may or may not be present. These pipelines work and are effective, but they still require a lot of effort to use and maintain, and thus, they are not very scalable.

3.1. The Pain Points of Conventional IaC Deployments

There are several drawbacks to using conventional methods of CI/CD for deploying Infrastructure. Let us understand them one by one.

3.1.1. Credential Management

The pipeline with over-privileged credentials is a security headache because the CI/CD system requires broad permissions to deploy Infrastructure. These credentials are frequently dispersed over multiple systems and kept in different secret managers. This opens up the door for credential misuse and is generally considered a security vulnerability [8][9].

3.1.2. Drift Detection

Once the pipeline deploys the Infrastructure, it is basically done. The pipeline has no idea if someone manually changes something in the cloud console. The Terraform state does not get updated with the changes being done outside of Terraform, and the actual state could be completely different. There is no way to know these differences between desired and actual states until the next time the pipeline runs, when the plan shows the objects modified outside of Terraform.

3.1.3. Rollback Capabilities

When an infrastructure deployment goes awry, rolling back is not as simple as reverting a commit. An investigation is needed to figure out what changed and potentially deal with stateful resources that cannot be easily rolled back. It is common for teams to spend hours manually fixing infrastructure issues instead of using automated rollbacks [4][8].

3.1.4. Auditability

Terraform plan provides documentation and details on the changes that will be applied. It is possible to maintain and store the plans for auditability. However, since it is a snapshot of the changes applied at a particular time, it does not give continuous visibility as the Infrastructure may have drifted over time.

3.2. GitOps Advantages: A Different Approach

GitOps completely changes how we think about deployments. Rather than having your CI/CD system push changes out to the Infrastructure, controllers sitting in the target environment watch Git and pull changes when they see them. At first glance, this pull-versus-push thing might seem like a minor detail, but it changes how everything works.

3.2.1. Security

AWS credentials are no longer being fed into Jenkins or GitHub Actions. Those over-privileged service accounts are no longer needed. The GitOps controller lives in the AWS account and uses workload identity to get just the permissions it needs, when needed [7][9].

3.2.2. Drift Detection

Manual changes show up as drift immediately. With GitOps, the controller constantly checks what is actually deployed against what is in Git. It spots that manual change immediately and can either fix it automatically or produce an alert.

3.2.3. Rollbacks

When something breaks, you revert the Git commit. That's it. The controller sees the revert and brings your Infrastructure back to its previous state. No more digging through Terraform state files trying to figure out what changed or manually fixing resources one by one [4][7]. Below is an example to explain the concept. Let us say we are managing Infrastructure for an application using Terraform. Let us also assume that this application is deployed in AWS. A traditional pipeline to deploy the Infrastructure to AWS would look like this.

3.3. Traditional CI/CD Flow

1. Terraform code is pushed by the developer into Git
2. Pipeline is triggered and uses stored credentials to authenticate against AWS
3. Terraform commands are executed by the pipeline
4. A successful pipeline run indicates infrastructure deployment
5. A manual change from this point onwards will not be detected until the next deployment

3.4. GitOps Flow

1. Terraform code is pushed by the developer into Git
2. GitOps controller detects the change
3. Terraform commands are executed by the controller
4. Controller continuously monitors the git state vs the actual state
5. A manual change is detected as drift as soon as the change is made

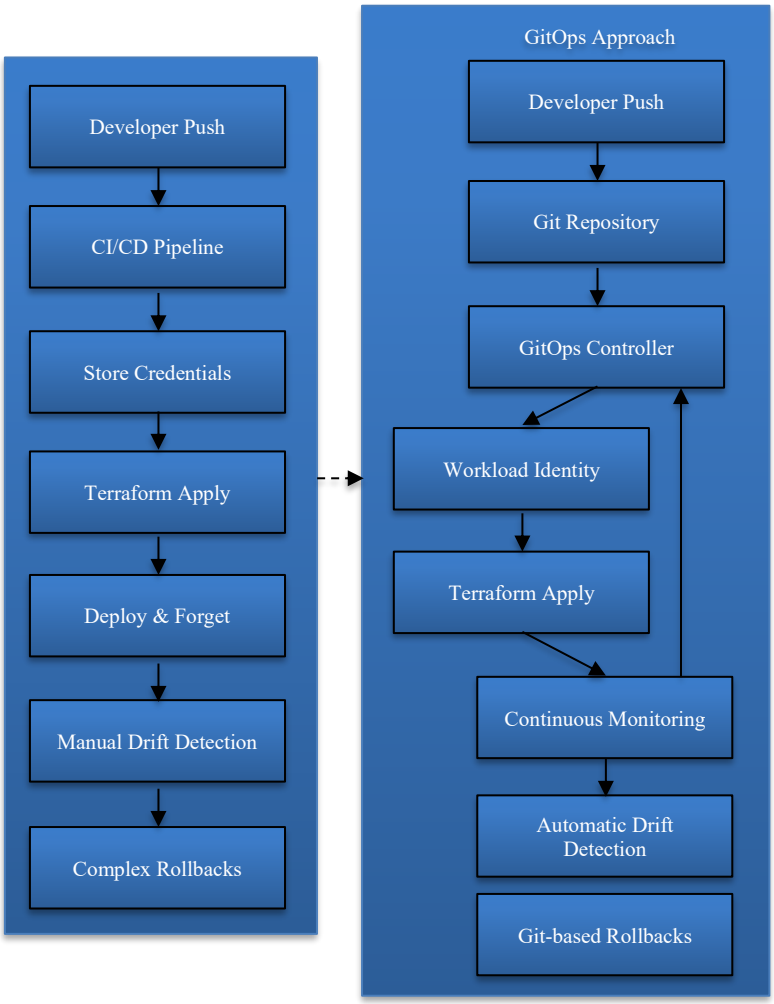

Fig. 2 Deploying Terraform: Traditional vs GitOps approach

Table 1. Differences between Traditional CI/CD vs GitOps

Aspect	Traditional CI/CD	GitOps
Deployment Model	Push-based (CI/CD pushes to production)	Pull-based (agents pull from Git)
Source of Truth	CI/CD pipeline configuration	Git repository
Change Management	Manual approvals, pipeline triggers	Git workflows (PRs, reviews)
Rollback Strategy	Pipeline rollback commands	Git revert operations
Security	Requires production access credentials	No outbound access needed from the cluster
Observability	Pipeline logs and monitoring	Git history + cluster state comparison
Drift Detection	Manual or scheduled checks	Continuous reconciliation
Multi-Environment	Separate pipelines per environment	Branch-based or repo-per-environment
Audit Trail	CI/CD logs and approvals	Complete Git history
Learning Curve	Familiar to most teams	Requires a Git-centric mindset shift
Tool Examples	Jenkins, GitLab CI, Azure DevOps	ArgoCD, Flux, Tekton
State Management	External state stores (Terraform Cloud)	Declarative manifests in Git
Failure Recovery	Manual intervention is often required	Automatic convergence to the desired state

3.5. The Controller Pattern: Bridging the Gap

The controller pattern is the brain behind the entire concept of GitOps. To leverage the same benefits as GitOps for infrastructure deployments, we need to introduce the controller patterns. Controllers are the programs that run continuously, watching the declared state and comparing it to the actual state. When it spots a difference, it can take steps to reconcile the differences. Kubernetes has controllers for everything, including deployments, services, and load balancers.

For Terraform, we need controllers that can do the same thing. It needs to watch for changes to Terraform code, keep an eye on your actual cloud resources, and run `terraform apply` when resources are out of sync [5][6].

This pattern allows us to leverage GitOps principles without throwing away Infrastructure as code using Terraform. No rewrite of the code is needed; it is just an alignment to the deployment strategy with the additional benefits of drift detection, easy rollbacks, and better security.

4. GitOps-Based Terraform Framework

Now that we have understood how GitOps works and how it can overcome some of the problems with infrastructure deployment using traditional CI/CD, let us understand how we can build such a system. How can we make GitOps principles work with Terraform so that it does not break existing workflows?

The answer is building a controller-based system that brings the same continuous reconciliation pattern from Kubernetes to infrastructure management.

4.1. Controller/Operator Architecture for Terraform Deployments

ontrollers are the engine of any GitOps system, but Terraform controllers would be more complex than the average Kubernetes controller because of the many moving parts involved. There are state files to manage, dependencies between resources, and changes that can take longer than normal to complete. Plus, Terraform deployments are generally not immutable and hence cannot just be started over if an existing deployment fails.

A Terraform controller should understand how Terraform works, including the planning phase, state locking, resource dependencies, etc. It needs to be smart enough to handle failures gracefully and work with other controllers that handle Infrastructure.

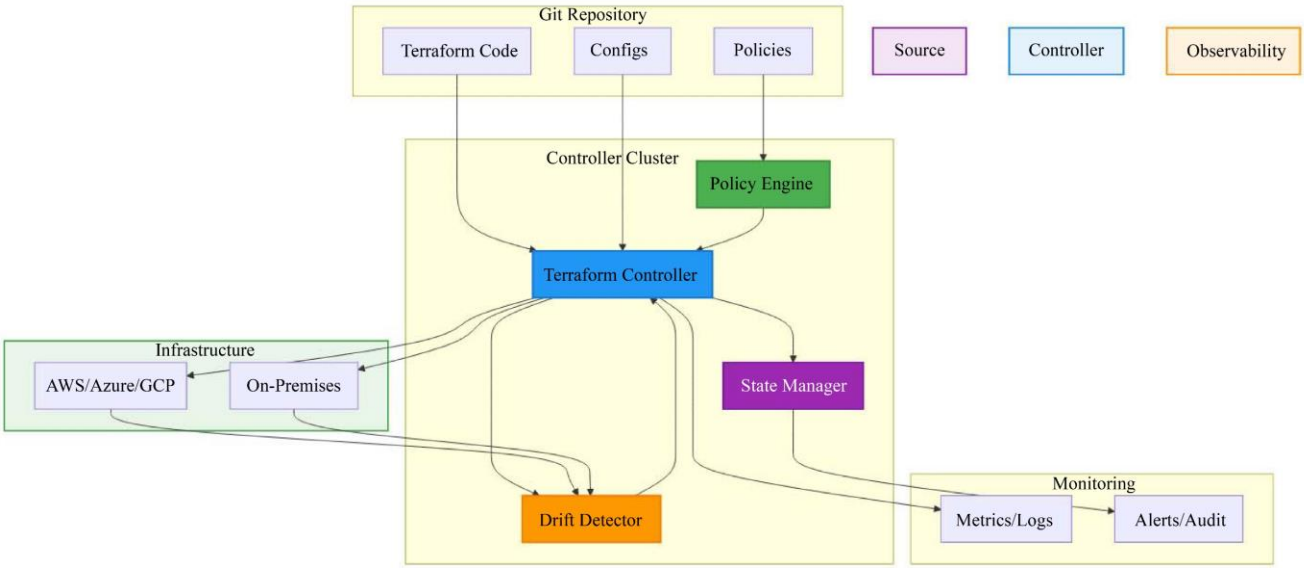

Fig. 3 Design of a practical GitOps controller for Terraform

The Terraform Controller orchestrates the entire flow. Unlike a traditional CI/CD pipeline that runs `terraform apply` and stops, a controller keeps running in the background, constantly checking that the Infrastructure matches the state defined in Git.

The State Manager handles state management. It needs centralized storage and a proper locking mechanism so different

The Policy Engine defines the guardrails. Before any Terraform operation runs, it checks that the changes meet security policies and other defined policies, including budgets.

The Drift Detector mimics the Terraform controller and is constantly comparing the actual Infrastructure to what is defined in Git.

4.2. Git-Centric Workflow Implementation

Now we know what a Terraform controller would look like. Let us see what a real-world developer workflow would look like.

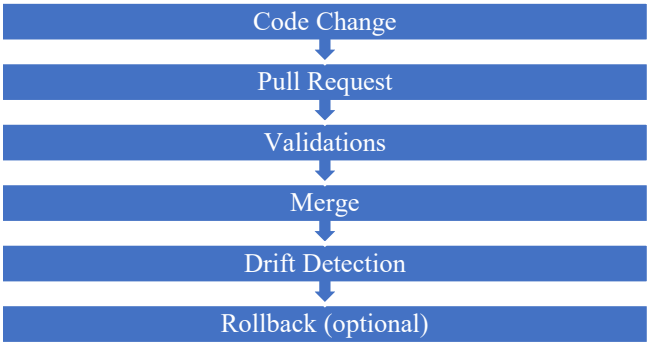

Fig. 4 Implementation of a Git-Centric workflow

4.2.1. Step 1: Code Changes

Let us assume that a developer needs to add a new S3 bucket for a microservice. Rather than logging in to the AWS console, he or she creates new resources in the Terraform configuration inside the appropriate Git repository:

```
# environments/production/storage.tf
resource "aws_s3_bucket" "microservice_data" {
  bucket = "myorg-prod-microservice-data-
${random_id.bucket_suffix.hex}"

  tags = {
    Environment = "production"
    Service     = "microservice-api"
    ManagedBy   = "gitops-terraform"
  }
}

resource "aws_s3_bucket_versioning" "microservice_data"
{
  bucket = aws_s3_bucket.microservice_data.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

4.2.2. Step 2: Pull Request and Review Process

The developer now opens up a pull request. The GitOps system automatically runs `terraform plan` and adds the results in the PR as a comment. This information can be used by the reviewer to help with the review process.

4.2.3. Step 3: Automated Validation and Policy Checks

The policy engine runs at this point and ensures that the modified code meets the standards. For example, it may check that the S3 bucket has versioning and encryption turned on and follows the naming conventions. This feedback is also included in PR as a comment alongside the plan.

4.2.4. Step 4: Merge and Automatic Deployment

After the PR has been approved and the code is merged, the controller detects a change in the git state vs the target state and begins creating the S3 bucket to bring the two states in unison.

4.2.5. Step 5: Continuous Monitoring and Drift Detection

Post deployment, the drift detector keeps a watch on the Infrastructure along with the newly created S3 bucket. If someone changes the configuration manually, the system catches it immediately. It can then fix it automatically or send an alert.

4.2.6. Step 6: Rollback if Needed

In case there are unseen issues with deployment, reverting the changes is possible via reverting the Git commits. The controller sees the revert and automatically rolls back the Infrastructure to its previous state. This workflow provides audit trails, rollbacks, and automatic drift detection, along with the existing benefits of Terraform.

4.3. Terraform State Management in GitOps Context

In order to use GitOps for Terraform deployment, we need to ensure the Terraform state is managed properly. Any misconfigurations can have severe consequences. With GitOps controllers, multiple parallel reads and writes to the state files across different environments exist. This complicates the general setup of state management in Terraform, where an S3 bucket can easily provide state storage and locking capabilities.

The best strategy is to create separate state buckets for different environments with IAM policies that follow least privilege principles. This will limit what each controller can see and do in terms of state manipulation.

Using a separate bucket also solves the problem of state locking to a certain degree. Turning on versioning on S3 buckets with appropriate lifecycle policies provides adequate backup without too much spending.

```
  State backend configuration for production environment
terraform {
  backend "s3" {
    bucket         = "myorg-terraform-state-prod"
    key            = "services/microservice-api/terraform.tfstate"
    region         = "us-west-2"
    encrypt        = true
    use_lockfile   = true

    # GitOps controller uses workload identity
    role_arn =
"arn:aws:iam::123456789012:role/GitOpsControllerRole"
  }
}
```

4.4. Real World Deployment Example

We will extend our example above to an architecture that is very common in the real world. Assume we are managing Infrastructure for a three-tier web app across dev, staging, and production environments.

4.4.1. Repository Structure

```
infrastructure/
├── modules/
│   ├── vpc/
│   ├── database/
│   └── application/
├── environments/
│   ├── development/
│   ├── staging/
│   └── production/
└── policies/
    ├── security.rego
    └── cost.rego
```

4.4.2. Environment Configuration (production/main.tf):

```
module "vpc" {
  source = "../modules/vpc"

  environment = "production"
  cidr_block  = "10.0.0.0/16"

  tags = local.common_tags
}

module "database" {
  source = "../modules/database"

  environment    = "production"
  vpc_id         = module.vpc.vpc_id
  subnet_ids     = module.vpc.private_subnet_ids
  instance_class = "db.r5.xlarge"

  tags = local.common_tags
}

module "application" {
  source = "../modules/application"

  environment = "production"
  vpc_id      = module.vpc.vpc_id
  subnet_ids  = module.vpc.public_subnet_ids

  database_endpoint = module.database.endpoint

  tags = local.common_tags
}

locals {
  common_tags = {
    Environment = "production"
    ManagedBy   = "gitops-terraform"
    Project     = "web-application"
  }
}
```

4.4.3. GitOps Controller Configuration:

```
apiVersion: gitops.io/v1
kind: TerraformWorkspace
Metadata:
  name: web-app-production
  namespace: gitops-system
Spec:
  Source:
    Git:
      url: https://github.com/myorg/infrastructure
      path: environments/production
      branch: main

  Terraform:
    version: "1.6.0"
    workspace: production

  Backend:
    s3:
      bucket: myorg-terraform-state-prod
      key: web-application/terraform.tfstate
      region: us-west-2

  Policy:
    enforce: true
    Sources:
      - path: policies/security.rego
      - path: policies/cost.rego

  reconciliation:
    interval: 10m
    retryInterval: 2m
    timeout: 30m
```

Below is what the developer workflow would look like to make an infrastructure change.

- Change Detection: Controller detects a change in the git state vs the target state
- Policy Validation: Policy engine performs compliance validation
- Plan Generation: Terraform plan is generated
- Execution: Once changes are approved, Terraform apply is performed
- State Update: Terraform state is updated.
- Monitoring: Drift detector starts monitoring resources

4.5. Security and Compliance Considerations

There are some security considerations when using GitOps for Terraform deployments. These security

considerations may not be a concern in the case of traditional deployments using CI/CD pipelines.

4.5.1. Workload Identity and Credential Management

One of the biggest security improvements is the removal of long-lived credentials from the CI/CD pipelines. The GitOps controller uses workload identity, such as AWS IAM Roles or Azure Managed Identity, to get temporary and limited-scoped credentials.

4.5.2. Policy as Code Integration

With Git being the single source of truth, integration with policy as code tools such as Open Policy Agent (OPA) becomes rather straightforward. OPA can ensure organization level standards are being met by performing automatic validations.

4.5.3. Audit Trails and Compliance

Git history becomes the audit trail. It is easy to trace back changes by just looking at the git history and generated Terraform plans and compliance reports from pull requests.

4.5.4. Secrets Management

GitOps controller should be integrated with a secrets manager to ensure that the credentials for both source and target, i.e., the git repository and target cloud environment, are maintained in a secure vault.

4.5.5. Network Security and Isolation

Proper network policies and firewall rules should be defined to ensure that the controller can only access the resources it absolutely needs.

This GitOps approach ties two important approaches of the new DevOps world: the declarative infrastructure management using Terraform and the operational and security benefits of GitOps.

5. Conclusion

Extending GitOps principles to infrastructure deployments has a lot of benefits, but it does come with challenges that must be addressed in order to have a successful implementation of this pattern. Below are some of the challenges that should be kept in mind while implementing such a system.

5.1. Emerging Opportunities and Current Limitations

5.1.1. Multi-Cloud Infrastructure Orchestration

Most organizations these days are operating across multiple cloud providers, and over 92% of enterprises have adopted multi-cloud strategies according to Flexera's 2024 State of the Cloud Report. GitOps controllers can help operational efficiencies across AWS, Azure, and GCP environments through declarative configurations in centralized Git repositories [7][8]. Projects like Crossplane can extend this capability by enabling infrastructure management for different cloud providers using a single configuration.

5.1.2. Policy Automation and Compliance

GitOps workflows make it easy for engineers to integrate policy management. These engines can be used to address governance requirements. Open Policy Agent (OPA) adoption has grown significantly, and the Cloud Native Computing Foundation reports over 100 production deployments across major organizations [5][6]. OPA can enforce compliance requirements for SOC 2, GDPR, and other regulatory frameworks.

5.1.3. Technical Challenges and Solutions

There are still significant challenges that must be overcome. Terraform state management complexity increases a lot with GitOps deployments across multiple controllers and environments. HashiCorp's Terraform Cloud addresses some of these concerns via enhanced remote state management and locking mechanisms [3][8].

There are some performance constraints that must be considered as well. GitOps controllers start to struggle with reconciliation loops as the size and number of attached git repositories increase. Performance becomes an issue when scaling from large to very large systems [4][9].

5.2. Key Value Propositions

There are three key value propositions for adapting the GitOps principles for infrastructure deployments:

5.2.1. Operational Consistency

GitOps for Infrastructure allows us to use the same deployment pattern as applications. A unified model reduces complexity and improves collaboration across development and operations teams [4][9].

5.2.2. Enhanced Security Posture

Removal of long-lived credentials from CI/CD pipelines. GitOps controllers use workload identity mechanisms to reduce credential exposure and improve overall security posture [6][7].

5.2.3. Improved Reliability

Continuous reconciliation and Git-based rollback capabilities provide operational excellence that traditional approaches lack. Organizations report increased deployment frequency and reduced mean time to recovery when implementing GitOps practices [8][9].

5.3. Implementation Strategy

There should be a phased rollout approach when using GitOps for infrastructure deployments:

Start with non-production environments to understand what controller patterns work. State management will also require experimentation to discover what works best. Policy frameworks and security configurations must also be leveraged early on as these become foundational elements[6][7].

From the get-go, have an observability and monitoring strategy in place. This will help uncover controller behavior and reconciliation patterns. Plan ahead for state management complexity and backup strategies [3][8].

A significant change in operating procedures can be achieved by transitioning to declarative, Git-driven infrastructure management. Businesses that are able to adopt these practices will have operational advantages in cloud-native infrastructure management.

References

[1] Argo Project, Argo CD - Declarative GitOps CD for Kubernetes. [Online]. Available: https://argo-cd.readthedocs.io/

[2] Flux Community, Flux: GitOps Family of Projects. [Online]. Available: https://fluxcd.io/

[3] HashiCorp, Enhanced GitOps Workflows with Terraform Cloud, 2022. [Online]. Available: https://www.hashicorp.com/en/resources/enhanced-gitops-workflows-with-terraform-cloud

[4] INNOQ, Implementing GitOps Without Kubernetes, 2025. [Online]. Available: https://www.innoq.com/en/articles/2025/01/gitops-kubernetes/

[5] Jason Dobies, and Joshua Wood, *Kubernetes Operators: Automating the Container Orchestration Platform*, O'Reilly Media, 2020. [Google Scholar] [Publisher Link]

[6] Kief Morris, *Infrastructure as Code: Managing Servers in the Cloud*, O'Reilly Media, 2021. [Google Scholar] [Publisher Link]

[7] NextLink Labs. How GitOps Goes Beyond Kubernetes. [Online]. Available: https://nextlinklabs.com/resources/insights/how-gitops-goes-beyond-kubernetes

[8] Terrateam, GitOps Beyond Kubernetes: Applying GitOps Principles to Infrastructure as Code, 2025. [Online]. Available: https://terrateam.io/blog/gitops-beyond-kubernetes

[9] Medium, GitOps in Modern Times: Beyond CI/CD Pipelines, 2025. [Online]. Available: https://medium.com/the-programmer/gitops-in-modern-times-beyond-ci-cd-pipelines-8af53c88fe84

[10] Akuity, Christian Hernandez, GitOps Best Practices: A Complete Guide to Modern Deployments, 2025. [Online]. Available: https://akuity.io/blog/gitops-best-practices-whitepaper