Mastering Advanced Terraform & DevOps

Taking your infrastructure automation to the next level with powerful tools, security practices, and GitOps workflows that make managing cloud resources simpler and more reliable.

Advanced Terraform CLI

Beyond the basics of plan and apply

Terraform Console: Your Interactive Playground

The `terraform console` command opens an interactive shell where you can test expressions, explore state data, and debug variable values without modifying your infrastructure. Think of it as a sandbox for experimenting with Terraform's language features.

Real-world scenario: You're writing a complex variable transformation and want to test it before committing. Instead of running expensive plan operations repeatedly, you open the console and iterate instantly.

```
$ terraform console
> var.environment
"production"
> length(var.server_names)
5
> [for s in var.servers : upper(s)]
["WEB-01", "WEB-02", "DB-01"]
```

Test expressions interactively without affecting your actual infrastructure state.

Common Console Use Cases

Testing Functions

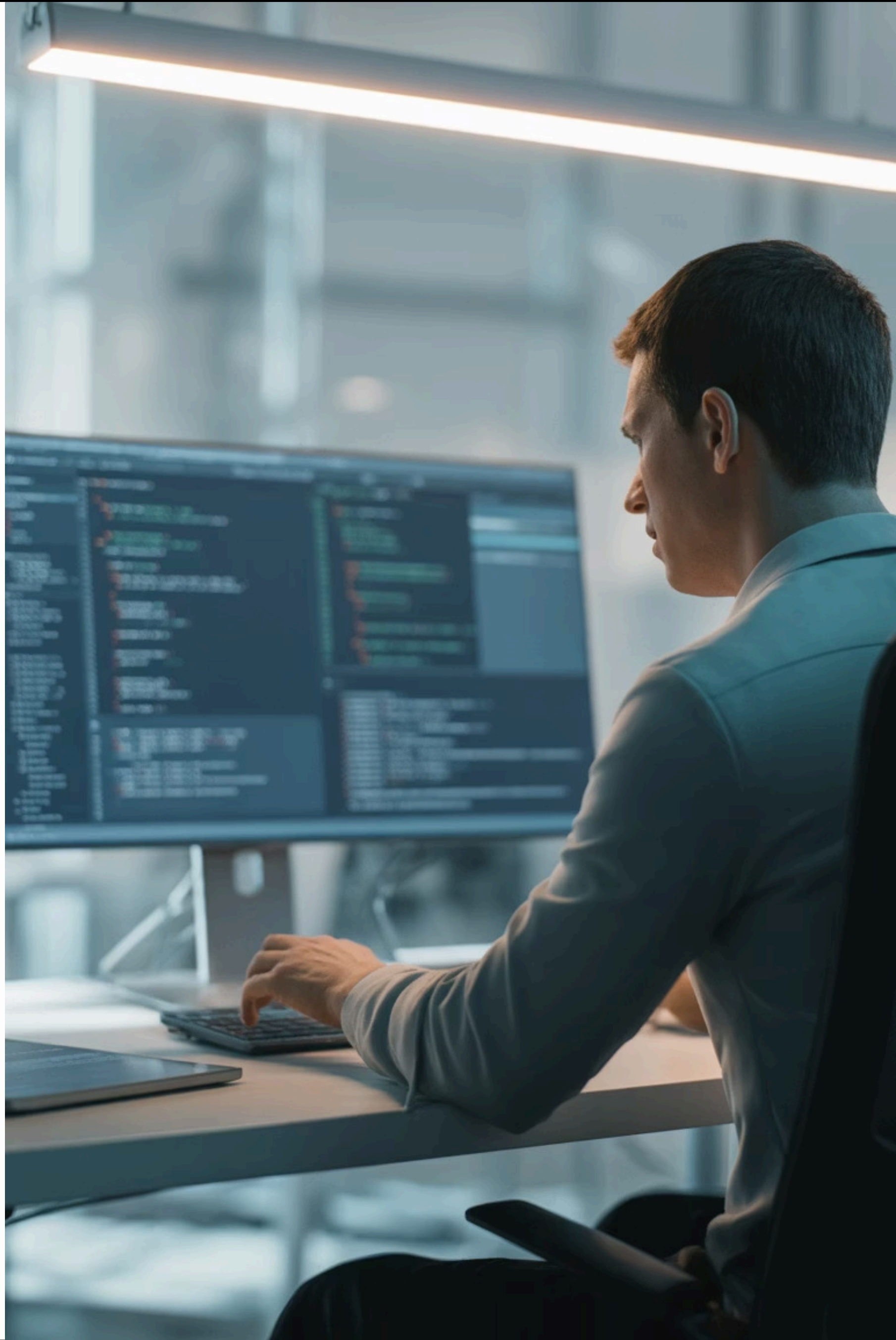
Verify string manipulation, date formatting, or mathematical operations work as expected

Querying State

Inspect current resource attributes without grepping through JSON files

Debugging Loops

Test for_each and count expressions to ensure correct iteration behavior



Terraform Graph: Visualize Dependencies

The `terraform graph` command generates a visual representation of your resource dependency tree. This is incredibly valuable when troubleshooting complex module relationships or understanding why resources are created in a specific order.

How it works: Terraform outputs a DOT format file that you can render using Graphviz or online tools. The resulting diagram shows every resource as a node and dependencies as arrows.

Practical example: Your database isn't ready when the application starts. Running terraform graph reveals that the security group dependency isn't explicit, causing race conditions during deployment.

```
$ terraform graph | dot -Tpng > graph.png

# View dependency relationships
# Identify circular dependencies
# Understand module connections
```

Understanding the Dependency Graph

1. **VPC Creation**

   Foundation network infrastructure is established first

2. **Subnets & Routes**

   Network segments depend on VPC existing

3. **Security Groups**

   Firewall rules reference the VPC and subnets

4. **EC2 Instances**

   Compute resources need all networking in place

Debug Logging: See What Terraform Is Thinking

When things go wrong, debug logs are your best friend. Terraform supports multiple log levels that reveal internal operations, API calls, and decision-making processes that aren't visible in standard output.

TF_LOG Environment Variable

Set to TRACE, DEBUG, INFO, WARN, or ERROR to control verbosity. TRACE shows everything including HTTP requests to cloud providers.

TF_LOG_PATH for File Output

Redirect logs to a file instead of cluttering your terminal. Essential for debugging CI/CD pipeline failures where output is captured.

```
export TF_LOG=DEBUG
export TF_LOG_PATH=./terraform-debug.log
terraform apply
```

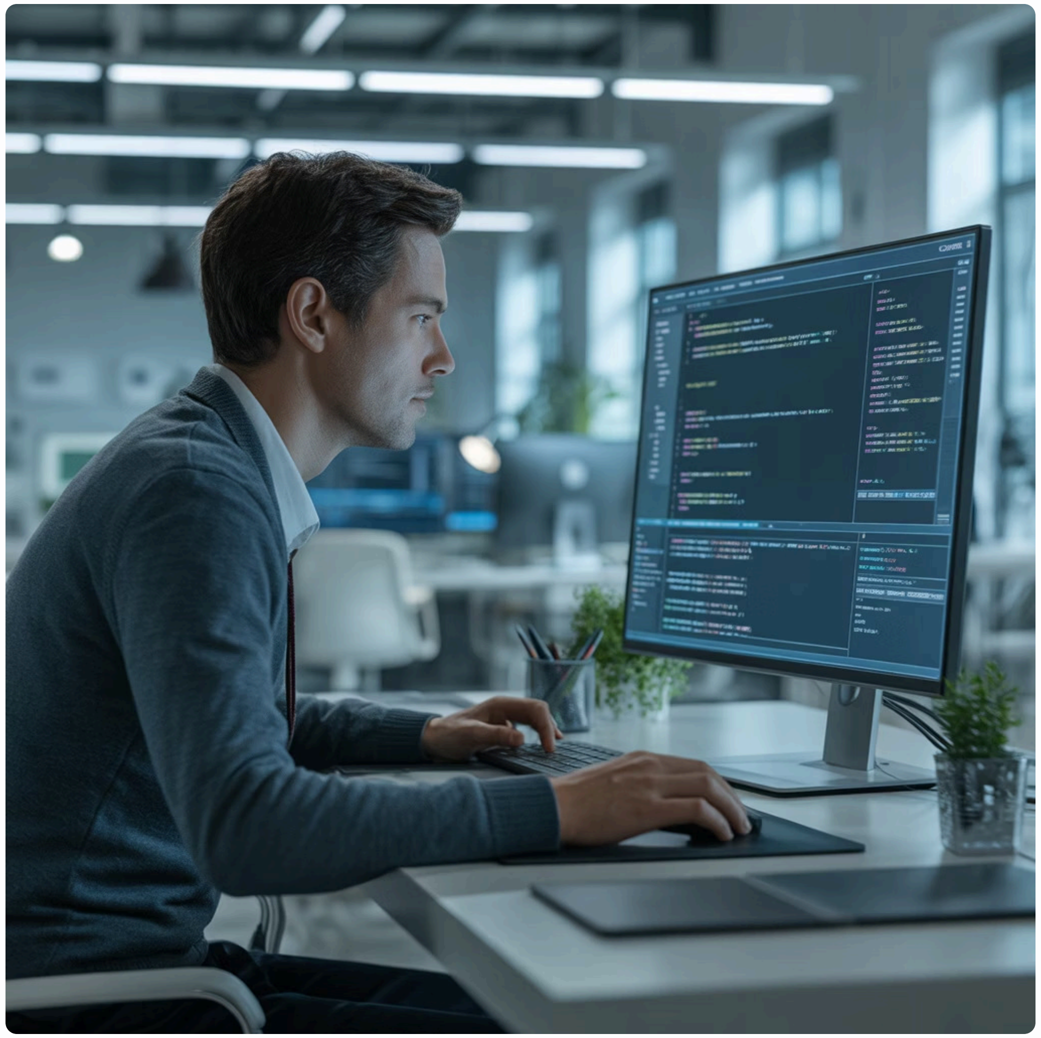
Real Debugging Scenario

The Problem

Your Terraform apply keeps timing out when creating an RDS instance, but there's no clear error message. Standard output shows "Still creating..." for 15 minutes before failing.

The Solution

Enable TRACE logging and discover that AWS API is rejecting the subnet group configuration due to subnets being in the same availability zone—a constraint not mentioned in the main error output.



Pro tip: Always enable debug logging in CI/CD pipelines by default. The performance overhead is minimal, and having logs when something fails in production is invaluable.

Kubernetes Provider

Managing Kubernetes resources with Terraform

Why Use Terraform for Kubernetes?

While kubectl and YAML manifests are the traditional approach, Terraform brings consistency to multi-cloud environments. You can manage AWS resources, Kubernetes clusters, and the applications running inside them—all in one workflow with unified state management.

Unified State

Track both infrastructure and application deployments in a single state file

Dependency Management

Ensure clusters exist before deploying applications, with automatic ordering

Reusable Modules

Package common Kubernetes patterns as modules for consistency across teams

Configuring the Kubernetes Provider

The Kubernetes provider needs credentials to authenticate with your cluster. The most common approach uses the kubeconfig file, but you can also provide credentials directly or use cloud provider authentication mechanisms.

Best practice: In production, use dynamic credentials from your cloud provider (EKS, AKS, GKE) rather than static kubeconfig files. This improves security and simplifies credential rotation.

```
terraform {
 required_providers {
 kubernetes = {
 source = "hashicorp/kubernetes"
 version = "~> 2.23"
 }
 }
}

provider "kubernetes" {
 host = aws_eks_cluster.main.endpoint
 cluster_ca_certificate = base64decode(
 aws_eks_cluster.main.certificate_authority[0].data
 )
 token = data.aws_eks_cluster_auth.main.token
}
```

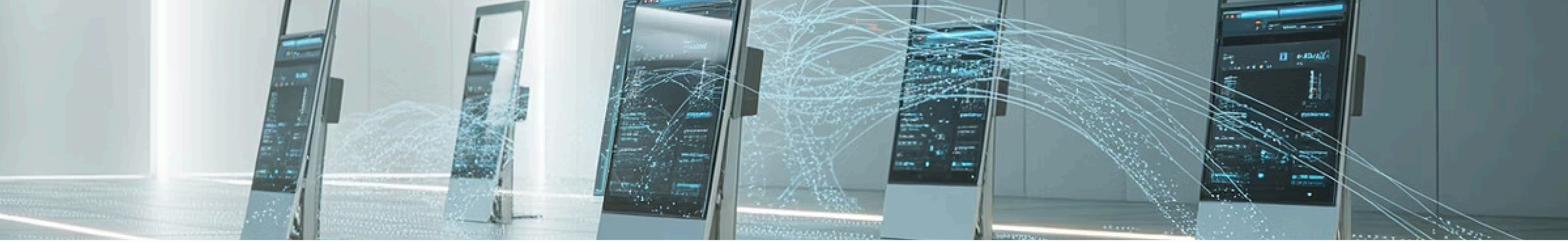
Deploying a Simple Application

Let's deploy a web application with Terraform. This example creates a deployment, service, and ingress—demonstrating how Terraform handles Kubernetes resources just like cloud infrastructure.

```
resource "kubernetes_deployment" "webapp" {
 metadata {
 name = "webapp"
 namespace = "production"
 }

 spec {
 replicas = 3

 selector {
 match_labels = {
 app = "webapp"
 }
 }

 template {
 metadata {
 labels = {
 app = "webapp"
 }
 }

 spec {
 container {
 name = "webapp"
 image = "nginx:1.21"

 port {
 container_port = 80
 }
 }
 }
 }
 }
}
```

Terraform + Ansible

The perfect combination for infrastructure



Why Combine Terraform and Ansible?

Terraform excels at provisioning infrastructure—creating servers, networks, and cloud resources. Ansible excels at configuring those resources—installing software, managing files, and maintaining desired state. Together, they create a complete automation pipeline.

1. **Terraform Provisions**

   Creates 10 EC2 instances with proper networking and security groups

2. **Ansible Configures**

   Installs applications, sets up monitoring, configures backups on those instances

3. **Continuous Management**

   Ansible maintains configuration drift while Terraform handles infrastructure changes

Integration Pattern: Inventory Generation

The simplest integration approach is having Terraform generate an Ansible inventory file. After provisioning infrastructure, Terraform outputs IP addresses and connection details that Ansible uses to configure the systems.

Terraform Output

```
output "web_servers" {
  value = {
    for instance in aws_instance.web :
    instance.tags["Name"] => instance.public_ip
  }
}

resource "local_file" "ansible_inventory" {
  content = templatefile("inventory.tpl", {
    web_ips = aws_instance.web[*].public_ip
    db_ips  = aws_instance.db[*].private_ip
  })
  filename = "${path.module}/inventory.ini"
}
```

Generated Inventory

```
[webservers]
web-01 ansible_host=54.123.45.67
web-02 ansible_host=54.123.45.68

[databases]
db-01 ansible_host=10.0.1.10
db-02 ansible_host=10.0.1.11

[all:vars]
ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/terraform
```

Using Provisioners for Immediate Configuration

Terraform's local-exec provisioner can trigger Ansible playbooks immediately after resource creation. This is useful when you need configuration to happen as part of the infrastructure deployment process.

```
resource "aws_instance" "app_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.medium"

  tags = {
    Name = "application-server"
  }

  provisioner "local-exec" {
    command = "ansible-playbook -i '${self.public_ip},' -u ubuntu configure-app.yml"

    environment = {
      ANSIBLE_HOST_KEY_CHECKING = "False"
    }
  }
}
```

Important: Provisioners should be a last resort. They make Terraform less declarative and harder to maintain. Consider using separate Ansible runs or configuration management tools for complex setups.

Real-World Integration Workflow

1. Terraform Apply

   Provisions infrastructure and outputs inventory data to a JSON file

2. Dynamic Inventory

   Ansible uses Terraform's output as a dynamic inventory source

3. Base Configuration

   Ansible runs common playbooks for security hardening and monitoring setup

4. Application Deployment

   Service-specific playbooks deploy and configure applications

5. Validation

   Automated tests verify the complete stack is functioning correctly

GitOps for Terraform

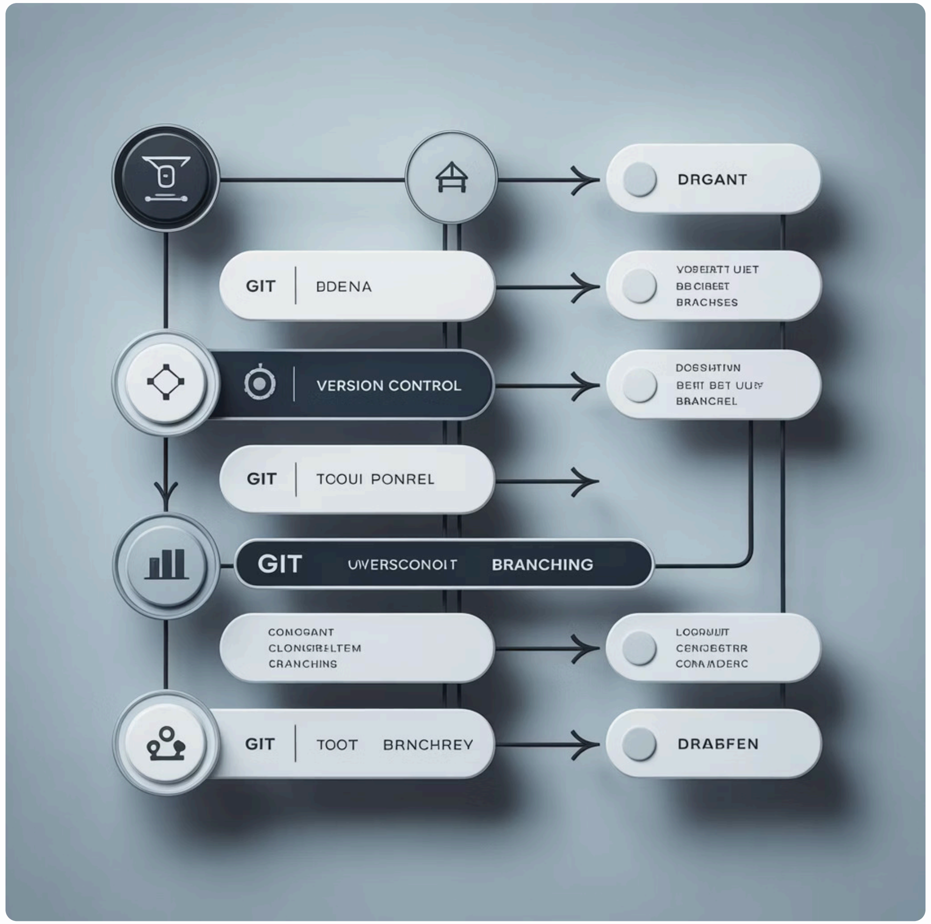
Infrastructure changes through Git workflows

What is GitOps?

GitOps treats Git as the single source of truth for infrastructure. Every change goes through pull requests, code review, and automated testing before being applied. This brings software development best practices to infrastructure management.

Core principles:

- Declarative infrastructure definitions in Git
- Automated deployment on Git commits
- Continuous reconciliation between Git state and actual infrastructure
- Complete audit trail through Git history



GitOps Workflow for Terraform

Developer Creates Branch

Engineer opens a pull request with Terraform changes to add new resources

Automated Validation

CI pipeline runs terraform fmt, validate, and plan automatically

Peer Review

Team reviews the plan output and code quality before approving

Merge to Main

Approved changes are merged, triggering the deployment pipeline

Automatic Apply

CD system runs terraform apply with no manual intervention required

Benefits of the GitOps Approach

Complete Audit Trail

Every infrastructure change is recorded in Git with author, timestamp, and reason. Need to know who added that security group rule six months ago? Check the Git history.

Easy Rollbacks

Infrastructure problems? Revert the Git commit and re-deploy. No scrambling to remember what changed or manually fixing configurations.

Consistent Environments

Development, staging, and production use the same code with different variable files. No more "it works on my machine" for infrastructure.

Reduced Human Error

Automation prevents typos and forgotten steps. The pipeline runs the same commands every time, eliminating manual mistakes during deployments.

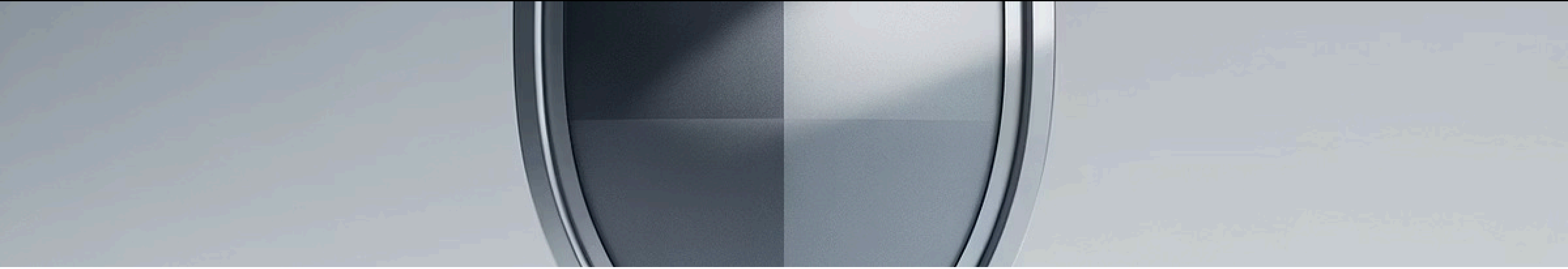
Implementing GitOps: Practical Example

Here's a GitHub Actions workflow that implements GitOps for Terraform. It runs plan on pull requests and apply on merges to main, giving teams confidence before deploying changes.

```
name: Terraform GitOps
on:
 pull_request:
 paths: ['terraform/**']
 push:
 branches: [main]
 paths: ['terraform/**']

jobs:
 terraform:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Setup Terraform
 uses: hashicorp/setup-terraform@v2

 - name: Terraform Init
 run: terraform init

 - name: Terraform Plan
 if: github.event_name == 'pull_request'
 run: terraform plan -no-color

 - name: Terraform Apply
 if: github.ref == 'refs/heads/main'
 run: terraform apply -auto-approve
```

Security with Checkov

Catch vulnerabilities before deployment



What is Checkov?

Checkov is a static code analysis tool that scans infrastructure-as-code for security and compliance issues. It checks your Terraform files against hundreds of best practices and compliance standards before you deploy, preventing vulnerabilities from reaching production.

Policy-as-Code

Enforces security policies across all Terraform configurations automatically

Compliance Standards

Validates against CIS, PCI-DSS, HIPAA, and other regulatory frameworks

Fast Feedback

Runs in seconds during development or CI/CD, catching issues immediately

Common Security Issues Checkov Catches

Before Checkov

```
resource "aws_s3_bucket" "data" {
  bucket = "company-sensitive-data"

  # No encryption!
  # Public access possible!
  # No versioning!
}
```

❌ **Critical vulnerabilities:**

- Unencrypted data at rest
- Missing access logging
- No versioning for recovery
- Public access not explicitly blocked

After Checkov Fix

```
resource "aws_s3_bucket" "data" {
 bucket = "company-sensitive-data"
}

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

resource "aws_s3_bucket_server_side_encryption_configuration" "data" {
 bucket = aws_s3_bucket.data.id
 rule {
 apply_server_side_encryption_by_default {
 sse_algorithm = "AES256"
 }
 }
}
```

Integrating Checkov in CI/CD

Running Checkov as part of your CI/CD pipeline ensures every infrastructure change is validated before deployment. Failed checks block the pipeline, preventing security issues from reaching production environments.

```
- name: Run Checkov Security Scan
  uses: bridgecrewio/checkov-action@master
  with:
    directory: terraform/
    framework: terraform
    output_format: cli
    soft_fail: false
    download_external_modules: true

- name: Upload Results
  uses: github/codeql-action/upload-sarif@v2
  with:
    sarif_file: results.sarif
```

950+

Built-in Policies

Covering AWS, Azure, GCP, Kubernetes, and more

5 sec

Average Scan Time

Near-instant feedback on security posture

100%

Open Source

Free to use with optional commercial features

Bringing It All Together

You now have the tools to implement professional-grade infrastructure automation. Advanced Terraform CLI features help you debug and understand your infrastructure. Kubernetes and Helm integration brings consistency to application deployments. Terraform and Ansible together handle both provisioning and configuration. GitOps workflows provide safety and auditability. And Checkov integration ensures security is never an afterthought.

Start Small

Implement one technique at a time in your existing workflows

Iterate Continuously

Refine your processes based on team feedback and pain points

Share Knowledge

Document patterns and teach others to scale best practices

The journey to infrastructure excellence is continuous. Each improvement makes your systems more reliable, secure, and maintainable. Start implementing these practices today and watch your infrastructure automation mature.