Infrastructure as Code and Cloud Services

Module Summary

This course provides an introduction to Infrastructure as Code (IaC) and its application with public cloud services. You will gain a foundational understanding of cloud computing, the principles of IaC, GitOps concepts and the benefits of automation in infrastructure provisioning.
During this module, you will create a group of 4 students and deploy an infrastructure on a public cloud using Terraform and GitHub Actions, following a set of best practices / methodologies.

Course 1: Introduction & IaC Fundamentals - Summary

What is the Cloud? Why Automate Infrastructure?

This section will explore the fundamental concepts of cloud computing, including its various service models (IaaS, PaaS, SaaS) and deployment models (public, private, hybrid). We will discuss the motivations behind adopting cloud services and the challenges associated with infrastructure management. The benefits of automating infrastructure, such as increased efficiency, reliability, centralized definitions and easier deployments, will be highlighted.

Terraform Principles & best practices: Reproducibility, Idempotence, Versioning

This section will delve into the core principles of Infrastructure as Code, using Terraform. We will define and discuss:

- **Reproducibility:** Ensuring that infrastructure can be consistently recreated in any environment.
- **Idempotence:** Guaranteeing that applying an IaC configuration multiple times will result in the same desired state without unintended side effects, enforcing drift detection and correction.
- **Versioning:** Managing infrastructure configurations using version control systems, allowing for tracking changes, rollbacks, and collaboration.

GitOps Concepts & Benefits

This section will introduce GitOps as an operational framework that leverages Git as a single source of truth for declarative infrastructure and applications. We will explore:

- The core principles of GitOps, including declarative configuration, version control, and automated delivery.
- The benefits of adopting GitOps, such as improved security, compliance, and developer experience.

Demo: Compare Manual vs. Automated Provisioning

We will compare the process and outcomes of manual infrastructure provisioning versus automated provisioning using IaC principles. The exercise will demonstrate the issues you will encounter managing a cloud infrastructure manually and how Terraform solves most of these issues.

Course 1

1. Introduction

In today's IT landscape, cloud computing has transformed the way organizations build, deliver, and manage technology solutions. Instead of owning and operating all physical servers and networking equipment, companies increasingly rely on **public cloud providers** to offer computing as a utility: on-demand, scalable, and cost-efficient.

But simply "moving to the cloud" is not enough. To truly unlock the power of cloud services, organizations need **automation**. This is where **Infrastructure as Code (IaC)** comes in, enabling teams to manage and scale infrastructure just like application code—through scripts, templates, and version control.

---

2. What is the Cloud?

2.1 Definition

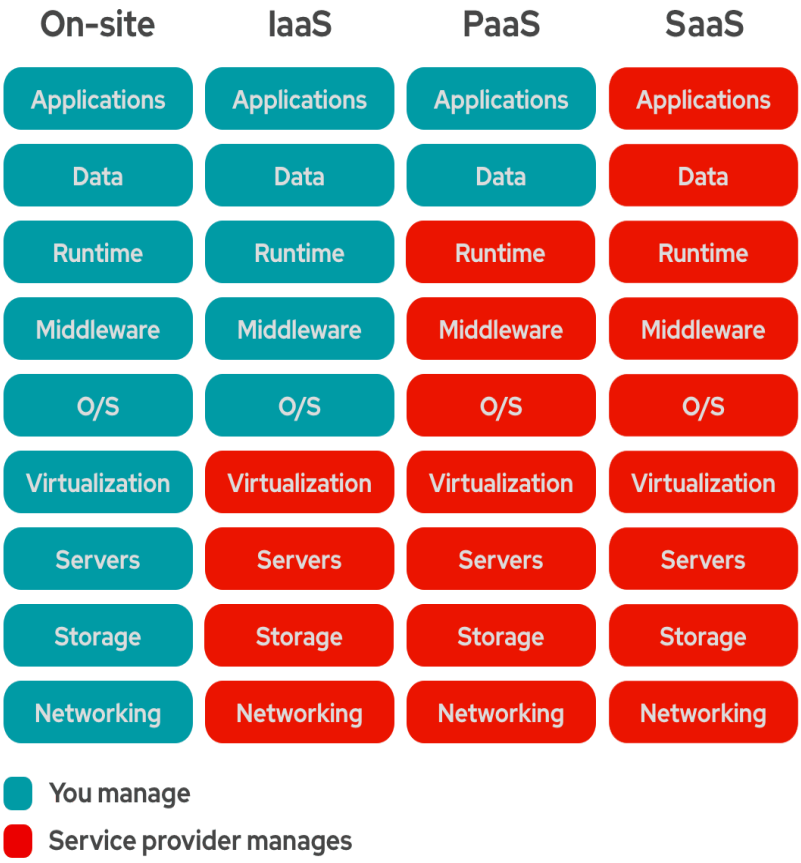
Cloud computing is the **on-demand delivery of computing services** (servers, storage, databases, networking, analytics, etc.) over the internet, with flexible pricing models.

Instead of buying and maintaining hardware, organizations rent IT resources from cloud providers such as **Amazon Web Services (AWS)**, **Microsoft Azure**, or **Google Cloud Platform (GCP)**.

2.2 Service Models

The cloud is commonly categorized into **three service models**:

1. **Infrastructure as a Service (IaaS)**
    - You rent servers, storage, and networking resources.
    - You manage the operating system and applications.
    - **Example**: AWS EC2, Azure Virtual Machines, Google Compute Engine.
2. **Platform as a Service (PaaS)**
    - The provider manages infrastructure and runtime.
    - You focus only on application code and data.
    - **Example**: Heroku, Azure App Service, Google App Engine.
3. **Software as a Service (SaaS)**
    - Fully managed software delivered through a browser.
    - No infrastructure or code management required.
    - **Example**: Gmail, Salesforce, Microsoft 365.



2.3 Deployment Models

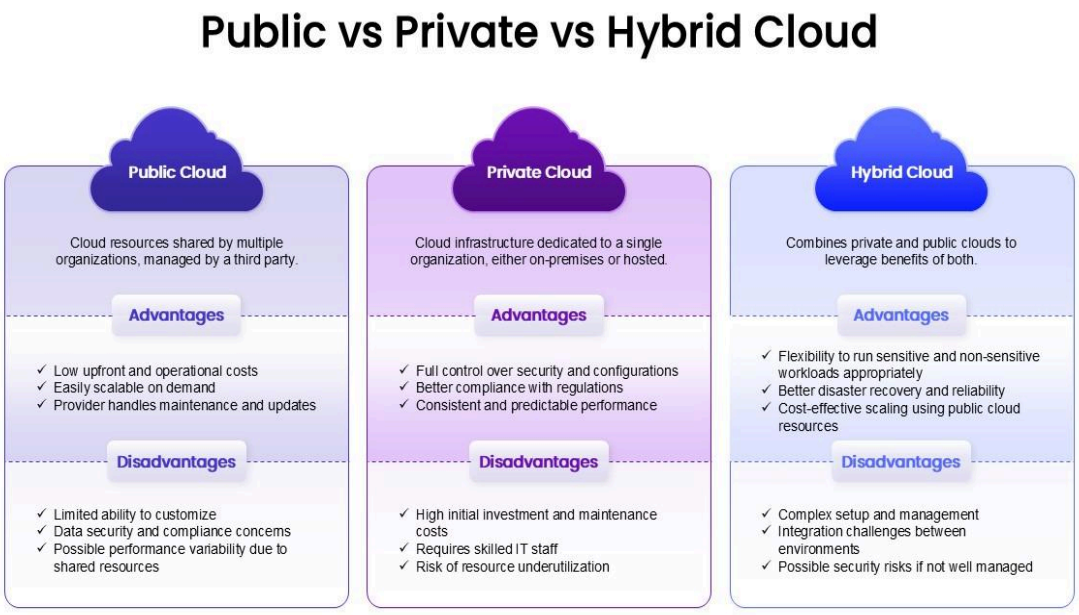
Cloud environments can be deployed in different ways depending on organizational needs:

- **Public Cloud**
    - Shared infrastructure owned by providers.
    - Cost-effective, scalable, and global reach.
    - Public cloud providers offer various levels of geographic distribution and availability:
        - **Globally Distributed:** Services that span across multiple regions and continents, offering high availability and low latency to users worldwide.
        - **Regional:** Services deployed within a specific geographic region (e.g., "us-east-1" for AWS, "eastus" for Azure, "us-central1" for GCP), typically comprising multiple isolated data centers.

        - **Zonal:** Services deployed within a single availability zone, which is an isolated location within a region. Zones are designed to be independent of each other to prevent failures from spreading across zones.
    - Examples: Google Cloud Platform, Amazon Web Services, Microsoft Azure

- **Private Cloud**
    - Dedicated infrastructure for a single organization.
    - Offers more control and security but higher cost.
    - Example: VMware private cloud.
- **Hybrid Cloud**
    - Mix of public and private environments.
    - Useful for compliance, gradual migrations, or burst workloads.

📊 **Diagram:**



---

3. Why Automate Infrastructure?

3.1 The Problem with Manual Management

Traditionally, IT teams managed servers and applications manually:

- Provisioning a new server required ordering hardware, installing OS, configuring security, and networking.
- Changes were made through GUIs or ad-hoc scripts.
- Scaling environments was slow and error-prone.

In the cloud, although hardware ordering is gone, **manual configuration still causes issues**:

- **Human error**: Misconfigurations, unintentional deletion of resources, etc.
- **Inconsistency**: Different environments (dev,uat, prod) may behave differently.
- **Scaling bottlenecks**: Manual setup can’t keep up with fast-moving workloads.
- **Poor documentation**: Your coworkers may not be aware of the changes made and you are exposed to the infamous "it works on my machine" problem.

3.2 Benefits of Automation

By automating infrastructure management, teams achieve:

- **Efficiency**
    - Servers, networks, and databases can be provisioned in seconds.
    - Complex environments reproducible with a single command.
- **Reliability**
    - Reduced human error through standardized configurations.

    - Automated testing and validation of deployments.

- **Centralized Definitions**
    - Infrastructure is stored as code in repositories.
    - Acts as a "source of truth" for the whole team.
- **Scalability**
    - Systems can auto-scale in response to demand.
    - Easily deploy to multiple regions.
- **Repeatability**
    - The same IaC can spin up dev, test, and production environments.
    - Disaster recovery becomes faster and more predictable.

---

4. Infrastructure as Code (IaC) in the Public Cloud

4.1 What is IaC?

Infrastructure as Code (IaC) is the practice of defining and managing infrastructure (servers, networks, databases, permissions) using **machine-readable files** instead of manual processes.

- IaC templates/scripts can be **version-controlled** just like application code.
- Changes can be reviewed, tested, and rolled back.
- Deployments are consistent across environments.

4.2 Common IaC Tools

- **Terraform** (HashiCorp)
    - Multi-cloud support.
    - Declarative language (HCL).
- **AWS CloudFormation**
    - AWS-native IaC solution.
    - YAML/JSON templates.
- **Azure Resource Manager (ARM) Templates**
    - Microsoft Azure-specific IaC.
- **Pulumi**
    - IaC using general-purpose languages (Python, TypeScript, Go).

4.3 IaC + Public Cloud = DevOps Agility

By combining **public cloud services** with **IaC automation**, organizations achieve:

- **Faster delivery cycles**: Applications move from development to production quickly.
- **Governance and compliance**: Infrastructure is documented and auditable.
- **Improved collaboration**: Developers and operations teams share the same source of truth.

---

5. Key Takeaways

- Cloud computing delivers IT resources as a service, with models like IaaS, PaaS, and SaaS.

- Public, private, and hybrid deployment models address different business needs.
- Manual infrastructure management is inconsistent, error-prone and could lead to slower delivery cycles.
- Infrastructure as Code (IaC) automates provisioning, ensures reliability, and enables scalability.
- Public cloud + IaC is a cornerstone of modern DevOps practices.

👉 Next Step: In the following chapter, we will dive deeper into **Terraform** as a popular IaC tool, exploring how it integrates with public cloud providers to automate real-world infrastructure deployments.

Terraform Principles & Best Practices

---

1. Introduction

Why do we use terraform instead of other IaC tools ?

- Terraform is one of the most widely used **Infrastructure as Code (IaC)** tools, enabling organizations to manage infrastructure in a declarative, automated, and predictable way.
- Terraform is cloud-agnostic, so it can be used across most contexts..
- Terraform is well maintained by Hashicorp and, since version 1.0, ensures backward compatibility.

To use Terraform effectively, it's essential to understand three **core principles**:

- **Reproducibility**

- **Idempotence**
- **Versioning**

These principles ensure infrastructure is **consistent**, **reliable**, and **collaboratively managed**.

---

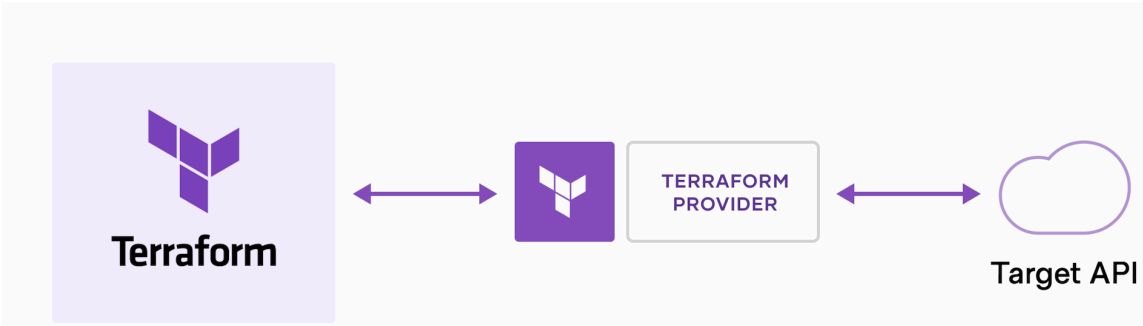
2. How does it work ?

- **Providers**: Terraform uses *providers* (like AWS, Azure, GCP, Kubernetes) to communicate with cloud platforms and services. Each provider exposes resources (VMs, buckets, networks, etc.) that Terraform can manage.

- **State**: Terraform keeps track of the infrastructure it manages in a state file. This file acts as Terraform's memory, mapping your code to real-world resources. It enables drift detection, efficient planning, and collaboration (when stored remotely).
- **Plan**: When you run `terraform plan`, Terraform compares your code (desired state) with the real-world infrastructure (current state) and shows you what changes it will make (create, update, delete).
- **Apply**: When you run `terraform apply`, Terraform executes those changes via the provider APIs, updating your infrastructure so that the actual state matches your declared configuration.

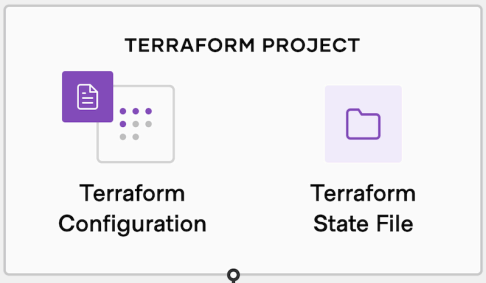
👉 In short: **You write code → Terraform checks what's different (plan) → Terraform makes it real (apply) via providers.**

📊 **Diagram:**



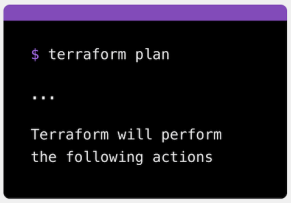
Write

Define infrastructure in configuration files



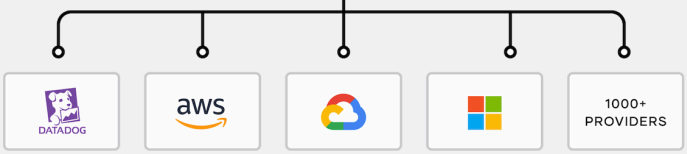
Plan

Review the changes Terraform will make to your infrastructure



Apply

Terraform provisions your infrastructure and updates the state file.



2. Reproducibility

2.1 Definition

Reproducibility means the ability to create **identical infrastructure environments** repeatedly, across development, staging, testing, and production.

The same Terraform configuration file should yield the **same results** each time it is applied, regardless of when or where it runs.

2.2 Why It Matters

- Guarantees consistency between environments.
- Reduces "it works on my machine" problems.
- Simplifies disaster recovery by quickly recreating infrastructure.

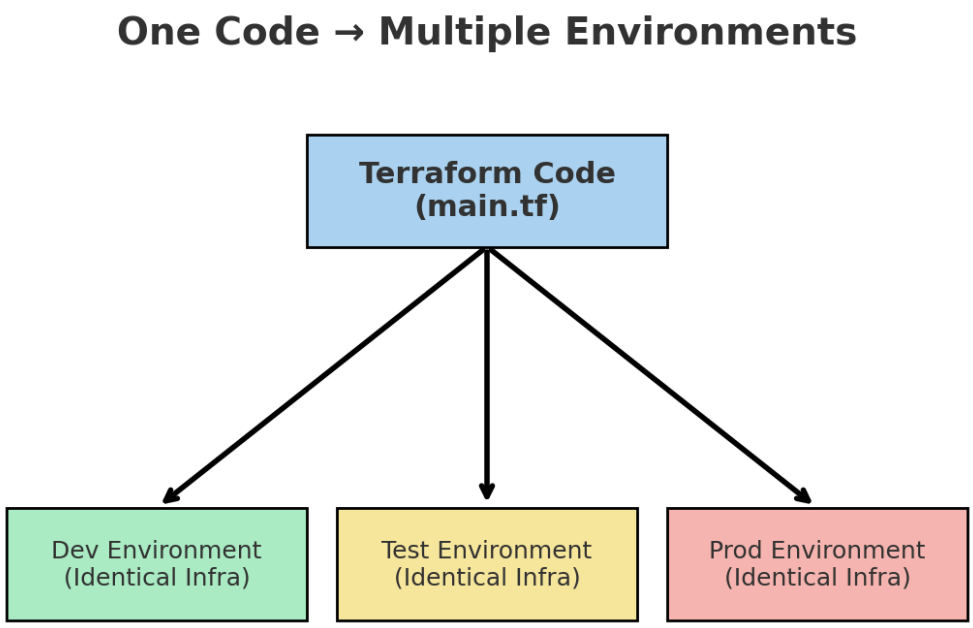
2.3 Example

A simple Terraform configuration to deploy an AWS S3 bucket:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-demo-bucket"
  acl    = "private"
}
```

- Running `terraform apply` in **dev**, **test**, or **prod** with this same file will always create the same bucket configuration.
- This ensures **predictable environments** across all stages of the software lifecycle.

📊 **Diagram:**



---

3. Idempotence

3.1 Definition

Idempotence ensures that applying the same Terraform configuration **multiple times** results in the **same final state**, without creating duplicates or causing unexpected changes.

Terraform continuously evaluates the **desired state** (defined in code) against the **actual state** (in the cloud provider).

3.2 Why It Matters

- Prevents accidental resource duplication.
- Enables safe, repeatable deployments.

- Provides **drift detection**: Terraform identifies when infrastructure has changed outside its control.

3.3 Example

If you define an AWS EC2 instance:

```
resource "aws_instance" "web" {

  ami           = "ami-12345678"

  instance_type = "t2.micro"

}
```

- First `terraform apply` → Terraform creates the instance.
- Second `terraform apply` → Terraform sees that the instance already exists and **does nothing**.
- If someone changes the instance type manually in the AWS console, Terraform will detect **drift** and correct it to match the declared code the next time you'll apply.

---

4. Versioning

4.1 Definition

Versioning refers to storing Terraform configurations in a **version control system (VCS)** such as Git.

This allows teams to:

- Track **changes** over time.
- Roll back to known working configurations.

- Collaborate safely using branches, pull requests, and reviews.

4.2 Why It Matters

- Provides **auditability** (who changed what, and when).
- Enables **rollback** in case of failures.
- Supports **collaboration** between multiple engineers.

4.3 Example Workflow

1. Developer updates Terraform code in Git (e.g., changing an instance type).
2. Change is reviewed in a Pull Request.
3. Merged into `main` branch.
4. Continuous Integration (CI) system runs `terraform plan`, creates a tag, then runs `terraform apply`.

We'll see a GitFlow example in the next chapter.

---

5. Best Practices Recap

- **Reproducibility**
    - Always use the same codebase for all environments. (immutable)
    - Parameterize with variables instead of hardcoding.
- **Idempotence**
    - Regularly run `terraform plan` to check for drift.
    - Avoid manual changes in the cloud console.

- **Versioning**
    - Store all Terraform files in Git (or another VCS).
    - Use branching and code reviews.
    - Use git tags.

---

6. Key Takeaways

- Terraform's power lies in **predictability and safety**.
- **Reproducibility** ensures environments are consistent.
- **Idempotence** guarantees safe, repeatable operations and drift correction.
- **Versioning** brings collaboration, traceability, and rollback capabilities.

By following these principles, Terraform enables scalable, automated, and reliable infrastructure management across any cloud environment.

👉 **Next Chapter: GitOps Concepts & Benefits**

In the next chapter, we'll build on IaC best practices and see how GitOps extends them into a full operational framework, managing not only infrastructure but also applications at scale. We will see how GitOps builds on the foundations of reproducibility, idempotence, and versioning, but applies them at scale—using Git as the single source of truth for both **infrastructure and applications**, and leveraging automation to improve **security, quality, compliance, time-to-market and developer experience**.

GitOps Concepts & Benefits

---

1. Introduction

In the previous chapter, we explored Terraform's **principles of reproducibility, idempotence, and versioning**—all essential for consistent infrastructure management.

GitOps takes these ideas a step further by defining a **framework and workflow** for managing both **infrastructure and applications**. It uses **Git repositories** as the **single source of truth** and ensures that whatever is defined in Git is automatically reflected in the runtime environment.

GitOps is widely adopted in modern **DevOps and cloud-native ecosystems** because it brings **automation, security, and collaboration** together.

---

2. What is GitOps?

GitOps is an operational model where:

- **Infrastructure and applications are defined declaratively** (using IaC tools like Terraform, Kubernetes manifests, Helm charts, etc.).
- **Git is the single source of truth** (all changes are committed, reviewed, and versioned in Git repositories).
- **Automated delivery pipelines** (CI/CD tools like GitLab CI, GitHub Actions, Jenkins) reconcile the desired state (Git) with the actual state (production environment).

---

3. Core Principles of GitOps

3.1 Declarative Configuration

- Infrastructure and application states are described **declaratively**, not imperatively.

- Instead of saying *“create this server step by step”*, you say *“this is the server I need”*.
- Tools (Terraform, Kubernetes, Helm) ensure that the actual environment matches this declaration.

3.2 Git as the Single Source of Truth

- Every change to infrastructure or applications must be committed to Git.
- Git provides a full history of changes, approvals, and rollbacks.
- Eliminates hidden/manual changes (“configuration drift”).

3.3 Automated Delivery

- A CI/CD system (GitHub Actions, GitLab CI, Jenkins, ArgoCD, FluxCD) monitors Git for changes.
- When code is merged or tagged, the pipeline automatically deploys the updated configuration.
- Ensures **continuous reconciliation** between Git and the runtime environment.

---

4. GitOps Workflow Example

Let’s use a simple **GitFlow-style workflow** for GitOps.

4.1 Branching Model

- **Main branch** → represents production-ready state (always stable).
- **Feature branches** → developers create new branches to work on changes.
- **Pull Request / Merge Request** → changes are reviewed before merging into main.
- **Tags/Releases** → deployment triggers are tied to version tags.

4.2 Example Scenario

Developer creates a feature branch:

```
git checkout -b feature/add-s3-bucket
```

1.
2. The developer edits Terraform or Kubernetes YAML files to define a new resource.

3. Code is pushed to GitHub/GitLab, and a **CI pipeline** runs `terraform plan` for validation.

4. After review, the feature branch is merged into **main**.

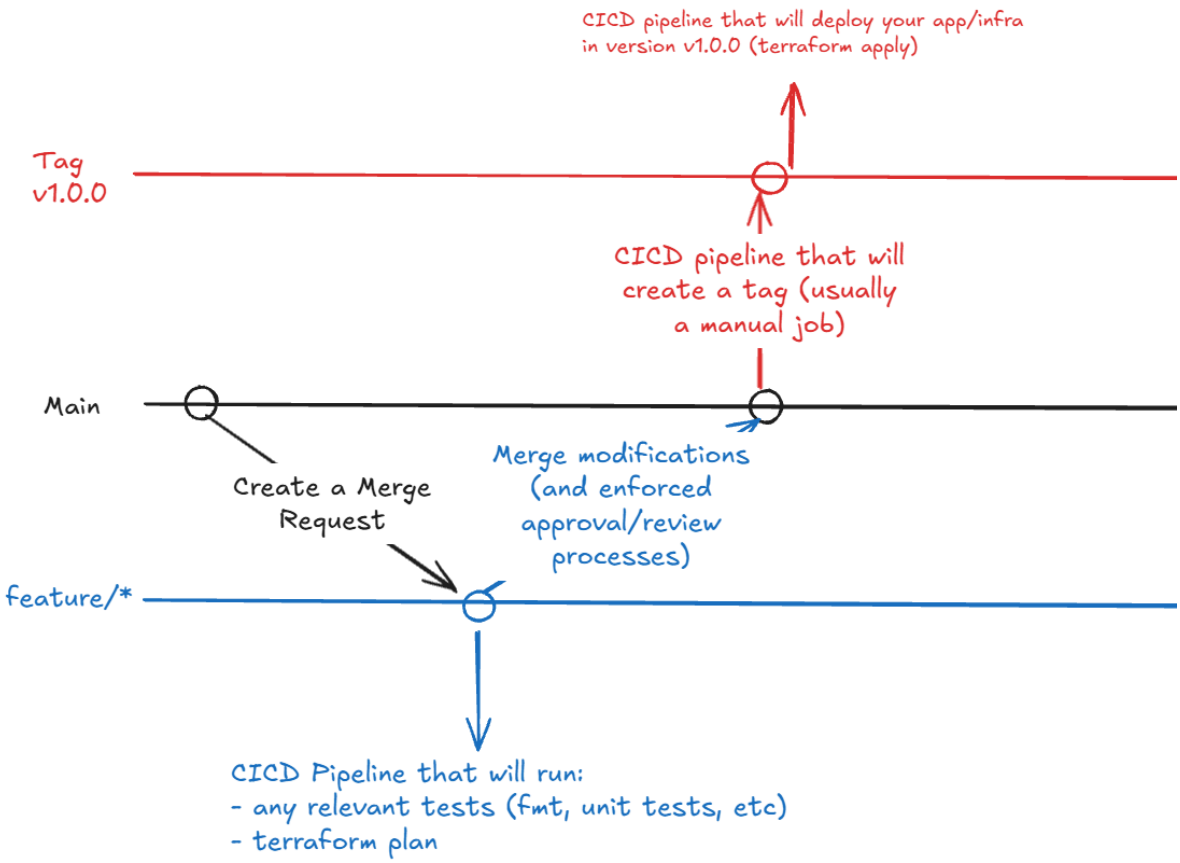
A **tag** (`v1.0.0`) is created, which triggers deployment (ideally automated by your CI/CD pipeline with semantic versioning).

```
git tag v1.0.0
```

```
git push origin v1.0.0
```

5. The CI/CD system (GitLab CI, GitHub Actions, Jenkins) automatically applies the configuration to the production environment.

📊 **Diagram:**



---

5. Benefits of GitOps

5.1 Improved Security & Compliance

- Git requires authentication, access control, and approval workflows.
- Every change is logged, traceable, and auditable.
- Easy rollback to previous states using Git history.
- Integrating code quality and security scanners into your pipelines continuously improves both reliability and security over time.

5.2 Stronger Developer Experience

- Developers use familiar **Git workflows** (branches, commits, pull requests).
- No need to manually interact with infrastructure tools.
- Faster onboarding since Git is universal.

5.3 Consistency & Reliability

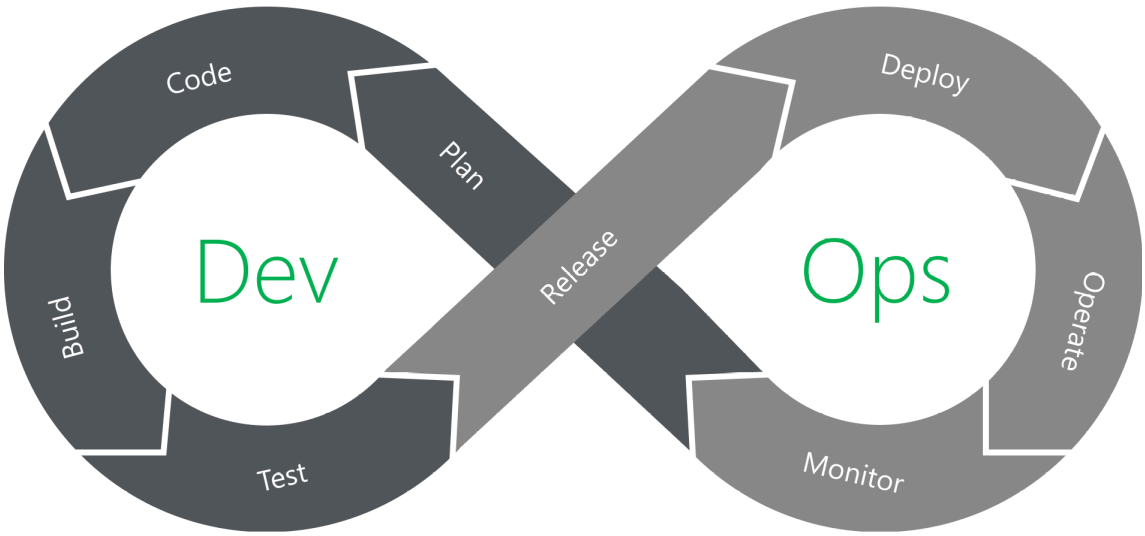
- Declarative + version-controlled = predictable environments.
- Drift is automatically detected and corrected.
- The same process works across dev, staging, and prod.
- Including testing steps within your flow will reduce regressions over time.

5.4 Faster Delivery

- Automation eliminates manual steps.
- Merging code = deploying infrastructure.
- Reduces deployment errors and accelerates release cycles.

Looking at the DevOps cycle, the goal of GitOps is to automate as many steps as possible through CI/CD pipelines.

📊 **DevOps Diagram:**



6. CICD pipelines tools (non-exhaustive)

- **GitLab CI/CD** → Full DevOps platform with integrated pipelines.
- **GitHub Actions** → Event-driven workflows directly inside GitHub.
- **Jenkins** → Highly customizable CI/CD server, often extended with GitOps plugins.

---

7. Key Takeaways

- **GitOps = Git + Declarative Config + Automated Delivery.**
- Git is the **single source of truth** for infrastructure and applications.
- Core principles: **declarative, version-controlled, automated reconciliation**.
- Benefits include: **security, compliance, consistency, faster delivery, and a better developer experience**.
- Common tools: **GitHub Actions, GitLab CI and Jenkins**.

👉 **Next Chapter: Demo**

We will do a quick demo to show the complete workflow:

- **Creating a new branch**
- **Do an edit on the terraform code**
- **Check plan within the CICD**
- **Merge code to main**
- **Versioning**
- **Deployment**