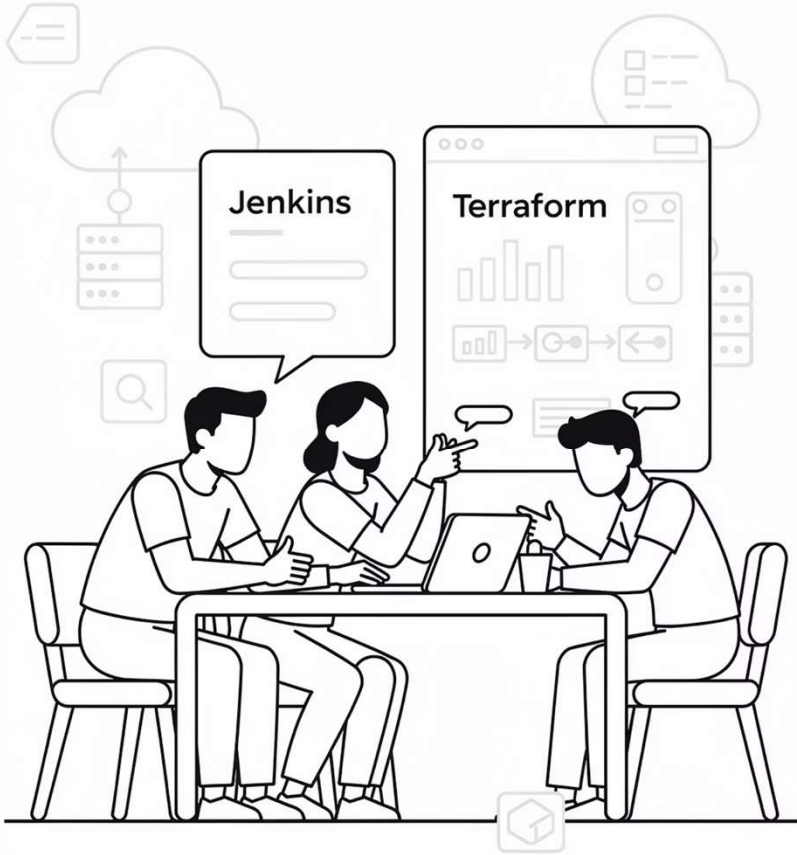
Deep CI/CD Integration

Jenkins + Git Platforms + Terraform in Production

Enterprise DevOps training that makes engineers think like platform teams



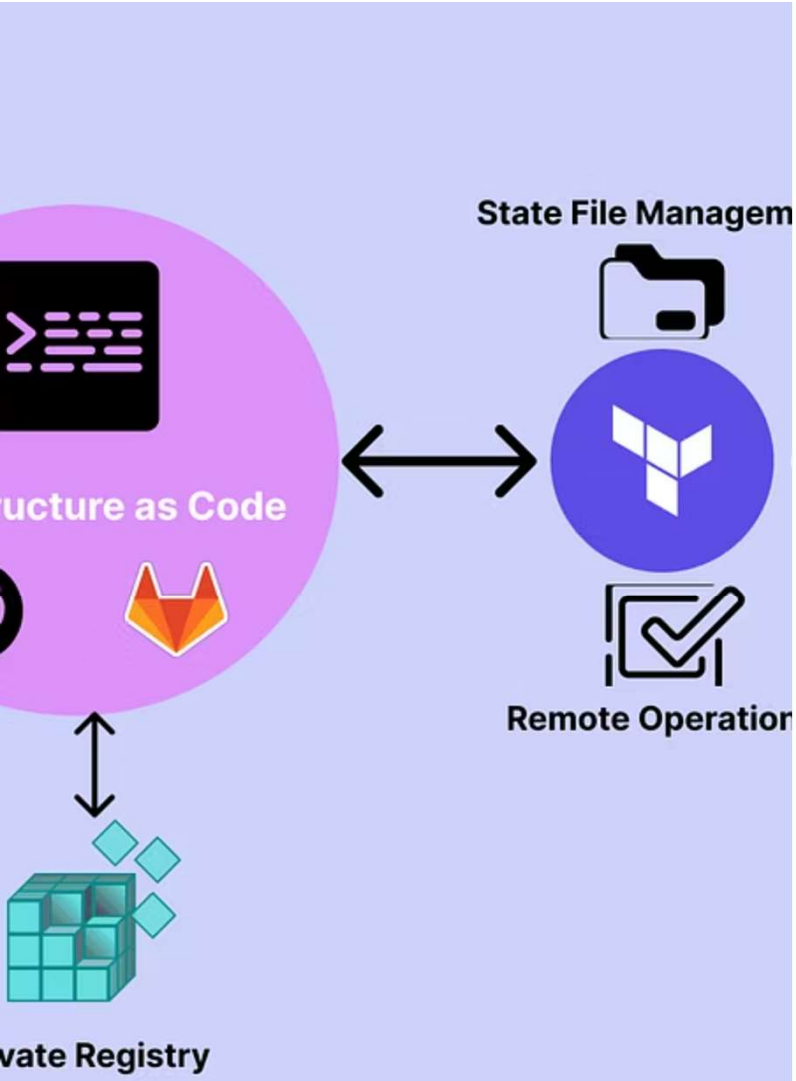
MODULE 1

Deep CI/CD Integration

Jenkins + GitHub / Bitbucket + Terraform

Duration: 2 Hours

Audience: DevOps & Cloud Engineers



Deep CI/CD Integration

Jenkins + Git Platforms + Terraform in Production

Audience: DevOps & Cloud Engineers

Speaker Notes: Introduce the idea that CI/CD for infrastructure is **risk management**, not just automation. This session focuses on **real production controls**, not demo pipelines.

Learning Objectives

By the end of this session, learners will be able to:

Design

Design multi-environment CI/CD pipelines

Implement

Implement PR-based validation and approval gates

Lock

Lock Terraform versions and modules

Secure

Secure remote state and credentials

Recover

Recover from real-world pipeline failures

Speaker Notes: Set expectations: this is **advanced operational Terraform**, not just writing .tf files.

Why Basic Pipelines Fail in Production

Problems with simple pipelines

- Direct terraform apply to production
- No approvals
- No validation
- State locking issues
- Credential exposure risks

Impact

❌ Outages
❌ State corruption
❌ Security breaches

Speaker Notes: Emphasize that **most outages are caused by bad change management**, not bad code.

What a Production-Grade Pipeline Solves

A proper CI/CD pipeline ensures:

1

Controlled deployments

Controlled deployments

2

Environment promotion

Environment promotion strategy

3

State protection

State protection

4

Change traceability

Change traceability

5

Failure recovery

Failure recovery mechanisms

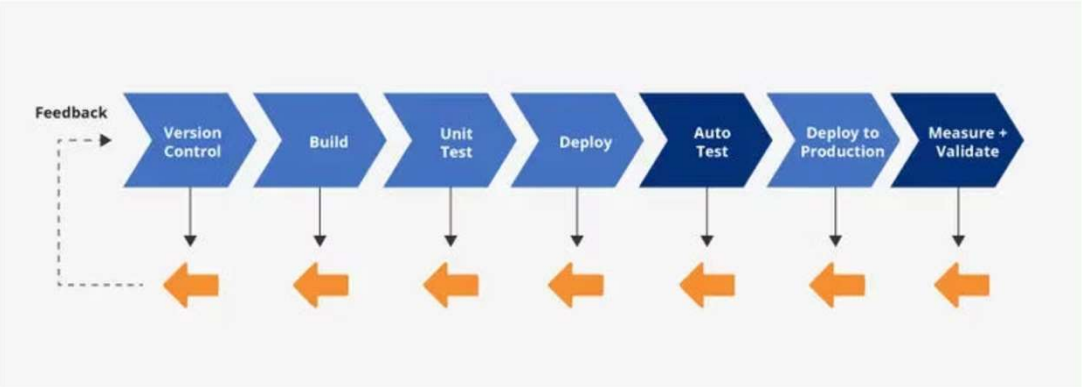
Speaker Notes: This is about **governance + automation** working together.

Multi-Environment Pipeline Model

Environment	Purpose	Risk Level
Feature	Dev testing	Low
Staging	Integration validation	Medium
Production	Live infra	High

Speaker Notes: Each environment represents a **risk boundary**. Promotion = risk acceptance.

End-to-End Deployment Flow



01

Developer → Feature Branch → Pull Request

02

Automated Terraform Validation

03

Manual Approval

04

Merge to Staging → Apply

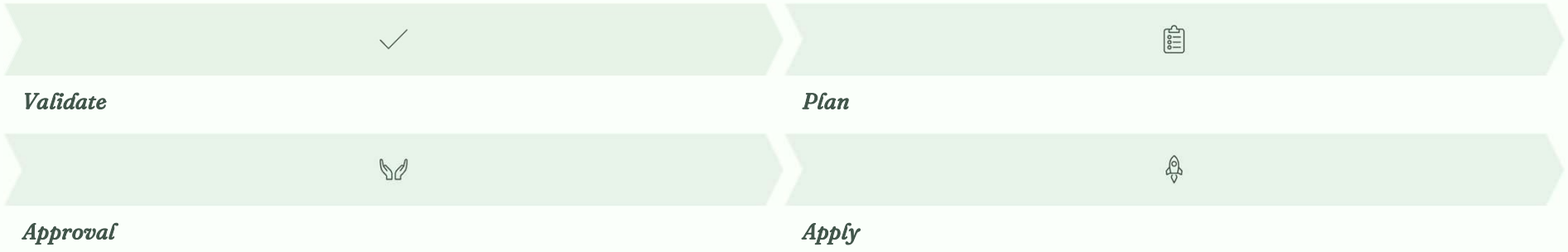
05

Merge to Production → Apply

Speaker Notes: Explain this as **progressive trust** – more validation as risk increases.

Jenkins Pipeline Structure

Show the pipeline stages:




```
stage('Validate') {
  sh 'terraform validate'
}
stage('Plan') {
  sh 'terraform plan -out=tfplan'
}
stage('Approval') {
  input "Approve?"
}
stage('Apply') {
  sh 'terraform apply tfplan'
}
```

Speaker Notes: Stress that **Apply never runs without a human decision** in production.

PR-Based Validation

When a PR is opened:

✓ *terraform fmt*

✓ *terraform validate*

✓ *terraform plan*

```
when {
  changeRequest()
}
```

Speaker Notes: This ensures **bad code never reaches main branches.**

Why Backend is Disabled in PR Pipelines

```
terraform init -backend=false
terraform plan -lock=false
```

Reason

PR checks should **not touch real state**

Speaker Notes: Avoids developers blocking each other with state locks.

Terraform Version Locking

```
terraform {
  required_version = "~> 1.6.0"
}
```

Speaker Notes: Without this, pipelines may break due to **unexpected CLI upgrades**.

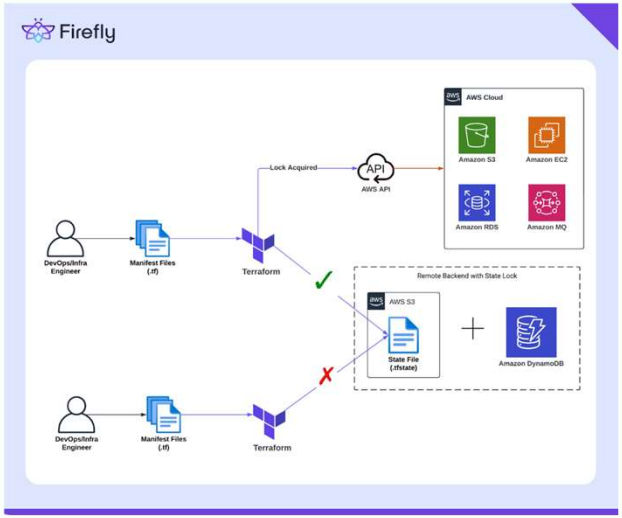
Provider & Module Locking

```
required_providers {
  aws = {
    version = "~> 5.0"
  }
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.0"
}
```

Speaker Notes: Prevents **uncontrolled infrastructure drift.**

Remote Backend & State Locking




```
backend "s3" {
  bucket         = "company-terraform-state"
  dynamodb_table = "terraform-locks"
}
```

Speaker Notes: State = **source of truth.** Locking prevents concurrent corruption.

Secure Credentials in Jenkins

```
withCredentials([...]) {
  sh 'terraform apply'
}
```

Speaker Notes: Credentials should come from **secure stores**, not pipeline code.

Pipeline Failure: Auth Issues

Symptoms

- Access denied
- Expired tokens

Fix

- ✓ Check Jenkins credential bindings
- ✓ Verify IAM roles

Speaker Notes: Most failures = **permissions misconfiguration.**

Pipeline Failure: State Lock

> Error acquiring the state lock

Recovery

```
terraform force-unlock LOCK_ID
```

Speaker Notes: Use only when certain no job is running.

Pipeline Failure: Corrupted State

```
terraform state pull > backup.tfstate
```

Restore from backup.

Speaker Notes: Highlight importance of **state versioning & backups**.

Module Summary

Multi-stage pipelines

PR validation

Version locking

Secure state management

Failure recovery

Final Takeaway

A CI/CD pipeline for Terraform is a safety system, not just automation.

Good pipelines:

- *Prevent bad changes*
- *Enforce governance*
- *Protect state*
- *Recover from failure*