Source Code Management & Terraform Versioning with Git

Mastering version control and infrastructure as code for reliable, collaborative deployments

Git Fundamentals: Branching Basics

Main Branch

Stable, production-ready code that represents your live infrastructure. Every commit here should be deployment-worthy.

Develop Branch

Integration hub where features and fixes come together. Acts as a staging area before production release.

Feature Branches

Isolated workspaces for new features or bug fixes. Enables parallel development without disrupting the main codebase.

Branching enables multiple team members to work simultaneously, experiment safely, and maintain clean separation between development stages.

Pull Requests & Merge Requests: The Gateway to Quality

PRs and MRs are formal requests to merge feature branches into develop or main branches. They're essential quality gates that transform code review from optional to systematic.

Key Benefits

- Enable structured code review and team discussion
- Trigger automated testing and validation pipelines
- Document decision-making and change rationale
- Catch issues before they reach production

Example workflow: Create feature branch → Commit changes → Open PR/MR → Review & test → Merge to target branch

Tagging in Git: Marking Milestones

Create Tags

Label specific commits as releases or milestones using semantic versioning like v1.0.0, v2.1.3

Tag Types

Lightweight tags are simple pointers; annotated tags include metadata, author, and description

Track Deployments

Tags provide version snapshots for infrastructure and application versions deployed to environments

Managing Terraform Code with Git

01

Store Configuration Files

Keep all Terraform files (.tf, .tfvars) in Git repositories like GitHub or Bitbucket for version tracking and collaboration

02

Configure .gitignore

Exclude local state files (.tfstate), .terraform directories, and sensitive data like credentials or API keys

03

Commit Strategically

Make frequent commits with clear, descriptive messages that explain infrastructure changes and their business impact

04

Branch & Review

Use feature branches for infrastructure updates, then submit PR/MR for peer review before applying to environments

Terraform Versioning: Why It Matters

Semantic Versioning Format

Terraform follows Major.Minor.Patch versioning (e.g., 1.8.4). Major versions may introduce breaking changes, minor versions add features, patches fix bugs.

The Risk of Version Drift

Different team members using different Terraform versions can lead to inconsistent behavior, state corruption, and deployment failures.

Version Locking Solution

```
terraform {
  required_version = "~> 1.8.0"
}
```

Lock versions in configuration to ensure predictable, reproducible deployments across all environments.

Controlling Terraform Versions in Git

Commit Version Requirements

Include `required_version` constraints in your terraform block and commit to Git. This creates a source of truth for the entire team.

Tag Stable Releases

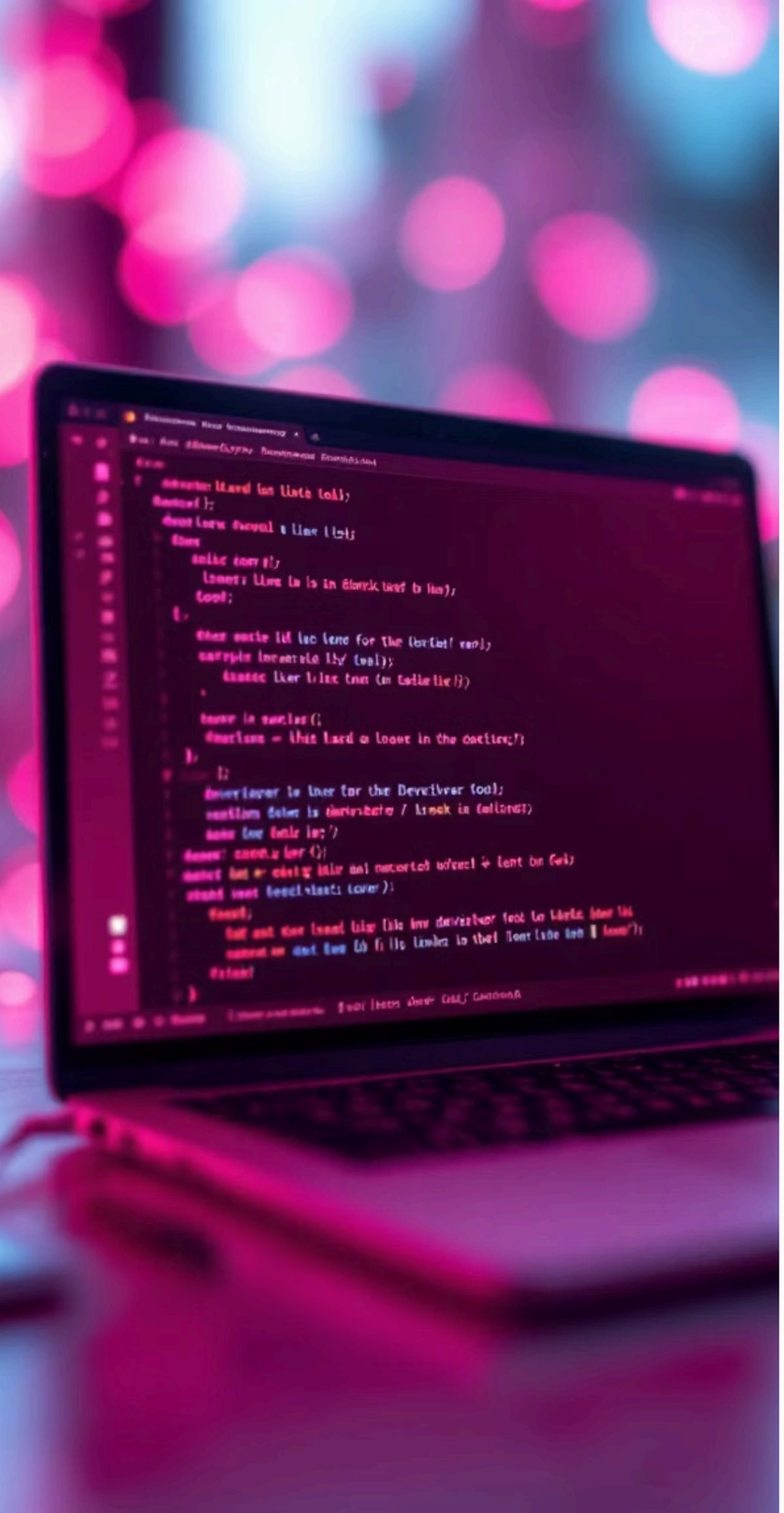
Use Git tags to mark stable Terraform module or infrastructure releases (e.g., v1.0.0-terraform-1.8.4). Tags create immutable reference points.

Coordinate Upgrades

When upgrading Terraform versions, create a dedicated branch to test compatibility. Run full test suites before merging to main.

Document Changes

Use PR descriptions to document migration steps, breaking changes, and rollback procedures for version upgrades.



Tools for Terraform Version Management

tfenv

A Terraform version manager that installs and switches between versions per project

- Automatically reads .terraform-version files
- Switches versions seamlessly
- Supports multiple projects with different versions

.terraform-version File

Specify required Terraform version in a file at your repo root

```
1.8.4
```

tfenv automatically detects and uses this version when you enter the directory

CI/CD Integration

Pipeline automation ensures version consistency

- Read version from .terraform-version
- Install exact version in build agents
- Fail builds on version mismatches

Best Practices Summary

Adopt Git Flow Branching

Use feature, develop, and main branches to isolate changes and integrate them safely through structured workflows

Mandate PR/MR Reviews

Enforce peer review and automated testing gates before merging to ensure code quality and catch issues early

Tag Every Release

Create Git tags for all infrastructure releases to track versions clearly and enable quick rollbacks when needed

Lock Terraform Versions

Specify version constraints in terraform blocks and manage upgrades through dedicated Git branches with full testing

Standardize with tfenv

Use version managers and .terraform-version files to ensure local development environments match CI/CD and production

Empower Your Infrastructure with Git & Terraform

The Winning Formula

Git + Terraform versioning = reliable, collaborative infrastructure as code that scales with your team and reduces deployment risk

Start Today

Integrate Git workflows with Terraform version control for safer deployments, better collaboration, and infrastructure you can trust

Get Started with Git & Terraform