Infrastructure as Code

7.5.1

Why Store Infrastructure as Code?

At this point, it is time to introduce a new term: **immutability**. This literally means "the state of being unchangeable," but in DevOps parlance, it refers to maintaining systems entirely as code, performing no manual operations on them at all.

These topics have touched several times on the concept of treating infrastructure as code. But thus far, it has mostly been preoccupied with the mechanics. You know that it makes sense to automate deployment of full stacks, which are virtual infrastructures (compute/storage/network) plus applications. You have seen several approaches to writing basic automation code, and at the mechanics of automation tools, as well as storing code safely and retrieving it from version control repositories.

You are now familiar with the idea of idempotency and related topics, and have seen how it might be possible to compose automation code that is very safe to run. This is code that does not break things, but instead puts things right and converges on the desired state described by a (partly or wholly) declarative model of the deployed system. You have also learned how code like this can be used to speed up operations, solving problems by brute force rather than detailed forensics and error-prone, time-consuming manual operations.

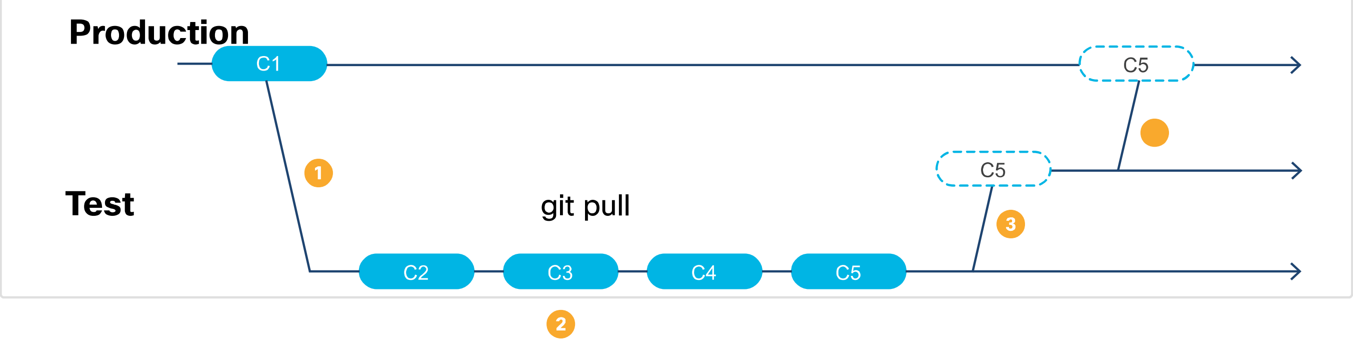
GitOps: modern infrastructure-as-code

Committing to immutability enables you to treat your automation codebase the way you would any application code:

- You can trust that the codebase describes what is actually running on bare metal or cloud servers.
- You can manage the codebase Agile procedures and structured use of version control to keep things clear and simple.

This is "GitOps", also referred to as "operations by pull request." In a typical GitOps setup, you might maintain a repository, such as a private repo on GitHub, with several branches called "Development," "Testing/UAT," and "Production."

GitOps is "Operations by Pull Request"



In this model, each branch deploys to segregated infrastructure.

- **Development -** Developers make changes in the Development branch, filing commits and making pull requests. These requests are fulfilled by an automated gating process and queued for automated testing. The operators see results of automated testing, and developers iterate until tests pass. The changes are then merged into the Test branch for the next level of review.
- **Test -** When changes are merged from Development into Test, the code is deployed to a larger test environment and is subjected to a more extensive set of automated tests.
- **Production -** Tested changes are again reviewed and merged to Production, from which release deployments are made.

By the time tested code is committed, evaluated, iterated, and merged to the Production branch, it has gone through at least two complete cycles of testing on successively more production-like infrastructure, and has also had conscious evaluation by several experts. The result is code that is reasonably free of bugs and works well. This operational model can be used to develop extensive self-service capability and self-healing compute/storage/network/PaaS, or implement large-scale distributed applications under advanced container orchestration.

Where can GitOps take you?

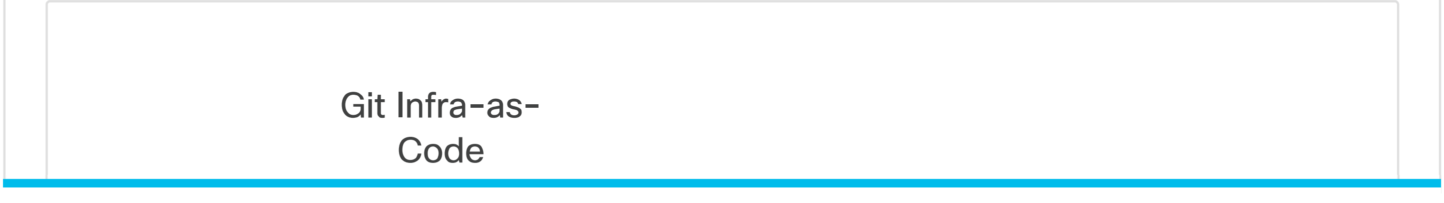
When your GitOps procedures and workflow, gating/CI-CD, automated testing and other components are in place, you can begin experimenting with elite deployment strategies that require a bedrock of flawless automation.

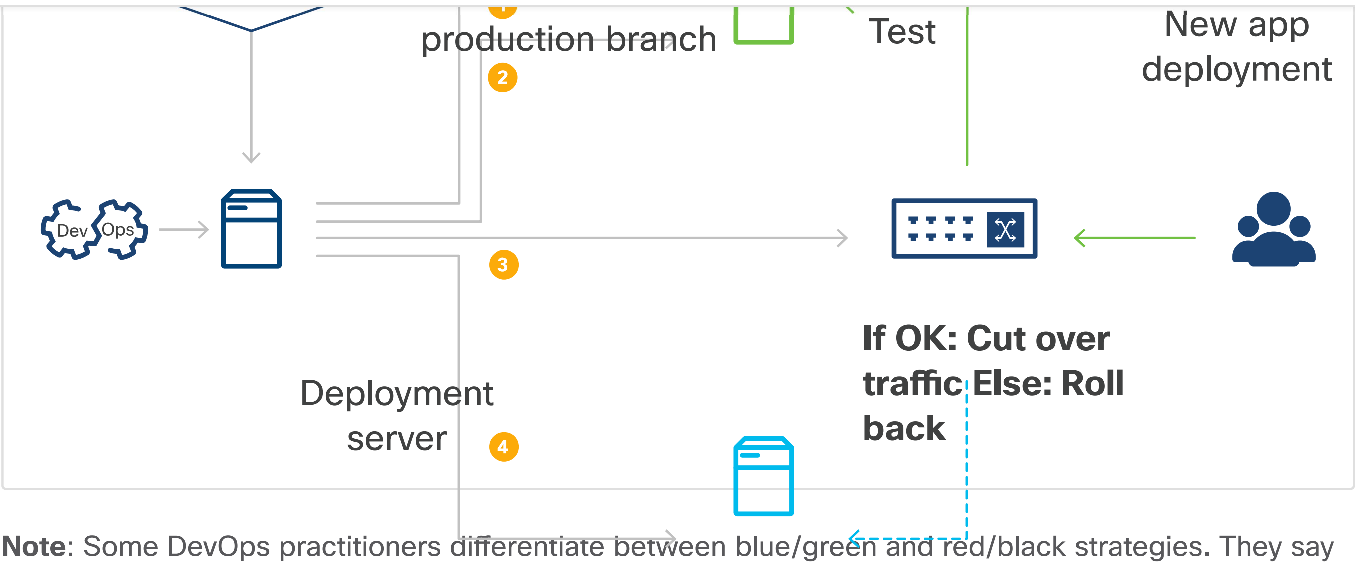
Blue/green deployments

Blue/green deployment is a method for reducing or eliminating downtime in production environments. It requires you to maintain two identical production environments (You do not have to call them Blue and Green. Any two colors such as Red and Black will do.). Develop the capability of quickly redirecting application traffic to one or the other (through ACI automation; load balancing; programmable DNS; or other means).

A release is deployed to the environment not currently in use (Green). After acceptance testing, redirect traffic to this environment. If problems are encountered, you can switch traffic back to the original environment (Blue). If the Green deployment is judged adequate, resources owned by the Blue deployment can be relinquished, and roles swapped for the next release.

Blue/Green Deployment





Note: Some DevOps practitioners differentiate between blue/green and red/black strategies. They say that in blue/green, that traffic is gradually migrated from one environment to the other, so it hits both systems for some period; whereas in red/black, traffic is cut over all at once. Some advanced DevOps practitioners, like Netflix, practice a server-level version of red/black they call "rolling red/black". With rolling red/black, servers in both environments are gradually updated, eventually ending up with one version distinct from one another, and can be rolled back individually as well. This means that *three* (not two) versions of the application or stack are running across both environments at any given time.

Canary testing

Canary testing is similar to rolling blue/green deployment, but somewhat more delicate. The migration between old and new deployments is performed on a customer-by-customer (or even user-by-user) basis, and migrations are made intentionally to reduce risk and improve the quality of feedback. Some customers may be very motivated to gain access to new features and will be grateful for the opportunity. They will happily provide feedback on releases along efficient channels.

7.4
Automation Tools

7.6
Automating Testing