Engineer

Infrastructure as code

managing infrastructure from versioned definition files

What it is

Infrastructure as code is the practice of describing the things a system runs on (machines, networks, firewall rules, load balancers, DNS records) in text files, keeping those files in version control, and letting a tool build the real environment from them. The files are the master copy. A change to the infrastructure starts as a change to a file, passes the same review as application code, and reaches the running system only when the tool applies it. The term is also written with hyphens, infrastructure-as-code, and shortened to IaC.

NIST's Special Publication 800-204C, a guide to DevSecOps for applications built from microservices, describes infrastructure as code as the code that provisions and configures the resources an application is deployed on, "in a repeatable and consistent manner". The same guide says these definitions are kept under version control with the same tools as the application's own source code. Repeatability is the whole point. An environment that can be produced from files can be produced again, next to the first one, after a disaster, or in a test account on a Tuesday afternoon.

Definitions come in two broad styles. A declarative definition states the end result, say a server of a given size in a given network with one port open, and leaves the tool to work out what to create, change or leave alone. An imperative definition is a list of steps: create this, attach that, then open the port. SP 800-204C files infrastructure as code under declarative code: the team writes down what it requires, and a tool turns that into a running instance. HashiCorp describes the Terraform language in the same terms, as a description of an intended goal rather than of the steps that reach it. The difference shows on the second run. A list of steps records how something was done once, while a declaration records what should exist, and that is what the tool needs to know when part of it already does.

There is no single language. SP 800-204C notes that, depending on the tool, the code is written either in a general scripting language such as Python, JavaScript or TypeScript, or in a configuration language of the tool's own, such as HashiCorp's HCL, which may or may not be compatible with a standard format like JSON. Terraform's language is of the second kind and lives in files ending in `.tf`. Azure accepts Resource Manager templates in JSON or in the shorter Bicep language. Ansible writes its playbooks in YAML and reaches machines over SSH with their existing operating-system credentials, without installing an agent on them.

It helps to separate two jobs that both fall under the name. Provisioning creates the pieces: the virtual machine, the disk, the network, the firewall. Configuration shapes what runs inside a machine that already exists: packages, accounts, service files. Some tools lean towards one job and some towards the other, and the boundary is soft. What makes either of them infrastructure as code is that the definition lives in a repository and the tool brings the real system into line with it.

Tools also differ in how they remember what they built. Terraform keeps a state file that maps the resources in the configuration to the real objects it created, and its documentation describes that state as the source of truth for the environment. Bicep keeps no state file, because Azure itself records what was deployed. Where the record lives decides what happens when two people run the tool at once, or when the record is lost.

Use the concept in a decision

Put "What is infrastructure as code?" to work in a real decision. We can help relate the concept to the system you are building or buying. The whole setup written down as files, so it can be recreated exactly and never lives only in one person's head.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

Before infrastructure as code, each environment was set up by hand and then patched by hand. Staging received a fix that production did not, a port opened for a test was never closed, one machine out of three got a newer package. SP 800-204C calls this drift: the configuration in place no longer gives the

operational state that was intended. It also names the usual way in, a change made in a web console or on the command line that goes around the code. When a release works in staging and fails in production, drift is a common reason.

Infrastructure as code moves the master copy out of the running system. The team changes the definition, and the running system follows it. Nobody logs in to change a firewall rule; they change the file that describes the firewall, and the tool carries the change out. Three things follow from that.

The first is review and history. A change to the network becomes a diff in a pull request that a second person reads before anything happens. Months later, the question of who opened a port, and why, has an answer in the commit log instead of in somebody's memory. Undoing a change is a revert followed by another run of the tool.

The second is repeatability. A well-written definition is idempotent: applied to an environment that already matches, it changes nothing, and applied to one that has drifted, it brings it back. There are two ways to get there: adjust what is there, or discard it and build a fresh one from the definition. The second way, taken as a rule, is immutable infrastructure.

The third is cheap environments. SP 800-204C describes the tools as automating the whole life of the infrastructure, from provisioning to de-provisioning, and notes that some of it lives too briefly to be worth maintaining at all. A team can create several test environments on demand and tear them down again, and the code itself can be checked before it touches anything: the guide asks for every piece of it to be scanned for vulnerabilities before it is merged. A branch can get its own short-lived copy of the production layout, and a reviewer can see a preview of a change before approving it.

The costs deserve a plain statement. Changing one firewall rule through a pull request is slower than changing it in a web console, and during an outage that feels expensive. Each tool has its own language and its own model of state to learn. And a file that can build an environment can also delete one: the lever reaches just as far in both directions, which is why the preview matters as much as the run that follows it.

What it looks like in practice

Here is one change travelling from an edit to the running system, in a setup where a CI job runs the tool and posts its dry run on the pull request:

```
Day 1   09:10  Edit the network definition: allow TCP 443 from the office
        09:12  Open a pull request; CI runs a dry run (the plan)
        09:14  Plan posted on the pull request: one rule added, none removed
        10:30  A reviewer reads the plan as well as the diff, and approves
        10:31  Merge; CI applies the plan that was reviewed
        10:33  A second run of the same definition reports no changes
Day 40         Someone asks why 443 is open: the commit log says who and why
```

The plan is the part that earns its keep. The diff shows what the author meant to change; the plan shows what the tool is about to do, including anything the author did not expect, such as a resource it intends to destroy. Terraform can save a plan to a file with `-out`, and `terraform apply` given that file performs the operations in it, so what runs at 10:31 is what the reviewer approved at 10:30. The quiet second run at 10:33 is the check that the definition and the system now agree.

On this site, the Infrastructure as Code service page describes repeatable server provisioning from versioned configuration rather than relying on somebody’s memory.

Getting there safely is mostly a matter of order, and a portfolio entry from September 2026 shows why. It adds a second server, hosted with another provider. The earlier machine had been set up by hand and protected later, so for a while a newly created host on a default password policy sat open to the internet. This time the server was created from the repository, and the protection was in place before the machine was. The firewall play is run ahead of the server play, and at the second provider the tags are assigned before the cloud firewall, since rules aimed at tags have nothing to attach to until the tags are there. A rerun does no harm, because the creating role takes over a machine that already exists. That role is the only one that costs money, so its own documentation records the region, the machine type and its specification, and the price per month in euros. The entry is also candid about what a dry run misses. Both firewalls are configured by calling the providers’ REST APIs from a general-purpose HTTP task, and that task does not support check mode, so a dry run passes over it. The real rehearsal is a run of the firewall play on its own, which costs nothing and can be undone.

Where it goes wrong

Infrastructure code tends to fail in ways that look like success. The tool finishes cleanly, the CI run is green, and the files and the real system quietly disagree.

A hand fix made during an incident is undone by the next run. Someone opens a port in the console at two in the morning, the outage ends, and a week later the routine run closes it again, because the files never heard about it. The opposite also happens: nobody runs the tool again, and the files stop describing reality. Put the emergency change into the code the same day, and run a scheduled check that compares the files with what is running; SP 800-204C asks for that comparison straight after each deployment and at intervals while the system runs. With Terraform the plan command's detailed exit code makes that a short nightly job:

```
terraform plan -detailed-exitcode
# exit 0: no changes, exit 1: error, exit 2: changes present
```

On a branch that has already been applied, an exit code of 2 means the real system has moved away from the code, and someone should find out why.

Deleting a resource from the file does not always delete it. Tools disagree about what a missing resource means. For Azure Resource Manager templates, Microsoft's what-if documentation states that a resource which exists but is not in the template is deleted only in complete mode. In any other mode it keeps running and keeps costing money. Learn your tool's rule for removal before relying on it, and read the preview for the deletions you expect as well as the ones you fear.

The dry run does not see everything. A preview is a model of the change, not a guarantee. Microsoft's what-if documentation lists properties reported as deleted when Azure only fills in a default during deployment, and calls the result noise; any property set through the `reference` function shows as changing on every run. The portfolio entry above shows the other gap: a task without check-mode support is skipped in a dry run, so the preview says nothing about it. Know your tool's known noise, and treat whatever the preview cannot see as untested until a rehearsal on something cheap and reversible covers it.

Secrets end up in the repository or in the state. A password written into a definition file stays in the history of every clone. Less obvious: Terraform's documentation warns that its state can hold secrets, and by default that state is a local file called `terraform.tfstate`. Keep secrets in a secret store and refer to them by name, keep state in remote storage with locking and access control, as HashiCorp recommends, and never edit the state file by hand; the `terraform state` command exists for the changes that are needed.

A script is dressed up as a definition. A shell script in the repository that creates a server is under version control, but it is not idempotent: run it twice and there are two servers, or an error halfway through. Use declarative resources, or steps that check before they act, such as the role above that adopts an existing machine. Ansible's documentation sets the standard plainly: when a system is already in the state a playbook describes, running the playbook again changes nothing.

Further reading

- Implementation of DevSecOps for a Microservices-based Application with Service Mesh (NIST SP 800-204C) — infrastructure as declarative, versioned code, drift and scanning.
- Terraform Language Documentation — the declarative language, its resources and its blocks.
- Command: plan — the dry run, saved plans and the detailed exit codes.
- State — what Terraform records, where it keeps it, and why it is sensitive.
- Introduction to Ansible — agentless automation over SSH, playbooks and desired state.
- Template deployment what-if — previewing an Azure deployment, its change types and its known noise.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is infrastructure as code?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/infrastructure-as-code/