Engineer

immutable infrastructure

❄ replacing servers instead of changing them in place

What it is

Immutable infrastructure is a way of running servers in which a machine, once it is in service, is never changed. HashiCorp's Well-Architected Framework describes it as infrastructure that, "once deployed, is never modified—only replaced". A new version of the application, an upgraded library, a security patch and a changed setting all arrive the same way: as a new machine or container, built from a new image, which takes the place of the old one. The old one is then switched off and deleted.

The opposite is mutable infrastructure, where a long-lived server is updated where it stands. In June 2013 Chad Fowler described what such a server turns into after years of upgrades: patched over and over, carrying configuration edits made by hand in the middle of an outage, running cron jobs that only one person knows about, until the team is afraid both to change it and to replace it. His remedy, which he called immutable deployments, was to build every system through automation and leave it untouched from then on. An upgrade or a new release then means building a fresh server or image and throwing the old one away.

Two kinds of image make the approach practical. A machine image, in the words of Packer's documentation, is a single static unit that holds a pre-configured operating system and the installed software. Packer, a community tool, builds them: from one source configuration it produces matching images for several platforms. Container images are the second kind, and Docker's documentation is plain about them: once an image has been created it cannot be modified, and the only options are to make a new image or to add changes on top of it as new layers.

The term describes how a system is operated, not a product you install. It also has a boundary. The application servers can be replaced on every release, but the data they work on has to live somewhere that outlasts each replacement, and that boundary is where most of the practical design work goes.

Use the concept in a decision

Put “What is immutable infrastructure?” 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

The first reason is drift. HashiCorp’s framework names it directly: manual updates and in-place modifications make servers diverge from their original configuration over time. Two servers that started from the same setup a year apart need not end up identical, because each received a different sequence of fixes, retries and hand edits. When every server is a fresh copy of one image, the image is the complete description of the server, and there is nothing left to drift.

The second reason is rollback. If a release misbehaves, the framework’s answer is to deploy the previous artifact again rather than repair the servers that are running. The previous image is a known state that has already run in production, so the team does not have to reconstruct what the server looked like yesterday, under pressure, from memory and shell history.

The third reason is fewer hands on production. The framework points out that connecting to production servers to apply updates by hand introduces human errors such as typos and forgotten steps, and that with immutable infrastructure nobody has to log in to a server by hand at all. Fewer people need a login on a live machine at all, and the ones who have one rarely need to use it.

The fourth reason is the audit trail. Auditors, the same page notes, want proof that production has not been modified outside approved processes, and manual changes leave incomplete records. When every change is a new artifact, version control and the CI/CD logs show what changed, when, and who approved it.

A security patch follows the same path as a feature release: it becomes a new artifact that replaces the running one, so it is built, tested and recorded the same way.

None of this is free. The framework notes that starting a virtual machine takes longer and consumes more resources than starting a container, while containers need an orchestrator to run them. Every change, even a one-line setting, now means a full build and a replacement, so the build has to be automatic and quick enough that nobody is tempted to skip it. State has to be moved off the replaceable machines before the first replacement, not after.

What it looks like in practice

A Packer template describes how to build a machine image. This one is based on the example in the documentation for Packer's Amazon EBS builder, with a `build` block added in the form shown on the build block reference:

```
source "amazon-ebs" "web" {
  access_key    = var.aws_access_key
  secret_key    = var.aws_secret_key
  region        = "us-east-1"
  source_ami    = "ami-fce3c696"
  instance_type = "t2.micro"
  ssh_username  = "ubuntu"
  ami_name      = "web {{timestamp}}"
}

build {
  sources = ["source.amazon-ebs.web"]

  provisioner "shell" {
    scripts = fileset(".", "scripts/{install,secure}.sh")
  }
}
```

The `source` block names the builder and the base image, `source_ami`, whose root volume is copied and provisioned on a running instance. The `build` block says which sources to build and how to provision them. The shell provisioner runs the two scripts inside that running machine before it is turned into a static artifact, and the artifact is a new image. The declarations of the two variables are left out here. One command runs every build in the template and produces the artifacts:

```
packer build web.pkr.hcl
```

Deployment is the second half. HashiCorp's framework puts the two tools side by side: images are built with Packer, deployed with Terraform, and replaced entirely when they need an update. When a change cannot be made in place, Terraform's default, according to its lifecycle reference, is to destroy the existing object and then create the replacement. Updates in this flow do not require logging in to a running production server. A change to the scripts produces a new image, the image produces new machines, and the old ones are removed.

The same discipline scales down to one host. The portfolio entry on running containers on Podman and Quadlet keeps provisioning and deployment apart. Root prepares the deployment path a single time; after that, releases are made by an ordinary user with no special rights, who may run exactly one script and nothing else. A release is a fresh image followed by a restart of the service, without a root shell or free-form commands. The entry on CI/CD with GitHub Actions goes further on the image side: the production frontend runs on a distroless base that has neither a shell nor a package manager, and a build is promoted from one environment to the next by the workflows, not by a person. An image like that leaves almost nothing to log in to and no tool to install anything with, so building its successor is the only sensible way to change it.

The entry on servers you can rebuild from scratch is a careful example of the other model. There, Ansible brings a running host to the state written in the repository, each play is idempotent, and the repository supplies the configuration for rebuilding the declared environment. The entry is open about what in-place change costs. Any change takes longer than a direct edit on the server, and a run that stops halfway leaves the host worse off than a run that declines to start; that is the reason a preflight play now runs ahead of it. A host left half-changed is a step towards the fragile server Fowler described, and replacing the machine instead of changing it is the other way around that risk.

Where it goes wrong

The model is simple to state and easy to erode. Most failures come from one exception that seemed harmless at the time.

A hot fix over SSH brings drift back. Under pressure, someone logs in to a running server and edits a file. The fix works until the next deployment replaces the machine, and then it is gone, along with any record of why it was needed. Until then, the running server no longer matches its image, which is exactly the condition the approach exists to prevent. The fix is to send every change, the urgent one included, through the image build, and to keep that build fast enough to be the quickest route. HashiCorp's guidance rules out connecting to a live server for hand edits or updates altogether.

Data written to the machine is deleted with it. Docker's documentation says that when a container is deleted, the file changes made inside it are deleted too. Uploads, generated files or a local database on a replaceable machine disappear at the next release. The fix is to keep state outside the part that gets replaced. Docker volumes persist data beyond the lifecycle of a single container, and a new container attached to the same volume finds the files still there; for machines, the same role falls to a separate disk or a database service that is not part of the image.

Replacement takes the service down. Terraform's default order is to destroy the existing object first and create the replacement second, which leaves a window with nothing running. The `create_before_destroy` setting in a resource's `lifecycle` block reverses that order:

```
resource "aws_instance" "web" {
  # ...
  lifecycle {
    create_before_destroy = true
  }
}
```

HashiCorp makes this opt-in because two copies of one object cannot always exist side by side: many object types demand a unique name or carry other limits. Its reference asks you to learn each resource type's constraints before relying on the setting.

A rebuild quietly starts from a newer base. Instead of a fixed `source_ami`, the Amazon EBS builder can fill it from a `source_ami_filter`, and with `most_recent` set to true the filter selects the newest matching image, which the documentation calls most useful for a daily distro build. That is convenient for staying current, but two builds of the same template on different days then start

from different bases, and rebuilding an old release no longer reproduces it. The fix for release images is to pin `source_ami` to a specific image and move it forward deliberately, in a reviewed commit.

Secrets are baked into the image. HashiCorp's framework is firm that artifacts must never contain secrets such as database passwords, API keys or encryption keys: whoever can read the artifact can pull them out, and changing a credential then forces a rebuild and redeploy of every artifact. The fix is to fetch secrets when the machine or container starts. HashiCorp's own pattern uses Vault, where a starting machine authenticates and receives temporary credentials that expire, and credentials are rotated in Vault without rebuilding anything. The template above follows the same rule for its own keys, which arrive as variables rather than as text in the file.

The previous artifact is gone when it is needed. Rollback means redeploying the previous artifact, which assumes it still exists. A cleanup job that deletes old images, or a habit of reusing one image name for every build, turns a rollback into a rebuild from old source on a base that may have moved. The fix is to keep the images of recent releases that ran in production, give each one a name that is never reused, and rehearse a rollback to one of them before the day it matters.

Further reading

- Create immutable infrastructure — the Well-Architected Framework on drift, rollback, image types and secrets.
- Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components — Chad Fowler's June 2013 post, read through an Internet Archive copy.
- Introduction to Packer — what Packer builds and what a machine image contains.
- Amazon EBS builder — the builder used above, with `source_ami` and `source_ami_filter`.
- lifecycle meta-argument reference — Terraform's destroy-then-create default and `create_before_destroy`.
- Persisting container data — why files written inside a container go with it, and how volumes keep them.



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 immutable infrastructure?” 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/immutable-infrastructure/