Engineer

# Deployment

moving a change to where people actually use it

## What it is

Deployment is the sequence of steps that takes a tested version of an application out of the build system and puts it into service: placing the built files or container image on the machines or platform that will run it, handing it the configuration for that environment, starting it, and moving real requests over to it. The word names both the act and the result, so a team talks about a deployment that stalled at the database step and about the deployments currently running in each environment. What gets deployed can be a website, an API, a scheduled job, a database change or the backend behind a mobile app.

Deploying and releasing are often kept apart. Deploying puts the new code where it can run; releasing decides who sees it. A feature hidden behind a flag can be deployed on Monday and released to customers on Thursday without touching the servers again. Keeping the two separate lets a team ship code in small, frequent steps while product decisions about timing stay with the people who own them.

Many deployments today ship a container image. The Open Container Initiative, set up under the Linux Foundation in June 2015, publishes three specifications around it: how an image is laid out, how a runtime runs one once it is unpacked, and how registries store and hand images out. An image is identified by its digest, a hash of its content, so the same digest always means the same bytes. A tag such as `1.4.2` is a readable pointer to an image, and pushing again under that tag points it at something new.

The Twelve-Factor App, a widely cited methodology for building web services, splits the path from code to running service into three stages kept strictly apart. The build stage turns a commit into an executable bundle. The release stage combines that bundle with one environment’s configuration. The run stage starts

processes from a chosen release. Each release gets a unique identifier and is never changed after it exists; any change makes a new release, which is what turns rolling back into a matter of choosing an earlier one.

How the switch happens varies with what the service can tolerate. A recreate deploy stops the old version, then starts the new one, and accepts a gap in between. A rolling deploy replaces instances a few at a time, so both versions serve traffic for a while; it is the default for a Kubernetes Deployment. A blue-green deploy runs the new version beside the old and moves all traffic at once, keeping the old one ready to take it back. A canary sends a small share of traffic to the new version first and widens it while error rates and latency stay healthy.

Use the concept in a decision

Put “What is deployment?” to work in a real decision. We can help relate the concept to the system you are building or buying. Tested, released and reversible, so changes reach customers on a Tuesday afternoon rather than late on a Friday night.

**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. Automated GIS SaaS deployment with GitHub Actions — deployment, data and reporting, all off the hands

**Automated GIS SaaS deployment with GitHub Actions** Get in touch →

## Why it matters

Deployment is the point where a change starts to help anyone, and also the point where a mistake starts to reach users. Teams that deploy rarely tend to collect many changes into each release, so every deploy carries more risk and, when it breaks, more candidates for the cause. Small, frequent deploys keep each step easy to understand and easy to reverse.

A deploy run by hand is only as consistent as the person running it. Steps get skipped, done in a different order or done slightly differently by a colleague covering a holiday. A pipeline performs the same steps every time and leaves a record of which commit, which artefact and which environment, with a timestamp. That record is where an incident review starts, and it is often the fastest way to answer what changed.

The strongest habit is to build once and promote the result. The artefact tested in staging should be byte for byte the one that reaches production, with only the configuration differing, and the environment supplies that configuration when

the artefact arrives. Recording the digest at build time and deploying by digest makes the promise checkable:

```
podman build -t registry.example.com/web:1.4.2 .
podman push --digestfile web.digest registry.example.com/web:1.4.2
kubectl -n staging set image deployment/web \
  web=registry.example.com/web@$(cat web.digest)
# after staging checks pass, the same digest goes to production
kubectl -n production set image deployment/web \
  web=registry.example.com/web@$(cat web.digest)
```

Rollback depends on the same discipline. When releases are immutable and identified, going back means pointing at the previous one, which takes seconds. When each environment rebuilds from source, going back means rebuilding an old commit and hoping its dependencies still resolve the same way.

## What it looks like in practice

Deployments of one platform to Azure were owned end to end, with every change travelling the same fixed route from development to staging to production, and no one pushing straight into a live environment. CI/CD builds the images and ships them, while Bicep templates keep the target infrastructure the same in each environment, so each promotion lands on matching infrastructure. Settings that differ per environment are kept apart from the secrets, which lets a single build travel through every stage and collect each environment's settings on arrival, with no rebuild along the way. Changes reach each environment along that defined path with no ad-hoc manual deploys, and releasing became a controlled step that runs the same way each time.

An earlier engagement shows what that consistency is worth in hours. Automating build, test and deployment saved roughly 4,000 hours by standardising the path from a change to production and removing manual steps that varied with whoever performed them. Releases became both faster and more reliable.

On the company's own platform, deploying is kept apart from provisioning. A container plane built on Podman and Quadlet has root prepare the deployment path once. After that, an unprivileged user redeploys by running a single fixed

script through a narrowly scoped rule that allows that one command with no arguments. The script builds the image and restarts the service, so a routine release needs no root shell and no arbitrary commands.

## Where it goes wrong

Most deployment trouble comes from a difference between what was checked and what actually changed.

**Production runs something staging never saw.** When each environment builds its own copy, dependency ranges resolve to newer versions, a base image moves between builds, or a build flag differs, and the artefact that passed testing is not the one serving customers. Build once, record the digest, promote that digest, and keep anything environment-specific in configuration rather than in the artefact.

**A tag moves, and a rollback lands on the wrong version.** A deploy that names `latest`, or reuses a version tag after a rebuild, depends on what the tag points at on the day it runs. The Kubernetes documentation advises against `:latest` in production because it becomes harder to tell which version is running and harder to roll back properly. Use a version tag that is never reused, and pin the digest where the platform allows.

**The pipeline reports success before the new version works.** Many deploy commands return as soon as the platform accepts the request, while the new instances are still starting and may crash a minute later. A step that waits for the new version to report ready, then sends a few real requests and checks the answers, turns a quiet failure into a red build while the previous release is still close at hand.

**New code and the old schema meet in the middle of a rollout.** During a rolling or canary deploy, old and new versions serve traffic side by side against one database. A migration that renames or drops a column breaks the old instances still running, and makes rolling back unsafe because the old code expects what was removed. Split such changes across deploys: add the new structure first, ship code that works with both, and remove the old structure only once no running version needs it.

# Further reading

- Build, release, run — the Twelve-Factor App's case for three separate stages and releases that are never changed once made.
- Open Container Initiative — the Linux Foundation project behind the image, runtime and distribution specifications that most container deployments rely on.
- OCI Image Format Specification — manifests, layers, configuration and the content digests that identify an image.
- Kubernetes Deployments — rolling updates, the recreate strategy, the progress deadline and rollbacks.
- Kubernetes images — tags, digests, pull policies and why `:latest` makes rollbacks harder.

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 deployment?" 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/deployment/