Engineer

Continuous delivery

keeping software ready to release safely on demand

What it is

Continuous delivery is the ability to release a software change safely when the team decides it is needed. Each change moves through a repeatable path that produces evidence about its readiness. Someone may still choose the production moment for operational, contractual or product reasons. That decision works when the tested candidate and the way to promote it are already available.

Continuous integration is the practice of integrating small changes frequently and checking the shared code. It can tell a team that a change builds and passes its tests. Continuous delivery also asks whether the resulting artifact can be installed in the intended environments, whether its configuration is understood, and how operators will tell that it is working. Continuous deployment takes the additional step of automatically deploying qualifying changes to production. A team that keeps an explicit production approval can still practice continuous delivery if approval is a decision about a ready change.

Deployment and release can also be separate events. An application may run new code in production behind a disabled feature flag. An operator can later expose the feature to a small group or enable it for everyone. The team still needs evidence that the code is safe to deploy in its dormant state, and that the later change of exposure has its own checks and owner.

Use the concept in a decision

Put “What is continuous delivery?” 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

A release that requires specialists to remember a sequence introduces a fresh failure opportunity every time. Put the repeatable steps in a workflow and publish what each step verified. An engineer can then distinguish a failed test from an unavailable environment or an approval that is still outstanding. The audit trail can answer which artifact was installed without relying on a message in chat or the mutable label `latest`.

Release readiness also makes it easier to ship small changes. If production happens rarely because the work is cumbersome, unrelated changes gather behind one decision. When something goes wrong, the investigation must consider the entire batch. A routine path lets the team ship smaller units and observe what each one changed. The aim is a decision the team can make with evidence, including the decision to wait.

The release path includes more than one repository. A service may depend on database migrations, runtime configuration and identity settings maintained elsewhere. A tested executable gives limited assurance when the environment changes independently. Make those dependencies visible, document which combinations work, and test representative configurations in an environment close enough to production to reveal relevant failures. A safe deployment can then be reproduced when a second person is on call.

What it looks like in practice

The portfolio case about building CI/CD workflows with environment promotion describes separate GitHub Actions workflows for quality checks, tests and deployment to distinct environments. It also records different image bases for the frontend, API and database, chosen for their workloads. The defensible lesson is that a route through environments and visible checks made releases routine in that project. The case gives no reason to assume that another project uses the same image policy or production approval.

Imagine a team adding a read-only export endpoint. The workflow records the source revision, runs tests and produces a uniquely identified package. It deploys that package to staging, where a test downloads an export from a realistic dataset and checks both its columns and access rules. The approver sees the package identifier and evidence that staging ran it. Production uses the same package with production configuration. A smoke test confirms that an authorized request succeeds and that the existing endpoint still works.

Give each step an outcome its next consumer can use. If staging fails, leave the candidate visible with the exact reason and stop promotion. If production fails, show which package was active before and which health signal changed. A team can rehearse the sequence against a permitted test service, including a stopped dependency, while release traffic stays untouched. For a database change, the rehearsal verifies that the previous application can still run against the intermediate schema during recovery.

Where it goes wrong

Rebuilding at every stage blurs which bytes passed the earlier tests. Even the same commit can resolve a changed dependency or use a different compiler image. Promote a recorded immutable artifact when the environments can consume it. If platform differences require a separate build, record each output and require the evidence appropriate to that target. A familiar version string alone establishes neither identity nor compatibility.

Approvals can become a second system of delivery. Treat a manual gate as an explicit decision by someone who can see the evidence, the target and any outstanding risk. If the approver has to repeat every manual command, the automation ends before the last important stage. Agree which observations need human judgment and make the usual steps executable by another authorized operator.

A green run can also be misleading if a flaky test passes on retry or an alert arrives after the whole audience is exposed. Investigate intermittent tests and distinguish a reliable health check from one that simply waits. Use a planned observation window for risky changes, with a clear stop condition and a record of who can act. When readiness breaks, make that interruption visible. The next release should resume from a known candidate and a verified state.

Further reading

- DORA's continuous delivery capability defines release readiness and explains how it differs from continuous deployment.
- DORA's continuous integration capability describes frequent integration and fast feedback on the shared code.
- DORA's deployment automation capability covers the repeatable steps used to deploy to different environments.
- GitHub Actions deployment controls documents environments, protection rules and deployment jobs.

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 continuous delivery?" 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/continuous-delivery/