Continuous delivery

Continuous delivery (CD) is the practice of using automation to release software in short iterations.

What is continuous delivery?

Continuous delivery is an approach where teams release quality products frequently and predictably from source code repository to production in an automated fashion.

READ ON BELOW

CI/CD topics

ARTICLE

Continuous delivery

The key principles of continuous delivery include repeatable/reliable processes, automation, version control, built-in quality metrics, and more.

ARTICLE

CD pipelines

A continuous delivery pipeline orchestrates automated builds, tests, and deployments into one release workflow.

ARTICLE

Continuous integration

Continuous integration is the practice of routinely integrating code changes into the main branch of a repository, and testing the changes early and often.

ARTICLE

Software testing

Continuous delivery leverages a battery of software testing strategies to create a seamless pipeline that automatically delivers completed code tasks.

ARTICLE

Continuous deployment

In continuous deployment, production release is automatic for changes that pass the test suite.

ARTICLE

Microservices

The guiding principle of microservices is to build applications by splitting business components into services that can be deployed and operated independently.

Featured tutorials

TUTORIAL

Continuous delivery tutorial

In this guide, we'll see how you can use Bitbucket Pipelines to adopt a continuous delivery workflow. Read on!

Try this tutorial →

TUTORIAL

Continuous integration tutorial

This tutorial will show you how to get started with continuous integration in three simple steps.

Try this tutorial →

TUTORIAL

Integration testing tutorial

Learn how to run integration tests in this tutorial with Bitbucket Pipelines.

Try this tutorial →

[CONTINUED]

Some organizations release products manually by handing it off from one team to the next. See the diagram below. Typically, developers are at the left end of this spectrum and operations personnel are at the receiving end. There is delay at every hand-off that leads to frustrated teams and dissatisfied customers. The product eventually goes live through a tedious and error-prone process

that delays revenue generation.

Figure 1: Manual release of products to customers

Now check out the continuous delivery pipeline below. It illustrates how developers write code on their laptops, and commit those changes to a source code repository, like Bitbucket. By code, we mean the system under test, the tests, and infrastructure that will be used to deploy and maintain the system. Bitbucket Pipelines can ship the product from test to staging to production, and help customers get their hands on those shiny new features.

Figure 2: Continuous delivery pipeline doing automated releases

A continuous delivery pipeline could have a manual gate right before production. A manual gate requires human intervention, and there could be scenarios in your organization that require manual gates in pipelines. Some manual gates might be questionable, whereas some could be legitimate. One legitimate scenario allows the business team to make a last-minute release decision. The engineering team keeps a shippable version of the product ready after every sprint, and the business team makes the final call to release the product to all customers, or a cross-section of the population, or perhaps to people who live in a certain geographical location.

The architecture of the product that flows through the pipeline is a key factor that determines the anatomy of the continuous delivery pipeline. A highly coupled product architecture generates a complicated graphical pipeline pattern where various pipelines could get entangled before eventually making it to production.

The product architecture also influences the different phases of the pipeline and what artifacts are produced in each phase. The pipeline first builds components - the smallest distributable and testable units of the product. For example, a library built by the pipeline can be termed a component. This is the component phase.

Loosely coupled components make up subsystems - the smallest deployable and runnable units. For example, a server is a subsystem. A microservice running in a container is also an example of a subsystem. This is the subsystem phase. As opposed to components, subsystems can be stood up and tested.

Thereon, the pipeline could be taught to assemble a system from loosely coupled subsystems in cases where the entire system should be released as a whole. This is the system phase. We recommend against this composition where subsystems are assembled into a system. See an illustration of this in Figure 3.

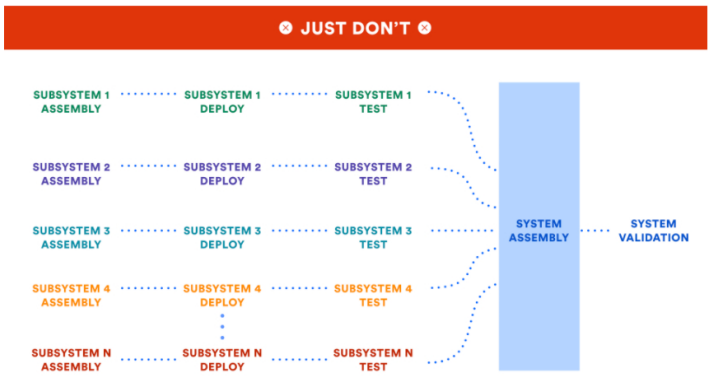

Figure 3: Subsystems having to be assembled into a system

This all-or-none approach causes the fastest subsystem to go at the speed of the slowest one. “The chain is only as strong as its weakest link” is a cliche we use to warn teams who fall prey to this architectural pattern.

Once validated, the assembled system is then promoted to production without any further modification, in the final phase, called production phase.

Note that these phases are more logical than physical, and created only to break down a large problem into multiple smaller sub-problems. You may have less phases or more, depending on your architecture and requirements.

Speed without quality is useless to our customers. Continuous testing is a technique where automated tests are integrated with the software delivery pipeline, and validate every change that flows through it. Tests execute in each phase of the pipeline to validate artifacts produced in that phase. Unit tests and static code analysis validate components in the component phase of the pipeline. Functional, performance and security tests validate subsystems in the subsystem phase. Integration, performance, and security tests validate systems in the system phase. Finally, smoke tests validate the product in the production phase.

Figure 4: Automated tests integrate with the pipeline

A monolithic product architecture, or a Big Ball of Mud, can result in a Big Ball of Tests. We recommend investing into microservices such that independently deployable artifacts can flow through pipelines without needing a highly integrated environment for certification. Also, independently deployable artifacts enable faster teams to not get bogged down by slower teams.

Value of continuous delivery

The software delivery pipeline is a product in its own right, and is a priority for business, otherwise we should not send our revenue-generating products through it.

Continuous delivery adds value in three ways. It improves velocity, productivity, and sustainability of software development teams.

Velocity

Velocity means responsible speed, and not suicidal speed. Pipelines are meant to ship quality products to our customers. Unless teams are disciplined, pipelines can shoot faulty code to Production, only faster! Automated software delivery pipelines help organizations respond to market changes better.

Productivity

A spike in productivity results when tedious tasks, like submitting a change request for every change that goes to Production, can be performed by pipelines instead of humans. This lets scrum teams focus on products that wow the world, instead of draining their energy on logistics. And that can make team members happier, more engaged in their work, and want to stay on the team longer.

Sustainability

Sustainability is key for all businesses, not just tech. “Software is eating the world” is no longer true – software has already consumed the world! Every company at the end of the day, whether in healthcare, finance, retail, or some other domain, uses technology to differentiate, and to outmaneuver their competition. Automation helps reduce/eliminate manual tasks that are error-prone and repetitive, thus positioning the business to innovate better and faster to meet their customers' needs.

When and who should do CD?

When is a good time to adopt continuous delivery? It was yesterday.

Teams need a single prioritized backlog where:

1. Continuous delivery has been embraced, instead of being relegated to the background
2. The acceptance criteria of user stories explicitly mention automated software delivery approaches instead of manual, and
3. The sprint DoD (Definition of Done) prevents teams from finishing sprints where the product shipped manually.