Continuous delivery pipeline 101

Learn how automated builds, tests and deployments are chained together in one release workflow.

What is a continuous delivery pipeline?

How does a pipeline relate to continuous delivery (CD)? As the name suggests, a continuous delivery pipeline is an implementation of the continuous paradigm, where automated builds, tests and deployments are orchestrated as one release workflow. Put more plainly, a CD pipeline is a set of steps your code changes will go through to make their way to production.

A CD pipeline delivers, as per business needs, quality products frequently and predictably from test to staging to production in an automated fashion.

For starters, let's focus on the three concepts: quality, frequently, and predictably.

We emphasize on quality to underscore that it's not traded off for speed. Business doesn't want us to build a pipeline that can shoot faulty code to production at high speed. We will go through the principles of "Shift Left" and "DevSecOps", and discuss how we can move quality and security upstream in the SDLC (software development life cycle). This will put to rest any concerns regarding continuous delivery pipelines posing risk to businesses.

Frequently indicates that pipelines execute at any time to release features, since they are programmed to trigger with commits to the codebase. Once the pipeline MVP (minimum viable product) is in place, it can execute as many times as it needs to with periodic maintenance cost. This automated approach scales without burning out the team. This also allows teams to make small incremental improvements to their products without the fear of a major catastrophe in production.

Cliche as it may sound, the nation of "to err is human" still holds true. Teams brace for impact during manual releases since those processes are brittle. Predictably implies that releases are deterministic in nature when done via continuous delivery pipelines. Since pipelines are programmable infrastructure, teams can expect the desired behavior every time. Accidents can happen, of course, since no software is bug-free. However, pipelines are exponentially better than manual error-prone release processes, since unlike humans, pipelines don't falter under aggressive deadlines.

Pipelines have software gates that automatically promote or reject versioned artifacts from passing through. If the release protocol is not honored, software gates remain closed, and the pipeline aborts. Alerts are generated and notifications are sent to a distribution list comprising team members who could have potentially broken the pipeline.

And that's how a CD pipeline works: A commit, or a small incremental batch of commits, makes its way to production every time the pipeline runs successfully. Eventually teams ship features - and ultimately - products in a secure and auditable way.

READ ON BELOW

Continuous delivery pipeline articles

ARTICLE

Deploying from branches

Learn how a branch-per-issue model fits with continuous delivery workflows including SaaS products, installed or mobile products, and Gitflow

ARTICLE

Continuous delivery with feature branches

Learn how to do continuous delivery with a Gitflow or feature branch workflow.

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 →

[CONTINUED]

Phases in a continuous delivery pipeline

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 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. Let's discuss the four common phases in continuous delivery:

1. Component phase
2. Subsystem phase
3. System phase
4. Production phase

Even if you foresee more than four phases or less than four in your organization, the concepts outlined below still apply.

A common misconception is that these phases have physical manifestations in your pipeline. They don't have to. These are logical phases, and can map to environmental milestones like test, staging, and production. For example, components and subsystems could be built, tested, and deployed in test. Subsystems or systems could be assembled, tested, and deployed in staging. Subsystems or systems could be promoted to production as part of production phase.

The cost of defects is low when discovered in test, medium when discovered in staging, and high in production. "**Shift Left**" refers to validations being pulled earlier in the pipeline. The gate from test to staging has far more defensive techniques built in nowadays, and hence staging doesn't have to look like a crime scene any more!

Historically, InfoSec came in at the fag end of the SDLC (software development life cycle) and rejected releases that could pose cyber-security threats to the company. While their intentions are noble, they caused frustration and delay. "**DevSecOps**" advocates security to be built into products from the design phase, instead of sending a (possibly insecure) finished product for evaluation.

Let's take a closer look into how "Shift Left" and "DevSecOps" can be addressed within the continuous delivery workflow. In these next sections, we will discuss each phase in detail.

CD Component 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. A component can be certified, among other things, by code reviews, unit tests, and static code analyzers.

Code reviews are important for teams to have a shared understanding of the features, tests, and infrastructure needed for the product to go live. A second pair of eyes can often do wonders. Over the years we may get immune to bad code in a way that we don't believe it's bad any more. Fresh perspectives can force us to revisit those weaknesses and refactor generously wherever needed.

Unit tests are almost always the first set of software tests that we run on our code. They do not touch the database or the network. Code coverage is the percentage of code that has been touched by unit tests. There are many ways to measure coverage, like line coverage, class coverage, method coverage etc.

While it is great to have good code coverage to ease refactoring, it is detrimental to mandate high coverage goals. Contrary to intuition, some teams with high code coverage have more production outages than teams with lower code coverage. Also, keep in mind that it is easy to game coverage numbers. Under acute pressure, especially during performance reviews, developers can revert to unfair practices to increase code coverage. And I won't be covering those details here!

Static code analysis detects problems in code without executing it. This is an inexpensive way to detect issues. Like unit tests, these tests run on source code and have low run-time. Static analyzers detect potential memory leaks, along with code quality indicators like cyclomatic complexity and code duplication. During this phase, SAST (static analysis security testing) is a proven way to discover security vulnerabilities.

Define the metrics that control your software gates, and influence code promotion from component phase to the subsystem phase.

CD Subsystem 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. As opposed to components, subsystems can be stood up and validated against customer use cases.

Just like a Node.js UI and a Java API layer are subsystems, databases are subsystems too. In some organizations, RDBMS (relational database management systems) is manually handled, even though a new generation of tools have surfaced that automate database change management and successfully do continuous delivery of databases. CD pipelines involving NoSQL databases are easier to implement than RDBMS.

Subsystems can be deployed and certified by functional, performance, and security tests. Let's study how each of these test types validate the product.

Functional tests include all customer use cases that involve internationalization (I18N), localization (L10N), data quality, accessibility, negative scenarios etc. These tests make sure that your product functions per customer expectations, honors inclusion, and serves the market it's built for.

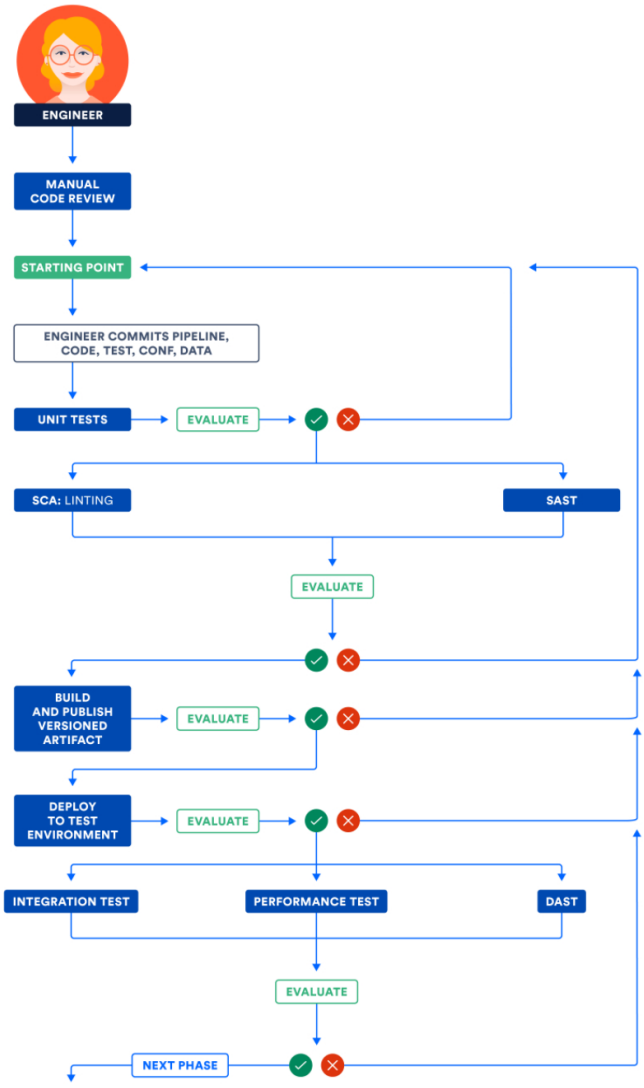
Determine your **performance** benchmarks with your product owners. Integrate your performance tests with the pipeline, and use the benchmarks to pass or fail pipelines. A common myth is that performance tests do not need to integrate with continuous delivery pipelines, however, that breaks

the continuous paradigm.

Major organizations have been breached in recent times, and cybersecurity threats are at their highest. We need to buckle up and make sure that there are no **security** vulnerabilities in our products - be that in the code we write or be that in 3rd-party libraries that we import into our code. In fact, major breaches have been discovered in OSS (open source software) and we should use tools and techniques that flag these errors and force the pipeline to abort. DAST (dynamic analysis security testing) is a proven way to discover security vulnerabilities.

The following illustration articulates the workflow discussed in the Component and Subsystem phases. Run independent steps in parallel to optimize the total pipeline execution time and get fast feedback.

A) CERTIFYING COMPONENTS AND/OR SUBSYSTEMS IN THE TEST ENVIRONMENT



CD System phase

Once subsystems meet functional, performance, and security expectations, the pipeline could be taught to assemble a system from loosely coupled subsystems in cases where the entire system has to be released as a whole. What that means is that the fastest team can go at the speed of the slowest team. Reminds me of the old saying, "A chain is only as strong as its weakest link".

We recommend against this composition anti-pattern where subsystems are composed into a system to be released as a whole. This anti-pattern ties all the subsystems at their hips for success. If you invest in independently deployable artifacts, you will be able to avoid this anti-pattern.

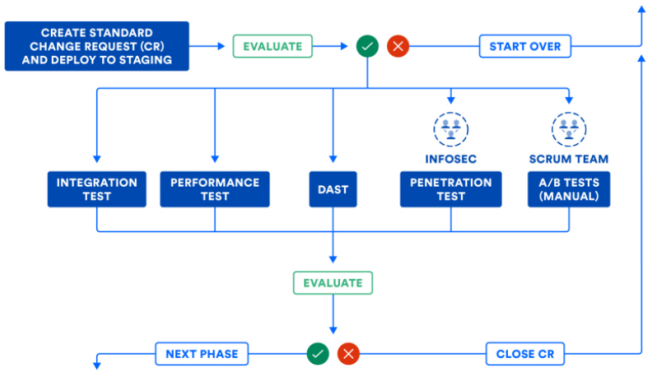
Where systems need to be validated as a whole, they can be certified by **integration**, **performance**, and **security** tests. Unlike subsystem phase, do not use mocks or stubs during testing in this phase. Also, focus on testing interfaces and network more than anything else.

The following illustration summarizes the workflow in the System phase, in case you have to assemble your subsystems using composition. Even if you can roll your subsystems to production, the following illustration helps establish software gates needed to promote code from this phase to the next.

The pipeline can automatically file change requests (CR) to leave an audit trail. Most organizations use this workflow for standard changes, which means, planned releases. This workflow should also be used for emergency changes, or unplanned releases, although some teams tend to cut corners. Note how the change request (CR) is closed automatically by the CD pipeline when errors force it to abort. This prevents CRs from being abandoned in the middle of the pipeline workflow.

The following illustration articulates the workflow discussed in the CD System phase. Note that some steps could involve human intervention, and these manual steps can be executed as part of manual gates in the pipeline. When mapped in its entirety, the pipeline visualization is a close resemblance to the value stream map of your product releases!

B) CERTIFYING SUBSYSTEMS AND/OR SYSTEM IN THE STAGING ENVIRONMENT



Once the assembled system is certified, leave the assembly unchanged and promote it to production.

CD Production phase

Whether subsystems can be independently deployed, or they need to be assembled into a system, those versioned artifacts are deployed to production as part of this final phase.

ZDD (zero downtime deployment) is a must to prevent downtime for customers, and should be practiced all the way from test to staging to production. Blue-green deployment is a popular ZDD technique where the new bits are deployed to a tiny cross-section of the population (called “green”), while the bulk is blissfully unaware on “blue” which has the old bits. If push comes to shove, revert everyone back to “blue” and very few customers will be affected, if any. If things look good on “green”, dial everyone up slowly from “blue” to “green”.

Some organizations require a manual gate (or two) before the pipeline deploys to production. There are scenarios where manual gates are legitimate, like, business might want to target a certain geographic or demographic cross-section of the population and gather data before releasing to the world.

However, I see manual gates being abused in certain organizations. They require teams to get manual approval