Learn about CI/CD

By

Stanley Stephen

Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Continuous Integration vs. Continuous Delivery vs. Continuous Deployment

We will see in this article what these three practices mean and what's required to use them.

CI and CD are two acronyms that are often mentioned when people talk about modern development practices. CI is straightforward and stands for continuous integration, a practice that focuses on making preparing a release easier. But CD can either mean continuous delivery or continuous deployment, and while those two practices have a lot in common, they also have a significant difference that can have critical consequences for a business.

We will see in this article what these three practices mean and what's required to use them.

What are the differences between continuous integration, continuous delivery, and continuous deployment?

Continuous Integration

Developers practicing continuous integration merge their changes back to the main branch as often as possible. The developer's changes are validated by creating a build and running automated tests against the build. By doing so, you avoid the integration hell that usually happens when people wait for release day to merge their changes into the release branch.

Continuous integration puts a great emphasis on testing automation to check that the application is not broken whenever new commits are integrated into the main branch.

Continuous Delivery

Continuous Delivery is an extension of continuous integration to make sure that you can release new changes to your customers quickly in a sustainable way. This means that on top of having automated your testing, you also have automated your release process and you can deploy your application at any point of time by clicking on a button.
In theory, with continuous delivery, you can decide to release daily, weekly, fortnightly, or whatever suits your business requirements. However, if you truly want to get the benefits of continuous delivery, you should deploy to production

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

as early as possible to make sure that you release small batches that are easy to troubleshoot in case of a problem.

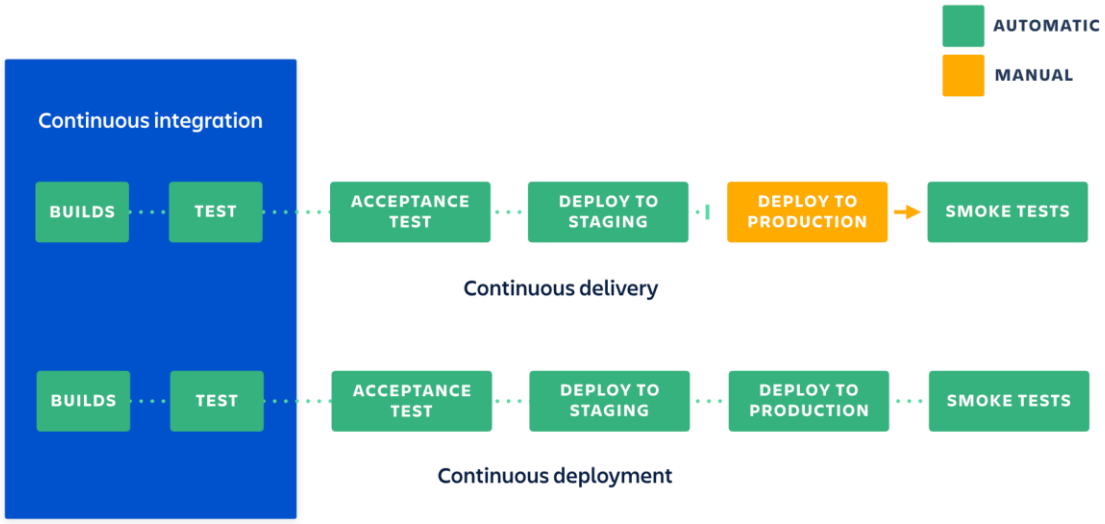
Continuous Deployment

Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a *Release Day* anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.

How the practices relate to each other

To put it simply continuous integration is part of both continuous delivery and continuous deployment. And continuous deployment is like continuous delivery, except that releases happen automatically.



What are the benefits of each practice?

We've explained the difference between continuous integration, continuous delivery, and continuous deployments but we haven't yet looked into the reasons why you would adopt them. There's an obvious cost to implementing each practice, but it's largely outweighed by their benefits.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Continuous Integration

What you need (cost)

- Your team will need to write automated tests for each new feature, improvement or bug fix.
- You need a continuous integration server that can monitor the main repository and run the tests automatically for every new commits pushed.
- Developers need to merge their changes as often as possible, at least once a day.

What you gain

- Less bugs get shipped to production as regressions are captured early by the automated tests.
- Building the release is easy as all integration issues have been solved early.
- Less context switching as developers are alerted as soon as they break the build and can work on fixing it before they move to another task.
- Testing costs are reduced drastically – your CI server can run hundreds of tests in the matter of seconds.
- Your QA team spend less time testing and can focus on significant improvements to the quality culture.

Continuous Delivery

What you need (cost)

- You need a strong foundation in continuous integration and your test suite needs to cover enough of your codebase.
- Deployments need to be automated. The trigger is still manual but once a deployment is started there shouldn't be a need for human intervention.
- Your team will most likely need to embrace feature flags so that incomplete features do not affect customers in production.

What you gain

- The complexity of deploying software has been taken away. Your team doesn't have to spend days preparing for a release anymore.
- You can release more often, thus accelerating the feedback loop with your customers.
- There is much less pressure on decisions for small changes, hence encouraging iterating faster.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Continuous Deployment

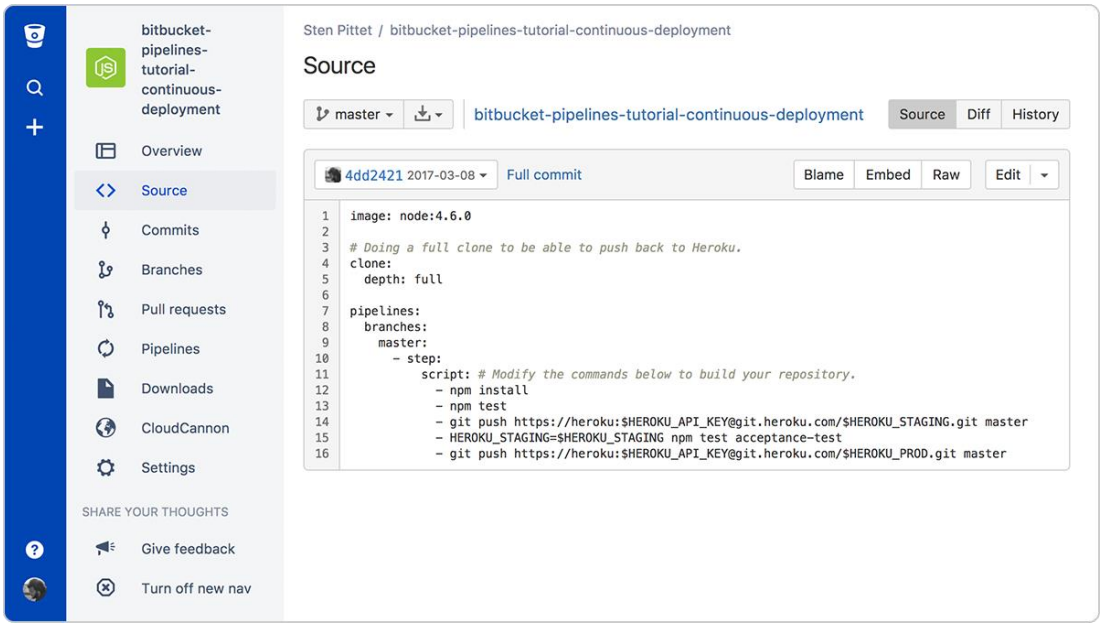
What you need (cost)

- Your testing culture needs to be at its best. The quality of your test suite will determine the quality of your releases.
- Your documentation process will need to keep up with the pace of deployments.
- Feature flags become an inherent part of the process of releasing significant changes to make sure you can coordinate with other departments (Support, Marketing, PR...).

What you gain

- You can develop faster as there's no need to pause development for releases. Deployments pipelines are triggered automatically for every change.
- Releases are less risky and easier to fix in case of problem as you deploy small batches of changes.
- Customers see a continuous stream of improvements, and quality increases every day, instead of every month, quarter or year.

One of the traditional cost associated with continuous integration is the installation and maintenance of a CI server. But you can reduce significantly the cost of adopting these practices by using a cloud service like Bitbucket Pipelines which adds automation to every Bitbucket repository. By simply adding a configuration file at the root of your repository you will be able to create a continuous deployment pipeline that gets executed for every new change pushed to the main branch.



Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Going from Continuous Integration to Continuous Deployment

If you're just getting started on a new project with no users yet, it might be easy for you to deploy every commit to production. You could even start by automating your deployments and release your alpha version to a production with no customers. Then you would ramp up your testing culture and make sure that you increase code coverage as you build your application. By the time you're ready to on-board users, you will have a great continuous deployment process where all new changes are tested before being automatically released to production.

But if you already have an existing application with customers you should slow things down and start with continuous integration and continuous delivery. Start by implementing basic unit tests that get executed automatically, no need to focus yet on having complex end-to-end tests running. Instead, you should try automating your deployments as soon as possible and get to a stage where deployments to your staging environments are done automatically. The reason is that by having automatic deployments, you will be able to focus your energy on improving your tests rather than having periodically to stop things to coordinate a release.

Once you can start releasing software on a daily basis, you can look into continuous deployment, but make sure that the rest of your organization is ready as well. Documentation, support, marketing. These functions will need to adapt to the new cadence of releases, and it is important that they do not miss on significant changes that can impact customers.

Read below guides

You can find some guides below that will go more in depth to help you getting started with these practices.

- Getting started with continuous integration
- Getting started with continuous delivery
- Getting started with continuous deployment

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

How to get started with Continuous Integration

Learn about how to adopt continuous integration and automated testing in 5 steps.

Continuous integration (CI) is a practice where a team of developers integrate their code early and often to the main branch or code repository. The goal is to reduce the risk of seeing "integration hell" by waiting for the end of a project or a sprint to merge the work of all developers.

One of the primary benefits of adopting CI is that it will save you time during your development cycle by identifying and addressing conflicts early. It's also a great way to reduce the amount of time spent on fixing bugs and regression by putting more emphasis on having a good test suite. Finally, it helps share a better understanding of the codebase and the features that you're developing for your customers.

The first step on your journey to continuous integration: setting up automated testing.

Getting started with automated testing

Understanding the different types of tests

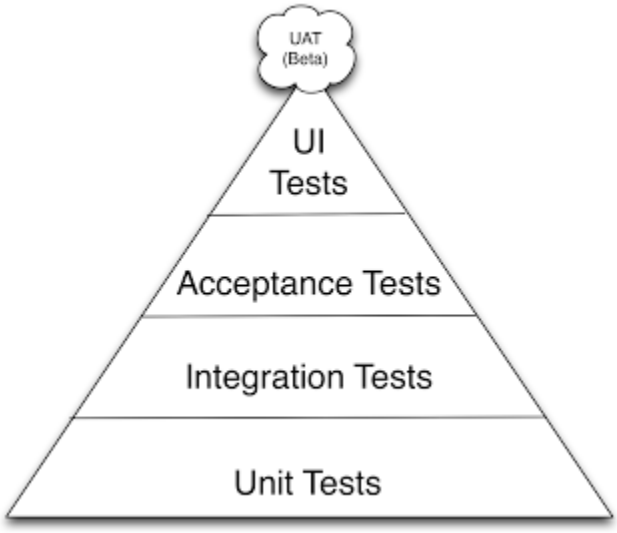
To get the full benefits of CI, you will need to automate your tests to be able to run them for every change that is made to the main repository. We insist on running tests on every branch of your repository and not just focus on the main branch. This way you will be able to capture issues early and minimise disruptions for your team.

There are many types of tests implemented, but it is not necessary to do everything at once if you're just getting started. You can start small with unit tests and work on extending your coverage over time.

- **Unit tests** are narrow in scope and typically verify the behaviour of individual methods or functions.
- **Integration tests** make sure that multiple components behave correctly together. This can involve several classes as well as testing the integration with other services.
- **Acceptance tests** are similar to the integration tests but they focus on the business cases rather than the components themselves.
- **UI tests** will make sure that the application functions correctly from a user perspective.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Not all the tests are equal, and you can visualize the tradeoffs that you will make with the test pyramid developed by Mike Cohn.



Unit tests are fast and cheap to implement as they're mostly doing checks on small pieces of code. On the other hand UI tests will be complex to implement and slow to run as they often require to get a full environment started as well as multiple services to emulate browser or mobile behaviours. Because of that, you may want to limit the number of complex UI tests and rely on good Unit testing at the base to have a fast build and get feedback to developers as soon as possible.

Running your tests automatically

To adopt continuous integration, you will need to run your tests on every change that gets pushed back to the main branch. To do so, you will need to have a service that can monitor your repository and listen to new pushes to the codebase. There are many solutions that you can choose from both on-premise and in the Cloud. You'll need to consider the following to pick your server:

- **Where is your code hosted?** Can the CI service access your codebase? Do you have a special restriction on where the code can live?
- **What OS and resources do you need for your application?** Is your application environment supported? Can you install the right dependencies to build and test your software?
- **How much resource do you need for your tests?** Some Cloud applications might have restrictions on the resources that you can use. If your software consumes a lot of resources, you might want to host your CI server behind your firewall.
- **How many developers are on your team?** When your team practice CI you will have many changes pushed back to the main repository every day. For the developers to get the fast feedback, you need to reduce the amount of queue time for the builds, and you will want to use a service or server that gives you the right concurrency.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

In the past, you typically had to install a separate CI server like Bamboo or Jenkins, but now you can find solutions on the Cloud that are much simpler to adopt. For instance, if your code is hosted on Bitbucket Cloud you can use the Pipelines feature in your repository to run tests on every push without the need to configure a separate server or build agents, and with no restriction on concurrency.

```
image: node:4.6.0 pipelines:    default:      -
 step:          script:            - npm install             -
 npm test
```

Example of configuration to test a Javascript repository with Bitbucket Pipelines.

Use code coverage to find untested code

Once you adopt automated testing it is a good idea to couple it with a test coverage tool that will give you an idea of how much of your codebase is covered by your test suite.

It is good to aim a coverage above 80% but be careful not to confuse high percentage of coverage with a good test suite. A code coverage tool will help you find untested code but it is the quality of your tests that will make the difference at the end of the day.

If you're just getting started don't rush in attaining 100% coverage of your codebase, but rather than that use a test coverage tool to find out critical parts of your application, that do not yet have tests and start there.

Refactoring is an opportunity to add tests

If you are about to make significant changes to your application you should start by writing acceptance tests around the features that may be impacted. This will provide you with a safety net to ensure that the original behaviour has not been affected after you've refactored code or added new features.

Adopting continuous integration

While automating your tests is a key part of CI it is not enough by itself. You may need to change your team culture to make sure that developers do not work days on a feature without merging their changes back to the main branch and you'll need to enforce a culture of a green build

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Integrate early and often

Whether you're using trunk-based development or feature branches, it is important that developers integrate their changes as soon as possible on the main repository. If you let the code sitting on a branch or the developer workstation for too long, then you expose yourself to the risk of having too many conflicts to look at when you decide to merge things back to the main branch.

By integrating early, you reduce the scope of the changes which makes it easier to understand conflicts when you have them. The other advantage is to make it easier to share knowledge among developers as they will get more digestible changes.

If you find yourself making some changes that can impact an existing feature you can use feature flags to turn off your changes in production until your work is completed.

Keep the build green at all time

If a developer breaks the build for the main branch, fixing it becomes the main priority. The more changes get into the build while it's broken, the harder it will be for you to understand what broke it - and you also have the risk of introducing more failures.

It is worth spending time on your test suite to make sure that it can fail fast and give feedback to the developer that pushed the changes as soon as possible. You can split your tests so that the fast ones (Unit Tests for example) run before the long-running tests. If your test suite always takes a long time to fail, you will waste a lot of developer time as they'll have to switch context to go back to their previous work and fix it.

Don't forget to set notifications to make sure that developers are alerted when the build breaks, and you can also go a step further by displaying the state of your main branches on a dashboard where everyone can see it.

Write tests as part of your stories

Finally, you'll need to make sure that every feature that gets developed has automated tests. It may look like you will slow down development but in fact, this is going to reduce drastically the amount of time that your team spends on fixing regression or bugs introduced in every iteration. You will also be able to make

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

changes to your codebase with confidence as your test suite will be able to rapidly make sure that all the previously developed features work as expected.

To write good tests, you will need to make sure that developers are involved early in the definition of the user stories. This is an excellent way to get a better-shared understanding of the business requirements and facilitate the relationship with product managers. You can even start by writing the tests before implementing the code that will fulfill them.

Write tests when fixing bugs

Whether you have an existing codebase or you just getting started, it is certain that you will have bugs occurring as part of your releases. Make sure that you add tests when you solve them to prevent them from occurring again.

CI will enable your QA Engineers to scale quality

Another role that will change with the adoption of CI and automation is the role of the QA Engineer. They no longer need to test manually trivial capabilities of your application and they can now dedicate more time to providing tools to support developers as well as help them adopt the right testing strategies.

Once you start adopting continuous integration, your QA Engineers will be able to focus on facilitating testing with better tooling and datasets as well as help developers grow in their ability to write better code. There will still be some exploratory testing for complex use cases, but this should be a less prominent part of their job.

Continuous integration in 5 steps

You should now have a good idea of the concepts behind continuous integration, and we can boil it down to this:

1. Start writing tests for the critical parts of your codebase.
2. Get a CI service to run those tests automatically on every push to the main repository.
3. Make sure that your team integrates their changes everyday.
4. Fix the build as soon as it's broken.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

5. Write tests for every new story that you implement.

While it may look easy, it will require true commitment from your team to be effective. You will need to slow down your releases at the beginning, and you need buy-in from the product owners to make sure that they do not rush developers in shipping features without tests.

Our recommendation is to start small with simple tests to get used to the new routine before moving on to implementing a more complex test suite that may be hard to manage.

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

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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 behaviour 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 honoured, 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.

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:

- Component phase
- Subsystem phase
- System phase
- Production phase

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

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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 anymore!

Historically, InfoSec came in at the far 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.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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, honours 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

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

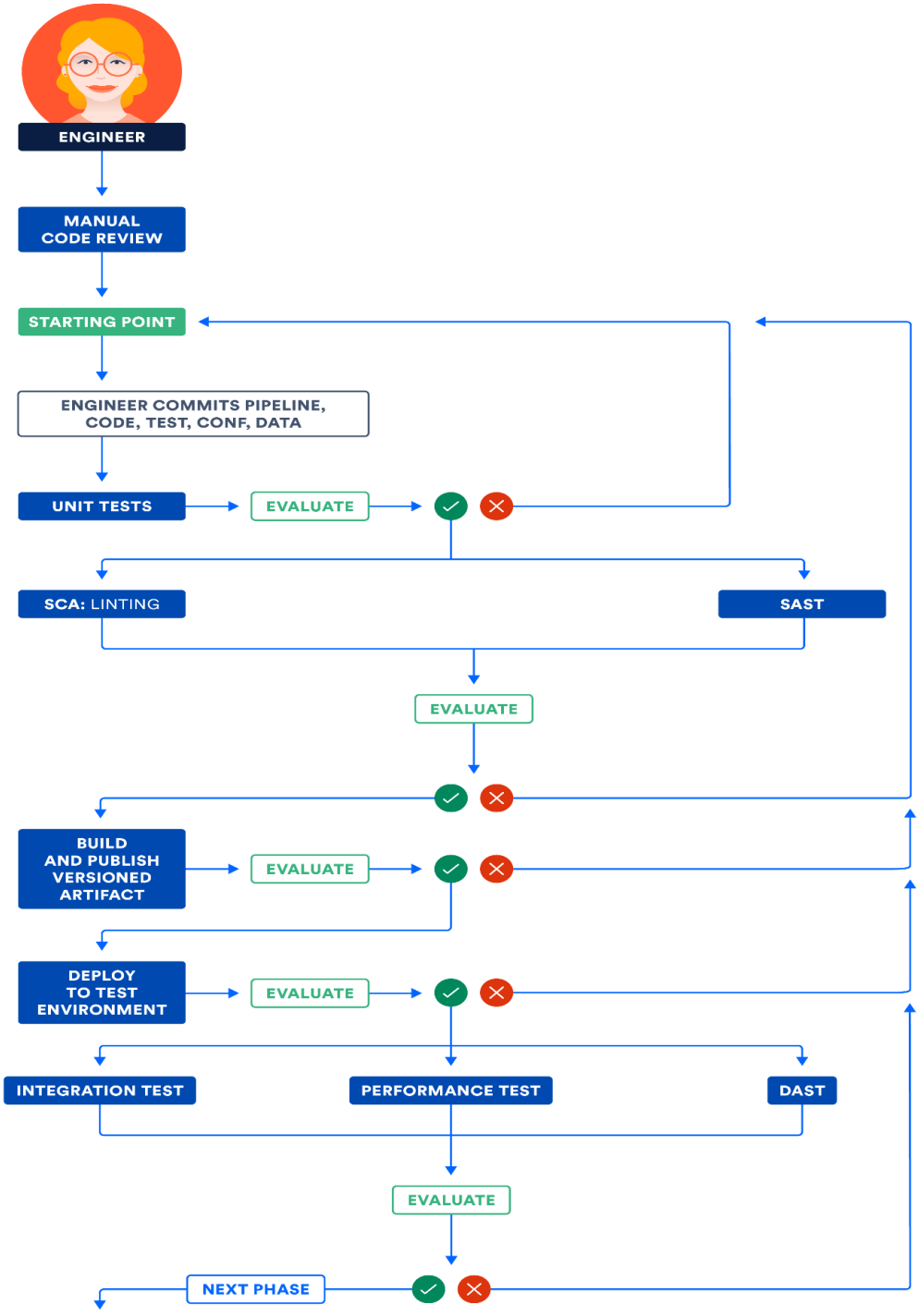
pipelines. A common myth is that performance tests do not need to integrate with continuous delivery pipelines, however, that breaks the continuous paradigm.

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.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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



Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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.

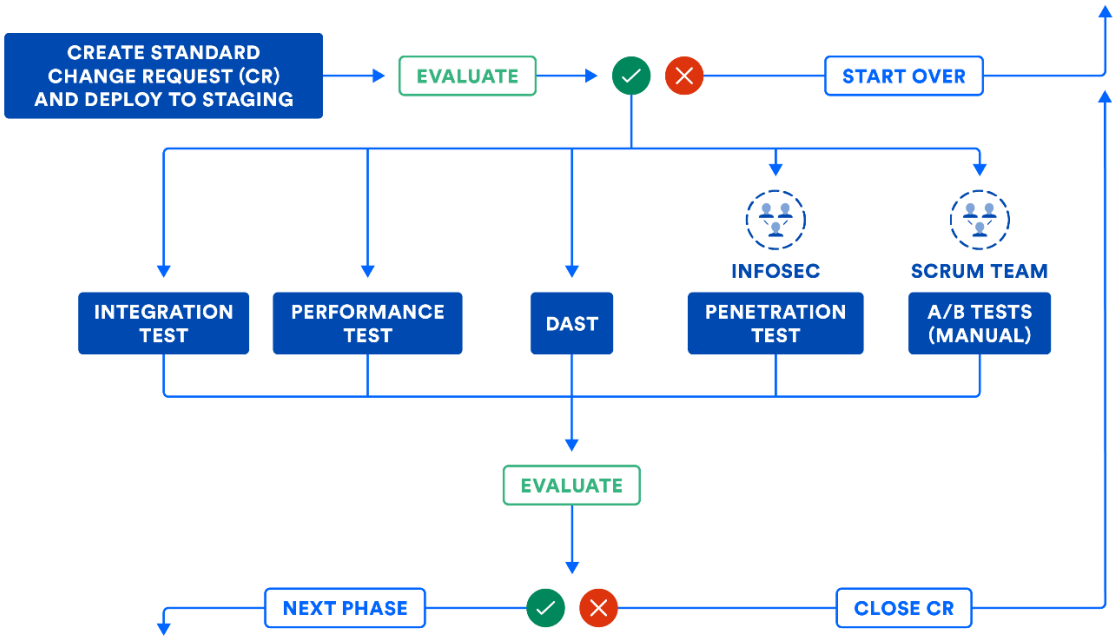
Where systems need to be validated as a whole, they can be certified by **integration**, **performance**, and **security** tests. Unlike subsystem phase, do not use mocks or stubs during testing in this phase. Also, focus on testing interfaces and network more than anything else.
The following illustration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

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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".

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

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 in a CAB (Change Approval Board) meeting. The reason, is more often than not, a mis-interpretation of segregation of duties or separation of concerns, and one department hands off to another seeking approval to move forward. I have also seen some CAB approvers demonstrate a shallow technical understanding of the changes going to production, hence making the manual approval process slow and dreary.

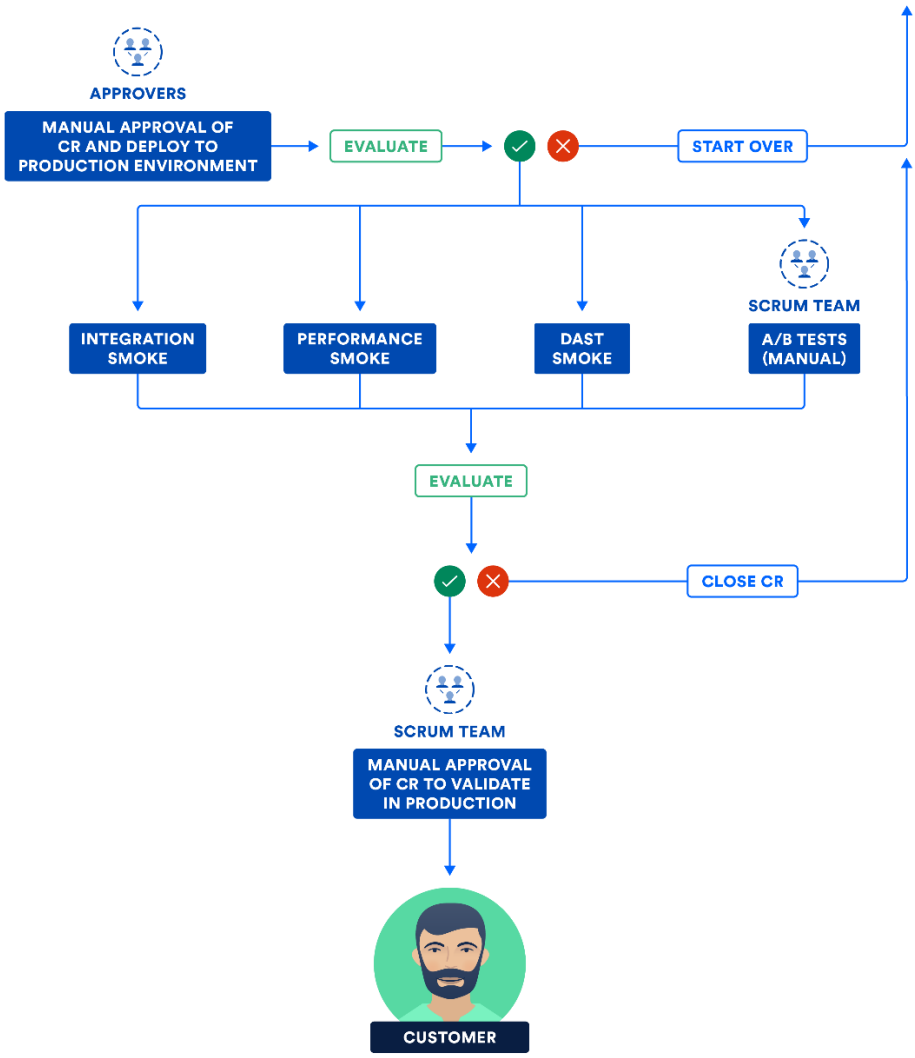
This is a good segway to call out the difference between continuous delivery and continuous deployment. Continuous delivery allows manual gates whereas continuous deployment doesn't. While both are referred to as CD, continuous deployment requires more discipline and rigor since there is no human intervention in the pipeline.

There is a difference between moving the bits and turning them on. Run smoke tests in production, which are a subset of the integration, performance, and security test suites. Once smoke tests pass, turn the bits on, and the product goes live in the hands of our customers!

The following diagram illustrates the steps carried out by the team in this final phase of continuous delivery.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

C) CERTIFYING SUBSYSTEMS AND/OR SYSTEM IN THE PRODUCTION ENVIRONMENT



Continuous delivery is the new normal

To be successful at continuous delivery or continuous deployment, it is critical to do continuous integration and continuous testing well. With a solid foundation, you will win on all three fronts: **quality**, **frequently**, and **predictability**.

A continuous delivery pipeline helps your ideas become products through a series of sustainable experiments. If you discover your idea isn't as good as you thought it was, you can quickly turn around with a better idea. Additionally, pipelines reduce the MTTR (mean time to resolve) production issues, thus reducing

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

downtime for your customers. With continuous delivery, you end up with productive teams and satisfied customers, and who doesn't want that?

Continuous Deployment

Continuous deployment (CD) benefits your software teams and customers. Learn what it is, the benefits, best practices, and more.

What is Continuous Deployment?

Continuous Deployment (CD) is a software release process that uses automated testing to validate if changes to a codebase are correct and stable for immediate autonomous deployment to a production environment.

The software release cycle has evolved over time. The legacy process of moving code from one machine to another and checking if it works as expected used to be an error prone and resource-heavy process. Now, tools can automate this entire deployment process, which allow engineering organizations to focus on core business needs instead of infrastructure overhead.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

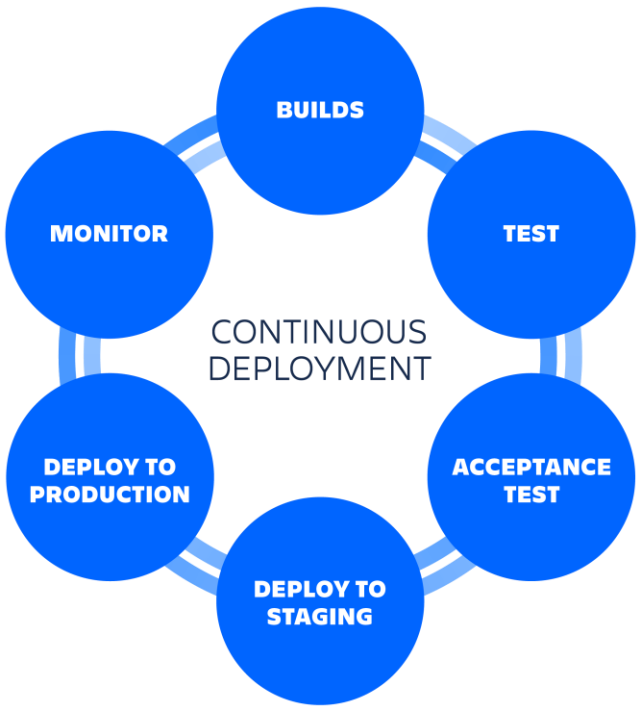
Continuous Deployment vs. Continuous Delivery

The distinction between continuous deployment vs. continuous delivery can be confusing because of the nomenclature. They are both abbreviated as CD and have very similar responsibilities. Delivery is the precursor to deployment. In delivery, there is a final manual approval step before production release.

The following is a mnemonic exercise to help remember the distinction between the two. Think about receiving a package from your favourite online retail store. When waiting for a package to arrive you coordinate with a **delivery** service. This is the delivery phase. Once the package has successfully arrived, you open the package and review its contents to make sure it matches expectations. If it does not, it may be rejected and returned. If the package is correct you are ready to **deploy** and use the new purchase!

In the delivery phase, developers will review and merge code changes that are then packaged into an artifact. This package is then moved to a production environment where it awaits approval to be opened for deployment. In the deployment phase, the package is opened and reviewed with a system of automated checks. If the checks fail the package is rejected.

When the checks pass the package is automatically deployed to production. Continuous deployment is the full end to end, automated software deployment pipeline.



Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

Advantages and disadvantages of continuous deployment

Continuous deployment offers incredible productivity benefits for modern software businesses. It allows businesses to respond to changing market demands and teams to rapidly deploy and validate new ideas and features.

With a continuous deployment pipeline in place, teams can react to customer feedback in real time. If customers submit bug reports or requests for features, teams can immediately respond and deploy responses. If the team has an idea for a new product or feature, it can be in the hands of customers as soon as the code has been pushed.

The benefits of continuous deployment, however, come at a price. While the return on investment is high, a continuous deployment pipeline can be an expensive initial engineering cost. In addition to the initial cost, on-going engineering maintenance may be required to ensure the pipeline continues to run fast and smooth.

Continuous Deployment Tools

Establishing continuous deployment requires substantial engineering investment. The following is a list of tools that are needed to build a continuous deployment pipeline.

- **Automated testing**

  The most critical dependency for continuous deployment is automated testing. In fact, the entire chain of continuous integration, delivery and deployment depends on it. Automated tests are used to prevent any regressions when new code is introduced and can replace manual reviews of new code changes.

- **Rolling deployments**

  The distinguishing feature between continuous deployment and delivery is the automated step of activating new code within a live environment. A continuous deployment pipeline must be able to undo a deployment in the event that bugs or breaking changes are deployed. Automated rolling deployment tools like green-blue deploys are a requirement for proper continuous deployment.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com

- **Monitoring and alerts**

  A robust continuous deployment pipeline will have real time monitoring and alerts. These tools provide visibility into the health of the overall system and into the before and after state of new code deployments. Additionally alerts can be used to trigger a rolling deployment 'undo' to revert a failed deploy.

Continuous Deployment Best Practices

Once a continuous deployment pipeline is established, ongoing maintenance and participation is required from the engineering team to ensure its success. The following best practices and behaviours will ensure an engineering team is getting the most value out of a continuous deployment pipeline.

- **Test-driven development**

  Test-driven development is the practice of defining a behaviour spec for new software features before development begins. Once the spec is defined developers will then write automated tests that match the spec. Finally, the actual deliverable code is written to satisfy the test cases and match the spec. This process ensures that all new code is covered with automated testing up front. The alternative to this is delivering the code first and then producing test coverage after. This leaves opportunity for gaps between the expected spec behaviour and the produced code.

- **Single method of deployment**

  Once a continuous deployment pipeline is in place, it's critical that it is the only method of deployment. Developers should not be manually copying code to production or live editing things. Manual changes external to the CD pipeline will desync the deployment history, breaking the CD flow.

- **Containerization**
  Containerizing a software application ensures that it behaves the same across any machine it is deployed on. This eliminates a whole class of issues where software works on one machine but behaves differently on another. Containers can be integrated as part of the CD pipeline so that the code behaves the same on a developer's machine as it does during automated testing, and production deployment.

Summary

Continuous deployment can be a powerful tool for modern engineering organizations. Deployment is the final step of the overall 'continuous pipeline' that consists of integration, delivery, and deployment. The true experience of continuous deployment is automation to the level at which code is deployed to production, tested for correctness, and automatically reverted when wrong, or accepted if correct.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca
Email: s.stanley.mca@gmail.com