CONTENTS

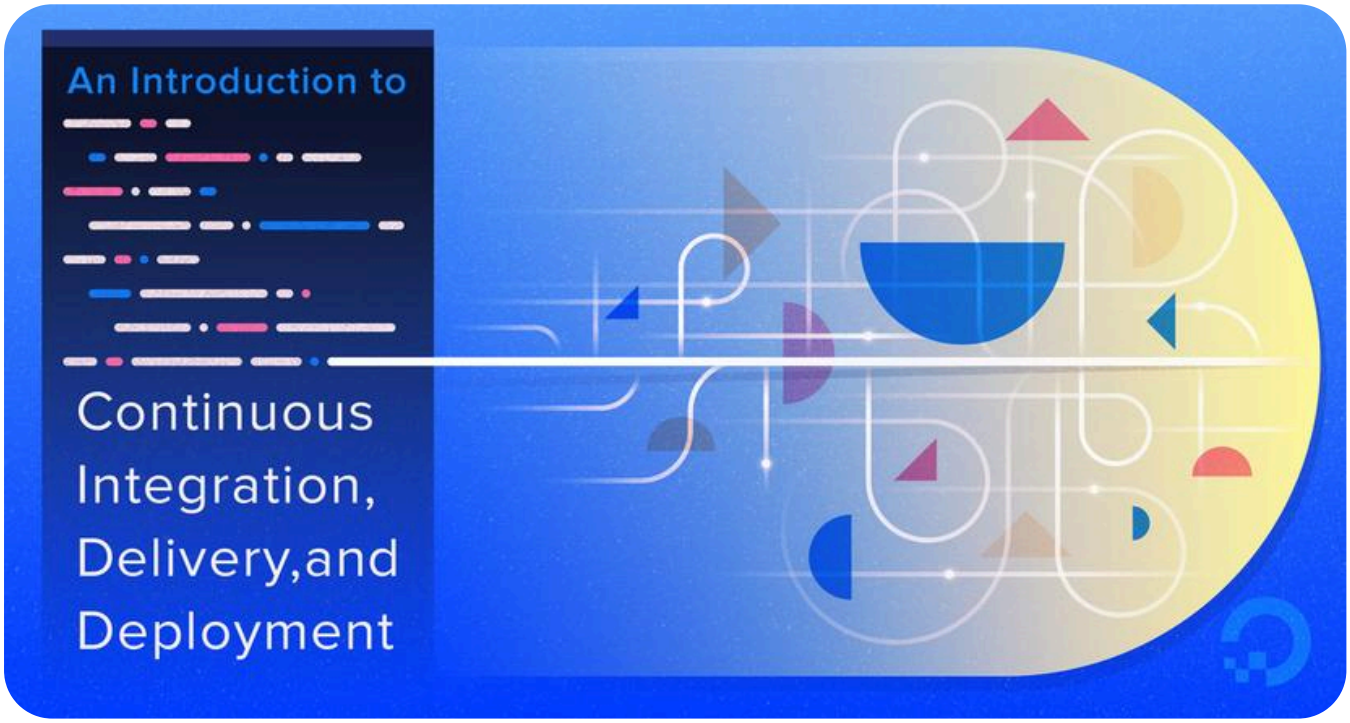
// TUTORIAL //

An Introduction to Continuous Integration, Delivery, and Deployment

Updated on March 18, 2022

CI/CD Conceptual

Justin Ellingwood



Introduction

Developing and releasing software can be a complicated process, especially as applications, teams, and deployment infrastructure grow in complexity. Often, challenges become more pronounced as projects grow. To develop, test, and release software in a quick and consistent way, developers and organizations have created three related but distinct strategies to manage and automate these processes.

that code can be released safely at any time. Continuous deployment takes this one step further by automatically deploying each time a code change is made.

In this guide, we will discuss each of these strategies, how they relate to one another, and how incorporating them into your application life cycle can transform your software development and release practices. To get a better idea of the differences between various open-source CI/CD projects, check out our CI/CD tool comparison.

What is Continuous Integration and Why Is It Helpful?

Continuous integration is a practice that encourages developers to integrate their code into a main branch of a shared repository early and often. Instead of building out features in isolation and integrating them at the end of a development cycle, code is integrated with the shared repository by each developer multiple times throughout the day.

The idea is to minimize the cost of integration by making it an early consideration. Developers can discover conflicts at the boundaries between new and existing code early, while conflicts are still relatively easy to reconcile. Once the conflict is resolved, work can continue with confidence that the new code honors the requirements of the existing codebase.

Integrating code frequently does not, by itself, offer any guarantees about the quality of the new code or functionality. In many organizations, integration is costly because manual processes are used to ensure that the code meets standards, does not introduce bugs, and does not break existing functionality. Frequent integration can create friction when an approach to automation does not align with quality assurance measures in place.

To address this friction within the integration process, in practice, continuous integration relies on robust test suites and an automated system to run those tests. When a developer merges code into the main repository, automated processes kick off a build of the new code. Afterwards, test suites are run against the new build to check whether any integration problems were introduced. If either the build or the test phase fails, the team is alerted so that they can work to fix the build.

The end goal of continuous integration is to make integration a simple, repeatable process that is part of the everyday development workflow in order to reduce integration

Continuous delivery is an extension of continuous integration. It focuses on automating the software delivery process so that teams can easily and confidently deploy their code to production at any time. By ensuring that the codebase is always in a deployable state, releasing software becomes an unremarkable event, without any complicated rituals. Teams can be confident that they can release whenever they need to without complex coordination or late-stage testing. As with continuous integration, continuous delivery is a practice that requires a mixture of technical and organizational improvements to be effective.

On the technology side, continuous delivery leans heavily on deployment pipelines to automate the testing and deployment processes. A **deployment pipeline** is an automated system that runs increasingly rigorous test suites against a build as a series of sequential stages. This picks up where continuous integration leaves off, so a reliable continuous integration setup is a prerequisite to implementing continuous delivery.

At each stage, the build either fails the tests, which alerts the team, or passes the tests, which results in automatic promotion to the next stage. As the build moves through the pipeline, later stages deploy the build to environments that mirror the production environment as closely as possible. This way the build, the deployment process, and the environment can be tested in tandem. The pipeline ends with a build that can be deployed to production at any time in a single step.

The organizational aspects of continuous delivery encourage prioritization of "deployability" as a principle concern. This has an impact on the way that features are built and hooked into the rest of the codebase. Thought must be put into the design of the code so that features can be safely deployed to production at any time, even when incomplete. A number of techniques have emerged to assist in this area.

Continuous delivery is compelling because it automates the steps between checking code into the repository and deciding on whether to release well-tested, functional builds to your production infrastructure. The steps that help assert the quality and correctness of the code are automated, but the final decision about what to release is left in the hands of the organization for maximum flexibility.

What is Continuous Deployment and Why Is It Helpful?

Continuous deployment is an extension of continuous delivery that automatically deploys each build that passes the full test cycle. Instead of waiting for a human

pushes features and fixes to customers quickly, encourages smaller changes with limited scope, and helps avoid confusion over what is currently deployed to production.

This fully automated deploy cycle can be a source of anxiety for organizations worried about relinquishing control to their automation system of what gets released. The trade-off offered by automated deployments is sometimes considered too dangerous for the payoff they provide.

Other groups leverage this approach as a means of ensuring that best practices are always followed. Without a final manual verification before deploying a piece of code, developers must take responsibility for ensuring that their code is well-designed and that the test suites are up-to-date. This consolidates any decision-making around what and when to commit to the main repository and what and when to release to production into a single decision point for the development team.

Continuous deployment also allows organizations to benefit from consistent early feedback. Features can immediately be made available to users and defects or unhelpful implementations can be caught early before the team devotes extensive effort in an unproductive direction. Getting fast feedback that a feature isn't helpful lets the team shift focus rather than sinking more energy into an area with minimal impact.

Key Concepts and Practices for Continuous Processes

While continuous integration, delivery, and deployment vary in the scope of their involvement, there are some concepts and practices that are fundamental to the success of each.

Small, Iterative Changes

One of the most important practices when adopting continuous integration is to encourage small changes. Developers should practice breaking up larger work into small pieces and committing those early. Special techniques like *branching by abstraction* and feature flags (see below) help to protect the functionality of the main branch from in-progress code changes.

Small changes minimize the possibility and impact of integration problems. By committing to the shared branch at the earliest possible stage and then continually throughout development, the cost of integration is diminished and unrelated work is

With trunk-based development, work is done in the main branch ("trunk") of the repository or merged back into the shared repository at frequent intervals. Short-lived feature branches are permissible as long as they represent small changes and are merged back as soon as possible.

The idea behind trunk-based development is to avoid large commits that violate of concept of small, iterative changes discussed above. Code is available to peers early so that conflicts can be resolved when their scope is small.

Releases are performed from the main branch or from a release branch created from the trunk specifically for that purpose. No development occurs on the release branches in order to maintain focus on the main branch as the single source of truth.

Keep the Building and Testing Phases Fast

Each of the processes relies on automated building and testing to validate correctness. Because the build and test steps must be performed frequently, it is essential that these processes be streamlined to minimize the time spent on these steps.

Increases in build time should be treated as a major problem because the impact is compounded by the fact that each commit kicks off a build. Because continuous processes force developers to engage with these activities daily, reducing friction in these areas is very worthwhile.

When possible, running different sections of the test suite in parallel can help move the build through the pipeline faster. Care should also be taken to make sure the proportion of each *type* of test makes sense. Unit tests are typically very fast and have minimal maintenance overhead. In contrast, acceptance testing is often complex and prone to breakage. To account for this, it is often a good idea to rely heavily on unit tests, conduct a fair number of integration tests, and minimize the number of more complex tests.

Consistency Throughout the Deployment Pipeline

Because a continuous delivery or deployment implementations is supposed to be testing release worthiness, it is essential to maintain consistency during each step of the process — the build itself, the deployment environments, and the deployment process itself:

- **Code should be built once at the beginning of the pipeline**: The resulting software should be stored and accessible to later processes without rebuilding. By using the exact same artifact in each phase, you can be certain that you are not introducing

legacy conditions from compromising the integrity of the tests. The staging environments should match the production environment as closely as possible to reduce unknown factors present when the build is promoted.
- **Consistent processes should be used to deploy the build in each environment**: Each deployment should be automated and each deployment should use the same centralized tools and procedures. Ad-hoc deployments should be eliminated in favor of deploying only with the pipeline tools.

Decouple Deployment and Release

Separating the deployment of code from its release to users is an extremely powerful part of continuous delivery and deployment. Code can be deployed to production without initially activating it or making it accessible to users. Then, the organization decides when to release new functionality or features independent from deployment.

This gives organizations a great deal of flexibility by separating business decisions from technical processes. If the code is already on the servers, then deployment is no longer a delicate part of the release process, which minimizes the number of participants and the amount of work involved at the time of release.

There are a number of techniques that help teams deploy the code responsible for a feature without releasing it. Feature flags set up conditional logic to check whether to run code based on the value of an environmental variable. Branching by abstraction allows developers to rewrite processes incrementally by creating an abstraction layer between process input and output. Careful planning to incorporate these techniques gives you the ability to decouple these two processes.

Types of Testing

Continuous integration, delivery, and deployment all rely heavily on automated tests to determine the efficacy and correctness of each code change. Different types of tests are needed throughout these processes to assert confidence in a given solution.

While the categories below in no way represent an exhaustive list, and although there is disagreement on the exact definition of each type, these broad categories of tests represent a variety of ways to evaluate code in different contexts.

Smoke Testing

Smoke tests are a special kind of initial checks designed to ensure core functionality as

The idea behind this type of test is to help to catch big red flags in an implementation and to bring attention to problems that might indicate that further testing is either not possible or not worthwhile. Smoke tests are not very extensive, but should be extremely quick. If a change fails a smoke test, it is an early signal that core assertions were broken and that you should not devote any more time to testing until the problem is resolved.

Context-specific smoke tests can be employed at the start of any new phase testing to assert that the basic assumptions and requirements are met. For instance, smoke tests can be used both prior to integration testing or deploying to staging servers, but the conditions to be tested will vary in each case.

Unit Testing

Unit tests are responsible for testing individual elements of code in an isolated and highly targeted way. The functionality of individual functions and classes are tested on their own. Any external dependencies are replaced with stub or mock implementations to focus the test completely on the code in question.

Unit tests are essential to test the correctness of individual code components for internal consistency and correctness before they are placed in more complex contexts. The limited extent of the tests and the removal of dependencies makes it easier to hunt down the cause of any defects. It also is the best time to test a variety of inputs and code branches that might be difficult to reproduce later on. Often, after any smoke tests, unit tests are the first tests that are run when any changes are made.

Unit tests are typically run by individual developers on their own workstation prior to submitting changes. However, continuous integration servers almost always run these tests again as a safeguard before beginning integration tests.

Integration Testing

After unit testing, integration testing is performed by grouping together components and testing them as an assembly. While unit tests validate the functionality of code in isolation, integration tests ensure that components cooperate when interfacing with one another. This type of testing has the opportunity to catch an entirely different class of bugs that are exposed through interaction between components.

Typically, integration tests are performed automatically when code is checked into a shared repository. A continuous integration server checks out the code, performs any

interact with other code as expected. A secondary aim of integration testing is to verify that the changes can be deployed into a clean environment. This is frequently the first testing cycle that is not performed on the developers' own machines, so unknown software and environmental dependencies can also be discovered during this process. This is usually also the first time that new code is tested against real external libraries, services, and data.

System Testing

Once integration tests are performed, another level of testing called system testing can begin. In many ways, system testing acts as an extension to integration testing. The focus of system tests are to make sure that groups of components function correctly as a cohesive whole.

Instead of focusing on the interfaces between components, system tests typically evaluate the outward functionality of a full piece of software. This set of tests ignores the constituent parts in order to gauge the composed software as a unified entity. Because of this distinction, system tests usually focus on user- or externally-accessible interfaces.

Acceptance Testing

Acceptance tests are one of the last types of tests that are performed on software prior to delivery. Acceptance testing is used to determine whether a piece of software satisfies all of the requirements from the business or user's perspective. These tests are sometimes built against the original specification and often test interfaces for some expected functionality and for usability.

Acceptance testing is often a more involved phase that might extend past the release of the software. Automated acceptance testing can be used to make sure the technological requirements of the design were met, but manual verification usually also plays a role.

Frequently, acceptance testing begins by deploying the build to a staging environment that mirrors the production system. From here, the automated test suites can run and internal users can access the system to check whether it functions the way they need it to. After releasing the software or offering beta access to users, further acceptance testing is performed by evaluating how the software functions in real-world use, and by collecting additional feedback.

Additional Terminology

- **Blue-Green Deployment**: Blue-green deployment is a strategy for testing code in a production-like environment and for deploying code with minimal downtime. Two sets of production-capable environments are maintained, and code is deployed to the inactive set where testing can take place. When ready to release, production traffic is routed to the servers with the new code, instantly making the changes available.
- **Branching by Abstraction**: Branching by abstraction is a method of performing major refactoring operations in an active project without long-lived development branches in the source code repository, which continuous integration practices discourage. An abstraction layer is built and deployed within an existing implementation so that the new implementation can be built out behind the abstraction in parallel.
- **Build (noun)**: A build is a specific version of software created from source code. Depending on the language, this might be compiled code or a consistent bundle of interpreted code.
- **Canary Releases**: Canary releases are a strategy for releasing changes to a limited subset of users. The idea is to make sure everything works correctly with production workloads while minimizing the impact if there are problems.
- **Dark launch**: Dark launching is the practice of deploying code to production that receives production traffic but does not impact the user experience. New changes are deployed alongside existing implementations and the same traffic is often routed to both places for testing. The old implementation is still hooked up to the user's interface, but behind the scenes, the new code can be evaluated for correctness using real user requests in the production environment.
- **Deployment Pipeline**: A deployment pipeline is a set of components that moves software through increasingly rigorous testing and deployment scenarios to evaluate its readiness for release. The pipeline typically ends by automatically deploying to production or providing the option to do so manually.
- **Feature Flags** or **Feature Toggles**: Feature flags are a technique of deploying new features behind conditional logic that determines whether or not to run based on the value of an environment variable. New code can be deployed to production without being activated by setting this flag appropriately. Feature flags often contain logic that allows for subsets of users to gain access to the new feature, creating a mechanism to gradually roll out the new code.
- **Promoting**: In the context of continuous processes, promoting means moving a software build through to the next stage of testing.
- **Soak Test**: Soak testing involves testing software under significant production or production-like load for an extended period of time.

Conclusion

and tools needed to implement these solutions represent a significant investment, the benefits of a well-designed and properly used system can be enormous.

To find out which CI/CD solution might be right for your project, take a look at our CI/CD tool comparison guide for more information.

About the authors

Justin Ellingwood Author

Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

David Mwangi • May 21, 2017

Very informative.

Reply

Ramakrishnan • April 5, 2018

Thanks for the article, useful and informative.

Reply

ngay20thang2nam1990 • February 21, 2018

Thanks for this tutorial, really useful for me.

Reply

McQueen • February 12, 2018

Mehmet Dogan • January 10, 2018

Is load testing missing? SOAK is new for me!

Reply

Jakub Nowak • October 16, 2017

Thanks for this tutorial, really useful for me :)

Reply

byebyebaby • September 7, 2017

This comment has been deleted

byebyebaby • September 7, 2017

This comment has been deleted

Derek Osaro • July 2, 2017

This comment has been deleted

Derek Osaro • July 2, 2017

This comment has been deleted

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

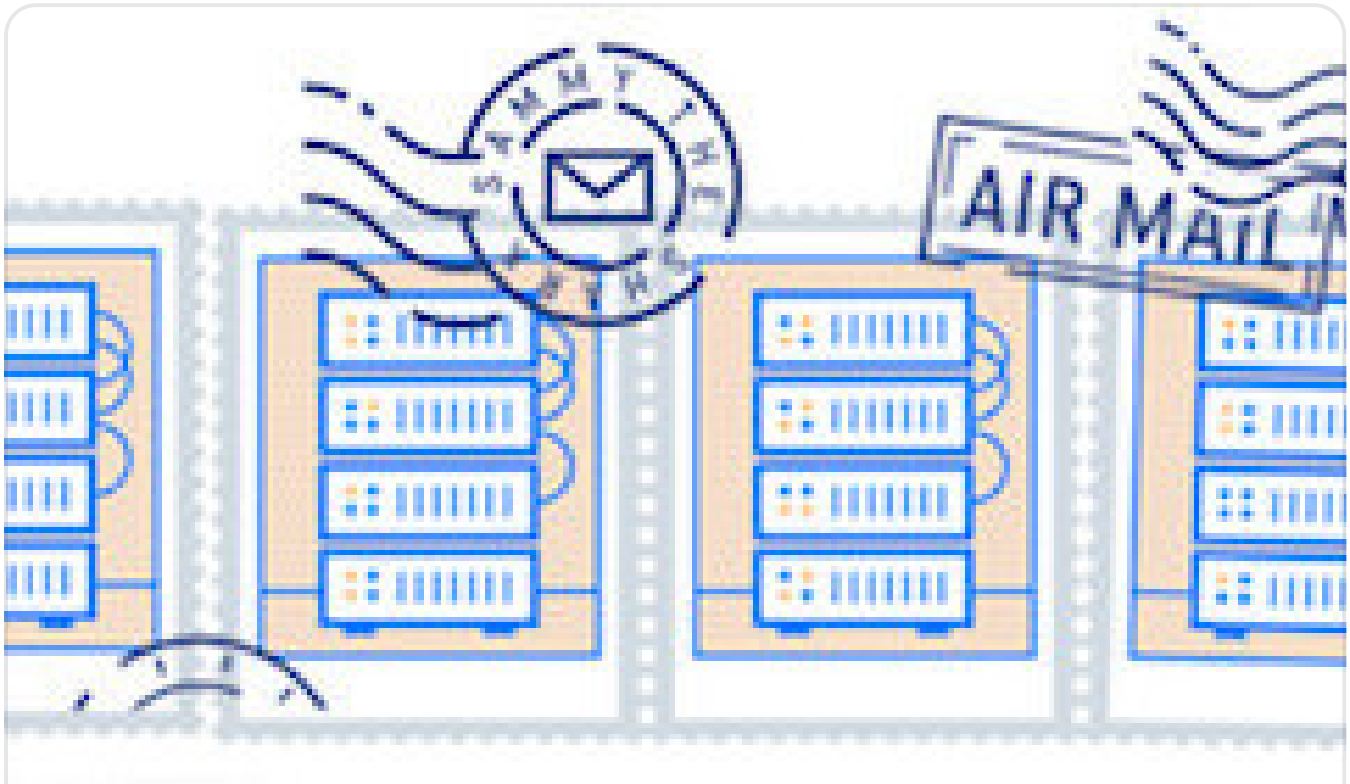
MySQL

Docker

Kubernetes

All tutorials →

Talk to an expert →



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →

Featured Tutorials

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

App Platform Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Products

Community

Solutions

Contact