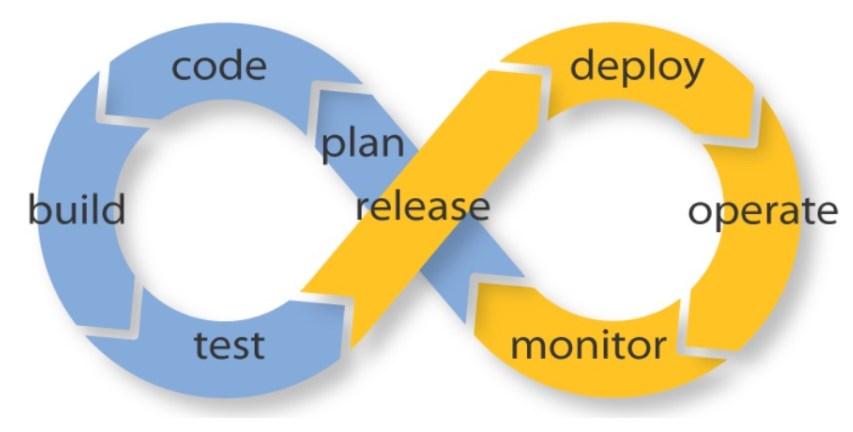
The adoption of CI/CD has changed how developers and testers ship software.  This document will provide insights into different tools and process changes which could help developers be more successful with CI/CD.

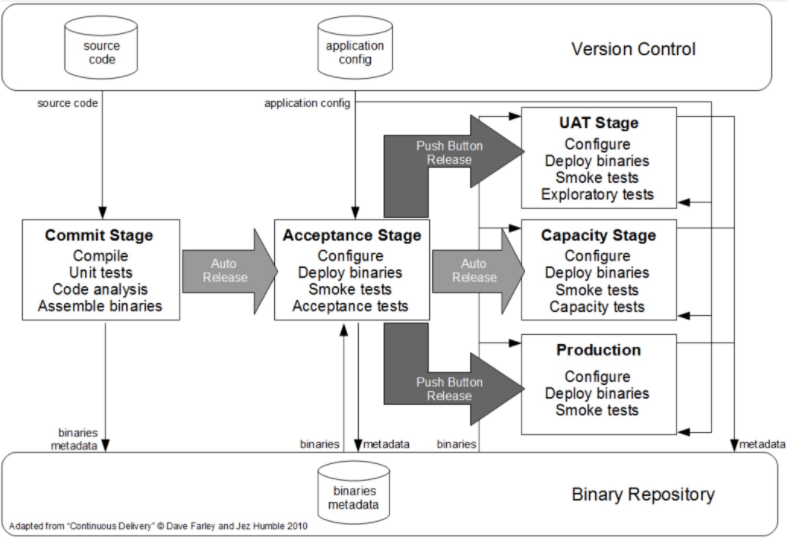


First it was [Waterfall](https://en.wikipedia.org/wiki/Waterfall_model), next it was [Agile](https://en.wikipedia.org/wiki/Agile_software_development), and now it's [DevOps](https://aws.amazon.com/devops/what-is-devops/).  This is how modern developers approach building great products.  With the rise of DevOps has come the new methods of Continuous Integration, Continuous Delivery, (CI/CD) and Continuous Deployment.  Conventional software development and delivery methods are rapidly becoming obsolete. Historically, in the agile age, most companies would deploy/ship software in monthly, quarterly, bi-annual, or even annual releases (remember those days?). Now, however, in the DevOps era, weekly, daily, and even multiple times a day is the norm.  This is especially true as SaaS is taking over the world and you can easily update applications on the fly without forcing customers to download new components.  Often times, they won’t even realize things are changing.

Development teams have adapted to the shortened delivery cycles by embracing automation across their software delivery pipeline. Most teams have automated processes to check in code and deploy to new environments.  This has been coupled with a focus on automating the testing process along the way as well, but that we’ll cover in a future article.  Instead, today, we’ll cover CI/CD/CD, what this is and how modern software companies are using tools to automate the process of shipping new code.

**Continuous integration** focuses on blending the work products of individual developers together into a repository. Often, this is done several times each day, and the primary purpose is to enable early detection of integration bugs, which should eventually result in tighter cohesion and more development collaboration. The aim of **continuous delivery** is to minimize the friction points that are inherent in the deployment or release processes. Typically, the implementation involves automating each of the steps for build deployments such that a safe code release can be done—ideally—at any moment in time. **Continuous deployment**is a higher degree of automation, in which a build/deployment occurs automatically whenever a major change is made to the code.

Each of these stages is part of a **delivery pipeline**. In their book,*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, [Humble and Farley explain](http://www.informit.com/articles/article.aspx?p=1621865&seqNum=2) that “Every change to your software goes through a complex process on its way to being released. That process involves building the software, followed by the progress of these builds through multiple stages of testing and deployment. This, in turn, requires collaboration between many individuals, and perhaps several teams. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool is what allows you to see and control the progress of each change as it moves from version control through various sets of tests and deployments to release to users.”



*A basic deployment pipeline.*

**Continuous Integration (CI)**

With continuous integration, developers frequently integrate their code into a main branch of a common repository. Rather than building features in isolation and submitting each of them at the end of the cycle, a developer will strive to contribute software work products to the repository several times on any given day.

The big idea here is to reduce integration costs by having developers do it sooner and more frequently. In practice, a developer will often discover boundary conflicts between new and existing code at the time of integration. If it’s done early and often, the expectation is that such conflict resolutions will be easier and less costly to perform.

Of course, there are tradeoffs. This process change does not provide any additional quality assurances. Indeed, many organizations find that such integration becomes more costly since they rely on manual procedures to ensure that new code doesn’t introduce new bugs, and doesn’t break existing code. To reduce friction during integration tasks, continuous integration relies on test suites and an automated test execution. It’s important, however, to realize that automated testing is quite different from continuous testing, as we near the end of this article.

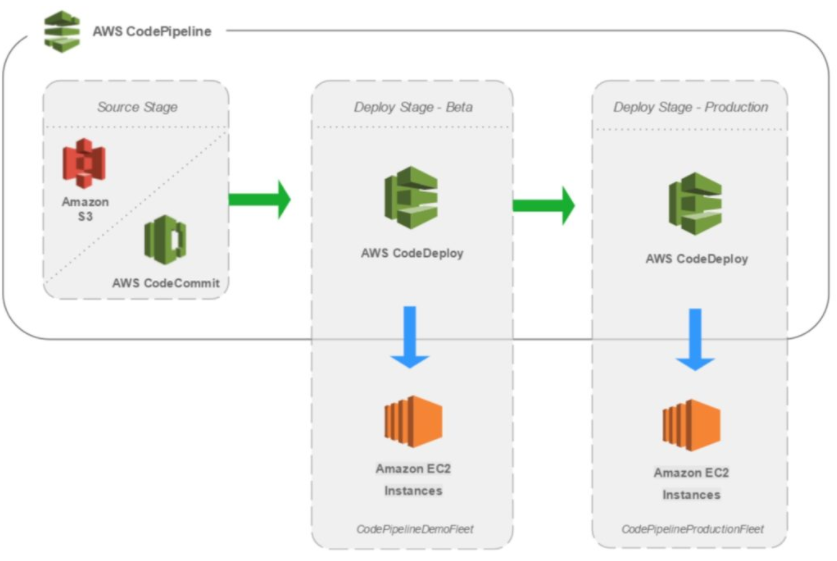
The goal of CI is to refine integration into a simple, easily-repeatable everyday development task that will serve to reduce overall build costs and reveal defects early in the cycle. Success in CI will depend on changes to the culture of the development team so that there is incentive for readiness, frequent and iterative builds, and eagerness to deal with bugs when they are found much earlier.

Continuous delivery is actually an extension of CI, in which the software delivery process is automated further to enable easy and confident deployments into production —at any time. A mature continuous delivery process exhibits a codebase that is always deployable—on the spot. With CD, software release becomes a routine event without emotion or urgency. Teams proceed with daily development tasks in the confidence that they can build a production-grade release—any old time they please—without elaborate orchestration or special late-game testing.

CD depends centrally on a deployment pipeline by which the team automates the testing and deployment processes. This pipeline is an automated system that executes a progressive set of test suites against the build. CD is highly automatable—and in some cloud-computing environments—easily configurable.

In each segment in the pipeline, the build may fail a critical test and alert the team. Otherwise, it continues on to the next test suite, and successive test passes will result in automatic promotion to the next segment in the pipeline. The last segment in the pipeline will deploy the build to a production-equivalent environment. This is a comprehensive activity, since the build, the deployment, and the environment are all exercised and tested together. The result is a build that is deployable and verifiable in an actual production environment.

A solid exhibit of a modern CI/CD pipeline is available on AWS. Amazon is one of the cloud-computing providers that offers an impressive CI/CD [pipeline environment](https://aws.amazon.com/getting-started/projects/set-up-ci-cd-pipeline/) and provides a walk-through procedure in which you can choose from among its many development resources and link them together in a pipeline that is readily configurable and easily monitored.



Many consider continuous delivery to be attractive primarily because it automates all of the steps that progress from submitting code into the repository through to releasing fully-tested, properly-functional builds that are ready for production. This is elaborate automation of the build and testing processes, but decisions about how and what should be released remains a manual process. Continuous deployment can improve and automate those activities.

**Continuous Deployment (CD)**

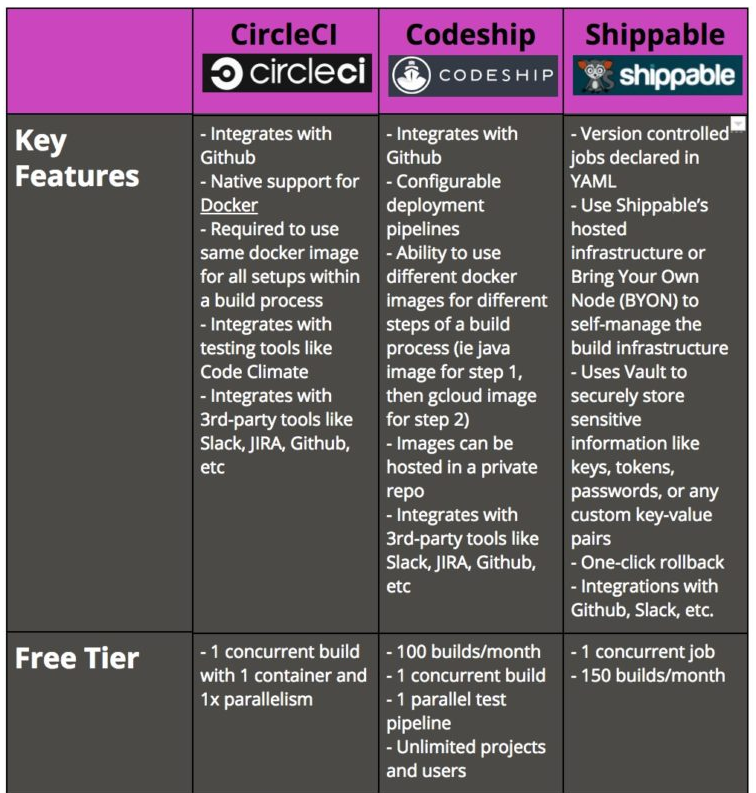
Continuous deployment extends continuous delivery so that the software build will automatically deploy if it passes all tests. In such a process, there is no need for a person to decide when and what goes into production. The last step in a CI/CD system will automatically deploy whatever build components/packages successfully exit the delivery pipeline. Such automatic deployments can be configured to quickly distribute components, features, and fixes to customers, and provide clarity on precisely what has is presently in production.

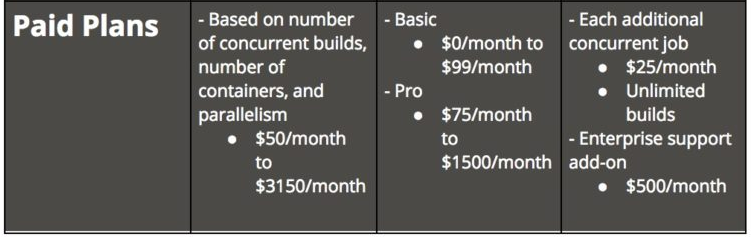
Organizations that employ continuous deployment will likely benefit from very quick user feedback on new deployments. Features are quickly delivered to users, and any defects that become evident can be handled promptly.  Quick user response on unhelpful or misunderstood features will help the team refocus and avoid devoting more effort into to functional area that is unlikely to produce a good return on that investment.

With the movement to DevOps, there’s also been a rise of new automation tools to help with the CI/CD pipeline.  These tools typically integrate with various developer tools including code repository systems like Github and bug tracking systems like Jira.  In addition, as SaaS has become a more popular delivery model, many of these tools are running in the cloud, similar to where modern developers are running their apps, like [GCP](https://cloud.google.com/) and AWS.

The most popular automation tool is [Jenkins](https://jenkins.io/) (formerly Hudson), which is an open source project supported by hundreds of contributors as well as a commercial company, [Cloudbees](https://www.cloudbees.com/" \t "_blank).  Cloudbees even hired the founder of Jenkins and offers several different Jenkins training programs and product add-ons. Beyond open source, there are also several more modern, commercial, software products available including CircleCI, Codeship, and Shippable.  These products have several different advantages and disadvantages to one another.  In order to really understand them, I’d encourage trying each of them specifically within your developer workflow to see how they work in your environment.  How do they work with your tools, your cloud platform, your container system, etc

At mabl, we’re [building on Google Cloud Platform](https://www.mabl.com/validating-100-million-pages-kubernetes/), so we were looking for a product that was compatible with, and preferably integrated with, GCP.  We took a look at CircleCI, Codeship, and Shippable and below is a short table highlighting some details for each:





We ultimately landed on [Codeship](http://codeship.com/) and couldn’t be happier with our decision and the support we get from the Codeship team.

Water Fall model



The major disadvantages of the Waterfall Model are as follows −

* No working software is produced until late during the life cycle.
* High amounts of risk and uncertainty.
* It is difficult to measure progress within stages.
* Cannot accommodate changing requirements.
* Adjusting scope during the life cycle can end a project.

Agile Model

In this , we divide



The disadvantages of the Agile Model are as follows −

* Not suitable for handling complex dependencies.
* More risk of sustainability, maintainability and extensibility.
* An overall plan, an agile leader and agile PM practice is a must without which it will not work.
* Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
* There is a very high individual dependency, since there is minimum documentation generated.
* Transfer of technology to new team members may be quite challenging due to lack of documentation.

