DevOps Assignment-1

Name: **Shreyas G Trivikram**

USN: **4NI19IS093**

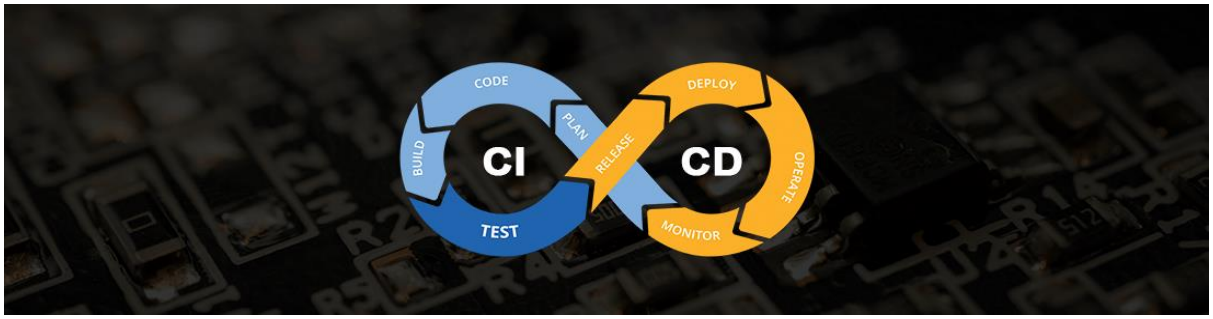
Continuous Integration and Continuous Deployment (CI/CD)

The software delivery process is automated through a CI/CD pipeline. It aids with the development of code, testing, and the secure release of updated software. The CI/CD pipeline decreases manual errors, gives engineers feedback, and enables quick product iterations.

Through the use of a CI/CD pipeline, continuous monitoring and automation are introduced. It spans the delivery and deployment phases as well as the integration and testing phases. The term "CI/CD pipeline" refers to these interconnected procedures.

What is Continuous Integration, Continuous Delivery, and Continuous Deployment?

CI/CD allows organizations to ship software quickly and efficiently. CI/CD facilitates an effective process for getting products to market faster than ever before, continuously delivering code into production, and ensuring an ongoing flow of new features and bug fixes via the most efficient delivery method.



Difference between CI and CD

Continuous integration (CI) is a technique where developers check and make minor changes to their code. This process is automated because of the size of the requirements and the number of steps needed to complete it, allowing teams to design, test, and bundle their applications in a dependable and repeatable manner. CI streamlines code updates, giving developers more time to make changes and contribute to better products.

The automated transmission of finished code to contexts like testing and development is known as continuous delivery (CD). Code delivery to various contexts is made automated and standardised via CD.

Continuous Deployment is the next step of continuous delivery. Every change that passes the automated tests is automatically placed in production, resulting in many production deployments.

Continuous deployment should be the goal of most companies that are not constrained by regulatory or other requirements.

How CI/CD relate to DevOps

The goal of DevOps is to boost an organization's capacity to produce applications and services more quickly than with conventional software development methods. DevOps' enhanced speed enables businesses to better service their clients and compete in their respective markets. Successful organisations in a DevOps environment "bake security in" to all stages of the development life cycle, a practise known as “DevSecOps”.

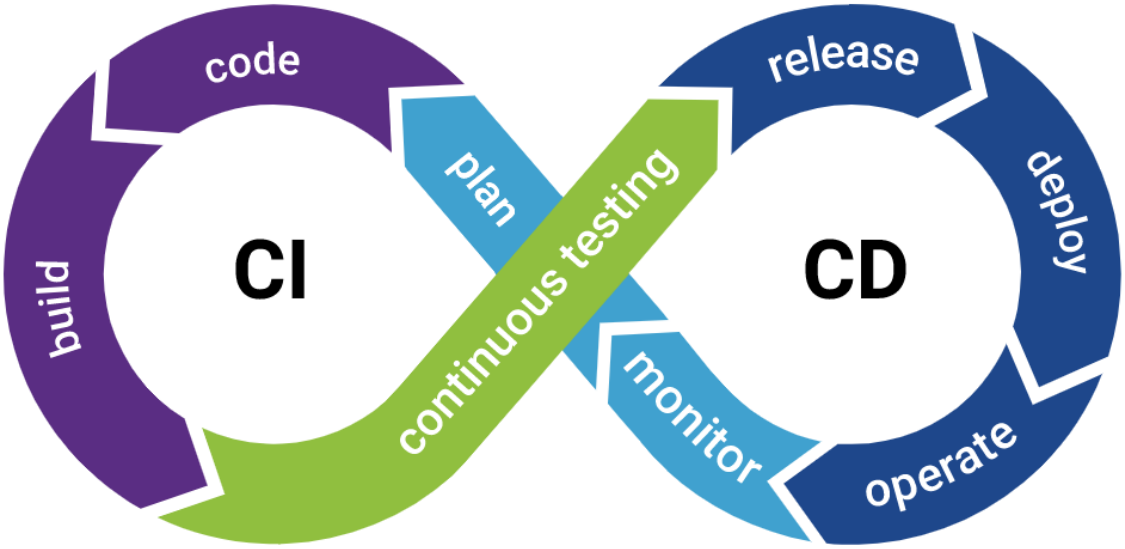
Part of the larger DevOps/DevSecOps paradigm is the CI/CD pipeline. Organizations require solutions to avoid friction points that slow down integration and delivery in order to properly instal and manage a CI/CD pipeline.

Stages of CI/CD Pipeline

The continuous integration/continuous delivery (CI/CD) pipeline is an agile DevOps workflow with the goal of delivering software on a regular basis with high reliability. DevOps teams may build code, integrate it, run tests, deliver releases, and distribute changes to the software collaboratively and in real-time thanks to the iterative, as opposed to linear, process.

The use of automation to guarantee code quality is an important component of the CI/CD workflow. Test automation is used to deploy code changes to various environments, deliver applications to production environments, and uncover dependencies and other issues earlier as the software changes move through the pipeline. The automation's task in this situation is to perform quality control, evaluating factors like performance, API usage, and security. This ensures the changes made by all team members are integrated comprehensively and perform as intended.

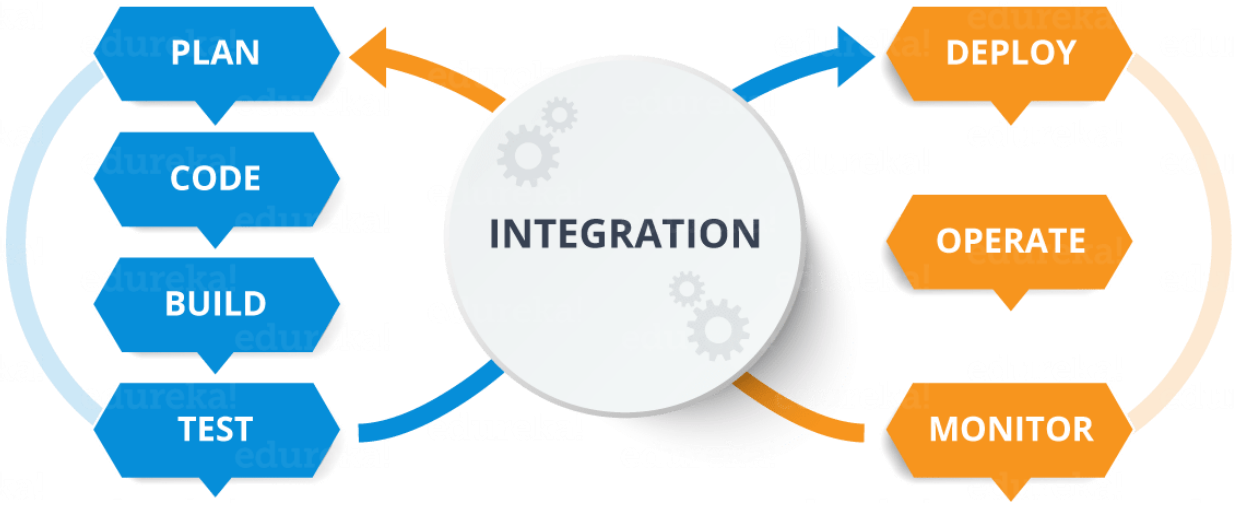
The ability to automate various phases of the CI/CD pipeline helps development teams improve quality, work faster and improve other DevOps metrics.

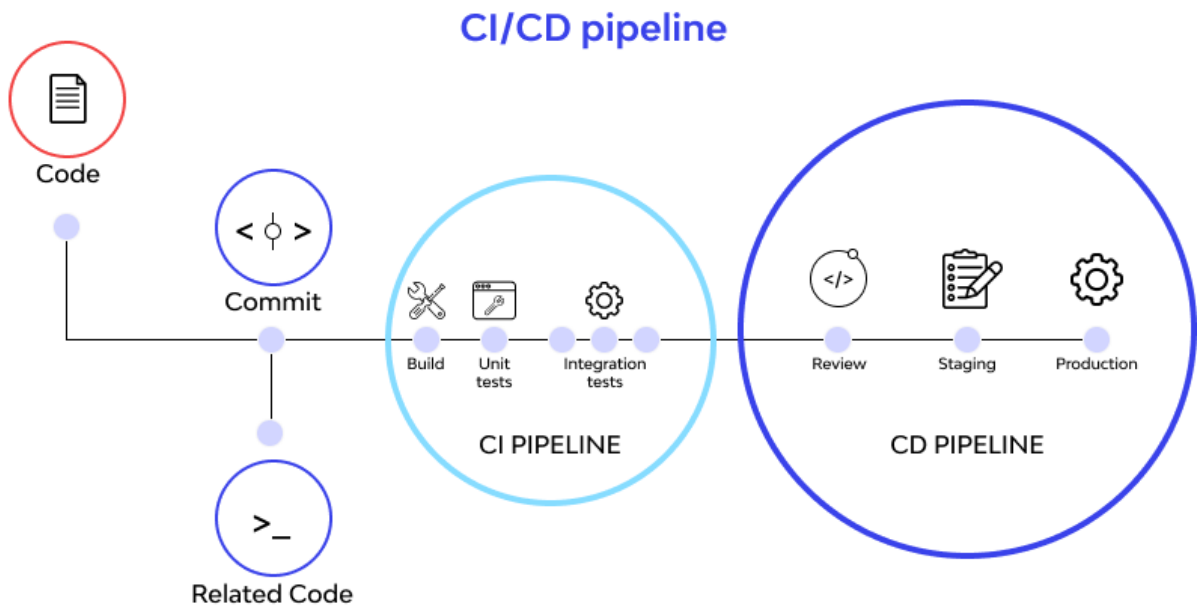


CI/CD pipeline phases

From source code to production, these phases make up the development lifecycle and workflow of the CI/CD pipeline:

- **Build:** This phase is part of the continuous integration process and involves the creation and compiling of code. Teams build off of source code collaboratively and integrate new code while quickly determining any issues or conflicts.
- **Test:** At this stage, teams test the code. Automated tests happen in both continuous delivery and deployment. These tests could include integration tests, unit tests, and regression tests.
- **Deliver:** Here, an approved codebase is sent to a production environment. This stage is automated in continuous deployment and is only automated in continuous delivery after developer approval.
- **Deploy:** Lastly, the changes are deployed and the final product moves into production. In continuous delivery, products or code are sent to repositories and then moved into production or deployment by human approval. In continuous deployment, this step is automated.





Implementation of CI/CD pipeline stages

Plan

Outline a feature that needs to be constructed or a problem that requires fixing.

Code

Convert flowcharts and use case ideas into code, peer-review code updates, and get design input.

Test

Verify code changes through testing, preferably automated testing. At this point, unit testing will usually suffice.

Repository

To keep track of changes made, push code to a shared repository like Github that uses version control tools. Some people view this as the pipeline's first stage.

Set up an Integration Testing Service

For example, Travis CI to continuously run automated tests, such as regression or integration tests, on software hosted on services like Github and BitBucket.

Set up a Service to Test Code Quality

Better Code Hub can help continuously check code for quality in CI/CD pipeline. This will enable the software development team to spend less time fixing bugs. Better Code Hub uses 10 guidelines to gauge quality and maintainability in order to future proof the code.

Build

This might overlap with compilation and containerization. Assuming the code is written in a programming language like Java, it'll need to be compiled before execution. Therefore, from the version control repository, it'll go to the build phase

where it is compiled. Some pipelines prefer to use Kubernetes for their containerization.

Testing Phase

Build the pipeline's initial verification tests as soon as possible. When a build fails, you are automatically notified and given the opportunity to investigate the cause by looking through the continuous integration logs.

Smoke testing may be conducted at this step to identify and reject a build that is really flawed before more time is wasted installing and testing it. Send the developed container (Docker) image to a Docker hub: This makes it simple to distribute your containers across other settings or platforms, or even to revert to a previous version.

Deploy the App, optionally to a Cloud-Based Platform

The majority of businesses are deploying their apps in the cloud. One example of a reasonably priced cloud platform is Heroku. Others could favour Amazon Web Service or Microsoft Azure (AWS).

Advantages of CI/CD pipelines

- Builds and testing can be easily performed manually.
- It can improve the consistency and quality of code.
- Improves flexibility and has the ability to ship new functionalities.
- CI/CD pipeline can streamline communication.
- It can automate the process of software delivery.
- Helps you to achieve faster customer feedback.
- CI/CD pipeline helps you to increase your product visibility.
- It enables you to remove manual errors.
- Reduces costs and labour.
- CI/CD pipelines can make the software development lifecycle faster.
- It has automated pipeline deployment.
- A CD pipeline gives a rapid feedback loop starting from developer to client.

- Improves communications between organization employees.
- It enables developers to know which changes in the build can turn to the brokerage and to avoid them in the future.
- The automated tests, along with few manual test runs, help to fix any issues that may arise.

Feature Flags

Feature flagging is a software development technique that has been growing and gaining popularity in recent times. However, some development teams still aren't quite familiar with it.

"Feature flag" is one of the many names of a powerful software technique. Also known as a feature toggle, feature switch, and feature flipper, this technique allows you to switch some piece of functionality in your app on and off, from outside of the code.

"From outside of the code" is key. Some people might argue that feature flags aren't really necessary, since you could achieve the same result by commenting out a piece of code and then uncommenting it. That wouldn't cut it even as a super primitive feature flag approach, though. And here's why: an essential part of a feature flagging (toggling) strategy relies on being able to activate and deactivate features, in a production environment, without having to change the code and redeploy the application. For example, you can use feature flags as a kill switch to disable buggy code instantly without having to roll back to a previous version of your application.

How to use Feature Flags (toggles) to enable DevOps

Continuous Integration/Continuous Delivery (CI/CD)

Feature flagging enables CI/CD, by allowing developers to merge their work into the shared repository frequently, without fear of breaking anything or causing the end user to access unfinished features.

One of the most important ideas in modern software engineering that originated in the agile methodologies is the idea that to go fast, you have to take smaller steps. The classic example of this idea is continuous integration (CI). CI states that all developers should merge their work into the shared repository (e.g., GitHub) at least once a day. Doing so helps prevent nasty conflicts by avoiding long-lived silos of code.

One thing that inevitably happens when a team does continuous integration is that developers will face the conundrum of committing partially complete features to the mainline. If they don't send their code, they run the risk of having long-lived feature branches that will make merging a pain down the road. However, by merging their

code, they might allow the end user to interact with features that aren't ready to be used just yet.

Feature toggles can come in handy in scenarios like this one. By putting a feature flag around their incomplete feature and keeping it off by default, developers can ship their code without worry. Even if the incomplete feature gets deployed to the end user, they won't be able to access the functionality. In this way, feature flags enable software teams to do trunk-based development and CI/CD, both key DevOps practices. When the day comes and the feature is finally ready to go, the flag will only have to be activated and the users will be able to access the feature.

A/B testing

DevOps has its roots in Agile. And the agile methodologies are all about taking tiny steps, getting feedback early and often, and using this feedback to course correct. When it comes to obtaining user feedback, you'd be hard-pressed to find a more effective technique than A/B testing.

With feature flags, you can easily serve different versions of your app for different groups simultaneously, which is pretty much what A/B testing is about. It's a research technique organization can use to see how customers react to a new feature or even a new version of an old piece of functionality. A/B testing consists of splitting your user base into two groups, A and B. Group A is called the treatment. This group of users receives the proposed new version of the app or feature. And group B is called the control. People in it receive the old version of the app or feature. The organization then closely monitors both groups of users to determine which version performs better. After the experiment is complete, the organization might choose to roll out the new version to the rest of the user base or roll it back.

Testing in production

Feature flags are a powerful enabler of testing in production. The software development technique allows organizations to put features behind a flag but keep them accessible only to selected personnel. That way, the feature can make it to production in a safe way. Available only to authorized people, the feature can be safely tested, with no risk of it being exposed to regular users.

Testing in production—the right way—is vital in many scenarios nowadays. For instance, some types of testing—loading and performance come to mind—require conditions that are hard if not downright impossible to replicate in testing or QA environments. Also, health checks and other types of monitoring can only be done with the "real thing."

Keep in mind that saying that testing in production is important doesn't mean you should stop testing before production. Production and non-production testing are complementary, and a solid quality strategy will make use of both.

To truly achieve DevOps, though, you must be able to quickly detect when problems happen in production so you can fix them as quickly as possible, or at least mitigate the damage they cause. And testing in production is a path for achieving those results.