1. What is CI/CD?

CICD is a term which is derived from two acronyms: CI and CD. CI stands for Continuous Integration and CD stands for Continuous Delivery/Continuous Deployment. CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment. CI/CD is a solution to the problems integrating new code can cause for development and operations teams (AKA "integration hell").

Continuous Integration is the process of frequently building and testing new code changes. Continuous Delivery is the process of deploying new versions of an application frequently. Both are about automating further stages of the pipeline, but they're sometimes used separately to illustrate just how much automation is happening. CI and CD together empower organizations to design and deliver high-quality software in shorter development life cycles and respond to the needs of the users promptly, resulting in greater customer satisfaction.

Continuous Integration and Continuous Delivery are often quoted as the pillars of successful DevOps. DevOps is a software development approach which bridges the gap between development and operations teams by automating build, test and deployment of applications. It is implemented using the CICD pipeline.

2. What are feature flags and how it is used?

Feature flags (also known as feature toggles or feature switches) are Feature flags are features turned on/off during runtime without deploying new code. These are great for better control and experimentation of features. These are software development technique that turns certain functionality on and off during runtime, without deploying new code. This allows for better control and more experimentation over the full lifecycle of features.

One of the most useful tools in the modern developer's toolbox is the feature flag. Feature flags are a software development concept that allow you to enable or disable a feature without modifying the source code or requiring a redeploy. They are also commonly referred to as feature toggles, release toggles or feature flippers. Feature flags determine at runtime which portions of code are executed. This allows new features to be deployed without making them visible to users or, even more importantly, you can make them visible to only a specific subset of users. Feature flagging is also useful for rolling back the code if needed.

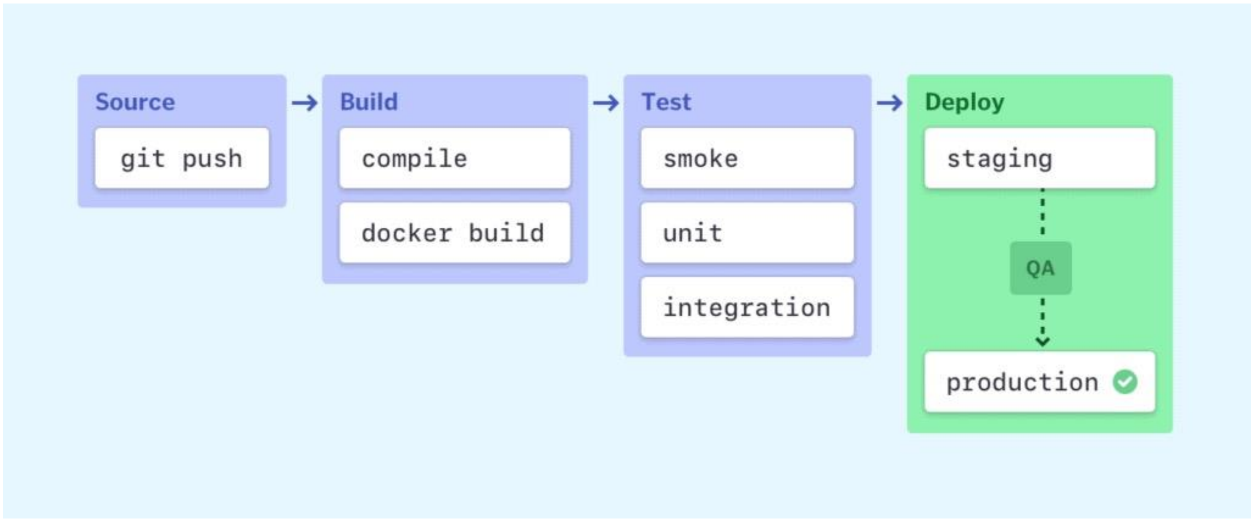
Feature flags are a system of code that allows conditional features to be accessed only when certain conditions are met. In other words, if a flag is on, new code is executed if the flag is off, the code is skipped. Also referred to as or release toggles, feature flags are a best practice in DevOps, often occurring within distributed version control systems.

3. Explain CI/CD pipeline with block diagram?

CI/CD enables the inclusion of freshly committed code into the code repository and facilitates testing at each stage — fulfilling the integration aspect — and ending it's run with a deployment of the complete application to the end-users — fulfilling the delivery aspect. The code will progress from code check-in through the test, build, deploy, and production stages. Engineers over the years have automated the steps for this process. The automation led to two primary processes known as **Continuous Integration** and **Continuous Delivery**. It's essentially a runnable specification of the steps that any developer needs to perform to deliver a new version of a software product. In the absence of an automated pipeline, engineers would still need to perform these steps manually, and hence far less productively.

Standard Definition: A CI/CD pipeline is defined as a series of interconnected steps that include stages from code commit, testing, staging, deployment testing, and finally, deployment into the production servers. We automate most of these stages to create a seamless software delivery.

Most software releases go through a couple of typical stages:



Source stage - In most cases, a pipeline run is triggered by a source code repository. A change in code triggers a notification to the CI/CD tool, which runs the corresponding pipeline. Other common triggers include automatically scheduled or user-initiated workflows, as well as results of other pipelines.

Build stage - We combine the source code and its dependencies to build a runnable instance of our product that we can potentially ship to our end users. Programs written in languages such as Java, C/C++, or Go need to be compiled, whereas Ruby, Python and JavaScript programs work without this step.

Regardless of the language, cloud-native software is typically deployed with Docker, in which case this stage of the CI/CD pipeline builds the Docker containers.

Failure to pass the build stage is an indicator of a fundamental problem in a project's configuration, and it's best to address it immediately.

Test stage - In this phase, we run automated tests to validate our code's correctness and the behavior of our product. The test stage acts as a safety net that prevents easily reproducible bugs from reaching the end-users.

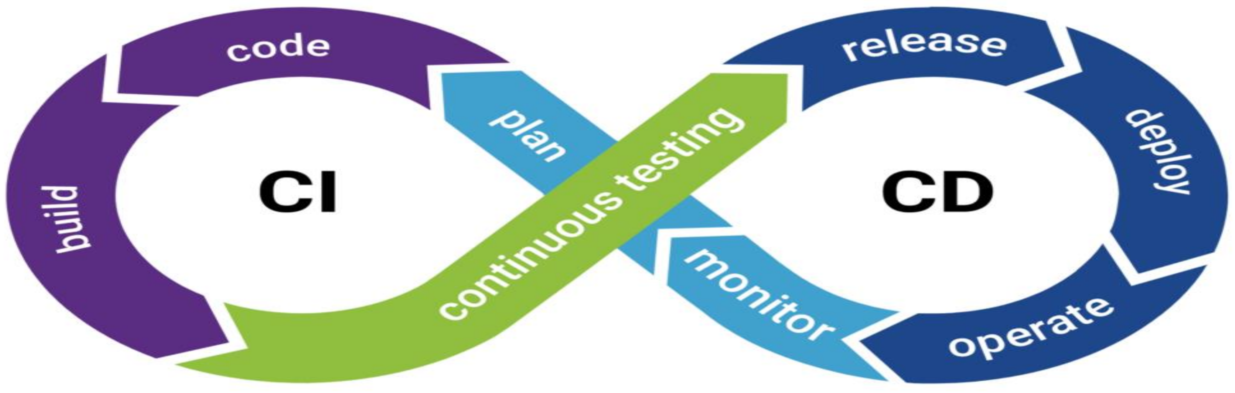
The responsibility of writing tests falls on the developers. The best way to write automated tests is to do so as we write new code in test- or behavior-driven development.

Depending on the size and complexity of the project, this phase can last from seconds to hours. Many large-scale projects run tests in multiple stages, starting with smoke tests that perform quick sanity checks to end-to-end integration tests that test the entire system from the user's point of view. An extensive test suite is typically parallelized to reduce run time.

Failure during the test stage exposes problems in code that developers didn't foresee when writing the code. It's essential for this stage to produce feedback to developers quickly, while the problem space is still fresh in their minds and they can maintain the state of flow.

Deploy stages - Once we have a built a runnable instance of our code that has passed all predefined tests, we're ready to deploy it. There are usually multiple deploy environments, for example, a "beta" or "staging" environment which is used internally by the product team, and a "production" environment for end-users.

Teams that have embraced the Agile model of development—guided by tests and real-time monitoring—usually deploy work-in-progress manually to a staging environment for additional manual testing and review, and automatically deploy approved changes from the master branch to production.



1. Comparison between hypervisor and docker?

A hypervisor allows users to build multiple versions of a complete operating system. Dockers can run multiple applications or instances of the same application. It does this with containers. The hypervisor allows users to run multiple instances of the entire operating system. Below are the enlisted ways in which they are different:

- Functioning Mechanism

The most significant difference between hypervisors and Dockers is the way they boot up and consume resources.

Hypervisors are of two types – the bare metal works directly on the hardware while type two hypervisor works on top of the operating system.

Docker, on the other hand, works on the host kernel itself. Hence, it does not allow the user to create multiple instances of operating systems.

Instead, they create containers that act as virtual application environments for the user to work on.

- Number of Application Instances Supported

A hypervisor allows the users to generate multiple instances of complete operating systems. Dockers can run multiple applications or multiple instances of a single application. It does this with containers.

- Memory Requirement

Hypervisors enable users to run multiple instances of complete operating systems. This makes them resource hungry.

They need dedicated resources for any particular instance among the shared hardware which the hypervisor allocates during boot.

Dockers, however, do not have any such requirements. One can create as many containers as needed.

Based on the application requirement and availability of processing power, the Docker provides it to the containers.

- Boot-Time

As Dockers do not require such resource allocations for creating containers, they can be created quickly to get started.

One of the primary reasons why the use of Dockers and containers is gaining traction is their capability to get started in seconds.

A hypervisor might consume up to a minute to boot the OS and get up and running.

Docker can create containers in seconds, and users can get started in no time.

- Architecture Structure

If we consider both hypervisor and Docker's architecture, we can notice that the Docker engine sits right on top of the host OS.

It only creates instances of the application and libraries.

Hypervisor though, has the host OS and then also has the guest OS further. This creates two layers of the OS that are running on the hardware.

If you are to run a portable program and want to run multiple instances of it, then containers are the best way to go. Hence you can benefit significantly with a Docker.

Dockers help with the agile way of working. Within each container, different sections of the program can be developed and tested.

In the end, all containers can be combined into a single program. Hypervisors do not provide such capability.

- Security

Hypervisors are much more secure since the additional layer helps keep data safe.

One of the major differences between the two is the capability to run operating systems or rather run-on operating systems.

- OS Support

Hypervisors are OS agnostic. They can run across Windows, Mac, and Linux.

Dockers, on the other hand, are limited to Linux only. That, however, is not a deterrent for Dockers since Linux is a strong eco-system. Many major players are entering into the Dockers' fray

Hypervisors and Dockers are not the same, and neither can be used interchangeably. People are often confused between the two because of their applications related to virtualization.

2. Comparison between Containers and Virtual machines?

SNo.	Virtual Machines (VM)	Containers
1.	VM is piece of software that allows you to install other software inside of it so you basically control it virtually as opposed to installing the software directly on the computer.	While a container is a software that allows different functionalities of an application independently.
2.	Applications running on VM system can run different OS.	While applications running in a container environment share a single OS.
3.	VM virtualizes the computer system.	While containers virtualize the operating system only.
4.	VM size is very large.	While the size of container is very light; i.e. a few megabytes.
5.	VM takes minutes to run, due to large size.	While containers take a few seconds to run.
6.	VM uses a lot of system memory.	While containers require very less memory.
7.	VM is more secure.	While containers are less secure.
8.	VM’s are useful when we require all of OS resources to run various applications.	While containers are useful when we are required to maximise the running applications using minimal servers.
9.	Examples of VM are: KVM, Xen, VMware.	While examples of containers are:RancherOS, PhotonOS, Containers by Docker.