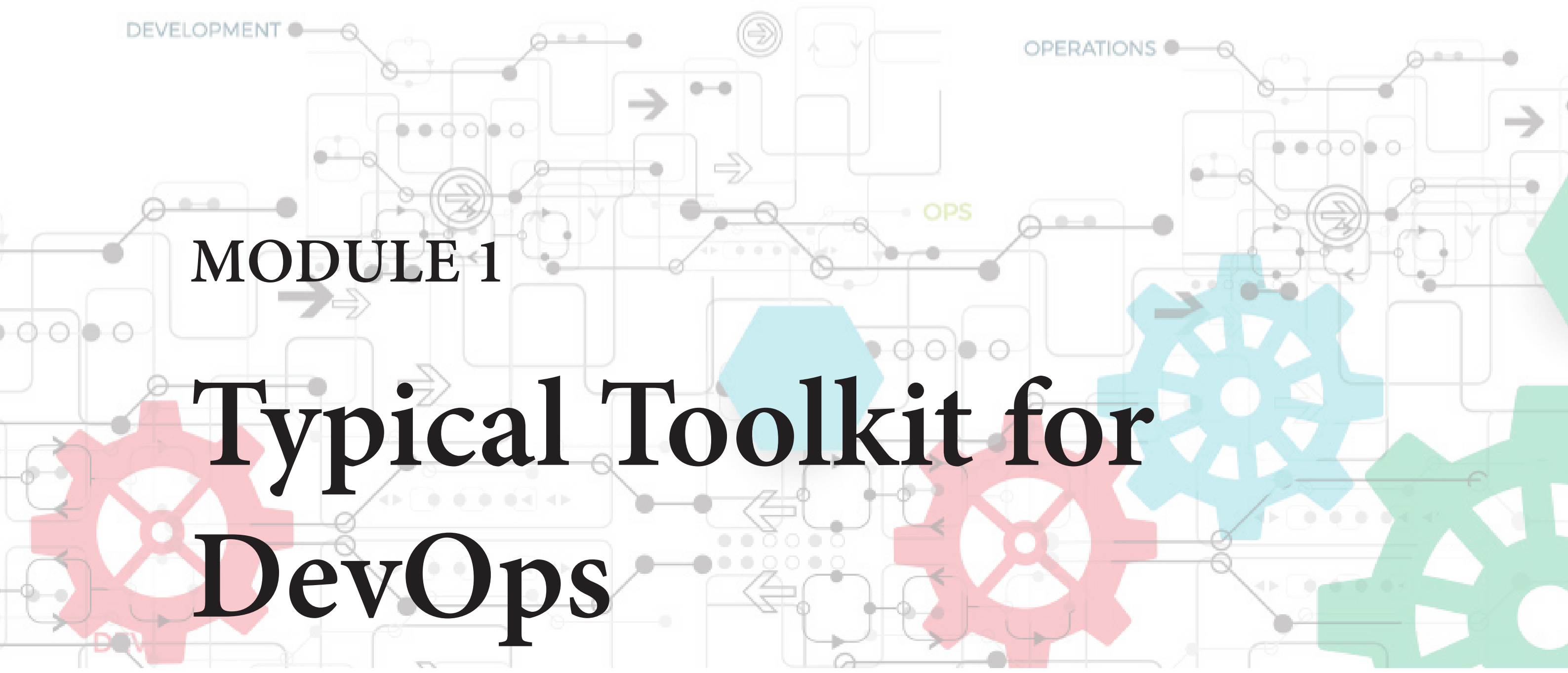


MODULE 1

Typical Toolkit for DevOps

You will learn about the 'Typical Toolkit for DevOps' in this module.

Module Objectives

At the end of this module, you will be able to:

- Define DevOps and the ways of achieving DevOps in an organization
- Identify the basics of continuous practices
- Explain the basics of Continuous integration (CI) and how it works
  - The best practices of CI
  - The benefits of CI
- Describe continuous delivery and the process flow
  - Benefits of continuous delivery
- Examine the basics of continuous deployment and what makes it different from continuous delivery
  - How CD works
  - Best practices of CD
  - Benefits of CD
- Identify version control systems and the associated concepts
  - Types of version control systems
  - Benefits of having version control systems

Module Topics

Let us take a quick look at the topics that we will cover in this module:

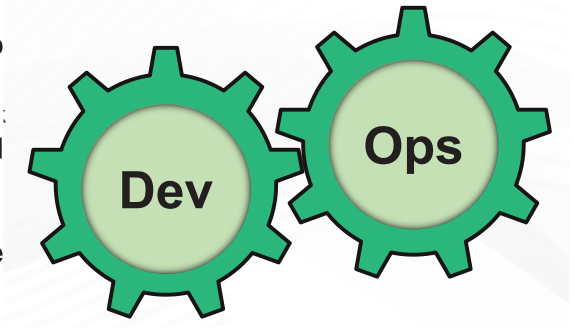
1. Introduction to DevOps and ways of achieving DevOps in an organization
2. Introduction to Continuous Integration (CI)
    - Best practices of CI
    - Benefits of CI
3. Introduction to Continuous Delivery (CDE*)
    - Benefits of continuous delivery
4. Introduction to Continuous Deployment (CD)
    - The CD process
    - Best practices of CD
    - Benefits of CD
5. Version control systems
    - Types of version control systems
    - Benefits of version control systems

***Note**: Though CD may refer to Continuous delivery and deployment, in this course, for the sake of clarity, continuous delivery has been abbreviated as CDE and continuous deployment as CD.*

1.1 DevOps: An Overview

Following are the key details of DevOps:

- DevOps emerged as a culture to overcome the issues associated with traditional development and delivery.
- DevOps is an enterprise capability for continuous service delivery that enables clients to seize market opportunities and reduce time to customer feedback.
- DevOps accelerates software delivery and speeds up the time to get the value out of the product.
- DevOps helps in balancing speed, cost, quality and risk and enhances the capacity to innovate.
- Adopting DevOps, improves the customer experience by reducing the time to customer feedback.



In your first semester, you would've learnt the basics of DevOps and the important concepts associated with it. This section is to give you a quick recap of DevOps. DevOps minimizes the friction and fills the gap between the development and operations team, ensuring smoother development and delivery. DevOps unites people, processes and products, that enables continuous delivery of value to the customers. Traditional IT organizations work in a siloed way, that impacts the development and delivery. DevOps enables creation of multidisciplinary teams that work together and create products efficiently.

Agile planning, continuous integration, delivery and deployment are some of the essential DevOps practices.

DevOps gives some great advantages over traditional setup. Adopting DevOps is important for the following reasons:

- Time for development can be estimated with more accuracy
- Development cycle time is reduced to a great extent
- Having the right data will help take correct business decisions
- With DevOps, teams fail fast and learn fast
- With more experimentation, learning can be validated and optimized

Let's now look at the ways of achieving DevOps in an organization.

1.2 Achieving DevOps

1. Continuous integration
2. Continuous delivery
3. Version control
4. Agile planning and lean project management
5. Monitoring and logging
6. Public and hybrid clouds
7. Infrastructure as code
8. Microservices
9. Containers

This section is to explain the ways through which teams can achieve DevOps.

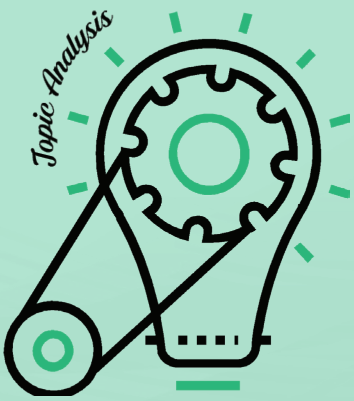
1. **Continuous integration:** Continuous integration enables merging and testing of code in a parallel fashion, this helps in early bug detection and recovery. Merge conflicts are also reduced to a great extent and this also reduces the feedback time. We'll learn more about continuous integration in the upcoming sections.

2. **Continuous delivery:** Continuous delivery of multiple versions to production and testing environments aid in bug fixing at early stages and changes can be accommodated in any stage of development.
3. **Version control:** Version control is extremely helpful for teams working in a distributed fashion. Teams can communicate efficiently during development. Version control systems also help in the efficient integration with software development tools for monitoring activities for deployment.
4. **Agile planning and lean project management:** Agile is all about developing software iteratively. Agile methodologies break the development process into sprints and deliver software high quality software with reduced bugs in a shorter period of time. Lean ways of project management help in efficient team management and teams can also adapt quickly to changing business requirements.
5. **Monitoring and logging:** Monitoring and Logging of running applications including production environments for application health as well as customer usage, helps organizations form a hypothesis and quickly validate or disprove strategies. Rich data are captured and stored in various logging formats.
6. **Public and hybrid clouds:** Cloud, with its deployment models SaaS (software as a service), PaaS (platform as a service), infrastructure as a service (IaaS) has largely helped organizations deploy their infrastructure easily and organizations can focus on the actual development with minimal worries on deployment.
7. **Infrastructure as code:** Considering the infrastructure as code is one of the important ways of achieving DevOps. This practice enables the automation and validation of creation and dismantling of environments for a stable and secure platform to host the applications.
8. **Microservices:** Microservices architecture is the way of breaking the application down to multiple services, depending on the business use case. These services are reusable, by which scalability and efficiency can be achieved.
9. **Containers:** Containers are an emerging way of virtualization. They are more efficient than virtual machines. Containers are also easily configurable.

One of the major goals of DevOps is reducing the time of the development cycle. The above mentioned practices not only help to reduce the development timeline, but also help in delivering multiple features with very minimal bugs.

We'll now look into continuous practices and continuous integration in detail.

What did You Grasp?



1. State True or False.

   Teams in organizations that have adopted DevOps work in a siloed fashion.

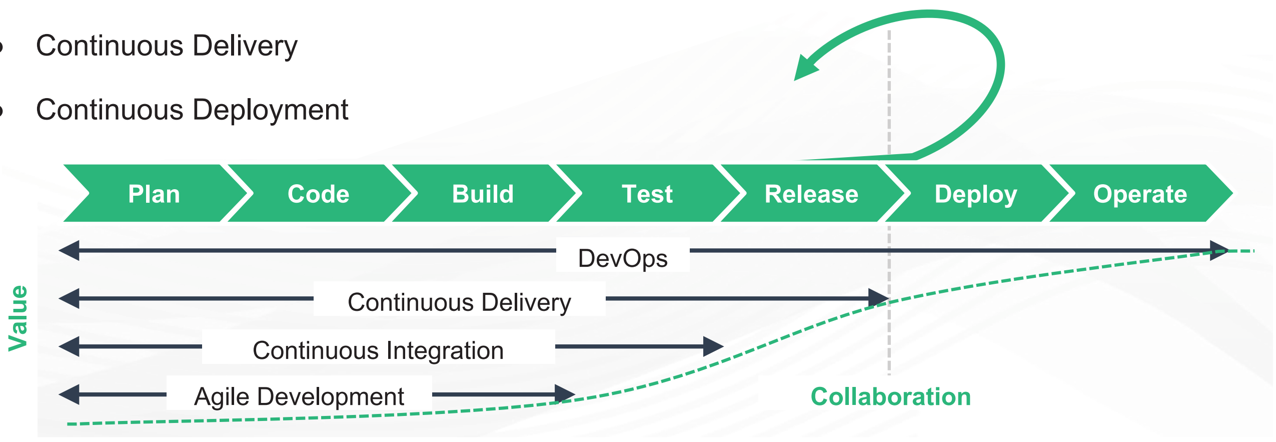
   A) True
   B) False

2.1 Continuous Practices

Continuous processes are the software development practices that help organizations develop and deliver software frequently and reliably. Continuous practices are highly correlated.

Following are the important continuous practices:

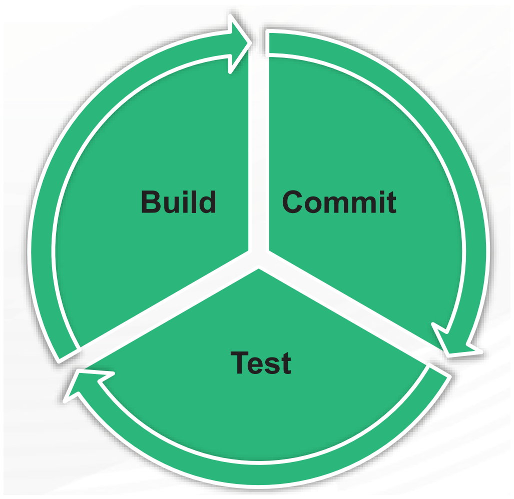
- Continuous Integration
- Continuous Delivery
- Continuous Deployment



In general, continuous practices are the software development practices that help teams develop and deliver better quality software frequently and reliably. To meet the increased competition in the market, organizations have started paying attention to continuous practices for developing and delivering high quality software at an accelerated pace. Continuous Integration (CI), Continuous Delivery (CDE), and Continuous Deployment (CD) are some of the continuous practices aimed at helping organizations to accelerate their development and delivery of software features without compromising quality. CI advocates integrating work-in-progress multiple times per day, CDE and CD are about ability to quickly and reliably release values to customers by bringing automation support as much as possible.

2.2 Continuous Integration (CI)

- Continuous integration is a practice that focuses on preparing for an early release.
- According to Martin Fowler, one of the authors of Agile manifesto, "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day."
- Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.



You would have learnt about continuous integration in your first semester. Now, we'll dive deep into continuous integration, the practices associated with it and the tools.

According to Martin Fowler, one of the authors of Agile manifesto, describes CI as follows: "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."

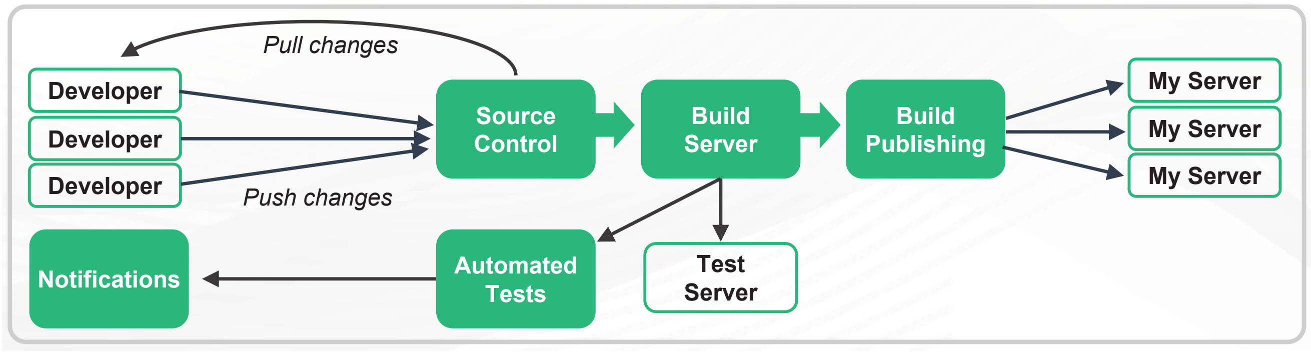
Amazon web services defines CI as follows: Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.

2.3 How does CI Work?

Following is the working mechanism of CI:

- In a continuous integration process, developers commit the code in a shared repository in frequent intervals. A source code version control system like Git is used.
- Unit tests are run locally before every commit, to ensure the proper functioning of the code, before it is integrated.

- Any continuous integration service will automatically build and unit tests are also run on newly checked-in code, to identify errors quickly.



Traditionally, developers worked in isolation for a long time and the developed code was integrated into the code repository only after the complete process is done. This became a tedious process and there were a lot of merge conflicts, which consumed a lot of time. Most importantly, bugs kept on accumulating and developers were able to identify and fix them only at a later stage, because unit tests were not implemented. These bugs came as a roadblock on the way to deliver updates to customers quickly.

Continuous integration refers to the build or integration stage of the software release process and entails both an automation component (e.g., a CI or build service) and a cultural component (e.g., learning to integrate frequently).

To begin with, a developer takes a copy of the source code from a centralized repository or a source code version control system like Git, on his local development environment. This source code version control system keeps an up to date version of the source code committed by multiple developers. From the version control system, the 'mainline' (the current version) is checked out to the local development machine, which will be the 'working copy' of the code. It is in this working copy, all the updates are done, and checked back into the version control system.

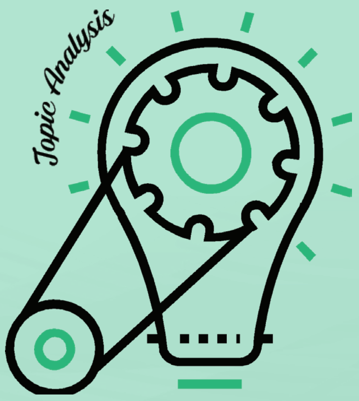
Once the updates are done, an automated build is carried out locally, which compiles the source code and run automated tests are done to make sure that the software or the 'build' is error-free. Only if the build passes the tests the changes can be committed to the centralized repository again. The important point to note here is that, other developers also might have updated the mainline with their changes. This will lead to merge conflicts, and it is the responsibility of the individual developer to update his working copy according to the mainline and fix the conflicts.

Only after the working copy is completely synchronized with the current version of mainline, it can be committed to the repository. The process doesn't end with committing the updated code. A build is again carried out on this integrated version, along with automated tests. Only if this builds passes the tests, the build is said to have succeeded. The integration process gets completed only if the committed changes build successfully.

A clash usually occurs when a second developer builds the commit. Otherwise, the integrated build will fail. In both these cases, errors are identified rapidly and it is very important to fix these errors quickly and make sure that the integrated build works properly. In a continuous integration environment, a failed build should not be kept in that state for a long time.

The result of this process is a stable piece of high quality software with very few bugs. Any DevOps organization will have multiple correct builds in a given day.

What did You Grasp?



1. Which of the following statements is true?
   A) CI is about doing frequent releases to production
   B) There is a cultural component involved in CI
   C) Updates are directly done to the mainline
   D) Integration process is completed as soon as the changes are merged into the mainline

3.1 Continuous Integration Practices

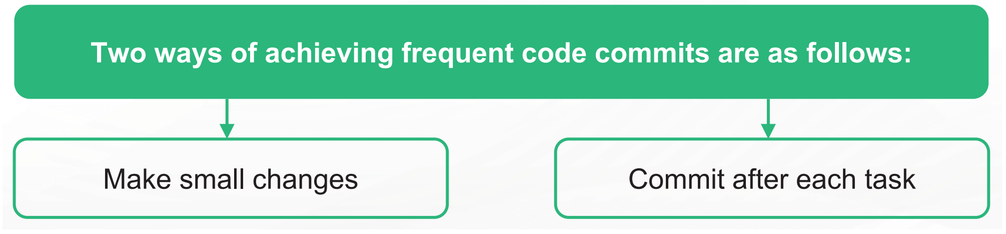
Some of the continuous integration practices are listed below:
Commit code frequently
Maintain a single source repository
Don't commit broken code
Automate the build
Every commit should build the mainline on an integration machine
Fix broken builds immediately
Keep the build fast
Write automated developer tests
All tests and inspections must pass
Run private builds
Avoid getting broken code
Automate deployment

Note down the best practices to be followed in continuous integration.

3.1.1 Commit Code Frequently

- Integrate early and often is one of the central tenets of CI.
- Commit code to your version control repository, at least once a day.

- Waiting for more than a day to commit the source code consumes a lot of time and prevents other developers from using the most updated version of the source code.



Everyone committing at the same time may lead to confusions and conflicts. Build errors may occur because of the collision between the changes.

One of the central tenets of CI is integrating early and often. Developers must commit code frequently in order to reap the complete benefits of CI. Waiting more than a day or so to commit code to the version control repository makes integration time-consuming and may prevent developers from being able to use the latest changes.

One of the following ways can be followed to commit code more frequently:

- **Make small changes:** Try not to change many components, all at once. Instead, choose a small task, write the tests and source code, run your tests, and then commit your code to the version control repository.
- **Commit after each task:** Assuming tasks/work items have been broken up so that they can be finished in a few hours, some development shops require developers to commit their code as they complete each task.

Try to avoid having everyone commit at the same time every day. There will be many more build errors to manage because of the collisions between changes. This is especially troublesome at the end of the day, when people are ready to leave. The longer you wait to integrate with others, the more difficult your integration will prove to be.

3.1.2 Maintain a Single Source Repository

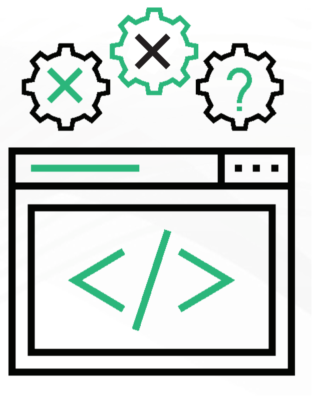
- Use a source code management system that serves as a centralized source code repository.
- Source code management tools are available to maintain the code in a single centralized location, from where developers can check out the latest version of the mainline build.
- Everything that a developer creates should be pushed to this source code repository, including: test scripts, properties files, database schema, install scripts, and third party libraries.
- Most importantly, there should be a mainline, a single branch of the project that is currently under development.

Software teams today work in a distributed fashion. When multiple people build individual pieces of code, it is important that the entire team has access to the source code and the complete version of the software is up to date. Teams have built several tools to manage this. These tools are collectively referred to as source code management tools. Everyone in the team should be aware of the functionalities of the source code management system. The source code repository is not only meant for the code, but developers should put in everything that is required for a build, including test scripts, properties files, database schema, installation scripts ad third party libraries.

Version control systems allow us to create multiple branches of the code, that can handle multiple streams of development. This feature should be used judiciously, as too many branches will put people in trouble. It is essential that there is be a single branch of the project that is currently under development, the mainline.

In short, developers should store in the source control management system, everything they need to build.

3.1.3 Don't Commit Broken Code



- Don't commit code that does not compile with other code or fails a test.
- Everyone in the development team should be aware that they should not commit code that doesn't work.
- Committing broken or non-functional code will affect the code in integration machine, and fellow developers will not get the correct version of the code.
- Every developer should follow the practice of carrying out a private build before checking the code into the centralized repository.

A dangerous assumption on a project is that everyone knows not to commit code that doesn't work to the version control repository. The ultimate mitigation for this risk is having a well-factored build script that compiles and tests the code in a repeatable manner.

Developers should always follow the practice of carrying out a private build (which closely resembles the integration build process) before committing code to the version control repository.

3.1.4 Automate the Build

- The build process has to be automated in a CI environment, which help build and launch the system using a single command.
- The command will run the build scripts necessary for automating the build process.
- A build process should include everything including getting the database schema from the repository and launching it in the production environment.

- Most of the development environments (IDEs) that we use these days have built management systems, but these are proprietary to the IDE and are fragile.
- Examples for build tools are Ant, Ruby Rake, MSBuild, etc.

The build process includes source code compilation, gathering of files, loading of schemas on to the databases, etc. Working on the build process manually can be a complicated and time consuming task. The build process has to be automated such that the entire process can be carried out using scripts that are executed by running a single command.

A build process should include everything including getting the database schema from the repository and launching it in the production environment. A build script allows us to do different things based on the use case. Many of the integrated development environments (IDEs) have some sort of a build management system, but these are proprietary to the IDE and could be fragile. IDE users can set up their own projects and use them for individual development. It is important to have a master build that is used on a server and is run using other scripts.

3.1.5 Keep the Build Fast

- Any build that takes an hour is considered unreasonable. Extreme programming (XP) has set a guideline that every build should take not more than 10 minutes.
- The important step in getting a faster build is setting up a deployment pipeline, which enables multiple builds in a sequence.
- The first build, also called commit build is the one that is triggered when a developer commits to the mainline. The commit build is done so quickly that it may bypass bugs and a balanced approach is necessary to identify bugs and gain speed.
- A two-stage deployment pipeline can do compilation and run multiple localized tests, and at the same time it adheres to the 10-minute guideline.

The major idea behind continuous integration is to provide rapid feedback. If a build takes a long time, rapid feedback is almost impossible. Any build that takes one hour is considered unreasonable. Extreme programming (XP) has set a guideline that the perfect time for a build is 10 minutes. If you have a 1-hour build, reducing it to 10 minutes may be a daunting task. In most of the cases testing takes much of the time.

The most crucial step is to start working on setting up a deployment pipeline. The idea behind a deployment pipeline (also known as build pipeline or staged build) is that there are in fact multiple builds done in sequence. The commit to the mainline triggers the first build - what I call the commit build. The commit build is the build that's needed when someone commits to the mainline. The commit build is the one that has to be done quickly, as a result, it will take a number of shortcuts that will reduce the ability to detect bugs. The trick is to balance the needs of bug finding and speed so that a good commit build is stable enough for other people to work on. Once the commit build is better than other people can work on the code with confidence.

Martin Fowler explains this in the following way:

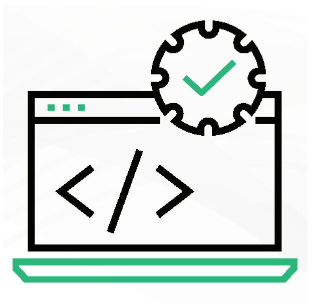
A simple example of this is a two stage deployment pipeline. The first stage would do the compilation and run tests that are more localized unit tests with the database completely stubbed out. Such tests can run very fast, keeping within the ten minute guideline. However, any bugs that involve larger scale interactions, particularly those involving the real database, won't be found. The second stage build runs a different suite of tests that do hit the real database and involve more end-to-end behavior. This suite might take a couple of hours to run.

In this scenario people use the first stage as the commit build and use this as their main CI cycle. The second-stage build runs when it can, picking up the executable from the latest good commit build for further testing. If this secondary build fails, then this may not have the same 'stop everything' quality, but the team does aim to fix such bugs as rapidly as possible, while keeping the commit build running. As in this example, later builds are often pure tests since these days, it's usually tests that cause the slowness.

If the secondary build detects a bug, that's a sign that the commit build could do with another test. As much as possible you want to ensure that any later-stage failure leads to new tests in the commit build that would have caught the bug, so the bug stays fixed in the commit build. This way, the commit tests are strengthened whenever something gets past them. There are cases where there's no way to build a fast-running test that exposes the bug, so you may decide to only test for that condition in the secondary build. Most of time, fortunately, you can add suitable tests to the commit build.

This example is of a two-stage pipeline, but the basic principle can be extended to any number of later stages. The builds after the commit build can also be done in parallel, so if you have two hours of secondary tests you can improve responsiveness by having two machines that run half the tests each. By using parallel secondary builds like this you can introduce all sorts of further automated testing, including performance testing, into the regular build process.

3.1.6 Every Commit Should Build the Mainline



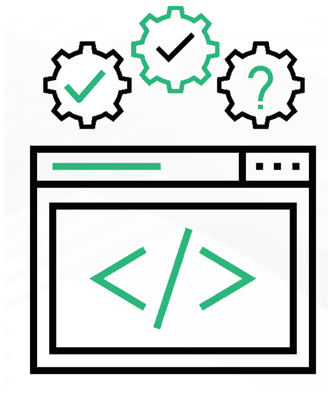
- Regular builds should happen on the integration machine and the build should succeed in order for the commit to be considered done.
- The developer who commits the latest code should be responsible for monitoring the mainline build, so that they can fix if anything breaks.
- A CI server acts as a monitor the repository. Every time the commit gets done, the server checks the source code onto the integration machine and initiates a build.
- The result of the build process is also notified to the committer through email.
- Regular builds should also ensure that the problems are identified well in the early.

Before every commit, developers should update and build. Regular builds should happen on integration machine and a commit is considered done only if the build succeeds. The developer who does the commit is responsible for the mainline build, so that they can track and fix if there is any breakage to the code. Manual build process is simple that a developer checks the mainline and starts the integration process. Continuous integration servers act as a monitor to the repository. Once a commit happens, the server automatically checks out the source code onto the integration machine.

Doing a regular build at night is not an advisable process, as the build stays unchecked for a long time, as a result bugs lie undetected for the whole day, before anyone discovers them. This makes the idea of continuous integration pointless, as it is all about early fault detection.

3.1.7 Fix Broken Builds Immediately

- A broken build is anything that prevents the build from reporting success.
- Though it is common for a mainline build to break, the key part is that any error in the mainline build has to be fixed right away.
- Although it's the team's responsibility, the developer who recently committed code must be involved in fixing the failed build.
- The best way to fix a broken mainline build is to revert the most recent commit from the mainline.
- It is not advisable to debug on a broken mainline, unless the cause of breakage is very obvious. Debugging should always be done in the local development environment after reverting to the latest known good build.

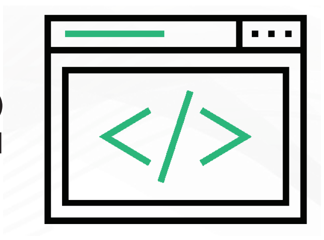


A broken build is anything that prevents the build from reporting success. This may be a compilation error, a failed test or inspection, a problem with the database, or a failed deployment. When operating in a CI environment, these problems must be fixed immediately; fortunately, in a CI environment, each error is discovered incrementally and therefore is likely very small. The most important aspect of CI is that one should always develop on a stable version of the code.

Though mainline build breakage is common, the key is to fix the issue right away. The project culture should convey that fixing a broken build is a top project priority. According to Kent Beck, the creator of Extreme Programming "nobody has a higher priority task than fixing the build". Not everyone in the team should involve in fixing the breakage, and it usually takes a couple of developers to fix it. The developer who recently committed the code should ideally take the responsibility. In any case, build fix has to be taken as a high priority task.

3.1.8 Write Automated Developer Tests

- A build should be fully automated. In order to run tests for a CI system, the tests must be automated.
- Verify that the software works, using automated developer tests. Run these tests with your automated build and run them often with CI.
- Testing can catch a lot of bugs, even from a program that runs.
- Methods like extreme programming and test-driven development (TDD) have put a lot of emphasis on self-testing code and for this, automated test can check a large part of code for bugs.

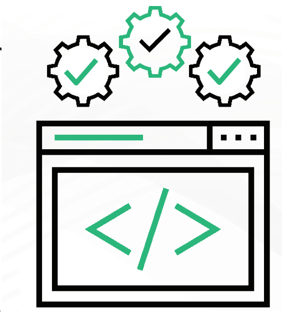


- Writing tests in an xUnit framework such as NUnit or JUnit will provide the capability of running these tests in an automated fashion.

A build includes compiling, linking and all the additional processes that are required by a program to execute. A program may run, but it may still contain bugs or errors. Testing is such a process that identifies errors even from a program that executes, i.e., a working program. Including automated tests in the build process can identify lots of such bugs, despite the program being complex. In order to run tests for a CI system, the tests must be automated. Writing the tests in an xUnit framework such as NUnit or JUnit will provide the capability of running these tests in an automated fashion.

Methods like extreme programming and test-driven development (TDD) have put a lot of emphasis on self-testing code. A suite of automated tests is needed for a self-testing code, that can check large chunks of code for bugs. TDD has popularized the xUnit family of testing tools. Though tests cannot guarantee a 100% clean code, they certainly guarantee a high-quality code with very few bugs.

3.1.9 All tests and Inspections must Pass



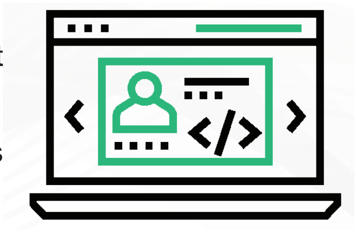
- In a CI environment, it is very important that 100% of tests must pass prior to committing code to the version control repository.
- Automated tests are as important as the compilation process.
- Keeping the mainline with code that doesn't pass the automated tests can result in low-quality software.
- Coverage tools help in identifying the source code that doesn't have a corresponding tool. This ensures that the entire code undergoes the testing phase.

In a CI environment, the technical criterion is that 100% of a project's automated test must pass for the build to pass. Automated tests are as important as the compilation process. Everyone accepts that code that does not compile will not work; therefore, code that has test errors will not work either. Accepting code that does not pass the tests can lead to lower-quality software.

Coverage tools assist in pinpointing source code that does not have a corresponding test. You can run a code coverage tool as part of an integration build. The same goes for running automated software inspectors. A general rule set of coding and design standards that all code must pass, can be used. More advanced inspections may be added that don't fail the build, but areas of the code that should be investigated, have to be identified.

3.1.10 Run Private Builds

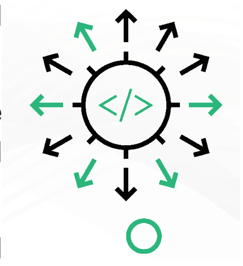
- To prevent integration failures, changes from other developers have to be received by getting the latest changes from the repository and running a full integration build, locally known as a private system build.



- Developers should run an integration build in their local development environment after the code undergoes unit tests.
- Code that the developer commits to the integration build server is less likely to fail, if it undergoes a private system build.

To prevent broken builds, developers should emulate an integration build on their local workstation IDE after completing their unit tests. This build allows you to integrate your new working software with the working software from all the other developers, obtaining the changes from the version control repository and successfully building locally with the recent changes. Thus, the code each developer commits will be less likely to fail on the integration build server.

3.1.11 Avoid getting Broken Code



- Getting a failed build from the repository means the latest code is not checked out from the repository.
- This consumes a lot of time and effort, as the developer has to do some workaround to fix the error that has caused the build failure for compiling and testing the code.
- Developers can wait for the change or help the other developer(s) fix the build failure and then get the latest code.
- Though it is the responsibility of the team, the developer(s) who caused the breakage should fix the bug, for the build to work fine.

When the build is broken, one should not check out the latest code from the version control repository. Otherwise, the developer will end up spending a lot of time, developing a workaround to the error that has caused the failure, just to compile and test the code. Ultimately, it's the responsibility of the team to fix, but the developers responsible for breaking the build should already be working on fixing their code and committing it back to the version control repository. Sometimes a developer may not have seen the email on the broken build. It is critical that all developers know the state of the code in the version control repository.

3.1.12 Automate Deployment

- Automating the deployment is similar to automated build, we have a script that does the deployment automatically into any environment.
- Automated deployment helps speed up the process and reduce errors.
- Automated deployment is an economically viable option because it requires the same capabilities that are required to deploy in a test environment.
- There should be a provision for automated rollback, since failures can happen at any point in time.
- For testing new features or user interfaces, it is always suggested to deploy trial build deployed to a subset of users, who will then decide if it can be deployed to all users.

In continuous integration, there will be multiple environments and the artifacts need to be moved across these environments many times a day. It is quintessential to do these deployments automatically into any environment. You need to have scripts to do this automated deployment in the test and production environments with similar ease. Deployment to production environment may happen everyday, but having automated deployment improves speed and reduces errors.

Automated deployment is also an economically viable option, since the same capabilities are required to deploy in test and production. If the application is deployed in production, there should be a capability for an automated rollback, since failures are anticipated at any point in time. Automated rollback also reduces the tension on deployment and enables frequent deployments without the fear of failure.

3.2 Benefits of Continuous Integration

BENEFITS

Following are the benefits of continuous integration:

- The foremost benefit of CI is that it reduces risk to a great extent.
- With CI there always exists the clarity of the current status, what works and what doesn't and the bugs in the software.
- CI doesn't eliminate bugs completely, but it can make them extremely easier to identify and fix.
- CI makes the process of frequent deployment seamless and as hassle-free as possible.
- Continuous integration complements other software development practices like scrum, rational unified process, XP, etc.

So far we have looked at the best practices of continuous integration. Let's now turn our focus towards the benefits.

- Continuous integration reduces risk of failure to a great extent and this is the first and foremost benefit that CI offers.
- At each and every point in time, especially in the case of long projects, the team has a clarity of where the project stands, what will work and what will not and the status of the bugs in the software.
- With CI you cannot eliminate bugs completely, but bugs can be identified easily and fixed.
- One of the major advantages of CI is that it enables frequent deployments as seamless as possible.
- CI complements other software development practices like scrum, rational unified process, XP, etc., in the following ways:
    - **Developer testing:** Developers who write tests most often use some xUnit-based framework such as JUnit or NUnit. These tests can be automatically executed from the build script. Since the practice of CI advocates that builds be run any time a change is made to the software, and that the automated tests are a part of these builds, CI enables automated regression tests to be run on the entire code base whenever a change is applied to the software.
    - **Coding standard adherence:** A coding standard is the set of guidelines that developers must adhere to on a project. On many projects, ensuring adherence is largely a manual process that is performed by a code review. CI can run a build script to report on adherence to the coding standards by running a suite of automated static analysis tools that inspect the source code against the established standard whenever a change is applied.

- **Refactoring:** As Fowler states, refactoring is “the process of changing the software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” Among other benefits, this makes the code easier to maintain. CI can assist with refactoring by running inspection tools that identify potential problem areas at every build.
- **Small releases:** This practice allows testers and users to get working software to use and review as often as required. CI works very well with this practice, because software integration is occurring many times a day and a release is available at virtually any time. Once a CI system is in place, a release can be generated with minimal effort.
- **Collective ownership:** Any developer can work on any part of the software system. This prevents ‘knowledge silos’, where there is only one person who has knowledge of a particular area of the system. The practice of CI can help with collective ownership by ensuring adherence to coding standards and the running of regression tests on a continual basis.

We’ve learnt the important concepts of CI and the benefits of implementing it. We’ll now move onto continuous deployment. Continuous delivery and deployment are almost similar processes with slight differences

What did You Grasp?

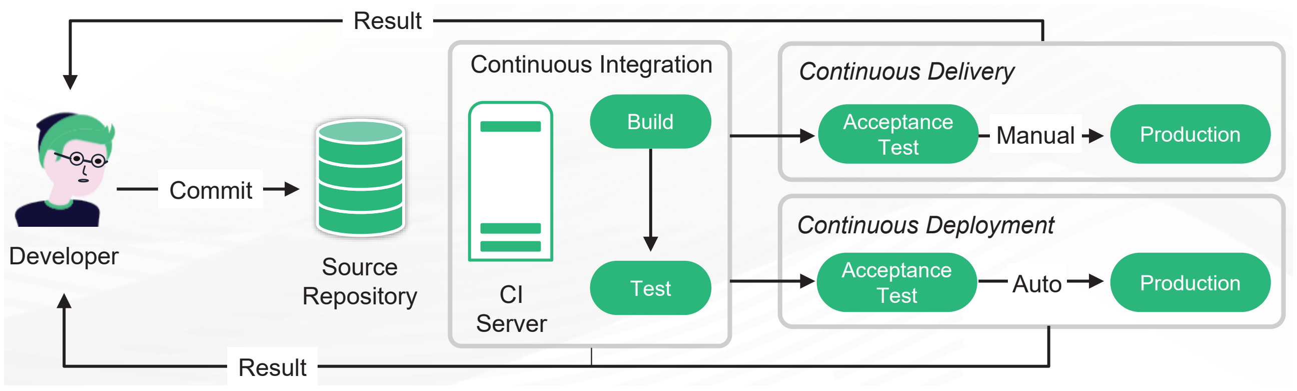
1. Which of the following is NOT a CI best practice?
   A) Builds should be fast
   B) Fix broken builds after a considerable number of bugs are accumulated
   C) It is necessary that all the tests are passed
   D) Run private builds

2. State True or False.
   A self testing code requires an automated test suite.
   A) True
   B) False

4.1 A Quick Recap of Continuous Delivery

- Continuous Delivery (CD) is the process to build, test, configure and deploy from a build to a production environment.
- CDE employs a set of practices, e.g., CI, and deployment automation to deliver software automatically to a production-like environment.

The figure illustrates how the application moves from continuous delivery or deployment after continuous integration.



You would have learnt about continuous delivery in your first semester. Now, in order to help you understand continuous deployment better, a recap of continuous delivery is given here.

Continuous delivery is an extension to continuous integration, practicing which will ensure that new changes are released to the customers in a quick and sustainable fashion. Continuous Delivery (CDE) is aimed at ensuring an application is always in production-ready state after successfully passing automated tests and quality checks. CDE employs a set of practices, e.g., CI, and deployment automation to deliver software automatically to a production-like environment. This practice offers several benefits such as reduced deployment risk, lower costs and getting user feedback faster. The figure above indicates that having a continuous delivery practice requires continuous integration practice.

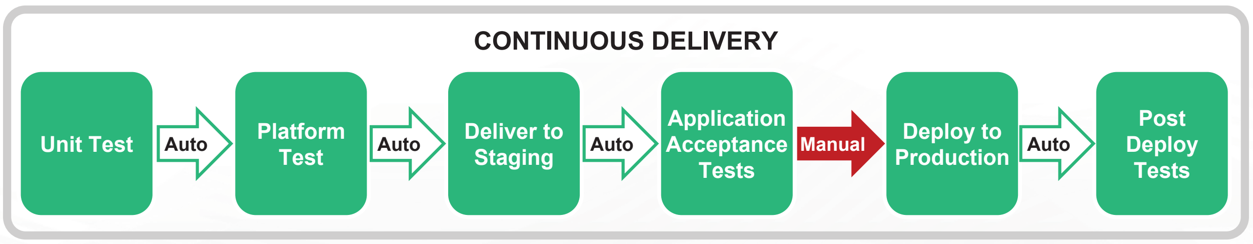
In continuous delivery, the code changes are automatically built, tested and prepared for release to the production. All code changes are deployed to a testing and/or production environment after the build is completed. If implemented properly, continuous delivery will help developers have a deployment-ready build artifact that has already passed through a set of standard testing processes.

Continuous delivery lets the developers do automated testing apart from unit tests, which help them verify the application updates across multiple dimensions before it is deployed to customers. These tests may include UI testing, load testing, integration testing, API reliability testing, etc. By means of these testing processes, developers can detect issues early.

With the advent of cloud, automating the creation and replication of multiple environments for testing has become easy and cost-effective, which were earlier difficult to do on-premise.

4.2 Continuous Delivery Process

The figure illustrates the processes involved in continuous delivery.



Continuous delivery automates the complete software release process. Every change made to code enables an automated process to build, test and stage the update. The final step of deploying the update to a production environment is done manually by the developer.

Once the code is committed, automated testing processes are triggered. Tests like unit tests and platform test are done and the first update is deployed in a staging environment. Once delivered to staging, application acceptance tests are done and the code is then deployed to production. There can be multiple, parallel test stages before a production deployment.

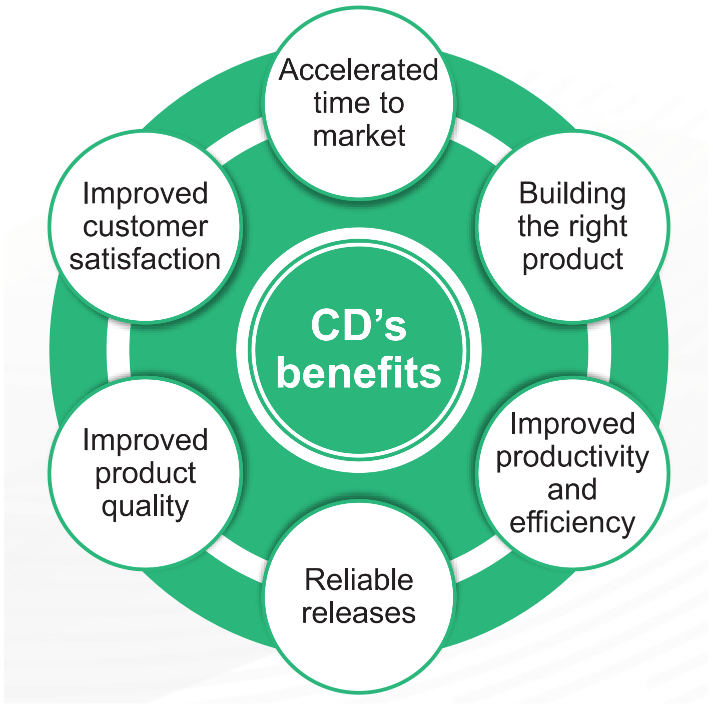
Till the process of application acceptance tests, all the steps are automatically carried out. The final step of deployment is manually triggered by the developer and again post-deployment tests are done automatically.

Being a lean process, the goal of the CD is to keep the production fresh by achieving the shortest path from coding to deployment.

Now that we have understood continuous delivery, we'll now look at how continuous deployment works and what is common between these two practices and what is not.

4.3 Benefits of Continuous Delivery

The following figure illustrates the benefits of continuous delivery practices.


CD's benefits
Accelerated time to market
Building the right product
Improved productivity and efficiency
Reliable releases
Improved product quality
Improved customer satisfaction

Continuous delivery practices offer innumerable benefits. The following are the six major benefits of continuous delivery.

1. **Accelerated Time to Market:** Continuous delivery practices increase the release frequency dramatically. The cycle time from a user story conception to production also decrease significantly. With CDE, business value can be delivered to customers very quickly.

2. **Building the Right Product:** Frequent releases let the application development teams obtain user feedback more quickly. The team's time is thus spent only on building the useful features, hence the right product. Without CDE, discrimination between the useful and not so useful features take plenty of time.

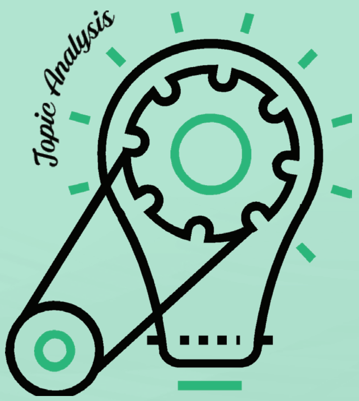
3. **Improved Productivity and Efficiency:** CDE improves productivity and efficiency of the development team. The CD pipeline automatically sets up the environments for development, testing, hence enhanced productivity. Human intervention is only required for approving the release of the application to production. CDE pipeline significantly reduces the time spent on troubleshooting and fixing the issues.

4. **Reliable Releases:** The release process becomes stable, as the risks associated with a release are significantly decreased in a CDE pipeline. The deployment process and scripts are tested repeatedly before deployment to production. So, most errors in the deployment process and scripts are discovered in early stages. As there are frequent releases, the number of code changes in each release will be very less. This makes identifying and fixing the issues a lot easier. The pipeline can automatically roll back a release if it fails. This further reduces the risk of a release failure.

5. **Improved Product Quality:** Continuous delivery improves the product quality significantly, as there are fewer bugs. The whole code base undergoes a series of tests and bugs are identified and fixed before moving to the next feature. This reduces the accumulation of bugs, which is a common thing in traditional practices.

6. **Improved Customer Satisfaction:** As better quality software is delivered, customer satisfaction improves, which is a critical metric for a successful business.

What did You Grasp?



1. State True or False.

In CDE, post-deployment tests are carried out manually.

A) True
B) False

5.1 Continuous Deployment

Following are the key details of continuous deployment:

- Continuous deployment is a bit more advanced than continuous delivery, that ensures that every change that passes all stages of the production pipeline is released to the customers automatically.
- The major difference between continuous delivery and deployment is that, in continuous delivery, release to production needs manual approval, it is done automatically in continuous deployment.
- Automatic deployment to production enables multiple production deployments a day with reduced lead time.
- With CD, changes are not stagnated in the development environment for a long time.
- While continuous delivery ensures that the code is released-ready, continuous deployment is the actual delivery of features to customers.

Continuous deployment is also an extension of continuous integration, which reduces the time between development and deployment by means of automated deployment. Continuous delivery is most popular in SaaS application, as they can be easily updated in the background.

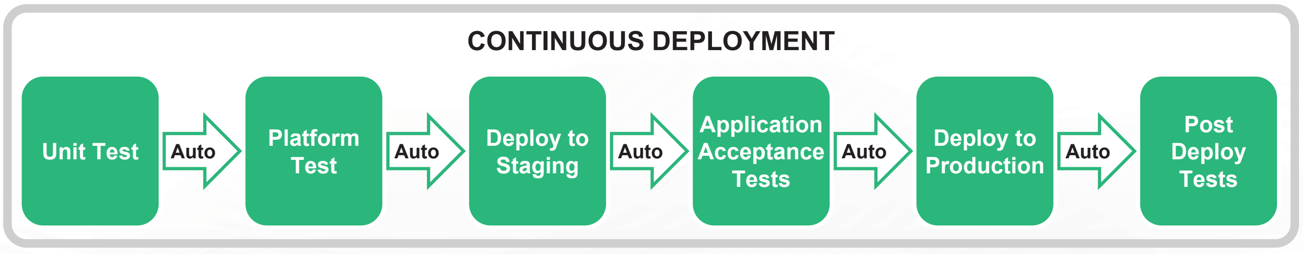
Continuous Deployment (CD) practice goes a step further than continuous delivery and automatically and continuously deploys the application to production or customer environments. There is a robust debate in academic and industrial circles about defining and distinguishing between continuous deployment and continuous delivery.

What primarily differentiates continuous deployment from continuous delivery is a production environment (i.e., actual customers); the goal of continuous deployment practice is to automatically and steadily deploy every change in the production environment. It is important to note that CD practice implies CDE practice, but the converse is not true. Whilst the final deployment in CDE is a manual step, there should be no manual steps in the CD, in which as soon as developers commit a change, the change is deployed to production through a deployment pipeline.

CDE practice is a pull-based approach for which a business decides what and when to deploy; CD practice is a push-based approach. In other words, the scope of CDE does not include frequent and automated release, and the CD is consequently a continuation of CDE. Whilst CDE practice can be applied to all types of systems and organizations, CD practice may only be suitable for certain types of organizations or systems.

5.2 Continuous Deployment Process

The following figure illustrates how the CD pipeline is structured.



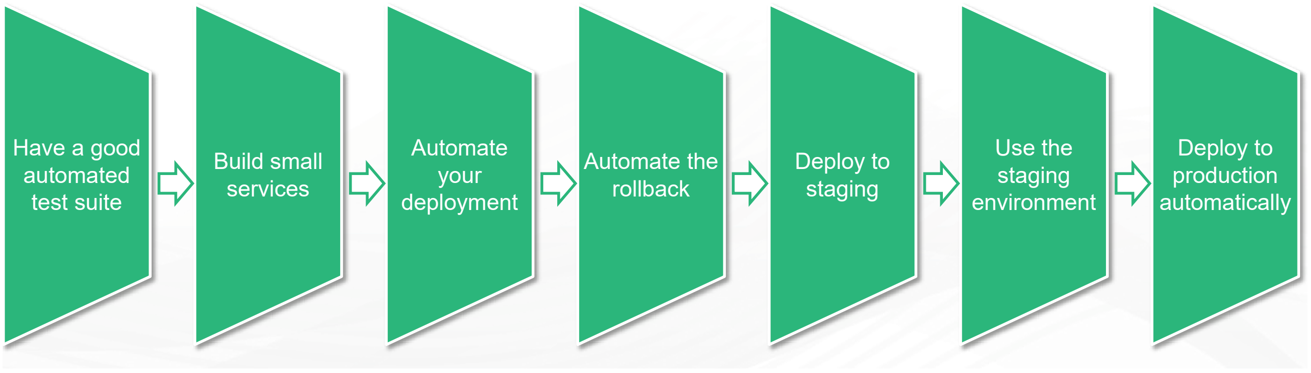
In continuous deployment every code change goes through the pipeline and the release to production is done automatically, so that there are multiple production deployments in a day. With the CD, you can be sure that the development process is very clear, with the main branch always in a stable releasable state.

Once the CI build succeeds, it enters the CD pipeline, where automated unit and platform tests are carried out. The application is then deployed to a staging environment and automated acceptance tests are carried out. Once the application passes the suite of these tests, it is then deployed to production automatically, without any manual approval.

With Continuous Deployment, any updated working version of the application is automatically pushed to production.

5.3 Best Practices of Continuous Deployment

The following practices are considered to be the best practices for continuous deployment:


Have a good automated test suite
Build small services
Automate your deployment
Automate the rollback
Deploy to staging
Use the staging environment
Deploy to production automatically

This section explains the steps to be followed to move to continuous deployment.

- **Have good automated test suite:** Teams should have an automated test suite. Every single piece of code need not be tested, but if the main branch features are not tested, it will screw up the entire development process. Without tests we'll end up with an application with bugs. Bugs will even break the features that were working in the past. The advantage of having a well-tested application is that changes can be introduced at any point in time, without the fear of breaking the existing code.

- **Build small services:** The application under development has to be split into multiple small parts, each solving a specific problem and can interact with other parts through clearly defined interfaces. This way new features can be introduced faster and the system will become stable and the team's workflow can also be improved.

- **Automate your deployment:** There should be one single command to deploy the application, all the deployment steps should be accessible through that command. That command should not require any additional configuration.

- **Automate the rollback:** A rollback process should be as seamless as possible such that it is as easy as releasing a new version. Reverting to an older version easily is very important from the database viewpoint. If it contains multiple steps, it will be hard to roll back.
- **Deploy to staging:** Continuous deployment to the production environment, if it is not doable, deployment to the staging environment should be done at least. Teams should make sure that the latest version of the application is safely running somewhere.
- **Use the staging environment:** It is always important to use the staging environment first before going for production. Any breakage can be identified firsthand and can be fixed immediately.
- **Deploy to production automatically:** If all the above steps are done, the application is ready to be deployed continuously in production. These steps ensure that only working code is deployed and even if there is any bug, it can be rolled back to the stable version in no time.

We've learnt about continuous integration, delivery and deployment. One important aspect of these continuous practices is that having a centralized source code repository and a version control system that keeps track of the changes that are made to the code. If anything doesn't work, we can always go back to most recent stable version.

We'll now look at version control systems.

5.4 Benefits of Continuous Deployment

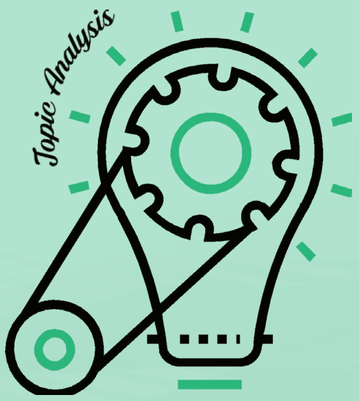
BENEFITS

Some of the benefits of continuous deployment, among others, are listed below:

- Eliminate manual intervention for Continuous Delivery and increase the focus on the product.
- Repetitive tasks are automated and focus can be on actual testing.
- Deployments are frictionless without compromising security.
- Scalability is achieved with automated processes.
- Existing tools and technologies (such as CI providers, DevOps tools, or scripts) can be connected to form a harmonious workflow.
- Integrate teams and processes with a unified pipeline.
- Software is delivered with fewer bugs and reduced risk.
- More features can be released to market very frequently.
- Both cloud-native and traditional applications can be shipped in a unified pipeline.
- Overall productivity is improved.

The above infographic lists the key benefits of CD. If executed well, continuous deployment practices reduce the time to market with increased quality and minimal risks.

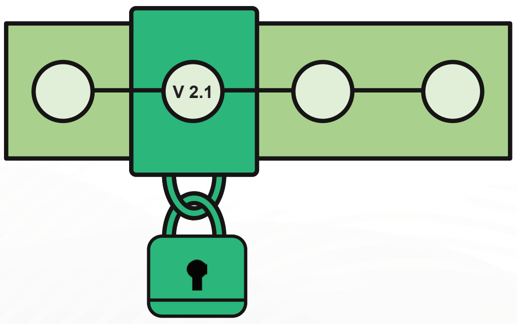
What did You Grasp?



1. Which of the following statements is/are true about continuous deployment?
   A) CD advocates the development of complex individual services
   B) There are some manual processes in CD pipeline
   C) In CD, a single script can deploy the application to production automatically
   D) CD is a pull-based approach

6.1 Version Control System (VCS)

- Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
- Generally, software source code files are version controlled and used in development, but in reality any type of file can be version controlled.



Version control systems offer the following important features:

- VCS allows to revert selected files back to a previous state
- The entire project can be reverted back to a previous state
- Changes can be compared over time
- Can check who last modified something that might be causing a problem
- Can see who introduced an issue and when

Version control systems (VCS) are a category of software tools that help a development team manage the changes to source code over time. Version control systems keep track of every change made to the code, in a special kind of database. As the code is edited, the version control system takes a snapshot of the files. The version control system saves that snapshot permanently so you can recall it later if you need it.

Whenever an error is encountered, the earlier version can be called back easily. VCS acts as a security guard to the source code repository, and minimizes the damage caused to it through technical glitches or human errors. Having a VCS is very critical for software teams. If any irreversible damage is done to the source code, then the complete project is lost. With VCS it is always easy to go back to the earlier stable version of the source code.

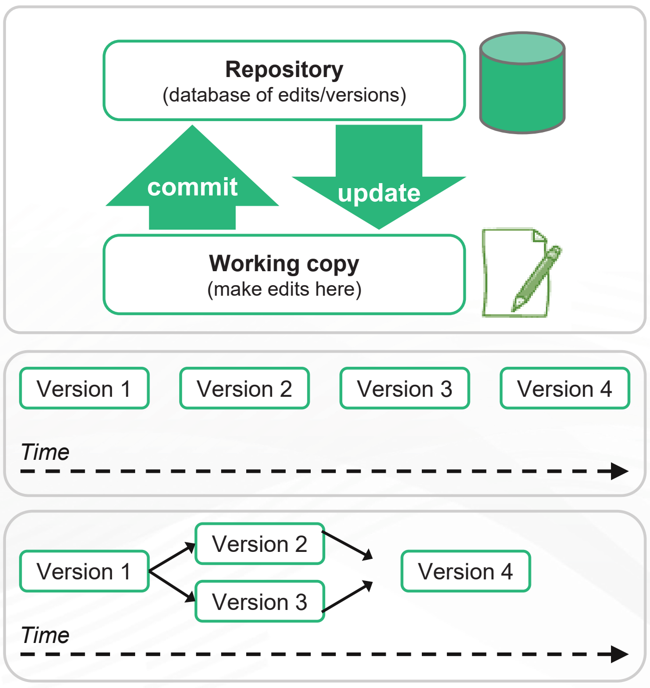
Software teams these days work in a distributed fashion. New code is written and appended to the source code continuously. Each developer may make changes to several branches of the source file tree. One may add a new feature, while another may fix a bug. VCS tracks the changes made by each and every developer and helps in preventing conflicts. Changes made to one of the branches may affect the code in any other part. With VCS these changes can be identified quickly and resolved.

Version control systems offer the following important features:

- VCS allows to revert selected files back to a previous state
- The entire project can be reverted back to a previous state
- Changes can be compared over time
- Can check who last modified something that might be causing a problem
- Can see who introduced an issue and when

6.2 Repository and Working Copy

- A repository is the master database of all edits to and the versions of the project.
- A working copy is the personal copy of all the files in the project. At any point in time, an individual developer makes changes only to the working copy that is checked out from the central repository to the local development environment.
- Changes made to the working copy are tested for stability and then committed to the repository. This way, other developers are not affected by the individual's changes.
- Changes made to the repository can either be linear or different edits are made simultaneously, that leads to branching.



Version control uses a repository, which is a database of changes and a working copy where the individual developer makes the changes.

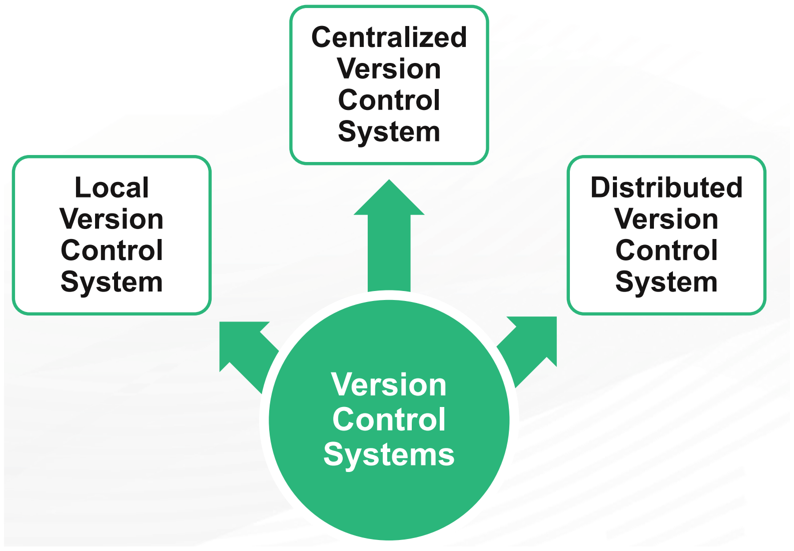
The working copy, also referred to as checkout, is the personal copy of all the files in the project. It is to this copy that the edits are made, without affecting the teammates. If edits are finalized, the changes are committed to the repository.

A repository is a database of all the edits to, and/or historical versions (snapshots) of the project. It is possible for the repository to contain edits that have not yet been applied to the working copy. The working copy can be updated to incorporate any new edits or versions that have been added to the repository since the last time it was updated.

In the simplest case, the database contains a linear history: each change is made after the previous one. Another possibility is that different users made edits simultaneously (this is sometimes called 'branching'). In that case, the version history splits and then merges again. Refer to the figure above to understand this concept.

6.3 Types of Version Control Systems

There are three types of version control systems as illustrated below:


Centralized Version Control System
Local Version Control System
Distributed Version Control System
Version Control Systems

There are three types of version control systems. They are as follows:

- **Local version control system:** Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over the files you don't mean to. To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes in files under revision control.
- **Centralized version control system:** The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion,

and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else are doing on the project. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save version changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

- **Distributed version control system:** In a Distributed Version Control System (DVCS), such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

  Furthermore, many of these systems deals pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

6.4 Benefits of Version Control Systems

BENEFITS

Version control systems enable teams to work together and ship the code in frequent intervals. Some of the benefits of the VCS are listed below:

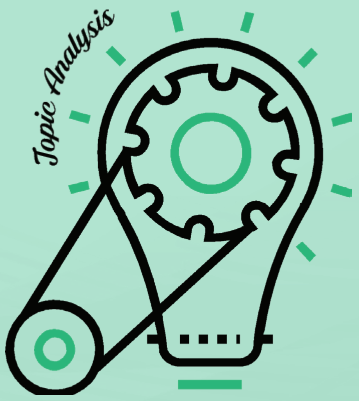
- Creation of workflows
- Working with versions
- Concurrent coding
- History of changes
- Automation of tasks

Version control systems offer numerous benefits to the development teams. Some of the important benefits of the VCS are given below.

- **Creation of workflows:** Version control workflows prevent the chaos of everyone using their own development process with different and incompatible tools. Version control systems provide process enforcement and permissions so everyone stays on the same page.

- **Working with versions:** Every version has a description for what the changes in the version do, such as fix a bug or add a feature. These descriptions help you follow changes in your code by version instead of by individual file changes. The code stored in versions can be viewed and restored from version control at any time as needed. This makes it easy to base new work of any version of the code.
- **Concurrent coding:** Version control synchronizes versions and makes sure that one change doesn't conflict with other changes from other developers. Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. In rare cases, when two people make conflicting edits to the same line of a file, then the version control system requests human assistance in deciding what to do.
- **History of changes:** Version control keeps a history of changes as your team saves new versions of your code. This history can be reviewed to find out who, why, and when changes were made. History gives you the confidence to experiment since you can roll back to a previous good version at any time. History lets you base work from any version of the code, such as to fix a bug in a previous release. Having access to historical versions is like insurance against computer crashes or data loss.
- **Automation of tasks:** Version control automation features save your team time and generate consistent results. Automate testing, code analysis, and deployment when new versions are saved in version control.

What did You Grasp?



1. State True or False.

   With VCS, only selected files can be reverted back to a previous state.

   A) True
   B) False

2. What is the database of all the edits and the versions of the source code called?

   A) Working copy
   B) Repository
   C) Checkout
   D) DBMS

3. Which of the following is a distributed version control system?
   A) Subversion
   B) Perforce
   C) Git
   D) CVS

In a nutshell, we learnt:

1. An overview of DevOps and the ways of achieving DevOps in an organization
2. An introduction to continuous practices
3. Introduction to Continuous integration (CI) and how it works
   - Best practices of CI
   - Benefits of CI
4. An overview of Continuous delivery and the process flow
5. Introduction to Continuous deployment with an emphasis on how it differs from continuous delivery
   - The CD process
   - Best practices of CD
6. Version control systems and the associated concepts
   - Types of version control systems
   - Benefits of having version control systems