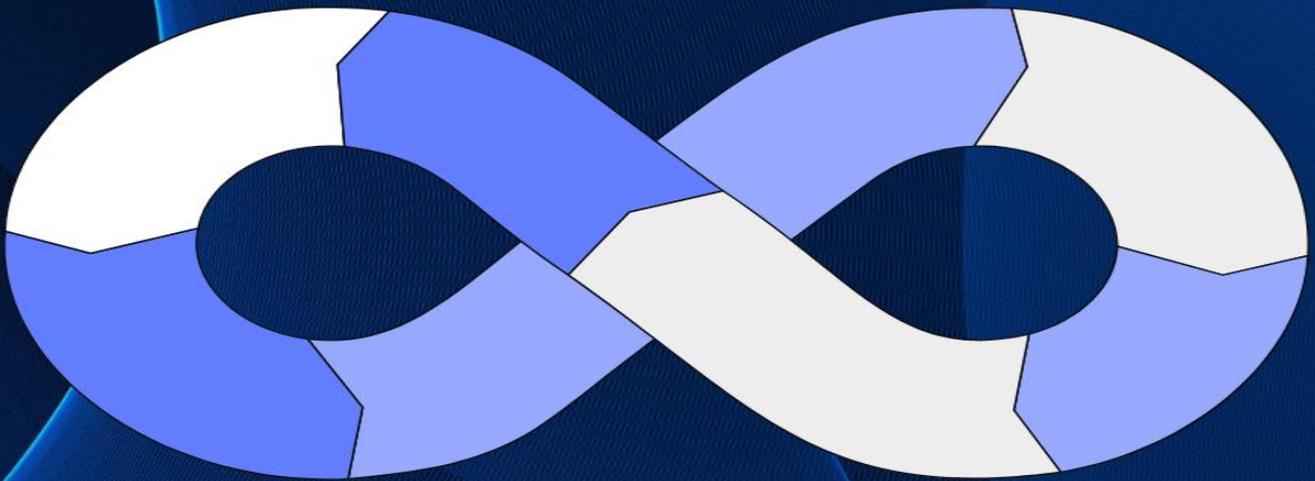


LEARN
in demand

DEVOPS



Jenkins - Docker - Ansible

1. Concepts of Continuous Integration

a. The agile software development process

- i. Software development life cycle

b. Continuous Integration

- i. Note
- ii. Agile runs on Continuous Integration

c. How to achieve Continuous Integration

- i. Development operations
- ii. Use a version control system
- iii. Use repository tools
- iv. Use a Continuous Integration tool
- v. Creating a self-triggered build
- vi. Automate the packaging
- vii. Using build tools
- viii. Automating the deployments
- ix. Note
- x. Automating the testing
- xi. Use static code analysis

d. Continuous Integration benefits

- i. Freedom from long integrations
- ii. Production-ready features
- iii. Analyzing and reporting
- iv. Catch issues faster
- v. Spend more time adding features
- vi. Rapid development

2. Setting up Jenkins

a. Introduction to Jenkins

- i. What is Jenkins made of?
- ii. Note
- iii. Why use Jenkins as a Continuous Integration server?
- iv. Jenkins as a centralized Continuous Integration server
- v. Hardware requirements

b. Running Jenkins inside a container

- i. Installing Jenkins as a service on the Apache Tomcat server
- ii. Note
- iii. Setting up the Jenkins home path
- iv. Why run Jenkins inside a container?
- v.
- vi. Setting up Jenkins on Ubuntu
- vii. Note
- viii. Setting up Jenkins on Fedora/Centos

- ix. Note
- x. Note
- xi. Tip

c. Sample use cases

- i. Netflix
- ii. Yahoo!
- iii. Note

3. Configuring Jenkins

a. Creating your first Jenkins job

- i. Note
- ii. Note
- iii. Adding a build step
- iv. Note
- v. Adding post-build actions
- vi. Configuring the Jenkins SMTP server
- vii. Running a Jenkins job
- viii. Note
- ix. Jenkins build log
- x. Note
- xi. Jenkins home directory

b. Jenkins backup and restore

- i. Creating a Jenkins job to take periodic backup
- ii. Restoring a Jenkins backup

c. Upgrading Jenkins

- i. Note
- ii. Upgrading Jenkins running on the Tomcat server
- iii. Upgrading standalone Jenkins master running on Ubuntu
- iv. Script to upgrade Jenkins on Ubuntu

d. Managing Jenkins plugins

- i. The Jenkins Plugins Manager
- ii. Installing a Jenkins plugin to take periodic backup
- iii. Configuring the periodic backup plugin
- iv. Note

e. User administration

- i. Enabling global security on Jenkins
- ii. Note
- iii. Using the Project-based Matrix Authorization Strategy

4. Continuous Integration Using Jenkins – Part I

a. Jenkins Continuous Integration Design



- i. The branching strategy
- ii. The Continuous Integration pipeline
- iii. Note
- iv. Toolset for Continuous Integration
- b. Setting up a version control system**
 - i. Installing Git
 - ii. Installing SourceTree (a Git client)
 - iii. Creating a repository inside Git
 - iv. Uploading code to Git repository
 - v. Configuring branches in Git
 - vi. Note
 - vii. Git cheat sheet
- c. Configuring Jenkins**
 - i. Installing the Git plugin
 - ii. Note
 - iii. Installing and configuring JDK
 - iv. Note
 - v. Installing and configuring Maven
 - vi. Installing the e-mail extension plugin
- d. The Jenkins pipeline to poll the feature branch**
 - i. Creating a Jenkins job to poll, build, and unit test code on the feature1 branch
 - ii. Note
 - iii. Creating a Jenkins job to merge code to the integration branch
 - iv. Creating a Jenkins job to poll, build, and unit test code on the feature2 branch
 - v. Creating a Jenkins job to merge code to the integration branch

5. Continuous Integration Using Jenkins – Part II

- a. Installing SonarQube to check code quality**
 - i. Setting the Sonar environment variables
 - ii. Running the SonarQube application
 - iii. Note
 - iv. Creating a project inside SonarQube
 - v. Installing the build breaker plugin for Sonar
 - vi. Creating quality gates
 - vii. Installing SonarQube Scanner
 - viii. Setting the Sonar Runner environment variables
- b. Installing Artifactory**
 - i. Setting the Artifactory environment variables
 - ii. Running the Artifactory application
 - iii. Creating a repository inside Artifactory
- c. Jenkins configuration**

- i. Installing the delivery pipeline plugin
 - ii. Installing the SonarQube plugin
 - iii. Note
 - iv. Installing the Artifactory plugin
 - d. The Jenkins pipeline to poll the integration branch**
 - i. Creating a Jenkins job to poll, build, perform static code analysis, and integration tests
 - ii. Creating a Jenkins job to upload code to Artifactory
 - iii. Note
 - iv. Note
- 6. Continuous Delivery Using Jenkins**
 - a. What is Continuous Delivery?**
 - b. Continuous Delivery Design**
 - c. Jenkins configuration**
 - i. Configuring Jenkins slaves on the testing server
 - ii. Note
 - iii. Slave agents via SSH tunneling
 - iv. Creating a Jenkins job to deploy code on the testing server

DOCKER

Table of Contents

The Basic Idea

- 1. Containerization versus virtualization**
 - a. Traditional virtualization
 - b. Containerization
- 2. Understanding Docker**
 - a. Difference between Docker and typical VMs
 - b. Dockerfile
 - c. Docker Networking/Linking
- 3. Docker installers/installation**
 - a. Types of Installers
 - b. Installing Docker on Ubuntu
 - c. Installing Docker on RedHat
- 4. Understanding the Docker setup**
 - a. Client-server communication
- 5. Downloading the first Docker image**
- 6. Running the first Docker container**
 - a. Troubleshooting Docker containers
- 7. Handling Docker Containers**

8. Clarifying Docker terms

- a. Docker images
- b. Docker containers
- c. Docker Registry

9. Working with Docker images

- a. The Docker Hub
- b. Searching Docker images

10. Working with an interactive container

- a. Tracking changes inside containers
- b. Controlling Docker containers
- c. Housekeeping containers
- d. Building images from containers
- e. Launching a container as a daemon

11. Building Images

12. Docker's integrated image building system

13. A quick overview of the Dockerfile's syntax

- a. The comment line
- b. The parser directives

14. The Dockerfile build instructions

- a. The FROM instruction
- b. The MAINTAINER instruction
- c. The COPY instruction
- d. The ADD instruction
- e. The ENV instruction
- f. The ARG instruction
- g. The environment variables
- h. The USER instruction
- i. The WORKDIR instruction
- j. The VOLUME instruction
- k. The EXPOSE instruction
- l. The LABEL instruction
- m. The RUN instruction
- n. The CMD instruction
- o. The ENTRYPOINT instruction
- p. The HEALTHCHECK instruction
- q. The ONBUILD instruction
- r. The STOPSIGNAL instruction
- s. The SHELL instruction
- t. The .dockerignore file

15. A brief on the Docker image management

16. Publishing Images

17. Understanding Docker Hub



18. Pushing images to Docker Hub

19. Automating the build process for images

20. Private repositories on Docker Hub

21. Organizations and teams on Docker Hub

22. How Docker Works ?

- a. Namespaces
- b. Cgroups
- c. The Union filesystem

23. Adding a nonroot user to administer Docker

- a. Getting ready
- b. How to do it...

24. Docker Networking Primer

25. A brief overview of container networking

- a. Note

26. Networking and Docker

- a. Linux bridges
- b. Open vSwitch
- c. NAT
- d. IPtables
- e. AppArmor/SELinux

27. The docker0 bridge

- a. The --net default mode
- b. The --net=none mode
- c. The --net=container:\$container2 mode
- d. The --net=host mode

28. Docker OVS

29. Linking Docker containers

- a. Links

30. What's new in Docker networking?

- a. Sandbox
- b. Endpoint
- c. Network

31. The Docker CNM model

32. Envisaging container as a service

- a. Building an HTTP server image
- b. Running the HTTP server image as a service

33. Exposing container services

- a. Publishing a container's port – the -p option
- b. NAT for containers
- c. Note
- d. Retrieving the container port
- e. Binding a container to a specific IP address

- f. Autogenerating the Docker host port
- g. Connecting to the HTTP service

34. Sharing Data with Containers

35. Data volume

36. The volume management command

37. Sharing host data

- a. The practicality of host data sharing

38. Sharing data between containers

- a. Data-only containers
- b. Mounting data volume from other containers
- c. The practicality of data sharing between containers

39. Docker inbuilt service discovery

40. Linking containers

41. Orchestration of containers

- a. Orchestrating containers using docker-compose
- b. Installing docker-compose
- c. The docker-compose file
- d. The docker-compose command
- e. Common usage

42. Debugging Containers

- a. Debugging a containerized application
- b. The docker exec command
- c. The docker ps command
- d. The docker top command
- e. The docker stats command
- f. The Docker events command
- g. The docker logs command
- h. The docker attach command
- i. Debugging a Dockerfile

43. The Docker use cases

- a. Integrating containers into workflows
- b. Docker for HPC and TC applications

ANSIBLE

Table of Contents:

Getting Started with Ansible

- 1. IT automation**
 - a. The history of IT automation
 - b. Advantages of IT automation
 - c. Disadvantages of IT automation
 - d. Types of IT automation
- 2. What is Ansible?**
- 3. Secure Shell (SSH)**
- 4. Installing Ansible**
 - a. Installing Ansible using the system's package manager
- 5. Ansible version and configuration**
- 6. Inventory parsing and data sources**
 - a. Static inventory
 - b. Inventory variable data
- 7. Automating Simple Tasks**
- 8. YAML**
- 9. Hello Ansible**
- 10. Working with playbooks**
 - a. Studying the anatomy of a playbook
 - b. Running a playbook
- 11. Ansible verbosity**
- 12. Variables in playbooks**
- 13. Creating the Ansible user**
- 14. Configuring a basic server**
 - a. Enabling EPEL
 - b. Installing Python bindings for SELinux
 - c. Upgrading all installed packages
 - d. Ensuring that NTP is installed, configured, and running
 - e. Ensuring that Firewalld is present and enabled
 - f. Adding a customized MOTD
 - g. Changing the hostname
 - h. Reviewing and running the playbook
- 15. Installing and configuring a web server**
- 16. Publishing a website**
- 17. Jinja2 templates**
 - a. Filters
 - b. Conditionals

- c. Cycles
- 18. Working with inventory files**
 - a. The basic inventory file
 - b. Groups in an inventory file
 - c. Regular expressions in the inventory file
- 19. Working with variables**
 - a. Host variables
 - b. Group variables
 - c. Variable files
 - d. Overriding configuration parameters with an inventory file
- 20. Working with dynamic inventory**
 - a. Amazon Web Services
- 21. Working with iterates in Ansible**
 - a. Standard iteration - with_items
 - b. Nested loops - with_nested
- 22. Delegating a task**
- 23. Working with conditionals**
 - a. Boolean conditionals
 - b. Checking if a variable is set
- 24. Working with include**
- 25. Working with handlers**
- 26. Working with roles**
 - a. Project organization
 - b. Anatomy of a role
 - c. Transforming your playbooks in a full Ansible project
- 27. Execution strategies**
- 28. Tasks blocks**
- 29. Security management**
 - a. Using Ansible vault
 - b. Vaults and playbooks
 - c. Encrypting user passwords
 - d. Hiding passwords
 - e. Using no_log

CHEF COOKBOOK DEVELOPMENT

Contents

1. Chef Infrastructure

a. Introduction

i. Using Version Control System

1. Note
2. Getting Ready

ii. Installing the Chef Development kit on your workstation

1. How to do it

iii. Using the Hosted Chef

1. Getting Ready
2. How to do it
3. How it works
4. There's more
5. Note

iv. Managing Virtual Machines with vagrant

1. TipGetting ReadyHow to do it
2. How it worksThere's mre...
3. **Creating and Usingcookbooks**Getting Ready
4. How to do it
5. How it works
6. There's more

v. Inspecting files on your Chefserver with knife

1. Getting Ready
2. How to do it
3. How it works
4. There's more..

vii. Defining Cookbook dependencies.

1. Getting Ready
2. How to do it
3. How it works
4. There's more
5. Tip

viii. Managing CookBook dependencies with berkshelf

1. Getting Ready
2. How to do it



3. How it works
4. Note
5. There's more
6. See Also

ix. Using Custom Knife Plugins

1. Getting Ready
2. How to do it
3. How it works
4. There's more

x. Deleting Node from the Chef Server

1. Getting Ready
2. How to do it
3. How it works
4. There's more

xi. Developing recipes with local mode

1. Getting Ready
2. How to do it
3. How it works
4. There's more

xii. Using Roles

1. Getting Ready
2. How to do it
3. How it works

xiii. Using Environments

1. Getting Ready
2. How to do it
3. How it works
4. There's more

xiv. Freezing Cookbooks

1. Note
2. Getting Ready
3. Make sure you have at least one cookbook (I'll use the ntp cookbook) registered with your chef server.
4. How to do it
5. How it works
6. There's more

xv. Running the chef client as a Daemon

1. Getting ready
2. How to do it
3. How it works
4. Tip
5. There's more

6. Note

2. Evaluating and Troubleshooting Cookbooks and chef Runs

a. Introduction

b. Testing your chef cookbooks with cookstyle and Rubocop

- i. How to do it
- ii. Carryout the following steps to test your cookbook; run cookstyle on the ntp cookbook.
- iii. How it works
- iv. There's more
- v. See also

c. Flagging Problems in your chef cookbooks with Foodcritic

- i. Getting Ready
- ii. How to do it
- iii. How it works
- iv. There's more
- v. See also

www.topitcourses.com

--

Continuous Integration with Jenkins

Contents

1. Concepts of Continuous Integration.....	6
The agile software development process.....	6
Software development life cycle.....	6
Continuous Integration	7
Note.....	9
Agile runs on Continuous Integration	9
How to achieve Continuous Integration	10
Development operations	10
Use a version control system	11
Use repository tools	17
Use a Continuous Integration tool.....	19
Creating a self-triggered build.....	20
Automate the packaging	21
Using build tools.....	22
Automating the deployments.....	23
Note.....	23
Automating the testing.....	24
Use static code analysis.....	25
Continuous Integration benefits	26
Freedom from long integrations	26
Production-ready features.....	26
Analyzing and reporting	27
Catch issues faster	27
Spend more time adding features	27
Rapid development	28
2. Setting up Jenkins.....	28
Introduction to Jenkins.....	28

What is Jenkins made of?	29
Note.....	33
Why use Jenkins as a Continuous Integration server?	34
Jenkins as a centralized Continuous Integration server	35
Hardware requirements	37
Running Jenkins inside a container	37
Installing Jenkins as a service on the Apache Tomcat server.....	38
Note.....	38
Setting up the Jenkins home path.....	39
Why run Jenkins inside a container?	42
Running Jenkins as a standalone application	45
Setting up Jenkins on Ubuntu	45
Note.....	46
Setting up Jenkins on Fedora/Centos	48
Note.....	48
Note.....	49
Tip	49
Sample use cases.....	51
Netflix.....	51
Yahoo!.....	52
Note.....	53
3. Configuring Jenkins.....	53
Creating your first Jenkins job.....	53
Note.....	55
Note.....	58
Adding a build step.....	58
Note.....	59
Adding post-build actions.....	60
Configuring the Jenkins SMTP server	61
Running a Jenkins job.....	62
Note.....	63
Jenkins build log.....	64
Note.....	65

Jenkins home directory	66
Jenkins backup and restore	68
Creating a Jenkins job to take periodic backup	69
Restoring a Jenkins backup	70
Upgrading Jenkins	71
Note	71
Upgrading Jenkins running on the Tomcat server	72
Upgrading standalone Jenkins master running on Ubuntu	73
Script to upgrade Jenkins on Ubuntu	77
Managing Jenkins plugins	78
The Jenkins Plugins Manager	78
Installing a Jenkins plugin to take periodic backup	81
Configuring the periodic backup plugin	82
Note	85
User administration	87
Enabling global security on Jenkins	88
Note	91
Using the Project-based Matrix Authorization Strategy	92
4. Continuous Integration Using Jenkins – Part I	97
Jenkins Continuous Integration Design	97
The branching strategy	98
The Continuous Integration pipeline	100
Note	102
Toolset for Continuous Integration	102
Setting up a version control system	105
Installing Git	105
Installing SourceTree (a Git client)	105
Creating a repository inside Git	105
Uploading code to Git repository	106
Configuring branches in Git	107
Note	108
Git cheat sheet	108
Configuring Jenkins	109

Installing the Git plugin	109
Note.....	111
Installing and configuring JDK.....	111
Note.....	113
Installing and configuring Maven	113
Installing the e-mail extension plugin	114
The Jenkins pipeline to poll the feature branch	115
Creating a Jenkins job to poll, build, and unit test code on the feature1 branch	115
Note.....	121
Creating a Jenkins job to merge code to the integration branch	127
Creating a Jenkins job to poll, build, and unit test code on the feature2 branch.....	128
Creating a Jenkins job to merge code to the integration branch	129
5. Continuous Integration Using Jenkins – Part II.....	130
Installing SonarQube to check code quality	131
Setting the Sonar environment variables.....	132
Running the SonarQube application	132
Note.....	133
Creating a project inside SonarQube.....	133
Installing the build breaker plugin for Sonar.....	136
Creating quality gates.....	136
Installing SonarQube Scanner.....	139
Setting the Sonar Runner environment variables	141
Installing Artifactory.....	141
Setting the Artifactory environment variables.....	142
Running the Artifactory application	143
Creating a repository inside Artifactory.....	144
Jenkins configuration.....	147
Installing the delivery pipeline plugin.....	147
Installing the SonarQube plugin.....	149
Note.....	153
Installing the Artifactory plugin	153
The Jenkins pipeline to poll the integration branch.....	156

Creating a Jenkins job to poll, build, perform static code analysis, and integration tests	157
Creating a Jenkins job to upload code to Artifactory	167
Note.....	168
Note.....	170
6. Continuous Delivery Using Jenkins	171
What is Continuous Delivery?.....	171
Continuous Delivery Design.....	173
Jenkins configuration.....	176
 Configuring Jenkins slaves on the testing server	176
 Note.....	181
 Slave agents via SSH tunneling.....	182
 Creating a Jenkins job to deploy code on the testing server.....	185

www.topitcourses.com



1. Concepts of Continuous Integration

Understanding the concepts of **Continuous Integration** is our prime focus in the current chapter. However, to understand Continuous Integration, it is first important to understand the prevailing software engineering practices that gave birth to it. Therefore, we will first have an overview of various software development processes, their concepts, and implications. To start with, we will first glance through the **agile software development process**. Under this topic, we will learn about the popular software development process, the waterfall model, and its advantages and disadvantages when compared to the agile model. Then, we will jump to the **Scrum framework** of software development. This will help us to answer how Continuous Integration came into existence and why it is needed. Next, we will move to the concepts and best practices of Continuous Integration and see how this helps projects to get agile. Lastly, we will talk about all the necessary methods that help us realize the concepts and best practices of Continuous Integration.

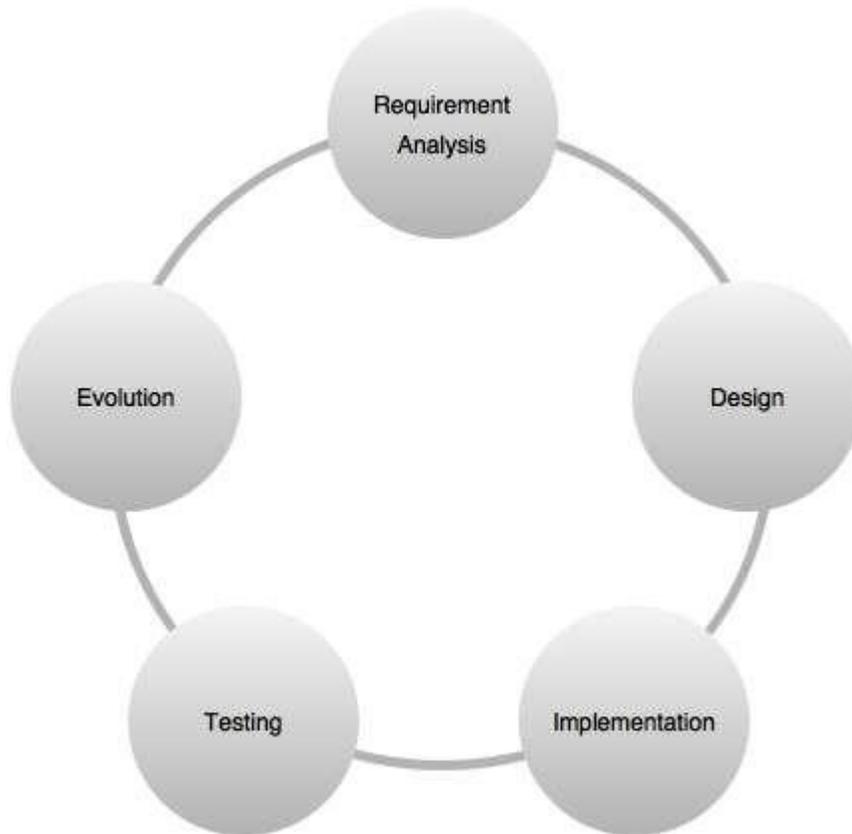
The agile software development process

The name agile rightly suggests **quick and easy**. Agile is a collection of software development methodologies in which software is developed through collaboration among self-organized teams. Agile software development promotes adaptive planning. The principles behind agile are incremental, quick, and flexible software development.

For most of us who are not familiar with the software development process itself, let's first understand what the software development process or software development life cycle is.

Software development life cycle

Software development life cycle, also sometimes referred to as **SDLC** in brief, is the process of planning, developing, testing, and deploying software. Teams follow a sequence of phases, and each phase uses the outcome of the previous phase, as shown in the following diagram:



Continuous Integration

Continuous Integration is a software development practice where developers frequently integrate their work with the project's **integration branch** and create a build.

Integration is the act of submitting your personal work (modified code) to the common work area (the potential software solution). This is technically done by merging your personal work (personal branch) with the common work area (Integration branch). Continuous Integration is necessary to bring out issues that are encountered during the integration as early as possible.

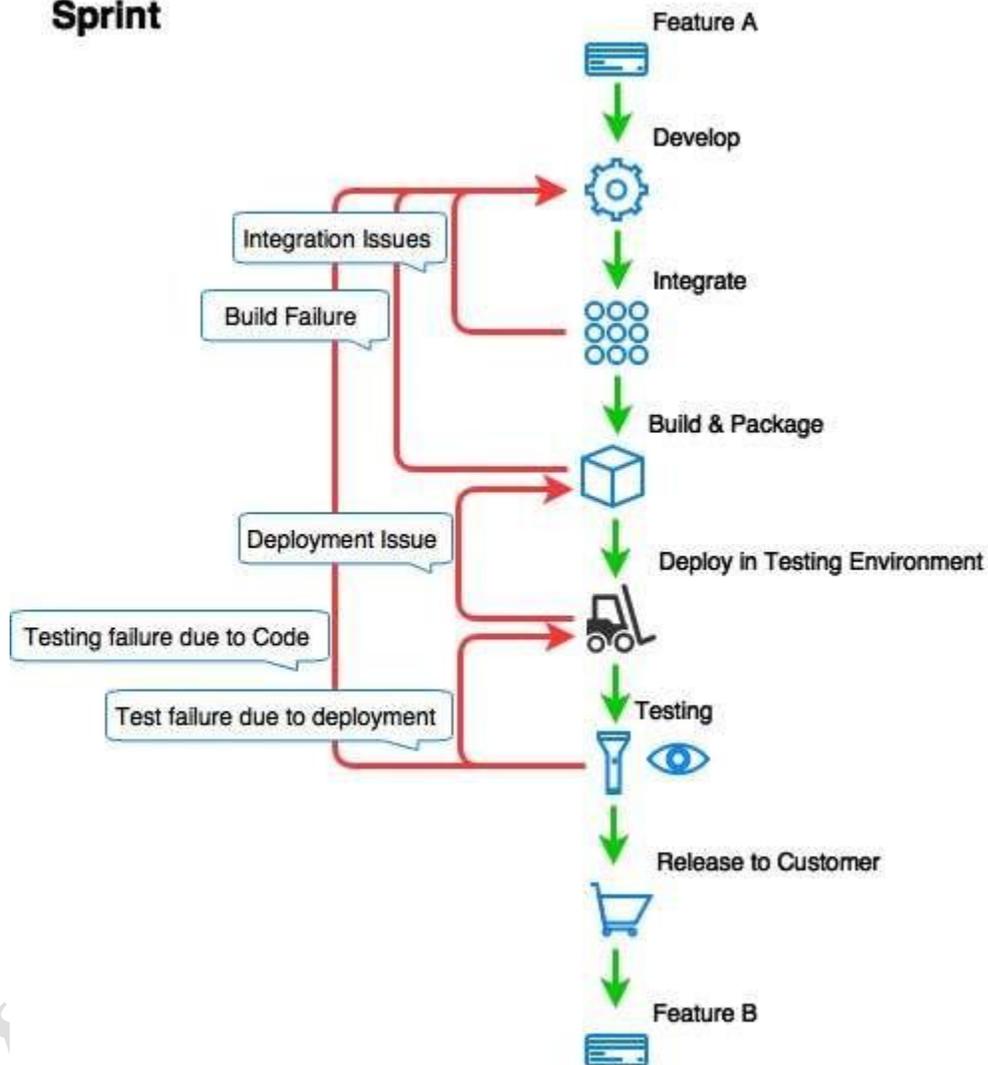
This can be understood from the following diagram, which depicts various issues encountered during a software development lifecycle. I have considered a practical scenario wherein I have chosen the Scrum development model, and for the sake of simplicity, all the meeting phases are excluded. Out of all the issues depicted in the following diagram, the following ones are detected early when Continuous Integration is in place:

- Build failure (the one before integration)
-

- Integration issues
- Build failure (the one after integration)

In the event of the preceding issues, the developer has to modify the code in order to fix it. A build failure can occur either due to an improper code or due to a human error while doing a build (assuming that the tasks are done manually). An integration issue can occur if the developers do not rebase their local copy of code frequently with the code on the Integration branch.

Sprint



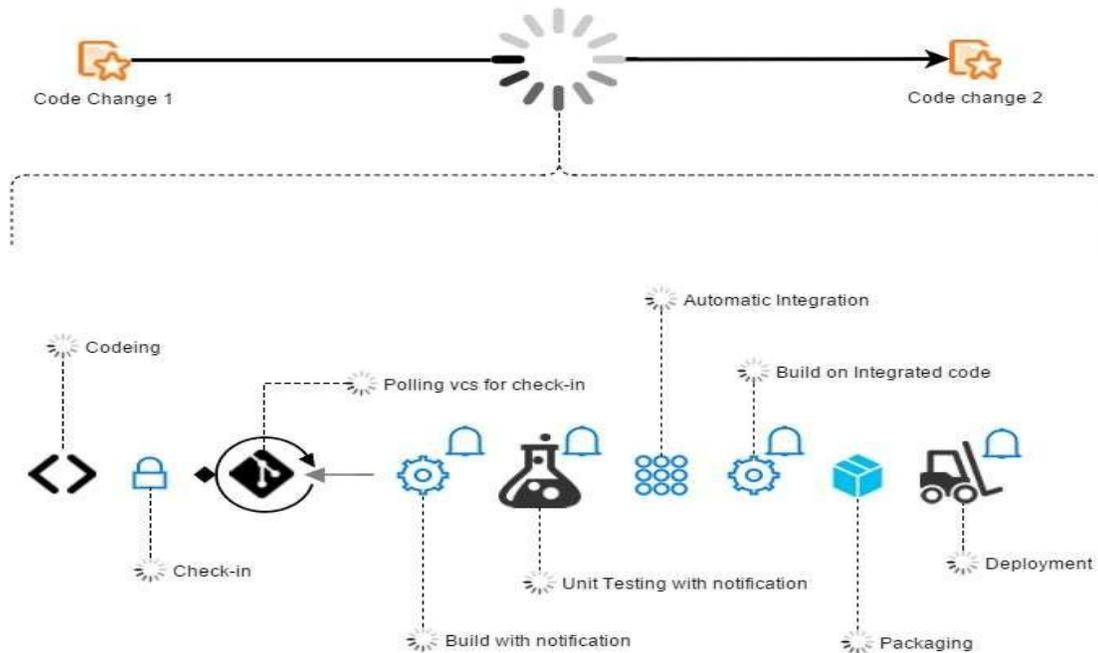
Note

In the preceding diagram, I have considered only a single testing environment for simplicity. However, in reality, there can be as many as three to four testing environments.

Agile runs on Continuous Integration

The agile software development process mainly focuses on faster delivery, and Continuous Integration helps it in achieving that speed. Yet, how does Continuous Integration do it? Let's understand this using a simple case.

Developing a feature may involve a lot of code changes, and between every code change, there can be a number of tasks, such as checking in the code, polling the version control system for changes, building the code, unit testing, integration, building on integrated code, packaging, and deployment. In a Continuous Integration environment, all these steps are made fast and error-free using automation. Adding notifications to it makes things even faster. The sooner the team members are aware of a build, integration, or deployment failure, the quicker they can act upon it. The following diagram clearly depicts all the steps involved in code changes:



Implementing Continuous Integration involves using various DevOps tools. Ideally, a DevOps engineer is responsible for implementing Continuous Integration. But, who is a DevOps engineer? And what is DevOps?

Development operations

DevOps stands for development operations, and the people who manage these operations are called DevOps engineers. All the following mentioned tasks fall under development operations:

- Build and release management
- Deployment management
- Version control system administration
- Software configuration management
- All sorts of automation

- Implementing continuous integration
- Implementing continuous testing
- Implementing continuous delivery
- Implementing continuous deployment
- Cloud management and virtualization

A DevOps engineer accomplishes the previously mentioned tasks using a set of tools; these tools are loosely called DevOps tools (Continuous Integration tools, agile tools, team collaboration tools, defect tracking tools, continuous delivery tools, cloud management tools, and so on).

A DevOps engineer has the capability to install and configure the DevOps tools to facilitate development operations. Hence, the name DevOps. Let's see some of the important DevOps activities pertaining to Continuous Integration.

Use a version control system

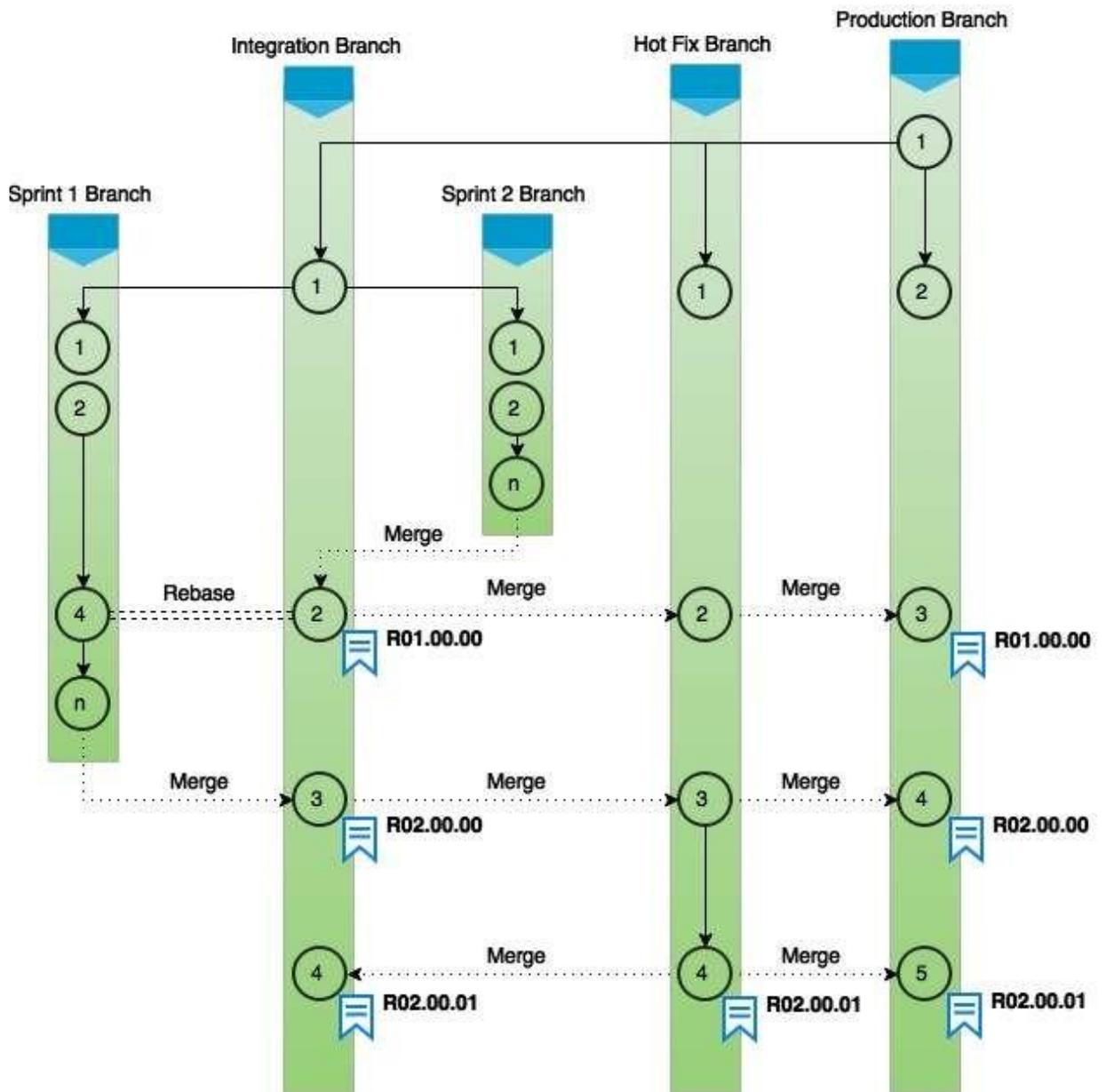
This is the most basic and the most important requirement to implement Continuous Integration. A version control system, or sometimes it's also called a **revision control system**, is a tool used to manage your code history. It can be centralized or distributed. Two of the famously centralized version control systems are SVN and IBM

Rational ClearCase. In the distributed segment, we have tools such as Git. Ideally, everything that is required to build software must be version controlled. A version control tool offers many features, such as labeling, branching, and so on.

When using a version control system, keep the branching to the minimum. Few companies have only one main branch and all the development activities happening on that. Nevertheless, most companies follow some branching strategies. This is because there is always a possibility that part of a team may work on a release and others may work on another release. At other times, there is a need to support older release versions. Such scenarios always lead companies to use multiple branches.

For example, imagine a project that has an Integration branch, a release branch, a hotfix branch, and a production branch. The development team will work on the release branch. They check-out and check-in code on the release branch. There can be more than one release branch where development is running in parallel. Let's say these are sprint 1 and sprint 2.

Once sprint 2 is near completion (assuming that all the local builds on the sprint 2 branch were successful), it is merged to the Integration branch. Automated builds run when there is something checked-in on the Integration branch, and the code is then packaged and deployed in the testing environments. If the testing passes with flying colors and the business is ready to move the release to production, then automated systems take the code and merge it with the production branch.



Typical branching strategies

From here, the code is then deployed in production. The reason for maintaining a separate branch for production comes from the desire to maintain a neat code with less number of versions. The production branch is always in sync with the hotfix branch. Any instant fix required on the production code is developed on the hotfix branch. The hotfix changes are then merged to the production as well as the

Integration branch. The moment sprint 1 is ready, it is first rebased with the Integration branch and then merged into it. And it follows the same steps thereafter.

Types of version control system

We have already seen that a version control system is a tool used to record changes made to a file or set of files over time. The advantage is that you can recall specific versions of your file or a set of files. Almost every type of file can be version controlled. It's always good to use a **Version Control System (VCS)** and almost everyone uses it nowadays. You can revert an entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover.

Looking back at the history of version control tools, we can observe that they can be divided into three categories:

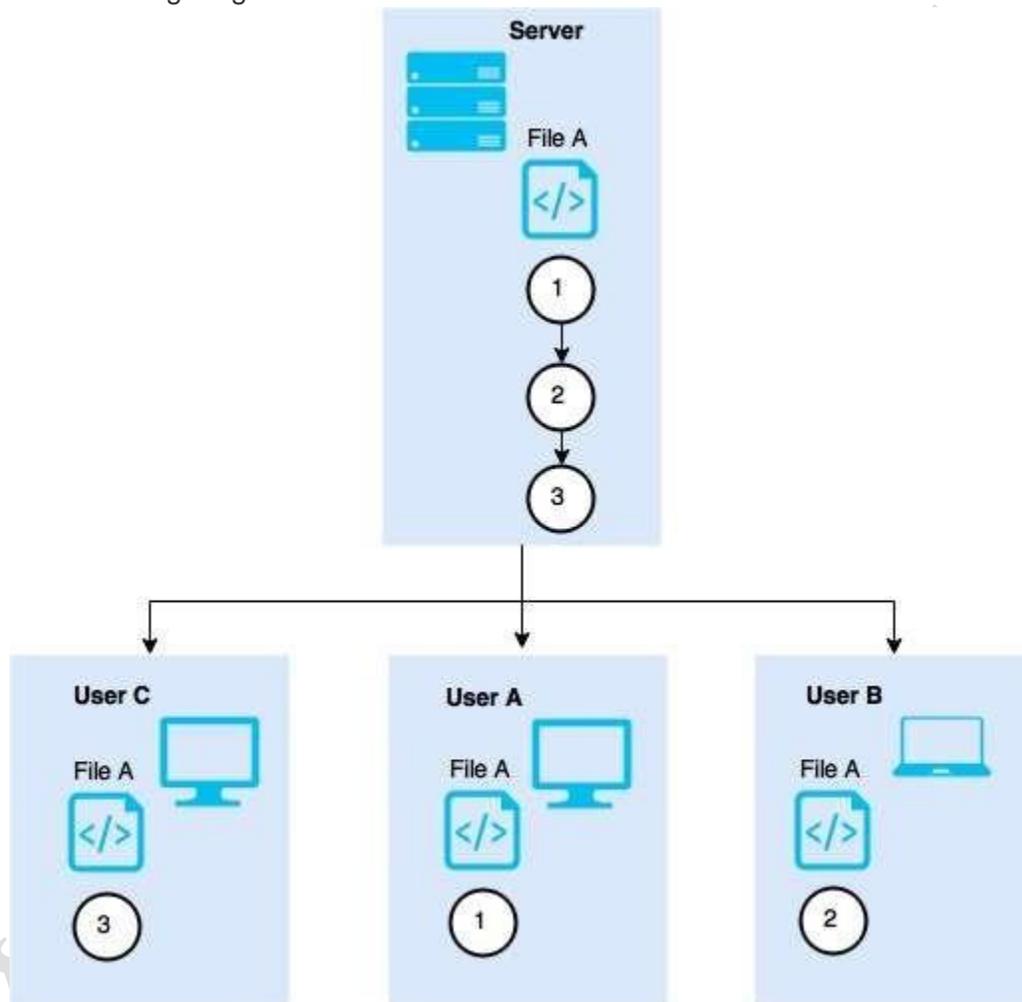
- Local version control systems
- Centralized version control systems
- Distributed version control systems

Centralized version control systems

Initially, when VCS came into existence some 40 years ago, they were mostly personal, like the one that comes with Microsoft Office Word, wherein you can version control a file you are working on. The reason was that in those times software development activity was minuscule in magnitude and was mostly done by individuals. But, with the arrival of large software development teams working in collaboration, the need for a centralized VCS was sensed. Hence, came VCS tools, such as Clear Case and Perforce. Some of the advantages of a centralized VCS are as follows:

- All the code resides on a centralized server. Hence, it's easy to administrate and provides a greater degree of control.
- These new VCS also bring with them some new features, such as labeling, branching, and baselining to name a few, which help people collaborate better.
- In a centralized VCS, the developers should always be connected to the network. As a result, the VCS at any given point of time always represents the updated code.

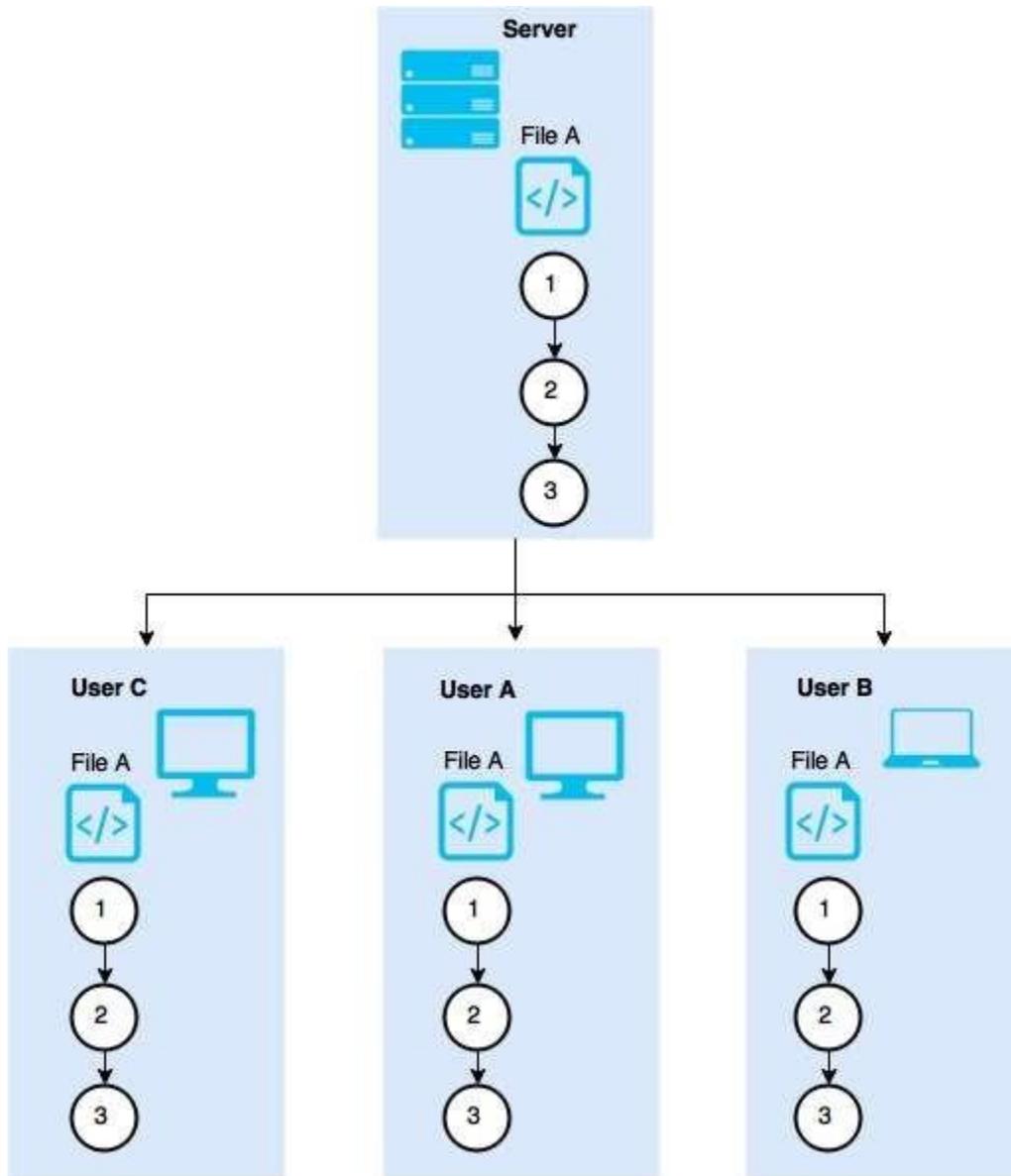
The following diagram illustrates a centralized VCS:



Distributed version control systems

Another type of VCS is the distributed VCS. Here, there is a central repository containing all the software solution code. Instead of creating a branch, the developers completely clone the central repository on their local machine and then create a branch out of the local clone repository. Once they are done with their work, the developer first merges their branch with the Integration branch, and then syncs the local clone repository with the central repository.

You can argue that this is a combination of a local VCS plus a central VCS. An example of a distributed VCS is Git.



Use repository tools

As part of the software development life cycle, the source code is continuously built into binary artifacts using Continuous Integration. Therefore, there should be a place to store these built packages for later use. The answer is to use a repository tool. But, what is a repository tool?

A repository tool is a version control system for binary files. Do not confuse this with the version control system discussed in the previous sections. The former is responsible for versioning the source code and the latter for binary files, such as `.rar`, `.war`, `.exe`, `.msi`, and so on.

As soon as a build is created and passes all the checks, it should be uploaded to the repository tool. From there, the developers and testers can manually pick them, deploy them, and test them, or if the automated deployment is in place, then the build is automatically deployed in the respective test environment. So, what's the advantage of using a build repository?

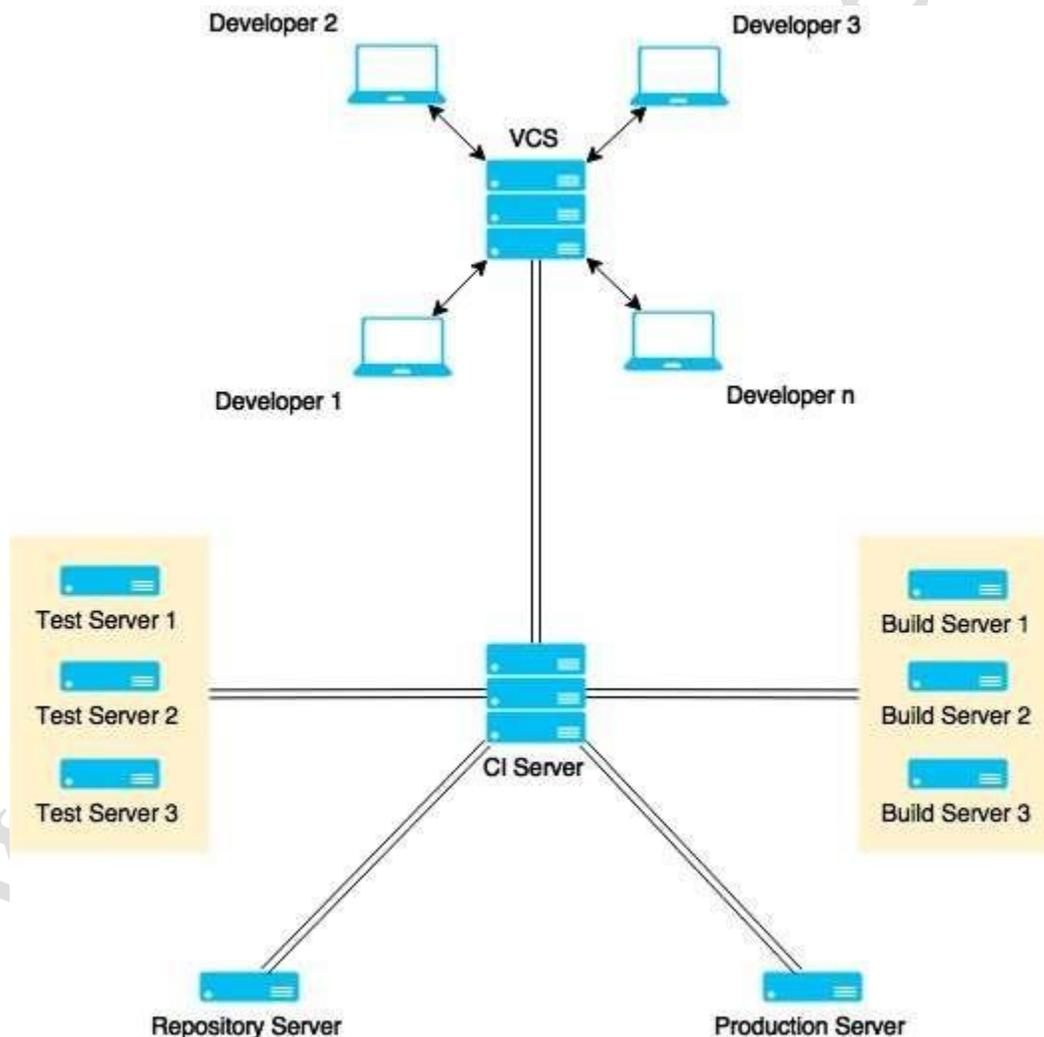
A repository tool does the following:

- Every time a build gets generated, it is stored in a repository tool. There are many advantages of storing the build artifacts. One of the most important advantages is that the build artifacts are located in a centralized location from where they can be accessed when needed.
 - It can store third-party binary plugins, modules that are required by the build tools. Hence, the build tool need not download the plugins every time a build runs. The repository tool is connected to the online source and keeps updating the plugin repository.
 - It records what, when, and who created a build package.
 - It creates a staging area to manage releases better. This also helps in speeding up the Continuous Integration process.
 - In a Continuous Integration environment, each build generates a package and the frequency at which the build and packaging happen is high. As a result, there is a huge pile of packages. Using a repository tool makes it possible to store all the packages in one place. In this way, developers get the liberty to choose what to promote and what not to promote in higher environments.
-

Use a Continuous Integration tool

What is a Continuous Integration tool? It is nothing more than an orchestrator. A continuous integration tool is at the center of the Continuous Integration system and is connected to the version control system tool, build tool, repository tool, testing and production environments, quality analysis tool, test automation tool, and so on. All it does is an orchestration of all these tools, as shown in the next image.

There are many Continuous Integration tools: Jenkins, Build Forge, Bamboo, and Team city to name a few.



Basically, Continuous Integration tools consist of various pipelines. Each pipeline has its own purpose. There are pipelines used to take care of Continuous Integration. Some take care of testing, some take care of deployments, and so on. Technically, a pipeline is a flow of jobs. Each job is a set of tasks that run sequentially. Scripting is an integral part of a Continuous Integration tool that performs various kinds of tasks. The tasks may be as simple as copying a folder/file from one location to another, or it can be a complex Perl script used to monitor a machine for file modification.

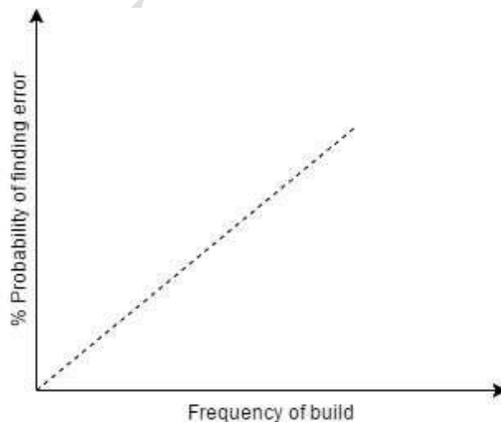
Creating a self-triggered build

The next important thing is the self-triggered automated build. Build automation is simply a series of automated steps that compile the code and generate executables. The build automation can take help of build tools, such as Ant and Maven. Self-triggered automated builds are the most important parts of a Continuous Integration system. There are two main factors that call for an automated build mechanism:

- Speed
- Catching integration or code issues as early as possible

There are projects where 100 to 200 builds happen per day. In such cases, speed is an important factor. If the builds are automated, then it can save a lot of time. Things become even more interesting if the triggering of the build is made self-driven without any manual intervention. An auto-triggered build on every code change further saves time.

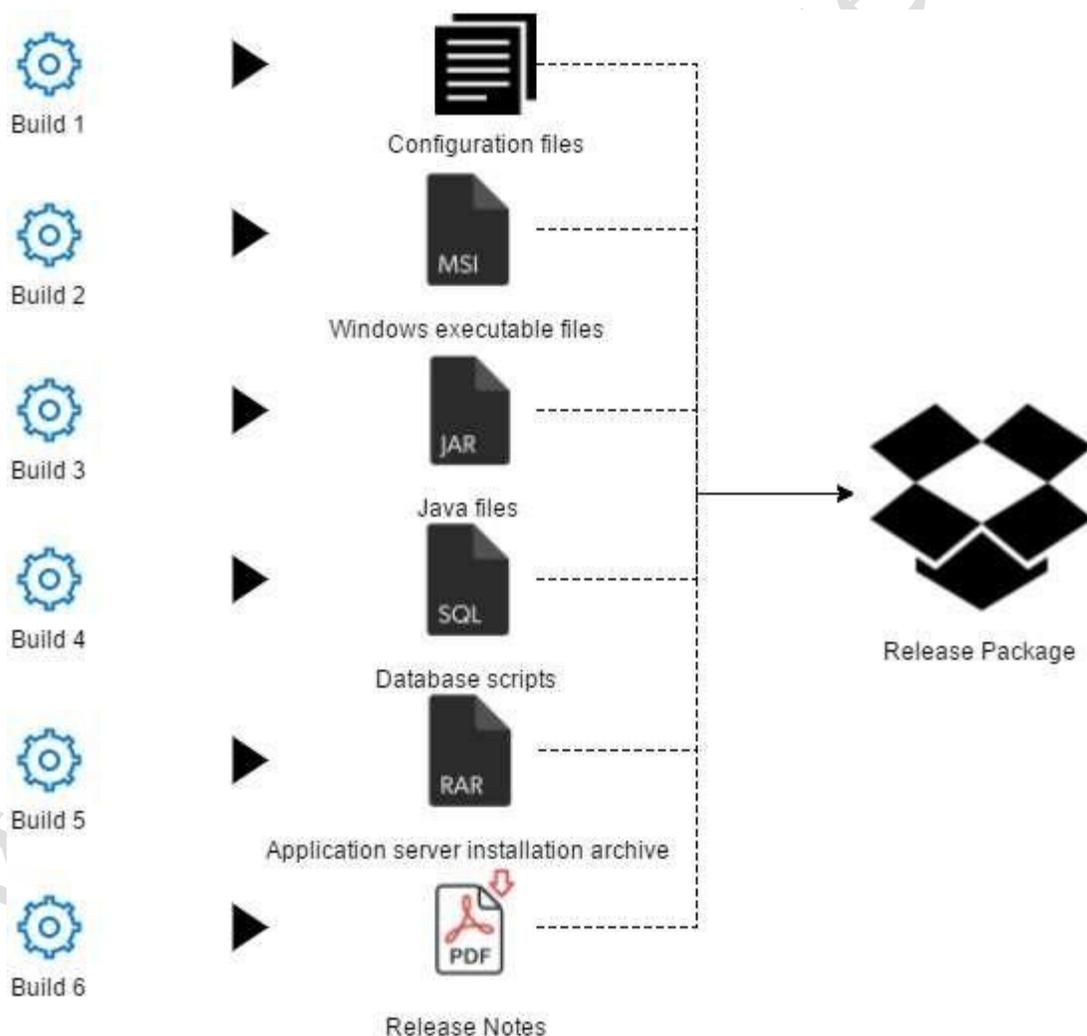
When builds are frequent and fast, the probability of finding errors (a build error, compilation error, and integration error) is also greater and faster.



Automate the packaging

There is a possibility that a build may have many components. Let's take, for example, a build that has a `.rar` file as an output. Along with this, it has some Unix configuration files, release notes, some executables, and also some database changes. All these different components need to be together. The task of creating a single archive or a single media out of many components is called packaging.

This again can be automated using the Continuous Integration tools and can save a lot of time.



Using build tools

IT projects can be on various platforms, such as Java, .NET, Ruby on Rails, C, and C++ to name a few. Also, in a few places, you may see a collection of technologies. No matter what, every programming language, excluding the scripting languages, has compilers that compile the code. Ant and Maven are the most common build tools used for projects based on Java. For the .NET lovers, there is MSBuild and TFS build. Coming to the Unix and Linux world, you have `make` and `omake`, and also `clearmake` in case you are using IBM Rational ClearCase as the version control tool. Let's see the important ones.

Maven

Maven is a build tool used mostly to compile Java code. It uses Java libraries and Maven plugins in order to compile the code. The code to be built is described using an XML file that contains information about the project being built, dependencies, and so on.

Maven can be easily integrated into Continuous Integration tools, such as Jenkins, using plugins.

MSBuild

MSBuild is a tool used to build Visual Studio projects. MSBuild is bundled with Visual Studio. MSBuild is a functional replacement for `nmake`. MSBuild works on project files, which have the XML syntax, similar to that of Apache Ant. Its fundamental structure and operation are similar to that of the Unix `make` utility. The user defines what will be the input (the various source codes), and the output (usually, a `.exe` or `.msi`). But, the utility itself decides what to do and the order in which to do it.

Automating the deployments

Consider an example, where the automated packaging has produced a package that contains `.war` files, database scripts, and some Unix configuration files. Now, the task here is to deploy all the three artifacts into their respective environments.

The `.war` files must be deployed in the application server. The Unix configuration files should sit on the respective Unix machine, and lastly, the database scripts should be executed in the database server. The deployment of such packages containing multiple components is usually done manually in almost every organization that does not have automation in place. The manual deployment is slow and prone to human errors. This is where the automated deployment mechanism is helpful.

Automated deployment goes hand in hand with the automated build process. The previous scenario can be achieved using an automated build and deployment solution that builds each component in parallel, packages them, and then deploys them in parallel. Using tools such as Jenkins, this is possible. However, there are some challenges, which are as follows:

- There is a considerable amount of scripting required to orchestrate build packaging and deployment of a release containing multiple components. These scripts by themselves are huge code to maintain that require time and resources.
- In most of the cases, deployment is not as simple as placing files in a directory. For example, there are situations where the deployment activity is preceded by steps to configure the environment.

Note

The field of managing the configuration on multiple machines is called **configuration management**. There are tools, such as Chef and Puppet, to do this.

Automating the testing

Testing is an important part of a software development life cycle. In order to maintain quality software, it is necessary that the software solution goes through various test scenarios. Giving less importance to testing can result in customer dissatisfaction and a delayed product.

Since testing is a manual, time-consuming, and repetitive task, automating the testing process can significantly increase the speed of software delivery. However, automating the testing process is a bit more difficult than automating the build, release, and deployment processes. It usually takes a lot of efforts to automate nearly all the test cases used in a project. It is an activity that matures over time.

Hence, when we begin to automate the testing, we need to take a few factors into consideration. Test cases that are of great value and easy to automate must be considered first. For example, automate the testing where the steps are the same, but they run every time with different data. You can also automate the testing where a software functionality is being tested on various platforms. In addition, automate the testing that involves a software application running on different configurations.

Previously, the world was mostly dominated by the desktop applications. Automating the testing of a GUI-based system was quite difficult. This called for scripting languages where the manual mouse and keyboard entries were scripted and executed to test the GUI application. Nevertheless, today the software world is completely dominated by the web and mobile-based applications, which are easy to test through an automated approach using a test automation tool.

Once the code is built, packaged, and deployed, testing should run automatically to validate the software. Traditionally, the process followed is to have an environment for SIT, UAT, PT, and Pre-Production. First, the release goes through SIT, which stands for System Integration Test. Here, testing is performed on an integrated code to check

its functionality all together. If pass, the code is deployed in the next environment, that is, UAT where it goes through a user acceptance test, and then similarly, it can lastly be deployed in PT where it goes through the performance test. Thus, in this way, the testing is prioritized.

It is not always possible to automate all of the testing. But, the idea is to automate whatever testing is possible. The previous method discussed requires the need to have many environments and also a number of automated deployments into various environments. To avoid this, we can go for another method where there is only one environment where the build is deployed, and then, the basic tests are run and after that, long running tests are triggered manually.

Use static code analysis

Static code analysis, also commonly called **white-box testing**, is a form of software testing that looks for the structural qualities of the code. For example, it reveals how robust or maintainable the code is. Static code analysis is performed without actually executing programs. It is different from the functional testing, which looks into the functional aspects of software and is dynamic.

Static code analysis is the evaluation of software's inner structures. For example, is there a piece of code used repetitively? Does the code contain lots of commented lines? How complex is the code? Using the metrics defined by a user, an analysis report can be generated that shows the code quality in terms of maintainability. It doesn't question the code functionality.

Some of the static code analysis tools, such as SonarQube come with a dashboard, which shows various metrics and statistics of each run. Usually, as part of Continuous Integration, the static code analysis is triggered every time a build runs. As discussed in the previous sections, static code analysis can also be included before a developer

tries to check-in his code. Hence, code of low quality can be prevented right at the initial stage.

Static code analysis support many languages, such as Java, C/C++, Objective-C, C#, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, and so on.

Continuous Integration benefits

The way a software is developed always affects the business. The code quality, the design, time spent in development and planning of features, all affect the promises that a company has made to its clients.

Continuous Integration helps the developers in helping the business. While going through the previous topics, you might have already figured out the benefits of implementing Continuous Integration. However, let's see some of the benefits that Continuous Integration has to offer.

Freedom from long integrations

When every small change in your code is built and integrated, the possibility of catching the integration errors at an early stage increases. Rather than integrating once in 6 months, as seen in the waterfall model, and then spending weeks resolving the merge issues, it is good to integrate frequently and avoid the merge hell. The Continuous Integration tool like Jenkins automatically builds and integrates your code upon check-in.

Production-ready features

Continuous Delivery enables you to release deployable features at any point in time. From a business perspective, this is a huge advantage. The features are developed, deployed, and tested within a timeframe of 2 to 4 weeks and are ready to go live with a click of a button.

Analyzing and reporting

How frequent are the releases? What is the success rate of builds? What is the thing that is mostly causing a build failure? Real-time data is always a must in making critical decisions. Projects are always in the need of recent data to support decisions. Usually, managers collect this information manually, which requires time and efforts. Continuous Integration tools, such as Jenkins provide the ability to see trends and make decisions. A Continuous Integration system provides the following features:

- Real-time information on the recent build status and code quality metrics.
- Since integrations occur frequently with a Continuous Integration system, the ability to notice trends in build, and overall quality becomes possible.

Continuous Integration tools, such as Jenkins provide the team members with metrics about the build health. As all the build, packaging, and deployment work is automated and tracked using a Continuous Integration tool; therefore, it is possible to generate statistics about the health of all the respective tasks. These metrics can be the build failure rate, build success rate, the number of builds, who triggered the build, and so on.

All these trends can help project managers and the team to ensure that the project is heading in the right direction and at the right pace.

Catch issues faster

This is the most important advantage of having a carefully implemented Continuous Integration system. Any integration issue or merge issue gets caught early. The Continuous Integration system has the facility to send notifications as soon as the build fails.

Spend more time adding features

In the past, development teams performed the build, release, and deployments. Then, came the trend of having a separate team to handle build, release, and deployment work. Yet again that was

not enough, as this model suffered from communication issues between the development team and the release team.

However, using Continuous Integration, all the build, release, and the deployment work gets automated. Therefore, now the development team need not worry about anything other than developing features. In most of the cases, even the completed testing is automated.

Rapid development

From a technical perspective, Continuous Integration helps teams work more efficiently. This is because Continuous Integration works on the agile principles. Projects that use Continuous Integration follow an automatic and continuous approach while building, testing, and integrating their code. This results in a faster development.

Since everything is automated, developers spend more time developing their code and zero time on building, packaging, integrating, and deploying it. This also helps teams, which are geographically distributed, to work together. With a good software configuration management process in place, people can work on large teams. **Test Driven Development (TDD)** can further enhance the agile development by increasing its efficiency.

"Behind every successful agile project, there is a Continuous Integration server."

Note:

- This is a **preview Devops Jenkins Docker Ansible e-Book** containing **only 30 pages**.
- It is provided to help you understand **how the full Devops Jenkins Docker Ansible e-Book and is structured**.
- The **complete Devops Jenkins Docker Ansible e-Book** includes detailed concepts, real-world examples, and career guidance.
- **Purchase the full Devops Jenkins Docker Ansible e-Book for just ₹449.**
- Buy now from our official website: <https://topitcourses.com/devops-docker-jenkins-ansible/>