Software Development Life Cycle Models

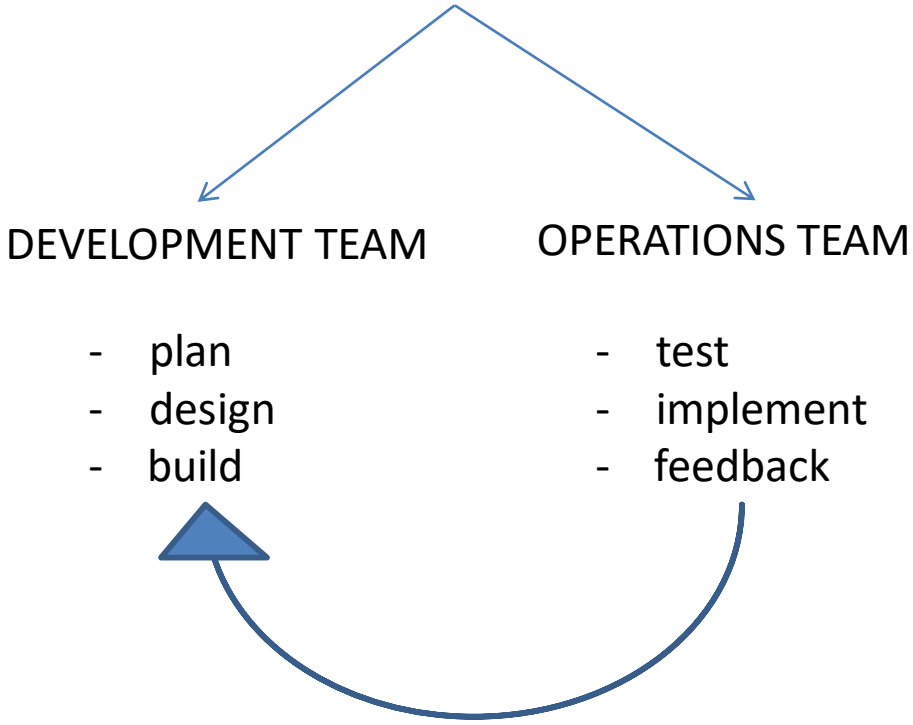
Waterfall model

vs

Agile model

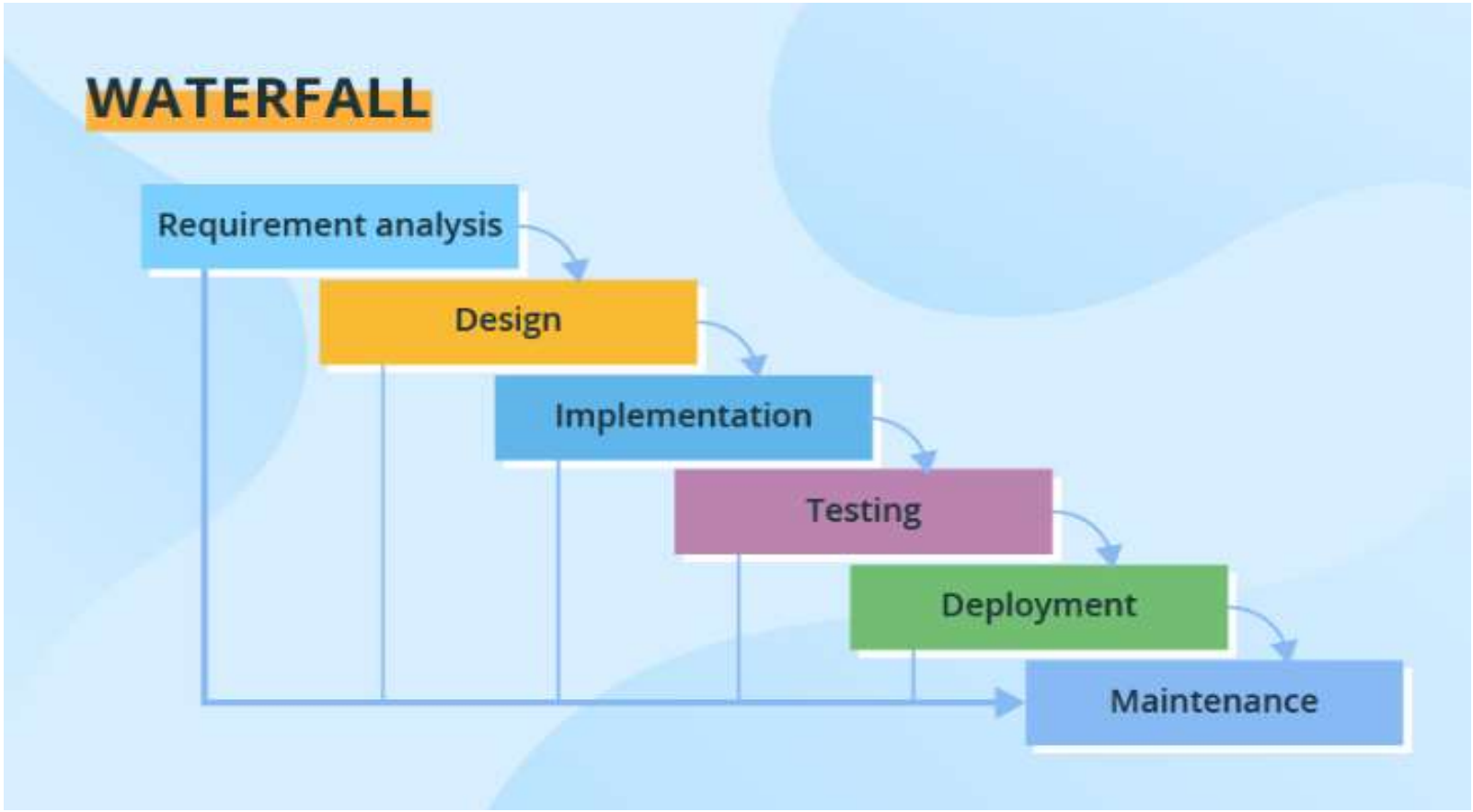
WHY DEVOPS?

Software Development



Software Development Life Cycle Models-

Waterfall model



Software Development Life Cycle Models-

Waterfall model-

ADVANTAGES-

1. Simple
2. Follows a straight-forward approach

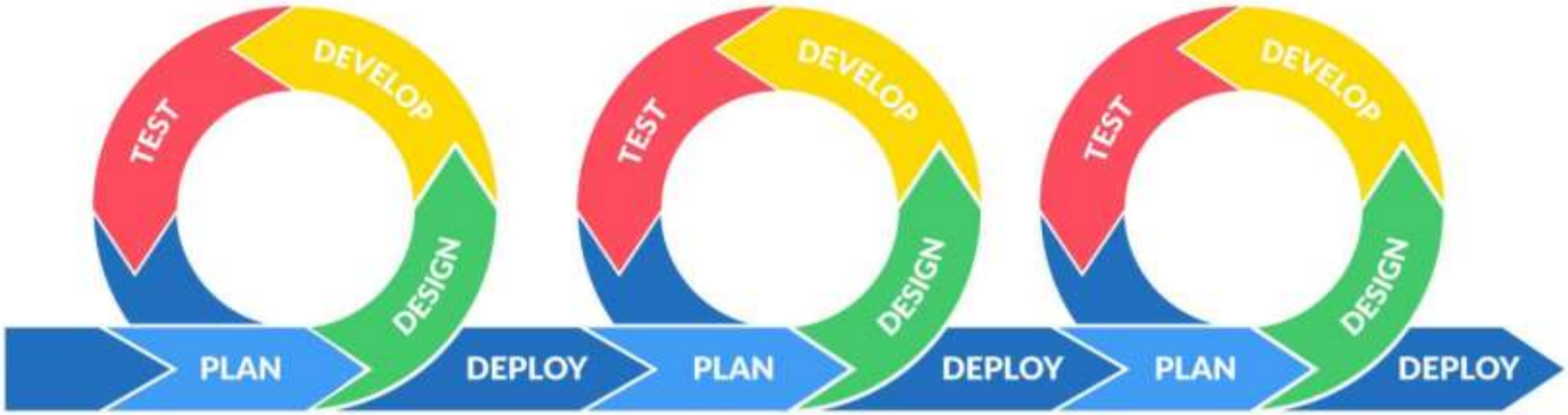
DISADVANTAGES-

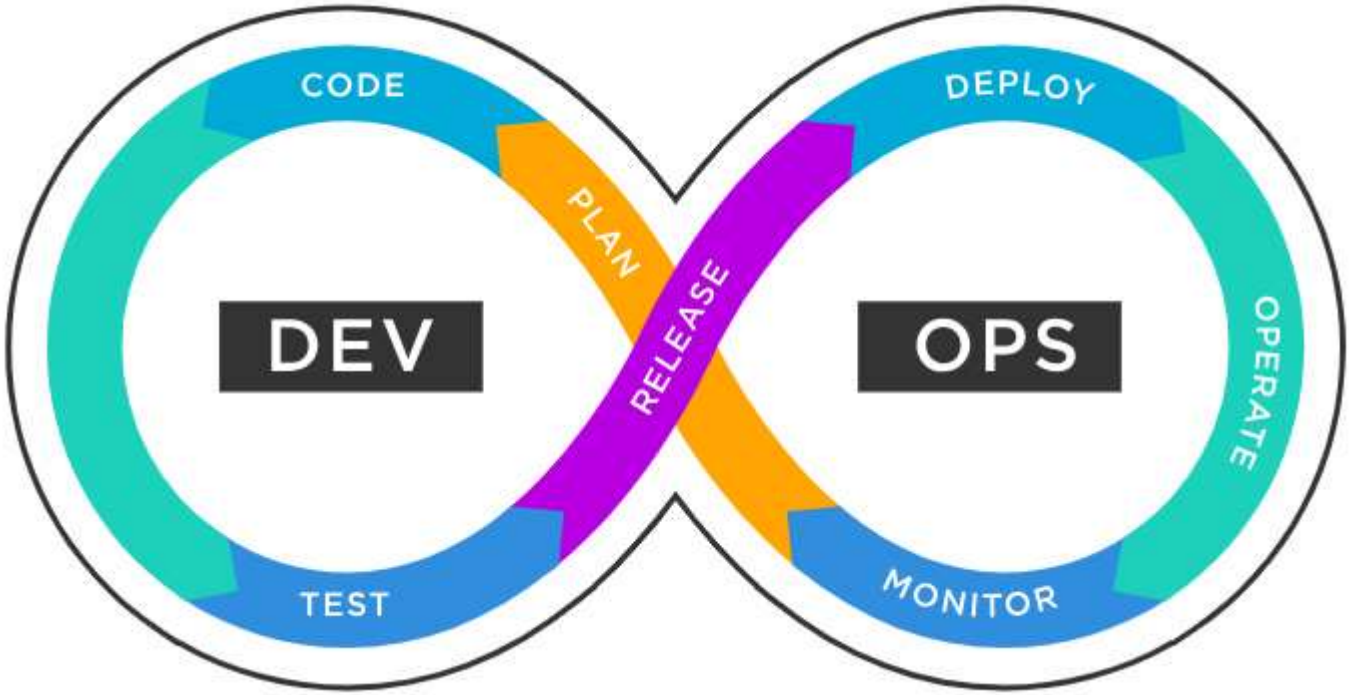
1. Unless you complete a particular stage, you cannot proceed to the next stage
2. Time consuming
3. Suitable only for stable applications.
4. Cannot be used used for large and object-oriented applications.

Software Development Life Cycle Models-

Agile methodology

Agile is a project management approach that emphasizes collaboration, flexibility, and customer satisfaction. It values delivering a working product incrementally and embracing change.





A DevOps pipeline is the set of tools, flows, and automated processes that enable teams to effectively and efficiently leverage various technologies in building and deploying software.

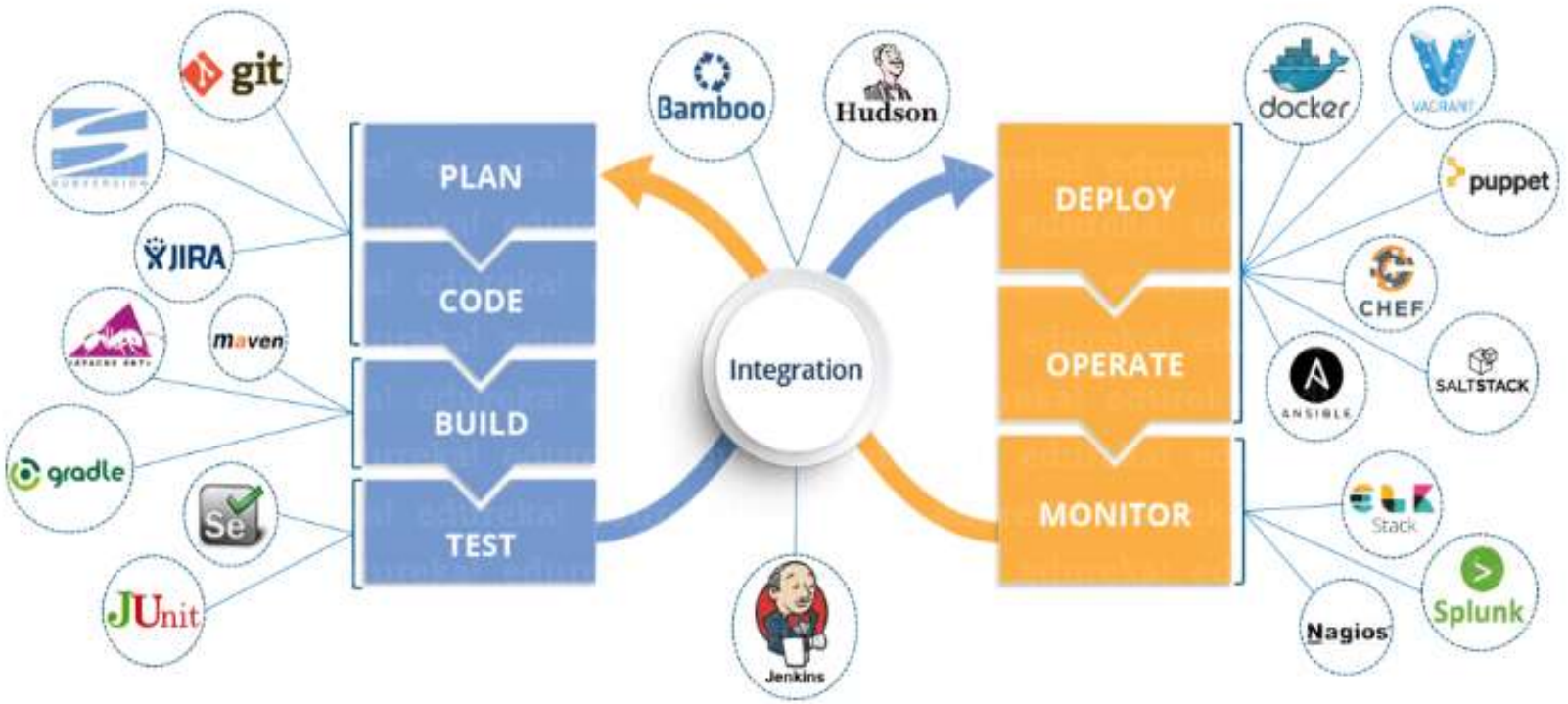
→The primary objective of the pipeline is to keep the software development process well focused and continually organized.

DevOps, a combination of 'development and operations', is a mix of practices, tools and cultural philosophies that enable an organization to quickly deliver applications and services. It also helps products go from the drawing board to market at a faster pace than traditional software development because operations and development engineers work closely together in the entire lifecycle, from design through the development process to production support.

Generally, DevOps represents a change in the culture of IT from one of separateness to one of working as a team. It's about creating rapid service delivery using Agile, lean practices in a system-oriented approach. DevOps also aims to use technology, especially automation tools, to leverage highly programmable and dynamic infrastructure.

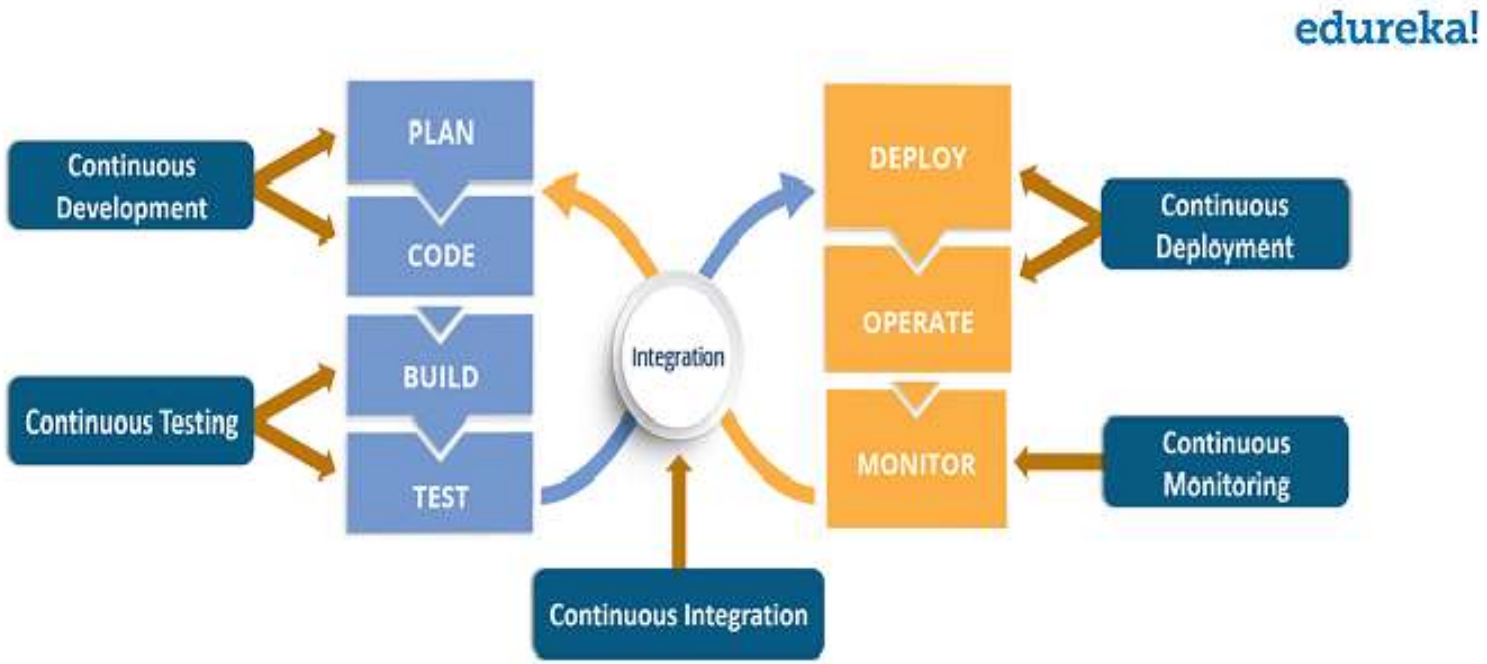
Various DevOps tools such as Git, Ansible, Docker, Puppet, Jenkins, Chef, Nagios, and Kubernetes.

DevOps Architecture



DevOps Lifecycle

The various phases such as continuous development, continuous integration, continuous testing, continuous deployment, and continuous monitoring constitute DevOps Life cycle. Now let us have a look at each of the phases of DevOps life cycle one by one.



SOURCE CODE MANAGEMENT

GIT

- A high quality distributed version control system that lets us manage and keep track of source code.
- Installed and managed on our local systems rather than on the cloud

GITHUB

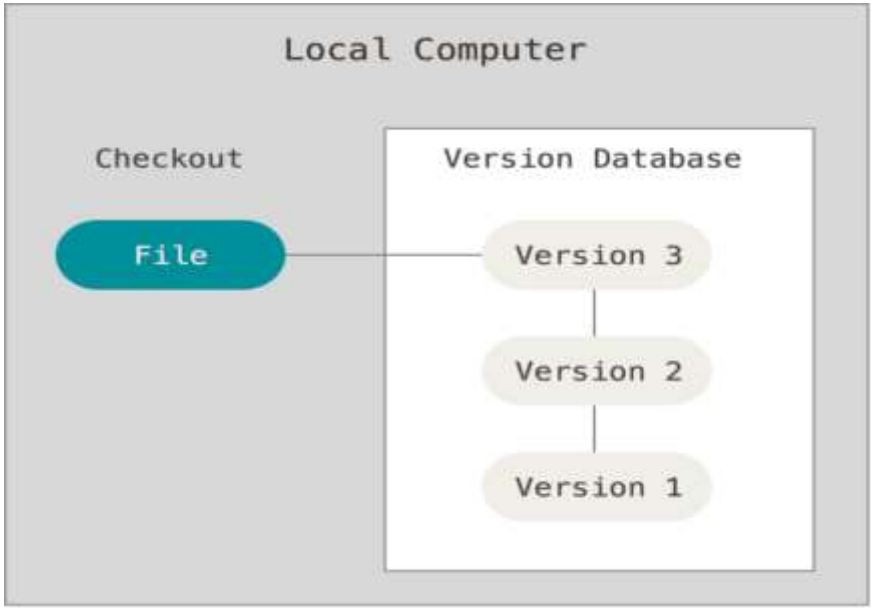
- A hosting service for managing our git repositories
- Exclusively cloud-based

SOURCE CODE MANAGEMENT

Version Control System

→ Version Control or Source Control is the practice of tracking and managing changes to the source code.

→ Version Control Systems(VCS) are software tools that helps software teams manage changes to source code over time.

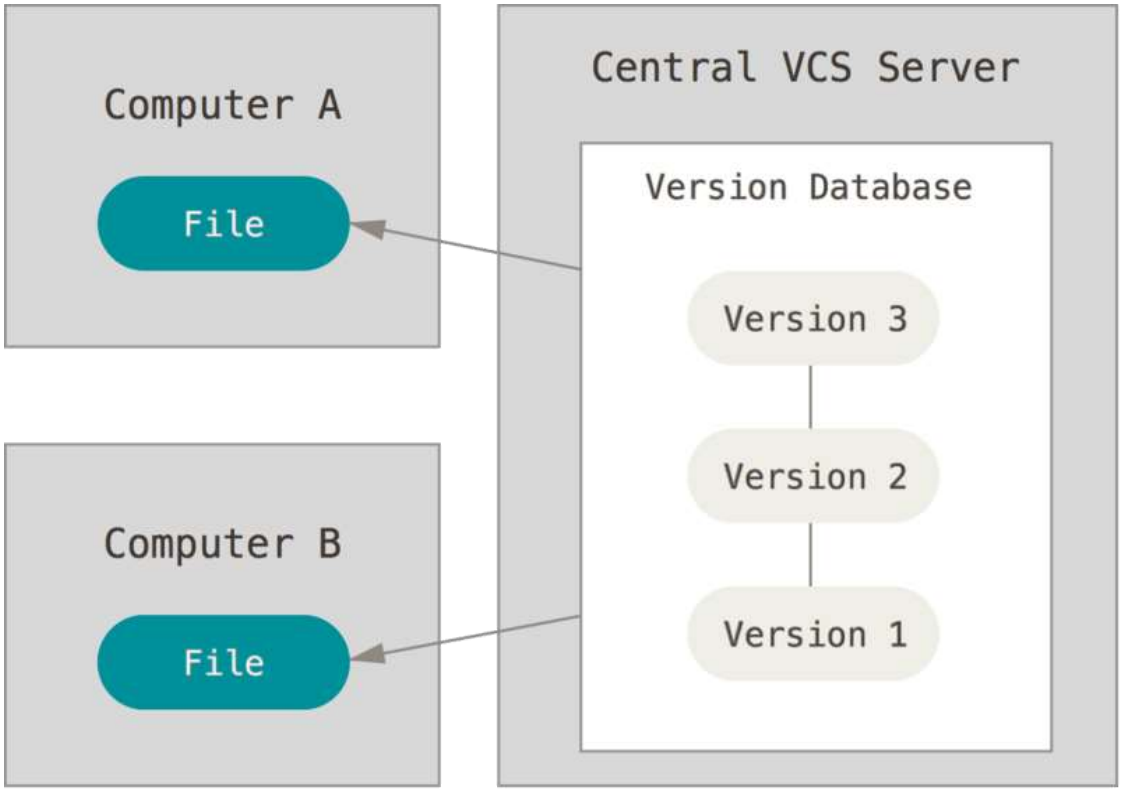


Local VCS

SOURCE CODE MANAGEMENT

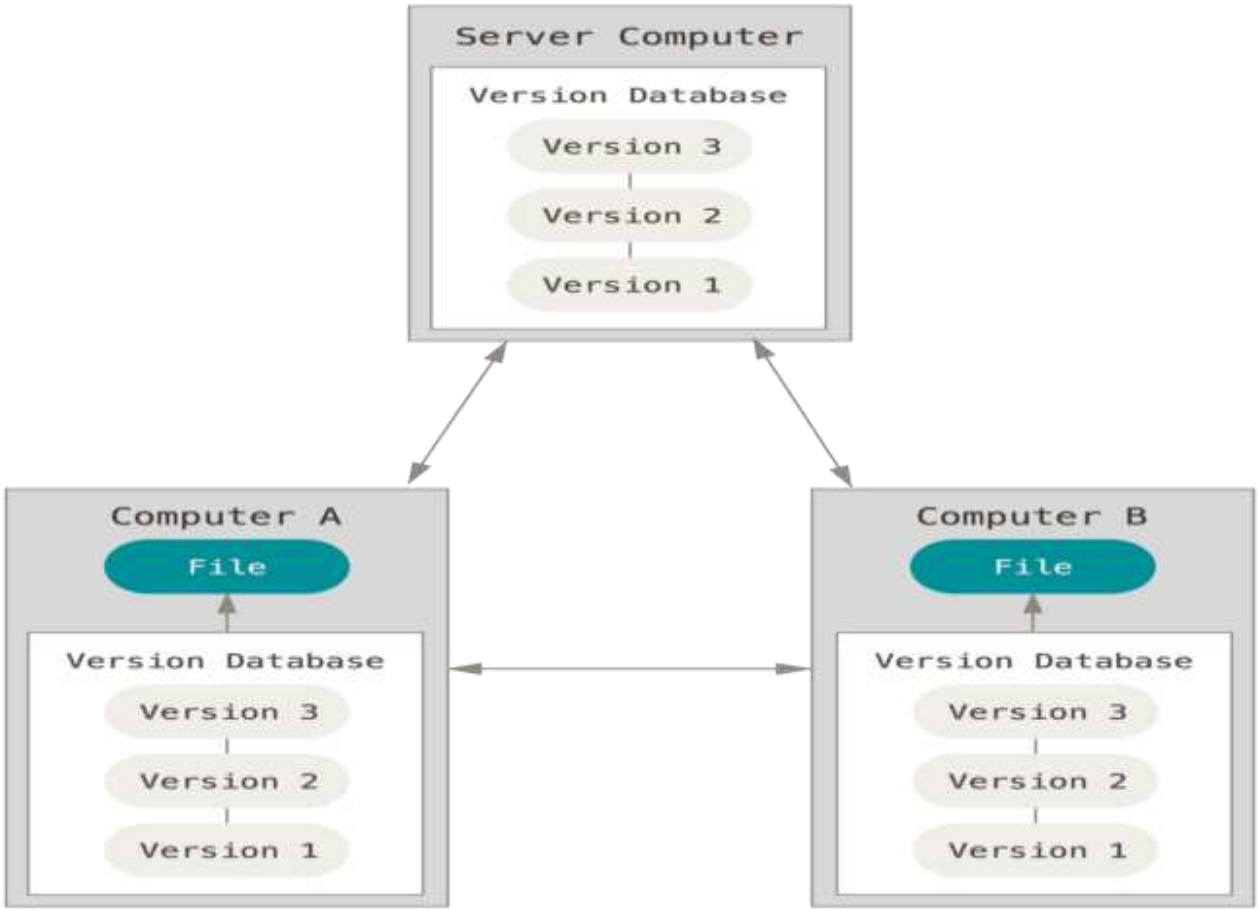
Centralized Version Control System [GIT]

→ Peer to peer approach



SOURCE CODE MANAGEMENT

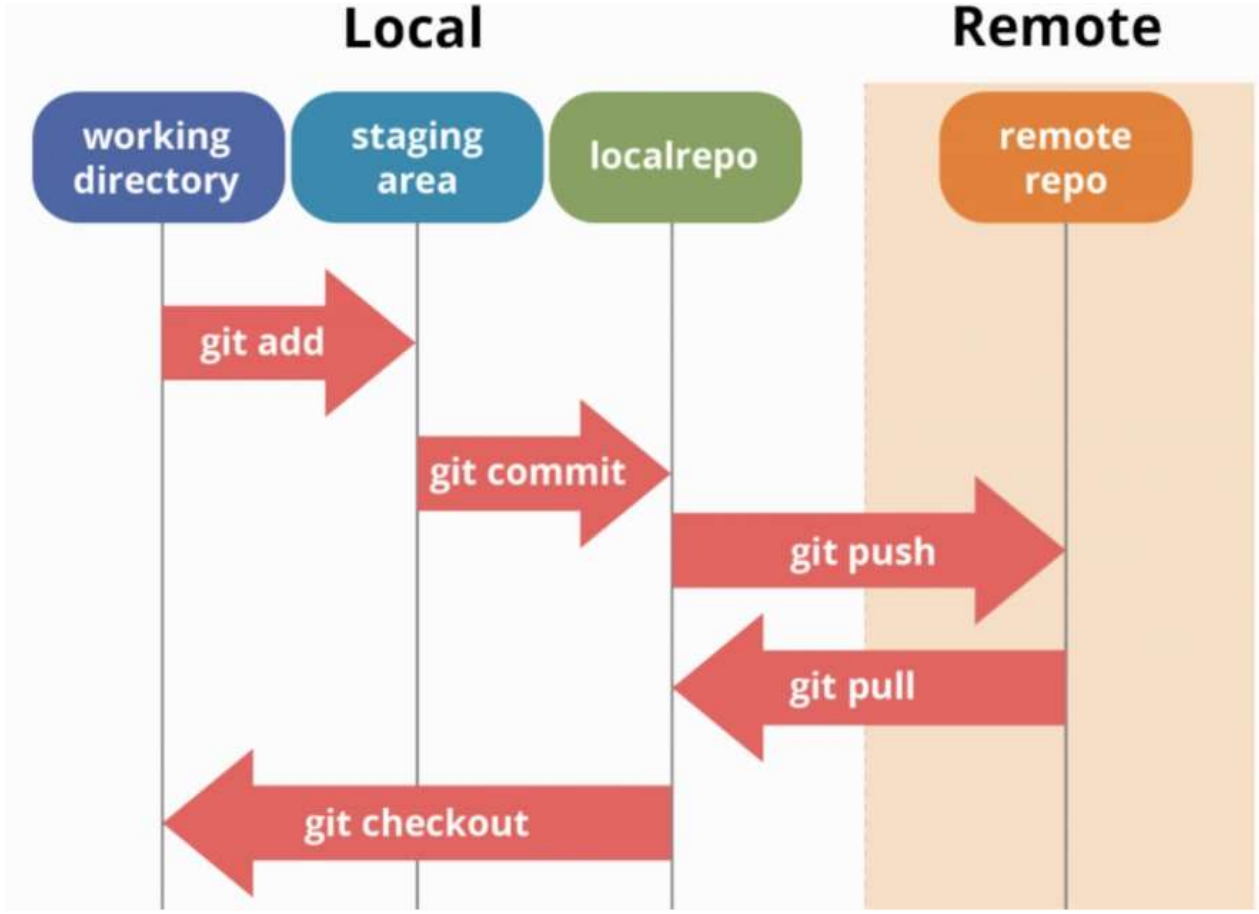
Distributed Version Control System [GIT]



SOURCE CODE MANAGEMENT

GIT	GITHUB
1. A Distributed VCS	1. A remote server for source code repository
2. A software tool	2. A service
2. Installed on our local system	2. Hosted on the web
3. Used to manage versions of source code	3. Used to have a copy of the local repository code stored on the website
4. Uses command line to interact with files	4. Provides a graphical interface to store files

GIT workflow



GIT Repositories

While working on Git, we actively use two repositories.

- **Local repository**: The local repository is present on our computer and consists of all the files and folders. This Repository is used to make changes locally, review history, and commit when offline.
- **Remote repository**: The remote repository refers to the server repository that may be present anywhere. This repository is used by all the team members to exchange the changes made.

Both repositories have their own set of commands. There are separate Git Commands that work on different types of repositories.

GIT Commands: Working with Local Repositories

1. **git init**

→The command *git init* is used to create an empty Git repository.

→After the git init command is used, a .git folder is created in the directory with some subdirectories. Once the repository is initialized, the process of creating other files begins.

2. **git add**

→Add command is used after checking the status of the files, to add those files to the staging area.

→ Before running the commit command, "*git add*" is used to add any new or modified files.

3. **git commit**

→The commit command makes sure that the changes are saved to the local repository.

→ The command "*git commit –m <message>*" allows you to describe everyone and help them understand what has happened.

4. **git status**

→ The *git status* command tells the current state of the repository.

→ The command provides the current working branch. If the files are in the staging area, but not committed, it will be shown by the git status. Also, if there are no changes, it will show the message no changes to commit, working directory clean.

GIT Commands: Working with Local Repositories

5. **git config**

→The *git config* command is used initially to configure the user.name and user.email. This specifies what email id and username will be used from a local repository.

→ When *git config* is used with *--global* flag, it writes the settings to all repositories on the computer.

git config --global user.name "any user name"

git config --global user.email <email id>

6. **git branch**

→ The *git branch* command is used to determine what branch the local repository is on.

→ The command enables adding and deleting a branch.

Create a new branch

git branch <branch_name>

List all remote or local branches

git branch –a

Delete a branch

git branch -d <branch_name>

GIT Commands: Working with Local Repositories

7. **git checkout**

→ The *git checkout* command is used to switch branches, whenever the work is to be started on a different branch.

→ The command works on three separate entities: files, commits, and branches.

Checkout an existing branch
git checkout <branch_name>

Checkout and create a new branch with that name
git checkout -b <new_branch>

8. **git merge**

→ The *git merge* command is used to integrate the branches together.The command combines the changes from one branch to another branch.

→ It is used to merge the changes in the staging branch to the stable branch.

git merge <branch_name>

9. **git log**

→ The *git log* command shows the order of the commit history for a repository.

→ The command helps in understanding the state of the current branch by showing the commits that lead to this state.

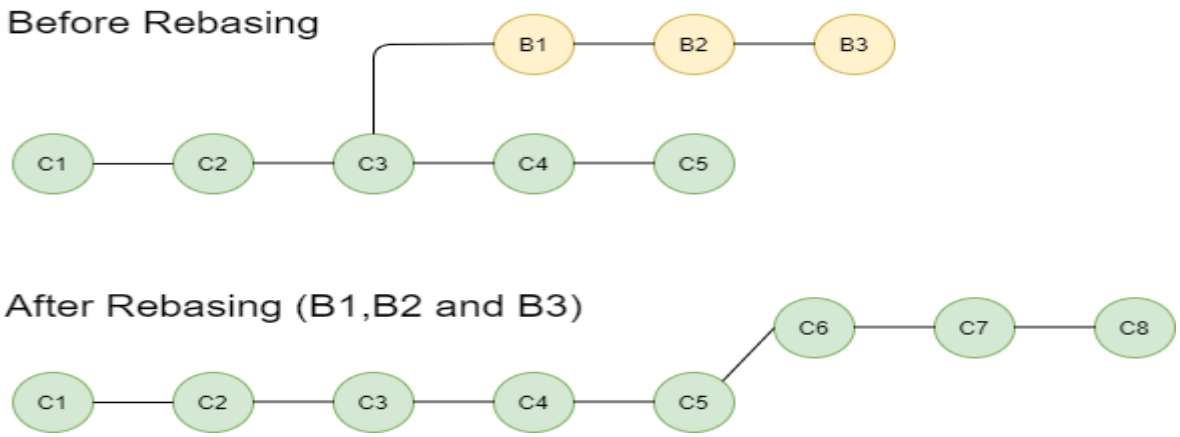
git log

GIT Commands: Working with Local Repositories

10. **git rebase**

→ The git rebase is a process of integrating a series of commits on top of another base tip. It takes all the commits of a branch and appends them to the commits of a new branch.

→git rebasing looks as follows:



git rebase <branchname>

GIT Commands: Working with Local Repositories

11. **git stash**

→ The *git stash* command takes your modified tracked files and saves it on a pile of incomplete changes that you can reapply at any time. To go back to work, you can use the *stash apply*.

→ The *git stash* command will help a developer switch branches to work on something else without committing to incomplete work.

Store current work with untracked files

git stash

Bring stashed work back to the working directory

git stash pop →Cut operation

git stash apply →Copy Operation

#To see the list of stashed work files

git stash list

GIT Commands: Working with Remote Repositories

1. **git remote**

→ The *git remote* command is used to create, view, and delete connections to other repositories.

→ The connections here are not like direct links into other repositories, but as bookmarks that serve as convenient names to be used as a reference.

git remote add origin <address>

2. **git push**

→ The command *git push* is used to transfer the commits or pushing the content from the local repository to the remote repository.

→ The command is used after a local repository has been modified, and the modifications are to be shared with the remote team members.

git push -u origin master

3. **git clone**

→ The *git clone* command is used to create a local working copy of an existing remote repository.

→ The command downloads the remote repository to the computer. It is equivalent to the *git init* command when working with a remote repository.

git clone <remote_URL>

GIT Commands: Working with Remote Repositories

4. **git pull**

→ The *git pull* command is used to fetch and merge changes from the remote repository to the local repository.

→ The command "git pull origin master" copies all the files from the master branch of the remote repository to the local repository.

git pull <branch_name> <remote URL>

1. What is Devops?

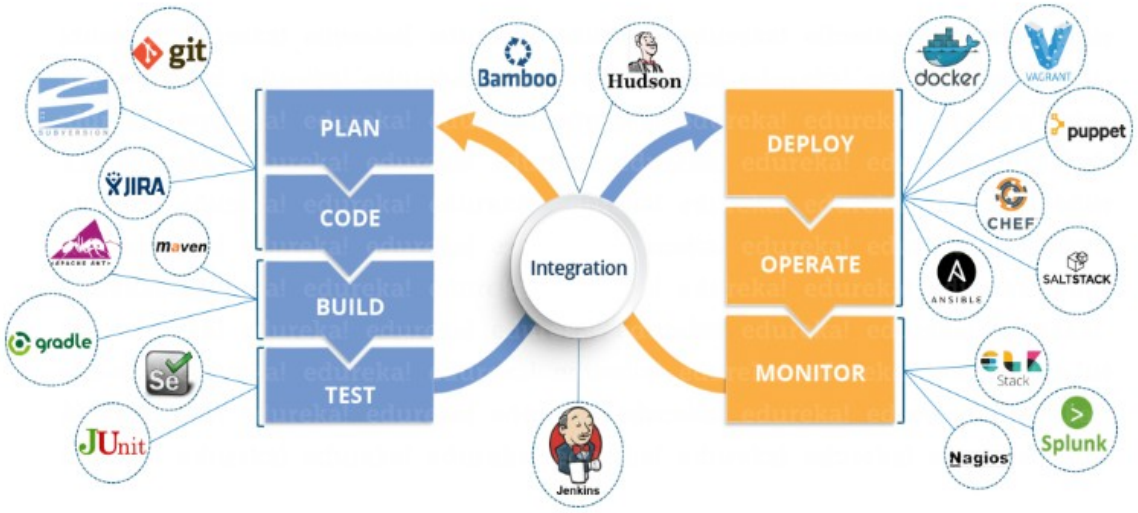
The DevOps is a combination of two words, one is software Development, and second is Operations. This allows a single team to handle the entire application lifecycle, from development to testing, deployment, and operations. DevOps helps you to reduce the disconnection between software developers, quality assurance (QA) engineers and system administrators.

DevOps, a combination of 'development and operations', is a mix of practices, tools and cultural philosophies that enable an organization to quickly deliver applications and services. It also helps products go from the drawing board to market at a faster pace than traditional software development because operations and development engineers work closely together in the entire lifecycle, from design through the development process to production support.

Generally, DevOps represents a change in the culture of IT from one of separateness to one of working as a team. It's about creating rapid service delivery using Agile, lean practices in a system-oriented approach. DevOps also aims to use technology, especially automation tools, to leverage highly programmable and dynamic infrastructure.

Various DevOps tools such as Git, Ansible, Docker, Puppet, Jenkins, Chef, Nagios, and Kubernetes.

2. DevOps architecture

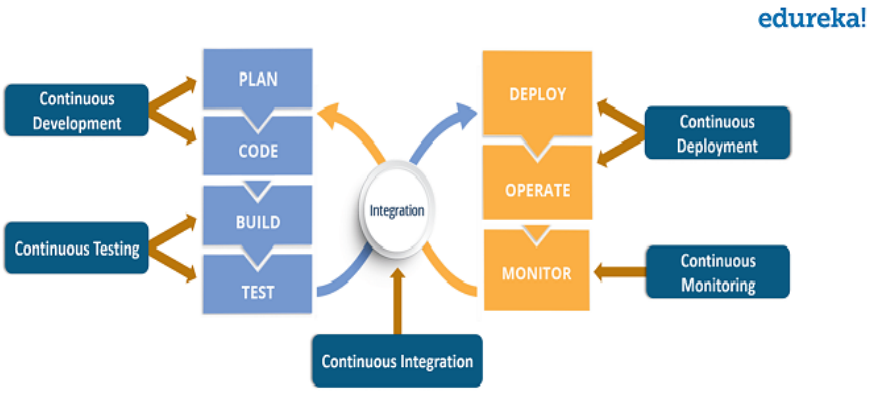


The various components that are used in the DevOps architecture:

i. Plan: Agile methodology helps plan development and improve productivity.
ii. Code: Good practices like Git make code use, tracking, and reusability easier.
iii. Build: Cloud and shared resources are used to control resource usage.
iv. Test: Automated testing saves time and prepares the app for production.
v. Deploy: Automated deployment captures insights and optimizes performance.
vi. Operate: DevOps collaborates throughout the service lifecycle.
vii. Monitor: Continuous monitoring reduces risk of failure and tracks app health.

3. DevOps Lifecycle

The DevOps lifecycle consists of seven phases: Continuous Development, Continuous Integration, Continuous Testing, Continuous Monitoring, Continuous Feedback, Continuous Deployment, and Continuous Operations. Each phase plays a crucial role in the software development process and contributes to the overall efficiency and effectiveness of the DevOps approach.



i. *Continuous Development* involves planning and coding the software, where the vision of the project is decided and the developers begin coding the application. Tools used- Git, Mercurial, SVN
ii. *Continuous Testing* involves constantly testing the developed software for bugs using automation testing tools such as TestNG, JUnit, and Selenium
iii. *Continuous Integration* is the heart of the DevOps lifecycle and involves committing changes to the source code frequently and building the code, including unit testing, integration testing, code review, and packaging. Tools used – Jenkins, Bamboo
iv. *Continuous Deployment* involves deploying the code to the production servers and ensuring that the code is correctly used on all servers. Configuration management tools play a crucial role in executing tasks frequently and quickly in this phase. Tools used – Docker, Ansible, Kubernetes, Chef, Puppet
v. *Continuous Monitoring* involves monitoring the operational factors of the DevOps process and recording important information about the use of the software. Tools used – Nagios

4. Version Control System

What is “version control”, and why should you care? 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. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the 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. In addition, you get all this for very little overhead.

Local Version Control Systems

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 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 to files under revision control.

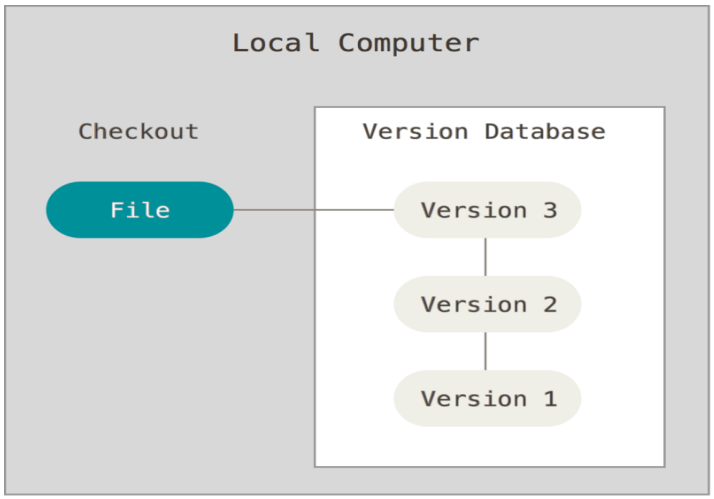


Figure 1. Local version control diagram

One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

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.

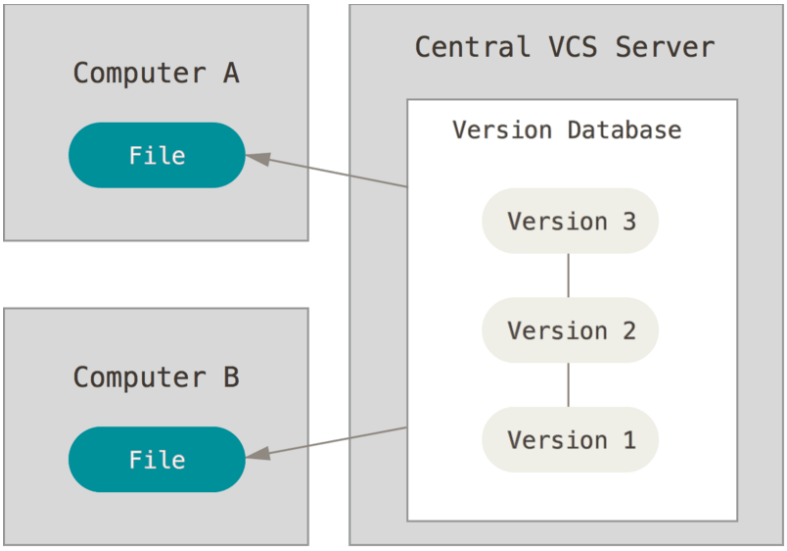


Figure 2. Centralized version control diagram

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. 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 versioned 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 VCSs 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 Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial 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.

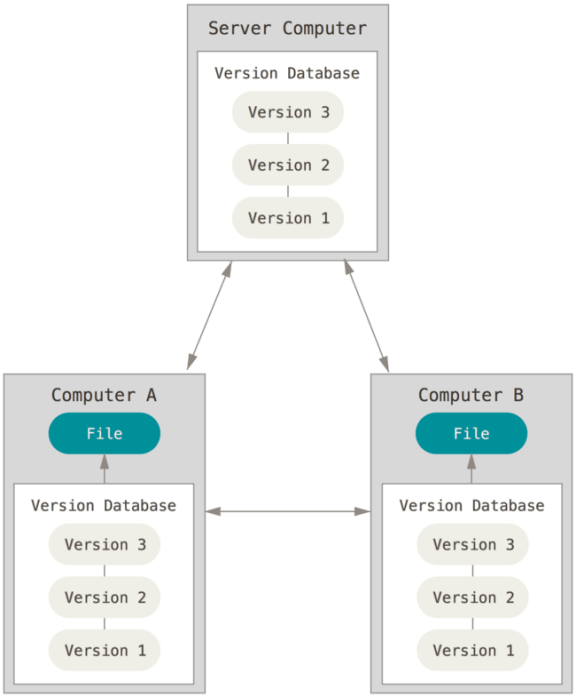


Figure 3. Distributed version control diagram

Furthermore, many of these systems deal 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.

`git rebase master`:

- When you run **git rebase master** from your feature branch, it means you want to bring in changes from the **master** branch into your feature branch.
- This is commonly used when you want to keep your feature branch up to date with the latest changes in the **master** branch.
- During this rebase, Git will replay your feature branch's commits on top of the current **master** branch tip.
- This results in a linear commit history where your feature branch appears to be "based on" the latest **master** branch commit.
- Example:

```
(master) A---B---C---D---E
                       \
(feature)               X---Y
```

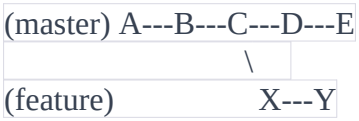
After **git rebase master**:

```
(master) A---B---C---D---E
                       \
(feature)               X'---Y'
```

git rebase feature:
Running **git rebase feature** from your **master** branch would mean you want to incorporate changes from your feature branch into the **master** branch.
This is less common and is usually done when you want to bring specific features or changes from a feature branch back into the **master** branch.
During this rebase, Git will replay your **master** branch's commits on top of the latest commit in your feature branch.

It can be useful when your feature branch contains a substantial change or feature that you want to include in the **master** branch.

Example:



After **git rebase feature**:

In summary, the target branch in **git rebase** determines which branch's changes will be replayed on top of the other. It's essential to choose the right direction based on your workflow and intentions. Typically, you'd use **git rebase master** to keep your feature branch up to date with **master** and **git rebase feature** less frequently when you want to merge feature changes into **master**.

Merge	Rebase
Git Merge lets you merge different Git branches.	Git Rebase allows you to integrate the changes from one branch into another.
Git Merge logs show you the complete history of commit merging.	Git Rebase logs are linear. As the commits are rebased, the history is altered to reflect this.
All the commits on a feature branch are combined into a single commit on the master branch.	All commits are rebased, and the same number of commits are added to the master branch.
Merge is best used when the target	Rebase is best used when the target

branch is supposed to be shared.	branch is private.
Merge preserves history.	Rebase rewrites history.

MODULE 1

Git: is one of the most sought-after DevOps tools used to handle small and large projects efficiently. Git is nothing without its commands, so here, all about Git commands that are needed to efficiently and effectively work on the tool.

Git is a widely used modern version control system for tracking changes in computer files.

Git makes it possible for several people involved in the project to work together and track each other's progress over time. In software development, the tool helps in Source Code Management. Git favors not only programmers but also non-technical users by keeping track of their project files.

While working on Git, we actively use two repositories.

- Local repository: The local repository is present on our computer and consists of all the files and folders. This Repository is used to make changes locally, review history, and commit when offline.
- Remote repository: The remote repository refers to the server repository that may be present anywhere. This repository is used by all the team members to exchange the changes made.

Both repositories have their own set of commands. There are separate Git Commands that work on different types of repositories.

Git has multiple levels of configuration:

1. Repository/Project level (Local) - repository/.git/config
2. User Account Level (Global Level) - Users/name/.gitconfig
3. System Level (Git Installation) - /usr/local/etc/.gitconfig

Git Commands: Working With Local Repositories

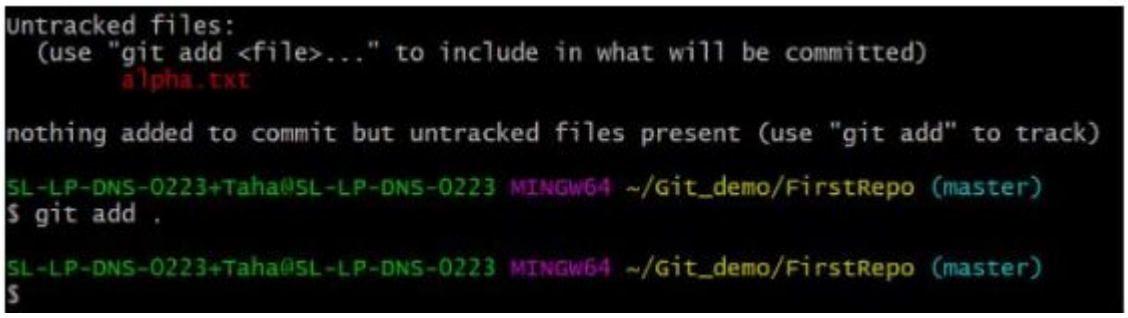
git init

- The command git init is used to create an empty Git repository.

- After the git init command is used, a .git folder is created in the directory with some subdirectories. Once the repository is initialized, the process of creating other files begins.

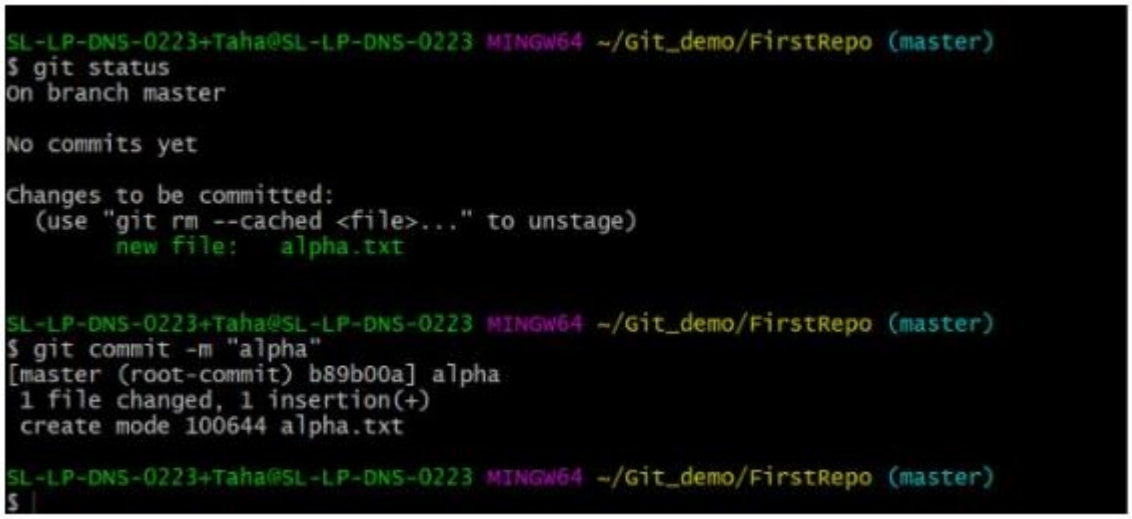
git add

- Add command is used after checking the status of the files, to add those files to the staging area.
- Before running the commit command, "git add" is used to add any new or modified files.



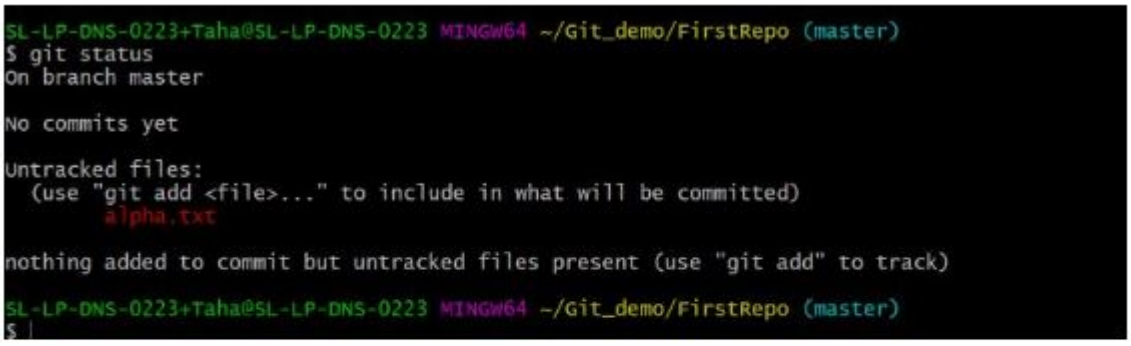
git commit

- The commit command makes sure that the changes are saved to the local repository.
- The command "git commit –m <message>" allows you to describe everyone and help them understand what has happened.



git status

- The git status command tells the current state of the repository.
- The command provides the current working branch. If the files are in the staging area, but not committed, it will be shown by the git status. Also, if there are no changes, it will show the message no changes to commit, working directory clean.

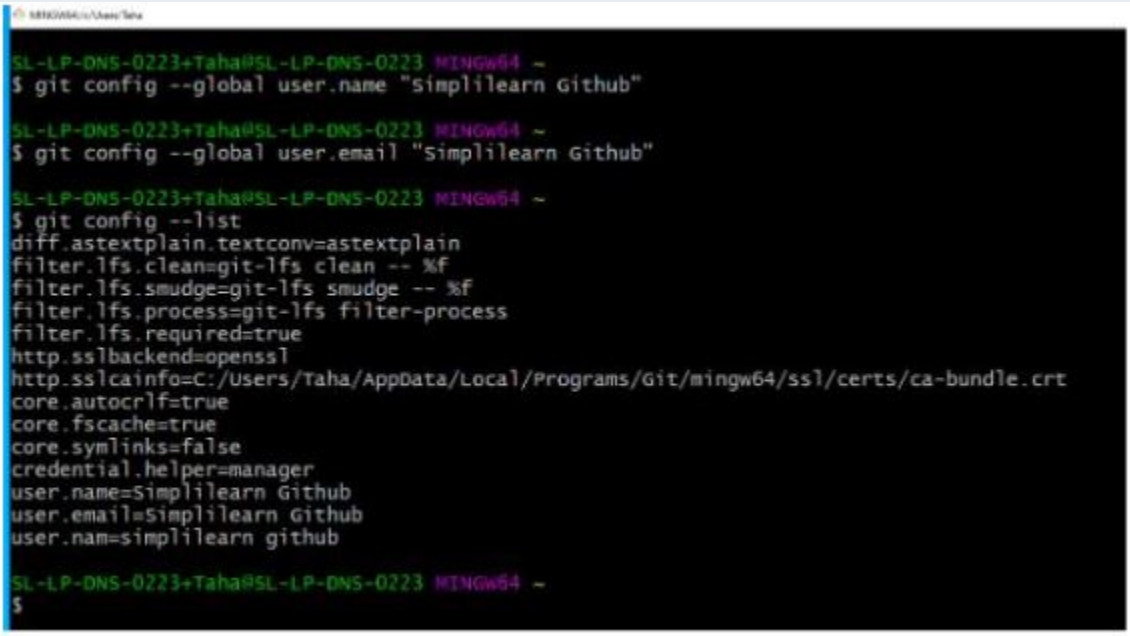


git config

- The git config command is used initially to configure the user.name and user.email. This specifies what email id and username will be used from a local repository.
- When git config is used with --global flag, it writes the settings to all repositories on the computer.

git config --global user.name “any user name”

git config --global user.email <email id>



git branch

- The git branch command is used to determine what branch the local repository is on.
- The command enables adding and deleting a branch.

```
# Create a new branch
git branch <branch_name>
```

```
# List all remote or local branches
git branch -a
```

```
# Delete a branch
git branch -d <branch_name>
```

git checkout

- The git checkout command is used to switch branches, whenever the work is to be started on a different branch.
- The command works on three separate entities: files, commits, and branches.

```
# Checkout an existing branch
 git checkout <branch_name>
```

```
# Checkout and create a new branch with that name
```

```
git checkout -b <new_branch>
```

git merge

- The git merge command is used to integrate the branches together. The command combines the changes from one branch to another branch.
- It is used to merge the changes in the staging branch to the stable branch.

```
git merge <branch_name>
```

Git Commands: Working With Remote Repositories

git remote

- The git remote command is used to create, view, and delete connections to other repositories.
- The connections here are not like direct links into other repositories, but as bookmarks that serve as convenient names to be used as a reference.

```
git remote add origin <address>
```

git clone

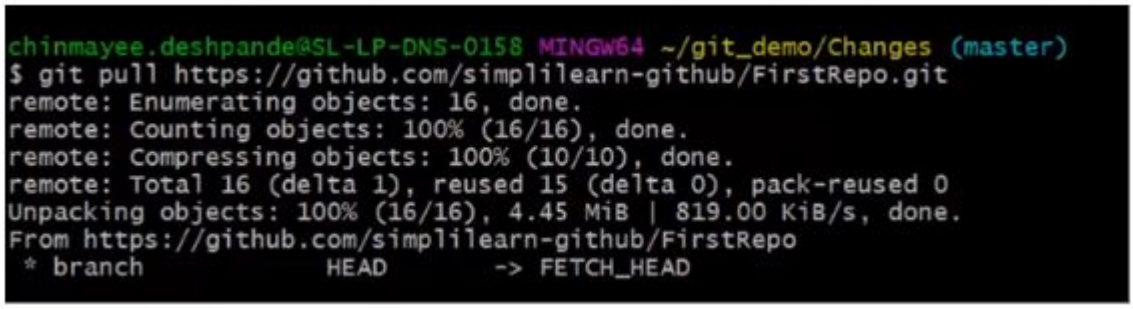
- The git clone command is used to create a local working copy of an existing remote repository.
- The command downloads the remote repository to the computer. It is equivalent to the Git init command when working with a remote repository.

```
git clone <remote_URL>
```

git pull

- The git pull command is used to fetch and merge changes from the remote repository to the local repository.
- The command "git pull origin master" copies all the files from the master branch of the remote repository to the local repository.

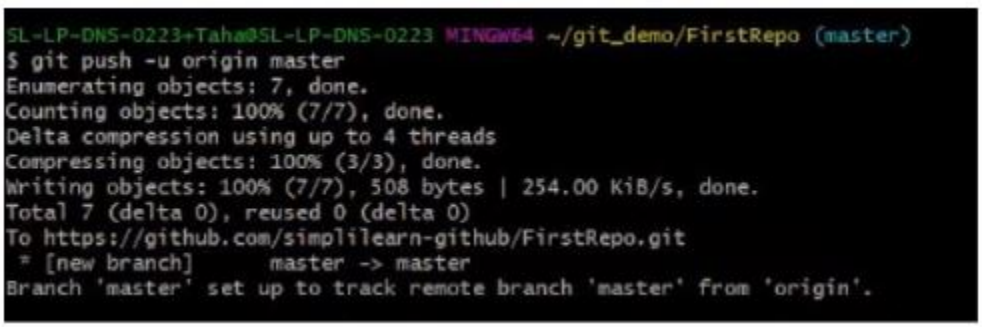
git pull <branch_name> <remote URL>



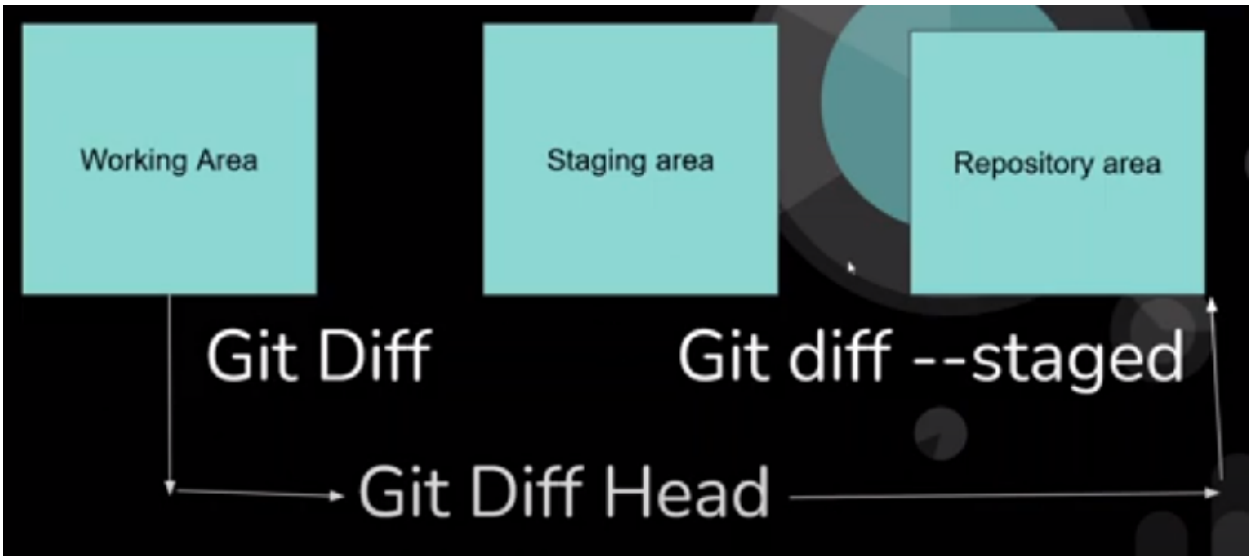
git push

- The command git push is used to transfer the commits or pushing the content from the local repository to the remote repository.
- The command is used after a local repository has been modified, and the modifications are to be shared with the remote team members.

git push -u origin master



Git diff



Git diff: To check the changes in the working area and the staged area

Git diff - - staged : to check the changes between the staging area and repository area

Git diff head: to check the changes between working area and repository area.

EXPERIMENT 1. ***Creating and pushing changes from local repository to remote repository in GitHub***

Problem statement → Create a repository in your local system and Github. Get inside the directory and create a file. Commit changes and push changes to the remote repository

SOLUTION :

git init //initialize the

git status

notepad text1.txt

git add text1.txt

git commit -m "my first commit in Demo1"

git log →. To check commit log

git remote add origin "https://github.com/namratasgit/demorepo.git"

git push origin master

EXPERIMENT 2a. ***Branching and Merging*** **[continue experiment 1]**

Problem statement → Create and move to a feature branch from the master branch [refer to exp 1], Make necessary changes, commit changes and push changes to remote repository. Switch back to master branch and merge recent commit of feature branch on master branch.

SOLUTION :

git branch NewBranch //*creating a new branch*

git checkout NewBranch //*switched to newbranch*

notepad demo2.txt

git add .

git commit -m "added demo.txt, my first commit"

git push origin NewBranch //*push New branch commits to remote repository*

git checkout master //*switch to master branch*

git merge NewBranch //*merge changes of feature branch with master branch*

git push origin master //*push the changes to the master branch*

git branch -d NewBranch //*delete the feature branch from local repository*

git push origin --delete NewBranch //*delete the feature branch from remote* repository

EXPERIMENT 2b. ***Branching and Merging***

Problem statement → Create an add.py and sub.py file in master branch. Create a feature branch, Switch to it and create a multiply.py and div.py file. Move back to master branch, create a mod.py file and perform a merge. Finally push the updated master branch to remote repository in Github

Solution :

//Create a repository in your Github account and local system. Move inside the repo in your local system, open git bash and perform the following actions-

//Master
git init
notepad add.py
git add .
git commit –m “Commit on add”
notepad sub.py
git add .
git commit –m “Commit on sub”
git branch feature
git checkout feature

//Feature
notepad multiply.py
git add .
git commit –m “Commit on multiply”
notepad div.py
git add .
git commit –m “Commit on div”
git checkout master

// Master
notepad mod.py
git add .
git commit –m “Commit on mod”
git merge feature
git remote add origin “http:.....” // add remote repo http link
git push origin master

EXPERIMENT 3. ***Resolving merge conflicts***

a. Create a new repository in github and add a simple text file with contents-

   Hello!
   I am developer_name
   I am from developer_city

b. Get into your local systems and create two separate folders named “Alice” and “Bob” in two separate drives.
c. Get inside both folders and open git bash.
d. Follow the steps below for each user mentioned-
   i. Both Alice and Bob clones the remote repository from Github
   ii. Both get into the cloned repository.
   iii. Alice creates a feature branch while Bob uses choses to stay in the master branch.

iv. **Feature** → Alice switches to the feature branch, creates a text file , commits the changes and switches back to the master branch
Master → Bob creates a text file, commits on the changes and performs a push to the remote repository.

v. **Master** → Alice pulls the remote repository, merges the feature branch 'dev1branch' to the master branch, gets a conflict and resolves it

vi. **Master** → After resolving conflict, the changed file is committed and push to the remote repository.

SOLUTION :

Alice

- git clone https://github.com/namratasgit/merge-conflict.git
- cd merge-conflict
- git branch dev1branch
- git checkout dev1branch
- vi Test.txt
- git add .
- git commit –m "Alice's Commit"
- git checkout master

- git pull
- cat Test.txt
- git checkout dev1branch
- cat Test.txt
- git checkout master
- git merge dev1branch

// Auto-merging Test.txt
CONFLICT (content): Merge conflict in Test.txt
//Edit the file.

- git add .
- git commit –m "final commit"
- git push origin master

Bob

- git clone https://github.com/namratasgit/merge-conflict.git
- cd merge-conflict
- vi Test.txt
- git add .
- git commit –m "bob's Commit"
- git push origin master

EXPERIMENT 4 : Git Rebase

➔ Create a repository in your local machine and perform the following steps-

//Master
git init
notepad a.txt
git add .
git commit –m "First commit"
git branch feature

//Feature
notepad b.txt
git add .
git commit –m "Seond commit"
git checkout master

//Master
notepad c.txt
git add .
git commit –m “Third commit”
git checkout feature
//Feature
git rebase master
git log
git checkout master

//Master
notepad d.txt
git add .
git commit –m “Fourth commit”
git log *//to check commit history on master*
git checkout feature

//Feature
notepad e.txt
git add .
git commit –m “Fifth commit”
git log *//to check commit history on feature*
git rebase master
git log

EXPERIMENT 5: Git stash

- ➔ Create a new repository in github and add a simple text file
- ➔ Get inside your working directory in local system and perform the following-

git clone https://github.com/namratasgit/stashrepo.git *//cloning a remote repo to local system*
cd stashrepo *// get into the cloned repo*
git init
git branch feature1 *//create new branch – feature1*
git checkout feature1 *// switch to feature branch*

// perform the following actions
notepad second.txt
git add .
git stash *// stash the second.txt file*
notepad third.txt
git add .
git commit –m “F2” *//commit the third.txt file*
git log
ls
git stash apply
ls
git stash
git checkout main *// switch to main branch*
git merge main *// perform merge*
git branch feature2 *//create new branch – feature2*
git checkout feature2 *// switch to feature branch*

//perform the following actions
notepad fourth.txt
git add .
git stash *// stash the fourth.txt file*
git checkout main *//switch to main branch*
git merge main // perform merge on main
git push origin main //push changes to github repo

Module 2
DOCKER

Contents-

1. Continuous Deployment
2. Why Docker?
3. Virtual Machines vs Docker
4. Container and Image
5. Docker Client, Docker Daemon, Docker Registry
6. Docker Pre-requisites
7. Docker Installation
8. Docker Commands
9. Dockerfile

1. Why Docker?

Before Docker

1. Why Docker?

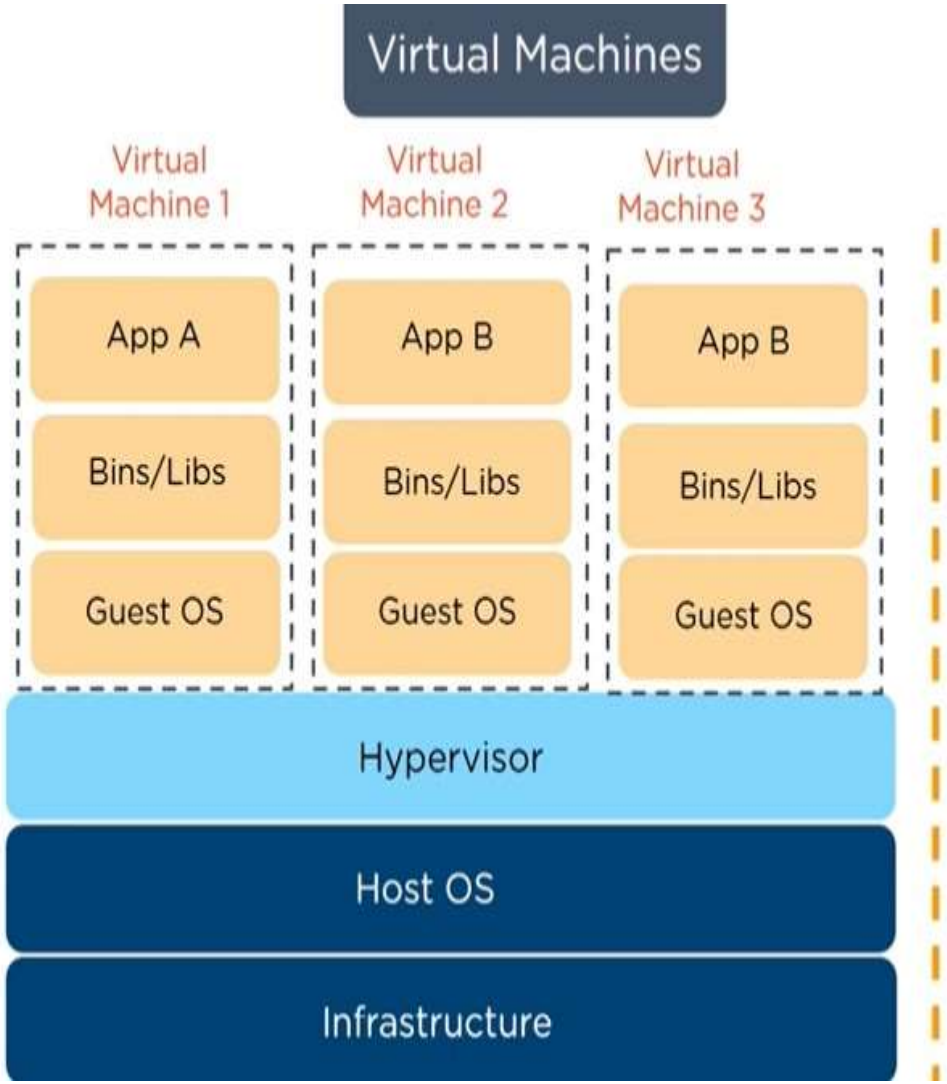
Before Docker

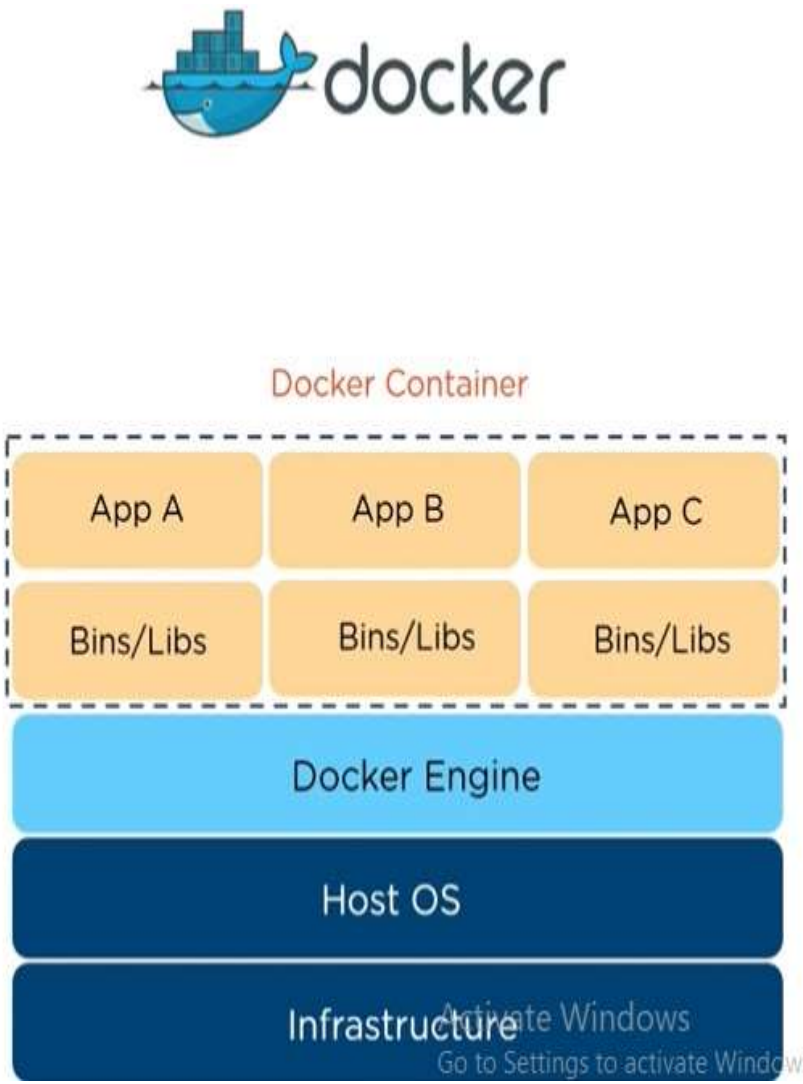
The code doesn't work on the other system due to the difference in computer environments
So, what could be the solution to this?

1. Why Docker?

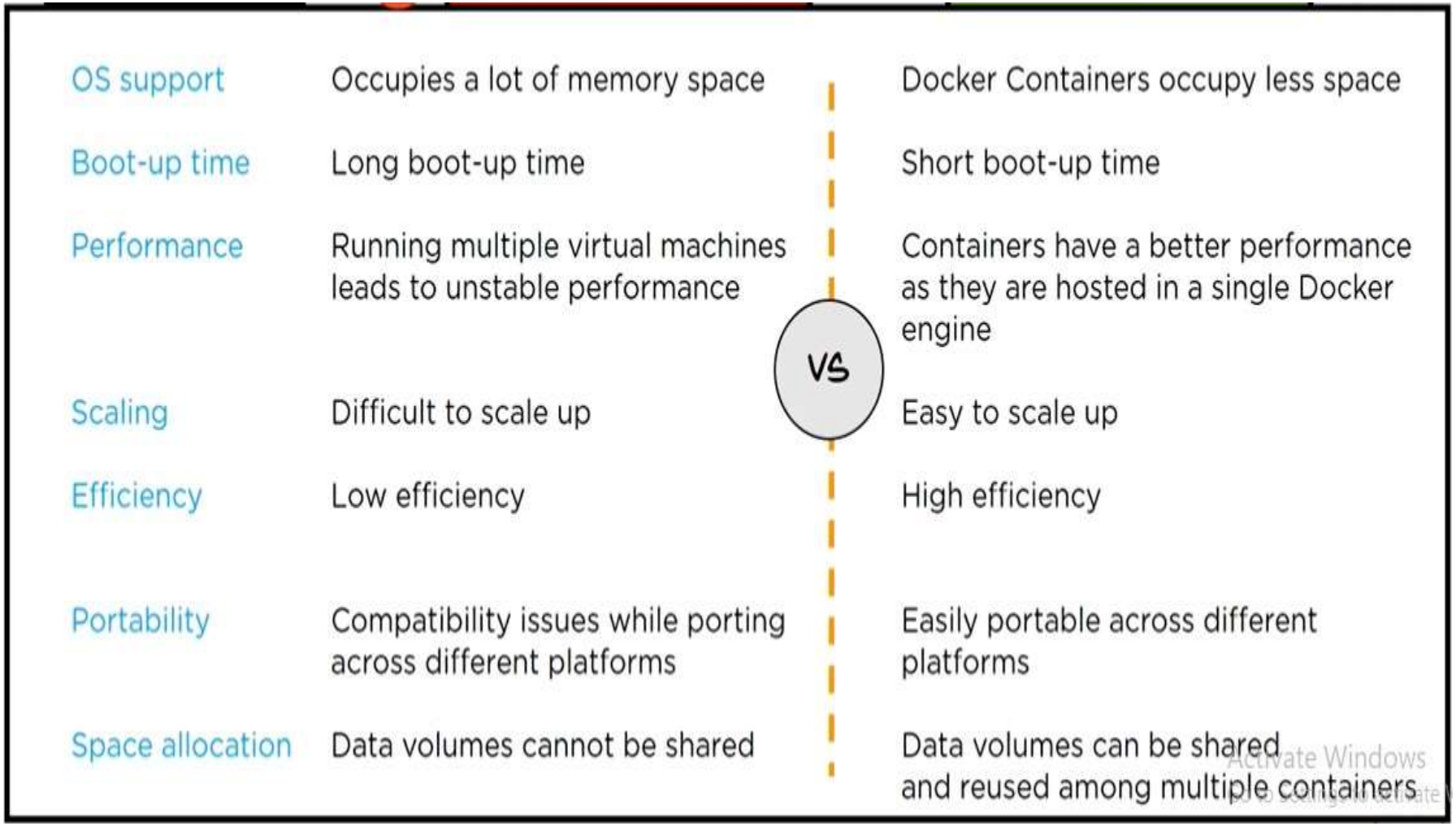
Solution???

2. Virtual Machines vs Docker



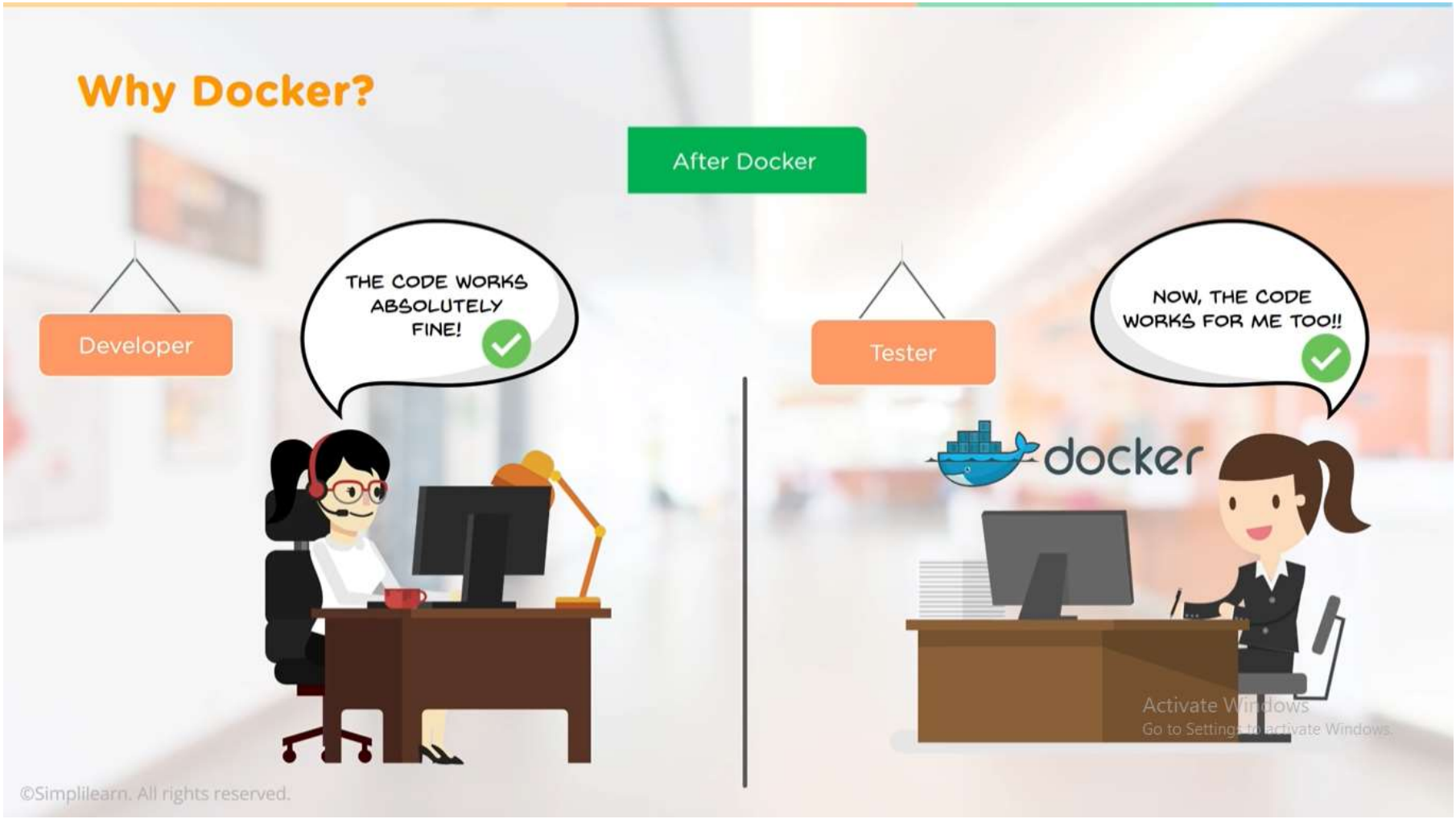


2. Virtual Machines vs Docker

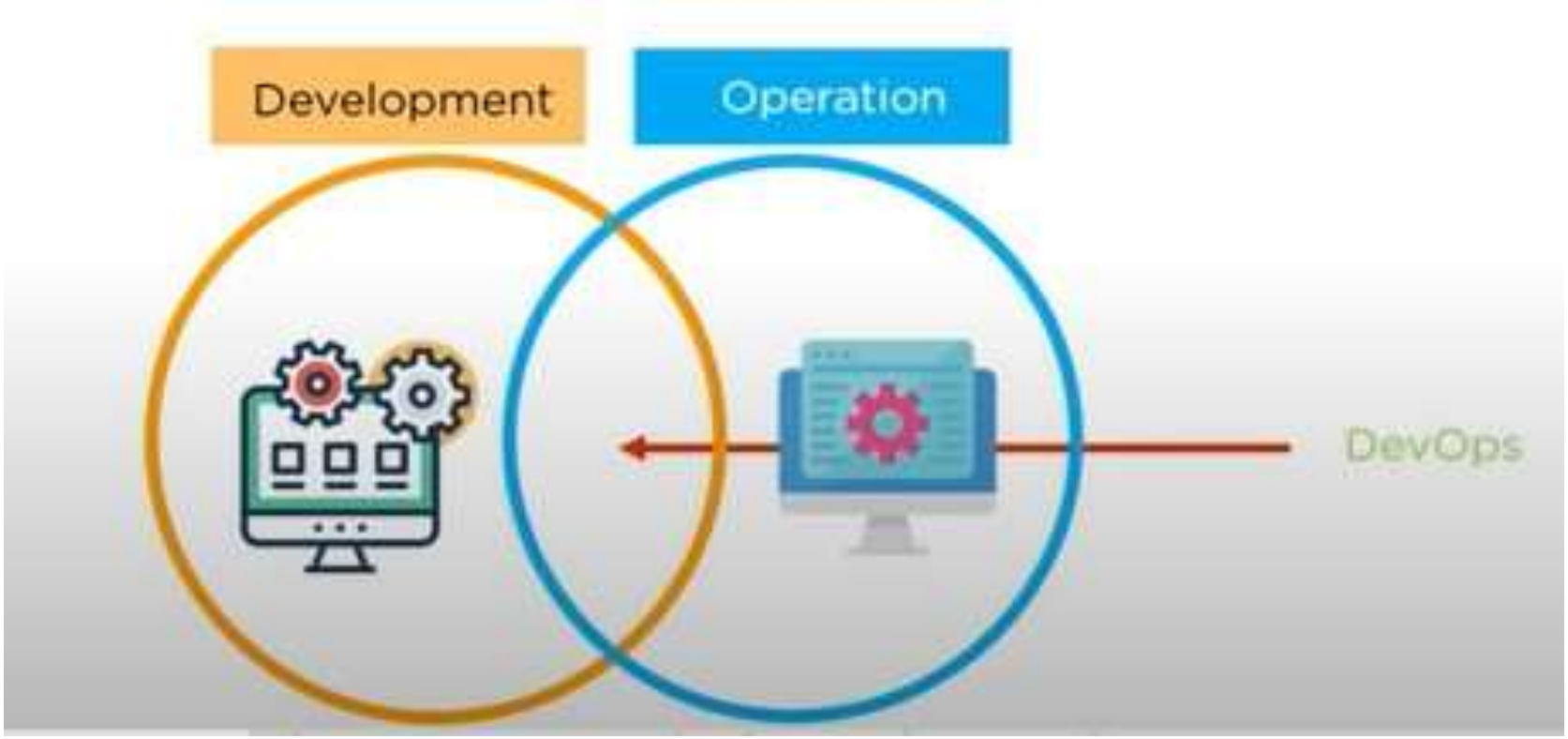


2. Virtual Machines vs Docker

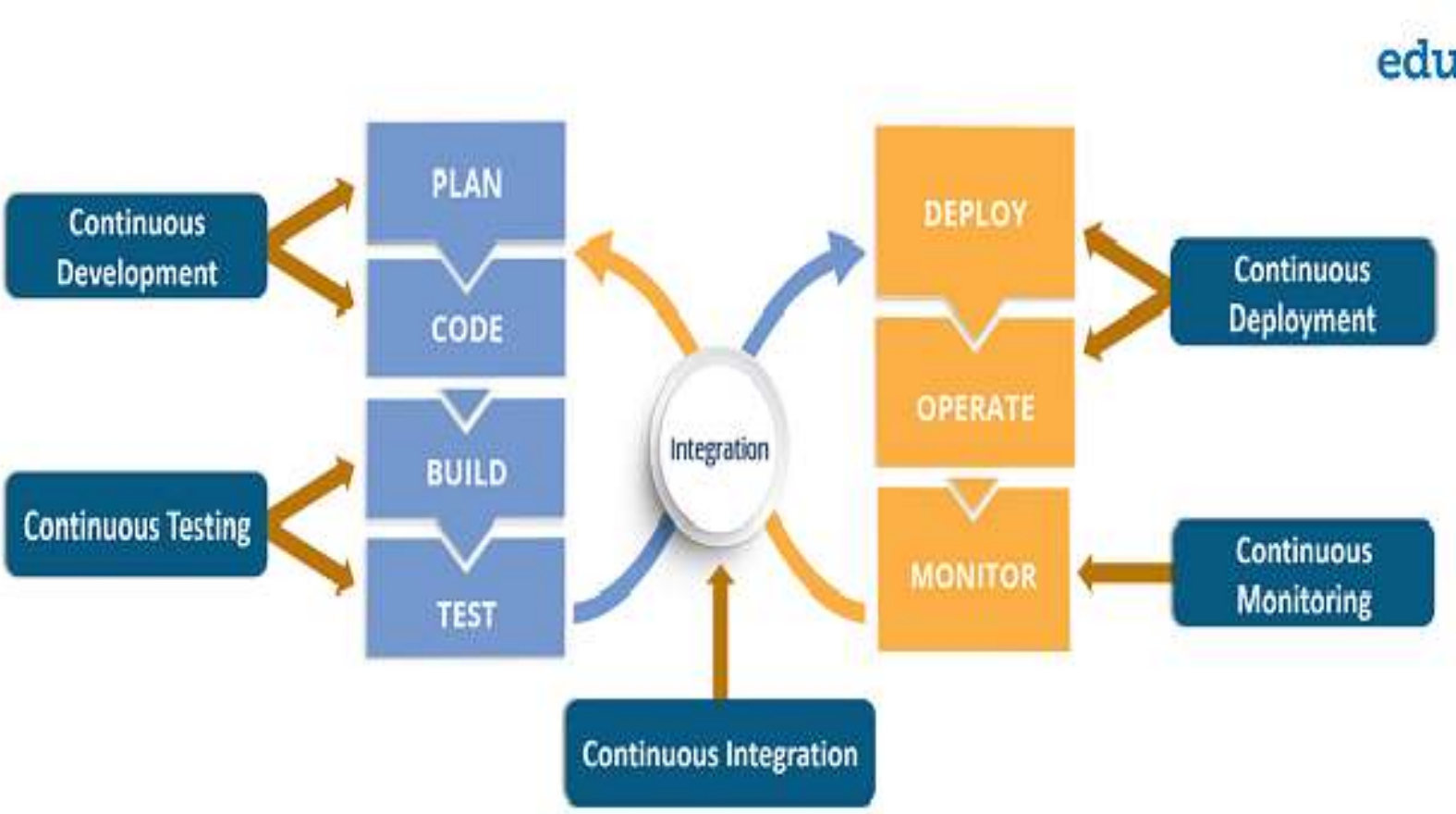
After Docker



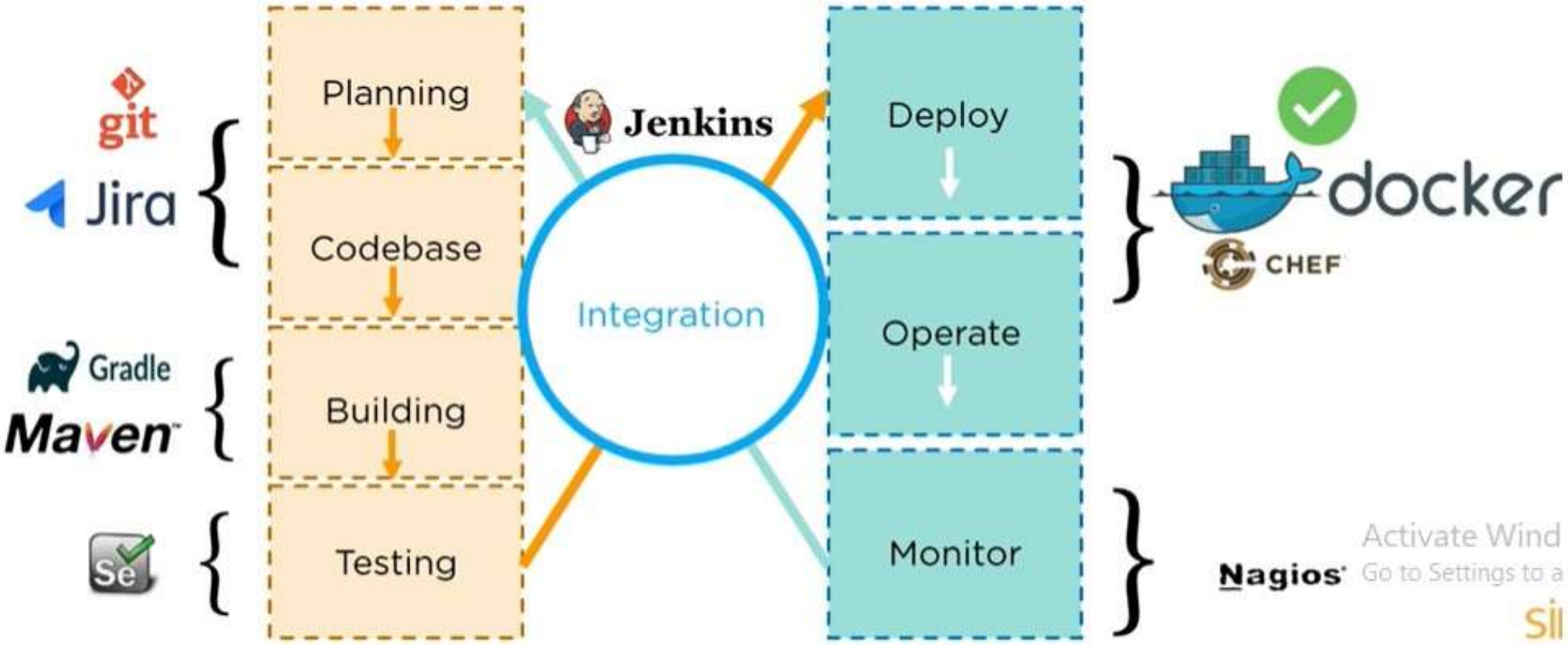
3. Continuous Deployment using Docker



3. Continuous Deployment using Docker



3. Continuous Deployment using Docker



3. Continuous Deployment using Docker

Docker is a tool which is used to automate the deployment of applications in lightweight containers so that applications can work efficiently in different environments

4. Docker Containers and Images

Containers

→OS level Virtualization technology that detaches applications and all of their dependencies from the rest of the system

→a portable executable package which includes applications and their dependencies.

4. Docker Containers and Images

Images

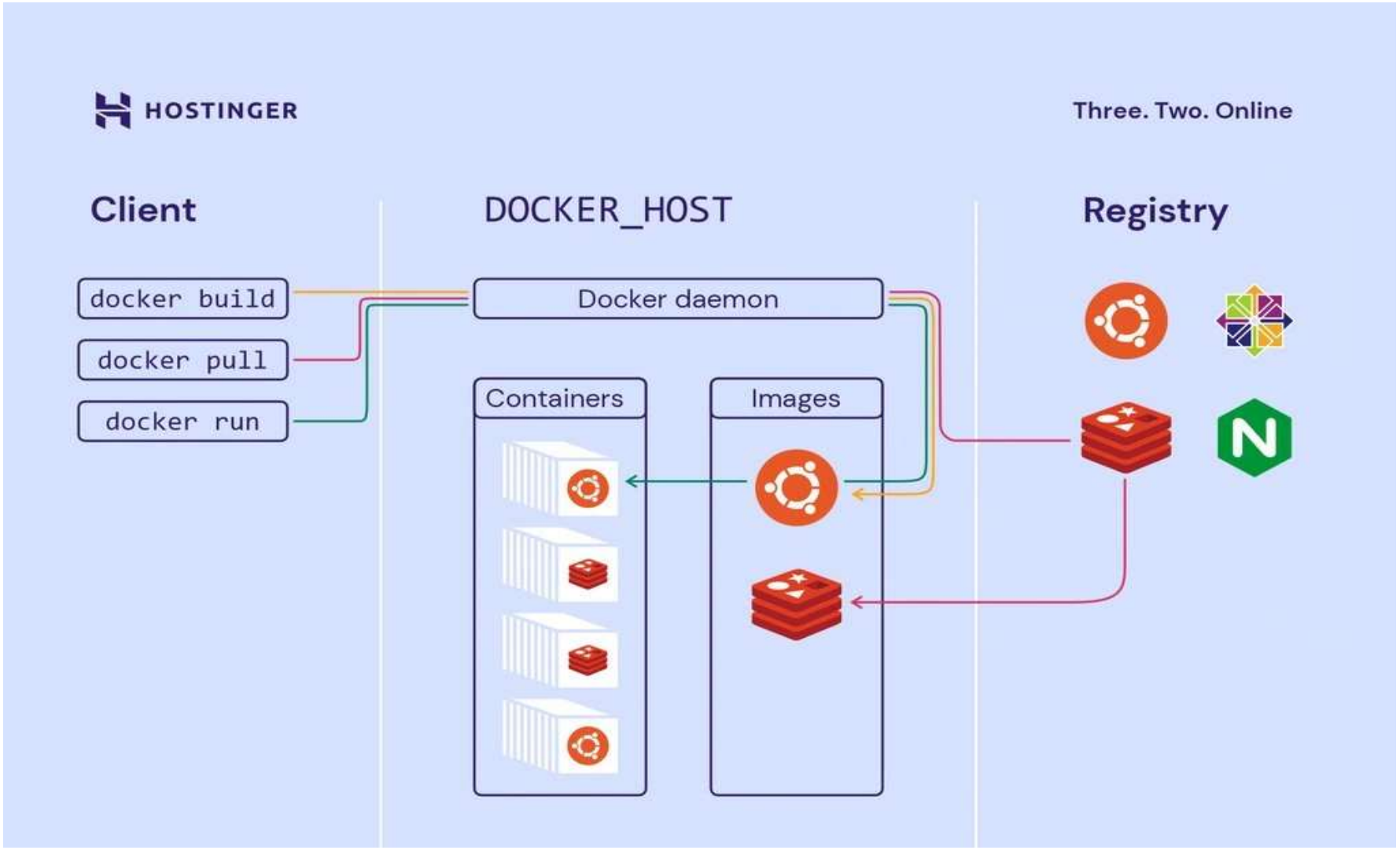
→a file used to execute code in a Docker container.

→Docker images act as a set of instructions to build a Docker container, like a template.

→Docker images also act as the starting point when using Docker.

→All of the configuration and instructions to start or stop containers are dictated by the Docker image. Whenever a user runs an image, a new container is created.

5. Docker Client, Docker Daemon, Docker Registry



5. Docker Client, Docker Daemon, Docker Registry

- The Docker architecture consists of four main components along with Docker containers which we've covered earlier.
- **Docker client** – the main component to create, manage, and run containerized applications. The Docker client is the primary method of controlling the Docker server via a CLI like Command Prompt (Windows) or Terminal (macOS, Linux).
- **Docker server** – also known as the Docker daemon. It waits for REST API requests made by the Docker client and manages images and containers.

5. Docker Client, Docker Daemon, Docker Registry

- **Docker images** – instruct the Docker server with the requirements on how to create a Docker container. Images can be downloaded from websites like **Docker Hub**. Creating a custom image is also possible – to do it, users need to create a Dockerfile and pass it to the server. It’s worth noting that Docker doesn’t clear any unused images, so users need to delete image data themselves before there’s too much of it.

5. Docker Client, Docker Daemon, Docker Registry

- **Docker registry** – an open-source server-side application used to host and distribute Docker images. The registry is extra useful to store images locally and maintain complete control over them. Alternatively, users can access the aforementioned Docker Hub – the world's largest repository of Docker images.

5. Dockerfile

i. A Dockerfile is text file with instructions to build a docker image
ii. When we run a docker file, a docker image is created
iii. When we run a docker image, a docker container is created

5. Dockerfile

Steps to create a dockerfile –

i. Create a Dockerfile.
ii. Add instructions in Dockerfile to create a Docker image.
iii. Run Dockerfile to create docker image
iv. Run Docker image to create docker container
v. Access the application running in docker container

5. Dockerfile

Various Commands-

1. FROM – used to define the base image on which we will be building.

Syntax – FROM base image

2. ADD- copies files from a source on the host into container's own filesystem at the set destination.

Syntax – ADD <source> <destination in container>

3. RUN – used to add layers to the base image by installing components.

Syntax – RUN command

5. Dockerfile

Various Commands-

4. CMD – used to run commands on the start of the container. These commands run only when there is no argument specified while running the container

Syntax – CMD [“executable”]

5. ENTRYPOINT- is used strictly to run commands the moment the container initializes. It run irrespective of whether argument is specified or not

Syntax – ENTRYPOINT [“executable”]

6. ENV- used to define environment variable in the container runtime

Syntax – ENV key=value

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install apache2
ADD . /var/www/html
#CMD apachectl -D FOREGROUND
ENTRYPOINT apachectl -D FOREGROUND
ENV name DevOps

docker tag mydockerfile namratasdocker/mydockerfile
docker login
docker push namratasdocker/mydockerfile
```

Modul e 2

Docker is a type of containerization platform that packages your application and all its dependencies together. This ensures that your application works seamlessly in any environment. Docker is a popular Platform As A Service (PAAS) product.

Docker is platform that helps developers build, share and run applications with container.

Docker Commands

docker - -version → gives the version of the docker

docker pull → to pull or download the image to form docker repo.

docker run → to create container from the image

docker ps → to see list of containers created by the user

docker ps -a → to see list of containers currently running

docker exec → to access the running containers

docker stop → to stop the running containers

docker kill → to kill the running containers

docker commit → to create new image of an edited container on the local system

docker login → to login docker hub repository

docker push → to push the image to docker hub repository

docker images → to list all the images which are created

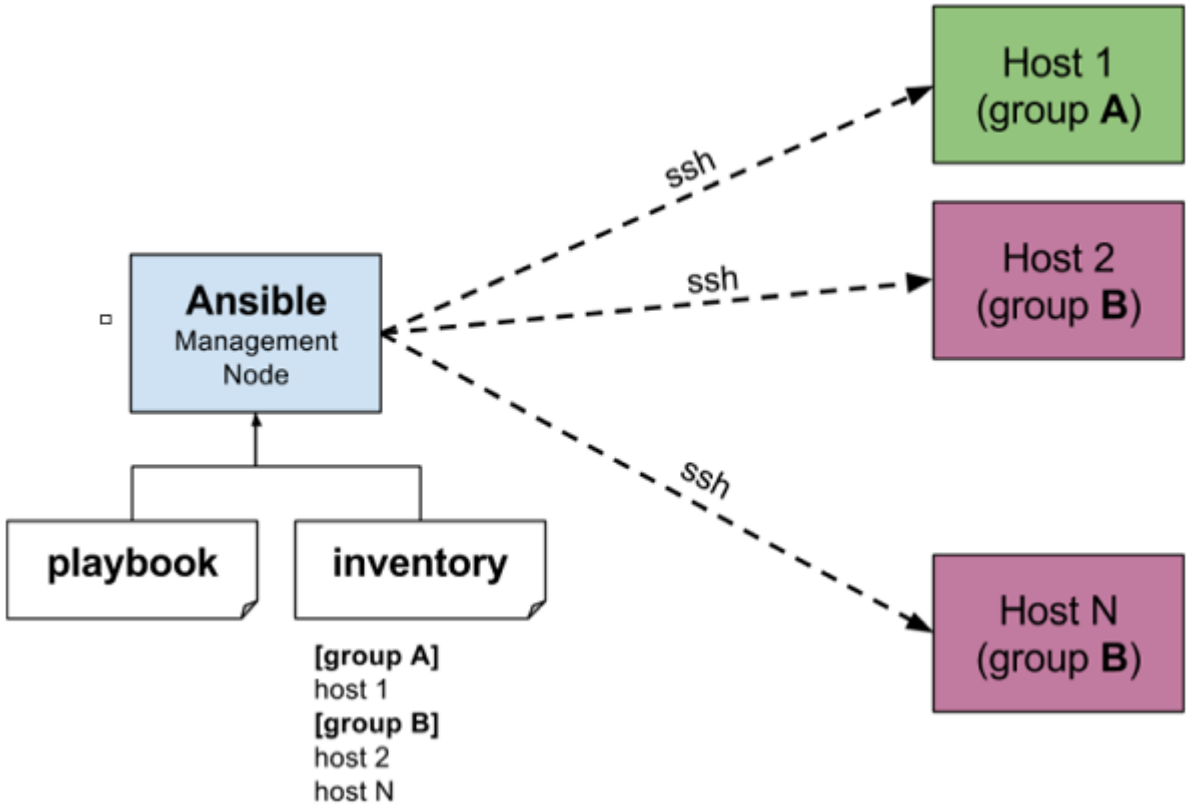
docker rm → used to remove one or more docker containers

docker rmi →used to remove one or more docker images

docker build → used to build the image from the dockerfile.

ANSIBLE

Ursula LeGuin



Steps to Install ANSIBLE in Ubuntu:

Step 1: Update System Packages

```
sudo apt update
sudo apt upgrade –y
```

Step 2: Install Ansible

```
sudo apt install ansible –y
```

Step 3: Configure Ansible Hosts File

```
sudo nano /etc/ansible/hosts
```

```
[group_name]
server1 ansible_host=ip_address_or_hostname
server2 ansible_host=ip_address_or_hostname
```

Step 4: Testing Ansible

```
ansible -m ping all
```

Step 5: Creating a Playbook

```
nano ~/playbook.yaml
```

```
---
- name: Install Apache
  hosts: group_name
  become: true
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
```

Step 6: Running Playbook

```
ansible-playbook ~/playbook.yaml
```

YAML Basics

Ansible uses YAML syntax for expressing Ansible playbooks. This chapter provides an overview of YAML. Ansible uses YAML because it is very easy for humans to understand, read and write when compared to other data formats like XML and JSON.

Every YAML file optionally starts with "**---**" and ends with "**...**"

Understanding YAML
In this section, we will learn the different ways in which the YAML data is represented.

key-value pair
YAML uses simple key-value pair to represent the data. The dictionary is represented in key: value pair.

Note − There should be space between : and value.

Example: A student record

```
--- #Optional YAML start syntax
james:
   name: james john
   rollNo: 34
   div: B
   sex: male
```

Abbreviation
You can also use abbreviation to represent dictionaries.

Example

James: {name: james john, rollNo: 34, div: B, sex: male}
Representing List
We can also represent List in YAML. Every element(member) of list should be written in a new line with same indentation starting with “- “ (- and space).

Example

```
---
countries:
   - America
   - China
   - Canada
   - Iceland
...
```

Abbreviation
You can also use abbreviation to represent lists.

Example

Countries: [‘America’, ‘China’, ‘Canada’, ‘Iceland’]

List inside Dictionaries

We can use list inside dictionaries, i.e., value of key is list.

Example

```
---
james:
   name: james john
   rollNo: 34
   div: B
   sex: male
   likes:
      - maths
      - physics
      - english
...
```

List of Dictionaries

We can also make list of dictionaries.

Example

```
---
- james:
   name: james john
   rollNo: 34
      div: B
   sex: male
   likes:
      - maths
      - physics
      - english

- robert:
      name: robert richardson
      rollNo: 53
      div: B
      sex: male
   likes:
      - biology
      - chemistry
...
```

YAML uses “|” to include newlines while showing multiple lines and “>” to suppress newlines while showing multiple lines. Due to this we can read and edit large lines. In both the cases indentation will be ignored.

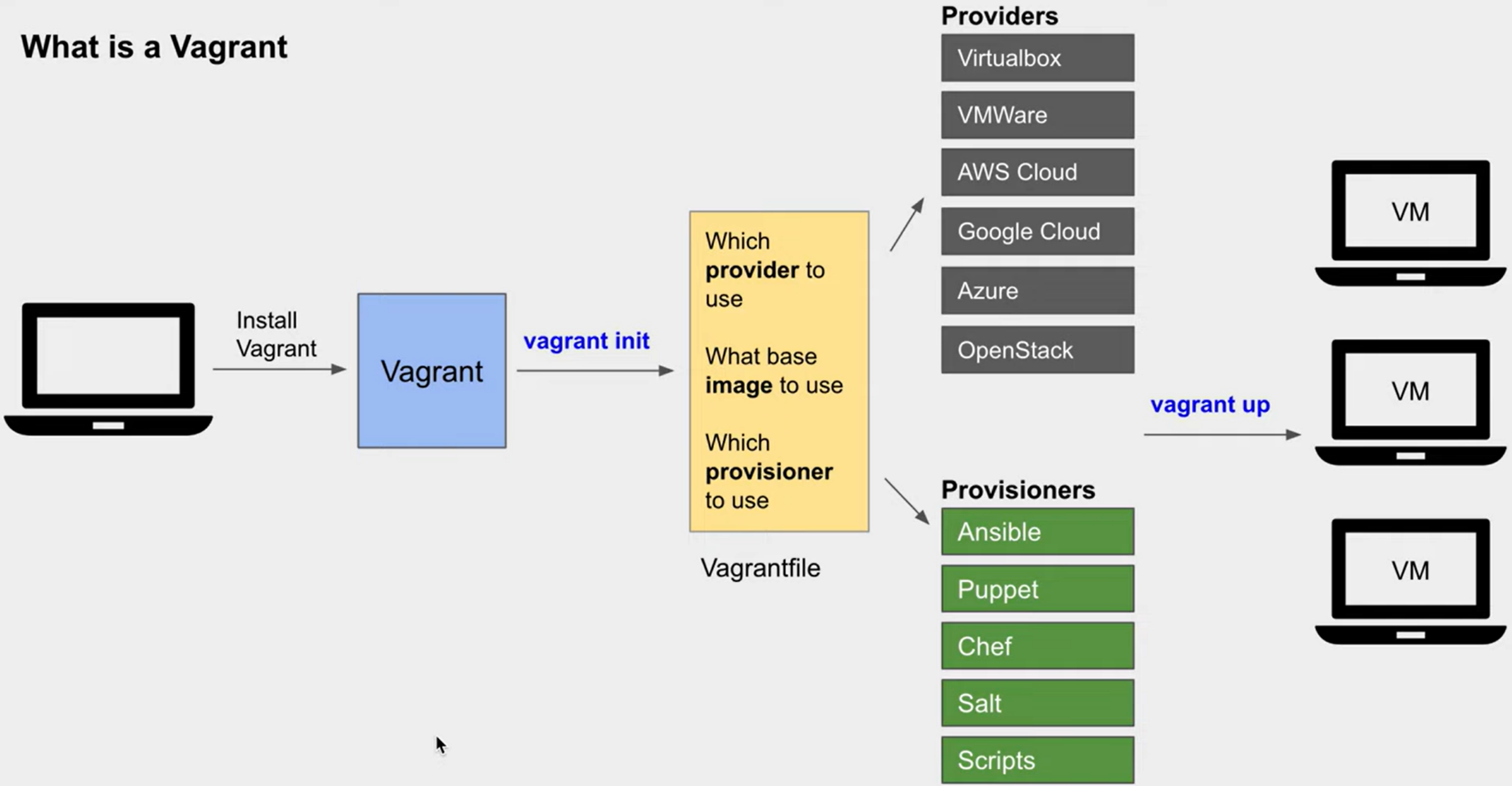
We can also represent Boolean (True/false) values in YAML. where boolean values can be case insensitive.

Example

```
---
- james:
   name: james john
   rollNo: 34
   div: B
   sex: male
   likes:
      - maths
      - physics
      - english

   result:
      maths: 87
      chemistry: 45
      biology: 56
      physics: 70
      english: 80
```

```
passed: TRUE

  messageIncludeNewLines: |
    Congratulation!!
    You passed with 79%

  messageExcludeNewLines: >
    Congratulation!!
    You passed with 79%
```

What is a Vagrant



What is a Vagrant

- open-source tool written in **Ruby**
- helps us to automate the creation and management of **Virtual Machines**
- we can specify the configuration of a virtual machine in a simple **configuration file**
- And then create the VM with using simple vagrant command **vagrant up**
- Can control VMs from command line using Vagrant commands

- Vagrant was first started as a personal side-project by **Mitchell Hashimoto** in January 2010
- The first version of Vagrant was released in **March 2010**
- Mitchell formed an organization **HashiCorp** to support the full-time development of Vagrant

Vagrant Commands

Command	Usage	Examples
vagrant init	Initializes a new Vagrant environment by creating a Vagrantfile	vagrant init centos/7
vagrant up	Creates and configures the guest machine	vagrant up
vagrant ssh	Logs in to the guest machine via SSH	vagrant ssh
vagrant ssh-config	Outputs OpenSSH valid configuration to connect to the VMs via SSH	vagrant ssh-config
vagrant halt	Stops the guest machine	vagrant halt
vagrant suspend	Suspends the guest machine	vagrant suspend
vagrant resume	Resumes a suspended guest machine	vagrant resume
vagrant reload	Reloads the guest machine by restarting it	vagrant reload
vagrant destroy	Stops and deletes all traces of the guest machine	vagrant destroy
vagrant status	Shows the status of the current Vagrant environment	vagrant status
vagrant package	Packages a running virtual environment into a reusable box	vagrant package --output mybox.box
vagrant provision	Runs any configured provisioners against the running VM.	vagrant provision
vagrant plugin install	Installs a Vagrant plugin	vagrant plugin install myplugin
vagrant plugin list	Lists all installed Vagrant plugins	vagrant plugin list
vagrant plugin uninstall	Uninstalls a Vagrant plugin	vagrant plugin uninstall myplugin

Vagrant Box - 7 Commands

Command	Use	Example
vagrant box add	Adds a box to your local box repository	vagrant box add ubuntu/focal64
vagrant box list	Lists all boxes in your local box repository	vagrant box list
vagrant box outdated	Checks if any boxes in your local box repository are outdated	vagrant box outdated
vagrant box update	Updates a box to a new version	vagrant box update ubuntu/focal64
vagrant box repackage	Repackages a box with a new name and metadata	vagrant box repackage ubuntu/focal64 --name my-new-box
vagrant box prune	Removes outdated boxes from your local box repository	vagrant box prune
vagrant box remove	Removes a box from your local box repository	vagrant box remove ubuntu/focal64

Mac OS X and Linux: ~/.vagrant.d/boxes

Windows: C:/Users/USERNAME/.vagrant.d/boxes

Useful TIPS

--help To get help for any Vagrant command e.g. **vagrant --help** or **vagrant init --help**

vboxmanage list vms If using Virtualbox

vboxmanage list runningvms If using Virtualbox

Ansible Controller Machine Setup

Step 1 - Install **VirtualBox** and **Vagrant** on your local machine.

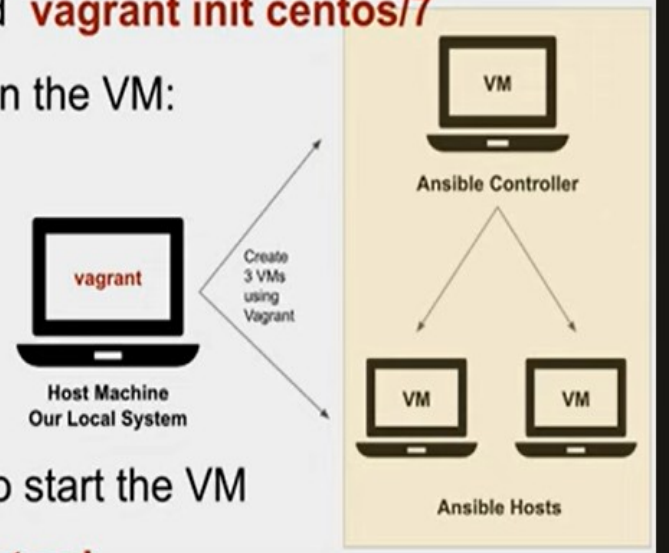
Step 2 - Open a terminal and navigate to the directory where you want to set up your Ansible project.

Step 3 - Create a new directory for your Ansible controller VM by running the command **mkdir ansible-controller**

Step 4 - Navigate to the directory and create a new file called Vagrantfile by running the command **vagrant init centos/7**

Step 5 - Edit the Vagrantfile and add the following lines to the end of the file to provision Ansible on the VM:

```
config.vm.provision "shell", inline: <<-SHELL
   sudo yum install epel-release -y
   sudo yum install ansible -y
 SHELL
```



Step 6 - Save & check its a valid vagrantfile **vagrant validate** Then run command **vagrant up** to start the VM

Step 7 - Once the VM is up and running, connect to it using SSH by running the command **vagrant ssh**

Check ansible is installed - **ansible --version**

Step 8 - Create a new directory for your Ansible project on the controller VM by running the command **mkdir ansible-project**

Step 9 - Navigate to the ansible-project directory and create a new file called hosts by running the command **touch hosts**

Step 10 - Create a new file called **playbook.yml**. This file will contain the tasks you want to perform on your managed hosts

As of now the hosts and the playbook file are empty

VagrantFile To Create Ansible Controller Virtual Machine:

```
Vagrant.configure("2") do |config|
  config.vm.box = "centos/7"
  config.vm.network "private_network", ip: "192.168.1.2"

  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"
    vb.cpus = 1
  end

  config.vm.provision "shell", inline: <<-SHELL
    sudo yum install epel-release -y
    sudo yum install ansible -y
  SHELL
end
```

Ansible Host Machines Setup

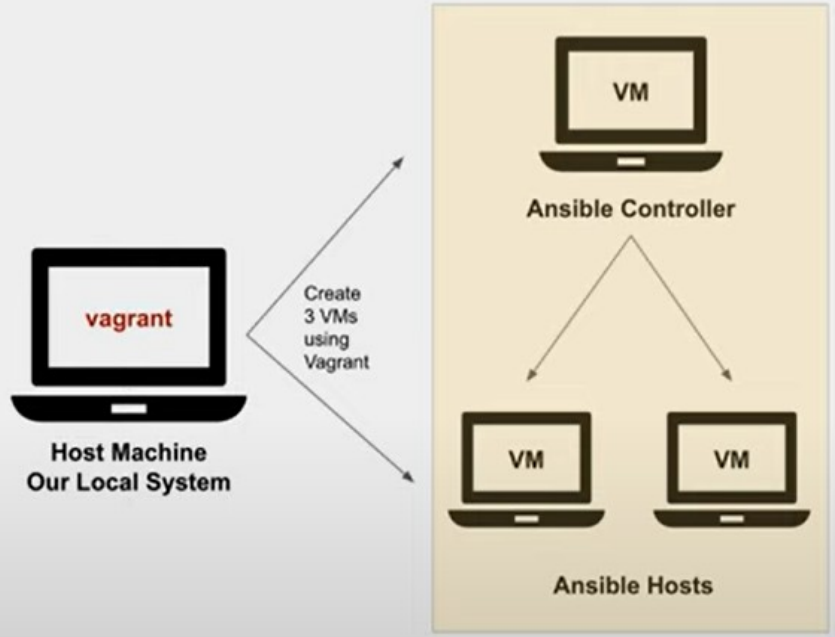
Step 1 - On terminal navigate to your Ansible Project folder

Step 2 - Create a new directory for your host machines by running the command **mkdir host-machines**

Step 3 - Navigate to host-machines directory and create a new **Vagrantfile** by running the command **vagrant init centos/7**

Step 4 - Edit the Vagrantfile and modify the following lines to set up two Vagrant machines:

```
config.vm.define "web" do |web|
  web.vm.box = "centos/7"
  web.vm.hostname = "web"
  web.vm.network "private_network", ip: "192.168.33.10"
end

config.vm.define "db" do |db|
  db.vm.box = "centos/7"
  db.vm.hostname = "db"
  db.vm.network "private_network", ip: "192.168.33.11"
end

config.vm.network "forwarded_port", guest: 80, host: 8080
```



Step 6 - Save & check its a valid vagrantfile **vagrant validate** Then run command **vagrant up** to start the VM

Step 7 - Check the status of machines **vagrant status**

Once the VMs are up, connect to them using SSH **vagrant ssh <machine-name>** e.g **vagrant ssh web**

This completes the process of setting up host machines

VagrantFile to generate Two Host Machines db and web:

```
Vagrant.configure("2") do |config|
  config.vm.box = "centos/7"

  config.vm.define "web" do |web|
    web.vm.hostname = "web"
    web.vm.network "private_network", ip: "192.168.33.10"
  end

  config.vm.define "db" do |db|
    db.vm.hostname = "db"
    db.vm.network "private_network", ip: "192.168.33.11"
  end
  config.vm.network "forwarded_port", guest: 80, host: 8080, auto_correct: true
  config.vm.usable_port_range = (8000..9000)
end
```

Making connection between controller and host machines

Step 1 - Make sure all machines are up and running

Step 2 - Run command **ip addr** on each machine and check they have IP addresses in the same range (e.g. **192.168.33.x**).

Step 3 - On Controller machine run the command **ssh-keygen** to generate an SSH key pair

Step 4 - Goto **~/.ssh** folder and check the public and private keys generated

Step 5 - Copy the public key to the host machines by running the command **ssh-copy-id <user>@<host>**

For example, to copy the public key to the web machine, run the command **ssh-copy-id vagrant@192.168.33.10**

Can do this manually by copying the contents of the .pub file generated by ssh-keygen and pasting it into the ~/.ssh/authorized_keys file on the host machines

Step 6 - Test the SSH connection by running the ssh command with the IP address of the host machines-

For example: **ssh vagrant@192.168.33.10**

ssh vagrant@192.168.33.11

Adding host and playbook file on Controller and Run Playbook

Step 1 - Connect to the ansible controller machine using ssh **vagrant ssh <machine name>**

Step 2 - Edit the **hosts** file to add the IP addresses or hostnames of the machines you want to control. For example:

```
[webservers]
192.168.33.10

[dbservers]
192.168.33.11
```

Step 3 - Run the command **ansible all -m ping -i hosts** to test the connection to the host machines

Step 4 - Now edit the playbook.yml file and add instructions for host machines

Step 5 - You can now run the playbook on the managed hosts by running the command **ansible-playbook -i hosts playbook.yml**

Step 6 - Once the playbook has run, you can access the web servers by opening a web browser and navigating to the IP addresses of your webservers

Palybook.yml file Code:

```
---
- name: Install Apache web server
  hosts: webservers
  become: true
  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: latest
    - name: Start Apache
      service:
        name: httpd
        state: started
        enabled: true
```

Ansible Tower

Ansible Tower is like Ansible at a more enterprise level. It is a web-based solution for managing your organization with an easy user interface that provides a dashboard with all of the state summaries of all the hosts. And allows quick deployments, and monitors all configurations.

The tower allows us to share the SSH credentials without exposing them, logs all the jobs, manage inventories graphically, and syncs them with a wide variety of cloud providers.

Previously, Ansible Tower called the AWX project, is the fix to this problem.

Prerequisites to Install Ansible Tower

There is the following prerequisite to install the Ansible Tower, such as:

- The following operating systems support Ansible Tower
    1. RedHat Enterprise Linux 6 64-bit
    2. RedHat Enterprise Linux 7 64-bit
    3. CentOS 6 64-bit
    4. CentOS 7 64-bit
    5. Ubuntu 12.04 LTS 64-bit
    6. Ubuntu 14.04 LTS 64-bit
    7. Ubuntu 16.04 LTS 64 bit
- You should have the latest stable release of Ansible.
- It required a 64-bit support kernel, runtime, and 20 GB hard disk.
- Minimum 2 GB RAM (4 GB RAM recommended) is required.
    1. Minimum 2 GB RAM is recommended for Vagrant trial installations
    2. And 4 GB RAM is recommended /100 forks

Ansible Tower Features

Here are some features of the Ansible Tower, such as

1. Ansible Tower Dashboard: It displays everything which is going on in your Ansible environment, such as the inventory status, the recent job activity, the hosts, and so on.

2. Multi-Playbook Workflows: It allows to chain any numbers of playbooks, any way of the usage of different inventories, runs different users, or utilizes various credentials.

3. Real-Time Job Updates: Ansible can automate the complete infrastructure. Also, you can see real-time job updates such as plays and tasks broken down by each machine either been successful or failure. Therefore, you can see the status of your automation and know what's next in the queue.

4. Scale Capacity with Cluster: You can connect multiple Ansible Tower nodes into an Ansible Tower cluster as the clusters add redundancy and capacity, which allows scaling Ansible automation across the enterprise.

5. Self-Service: You can launch playbooks with just a single click through this feature.

6. Remote Command Execution: With this command, you can run simple tasks such as restart any malfunctioning service, add users, reset passwords on any host or group of hosts in the inventory.

7. Manage and Track Inventory: It manages your entire infrastructure by pulling inventory from public cloud providers such as Microsoft Azure, amazon web services, etc.

8. Integrated Notification: This notifies you when a job succeeds or fails across the entire organization at once, or customize on a pre-job basis.

9. Schedule Ansible Jobs: It schedule different kinds of jobs such as playbook runs, cloud inventory updates, and source control updates to run according to the need.

10. REST API and Tower CLI Tool: Every feature present in Ansible Tower is available through the Ansible Tower's REST API, which provides the ideal API for the systems management infrastructure. The Ansible Tower's CLI tool is available for launching jobs from CI systems such as Jenkins, or when you need to integrate with other command-line tools.

Ansible Pip

Ansible pip module is used when you need to manage python libraries on the remote servers.

There are two prerequisites if you need to use all the features in the pip module.

- The pip package should already be installed on the remote server.
- Virtualenv package should be installed on the remote server already if you need to manage the packages in the python virtual environment.

NOTE: If you get the error "unable to find any of pip2, pip to use. Pip needs to be installed". The pip module is not available on the remote server during the execution.

Installing a Pip Module

To install a new python library, you need to set the name of the package against the "name" parameter. By default, the "state" parameter is "present", the module will try to install the library.

If the library is already installed, then nothing will be done. And if a new version of the library exists, it will not be upgraded.

```
- hosts: all
  tasks:
  - name: Installing NumPy python library using pip module
    pip:
      name: NumPy
```

Installing Using a Requirement File

Another way to install the libraries is via the requirements file. If you have any requirements file with all the libraries in the remote servers, give it as input to the "requirements" parameters.

Also, you can use the copy module beforehand to copy the requirements file to every remote server. In the following code, install the requirements file in the location/tmp/req.txt.

```
- hosts: all
  tasks:
  - name: Installing python libraries using requirements file
    pip:
      requirements: req.txt
      chdir: /tmp
```

```
req.txt
-------
nltk==3.0.0
numpy<2.0.0 scipy>=1.0.0
```

Installing Multiple Python Libraries

To install the multiple packages, set all the libraries against the "name" parameter, separated by a comma.

```
- hosts: all
  tasks:
  - name: Installing multiple python packages
    pip:
      name: NumPy,SciPy
```

Installing a Particular Version of Pip Library

There is a "version" parameter, which can be used to install only the mentioned version of a library. In the following code, install the version of nltk library.

```
- hosts: all
  tasks:
  - name: Installing a required version of python library
    pip:
      name: nltk
      version: '3.0.0'
```

Reinstall a Python Library

You can reinstall the python library by using the "forcereinstall" value for the "state" parameter.

This will reinstall the latest version of the library. You can use the "version" parameter along with it. The following code will install the version 3.0.0 of the nltk library.

```
- hosts: all
  tasks:
  - name: Reinstalling a python library
    pip:
      name: nltk
      version: 3.0.0
      state: forcereinstall
```

Removing a Python Library

You can delete a python library by changing the state to "absent". In the following code, we will remove the "NumPy" and "SciPy" python libraries from the remote servers.

```
- hosts: all
  tasks:
  - name: Removing Python libraries
    pip:
      name: NumPy,SciPy
      state: absent
```

Ansible vs Chef

Ansible and Chef both are the most popular configuration management tools. Both tools can accomplish many of the same tasks, they each have different strengths, and they perform their tasks in different ways.

This tutorial will explore the strengths and differences of these tools. Before move further, take a glance at Ansible and Chef.

Ansible

Ansible is an open-source IT engine that automates application deployment, cloud provisioning, intra service orchestration, and other IT tools.

Ansible is easy to deploy because it does not use any **agents** or **custom security** infrastructure on the client-side, and by pushing modules to the clients. These modules are executed locally on the client-side, and the output is pushed back to the Ansible server.

It can easily connect to clients using **SSH-Keys**, simplifying though the whole process. Client details, such as **hostnames** or **IP addresses** and **SSH port**s, are stored in the files, which are called inventory files. If you created an inventory file and populated it, then Ansible can use it.

Ansible uses the playbook to describe automation jobs, and playbook, which uses simple language, i.e., **YAML**. YAML is a human-readable data serialization language & commonly used for configuration files, but it can be used in many applications where data is being stored.

A significant advantage is that even the IT infrastructure support guys can read and understand the playbook and debug if needed.

Ansible is designed for multi-tier deployment. Ansible does not manage one system at a time, and it models IT infrastructure by describing all of your systems are interrelated. Ansible is entirely agentless, which means Ansible works by connecting your nodes through **SSH** (by default). Ansible gives the option to you if you want another method for the connection like **Kerberos**.

Chef

The chef is a powerful automation platform that transforms infrastructure into the code. Whether you are operating in the on-premises, cloud, or a hybrid environment.

Chef automates how the infrastructure is **deployed, configured,** and **managed** across your network. A chef is an open-source cloud configuration that translates system administration tasks into reusable definitions, otherwise known as recipes and cookbooks.

Chef runs on different platforms such as Windows, AIX, Enterprise Linux distributions, Solaris, FreeBSD, Cisco IO, and Nexus.

It also supports cloud platforms such as Amazon Web Services (AWS), Google Cloud Platform, OpenStack, IBM Bluemix, HPE Cloud, Microsoft Azure, VMware vRealize Automation, and Rackspace.

Below are some main differences between the Ansible and Chef:

Parameters	Ansible	Chef
Availability	Ansible runs with a single active node, called the Primary instance. If the primary goes down, there is a Secondary instance to take its place.	When there is a failure on the primary server, which is a chef server, it has a backup server to take the place of the primary server.
Easy to setup	Ansible has only a master running on the server machine, but no agents running on the client machine. It uses an SSH connection to log in to client systems or the nodes you want to configure. Client machine VM requires no unique setup. That's why it is faster to setup!	Chef has a master-agent architecture. Chef server runs on the master machine, and Chef client runs as an agent on each client machine. And also, there is an extra component called workstation, which contains all the tested configurations and then pushed to the central chef server. That's why it is not that easy.
Management	Easy to manage the configurations as it uses YAML (Yet Another Markup Language). The server pushes configurations to all the nodes. Suitable for real-time application, and there is immediate remote execution.	You need to be a programmer to manage the configurations as it offers configurations in Ruby DSL. The client pulls the configurations from the Server.
Configuration language	Ansible uses YAML (Python). It is quite easy to learn and its administrator oriented. Python is inbuilt into most Unix and Linux deployments, so setting the tool up and running is quicker.	Chef uses Ruby Domain Specific Language (Ruby DSL). It has a Steep Learning Curve and its developer-oriented.
Interoperability	The Ansible server has to be on Linux/Unix machine. As well as Ansible supports windows machines.	Chef Server works only on Linux/Unix, but Chef Client and Workstation can be on windows as well.

Pricing	The pricing for Ansible Tower for standard IT operations up to 100 nodes is $10,000 per year. This includes 8*5 support, whereas premium offers 24*7 support for $14000 per year.	Chef Automate gives you everything you need to build, deploy in $137 node per year.
Authoritative configuration	Ansible's authoritative configuration comes from its deployed playbooks, which are perfect as source control systems. Or the Ansible method is more accessible and makes more sense.	The chef relies on its server as the authoritative configuration, and those servers require uploaded cookbooks, which means making sure the latter are consistent and identical.

Ansible Vault

Ansible Vault is a feature which allows user to encrypt values and data structures within Ansible projects. This provides the ability to secure any secrets or sensitive data that is necessary to run Ansible plays successfully but should not be publicly visible, such as private keys or passwords. Ansible automatically decrypts the vault-encrypted content at runtime when the key is provided.

To integrate these secrets with regular Ansible data, both the Ansible and Ansible-playbook commands, for executing ad hoc tasks and structured playbook respectively, have support for decrypting vault-encrypted content at runtime.

Ansible Vault is implemented with file-level granularity; it means files are either entirely encrypted or unencrypted. It uses the AES256 algorithm to provide symmetric encryption keyed to a user-supplied password.

This means the same password is used to encrypt and decrypt the content, which is helpful from a usability standpoint. Ansible can identify and decrypt any vault-encrypted files it finds while executing a task or playbook.

Though there is a proposal to change this, at the time of writing this, users can only pass in a single password to Ansible. It means that each of the encrypted files involved must share a password.

Using Ansible Vault

The simple use of the Ansible vault is to encrypt variables files. It can encrypt any YAML file, but the most common files to encrypt are:

- A role's defaults/ main.yml file
- A role's vars/main.yml file

- Files within the group_vars directory
- Any other file used to store variables

Encrypting an Existing File

You can encrypt a regular plaintext variable file by using the ansible vault and define the password that needed later to decrypt it.

```
#encrypt a role's defaults/main.yml file
ansible-vault encrypt defaults/main.yml
>New vault password:
>Confirm new vault password:
>Encryption successful
```

The ansible-vault command will prompt you a password twice. After that, the file will be encrypted.

Creating an Encrypted File

To create an encrypted data file, use the ansible-vault to create command, and pass the filename.

```
$ansible-vault create <file name>
```

You will be prompted to create a password and then confirm it by re-typing it.

Once your password is confirmed, a new file will be created and will open an editing a window. By default, the editor for Ansible vault is VI. You can add data, save it, and exit from it.

Editing Encrypted Files

If you want to edit the encrypted file, you can edit it using ansible-vault edit command. This command will decrypt the file to a temporary file and allow you to edit the file.

```
$ansible-vault edit <file name>
```

You will be prompted to insert the vault password. The decrypted file will open in a VI editor, and then you can make the required changes. Save the changes and removing the temporary file.

Rekeying Encrypted Files

If you want to change your password on a vault on a vault-encrypted file, you can do it by using the rekey command.

1. $ansible-vault rekey <file1> <file2> <file3>

The above command can rekey multiple data files at once and ask for the original password and the new password.

Encrypting Unencrypted Files

If you have existing files which you want to encrypt, use the ansible-vault encrypt command. This command can operate on multiple files at once.

1. $ansible-vault encrypt <file1> <file2> <file3>

Decrypting Encrypted Files

If you have existing files that you no longer want to keep encrypted, you can decrypt them permanently by running the ansible-vault decrypt command. This command will save them unencrypted to the disk.

1. $ansible-vault decrypt <file1> <file2> <file3>

Viewing Encrypted Files

If you want to view the contents of an encrypted file without editing it, then you can use the ansible-vault view command.

1. $ansible-vault view <file1> <file2> <file3>

Jenkins

Jenkins is an open-source automation tool written in Java programming language that allows continuous integration.

What is Jenkins?

Jenkins **builds** and **tests** our software projects which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

It also allows us to continuously **deliver** our software by integrating with a large number of testing and deployment technologies.

Jenkins offers a straightforward way to set up a continuous integration or continuous delivery environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks.

With the help of Jenkins, organizations can speed up the software development process through automation. Jenkins adds development life-cycle processes of all kinds, including build, document, test, package, stage, deploy static analysis and much more.

Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

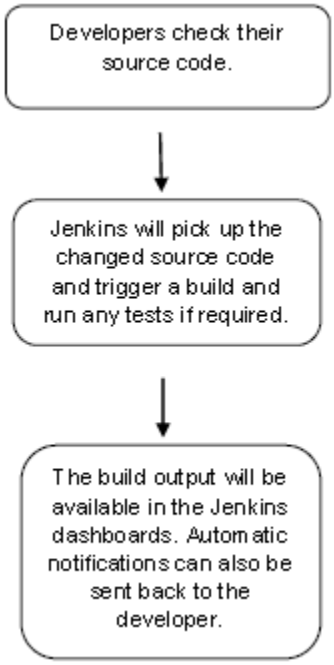
For example: If any organization is developing a project, then **Jenkins** will continuously test your project builds and show you the errors in early stages of your development.

Possible steps executed by Jenkins are for example:

- Perform a software build using a build system like Gradle or Maven Apache
- Execute a shell script
- Archive a build result

- Running software tests

Work Flow:



History of Jenkins

Kohsuke Kawaguchi, who is a Java developer, working at SUN Microsystems, was tired of building the code and fixing errors repetitively. In 2004, he created an automation server called **Hudson** that automates build and test task.

In 2011, Oracle who owned Sun Microsystems had a dispute with Hudson open source community, so they forked Hudson and renamed it as **Jenkins**.

Both Hudson and Jenkins continued to operate independently. But in short span of time, Jenkins acquired a lot of contributors and projects while Hudson remained with only 32 projects. Then with time, Jenkins became more popular, and Hudson is not maintained anymore.

What is Continuous Integration?

Continuous Integration *(CI)* is a development practice in which the developers are needs to commit changes to the source code in a shared repository at regular intervals. Every commit made in the repository is then built. This allows the development teams to detect the problems early.

Continuous integration requires the developers to have regular builds. The general practice is that whenever a code commit occurs, a build should be triggered.

Continuous Integration with Jenkins

Let's consider a scenario where the complete source code of the application was built and then deployed on test server for testing. It sounds like a perfect way to *develop software*, but this process has many problems.

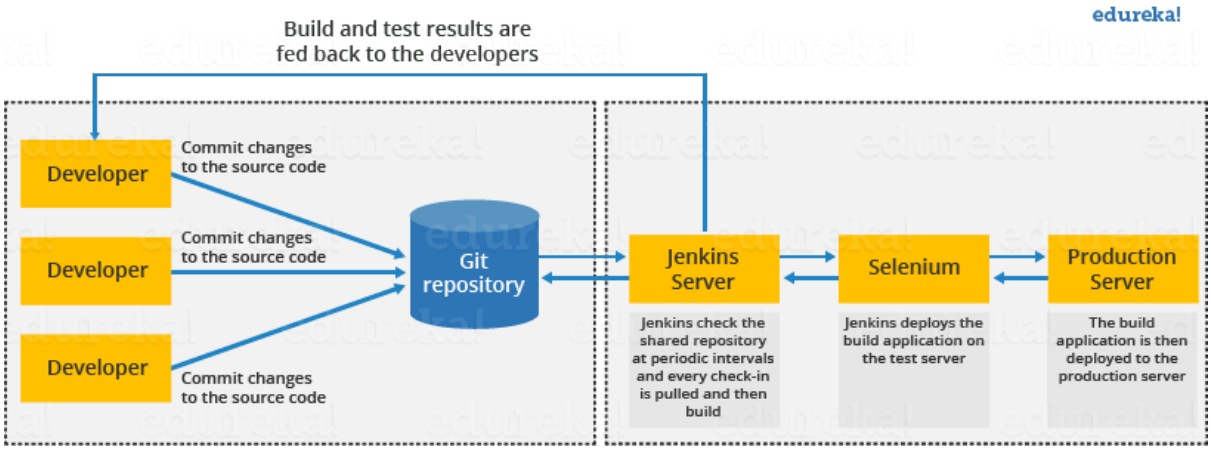
- Developer teams have to wait till the complete software is developed for the test results.
- There is a high prospect that the test results might show multiple bugs. It was tough for developers to locate those bugs because they have to check the entire source code of the application.
- It slows the software delivery process.
- Continuous feedback pertaining to things like architectural or coding issues, build failures, test status and file release uploads was missing due to which the quality of software can go down.
- The whole process was manual which increases the threat of frequent failure.

It is obvious from the above stated problems that not only the software delivery process became slow but the quality of software also went down. This leads to customer dissatisfaction.

So to overcome such problem there was a need for a system to exist where developers can continuously trigger a build and test for every change made in the source code.

This is what Continuous Integration (CI) is all about. Jenkins is the most mature Continuous Integration tool available so let us see how Continuous Integration with Jenkins overcame the above shortcomings.

Let's see a generic flow diagram of Continuous Integration with Jenkins:



Let's see how Jenkins works. The above diagram is representing the following functions:

- First of all, a developer commits the code to the source code repository. Meanwhile, the Jenkins checks the repository at regular intervals for changes.

- Soon after a commit occurs, the Jenkins server finds the changes that have occurred in the source code repository. Jenkins will draw those changes and will start preparing a new build.
- If the build fails, then the concerned team will be notified.
- If built is successful, then Jenkins server deploys the built in the test server.
- After testing, Jenkins server generates a feedback and then notifies the developers about the build and test results.
- It will continue to verify the source code repository for changes made in the source code and the whole process keeps on repeating.

Advantages of Jenkins

- It is an open source tool.
- It is free of cost.
- It does not require additional installations or components. Means it is easy to install.
- Easily configurable.
- It supports 1000 or more plugins to ease your work. If a plugin does not exist, you can write the script for it and share with community.
- It is built in java and hence it is portable.
- It is platform independent. It is available for all platforms and different operating systems. Like OS X, Windows or Linux.
- Easy support, since it open source and widely used.
- Jenkins also supports cloud based architecture so that we can deploy Jenkins in cloud based platforms.

Disadvantages of Jenkins

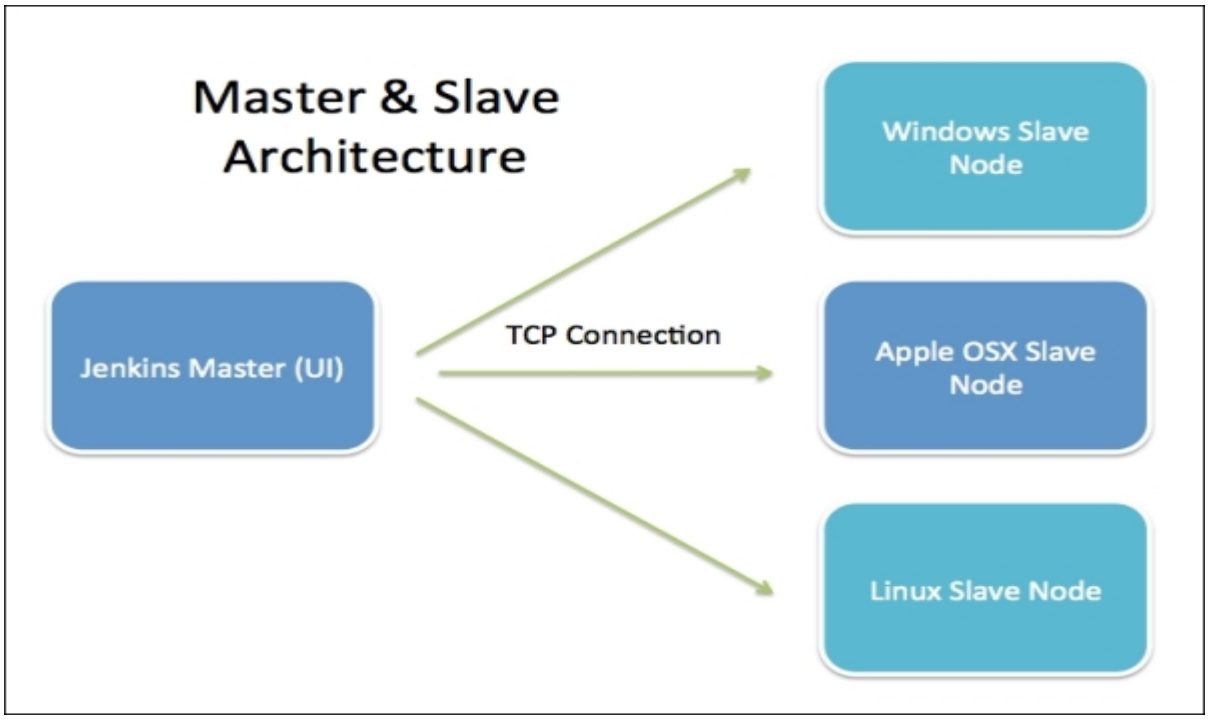
- Its interface is out dated and not user friendly compared to current user interface trends.
- Not easy to maintain it because it runs on a server and requires some skills as server administrator to monitor its activity.
- CI regularly breaks due to some small setting changes. CI will be paused and therefore requires some developer's team attention.

Jenkins Architecture

Jenkins follows Master-Slave architecture to manage distributed builds. In this architecture, slave and master communicate through TCP/IP protocol.

Jenkins architecture has two components:

- Jenkins Master/Server
- Jenkins Slave/Node/Build Server



Jenkins Master

The main server of Jenkins is the Jenkins Master. It is a web dashboard which is nothing but powered from a war file. By default it runs on 8080 port. With the help of Dashboard, we can configure the jobs/projects but the build takes place in Nodes/Slave. By default one node (slave) is configured and running in Jenkins server. We can add more nodes using IP address, user name and password using the ssh, jnlp or webstart methods.

The server's job or master's job is to handle:

- Scheduling build jobs.
- Dispatching builds to the nodes/slaves for the actual execution.
- Monitor the nodes/slaves (possibly taking them online and offline as required).
- Recording and presenting the build results.
- A Master/Server instance of Jenkins can also execute build jobs directly.

Jenkins Slave

Jenkins slave is used to execute the build jobs dispatched by the master. We can configure a project to always run on a particular slave machine, or particular type of slave machine, or simple let the Jenkins to pick the next available slave/node.

As we know Jenkins is developed using Java is platform independent thus Jenkins Master/Servers and Slave/nodes can be configured in any servers including Linux, Windows, and Mac.