ABHISHEK K L
Assist.Prof, MCA-RIT

MODULE-IV

Continuous Integration with Jenkins

Continuous Integration (CI) is a DevOps software development practice that enables the developers to merge their code changes in the central repository to run automated builds and tests. It refers to the process of automating the integration of code changes coming from several sources.

Features of Continuous Integration

Following are some of the main **features or practices** for Continuous Integration.

1. **Maintain a single source repository** – All source code is maintained in a single repository. This avoids having source code being scattered across multiple locations. Tools such as Subversion and Git are the most popular tools for maintaining source code.
2. **Automate the build** – The build of the software should be carried out in such a way that it can be automated. If there are multiple steps that need to be carried out, then the build tool needs to be capable of doing this. For .Net, MSBuild is the default build tool and for Java based applications you have tools such as Maven and Grunt.
3. **Make your build self-testing** – The build should be testable. Directly after the build occurs, test cases should be run to ensure that testing can be carried out for the various functionality of the software.
4. **Every commit should build on an integration machine** – The integration machine is the build server and it should be ensured that the build runs on this machine. This means that all dependent components should exist on the Continuous Integration server.
5. **Keep the build fast –** The build should happen in minutes. The build should not take hours to happen, because this would mean the build steps are not properly configured.
6. **Everyone can see what is happening –** The entire process of build and testing and deployment should be visible to all.
7. **Automate deployment –** Continuous Integration leads to Continuous deployment. It is absolutely necessary to ensure that the build should be easy to deploy onto the production environment.

What Does Build Mean?

The term build may refer to the process by which source code is converted into a stand-alone form that can be run on a computer or to the form itself. One of the most important steps of a software build is the compilation process, where source code files are converted into executable code. The process of building software is usually managed by a build tool. Builds are created when a certain point in development has been reached or the code has been deemed ready for implementation, either for testing or outright release.

A build is also known as a software build or code build.

Build automation is the process of automating the retrieval of source code, compiling it into binary code, executing automated tests, and publishing it into a shared, centralized repository.

Need /Importance of Continuous Integration

1. Reduces Risk

The frequent testing and deployment of code reduce the project's risk level, as now the code defects and bugs can be detected earlier. This states that these bugs and errors can be easily fixed and take less time, making the overall process cheaper. The general working speeds up the feedback mechanism that makes the communication smoother and effective.

2. Better Communication

The Continuous Integration process collaborates with the Continuous Delivery workflow that makes code sharing easy and regularized. This makes the process more transparent and collaborative among team members. In the long term, this makes the communication speed more efficient and makes sure that everyone in the organization is on the same page.

3. Higher Product Quality

Continuous Integration provides features like Code review and Code quality detection, making the identification of errors easy. If the code does not match the standard level or a mistake, it will be alerted with emails or SMS messages. Code review helps the developers to improve their programming skills continually.

4. Reduced Waiting Time

The time between the application development, integration, testing, and deployment is considerably reduced. When this time is reduced, it, in turn, reduces the waiting time that may occur in the middle. CI makes sure that all these processes continue to happen no matter what.

What is Jenkins

Jenkins is an open-source automation tool written in Java with plugins built for Continuous Integration purposes and used to build and test software projects 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 to continuously deliver software by integrating with a large number of testing and deployment technologies.

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.

ABHISHEK K L
Assist.Prof, MCA-RIT

Jenkins Features

Jenkins offers many attractive features for developers:

1. **Easy Installation**
   Jenkins is a platform-agnostic, self-contained Java-based program, ready to run with packages for Windows, Mac OS, and Unix-like operating systems.

2. **Easy Configuration**
   Jenkins is easily set up and configured using its web interface, featuring error checks and a built-in help function.

3. **Available Plugins**
   There are hundreds of plugins available in the Update Center, integrating with every tool in the CI and CD toolchain.

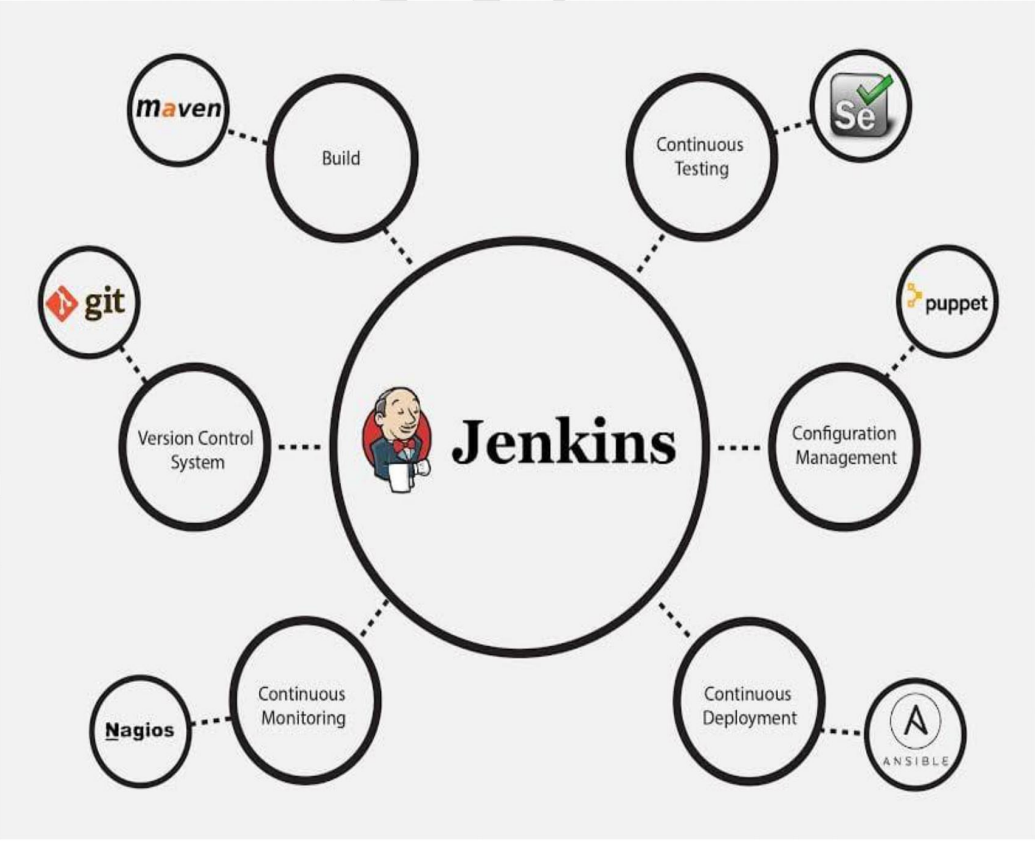
4. **Extensible**
   Jenkins can be extended by means of its plugin architecture, providing nearly endless possibilities for what it can do.

5. **Easy Distribution**
   Jenkins can easily distribute work across multiple machines for faster builds, tests, and deployments across multiple platforms.

6. **Free Open Source**
   Jenkins is an open-source resource backed by heavy community support.

why we use it?

The image below depicts that Jenkins is integrating various DevOps stages:



ABHISHEK K L
Assist.Prof, MCA-RIT

Advantages of Jenkins

1. **Open Source and Free:** Developers don't need to take tension about the money; it is free of cost. It is platform-independent.

2. **Plug-ins and Integration:** It is one of the most important features that make it most widely used. It has its type of plug-in, which helps the developer a lot in executing the jobs. Jenkins plug-ins can be developed by anyone and for anyone. Dashboard view plug-in, test analysis plug-in, build pipeline plug-in, and many more like this makes the developer familiar with the Jenkins tool.

3. **Hosting Option:** It is yet another important feature of the Jenkins, which can be installed on any operating system like Windows, MacOS, Linux, etc. You can also run Jenkins on the cloud by downloading and deploying it on a VM. You can also use a Docker container in it.

4. **Community Support:** Jenkins has great support from the developer community. You can assume its popularity and community support that it has more than 1000000 users all over the world, while it was officially published in 2011.

5. **Integration with other CI/CD platforms:** Jenkins supports many CI/CD platforms, not only the pipeline. It can make interaction with other tools also. Several plug-ins are available in it, which allows users to make connections with other CI/CD platforms.

6. **Keep your team in sync:** Jenkins focuses on a centralized way of working. All the members of the team move in sync.
7. **Easy to debug:** It is very easy to find out the errors in the Jenkins. The developer can easily check the bug and resolve it.

8. **Less time to deliver the project:** It happens because of its continuous integration feature.

9. **Flexible in creating the jobs:** It is very flexible in creating the jobs. It can create jobs both in freestyle and in the pipeline process very easily.

10. **Source Code Management (SCM):** Jenkins supports different types of source code repositories like SVN, Git, etc. The developer can set different trigger after making changes in the codes. He can do it every time.

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.
- All plug-ins are not compatible with the declarative pipeline syntax.
- Jenkins has many plug-ins in its library, but it seems like they are not maintained by the developer team from time to time. This is when it becomes very important that whatever plug-ins you are going to use; are getting a regular update or not.
- Lots of plug-ins have a problem with the updating process.

- It is dependent on plug-ins; sometimes, you can't find even basic things without plug-ins.

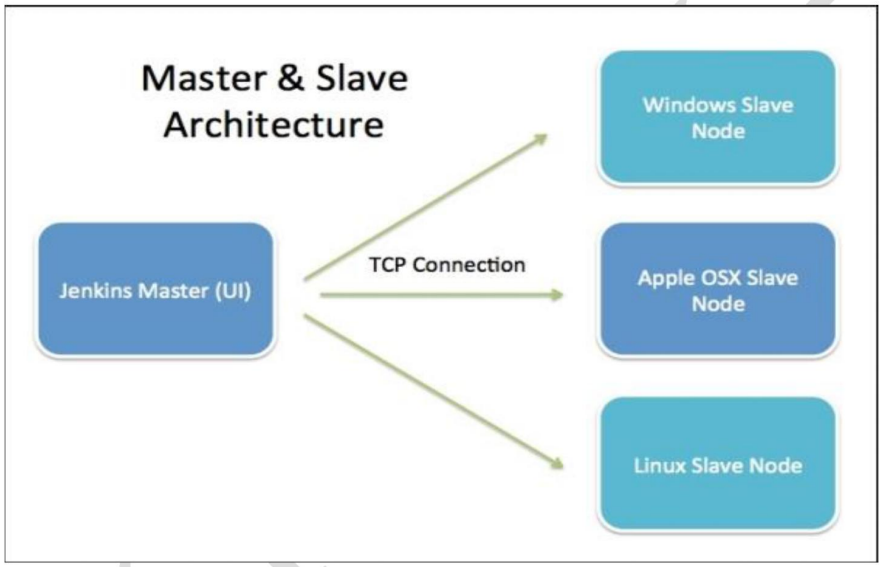
Jenkins Architecture

Standalone Jenkins instances can be an intensive disk and CPU Resource-Consuming process. To avoid this Jenkins follows Master-Slave architecture to manage distributed builds by implementing slave nodes which essentially would help to offload a part of the master node's responsibilities.

In this architecture, slave and master communicate through TCP/IP protocol. Jenkins architecture has two components:

1. **Jenkins Master/Server**
2. **Jenkins Slave/Node/Build Server**

In the below image, the Jenkins master is in charge of the UI and the slave nodes are of different OS types.



Jenkins Master

The master is the base installation of the Jenkins tool and does the basic operations and serves the user interface while the slaves do the actual work.

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.

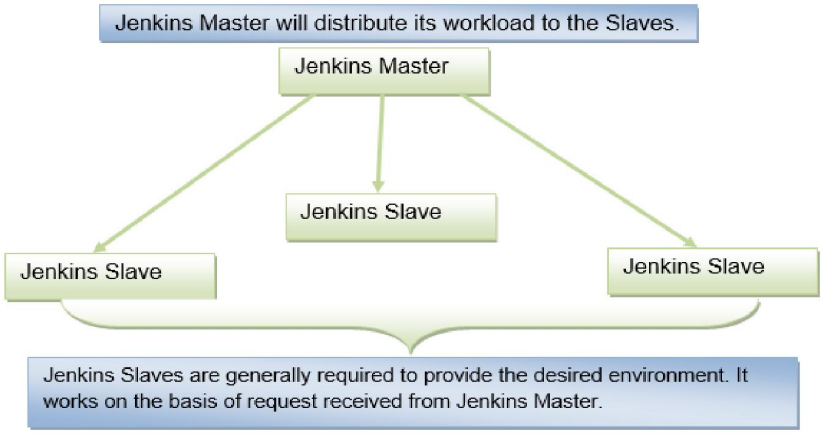
The server's job or master's job is to handle:

- 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

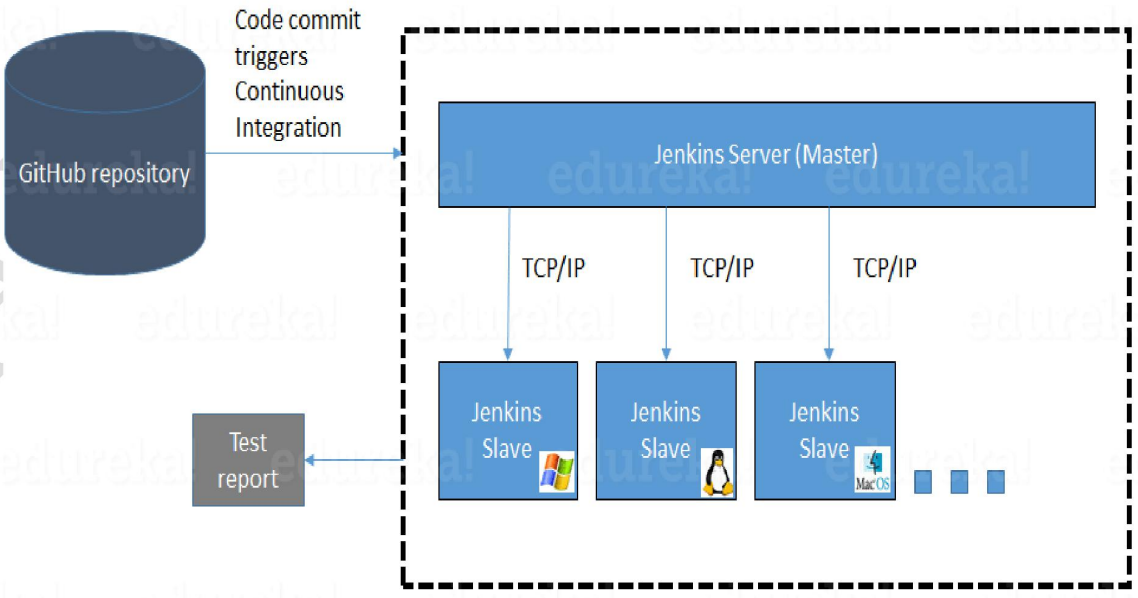
A slave is just a device that is configured to act as an executor on behalf of the master. A Slave is a Java executable that runs on a remote machine. Following are the characteristics of Jenkins Slaves:

- It hears requests from the Jenkins Master instance.
- Slaves can run on a variety of operating systems.
- The job of a Slave is to do as they are told to, which involves executing build jobs dispatched by the Master.
- You can configure a project to always run on a particular Slave machine, or a particular type of Slave machine, or simply let Jenkins pick the next available Slave.



Now let us look at an example in which Jenkins is used for testing in different environments like: Ubuntu, MAC, Windows etc.

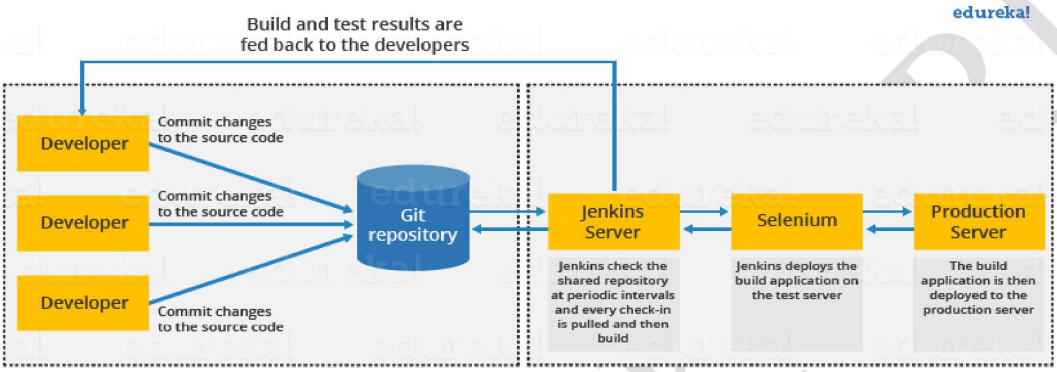
The diagram below represents the same:



The following functions are performed in the above image:

- Jenkins checks the Git repository at periodic intervals for any changes made in the source code.
- Each builds requires a different testing environment which is not possible for a single Jenkins server. In order to perform testing in different environments Jenkins uses various Slaves as shown in the diagram.
- Jenkins Master requests these Slaves to perform testing and to generate test reports.

Continuous Integration with Jenkins



The above diagram is depicting the following functions:

- First, a developer commits the code to the source code repository. Meanwhile, the Jenkins server checks the repository at regular intervals for changes.
- Soon after a commit occurs, the Jenkins server detects the changes that have occurred in the source code repository. Jenkins will pull 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 deploys the built in the test server.
- After testing, Jenkins generates a feedback and then notifies the developers about the build and test results.
- It will continue to check the source code repository for changes made in the source code and the whole process keeps on repeating.

Before and After Jenkins

Before Jenkins	After Jenkins
The entire source code was built and then tested. Locating and fixing bugs in the event of build and test failure was difficult and time-consuming, which in turn slows the software delivery process.	Every commit made in the source code is built and tested. So, instead of checking the entire source code developers only need to focus on a particular commit. This leads to frequent new software releases.
Developers have to wait for test results	Developers know the test result of every commit made in the source code on the run.
The whole process is manual	You only need to commit changes to the source code and Jenkins will automate the rest of the process for you.

Jenkins Pipeline

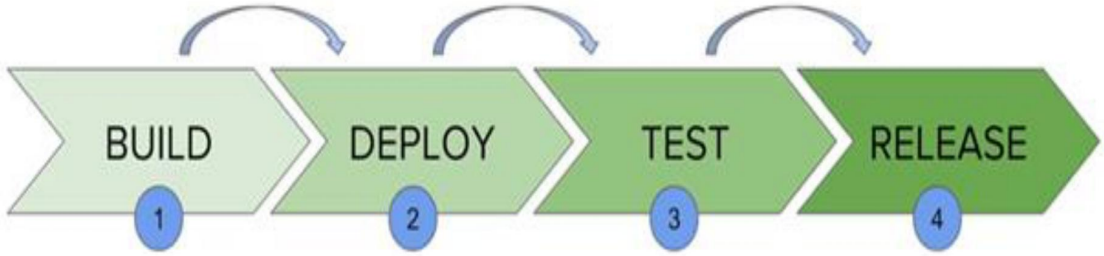
In Jenkins, a pipeline is a collection of events or jobs which are interlinked with one another in a sequence. It is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins.

In other words, a Jenkins Pipeline is a collection of jobs or events that brings the software from version control into the hands of the end users by using automation tools. It is used to incorporate continuous delivery in our software development workflow.

A pipeline has an extensible automation server for creating simple or even complex delivery pipelines "as code", via DSL (Domain-specific language).

What is Continuous Delivery Pipeline?

In a Jenkins Pipeline, every job has some sort of dependency on at least one or more jobs or events.



- The above diagram represents a continuous delivery pipeline in Jenkins. It contains a collection of states such as build, deploy, test and release. These jobs or events are interlinked with each other. Every state has its jobs, which work in a sequence called a continuous delivery pipeline.

- A continuous delivery pipeline is an automated expression to show your process for getting software for version control. Thus, every change made in your software goes through a number of complex processes on its manner to being released. It also involves developing the software in a repeatable and reliable manner, and progression of the built software through multiple stages of testing and deployment.

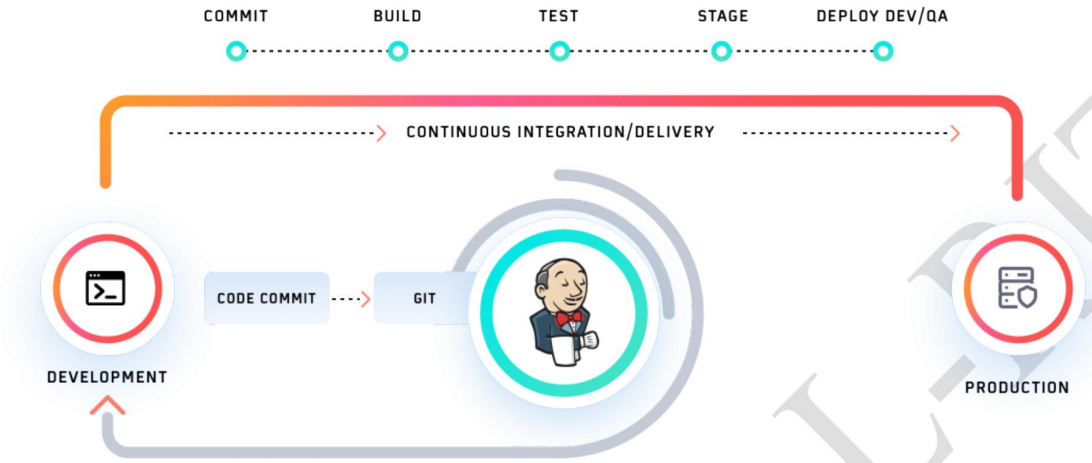
- In simple words, continuous delivery is the capability to release software at all times. It is a practice which ensures that the software is always in a production-ready state.

- It means that every time a change is made to the code or the infrastructure, the software team must work in such a way that these changes are built quickly and tested using various automation tools after which the build is subjected to production.

- By speeding up the delivery process, the development team will get more time to implement any required feedback. This process, of getting the software from the build to the production state at a faster rate is carried out by implementing continuous integration and continuous delivery.

- Continuous delivery ensures that the software is built, tested, and released more frequently. It reduces the cost, time, and risk of incremental software releases. To carry out continuous delivery, Jenkins introduced a new feature called Jenkins pipeline

- Inside Jenkins CI/CD, a pipeline is defined as a series of events or tasks which are interconnected in a particular order. In simple terms, Jenkins pipeline is a set of modules or plugins which enable the implementation and integration of Continuous Delivery pipelines within Jenkins.



Advantages of Jenkins Pipeline

- By using Groovy DSL (Domain Specific Language), it models easy to complex pipelines as code.
- Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- It supports complex pipelines by incorporating conditional loops, fork or join operations and allowing tasks to be performed in parallel
- It is durable in terms of unplanned restart of the Jenkins master
- The code is stored in a text file called the Jenkinsfile which can be checked into a SCM (Source Code Management)
- It can integrate with several other plugins

JenkinsFile

A Jenkinsfile is a text file that stores the entire workflow as code and it can be checked into a SCM on your local system. It can be reviewed in a Source Code Management (SCM) platform such as Git. This enables the developers to access, edit and check the code at all times.

The Jenkinsfile is written using the Groovy **Domain-Specific Language** and can be generated using a text editor or the Jenkins instance configuration tab.

Jenkins Pipeline Concepts

Pipeline

This is a user defined block which contains all the processes such as build, test, deploy, etc. It is a collection of all the stages in a Jenkinsfile. All the stages and steps are defined within this block. It is the key block for a declarative pipeline syntax.

```
pipeline {

}
```

Node

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline. It is a key part of the scripted pipeline syntax.

```
node {

}
```

There are various mandatory sections which are common to both the declarative and scripted pipelines, such as stages, agent and steps that must be defined within the pipeline. These are explained below:

Agent

An agent is a directive that can run multiple builds with only one instance of Jenkins. This feature helps to distribute the workload to different agents and execute several projects within a single Jenkins instance. It instructs Jenkins to allocate an executor for the builds.

A single agent can be specified for an entire pipeline or specific agents can be allotted to execute each stage within a pipeline. Few of the parameters used with agents are:

Any

Runs the pipeline/ stage on any available agent.

None

This parameter is applied at the root of the pipeline and it indicates that there is no global agent for the entire pipeline and each stage must specify its own agent.

Label

Executes the pipeline/stage on the labelled agent.

Docker

This parameter uses docker container as an execution environment for the pipeline or a specific stage. In the below example I'm using docker to pull an ubuntu image. This image can now be used as an execution environment to run multiple commands.

```
pipeline {
    agent {
        docker {
            image 'ubuntu'
        }
    }
}
```

Stages

This block contains all the work that needs to be carried out. The work is specified in the form of stages. There can be more than one stage within this directive. Each stage performs a specific task. In the following example, I've created multiple stages, each performing a specific task.

```
pipeline {
        agent any
        stages {
                stage ('Build') {
                        ...
                }
                stage ('Test') {
                        ...
                }
                stage ('QA') {
                        ...
                }
                stage ('Deploy') {
                        ...
                }
                stage ('Monitor') {
                        ...
                }
        }
}
```

Steps

A series of steps can be defined within a stage block. These steps are carried out in sequence to execute a stage. There must be at least one step within a steps directive. In the following example I've implemented an echo command within the build stage. This command is executed as a part of the 'Build' stage.

```
pipeline {
        agent any
        stages {
                stage ('Build') {
                        steps {
                                echo 'Running build phase...'
                        }
                }
        }
}
```

Two types of syntax are used for defining your JenkinsFile.

Scripted
Declarative

1. Declarative pipeline syntax

- Declarative pipeline is a relatively new feature that supports the pipeline as code concept. It makes the pipeline code easier to read and write.
- This code is written in a Jenkinsfile which can be checked into a source control management system such as Git. The declarative pipeline is defined within a 'pipeline' block

In Declarative Pipeline syntax, the pipeline block defines all the work done throughout your entire Pipeline.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any (1)
    stages {
        stage('Build') { (2)
            steps {
                // (3)
            }
        }
        stage('Test') { (4)
            steps {
                // (5)
            }
        }
        stage('Deploy') { (6)
            steps {
                // (7)
            }
        }
    }
}
```

1. Execute this Pipeline or any of its stages, on any available agent.
2. Defines the "Build" stage.
3. Perform some steps related to the "Build" stage.
4. Defines the "Test" stage.
5. Perform some steps related to the "Test" stage.
6. Defines the "Deploy" stage.
7. Perform some steps related to the "Deploy" stage.

Here is an example of a JenkinsFile using Declarative Pipeline syntax

Jenkinsfile (Declarative Pipeline)

```
pipeline { (1)
    agent any (2)
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') { (3)
            steps { (4)
                sh 'make' (5)
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' (6)
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

1. pipeline is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.
2. agent is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.
3. stage is a syntax block that describes a stage of this Pipeline. Read more about stage blocks in Declarative Pipeline syntax on the Pipeline syntax page. As mentioned above, stage blocks are optional in Scripted Pipeline syntax.
4. steps is Declarative Pipeline-specific syntax that describes the steps to be run in this stage.
5. sh is a Pipeline step (provided by the Pipeline: Nodes and Processes plugin) that executes the given shell command.junit is another Pipeline step (provided by the
6. JUnit plugin) for aggregating test reports.

Scripted Pipeline Syntax

- scripted pipeline is a traditional way of writing the code. In this pipeline, the Jenkinsfile is written on the Jenkins UI instance. scripted pipeline is defined within a 'node' block.
- The scripted pipeline uses stricter groovy based syntaxes because it was the first pipeline to be built on the groovy foundation. Since this Groovy script was not typically desirable to all the users, the declarative pipeline was introduced to offer a simpler and more optioned Groovy syntax.

In Scripted Pipeline syntax, one or more node blocks do the core work throughout the entire Pipeline. Although this is not a mandatory requirement of Scripted Pipeline syntax, confining your Pipeline's work inside of a node block does two things:

i. Schedules the steps contained within the block to run by adding an item to the Jenkins queue. As soon as an executor is free on a node, the steps will run.
ii. Creates a workspace (a directory specific to that particular Pipeline) where work can be done on files checked out from source control.

Jenkinsfile (Scripted Pipeline)

```
node { 1
    stage('Build') { 2
        // 3
    }
    stage('Test') { 4
        // 5
    }
    stage('Deploy') { 6
        // 7
    }
}
```

Jenkinsfile (Scripted Pipeline)

```
node { 7
    stage('Build') { 3
        sh 'make' 5
    }
    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml' 6
    }
    if (currentBuild.currentResult == 'SUCCESS') {
        stage('Deploy') {
            sh 'make publish'
        }
    }
}
```

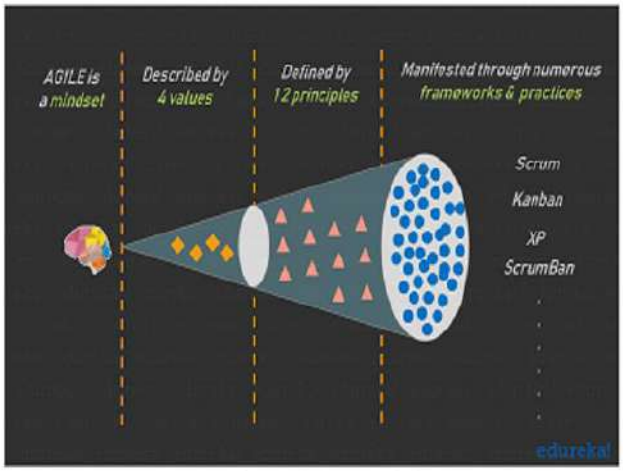
ABHISHEK K L
Assist.Prof, MCA-RIT

Overview of Agile Software Development

What is Agile Methodology?

Agile is a mind-set, a philosophy, more like a way of thinking, a movement focusing on communication, team-thinking, intrinsic motivation, innovative talks, flow, and value creation while developing a product.

At its core, Agile is a set of principles for actions that keep a software development team organized and efficient. Agile started when in 2001, a team of software developers got together in Utah for a weekend of fun and discussion. They compiled their views and principles into a document that is easy to understand and apply. Which we refer to as Agile Manifesto.



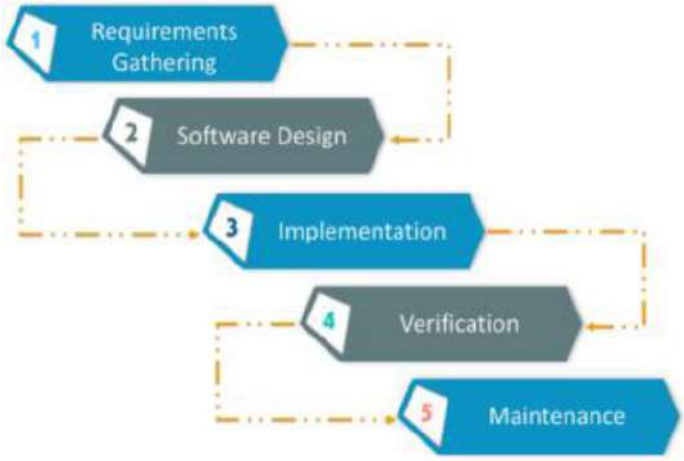
Agile Methodology meaning a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project. In the Agile model in software testing, both development and testing activities are concurrent, unlike the Waterfall model.

What is Agile Software Development?

The Agile software development methodology is one of the simplest and effective processes to turn a vision for a business need into software solutions. Agile is a term used to describe software development approaches that employ continual planning, learning, improvement, team collaboration, evolutionary development, and early delivery. It encourages flexible responses to change.

Why do we need Agile methodology?

Before Agile came into the picture, we had the Waterfall model of software development. The waterfall model can be defined as a sequential process in the development of a system or software that follows a top-down approach. This model was a straight forward and linear model. The waterfall model had various phases such as Requirements Gathering, Software Design, Implementation, Testing, Deployment, and Maintenance.



This model however suffered a few drawbacks such as follows:

- This model was too time-consuming. Unless you complete a particular stage, you cannot proceed to the further stages.
- This model was suitable only for projects where requirements are stable.
- The working software is delivered only after completion of the final stage of the model.
- It is very difficult to go back to the previous stages and make some changes that you had not thought about in the initial phase.

Due to the above-mentioned drawbacks of the Waterfall model, the leaders from the different development methodologies decided to meet to find a way forward through these problems. These leaders agreed upon a lightweight development method and they were also able to give wordings for the same in the form of a manifesto. It was called" The Manifesto for Agile Software Development".

The Manifesto describes 4 values and 12 supporting principles.

The values set out in the Agile Manifesto are:

1. **Individuals and interactions over processes and tools.**
2. **Working software over comprehensive documentation.**
3. **Customer collaboration over contract negotiation.**
4. **Responding to change over following a plan.**

1. Individuals and interactions over processes and tools.

People are the most important ingredient of success. A good process will not save the project from failure if the team doesn't have strong players, but a bad pro- cess can make even the strongest of players ineffective. Even a group of strong players can fail badly if they don't work as a team.

A strong player is not necessarily an ace programmer. A strong player may be an average programmer, but someone who works well with others. Working well with others, communicating and interacting, is more important than raw programming talent. A team of average programmers who communicate well are more likely to succeed than a group of superstars who fail to interact as a team.

The right tools can be very important to success. Compilers, IDEs, source-code control systems, etc. are all vital to the proper functioning of a team of developers. However, tools can be overemphasized. An overabundance of big, unwieldy tools is just as bad as a lack of tools.

My advice is to start small. Don't assume you've outgrown a tool until you've tried it and found you can't use it. Instead of buying the top-of-the-line, mega expensive, source-code control system, find a free one and use it until you can demonstrate that you've outgrown it. Before you buy team licenses for the best of all CASE tools, use white boards and graph paper until you can reasonably show that you need more. Before you commit to the top-shelf behemoth database system, try flat files. Don't assume that bigger and better tools will automatically help you do better. Often they hinder more than they help.

Remember, building the team is more important than building the environment. Many teams and managers make the mistake of building the environment first and expecting the team to gel automatically. Instead, work to create the team, and then let the team configure the environment on the basis of need.

2. Working software over comprehensive documentation.

Software without documentation is a disaster. Code is not the ideal medium for communicating the rationale and structure of a system. Rather, the team needs to produce human-readable documents that describe the system and the rationale for their design decisions.

However, too much documentation is worse than too little. Huge software documents take a great deal of time to produce and even more time to keep in sync with the code. If they are not kept in sync, then they turn into large, complicated lies and become a significant source of misdirection.

It is always a good idea for the team to write and maintain a rationale and structure document, but that document needs to be short and salient. By "short," I mean one or two dozen pages at most. By "salient," I mean it should discuss the overall design rationale and only the highest-level structures in the system. If all we have is a short rationale and structure document, how do we train new team members to work on the system? We work closely with them. We transfer our knowledge to them by sitting next to them and helping them.

We make them part of the team through close training and interaction. The two documents that are the best at transferring information to new team members are the code and the team. The code does not lie about what it does. It may be hard to extract rationale and intent from the code, but the code is the only unambiguous source of information. The team members hold the ever-changing road map of the system in their heads. There is no faster and more efficient way to transfer that road map to others than human to-human interaction.

Many teams have gotten hung up in the pursuit of documentation instead of software. This is often a fatal flaw. There is a simple rule called Martin's first law of documentation that prevents it: Produce no document unless its need is immediate and significant.

3. Customer collaboration over contract negotiation.

Software cannot be ordered like a commodity. You cannot write a description of the software you want and then have someone develop it on a fixed schedule for a fixed price. Time and time again,

attempts to treat software projects in this manner have failed. Sometimes the failures are spectacular.

It is tempting for the managers of a company to tell their development staff what their needs are, and then expect that staff to go away for a while and return with a system that satisfies those needs. However, this mode of operation leads to poor quality and failure.

Successful projects involve customer feedback on a regular and frequent basis. Rather than depending on a contract or a statement of work, the customer of the software works closely with the development team, providing frequent feedback on their efforts.

A contract that specifies the requirements, schedule, and cost of a project is fundamentally flawed. In most cases, the terms it specifies become meaningless long before the project is complete.2 The best contracts are those that govern the way the development team and the customer will work together.

As an example of a successful contract, take one I negotiated in 1994 for a large, multiyear, half-million-line project. We, the development team, were paid a relatively low monthly rate. Large pay-outs were made to us when we delivered certain large blocks of functionality. Those blocks were not specified in detail by the contract. Rather, the contract stated that the pay-out would be made for a block when the block passed the customer's acceptance test. The details of those acceptance tests were not specified in the contract.

During the course of this project, we worked very closely with the customer. We released the software to him almost every Friday. By Monday or Tuesday of the following week, he would have a list of changes for us to put into the software. We would prioritize those changes together and then schedule them into subsequent weeks. The customer worked so closely with us that acceptance tests were never an issue. He knew when a block of functionality satisfied his needs because he watched it evolve from week to week.

The requirements for this project were in a constant state of flux. Major changes were not uncommon. There were whole blocks of functionality that were removed and others that were inserted. Yet the contract, and the project, survived and succeeded. The key to this success was the intense collaboration with the customer and a contract that governed that collaboration rather than trying to specify the details of scope and schedule for a fixed cost.

4. Responding to change over following a plan.

It is the ability to respond to change that often determines the success or failure of a software project. When we build plans, we need to make sure that our plans are flexible and ready to adapt to changes in the business and technology.

The course of a software project cannot be planned very far into the future. First of all, the business environment is likely to change, causing the requirements to shift. Second, customers are likely to alter the requirements once they see the system start to function. Finally, even if we know the requirements, and we are sure they won't change, we are not very good at estimating how long it will take to develop them.

It is tempting for novice managers to create a nice PERT or Gantt chart of the whole project and tape it to the wall. They may feel that this chart gives them control over the project. They can

track the individual tasks and cross them off the chart as they are completed. They can compare the actual dates with the planned dates on the chart and react to any discrepancies.

What really happens is that the structure of the chart degrades. As the team gains knowledge about the system, and as the customers gain knowledge about their needs, certain tasks on the chart become unnecessary. Other tasks will be discovered and will need to be added. In short, the plan will undergo changes in shape, not just changes in dates.

A better planning strategy is to make detailed plans for the next two weeks, very rough plans for the next three months, and extremely crude plans beyond that. We should know the tasks we will be working on for the next two weeks. We should roughly know the requirements we will be working on for the next three months. And we should have only a vague idea what the system will do after a year. This decreasing resolution of the plan means that we are only investing in a detailed plan for those tasks that are immediate. Once the detailed plan is made, it is hard to change since the team will have a lot of momentum and commitment. However, since that plan only governs a few weeks' worth of time, the rest of the plan remains flexible.

12 Principles of Agile Manifesto

The above values inspired the following 12 principles, which are the characteristics that differentiate a set of agile practices from a heavyweight process:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

The MIT Sloan Management Review published an analysis of software development practices that help companies build high-quality products. The article found a number of practices that had a significant impact on the quality of the final system. One practice was a strong correlation between quality and the early delivery of a partially functioning system. The article reported that the less functional the initial delivery, the higher the quality in the final delivery.

Another finding of this article is a strong correlation between final quality and frequent deliveries of increasing functionality. The more frequent the deliveries, the higher the final quality. An agile set of practices delivers early and often. We strive to deliver a rudimentary system within the first few weeks of the start of the project. Then, we strive to continue to deliver systems of increasing functionality every two weeks.

Customers may choose to put these systems into production if they think that they are functional enough. Or they may choose simply to review the existing functionality and report on changes they want made.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

This is a statement of attitude. The participants in an agile process are not afraid of change. They view changes to the requirements as good things, because those changes mean that the team has learned more about what it will take to satisfy the market.

An agile team works very hard to keep the structure of its software flexible so that when requirements change, the impact to the system is minimal. Later in this book we will learn the principles and patterns of object-oriented design that help us to maintain this kind of flexibility.

3. **Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.**

We deliver working software, and we delivery it early (after the first few weeks) and often (every few weeks thereafter). We are not content with delivering bundles of documents or plans. We don't count those as true deliveries. Our eye is on the goal of delivering software that satisfies the customer's needs.

4. **Business people and developers must work together daily throughout the project.**

In order for a project to be agile, there must be significant and frequent interaction between the customers, developers, and stakeholders. A software project is not like a fire-and-forget weapon. A software project must be continuously guided.

5. **Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.**

An agile project is one in which people are considered the most important factor of success. All other factors—process, environment, management, etc.—are considered to be second order effects, and they are subject to change if they are having an adverse effect upon the people. For example, if the office environment is an obstacle to the team, the offifice environment must be changed. If certain process steps are an obstacle to the team, the process steps must be changed.

6. **The most efficient and effective method of conveying information to and within a development team is face to-face conversation.**

In an agile project, people talk to each other. The primary mode of communication is conversation. Documents may be created, but there is no attempt to capture all project information in writing. An agile project team does not demand written specs, written plans, or written designs. Team members may create them if they perceive an immediate and significant need, but they are not the default. The default is conversation.

7. **Working software is the primary measure of progress.**

Agile projects measure their progress by measuring the amount of software that is currently meeting the customer's need. They don't measure their progress in terms of the phase that they are in or by the volume of documentation that has been produced or by the amount of infrastructure code they have created. They are 30% done when 30% of the necessary functionality is working.

8. **Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.**

An agile project is not run like a 50-yard dash; it is run like a marathon. The team does not take off at full speed and try to maintain that speed for the duration. Rather, they run at a fast, but sustainable, pace.

Running too fast leads to burnout, shortcuts, and debacle. Agile teams pace themselves. They don't allow themselves to get too tired. They don't borrow tomorrow's energy to get a bit more done today. They work at a rate that allows them to maintain the highest quality standards for the duration of the project.

9. Continuous attention to technical excellence and good design enhances agility.

High quality is the key to high speed. The way to go fast is to keep the software as clean and robust as possible. Thus, all agile team members are committed to producing only the highest quality code they can. They do not make messes and then tell themselves they'll clean it up when they have more time. If they make a mess, they clean it up before they finish for the day.

10. Simplicity—the art of maximizing the amount of work not done—is essential.

Agile teams do not try to build the grand system in the sky. Rather, they always take the simplest path that is consistent with their goals. They don't put a lot of importance on anticipating tomorrow's problems, nor do they try to defend against all of them today. Instead, they do the simplest and highest-quality work today, confident that it will be easy to change if and when tomorrow's problems arise.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

An agile team is a self-organizing team. Responsibilities are not handed to individual team members from the outside. Responsibilities are communicated to the team as a whole, and the team determines the best way to fulfil them. Agile team members work together on all aspects of the project. Each is allowed input into the whole. No single team member is responsible for the architecture or the requirements or the tests. The team shares those responsibilities, and each team member has influence over them.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

An agile team continually adjusts its organization, rules, conventions, relationships, etc. An agile team knows that its environment is continuously changing and knows that they must change with that environment to remain agile.

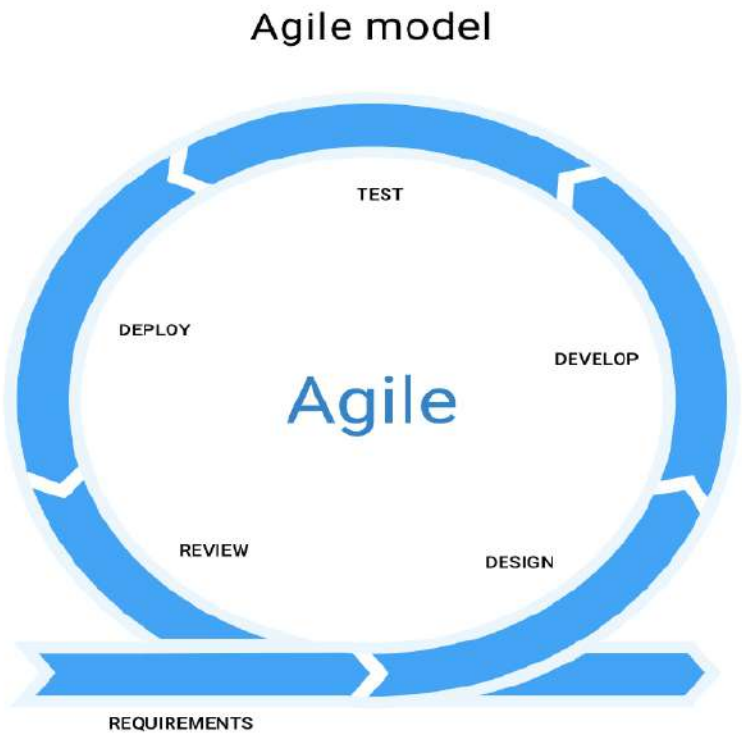
ABHISHEK K L
Assist.Prof, MCA-RIT

Agile Model Vs Waterfall Model

Agile Model	Waterfall Model
Agile methodology definition: Agile methodologies propose incremental and iterative approach to software design	**Waterfall Model:** Development of the software flows sequentially from start point to end point.
The **Agile process** in software engineering is broken into individual models that designers work on	The design process is not broken into an individual models
The customer has early and frequent opportunities to look at the product and make decision and changes to the project	The customer can only see the product at the end of the project
Small projects can be implemented very quickly. For large projects, it is difficult to estimate the development time.	All sorts of project can be estimated and completed.
Error can be fixed in the middle of the project.	Only at the end, the whole product is tested. If the requirement error is found or any changes have to be made, the project has to start from the beginning
Development process is iterative, and the project is executed in short (2-4) weeks iterations. Planning is very less.	The development process is phased, and the phase is much bigger than iteration. Every phase ends with the detailed description of the next phase.
Documentation attends less priority than software development	Documentation is a top priority and can even use for training staff and upgrade the software with another team
Every iteration has its own testing phase. It allows implementing regression testing every time new functions or logic are released.	Only after the development phase, the testing phase is executed because separate parts are not fully functional.
In agile testing when an iteration end, shippable features of the product is delivered to the customer. New features are usable right after shipment. It is useful when you have good contact with customers.	All features developed are delivered at once after the long implementation phase.
Testers and developers work together	Testers work separately from developers
At the end of every sprint, user acceptance is performed	User acceptance is **performed** at the end of the project.
It requires close communication with developers and together analyse requirements and planning	Developer does not involve in requirement and planning process. Usually, time delays between tests and coding

ABHISHEK K L
Assist.Prof, MCA-RIT

PHASES OF AGILE METHODOLOGY



Phase 1: Requirements

Before a Product Owner can even start designing a project, they need to create the initial documentation that will list the initial requirements. They are:

- The end result the project is going to achieve. For example, a text editor;
- The features that it will support. For example, different font sizes;
- The features that it will not initially support. For example, adding animations to the text or ability to embed video;

A general recommendation is to lower these initial requirements as hard as one can, adding only the definitely necessary features and ignoring ones that won't be used often. Developers can work on them later, once the app is deployed and the core features work well. On further iterations, the Client and the Product Owner review the requirements and make them more relevant.

Phase 2: Design

There are two ways to approach design in the software development — one is the visual design and the other is the architectural structure of the app.

Software Design

During the first iteration, the Product Owner assembles their development team and introduces the requirements created during the previous stage. The team then discusses how to tackle these requirements, and proposes the tools needed to achieve the best result. For example, the team defines the programming language, frameworks, and libraries that the project is going to be using. On further iterations, the developers discuss the feature implementation and the internal structure of the come.

UI/UX Design

During this SDLC stage, the designers create a rough mock-up of the UI. If the product is consumer-grade, the user interface and user experience are most important. So it's generally a good idea to review the possible competitors to see what they are doing right — and especially what they are doing wrong.
Further iterations are spent refining the initial design and/or reworking it to suit the new features.

Phase 3. Development and Coding

The development phase is about writing code and converting design documentation into the actual software within the software development process. This stage of SDLC is generally the longest as it's the backbone of the whole process. There aren't many changes between the iterations here.

Phase 4. Integration and Testing

This stage is spent on making sure that the software is bug-free and compatible with everything else that the developers have written before. The Quality Assurance team conducts a series of tests in order to ensure the code is clean and business goals of the solution are met.
During the further iterations of this SDLC stage, the testing becomes more involved and accounts not only for functionality testing, but also for systems integration, interoperability, and user acceptance testing, etc.

Phase 5. Implementation and Deployment

The application is deployed on the servers and provided to the customers — either for the demo or the actual use. Further iterations update the already installed software, introducing new features and resolving bugs.

Phase 6. Review

Once all previous development phases are complete, the Product Owner gathers the Development Team once again and reviews the progress made towards completing the requirements. The team introduces their ideas toward resolving the problems that arose during the previous phases and the Product Owner takes their propositions into consideration.

Afterwards, the Agile software development lifecycle phases start anew — either with a new iteration or by moving toward the next stage and scaled Agile.

Advantages of Agile methodology

The various advantages offered by AGILE methodology are as follows:

- Agile software development allows the team to work together more efficiently and effectively to develop complex projects.

- It consists of practices that exercise iterative and incremental techniques that are easily adopted and display great results.
- In AGILE, the delivery of the software is persistent.
- After every Sprint, working feature(s) is/are delivered to the customer. This increases the level of satisfaction in them.
- Customers can have a look at the developed features and check if they meet their expectations.
- If the customers have any feedback or they want any changes in the features, then it can be accommodated in the current or maybe the next release of the product.
- Changes can be made even in the later stages of the development of the product.
- In AGILE, the business people and the developers of the product interact daily.
- A significant amount of attention is paid to the design of the product.

How to implement Agile methodology?

The AGILE methodology can be implemented with the help of various frameworks such as Scrum, Kanban, EXtreme Programming (XP), Lean, Crystal, Adaptive Project Framework (APF), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), Agile Unified Process (AUP) and so on.

ABHISHEK K L
Assist.Prof, MCA-RIT

Introduction to DevOps

What is DevOps?

DevOps is the combination of **cultural philosophies, practices, and tools** which **collaborates Development** and **IT Operations** to make software production and deployment in an **automated & repeatable** way to **increase** the **organization's speed** to **deliver software applications and services**.

Why is DevOps is Needed?

- Before DevOps, the development and operation team worked in complete isolation.
- Testing and Deployment were isolated activities done after design-build. Hence they consumed more time than actual build cycles.
- Without using DevOps, team members are spending a large amount of their time in testing, deploying, and designing instead of building the project.
- Manual code deployment leads to human errors in production
- Coding & operation teams have their separate timelines and are not in synch causing further delays.

To bride these gaps DevOps is needed.

Note: There is a demand to increase the rate of software delivery by business stakeholders. As per Forrester Consulting Study, only 17% of teams can use delivery software fast enough. This proves the pain point.

When to adopt DevOps?

DevOps should be used for large distributed applications such as e-commerce sites or applications hosted on a cloud platform.

When not to adopt DevOps?

It should not be used in a mission-critical application like bank, power and other sensitive data sites. Such applications need strict access controls on the production environment, a detailed change management policy, access control policy to the data centers.

How is DevOps different from traditional IT?

Let's compare traditional software waterfall model with DevOps to understand the changes DevOps bring.

We assume the application is scheduled to go live in 2 weeks and coding is 80% done. We assume the application is a fresh launch and the process of buying servers to ship the code has just begun-

Old Process	DevOps
After placing an order for new servers, the Development team works on testing. The Operations team works on extensive paperwork as required in enterprises to deploy the infrastructure.	After placing an order for new servers Development and Operations team work together on the paperwork to set-up the new servers. This results in better visibility of infrastructure requirement.
Projection about failover, redundancy, data center locations, and storage requirements are skewed as no inputs are available from developers who have deep knowledge of the application.	Projection about failover, redundancy, disaster recovery, data center locations, and storage requirements are pretty accurate due to the inputs from the developers.
Operations team has no clue on the progress of the Development team. Operations team develop a monitoring plan as per their understanding.	In DevOps, the Operations team is completely aware of the progress the developers are making. Operations team interact with developers and jointly develop a monitoring plan that caters to the IT and business needs. They also use advance Application Performance Monitoring (APM) Tools
Before go-live, the load testing crashes the application. The release is delayed.	Before go-live, the load testing makes the application a bit slow. The development team quickly fixes the bottlenecks. The application is released on time.

Why is DevOps used?

1. **Predictability:** DevOps offers significantly lower failure rate of new releases.
2. **Reproducibility:** Version everything so that earlier version can be restored.
3. **Maintainability:** Effortless process of recovery in the event of a new release crashing or disabling the current system.
4. **Time to market:** DevOps reduces the time to market up to 50% through streamlined software delivery. This is particularly the case for digital and mobile applications
5. **Greater Quality:** DevOps helps the team to provide improved quality of application development as it incorporates infrastructure issues.
6. **Reduced Risk:** DevOps incorporates security aspects in the software delivery lifecycle. It helps in reduction of defects across the lifecycle.
7. **Cost Efficiency:** DevOps offers cost efficiency in the software development process which is always an aspiration of IT companies' management.

DevOps Lifecycle

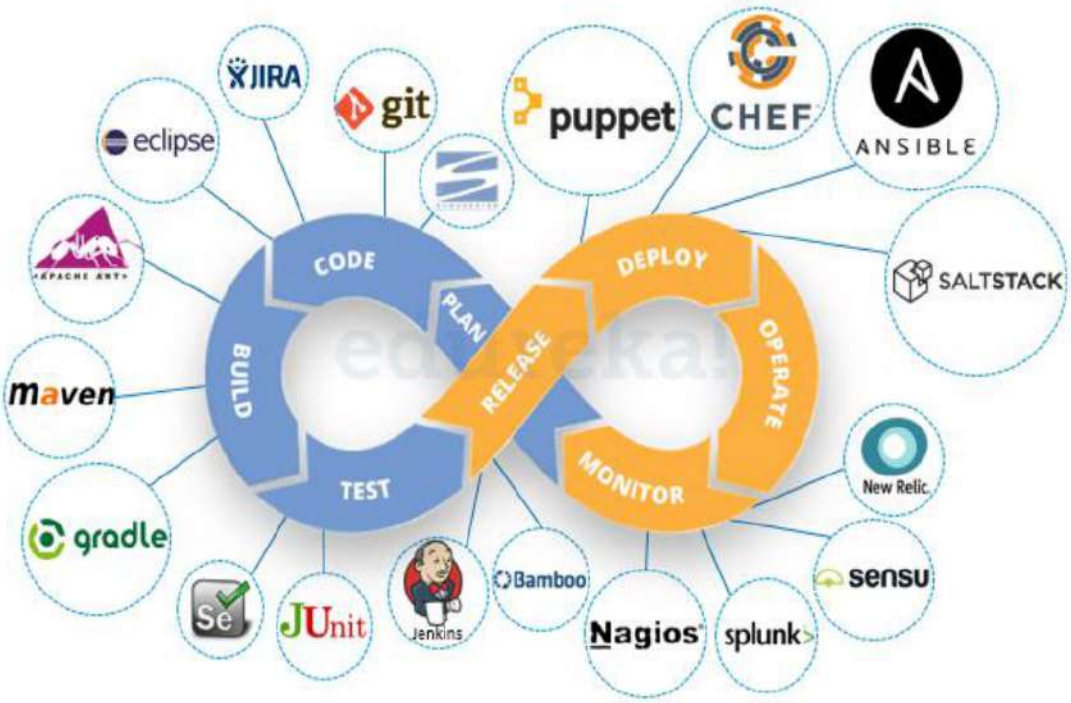


Figure: Life Cycle of DevOps

DevOps Lifecycle can be broadly broken down into the below DevOps Stages:

1. **Continuous Development**
2. **Continuous Integration**
3. **Continuous Testing**
4. **Continuous Deployment**
5. **Continuous Monitoring**

1. Continuous Development

Tools Used: Git, SVN, Mercurial, CVS

- This is the phase that involves 'planning 'and 'coding'of the software.
- This is the phase where the team members sit down and visualize the outcome or, in other words, how the software application will turn out to be.
- Once the vision is in place, the developers start writing and maintaining the code. Developing the source code for application begins by choosing from the different programming languages. JavaScript, C/C++, Ruby, and Python are prominently used for coding applications in DevOps.
- There are no DevOps tools required for planning, but many version control tools are used to maintain code. **The process of maintaining the code using version control tools is called as Source Code Management (SCM).**
- Popular tools for version control include **Git, TFS, Subversion, and Mercurial**.

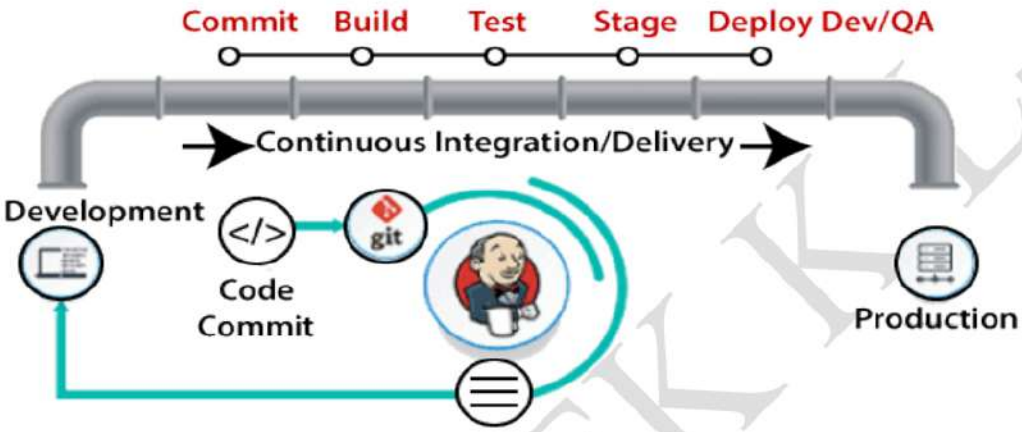
2. Continuous Integration

Tools: Jenkins, TeamCity, Travis

Continuous Integration is a development practice where developers integrate code into a shared repository frequently where each integration is verified by an automated build and automated tests.

This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present.

The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.



Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.

3. Continuous Testing

Tools: Jenkins, Selenium TestNG, JUnit

This phase, where the developed software is continuously testing for bugs, defects and flaws. This is also the phase where the usability of the software is tested using the set of best practices for QA, and it is determined whether the software meets the specifications defined by the client. Continuous testing is carried out using automation testing tools, which can be open source tools like Selenium or advanced test management tools like TestNG, JUnit, Selenium.

Selenium does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.

Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

4. Continuous Deployment

Tools Used: Configuration Management – Chef, Puppet, Ansible
Containerization – Docker, Vagrant

In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.

In this phase **Chef, Puppet, Ansible are used as** Configuration Management tools. Configuration Management is the act of releasing deployments to servers, scheduling updates on all servers and most importantly keeping the configurations consistent across all the servers.

Containerization tools are also playing an essential role in the deployment phase. Vagrant and Docker are popular tools that are used for this purpose.

These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.

Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

5. Continuous Monitoring

Tools Used: Splunk, ELK Stack, Nagios, New Relic

Monitoring the performance of a software product is essential to determine the overall efficiency of the product. Through continuous monitoring, developers can identify general patterns and gray areas in the app where more effort is required.

Different system errors such as 'server not reachable', 'low memory', etc., are resolved in the continuous monitoring phase. It also maintains the availability and security of the services.

Network issues and other problems are automatically fixed during this phase at the time of their detection.

Tools such as Nagios, Splunk, Sensu, ELK Stack, and NewRelic are used by the operations team to monitor user activities for improper behavior.

As a result, during continuous monitoring, developers can proactively check the overall health of the system. Proactive checking improves the reliability and productivity of the system and also reduces maintenance costs.

Moreover, important and major issues are directly reported to the development team to be corrected in the initial stages. This leads to faster resolution of issues.

Compared to the software development teams, the IT operations teams are more involved in this phase. Their role is pivotal in supervising user activity, checking the system for unusual behaviour, and tracing the presence of bugs.

DevOps Roles

The role of a DevOps engineer combines aspects of a technical role and an IT operations role.

1. **DevOps Evangelist –** The principal officer (leader) responsible for implementing DevOps.
2. **Release Manager –** The one releasing new features & ensuring post-release product stability.
3. **Automation Expert –** The guy responsible for achieving automation & orchestration of tools.
4. **Software Developer/ Tester –** The one who actually develops the code and tests it.
5. **Quality Assurance –** The one who ensures quality of the product confirms to its requirement.
6. **Security Engineer** – The one always monitoring the product's security & health.

Core responsibilities of DevOps Engineer:

- Building and setting up new development tools and infrastructure.
- Understanding the needs of stakeholders and conveying this to developers.
- Working on ways to automate and improve development and release processes.
- Testing and examining code written by others and analysing results.
- Ensuring that systems are safe and secure against cybersecurity threats
- Identifying technical problems and developing software updates and 'fixes'
- Selecting and deploying appropriate CI/CD tools
- Strive for continuous improvement and build continuous integration, continuous development, and constant deployment pipeline (CI/CD Pipeline)

Essential Skills for a DevOps Engineer

A DevOps Engineer's role requires technical skills in the development cycle and operations skills for maintenance and support. Computer Science or Computer Technology graduates can bring some of the technical skills necessary to become a DevOps engineer. However, the skills required for managing the operations usually come through the experience or by enrolling in specific development programs, which can help further the career in the set direction.

Pre-requisite skills required for a DevOps Engineer include:

- Experience working on Linux based infrastructure.
- Excellent understanding of Ruby, Python, Perl, and Java

- Configuration and managing databases such as MySQL, Mongo.
- Excellent troubleshooting skills.
- Working knowledge of various tools, open-source technologies, and cloud services
- Excellent organisational and time management skills, and the ability to work on multiple projects at the same time.
- excellent team working and communication skills
- knowledge of programming languages.
- strong problem-solving skills.

Differences between DevOps and Agile

Parameter	DevOps	Agile
Definition	DevOps is a practice of bringing development and IT operation teams together.	Agile refers to the continuous iterative approach, which focuses on collaboration, customer feedback, small, and rapid releases.
Purpose	DevOps purpose is to manage end to end engineering processes.	The agile purpose is to manage complex projects.
Team skillset	The DevOps divides and spreads the skill set between development and the operation team.	The Agile development emphasizes training all team members to have a wide variety of similar and equal skills.
Implementation	DevOps is focused on collaboration, so it does not have any commonly accepted framework.	Agile can implement within a range of tactical frameworks such as safe , scrum etc,.
Target areas	End to End business solution and fast delivery.	Software development.
Feedback	Feedback comes from the internal team.	In Agile, feedback is coming from the customer.
Automation	Automation is the primary goal of DevOps. It works on the principle of maximizing efficiency when deploying software.	Agile does not emphasize on the automation.
Documentation	DevOps teams relying on detailed specifications to document their knowledge and understand the software.	Agile teams participate in daily stand-up meetings, opting for face-to-face communication over documentation

ABHISHEK K L
Assist.Prof, MCA-RIT

Source Code Management With Git

What is version control?

- 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.
- Version control systems are software tools that help software teams manage changes to source code over time.

Why Version Control system is so Important?

- As we know that a software product is developed in collaboration by a group of developers they might be located at different locations and each one of them contributes in some specific kind of functionality/features.
- So, in order to contribute to the product, they made modifications in the source code (either by adding or removing).
- A version control system is a kind of software that helps the developer team to efficiently communicate and manage(track) all the changes that have been made to the source code along with the information like who made and what change has been made.
- Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.
- Developers can compare earlier versions of the code with an older version to fix the mistakes.

Benefits of the version control system:

1. Enhances the project development speed by providing efficient collaboration.
2. Leverages the productivity, expedite product delivery, and skills of the employees through better communication and assistance.
3. Employees or contributor of the project can contribute from anywhere irrespective of the different geographical locations through this VCS.
4. Helps in recovery in case of any disaster or contingent situation.
5. The ability to trace each change with a message describing the purpose and intent of the change and connect it to project management and bug tracking software.
6. Informs us about Who, What, When, Why changes have been made.

Types of Version Control Systems:

1. Centralized Version Control Systems **(CVCS)**
2. Distributed Version Control Systems **(DVCS)**

Centralized Version Control System (CVCS)

- A centralized version control system has a single server that contains all the file versions. This enables multiple clients to simultaneously access files on the server, pull them to their local computer or push them onto the server from their local computer.

- This way, everyone usually knows what everyone else on the project is doing. Administrators have control over who can do what. This allows for easy collaboration with other developers or a team.

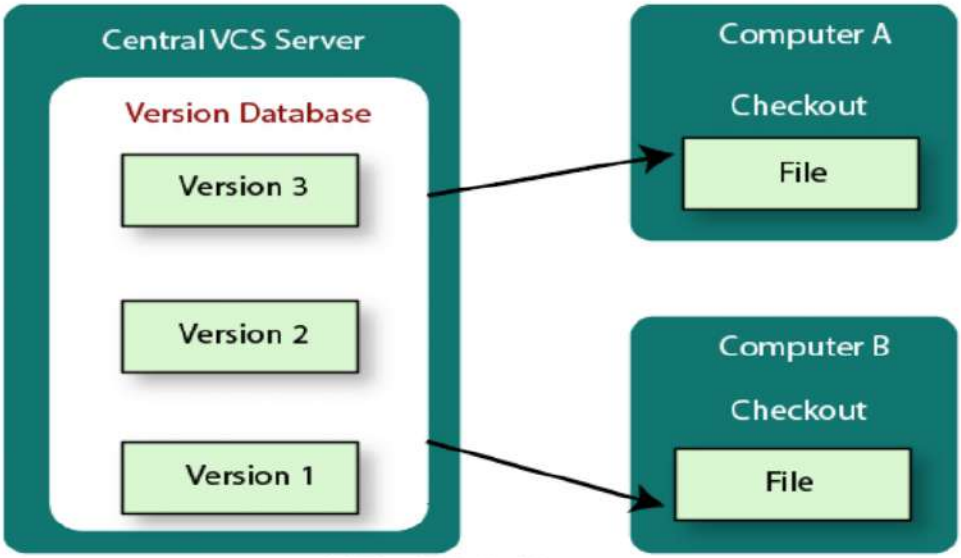


Fig: Centralized Version Control System

- The main concept of a centralized system is that it works in a client and server relationship. The repository is located in one place and provides access to many clients. It's very similar to FTP in where you have an FTP client which connects to an FTP server. All changes, users, commits and information must be sent and received from this central repository.

- The most well-known examples of centralized version control systems are **Microsoft Team Foundation Server (TFS) and Subversion (SVN).**

Drawbacks of CVCS

- The biggest issue with this structure is that everything is stored on the centralized server. If something were to happen to that server, nobody can save their versioned changes, pull files or collaborate at all.
- if the central database became corrupted, and backups haven't been kept, you lose the entire history of the project except whatever single snapshots people happen to have on their local machines.
- Dependent on access to the server.
- It can be slower because every command connects to the server.

Distributed Version Control System (DVCS)

- With distributed version control systems, clients don't just check out the latest snapshot of the files from the server, they fully mirror the repository, including its full history.
- Thus, everyone collaborating on a project owns a local copy of the whole project, i.e. owns their own local database with their own complete history.
- In distributed version control, each user has their own copy of the entire repository, not just the files but the history as well. Think of it as a network of individual repositories.
- With this model, if the server becomes unavailable or dies, any of the client repositories can send a copy of the project's version to any other client or back onto the server when it becomes available. It is enough that one client contains a correct copy which can then easily be further distributed.
- The two most popular distributed version control systems are **Git and Mercurial.**

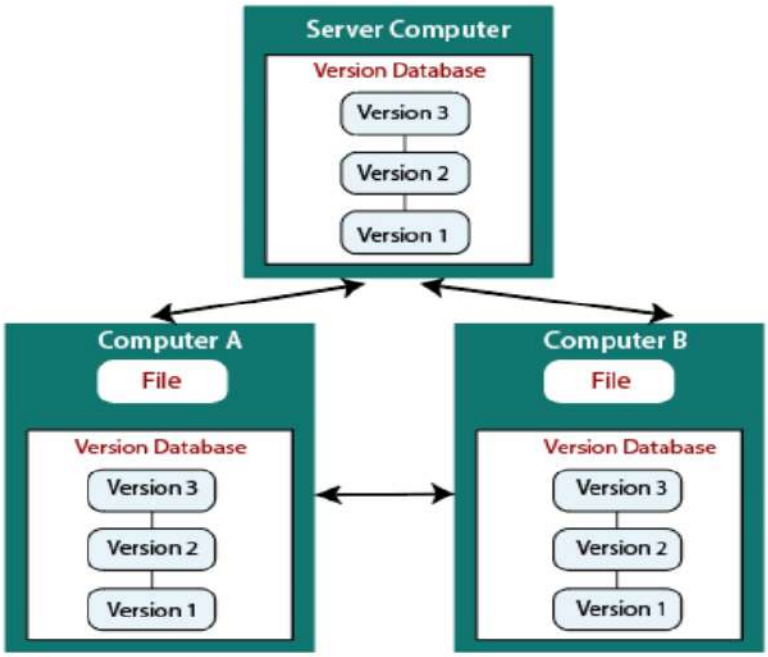


Fig: Distributed Version Control System

Difference between Centralized Version Control System and Distributed Version Control System

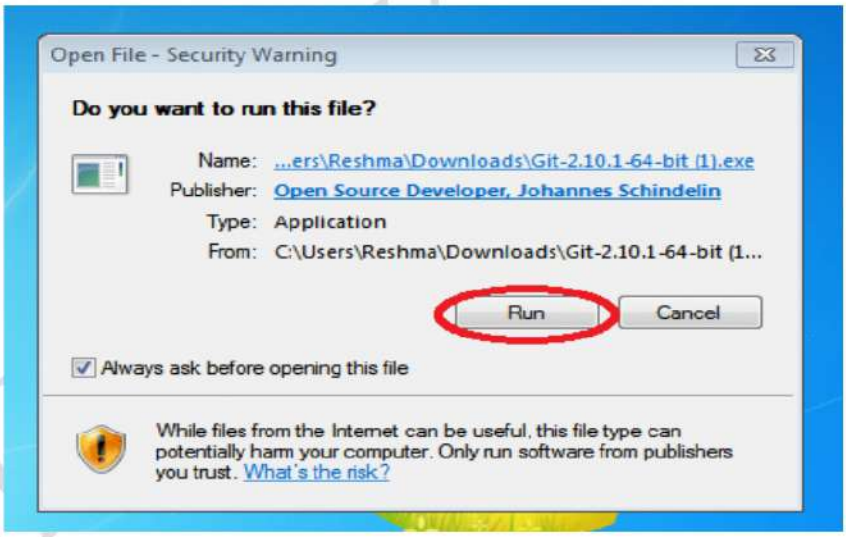
Centralized Version Control System	Distributed Version Control System
In CVCS, the repository is placed at one place and delivers information to many clients.	In DVCS, every user has a local copy of the repository in place of the central repository on the server-side.
In CVCS, the server provides the latest code to all the clients across the globe.	In DVCS, every user can check out the snapshot of the code, and they can fully mirror the central repository.

CVCS is easy to administrate and has additional control over users and access by its server from one place.	DVCS is fast comparing to CVCS as you don't have to interact with the central server for every command.
The popular tools of CVCS are **SVN** (Subversion) and **CVS**.	The popular tools of DVCS are **Git** and **Mercurial**.
CVCS is easy to understand for beginners.	DVCS has some complex process for beginners.
If the server fails, no system can access data from another system.	if any server fails and other systems were collaborating via it, that server can restore any of the client repositories

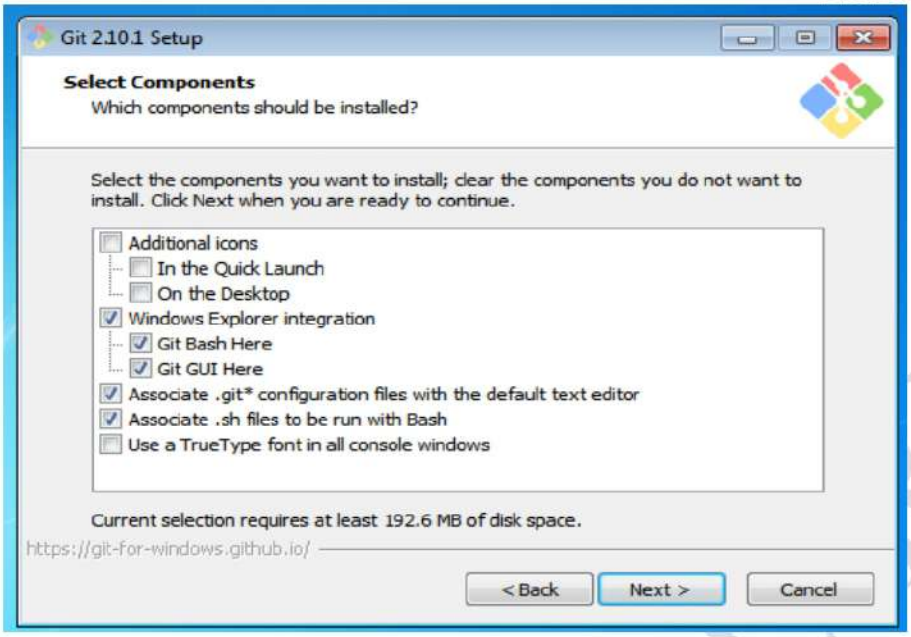
Install Git on Windows

Step 1:
To download the latest version of Git, click on the link below:
https://git-scm.com/download/win/

Step 2:
After your download is complete, **Run** the .exe file in your system.



Step 3:
After you have pressed the **Run** button and agreed to the license, you will find a window prompt to select components to be installed.

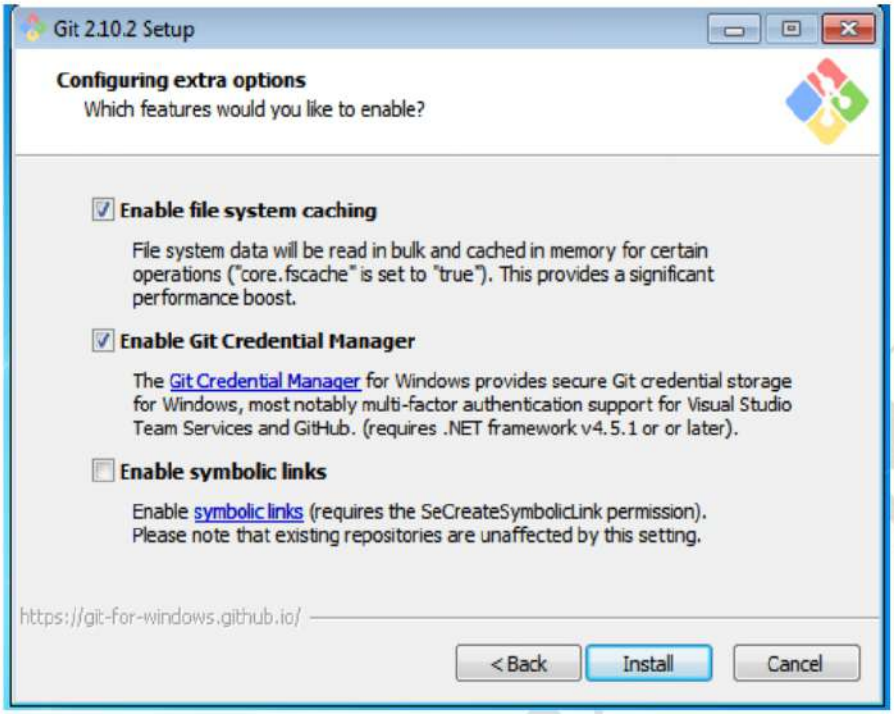


After you have made selection of your desired components, click on **Next>**.

Step 4:
The next prompt window will let you choose the adjustment of your path environment. This is where you decide how do you want to use Git.

You can select any of the three options according to your needs. But for beginners, I recommend using **Use Git from Git Bash Only**

Step 5:
The next step is to choose features for your Git. You get three options and you can choose any of them, all of them or none of them as per your needs. Let me tell you what these features are:



The first is the option to **Enable file system caching**.

Caching is enabled through Cache manager, which operates continuously while Windows is running. File data in the system file cache is written to the disk at intervals determined by the operating system, and the memory previously used by that file data is freed.

The second option is to enable **Git Credential Manager**.

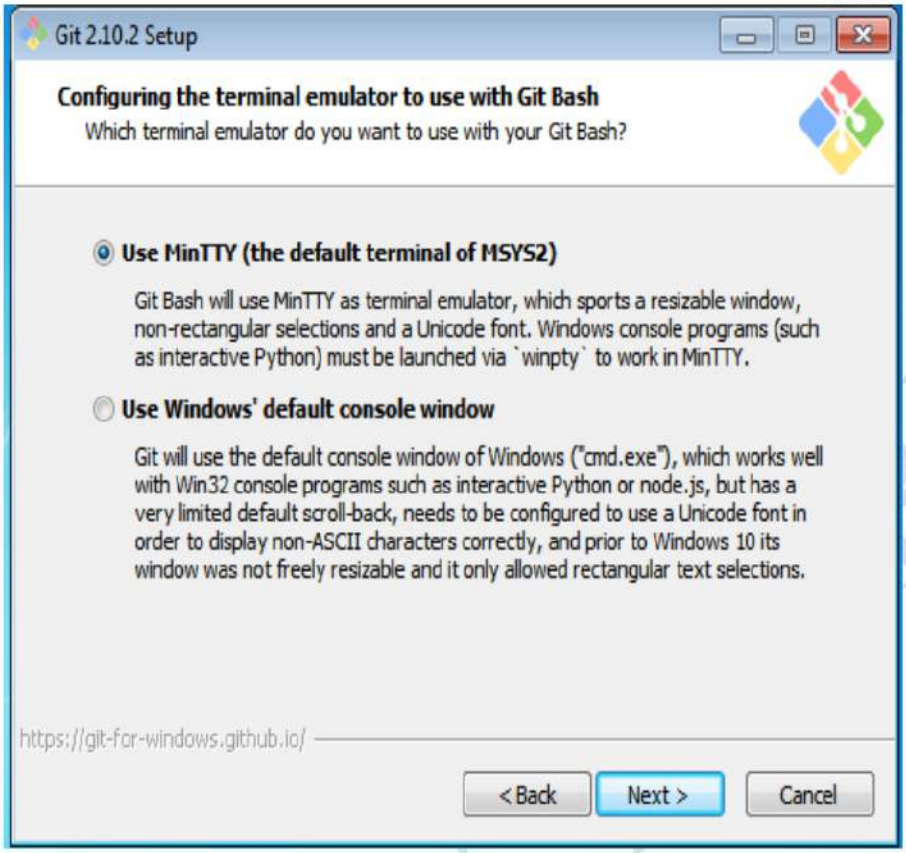
The **Git Credential Manager** for Windows (GCM) is a credential helper for Git. It securely stores your credentials in the Windows CM so that you only need to enter them once for each remote repository you access. All future Git commands will reuse the existing credentials.

The third option is to **Enable symbolic links**.

Symbolic links or symlinks are nothing but advanced shortcuts. You can create symbolic links for each individual file or folder, and these will appear like they are stored in the folder with symbolic link.

I have selected the first two features only.

Step 6:
Choose your terminal.

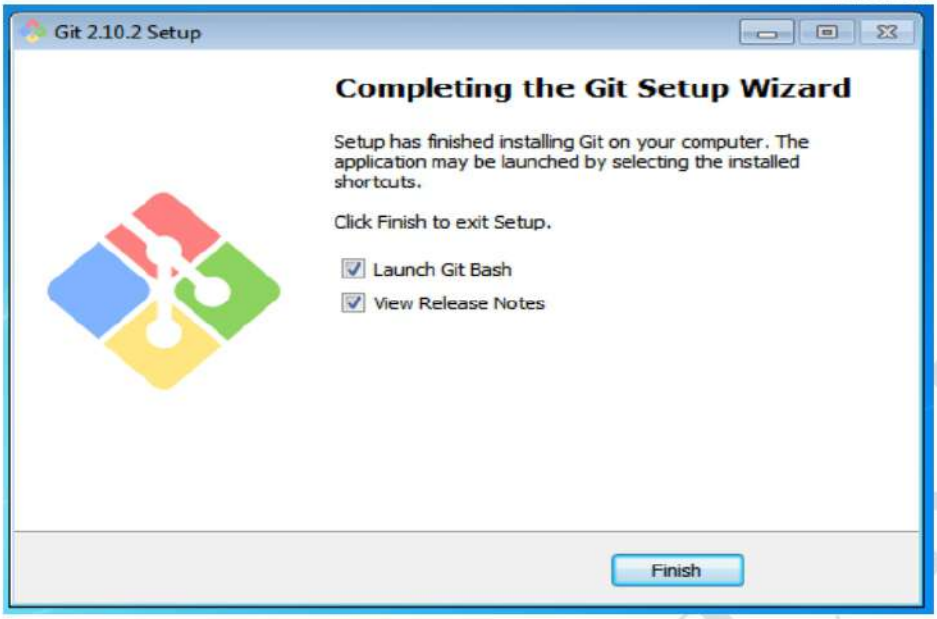


You can choose one from the options.

The default terminal of MYSYS2 which is a collection of GNU utilities like bash, make, gawk and grep to allow building of applications and programs which depend on traditionally UNIX tools to be present.

Or you can choose the window's default console window (cmd.exe).

Step 7:
Now you have got all you need. Select **Launch Git Bash** and click on **Finish**.

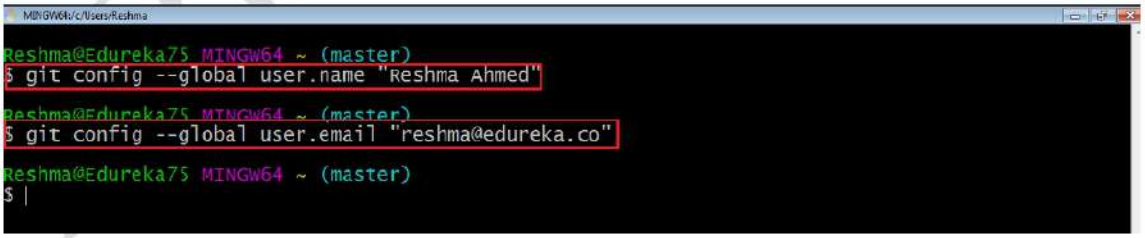


This will launch Git Bash on your screen which looks like the snapshot below:

Step 8:
Let us proceed with configuring Git with your username and email. In order to do that, type the following commands in your Git Bash:

$ git config - - global user.name "your name"

$ git config - - global user.email "your email"



It is important to configure your Git because any commits that you make are associated with your configuration details. If you want to view all your configuration details, use the command below:

$ git config - - list

ABHISHEK K L
Assist.Prof, MCA-RIT

Creating Git Folder

$ mkdir myproject
$ cd myproject

mkdir **make**s a **new directory**.

cd **changes** the **current working directory**.

Now that we are in the correct directory. We can start by initializing Git!

Note: If you already have a folder/directory you would like to use for Git: Navigate to it in command line, or open it in your file explorer, right-click and select "Git Bash here"

Basics of how Git works

First and foremost, it is important to understand the 3 areas where your code lives inside git:

i. **Working tree**
The working tree contains the files that you are currently working on. When you open the files for a project that is managed as a Git repository, you gain access to the Working Tree.

ii. **Staging area (Index)**
The staging area is like backstage in a theatre, i.e. this area contains all the added files that contain new/changed code, which is ready to be joined to the next commit. All new/changed files are first pushed to the staging area.

iii. **Repository**
Git repository contains all files of the project. It's like your project's database. It can be a local git repository or remotely present in any web hosting services like Github, Bitbucket, etc

Every file under Git goes through these Four stages:

1. **Untracked:** In this stage, the Git repository is unable to track the file, which means that the file is in working directory and it is never staged nor it is committed.

2. **Modified:** In a git repository, you can make changes to your copy of the project without hampering the original code. This is called Modification, i.e. making some additions to the original project. the file is modified but the change is not yet staged.

3. **Staged:** Staged files are modified files that are marked to go in your next commit. git add command tracks the new changes and pushes them to the staging area.

4. **Committed:** You must commit the file in order for the changes to be reflected and stored in the local database (.git directory). To make it accessible to all of your team members, use the git push command to push it to a remote central repository.

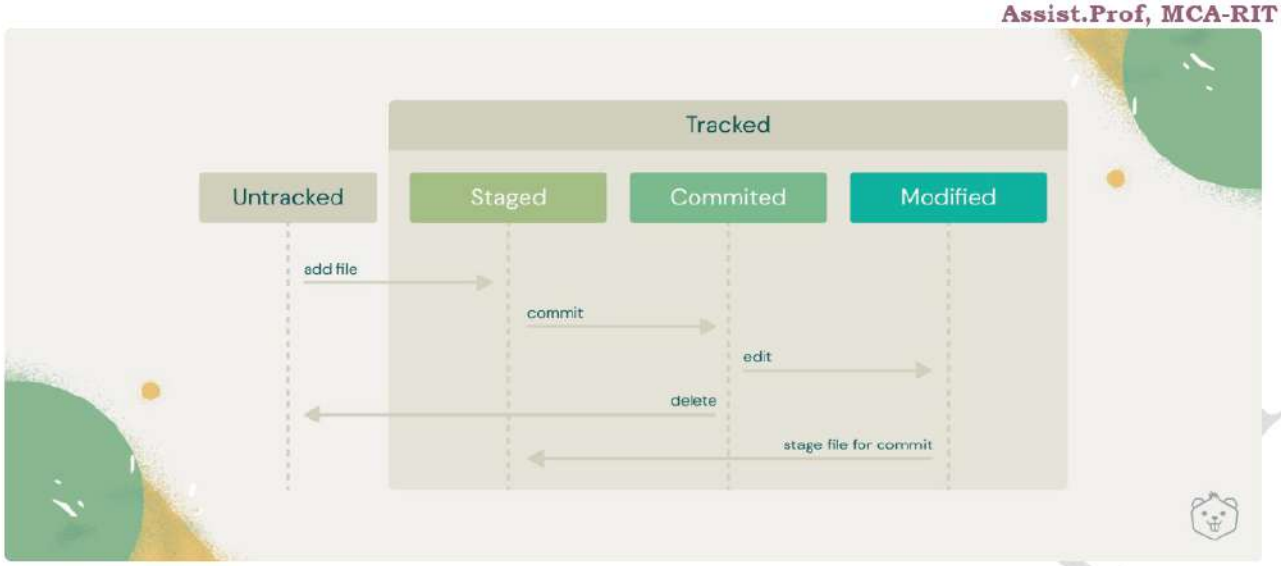


Fig: file stages under Git

Basic Git Workflow

1. When you browse and work on files in your repository, you are on a working tree, and all of your files are untracked at first.
2. The files you want to record are then staged and moved to index.
3. The staged files are then committed and saved in the local repository.
4. When you're ready to make them public, add them to a remote repository hosting service such as Github.

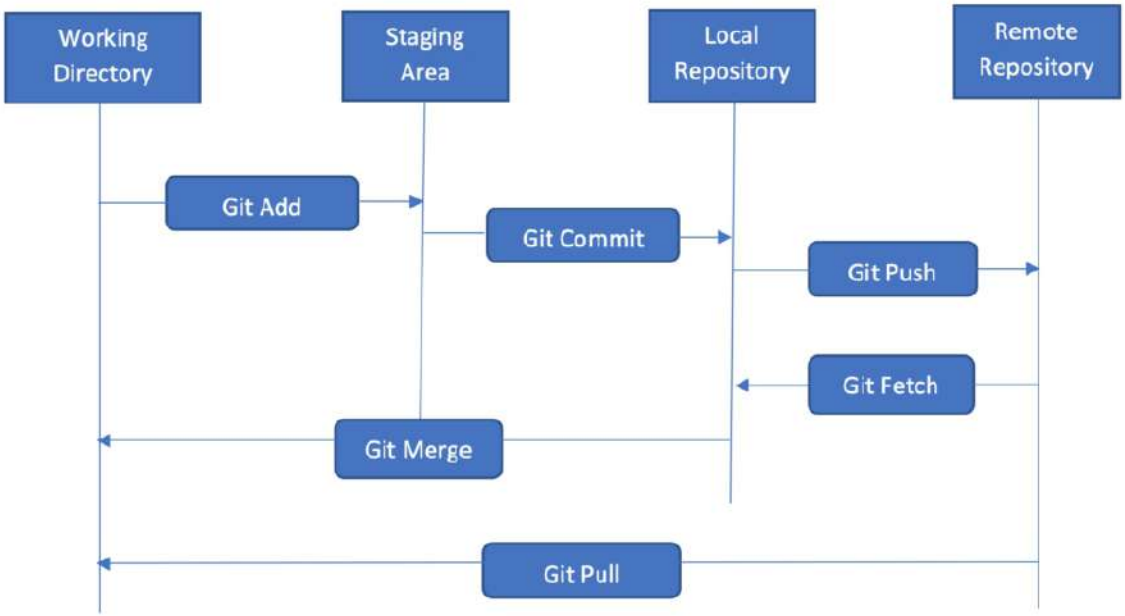


Fig: GIT Workflow Diagram

Git basic terminologies

- **git add:** A git command used to add a file from working directory to the staging area.
- **git commit:** A git command used to add all files that are staged to the local repository.
- **git push:** A git command used to add all committed files in the local repository to the remote repository.
- **git fetch:** A git command used to get files from remote repository to the local repository but not into the working directory.
- **git merge:** A git command used to get files from local repository into the working directory.
- **git pull:** A git command used to get files from remote repository directly into the working directory. It is equivalent to git fetch and git merge.

Create a Local Repository for a Blank (New) Project / Initialize Git:

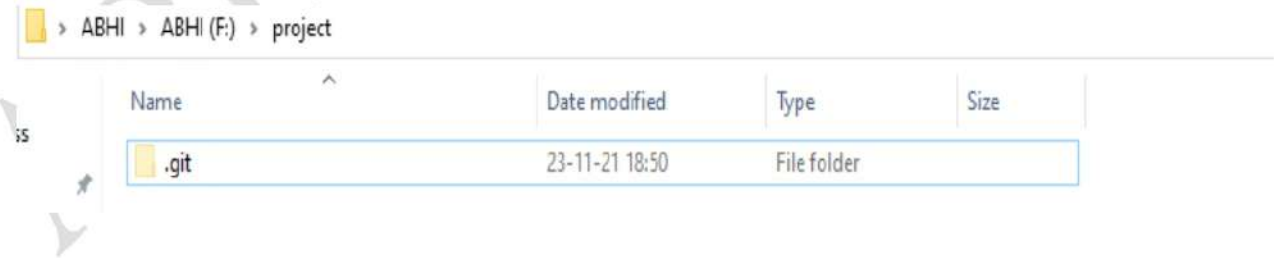
To create a blank repository, open command line on your desired directory and run the init command as follows:

$ git init

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git init
Initialized empty Git repository in C:/Users/HiMaNshU/Desktop/.git/

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

The above command will create an empty hidden **.git** repository. This subdirectory contains the metadata that Git uses to store its information.



Note: Suppose we want to make a git repository on our desktop. To do so, open Git Bash on the desktop and run the above command.

Now we can create and add files on this repository for version control.

To create a file, run the cat command as follows:

$ cat >>file Name
$ cat >>readme.txt

list the files in the current working directory: **$ ls**

Now you have readme.txt file in your current directory

Now check the status of the file using: Git status command

Let us type in the command to see what happens:

```
Admin@MCA-ABHISHEK MINGW64 /f/project (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be com
mitted)
        readme.txt

nothing added to commit but untracked files present (use
"git add" to track)
```

As we can see from the above output, the status is showing as " **added to commit but untracked files present (use "git add" to track)**". The status command also displays the suggestions. As in the above output, it is suggesting to use the add command to track the file.

Git add files

Git add command is a straight forward command. It adds files to the staging area. We can add single or multiple files at once in the staging area. It will be run as:

1. adding single file into staging area

$ git add <File name>

Ex: **$ git readme.txt**

In Unix systems the end of a line is represented with a line feed (LF). In windows a line is represented with a carriage return (CR) and a line feed (LF) thus (CRLF). when you get code from git that was uploaded from a unix system they will only have an LF.

If you are a single developer working on a windows machine, and you don't care that git automatically replaces LFs to CRLFs, you can turn this warning off by typing the following in the git command line

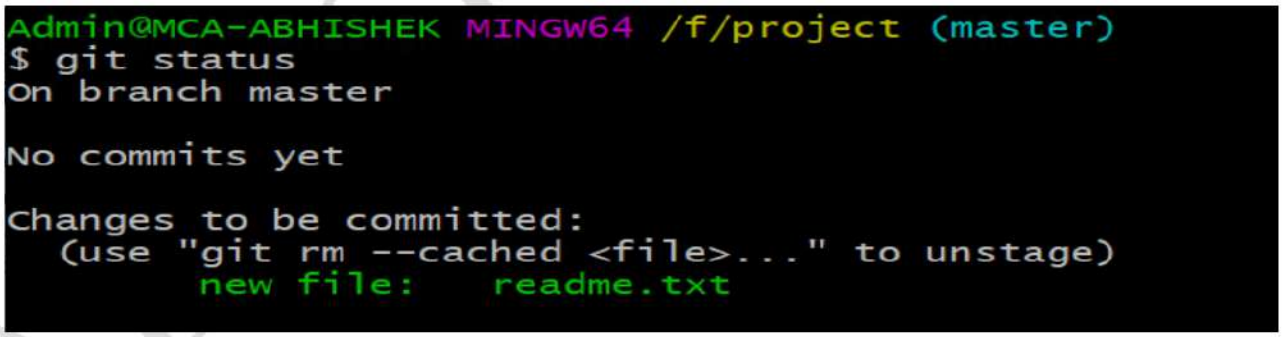
git config core.autocrlf true

2. adding multiple files at once into staging area

Git facilitates us with a unique option of the add command by which we can add all the available files at once. To add all the files from the repository, run the add command with **-A** option. We can use '.' Instead of **-A** option. This command will stage all the files at a time. It will run as follows:

$ git add -A or **$ git add .**

After adding a file into the staging area, now if you execute git status it will show



From the above output, we can see that the status after staging the file is showing as "**changes to be committed**".

Every change must be committed using $ git commit command.

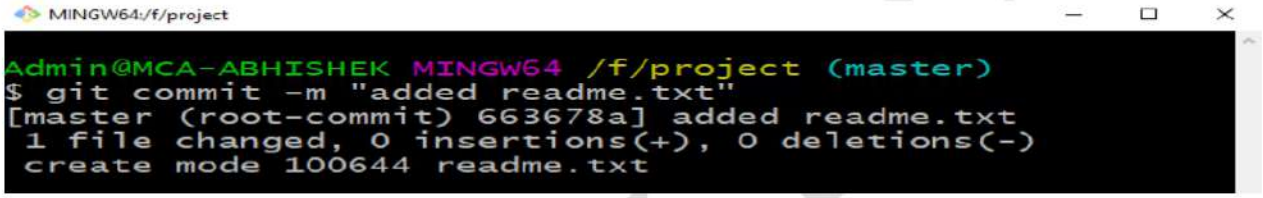
The git commit command

The commit command will commit the changes and generate a commit-id.

$ git commit –m "Commit message"

The git commit command uses '-m' as a flag for a message to set the commits with the content where the full description is included, and a message is written in an imperative sentence up to 50 characters long and defining **"what was changed", and "why was the change made".**

After Executing git commit command the output will look like this:



Commit without argument

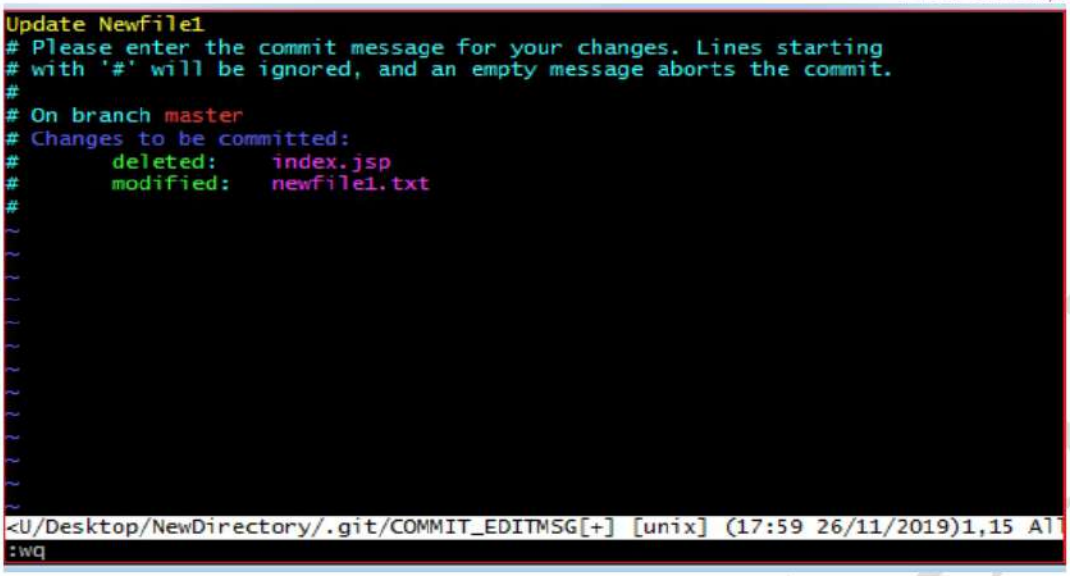
The commit command without any argument will open the default text editor and ask for the commit message. We can specify our commit message in this text editor. It will run as follows:

$ git commit

The above command will prompt a default editor and ask for a commit message. We have made a change to **readme.txt** and want it to commit it. It can be done as follows:

Consider the below output:

As we run the command, it will prompt a default text editor and ask for a commit message. The text editor will look like as follows:



Press the **Esc** key and after that '**I**' for insert mode. Type a commit message whatever you want. Press **Esc** after that '**:wq**' to save and exit from the editor. Hence, we have successfully made a commit.

Adding a file into staging area and committing at once for already tracked file

Git commit –a –m "changed abhi.txt file"

Git Clone

In Git, cloning is the act of making a copy of any target repository. Also, the git clone is a command- which is used to make a local copy of a remote repository. It accesses the repository through a remote URL.

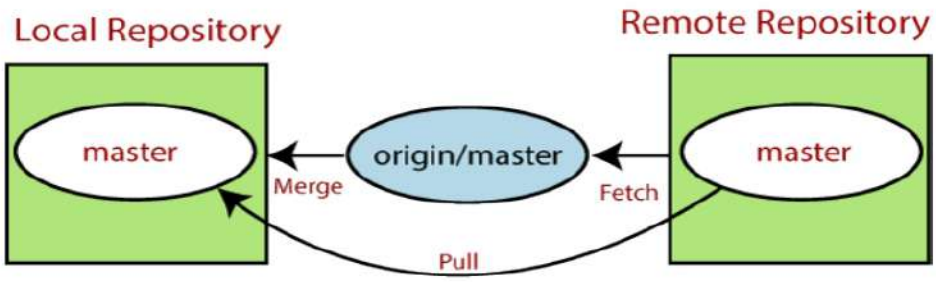
Usually, the original repository is located on a remote server, often from a Git service like GitHub, Bitbucket. The remote repository URL is referred to the origin. you can sync between the two locations.

Syntax: **$ git clone <repository URL>**

It clones the remote repository including **.git** into the specified folder.

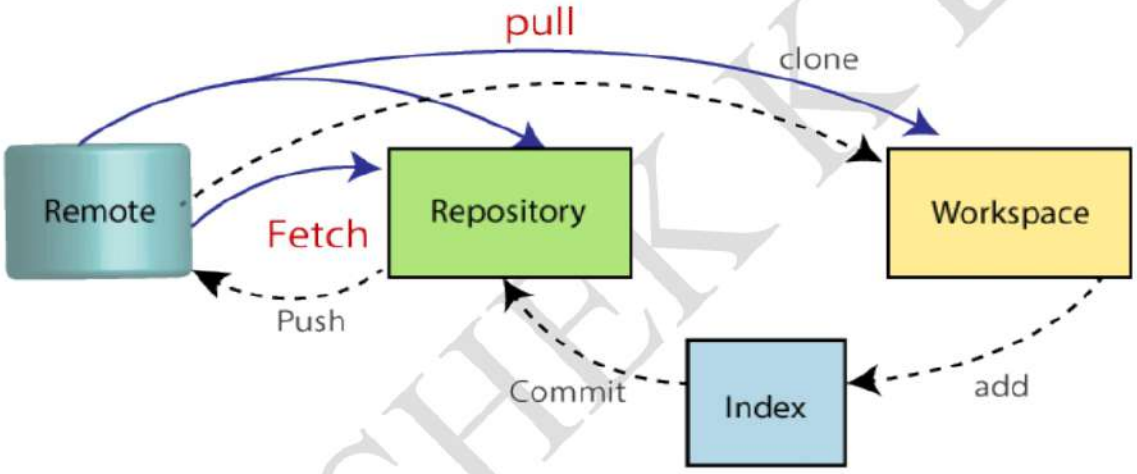
Git Pull / Pull Request

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The git pull command is used to pull a repository.



Pull request is a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.

The below figure demonstrates how pull acts between different locations and how it is similar or dissimilar to other related commands.



The "git pull" command

The pull command is used to access the changes (commits)from a remote repository to the local repository. It updates the local branches with the remote-tracking branches. Remote tracking branches are branches that have been set up to push and pull from the remote repository. Generally, it is a collection of the fetch and merges command. First, it fetches the changes from remote and combined them with the local repository.

Before pulling or fetching data from remote repository first you need to add remote repository link using **git remote add** command as shown below:

$ git remote add origin "URL of Repository"

Origin is the default name for the remote server, which is given by Git.

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git remote add origin "https://github.com/klabhishek1992/Demo.git"
```

Now use git pull command.

Syntax: **$ git pull origin <branch name in RemoteRepo>**

Ex: $ git pull origin master

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git pull origin master
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 17 (delta 0), reused 15 (delta 0), pack-reused 0
Unpacking objects: 100% (17/17), 2.86 KiB | 6.00 KiB/s, done.
From https://github.com/klabhishek1992/Demo
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> origin/master
```

Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.

We can fetch the complete repository with the help of fetch command from a repository URL.

Fetch + merge=Pull

Before fetching data from remote repository first you need to add remote repository link using **git remote add** command as shown below:

$ git remote add origin "URL of Repository"

Now simply use **$ git fetch** command as shown below:

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git fetch
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 17 (delta 0), reused 15 (delta 0), pack-reused 0
Unpacking objects: 100% (17/17), 2.86 KiB | 6.00 KiB/s, done.
From https://github.com/klabhishek1992/Demo
 * [new branch]      master     -> origin/master
```

Now to see changes in your Working directory use git merge command as shown below:

$ git merge origin/master

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git merge origin/master
```

ABHISHEK K L
Assist.Prof, MCA-RIT

Git Branch

What is a Branch in Git?

Branch in Git is similar to the branch of a tree. Analogically, a tree branch is attached to the central part of the tree called the trunk. While branches can generate and fall off, the trunk remains compact and is the only part by which we can say the tree is alive and standing. Similarly, a branch in Git is a way to keep developing and coding a new feature or modification to the software and still not affecting the main part of the project. We can also say that branches create another line of development in the project.

Git Master Branch

The primary or default branch in Git is the master branch (similar to a trunk of the tree). It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Note: Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

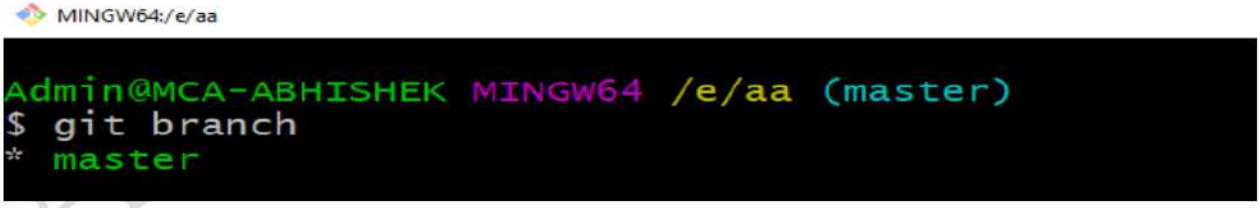
Operations on Branches

1. **Create a Branch**
2. **Checkout a branch**
3. **Merge a Branch**
4. **Delete a Branch**

1. **Create a Branch**
   This is the first step in the process, you can start on a default branch or create a new branch for the development.

   Open ***Git Bash*** and navigate to the local working repository. Type the following command to view all of your branches.

   $ git branch



Now here there is only one branch i.e master branch which is a default branch.

You can create a new branch with the help of the git branch command. This command will be used as:

Syntax: $ git branch <branch name>

Ex: **$ git branch firstbranch**

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git branch firstbranch
```

This command will create the branch **firstbranch** locally in Git directory.

MINGW64:/e/aa

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git branch
  firstbranch
* master
```

Note: **The creation of branch in the local working directory is now complete. Notice the "*" in front of the "*master*" branch. The star means the current branch on which we are currently on.**

2. Checkout a branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the git checkout command. To switch between the branches, below command is used:

$ git checkout <branch name>

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git checkout firstbranch
Switched to branch 'firstbranch'
```

After executing the above command, you will be in firstbranch as shown below:

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (firstbranch)
$
```

Now if you use the **$ git branch,** gives the below output:

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (firstbranch)
$ git branch
* firstbranch
  master
```

Notice the "*" in front of the "*firstbranch*" branch, which means that currently you are in firstbranch.

3. Merge a Branch

Merge A Branch: An already running branch can merge with any other branch in your Git repository. Merging a branch can help when you are done with the branch and want the code to integrate into another branch code. You can merge two branches with the help of git merge command

$ git merge <branch name>

Before merging you must checkout to Destination branch (To which branch you are merging) For Ex: If you want to merge firstbranch with the master branch then you first checkout to master branch as shown below:

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (firstbranch)
$ git checkout master
Switched to branch 'master'

Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
```

Now use the below command to merge the branches:

$ git merge firstbranch

```
Admin@MCA-ABHISHEK MINGW64 /e/aa (master)
$ git merge firstbranch
Updating 74d838f..8524e4b
Fast-forward
 abhi.txt  | 0
 abhi1.txt | 1 +
 2 files changed, 1 insertion(+)
 delete mode 100644 abhi.txt
 create mode 100644 abhi1.txt
```

Rename a local Git branch

A local Git branch exists only on your computer. You make changes and tests here without other developers noticing. Renaming it can therefore be done quickly.

1. In the command line, select the Git branch you want to rename. The command for this is "**git checkout old-name**".
2. You will get a confirmation that you have selected the correct branch. This will read **"Switched to branch 'old-name'"**.
3. Now perform the actual rename for the local Git branch. The appropriate command for this is: **"git branch -m new-name".**

Alternatively, you have the option to rename the Git branch via the master. To do this, use the following steps:

1. Switch to the master via the command **"git checkout master"**.

2. Now enter the following command if you want to rename a Git branch: "**git branch -m old-name new-name**".
3. To ensure that the rename was successful, retrieve the current status of the branch using the "**git branch -a**" command.

4. Delete a Branch

An already running branch can delete from your Git repository. Deleting a branch can help when the branch has done its job, i.e., it's already merged, or you no longer need it in your repository for any reason.

Deleting a branch LOCALLY

Git will not let you delete the branch you are currently on so you must make sure to checkout a branch that you are NOT deleting.

For example: **$ git checkout master**

Delete a branch using the command:
$ git branch -d <branchname>.

For example: **$ git branch -d firstbranch**

The -d option will delete the branch only if it has already been pushed and merged with the remote branch.
Use -D instead if you want to force the branch to be deleted, even if it hasn't been pushed or merged yet.

Deleting a branch REMOTELY

Here's the command to delete a branch remotely:

$ git push <remote> --delete <branch>.

For example: **$ git push origin --delete firstbranch**

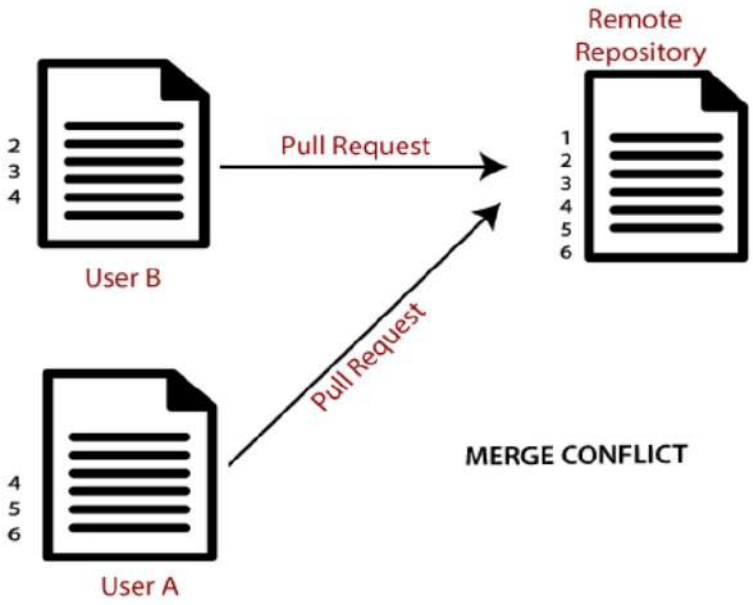
The branch is now deleted remotely.

Git vs SVN

Git	SVN
It's a distributed version control system.	It's a Centralized version control system
Git is an SCM (source code management).	SVN is revision control.
Git has a cloned repository.	SVN does not have a cloned repository.
The Git branches are familiar to work. The Git system helps in merging the files quickly and also assist in finding the unmerged ones.	The SVN branches are a folder which exists in the repository. Some special commands are required For merging the branches.
Git does not have a Global revision number.	SVN has a Global revision number.
Git has cryptographically hashed contents that protect the contents from repository corruption taking place due to network issues or disk failures.	SVN does not have any cryptographically hashed contents.
Git stored content as metadata.	SVN stores content as files.
Git has more content protection than SVN.	SVN's content is less secure than Git.
Linus Torvalds developed git for Linux kernel.	CollabNet, Inc developed SVN.
Git is distributed under GNU (General public license).	SVN is distributed under the open-source license.

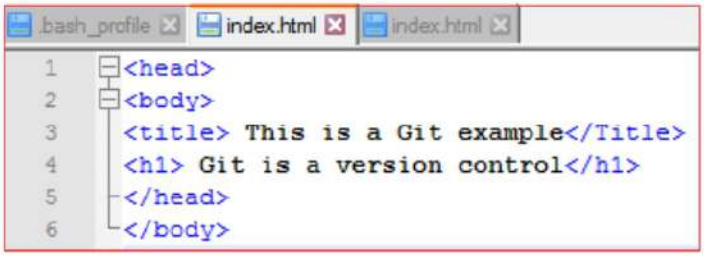
Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



Let's understand it by an example.

Suppose remote repository has cloned by two of my team member **user1 and user2**. The **user1** made changes as below in my projects index file.



Update it in the local repository with the help of git add command.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git add index.html
```

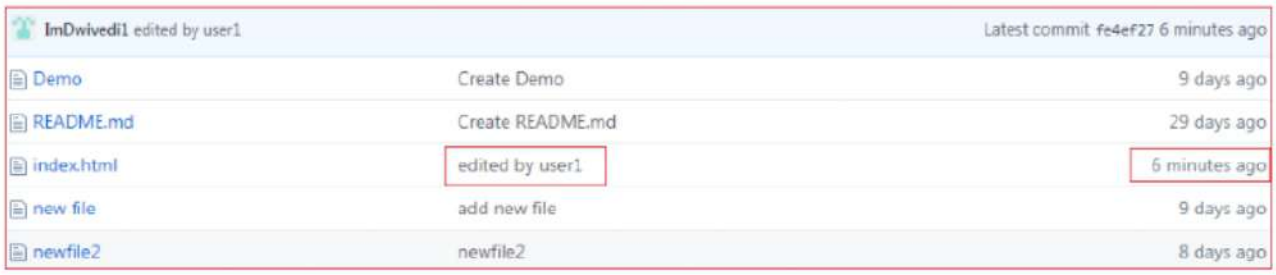
Now commit the changes and update it with the remote repository. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git commit -m "edited by user1"
[master fe4ef27] edited by user1
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 345.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/Git-Example
   039c01b..fe4ef27  master -> master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
```

Now, my remote repository will look like this:



It will show the status of the file like edited by whom and when.

Now, at the same time, **user2** also update the index file as follows.

bash_profile | index.html | index.html

```
1  <head>
2  <body>
3   <title> This is a Git example</Title>
4   <h2> Git is a version control system</h2>
5  </head>
6  </body>
```

User2 has added and committed the changes in the local repository. But when he tries to push it to remote server, it will throw errors. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git add index.html

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git commit -m " edited by user2"
[master 3ee71e0]  edited by user2
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git push origin master
To https://github.com/ImDwivedi1/Git-Example
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ImDwivedi1/Git-Example'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$
```

In the above output, the server knows that the file is already updated and not merged with other branches. So, the push request was rejected by the remote server. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**. It will suggest you to pull the repository first before the push. See the below command:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git pull --rebase origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            master     -> FETCH_HEAD
   039c01b..fe4ef27  master     -> origin/master
First, rewinding head to replay your work on top of it...
Applying:  edited by user2
Using index info to reconstruct a base tree...
M       index.html
Falling back to patching base and 3-way merge...
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: Failed to merge in the changes.
hint: Use 'git am --show-current-patch' to see the failed patch
Patch failed at 0001  edited by user2
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort"
.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master|REBASE 1/1)
$
```

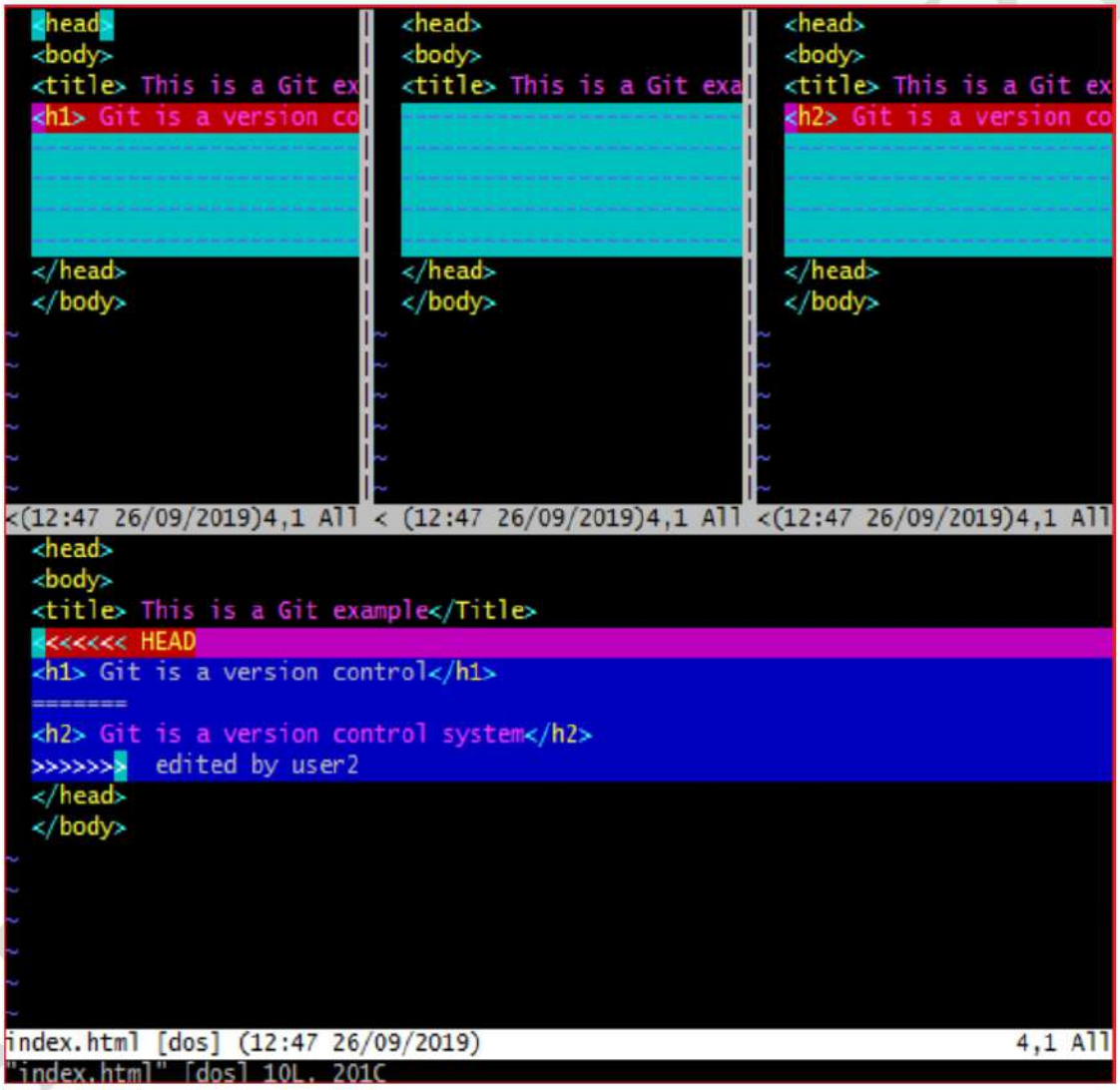
In the given output, git rebase command is used to pull the repository from the remote URL. Here, it will show the error message like **merge conflict in <filename>**.

Resolve Conflict:

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

$ git mergetool

In my repository, it will result in:



The above output shows the status of the conflicted file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the rebase command. It will be used as follows:

$ git rebase --continue

Hence, the conflict has resolved. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master|REBASE 1/1)
$ git rebase --continue
Applying:  edited by user2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 373 bytes | 124.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ImDwivedi1/Git-Example
   fe4ef27..b3db7dc  master -> master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user2repo (master)
$
```

In the above output, the conflict has resolved, and the local repository is synchronized with a remote repository.

To see that which is the first edited text of the merge conflict in your file, search the file attached with conflict marker **<<<<<<<**. You can see the changes from the **HEAD** or base branch after the line **<<<<<<< HEAD** in your text editor. Next, you can see the divider like **=======**. It divides your changes from the changes in the other branch, **followed by >>>>>>> BRANCH-NAME**. In the above example, user1 wrote "<h1> Git is a version control</h1>" in the base or HEAD branch and user2 wrote "<h2> Git is a version control</h2>".

Decide whether you want to keep only your branch's changes or the other branch's changes, or create a new change. Delete the conflict markers **<<<<<<<, =======, >>>>>>>** and create final changes you want to merge.

Git merge conflicts resolving commands

1. **git pull --rebase origin master**
2. **git mergetool**
3. **git rebase --continue**

MODULE-V

Containerization with Docker

What is Docker?

Docker is an open-source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

Docker is an OS virtualized software platform that allows IT organizations to easily create, deploy, and run applications in Docker containers, which have all the dependencies within them.

Features of Docker:

1. **Faster and Easier configuration:**
   It is one of the key features of Docker that helps you in configuring the system in a faster and easier manner. Due to this feature, codes can be deployed in less time and with fewer efforts. The infrastructure is not linked with the environment of the application as Docker is used with a wide variety of environments.

2. **Increase in productivity:**
   It helps in increasing productivity by easing up the technical configuration and rapidly deploying applications. Moreover, it not only provides an isolated environment to execute applications, but it reduces the resources as well.

3. **Application isolation:**
   Docker provides containers that are used to run applications in an isolated environment. Since each container is independent, Docker can execute any kind of application.

4. **Swarm:**
   Swarm is a clustering and scheduling tool for Docker containers. At the front end, it uses the Docker API, which helps us to use various tools to control it. It is a self-organizing group of engines that enables pluggable backends.

5. **Services:**
   Services is a list of tasks that specifies the state of a container inside a cluster. Each task in the Services lists one instance of a container that should be running, while Swarm schedules them across the nodes.

6. **Routing Mesh**
   It routes the incoming requests for published ports on available nodes to an active container. This feature enables the connection even if there is no task is running on the node.

Why we need Docker

1. **Portability**
   Once you have tested your containerized application you can deploy it to any other system where Docker is running and you can be sure that your application will perform exactly as it did when you tested it.

2. **Performance**
   Although virtual machines are an alternative to containers, the fact that containers do not contain an operating system (whereas virtual machines do) means that containers have much smaller footprints than virtual machines, are faster to create, and quicker to start.

3. **Scalable and Cost effective**
   Running your service on hardware that is much cheaper than standard servers. You can quickly create new containers if demand for your applications requires them. When using multiple containers, you can take advantage of a range of container management options.

What is Virtualization?

Virtualization is the technique of importing a Guest operating system on top of a Host operating system. This technique was a revelation at the beginning because it allowed developers to run multiple operating systems in different virtual machines all running on the same host. This eliminated the need for extra hardware resource.

The advantages of Virtual Machines or Virtualization are:

1. **Multiple operating systems can run on the same machine.**
2. **Maintenance and Recovery were easy in case of failure conditions.**
3. **Total cost of ownership was also less due to the reduced need for infrastructure.**

In the diagram, you can see there is a host operating system on which there are 3 guest operating systems running which is nothing but the virtual machines.

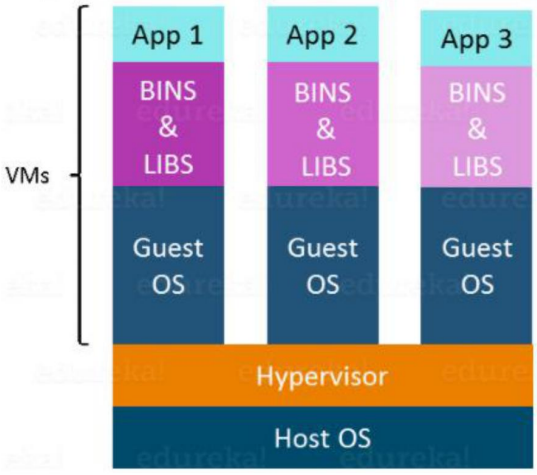


Fig: Virtualization

Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will

have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.

Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

- Running multiple Virtual Machines leads to unstable performance.
- Hypervisors are not as efficient as the host operating system.
- Boot up process is long and takes time.
- These drawbacks led to the emergence of a new technique called Containerization. Now let me tell you about Containerization.

What is Containerization?

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware, Containerization brings abstraction to the operating system. Do note that Containerization is also a type of Virtualization.

Containerization is however more efficient because there is no guest OS here and utilizes a host's operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

Advantages of Containerization over Virtualization:

- Containers on the same OS kernel are lighter and smaller
- Better resource utilization compared to VMs
- Boot-up process is short and takes few seconds

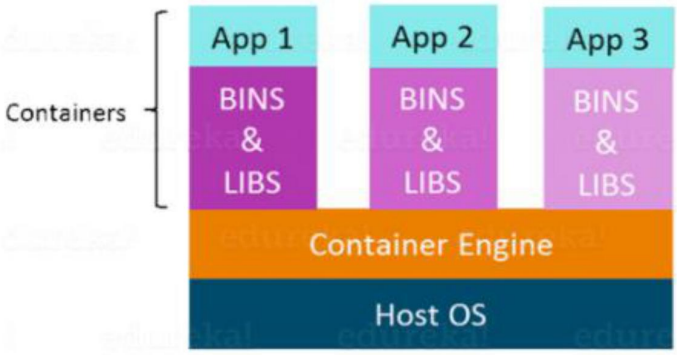
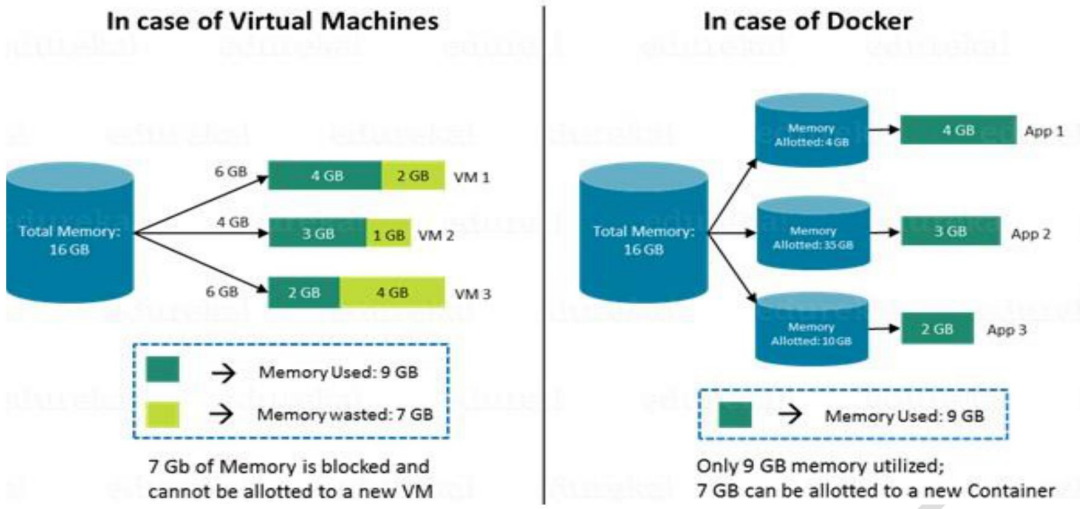


Fig: Containerization

In the diagram on the right, you can see that there is a host operating system which is shared by all the containers. Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

All these containers are handled by the containerization layer which is not native to the host operating system. Hence a software is needed, which can enable you to create & run containers on your host operating system.

Docker vs Virtualization



Virtualization	Containerization
Virtualizes hardware resources	Virtualizes only OS resources
Takes more time for Booting	Takes less time for Booting
Each VM runs in its own OS.	All containers share the host OS
Requires more Memory	Requires less memory
Requires the complete OS installation for every VM	Installs the container only on a host OS
Heavyweight	Lightweight
It is not portable	It is very portable. We can build, ship and run anywhere
Once the memory is allocated for virtual machine, the unused memory can't be shared/reallocate which leads to Wastage of Memory	The unused memory can be shared across other Containers.
Use Hardware level virtualization	Uses OS Virtualization
Limited performance	Native performance
Fully isolated	Process-level isolation
More Secure	Less Secure
Ex: Docker, LXC, LXCD , CG Manager	VMware, vSphere, Virtual Box, Hyper-V

Docker Architecture

Docker uses a client-server architecture. The Docker client consists of Docker build, Docker pull, and Docker run. The client approaches the Docker daemon that further helps in building, running, and distributing Docker containers. Docker client and Docker daemon can be operated on the same system; otherwise, we can connect the Docker client to the remote Docker daemon. Both communicate with each other by using the REST API, over UNIX sockets or a network.

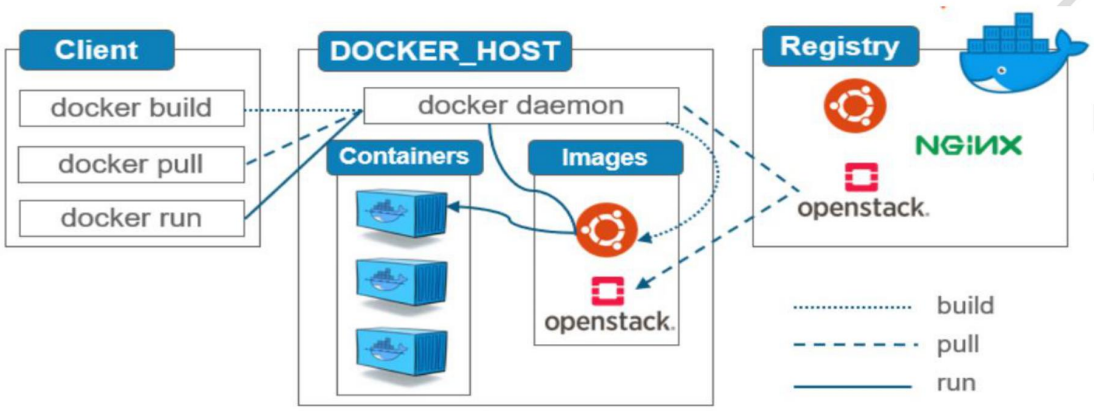


Fig: Docker Architecture

Docker Engine

It is the core part of the whole Docker system. Docker Engine is an application which follows client-server architecture. It is installed on the host machine. There are three components in the Docker Engine:

Server: It is the docker daemon called dockerd. The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker daemon pulls and builds container images as requested by the client. Once it pulls a requested image, it builds a working model for the container by utilizing a set of instructions known as a build file. The build file can also include instructions for the daemon to pre-load other components prior to running the container, or instructions to be sent to the local command line once the container is built.

Rest API: An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client. It is used to instruct docker daemon what to do.

Command Line Interface (CLI): It is a client which is used to enter docker commands.

Docker Client

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

The Docker client can reside on the same host as the daemon or connect to a daemon on a remote host. A docker client can communicate with more than one daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Docker Client uses Command Line Interface (CLI) to run the following commands -**docker build, docker pull and docker run.**

Docker host

The Docker host provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API.

Docker Registry

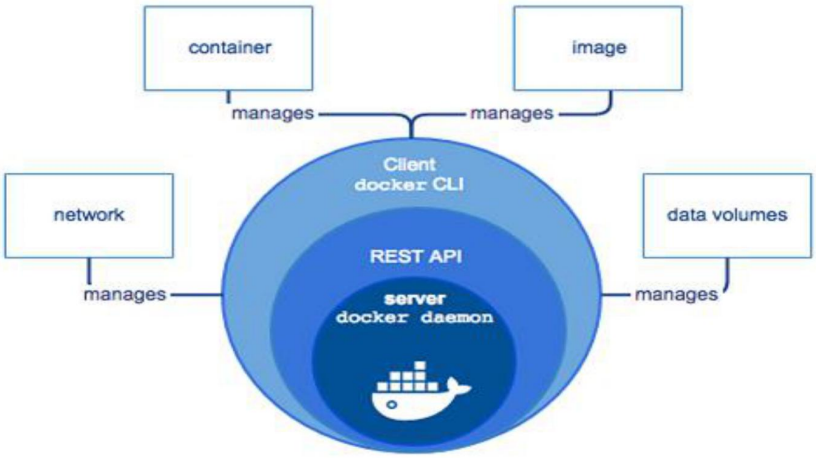
Docker registry is a repository which manages and stores the Docker images that are used for creating Docker containers. There are two types of registries in the Docker -
Pubic Registry - Public Registry is also called as Docker hub.
Private Registry - It is used to share images within the enterprise.

The Registry can be either a user's local repository or a public repository like a Docker Hub allowing multiple users to collaborate in building an application. Even with multiple teams within the same organization can exchange or share containers by uploading them to the Docker Hub, which is a cloud repository similar to GitHub.

Components Docker

The four key components that make up the entire Docker architecture are:

1. The Docker Daemon or the server.
2. The Docker Command Line Interface or the client
3. Docker Registries
4. Docker Objects –
    i. Images
    ii. Containers
    iii. Network
    iv. Storage



1. Docker daemon/Server

The docker daemon called dockerd. The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
Rest API: An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client. It is used to instruct docker daemon what to do.

2. Docker Client

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Docker Client uses **Command Line Interface (CLI)** to run the following commands -**docker build, docker pull and docker run.**

3. Docker Registry

Docker registry is a repository which manages and stores the Docker images that are used for creating Docker containers. There are two types of registries in the Docker -
Pubic Registry - Public Registry is also called as Docker hub.
Private Registry - It is used to share images within the enterprise.

You can even create your private repository inside Dockerhub and store your custom Docker images using the Docker push command. Docker allows you to create your own private Docker registry in your local machine using an image called 'registry'. Once you run a container associated with the registry image, you can use the Docker push command to push images to this private registry.

4. Docker Objects

When you are working with Docker, you use images, containers, volumes, networks; all these are Docker objects.

i.Docker Images

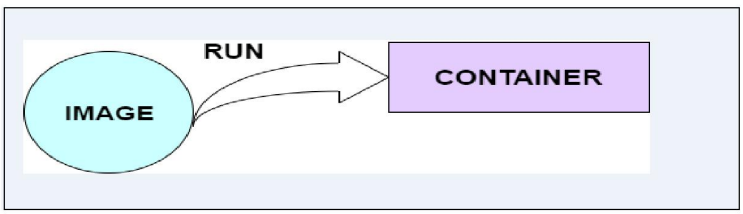
An image is a read-only template with instructions for creating a Docker container. Docker image can be pulled from a Docker hub and used as it is, or you can add additional instructions to the base image and create a new and modified docker image. You can create your own docker images also using a dockerfile. Create a dockerfile with all the instructions to create a container and run it; it will create your custom docker image.

It uses a private container registry to share container images within the enterprise and also uses public container registry to share container images within the whole world.

ii.Docker Containers

In simple terms, an image is a template, and a container is a copy of that template. You can have multiple containers (copies) of the same image.

Let's understand about containers in a different way that, if an image is a class, then a container is an instance of a class—a runtime object. Or you can say they are running instances of the Images and they hold the entire package and dependencies needed to run the application.



The container is also inherently portable. Another benefit is that it runs completely in isolation. Even if you are running a container, it's guaranteed not to be impacted by any host OS securities or unique setups, unlike with a virtual machine or a non-containerized environment. The memory for a Docker environment can be shared across multiple containers, which is really useful, especially when you have a virtual machine that has a defined amount of memory for each environment.

By using the Docker API or CLI, it is possible to ship, create, start, stop or delete a container.

iii.Networks

Docker networking is a passage through which all the isolated container communicates and share data or information. There are mainly five network drivers in docker: **Bridge Driver, Host Driver, Overlay Driver, Macvlan and None.**

iv.Storage

As soon as you exit a container, all your progress and data inside the container are lost. To avoid this, you need a solution for persistent storage. Docker provides several options for persistent storage using which you can share, store, and backup your valuable data. These are - **Data Volumes, Volume Container, Directory Mounts and Storage Plugins.**

APPENDIX

Docker Networks

Docker networking is a passage through which all the isolated container communicate. There are mainly five network drivers in docker:

1. **Bridge:** It is the default network driver for a container. You use this network when your application is running on standalone containers, i.e. multiple containers communicating with the same docker host.

2. **Host:** This driver removes the network isolation between docker containers and docker host. You can use it when you don't need any network isolation between host and container.

3. **Overlay:** This network enables swarm services to communicate with each other. You use it when you want the containers to run on different Docker hosts or when you want to form swarm services by multiple applications.

4. **None:** This driver disables all the networking.

5. **Macvlan:** This driver assigns mac address to containers to make them look like physical devices. It routes the traffic between containers through their mac addresses. You use this network when you want the containers to look like a physical device, for example, while migrating a VM setup.

Docker Storage

You can store data within the writable layer of a container but it requires a storage driver. Being non-persistent, it perishes whenever the container is not running. Moreover, it is not easy to transfer this data. With respect to persistent storage, Docker offers four options:

1. **Data Volumes:** They provide the ability to create persistent storage, with the ability to rename volumes, list volumes, and also list the container that is associated with the volume. Data Volumes are placed on the host file system, outside the containers copy on write mechanism and are fairly efficient.

2. **Volume Container:** It is an alternative approach wherein a dedicated container hosts a volume and to mount that volume to other containers. In this case, the volume container is independent of the application container and therefore you can share it across more than one container.

3. **Directory Mounts:** Another option is to mount a host's local directory into a container. In the previously mentioned cases, the volumes would have to be within the Docker volumes folder, whereas when it comes to Directory Mounts any directory on the Host machine can be used as a source for the volume.

4. **Storage Plugins:** Storage Plugins provide the ability to connect to external storage platforms. These plugins map storage from the host to an external source like a storage array or an appliance. You can see a list of storage plugins on Docker's Plugin page.