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.

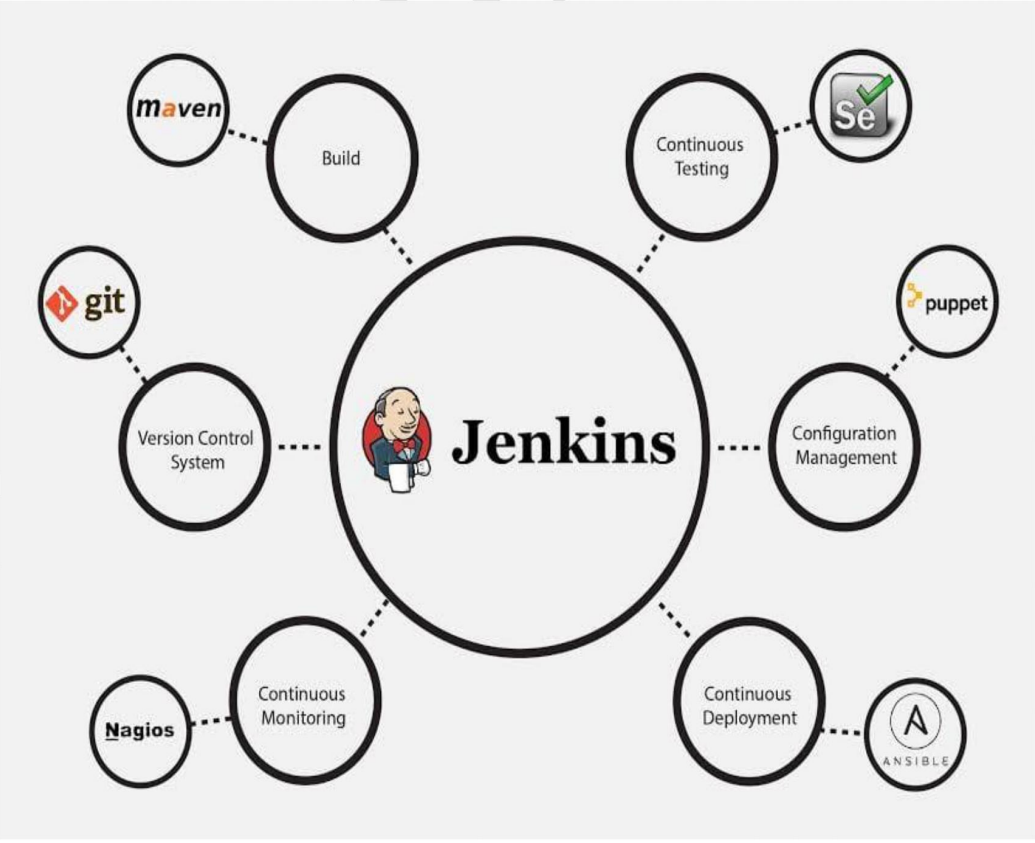
4. **Extensible**
   Jenkins can be extended by means of its plugin architecture, providing nearly endless possibilities for what it can do.

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.

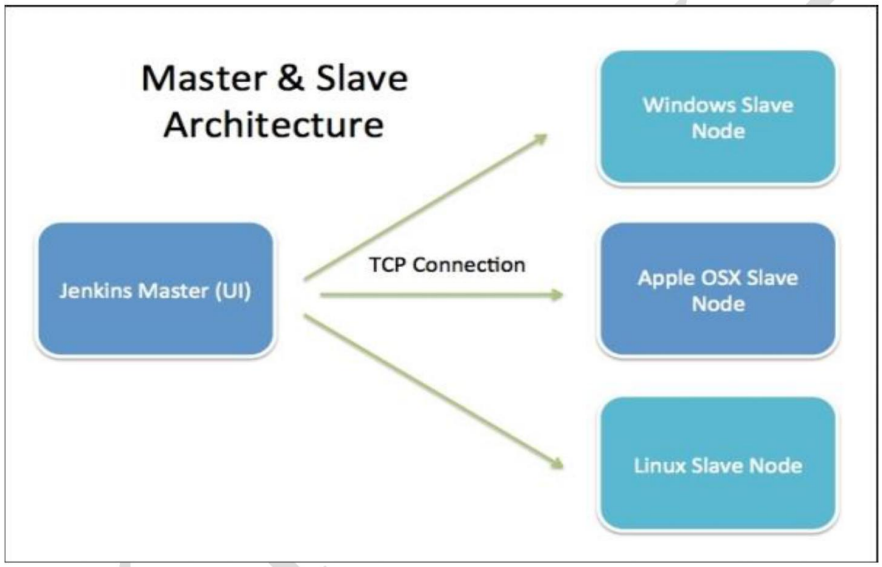
Jenkins Architecture

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.

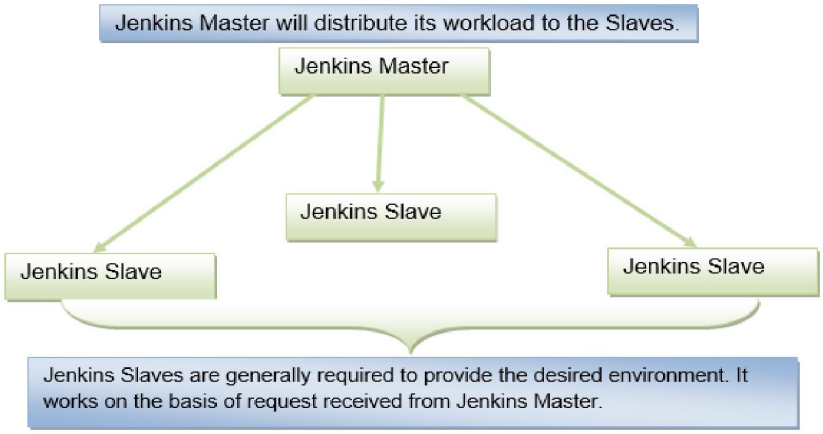
The server's job or master's job is to handle:

- 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

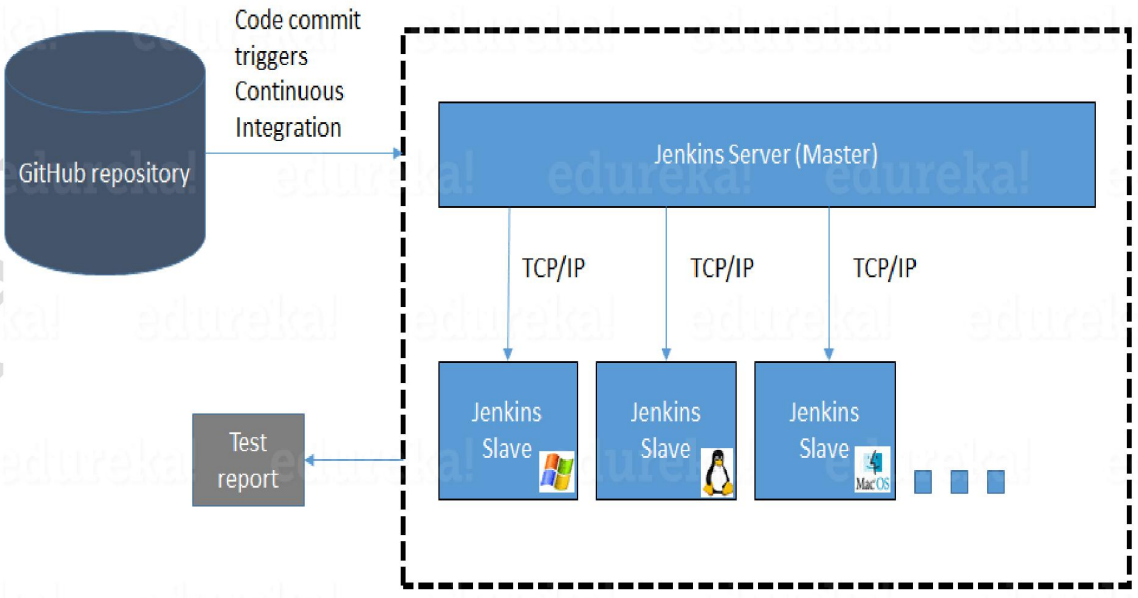
A slave is just a device that is configured to act as an executor on behalf of the master. A Slave is a Java executable that runs on a remote machine. Following are the characteristics of Jenkins Slaves:

- 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.

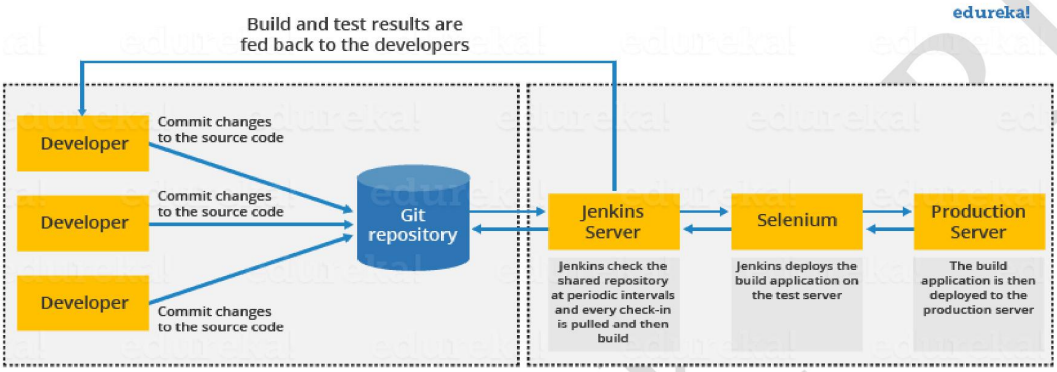
The diagram below represents the same:



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

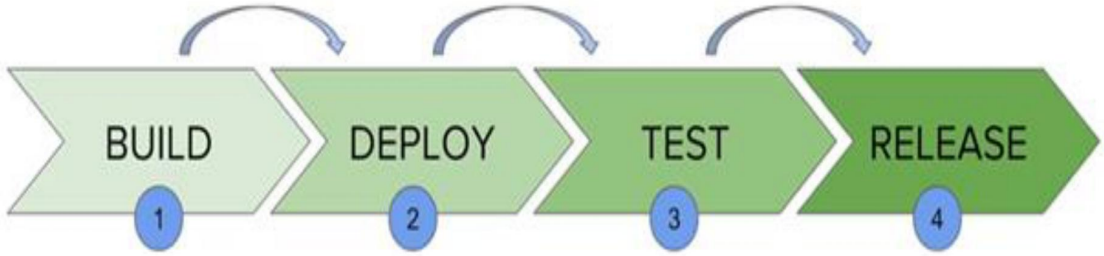
In Jenkins, a pipeline is a collection of events or jobs which are interlinked with one another in a sequence. It is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins.

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.

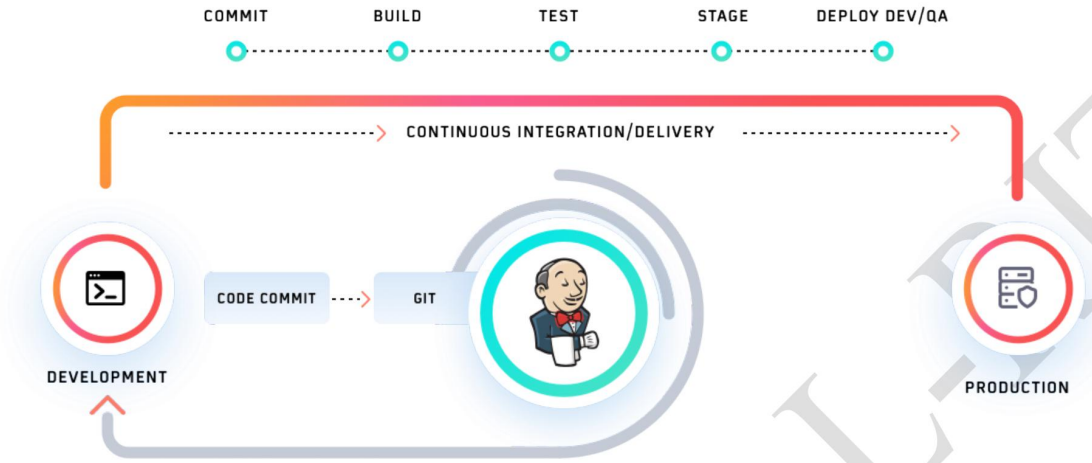
- In simple words, continuous delivery is the capability to release software at all times. It is a practice which ensures that the software is always in a production-ready state.

- 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'
        }
    }
}
```