Welcome to Jenkins

Introduction to Jenkins

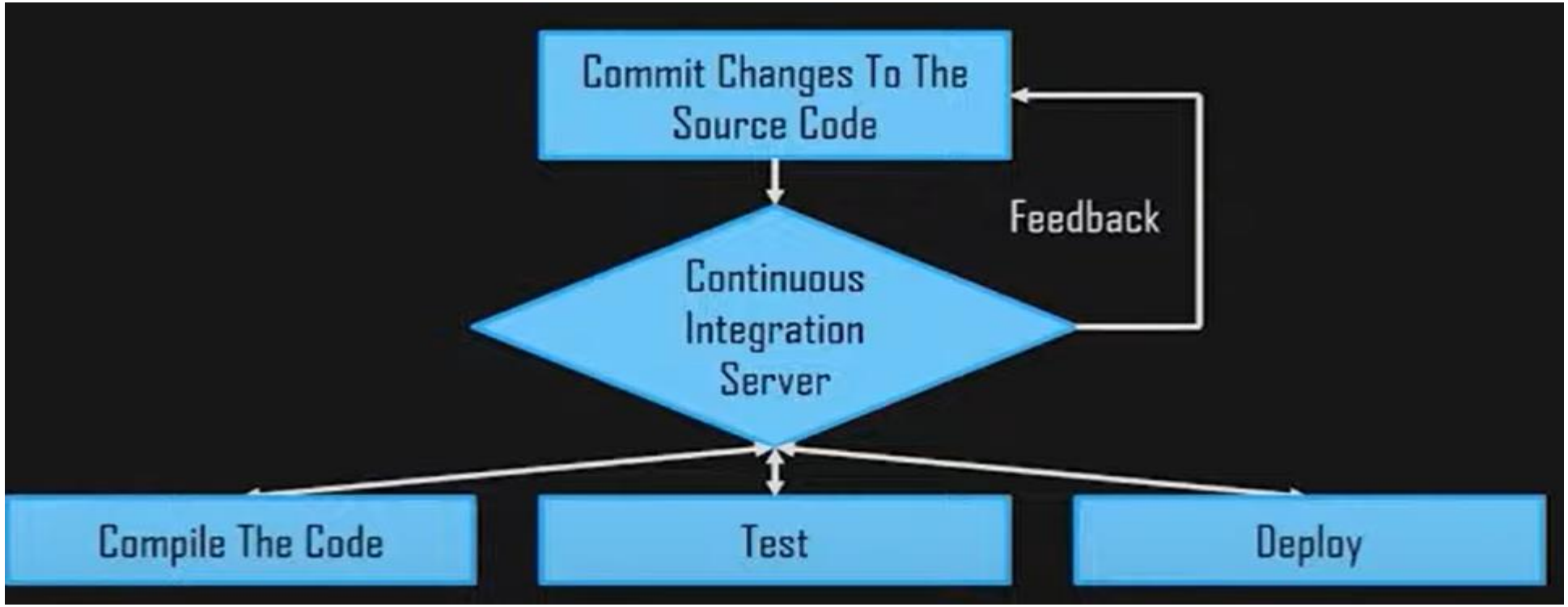
- Jenkins is a powerful application that allows continuous integration and continuous delivery of projects, regardless of the platform you are working on.
- It is a free source that can handle any kind of build or continuous integration.
- You can integrate Jenkins with a number of testing and deployment technologies
- Jenkins is a self-contained automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.
- Jenkins is written in Java with plugins built for Continuous Integration purpose.
- Jenkins is used to build and test your 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 you to continuously deliver your software by integrating with a large number of testing and deployment technologies.
- Continuous Integration is the most important part of DevOps that is used to integrate various DevOps stages.
- Jenkins is the most famous Continuous Integration tool.

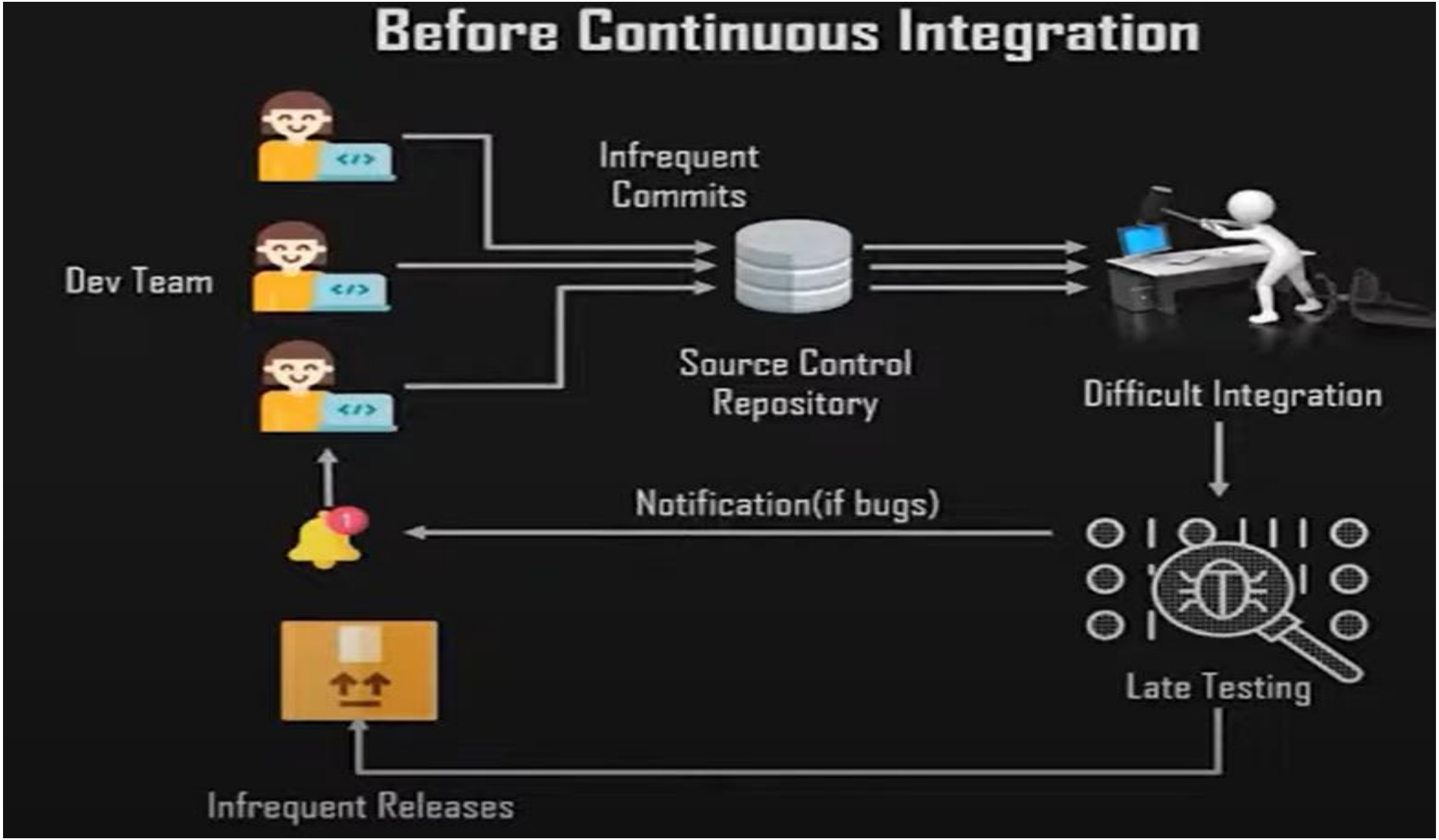
Introduction to Jenkins

- Jenkins achieves Continuous Integration with the help of plugins. Plugins allows the integration of Various DevOps stages. If you want to integrate a particular tool, you need to install the plugins for that tool. For example: Git, Maven etc.

What is Continuous Integration

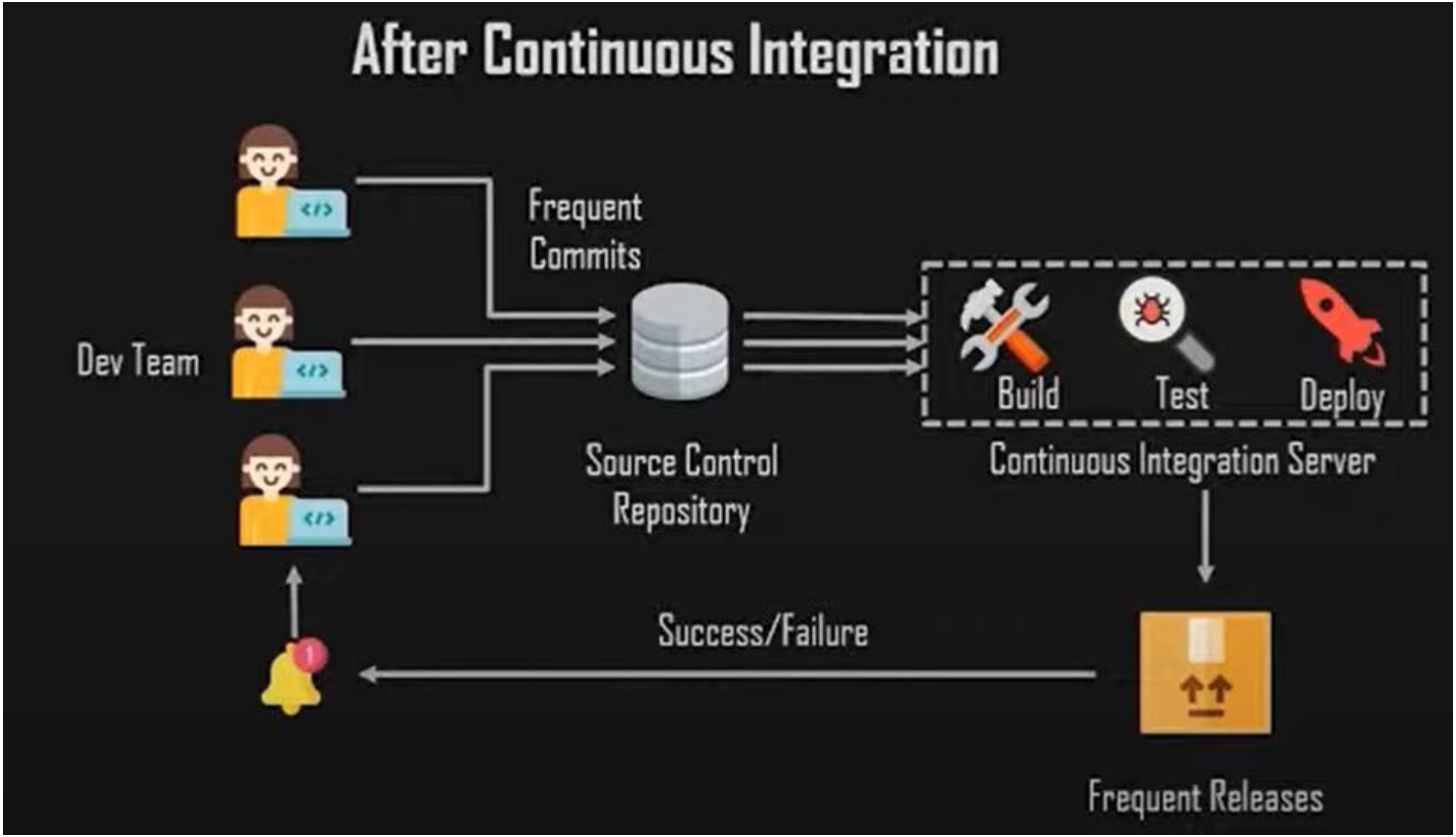
- Continuous Integration is a development practice in which the developers are required to commit changes to the source code in a shared repository several times a day or more frequently.
- Every commit made in the repository is then built. This allows the teams to detect the problems early. Apart from this, depending on the Continuous Integration tool, there are several other functions like deploying the build application on the test server, providing the concerned teams with the build and test results etc.




Before Continuous Integration
Infrequent
Commits
Dev Team
Source Control
Repository
Difficult Integration
Notification(if bugs)
Late Testing
Infrequent Releases

Issues faced before Continuous Integration

- Long Waits to test code
- Difficult to Debug
- Slow Software Delivery process
- Infrequent feedback cycles


After Continuous Integration
Frequent
Commits
Dev Team
Source Control
Repository
Build
Test
Deploy
Continuous Integration Server
Success/Failure
Frequent Releases

History of Jenkins

- The Jenkins project was started in 2004 (originally called Hudson) by Kohsuke Kawaguchi, while he worked for Sun Microsystems.
- Kohsuke was a developer at Sun and got tired of incurring the wrath of his team every time his code broke the build.
- The Hudson was renamed to Jenkins in 2011 after a dispute with Oracle, which had forked the project and claimed rights to the project name.
- The Oracle fork, Hudson, continued to be developed for a time before being donated to the Eclipse Foundation. Oracle's Hudson is no longer maintained and was announced as obsolete in February 2017.
- On April 20, 2016 version 2 was released with the Pipeline plugin enabled by default.
- The plugin allows for writing build instructions using a domain specific language based on Apache Groovy.

Other CI Tools

- Jenkins 48.11%
- TeamCity 6.83%
- Bamboo 1.76%
- Buddy
- GitLab CI
- CircleCI 5.61%
- TravisCI 1.51%

Stages in CI-CD

- The most common stages in CI-CD pipeline are
  - The Trigger
  - Code Checkout
  - Compile the code
  - Run unit tests
  - Package the code
  - Delivery or Deployment

Stages in CI-CD – The Trigger

- You can either configure your CI/CD tool to poll for changes to a Git repository, or schedule to trigger periodically
- The main reason for doing this is to make running the pipeline automatic, and friction-free.
- If you leave the pipeline to be run manually, people will sometimes forget to run it, or choose not to. It’s far better to just make this an automated step.
- There is huge confidence that comes from knowing that your code is going to be tested on every single change.

Stages in CI-CD – Code Checkout

- In this stage, the CI server will check out the code from the source code repository, such as GitHub or Bitbucket.
- The CI/CD tool usually receives information from a poll, or a scheduler, which says which specific commit triggered the pipeline.
- The pipeline then checks out the source code at a given commit point, and starts the process.
- Continuous Download

Stages in CI-CD – Compile the code

- If you’re developing in a compiled language like Java, the first thing you’ll probably need to do is compile your program.

- This means that your CI tool needs to have access to whatever build tools you need to compile your app. For example, if it’s Java, you’ll use something like Maven or Gradle.

- Ideally this stage should run in a clean, fresh environment.

- Continuous Build

Stages in CI-CD – Run tests

- The next key element of your CI/CD pipeline is unit testing. This is the stage where you configure your CI/CD tool to execute the tests that are in your codebase.

- You might use Maven or Gradle to do this in Java, or test in JavaScript.

- The aim at this point is not only to verify that all the unit tests pass, but that the tests are being maintained and enhanced as the code base grows.

- If the application is growing rapidly, but the number of tests stays the same, this isn’t great, because it could mean that there are large parts of the code base which are untested.

- Continuous Testing

Stages in CI-CD – Package the code

- Once all of the tests are passing, you can now move on to packaging the code. Exactly how you package your application depends on your programming language and target environment.

- If you're using Java, you might build a JAR file. If you're using Docker containers, you might build a Docker image.

- Whatever packaging format you choose, a good practice is that you should build the binary only once. Don't build a different binary for each environment, because this will cause your pipeline to become very complex.

- Continuous Delivery

Stages in CI-CD – Delivery or Deployment

- Finally, when the application has been tested, it can move into the delivery or deployment stage.

- At this stage, you have an artifact ready to be deployed (continuous delivery). Or, you can continue to CI/CD heaven and automatically deploy your software (continuous deployment).

- To achieve continuous deployment, you need a production environment to deploy into. If you're deploying to a public cloud, you can use the cloud provider's API to deploy your application. Or if you're using Kubernetes, you might use a Helm chart to deploy your app.

- And finally, this is the end of your pipeline!

- The next time a developer commits code into the repository, it will be run all over again.

Why Jenkins is so powerful?

- **Open-Source:** This is obviously a key reason for Jenkins' widespread appeal. It's free, so any organization can get started with it regardless of budgetary constraints. It's also easy to install: Jenkins was developed as a self-contained Java program, which means it can run on most devices and operating systems.
- **Community Support:** Because it is open-source and has been the go-to CI tool for many years, Jenkins now has a strong community behind it. This reinforces Jenkins' popularity, as it's easy to find tutorials and other resources to get the most out of it.
- **Plugins:** One of the main benefits of Jenkins is the huge catalog of plugins that can extend its functionality. At latest count, Jenkins has more than 1,800 community-developed plugins that allow any organization to adapt or enhance the base functionality to suit their own DevOps environment.
- **Distributed:** One server is enough for complex projects, so Jenkins uses a Master-Slave architecture to manage distributed builds. The main server is the 'Master', which can then distribute workload to other 'slave' servers, allowing for multiple builds and test environments to be running simultaneously. This approach could be used, for example, to build and test code on different operating systems.

Jenkins Setup

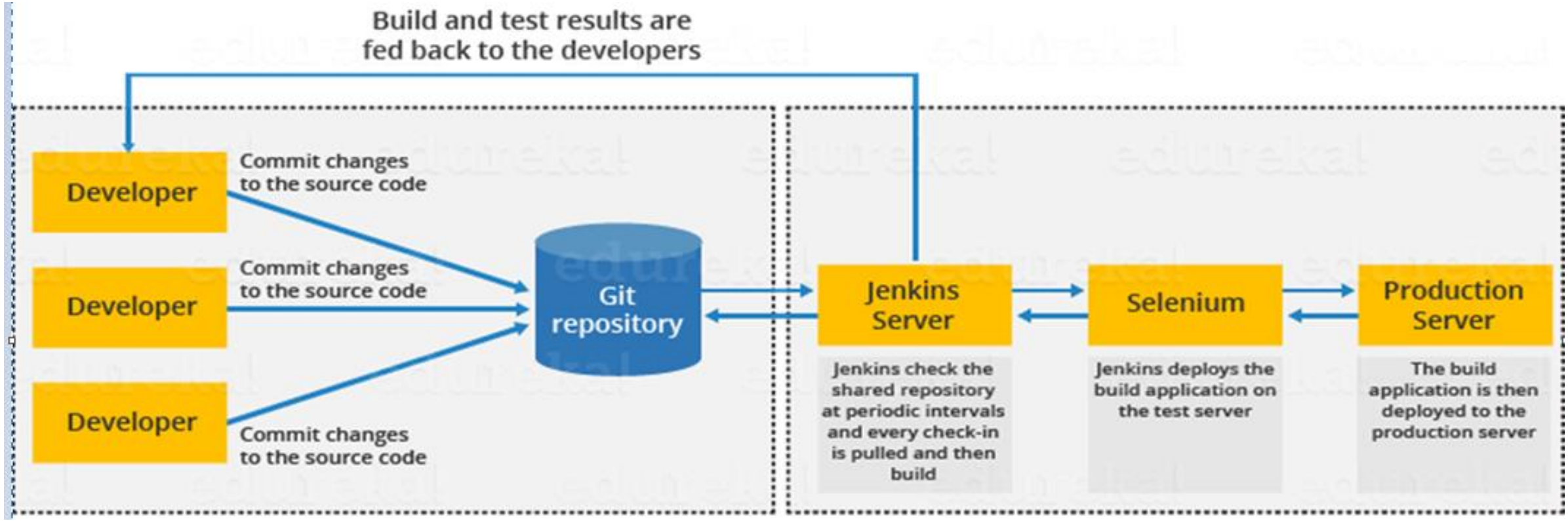
- Follow the Instructions given in below URL

Jenkins download and deployment

- Once Jenkins is installed, it runs on port no 8080 by default

    - Access the Jenkins from browser
    - Unlock Jenkins
    - Install Plugins
    - Create Admin user

Jenkins Architecture



Alternatives to setup the Jenkins

- Using Package Manager
  - Chocolatey
  - Brew
  - Yum
- Using Containers

Jenkins Job types

- Jenkins is a flexible and highly configurable automation server that can be used to automate many different types of tasks. Some common types of jobs in Jenkins include:
- **Freestyle project**: a basic type of Jenkins job that provides a wide range of options for customizing and configuring the build process.
- **Pipeline:** a type of Jenkins job that uses a Groovy script to define the entire build process, from building and testing the code to deploying it to production.
- **Multi-configuration project:** a type of Jenkins job that allows for running the same build with different configurations.
- **Folder:** a type of Jenkins job that allows you to organize and manage multiple jobs in a hierarchical structure.
- **Maven project:** a type of Jenkins job that is optimized for building and testing Maven-based projects.
- **Multi-Branch project:** a type of Jenkins job that allows for running the same build with different branches.

CRON Jobs in Jenkins

- Cron is the baked in task scheduler - run things at fixed times, repeat them etc. In fact, Jenkins uses something like the cron syntax when you are specifying specific times you want a Job to run.
- Syntactically, CRON contains 5 places for each time entry, thus a conventional one-liner syntax of the CRON template would be somehow like:

0 23 * * *
// Staged as
{Minute} {Hour} {DayOfMonth} {Month} {DayofWeek}

CRON Jobs in Jenkins

- **Minute (0-59)** – tells at what minute of the hour the Job should build.
- **Hour (0-23 With 0 is the midnight)** - tells at what hour the job should build in the 24-hour clock.
- **Day of the month (1-31)** - tells on which day of the month your job should build, e.g., 23rd of each month.
- **Month (1-12)** - tells the month when your job should build. You can number or name the month as well. i.e., April, May, etc.
- **Day of the week (0-7 With 0 or 7 both being Sunday)** - tells the day of the week when you want to build your job. It can also be a number or name like Mon etc.

- // Every day at 6 in the evening
- 0 18 * * *
- You need to make the first entry (minute) fixed if you want to work on the hour. The above expression means to run your job at the 0th minute of 18th hour DAILY without any specification of day, month or week.

CRON Jobs in Jenkins

- After every 20 minutes
  *0/20 * * * *
- After every 4 hours
  0 */4 * * *
  // OR
  0 0,4,8,12,16,20 * * *
- In the first case, it will run at 0th minute, 20 minutes after, 40 minutes after means after every 20 minutes. 20 is the offset here which tells after how many minutes this should trigger.
- Similarly, in the next example, the first entry (minute) is fixed and the second entry (hour) contains an offset which tells keep repeating it after 4 hours. In the next line, the less verbose and more readable syntax of the same CRON is also given.

- Every day at 11 in the morning and at night both
  0 11, 22 * * *
- You can define the multiple values for same entry using a comma. Here, the hour place has two values, first for 11 am and the next for 11 pm.

CRON Jobs in Jenkins

- Every Saturday and Sunday at 5 PM

  0 17 * * 6,7
- This will run every Saturday (6) and Sunday (7) at 5 pm (17).

Creating Users in Jenkins

- Generally, in a large organization, there are multiple, separate teams to manage and run jobs in Jenkins. But managing this crowd of users and assigning roles to them can prove troublesome.

- By default, Jenkins comes with very basic user creation options. You can create multiple users but can only assign the same global roles and privileges to them. This is not ideal, especially for a large organization.

- The Role Strategy Plugin enable you to assign different roles and privileges to different users. You will first need to install the plugin in your Jenkins mange environment.

User Authorization Strategies

- **Matrix Based Security**
- **Role Based Matrix Authorization Strategy (Project Based Roles)**
- **Role Based Strategy**

Matrix Based Security

- It allows you to grant specific permissions to users and groups
- Create User
  - Manage Jenkins → Manage Users → Craete User
- Manage Jenkins → Configure Global Security
  - Under Authorization → Select Matrix based security
  - Add user
  - Provide the permissions (Overall Read permission is mandatory)

Creating Project Based Roles in Jenkins

- Manage Jenkins → Manage Users
  - Create User
- Manage Jenkins → Configure Global Security → Authorization
  - Enable Project based Matrix Authorization Strategy
  - Adduser → Provide username that we created in above step
  - Select Overall Read permissions
- Click on Dashboard
  - Select the Job
  - Click on Configure
  - Select Enable Project-based security
  - Add user
  - Select permissions needed to maintain the Job

Role Based Strategy

- Create User in Jenkins
  - Manage Jenkins → Manage Users → Create User
- Install Plugin
  - Manage Jenkins → Manage Plugins → Available → Search fro Role Based Authorization Strategy
  - Install without Restart
  - Restart Jenkins after installing the plugin
- Manage Jenkins → Configure Global Security
  - Under Authorization → Select Role Based Strategy
- Create Roles
  - Manage Jenkins → Manage and Assign Roles → Manage Roles
  - For every Role Overall Read permission is mandatory
- Assign Roles to Users
  - Manage Jenkins → Manage and Assign Roles → Assign Roles
  - Add user
  - Map the user to Role

Email Configuration in Jenkins

- Manage Jenkins → Configure System
- At the bottom of the page you can see Email Notification
- Provide below details
  - SMTP Server Name
  - Default email suffix
  - Username
  - Password
  - Port
- Once Email Configuration is successful then Go to the Job → Configure
- Add Post Build action → Email Notification and provide recipient email details separated by space

Master Slave

- Sometimes many build machines are required if there are instances wherein there are a larger and heavier projects which get built on a regular basis.
- And running all of these builds on a central machine may not be the best option.
- In such a scenario, one can configure other Jenkins machines to be slave machines to take the load off the master Jenkins server.

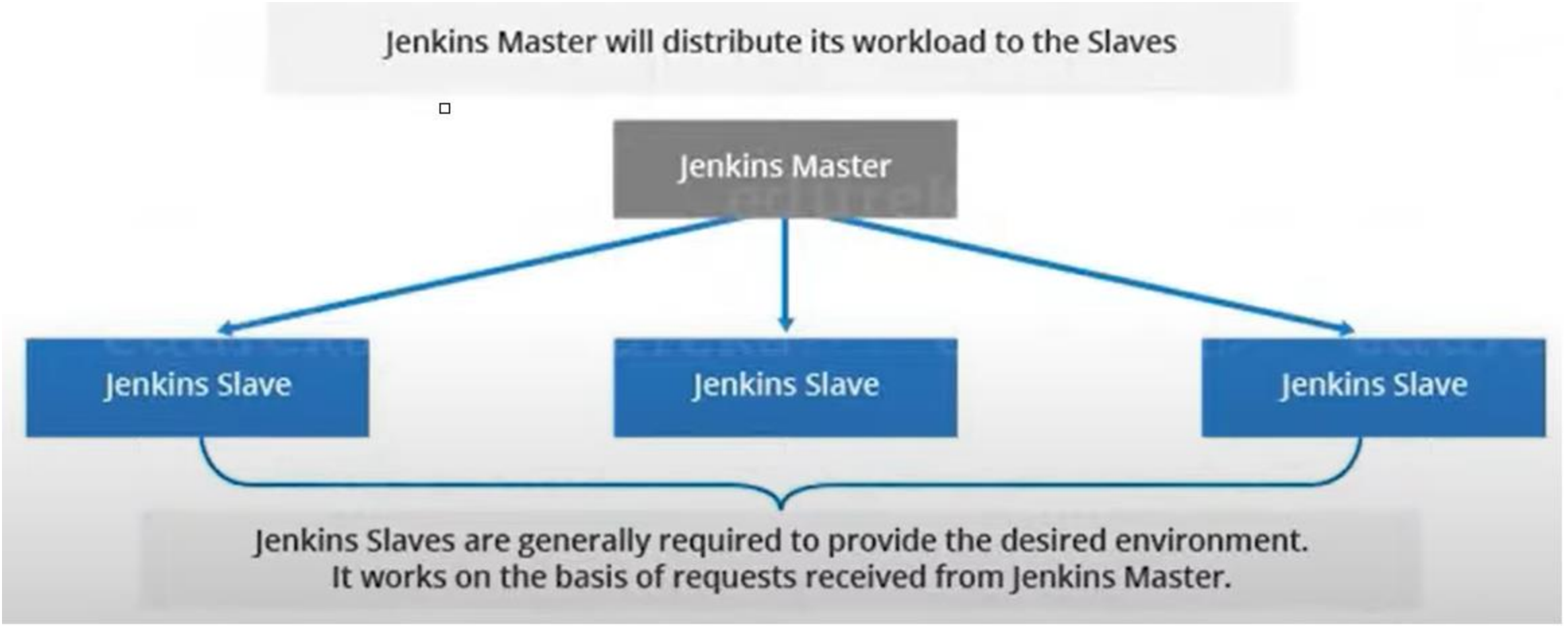
- Sometimes you might also need several different environments to test your builds. In this case using a slave to represent each of your required environments is almost a must.

- A slave is a computer that is set up to offload build projects from the master and once setup this distribution of tasks is fairly automatic.
- The exact delegation behavior depends on the configuration of each project; some projects may choose to "stick" to a particular machine for a build, while others may choose to roam freely between slaves.

Master Slave

- Since each slave runs a separate program called a "slave agent" there is no need to install the full Jenkins (package or compiled binaries) on a slave.
- There are various ways to start slave agents, but in the end the slave agent and Jenkins master needs to establish a bi-directional communication link (for example a TCP/IP socket.) in order to operate.

Jenkins Distributed Architecture



Jenkins Scalability

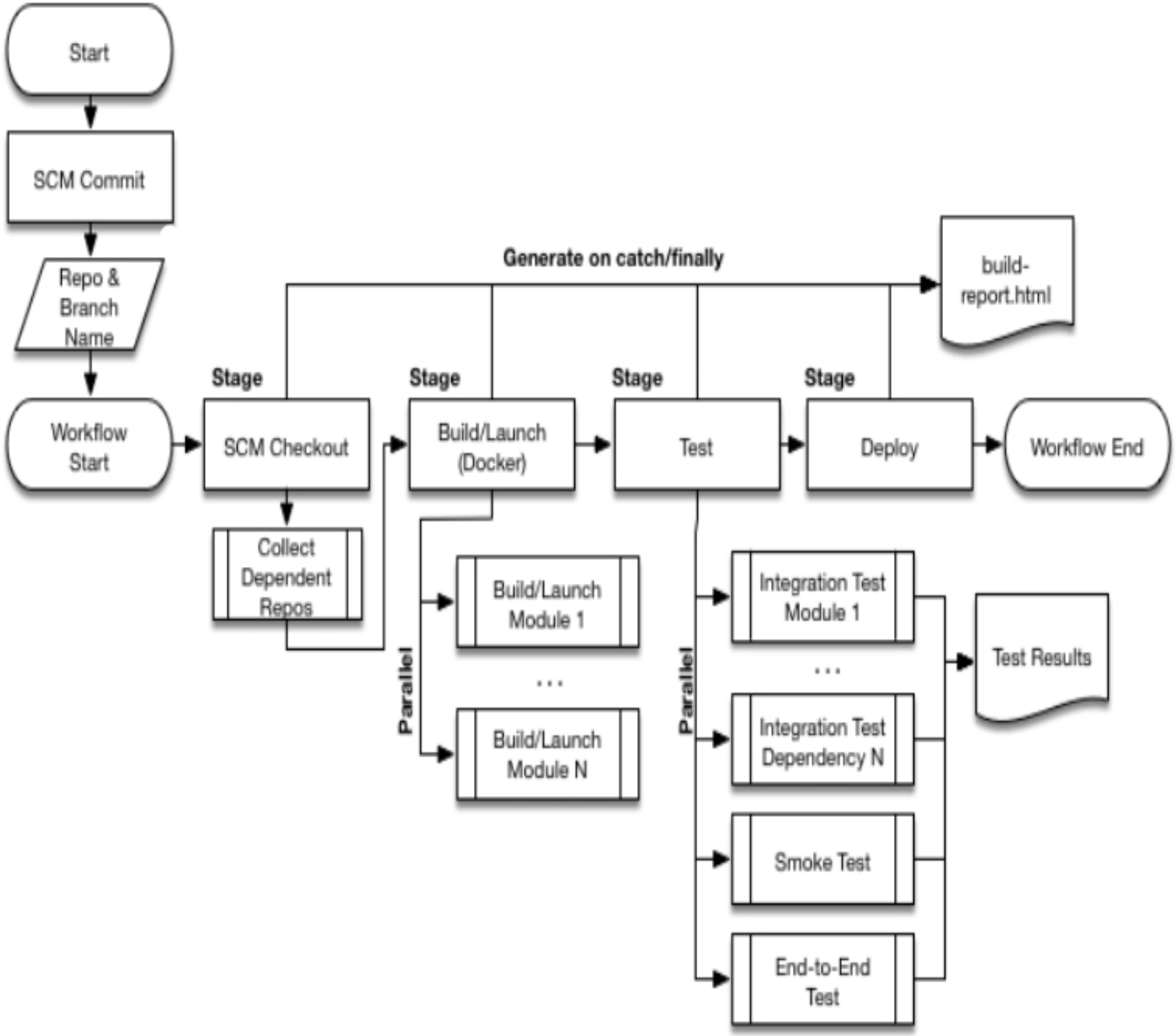
- Scalability is the ability of a system to expand its capabilities to handle the additional load.
- Jenkins scaling depends on Master/Slaves Model
  - Master – Main Jenkins Instance
  - Slave – Agent Instances
  - Master is responsible for distributing jobs across slaves.

Pipeline

- To carry out continuous delivery, Jenkins introduced a new feature called Jenkins pipeline.
- A pipeline is a collection of steps in the build job which trigger each other in a specified sequence that brings the software from version control into the hands of the end users by using automation tools.
- The key feature of this pipeline is to define the entire deployment flow through code. It means that all the standard jobs defined by Jenkins are manually written as one whole script and they can be stored in a version control system.
- It basically follows the ‘pipeline as code’ discipline. Instead of building several jobs for each phase, you can now code the entire workflow and put it in a **Jenkinsfile**.

Pipeline


Development


Start
SCM Commit
Repo & Branch Name
Workflow Start
Generate on catch/finally
build-report.html
Stage
SCM Checkout
Collect Dependent Repos
Stage
Build/Launch (Docker)
Parallel
Build/Launch Module 1
...
Build/Launch Module N
Stage
Test
Parallel
Integration Test Module 1
...
Integration Test Dependency N
Smoke Test
End-to-End Test
Test Results
Stage
Deploy
Workflow End


Production

Jenkinsfile

- Jenkins Pipeline can be defined by a text file called Jenkinsfile. You can implement pipeline as code using Jenkinsfile, and this can be defined by using a DSL (Domain Specific Language).
- With the help of Jenkinsfile, you can write the steps required for running a Jenkins Pipeline.
- The benefits of using Jenkinsfile are:
  - You can make pipelines automatically for all branches and can execute pull requests with just one Jenkinsfile.
  - You can review your code on the pipeline.
  - You can review your Jenkins pipeline.
  - This is the singular source for your pipeline and can be customized by multiple users.
- Jenkinsfile can be defined by using either Web UI or with a Jenkinsfile.

Pipeline

- Jenkins Pipelines can be either
  - Scripted Pipeline.
  - Declarative Pipeline

Scripted Pipeline

- Traditional way of writing the code
- Stricter groovy syntax
- Code is written in the Jenkins UI Instance
- The code is defined within a ‘node’ block.

Declarative Pipeline

- Makes use of numerous, generic, predefined build steps/stages (i.e. code snippets) to build our job according to our build/automation needs
- Recent feature
- which are written in Jenkins DSL language.
- Code is written locally in a file and is checked into a SCM.
- The code is defined within a ‘pipeline’ block.

Scripted Pipeline

- A Jenkins Scripted Pipeline is a sequence of stages to perform CI/CD-related tasks that can be specified as code, enabling you to develop a pipeline script and add it to your code repository so you can version it.
- Plus, the pipeline functionality comes right out of the box from Jenkins 2.0 onward, meaning that no configuration needs to be made to be able to create them.
- In Jenkins, Pipelines are specified by using a specific DSL following almost the same structure as Groovy to specify its statements and expressions. This makes pipelines easy to use for Groovy savants.
- In Scripted Pipeline syntax, one or more node blocks do the core work throughout the entire Pipeline.

Scripted Pipeline fundamentals

Jenkinsfile (Scripted Pipeline)

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

Scripted Pipeline fundamentals

1. Execute this Pipeline or any of its stages, on any available agent.
2. Defines the "Build" stage. stage blocks are optional in Scripted Pipeline syntax. However, implementing stage blocks in a Scripted Pipeline provides clearer visualization of each `stage’s subset of tasks/steps in the Jenkins UI.
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.

Scripted Pipeline Elements

Node Blocks

- The first block to be defined is the “node”:

```
node {
}
```

- A “node” is part of the Jenkins distributed mode architecture, where the workload can be delegated to multiple “agent” nodes.
- A “master” node handles all the tasks in your environment. Jenkins agent nodes offloads builds from the master node, thus performing all the pipeline work specified in the node block.
- This block is not mandatory but it is desired and can be considered as good practice, since with the code included in this block, Jenkins will schedule and run all the steps once any node is available and creates a specific workspace directory.

Scripted Pipeline Elements

Stage Blocks

- The next required section is the “stage”:

```
stage() {
}
```

- Your pipeline will consist of several steps that can be grouped in stages. Among these stages you might have:
  - Pull code from repository
  - Build your project
  - Deploy your application
  - Perform functional tests
  - Perform performance tests
- Jenkins provides an interface that generates pipeline sentences for predefined actions that can be added to any of the script stages. On your pipeline script page, click on “Pipeline Syntax” to access the following page:

Scripted Pipeline Elements

- Each stage block specifies the tasks to be performed. For example, a full scripted pipeline might be:

```
node {
        stage (‘Build’) {
                bat "msbuild
${C:\\Jenkins\\my_project\\workspace\\test\\my_project.sln}"
                }

        stage('Selenium tests') {
                dir(automation_path) {
                //changes the path to “automation_path”
                bat "mvn clean test -Dsuite=SMOKE_TEST -Denvironment=QA"
                }
        }
}
```

Declarative Pipeline fundamentals

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

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.

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

Declarative Pipeline When Condition

- Execution of the pipeline stages can be controlled with conditions.
- These conditions must be defined in the when block within each stage.
- Jenkins supports a set of significant conditions that can be defined to limit stage execution.
- Each when block must contain at least one condition. If more than one condition is declared in the when block, all conditions should return true for that stage being executed

Declarative Pipeline When Condition

```
pipeline {
    agent any
    stages {
        stage("Test") {
            when {
                equals(actual: currentBuild.number, expected: 1)
            }
            steps {
                echo "Hello World!"
            }
        }
    }
}
```

- Stage Test in the above example is run only and only one time at the first run of the pipeline job. This stage is not run from build two onwards.

Declarative Pipeline When Condition

```
pipeline {
    agent any
    stages {
        stage("Test") {
            when {
                branch "release-*"
            }
            steps {
                echo "Hello World!"
            }
        }
    }
}
```

Declarative Pipeline When Condition

```
pipeline {
    agent any
    environment {
        ENV = "testing"
    }
    stages {
        stage("Test") {
            when {
                environment(name: "ENV", value: "testing")
            }
            steps {
                echo "Test stage."
            }
        }
    }
}
```

Scripted vs Declarative Pipeline

- Jenkins provides two different syntaxes for pipelines. When DevOps engineers write a Jenkins pipeline, they can choose between declarative and scripted. The differences between the two are both subtle and significant.

- When the Jenkins pipeline was first introduced, the scripted pipeline was the only available option. Software developers enthusiastically embraced the scripted pipeline for two big reasons:
  - It provided a domain specific language that simplified many tasks a Jenkins developer would perform.
  - At the same time, it allowed developers to inject Groovy code into their pipelines any time.
- Groovy is a JVM-based programming language, which means a scripted Jenkins pipeline that uses Groovy has access to the vast array of APIs that are packaged with the JDK. The scripted pipeline developer has an immense amount of power.

Scripted vs Declarative Pipeline

Scripted Pipeline drawbacks

- The development community found that pipeline builders with strong Java and Groovy skills, but little experience with Jenkins, would often write complicated Groovy code to add functionality that's already available through the Jenkins DSL.
- A Jenkins pipeline should be a simple, easy-to-read and easy-to-manage component. Excessive code in a scripted pipeline violates one of the fundamental CI/CD principles.
- As a result, Jenkins corrected this trend when it introduced the declarative pipeline.

Scripted vs Declarative Pipeline

- In contrast to the scripted pipeline, the declarative Jenkins pipeline doesn't permit a developer to inject code. A Groovy script or a Java API reference in a declarative pipeline will cause a compilation failure.

- While this restriction might sound limiting on the surface, it's not.

- The Jenkins DSL provides a variety of facilities to introduce simple conditional logic. A declarative pipeline supports conditional statement usage, allows access to environment variables and provides facilities to add logging and error handling. The tradeoff is that declarative pipelines don't allow deep integration into Groovy and Java APIs.
- It's always been a best practice to keep complex code out of a Jenkins pipeline. The declarative pipeline simply enforces this.

Scripted vs Declarative Pipeline

- Scripted and declarative pipelines are different only in their programmatic approach. One uses a declarative programming model, while the other uses an imperative programming model.

- But they both run on the same Jenkins pipeline sub-system. There are no differences in the runtime performance, scalability and problem solvability perspective in the declarative vs. scripted pipeline debate. They only differ in the syntactic approach used to achieve an end goal.

Declarative

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {

            }
        }
        stage('Test') {
            steps {

            }
        }
        stage('Deploy') {
            steps {

            }
        }
    }
}
```

Scripted

```
node {
    stage('Build') {

    }
    stage('Test') {

    }
    stage('Deploy') {

    }
}
```

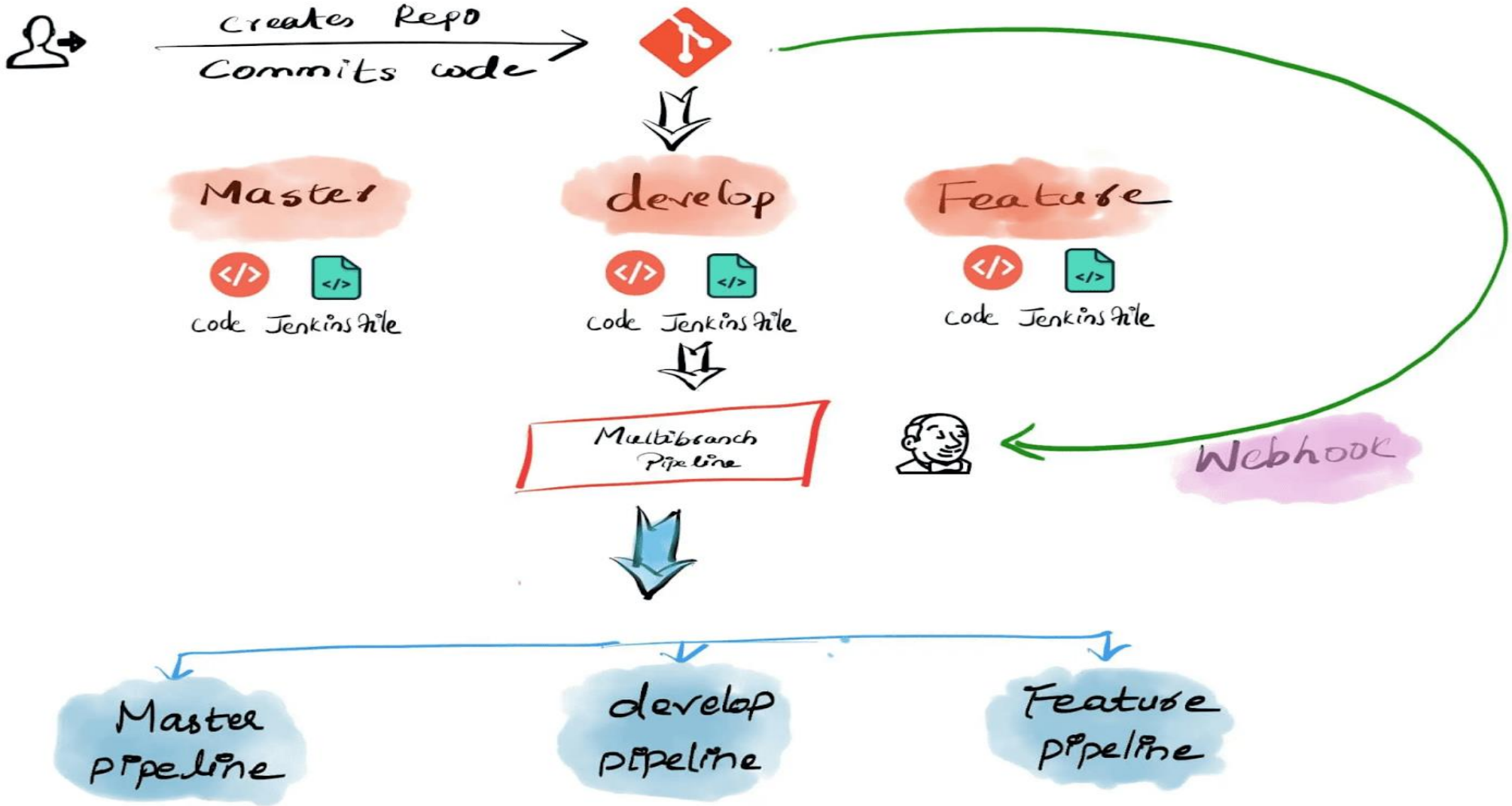
Multibranch Pipeline in Jenkins

- When you start with Jenkins, you'll probably be working with freestyle or pipeline jobs.
- However, there might be times when your application needs to have multiple branches in a single git repository.
- Every time you create a new branch, you need to create a new Jenkins job, but that translates into a lot of work when you're creating multiple branches. This is one of the problems that multibranch pipeline jobs solve.
- A multibranch job is simply a folder of pipeline jobs.
- For every branch you have, Jenkins will create a folder. So instead of creating a pipeline job for each of the branches you have in a git repository, you could use a multibranch job. This means that you'll have to create only one job.
- You also need to define where the Jenkinsfile is, and it's important that this file is at the same location in all the branches you create.
- You don't have to worry about doing maintenance as Jenkins will remove the job as well when the branch is removed. And if you need to have custom stages for branches different than the one you use for production, you can do that, too, by defining the custom behavior in the Jenkinsfile.

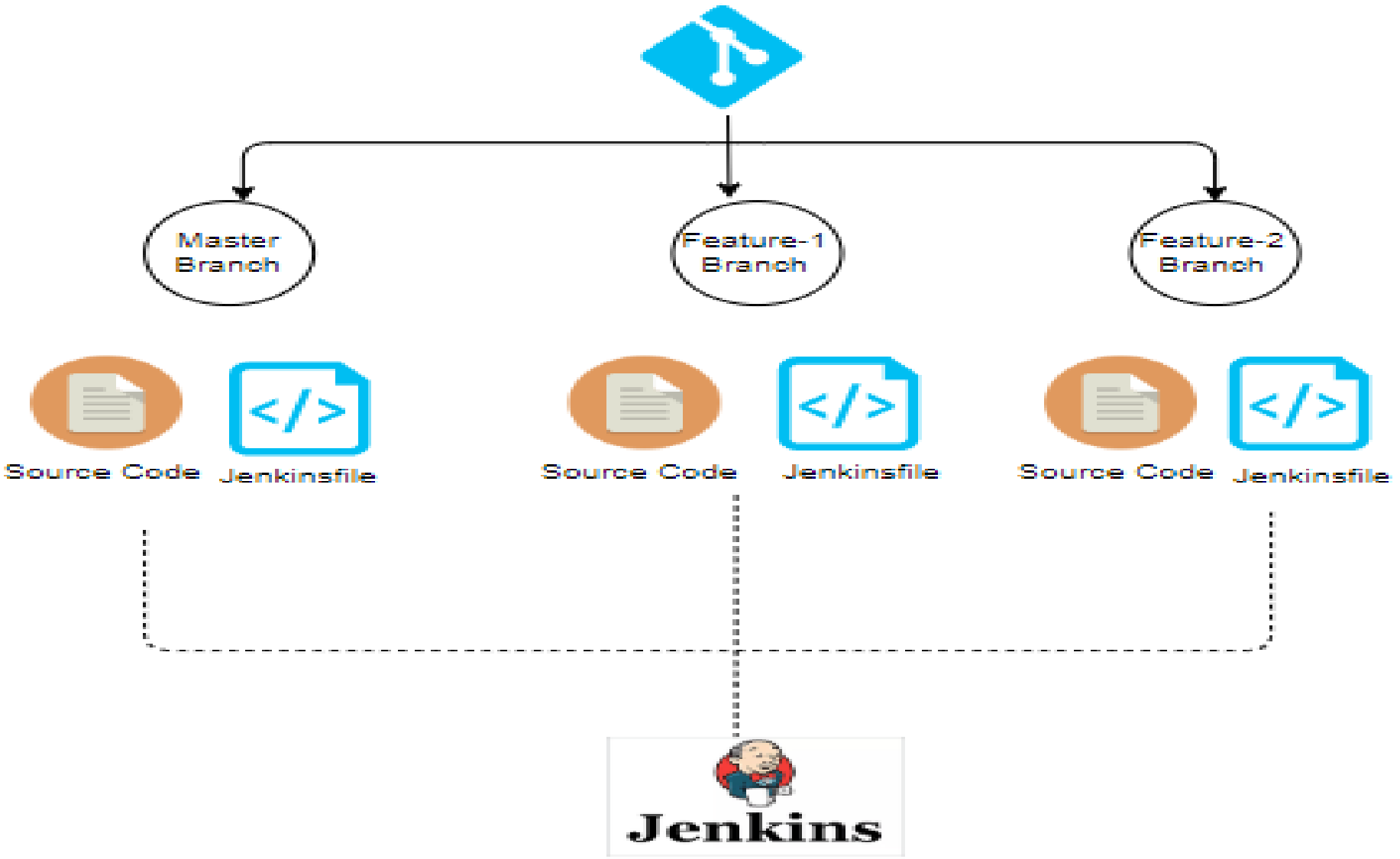
Multibranch Pipeline in Jenkins

- The Multibranch Pipeline project type enables you to implement different Jenkinsfiles for different branches of the same project.
- In a Multibranch Pipeline project, Jenkins automatically discovers, manages and executes Pipelines for branches which contain a Jenkinsfile in source control.
- If you want to create automated pull request based or branch-based Jenkins Continuous Integration & Continuous Delivery pipeline, the Jenkins multibranch pipeline is a way to go.
- A multibranch pipeline is a pipeline that has multiple branches. The main advantage of using a multibranch pipeline is to build and deploy multiple branches from a single repository.
- Having a multibranch pipeline also allows you to have different environments for different branches. However, it is not recommended to use a multibranch pipeline if you do not have a standard branching and CI/CD strategy.
- Working with multiple branches at the same time isn’t a problem in Jenkins. You saw how easy it is to create a multibranch pipeline job where Jenkins creates a new independent job automatically for every branch you create.

Multibranch Pipeline in Jenkins



Multibranch Pipeline in Jenkins



Exception Handling in Jenkins

catchError

- If the body throws an exception, mark the build as a failure, but nonetheless continue to execute the Pipeline from the statement following the catchError step.
- The behavior of the step when an exception is thrown can be configured to print a message, set a build result other than failure, change the stage result, or ignore certain kinds of exceptions that are used to interrupt the build.
- This step is most useful when used in Declarative Pipeline or with the options to set the stage result or ignore build interruptions. Otherwise, consider using plain try-catch(-finally) blocks. It is also useful when using certain post-build actions (notifiers) originally defined for freestyle projects which pay attention to the result of the ongoing build.
- If the shell step fails, the Pipeline build’s status will be set to failed, so that the subsequent mail step will see that this build is failed. In the case of the mail sender, this means that it will send mail.
- When running a Jenkins job, we want stage to be able to fail buit I don’t want whole the job to be failed.
- catchError will mark the Job as failed but rest of the pipeline continues to execute

Metrics

- Jenkins provides many plugins to present the metrics for builds which are carried out over a period of time.
- These metrics are useful to see the build and to understand how frequently they fail/pass over time.
- Let's see the "Build History Metrics Plugin", this plugin is used to calculate the following metrics for all of the builds once installed:
  - MTTF (Metrics Time to Failure)
  - MTTR (Mean Time to Recovery)
    - The time it takes to recover after the occurrence of the failure
    - It is the time spent during the intervention in a given process
    - MTTR = Downtime / Number of failures
    - The average time it takes to recover from a product or system failure. This includes the full time of the outage—from the time the system or product fails to the time that it becomes fully operational again.
  - Standard Deviation of Build Times
    - A standard deviation (or σ) is a measure of how dispersed the data is in relation to the mean. Low standard deviation means data are clustered around the mean, and high standard deviation indicates data are more

Trends

- To see overall **trends** in Jenkins, there are plugins available to gather information from within the builds and Jenkins and display them in a graphical format.
- One example of such a plugin is the ‘Hudson global-build-stats plugin’.

Jenkins Backup Plugin

- It is important to have a Jenkins backup with its data and configurations. It includes job configs, plugins, build logs, plugin configuration, etc.
- Jenkins provides a backup plugin which can be used to get backup critical configuration settings related to Jenkins

Q & A