

DevOps



Caltech

Center for Technology &
Management Education

Post Graduate Program in DevOps

©Simplilearn. All rights reserved.

Learning Objectives

By the end of this lesson, you'll be able to:

- 🕒 Define CI/CD Pipeline
- 🕒 Differentiate build jobs and Pipelines
- 🕒 Create a Continuous Integration Pipeline
- 🕒 Explain the Declarative Pipeline Syntax
- 🕒 Build a Pipeline from a Jenkinsfile




Introducing Pipelines

Pipelines

- Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery Pipelines into Jenkins.
- It is basically Jenkins jobs enabled by the Pipeline plugin.

A **Continuous Delivery (CD) Pipeline** is an automated expression of your process for getting software from version control right through to your users and customers.

Why Pipeline?



Freestyle jobs do not create a persistent record of execution, allow one script to address all the steps in a complex workflow, or have the other advantages of Pipelines.

In contrast, Pipelines enable you to define the whole application lifecycle.

Pipeline functionality helps Jenkins to support continuous delivery (CD).

The Pipeline plugin was built with requirements for a flexible, extensible, and script-based CD workflow capability in mind.

Advantages of Pipelines

Extensible:

The Pipeline plugin supports custom extensions to its DSL and multiple options for integration with other plugins.

Durable:

Pipelines can survive both planned and unplanned restarts of Jenkins master.

Pausable:

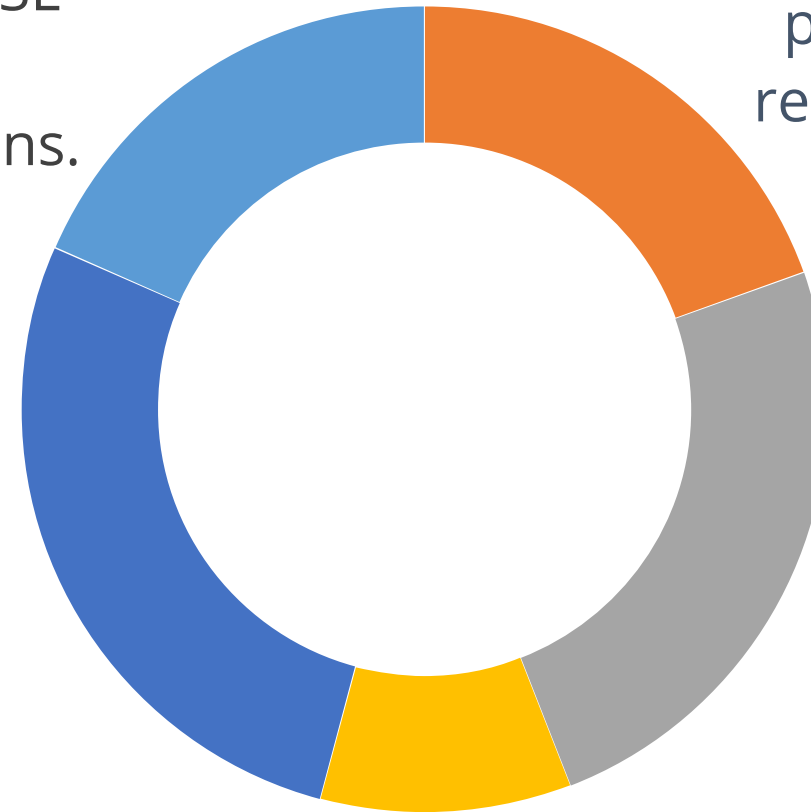
Pipelines can optionally stop and wait for human input or approval before completing the jobs for which they were built.

Versatile:

Pipelines support complex real-world CD requirements, including the ability to fork or join, loop, and work in parallel with each other.

Efficient:

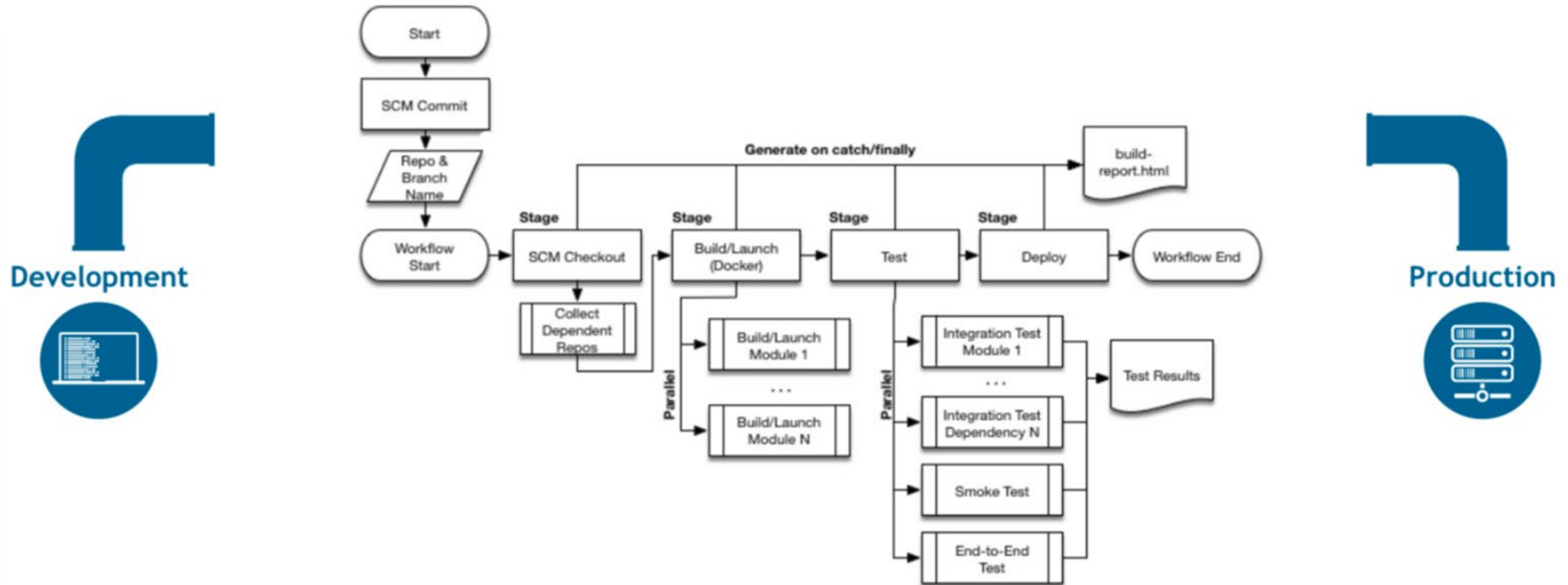
Pipelines can restart from any of several saved checkpoints.



Understanding Pipeline Structure

Pipeline Workflow

The flowchart below shows a Continuous Delivery scenario enabled by the Pipeline plugin:



Pipeline Terminology

Here are a few keywords in the Jenkins Pipeline and their definitions:

Pipeline:

- A Pipeline is a user-defined model of a CD pipeline.
- A Pipeline's code defines your entire build process, which typically includes stages for building an application, testing it, and then delivering it.

Node:

- A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.

Pipeline Terminology

Here are a few keywords in the Jenkins Pipeline and their definitions:

Stage:

- A stage block defines a conceptually distinct subset of tasks performed through the entire Pipeline.
- For example: Build, Test, and Deploy

Step:

- A step is a single task.
- A step tells Jenkins what to do at a particular point in time.
- For example: checkout code from repository or execute a script

Defining a Pipeline

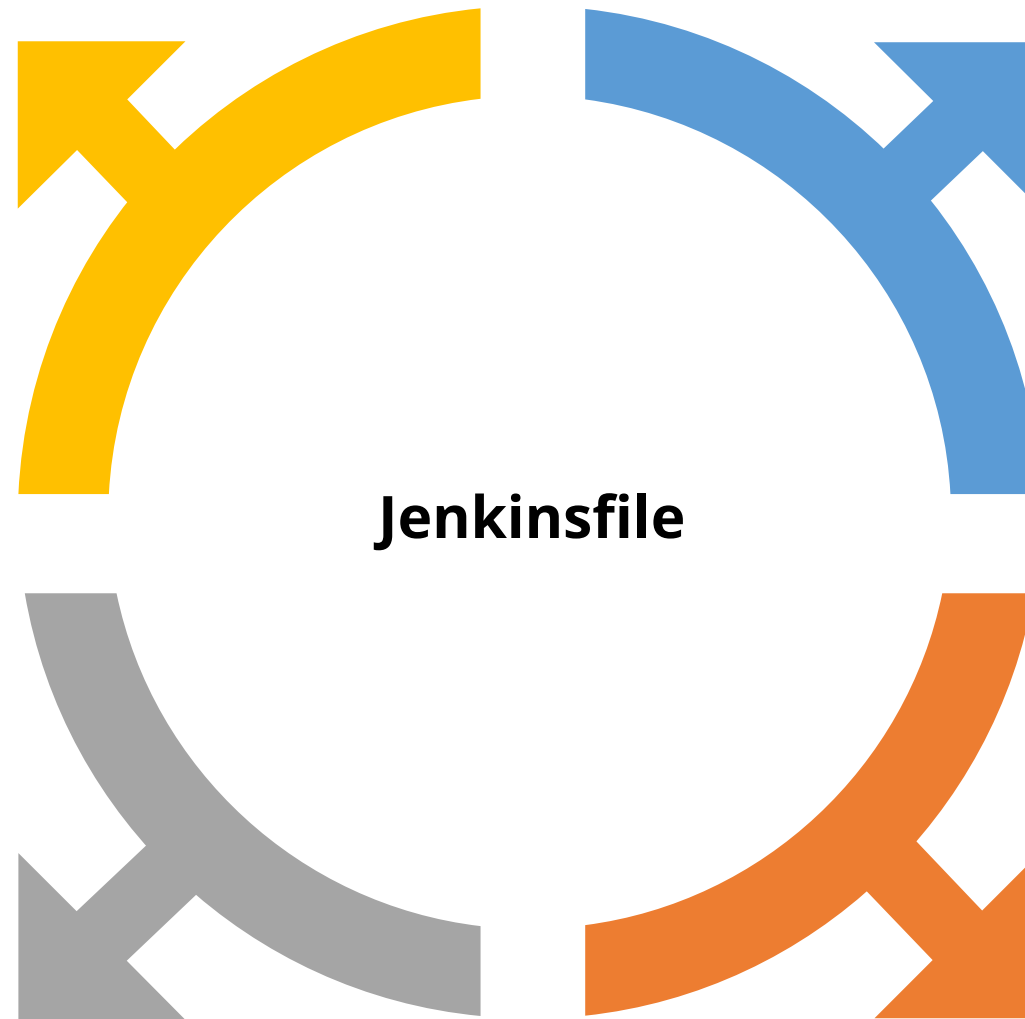
Defining a Pipeline

- Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines as code via the Pipeline Domain-Specific Language (DSL) syntax.
- The definition of a Jenkins Pipeline is written into a text file called a **Jenkinsfile**.
- A Jenkinsfile can be written using two types of syntax: **Declarative** and **Scripted**.
- Both Declarative and Scripted Pipeline are DSLs to describe portions of your software delivery pipeline.

Benefits of Jenkinsfile

Creates a single source of truth for the Pipeline, which can be viewed and edited by multiple members of the project.

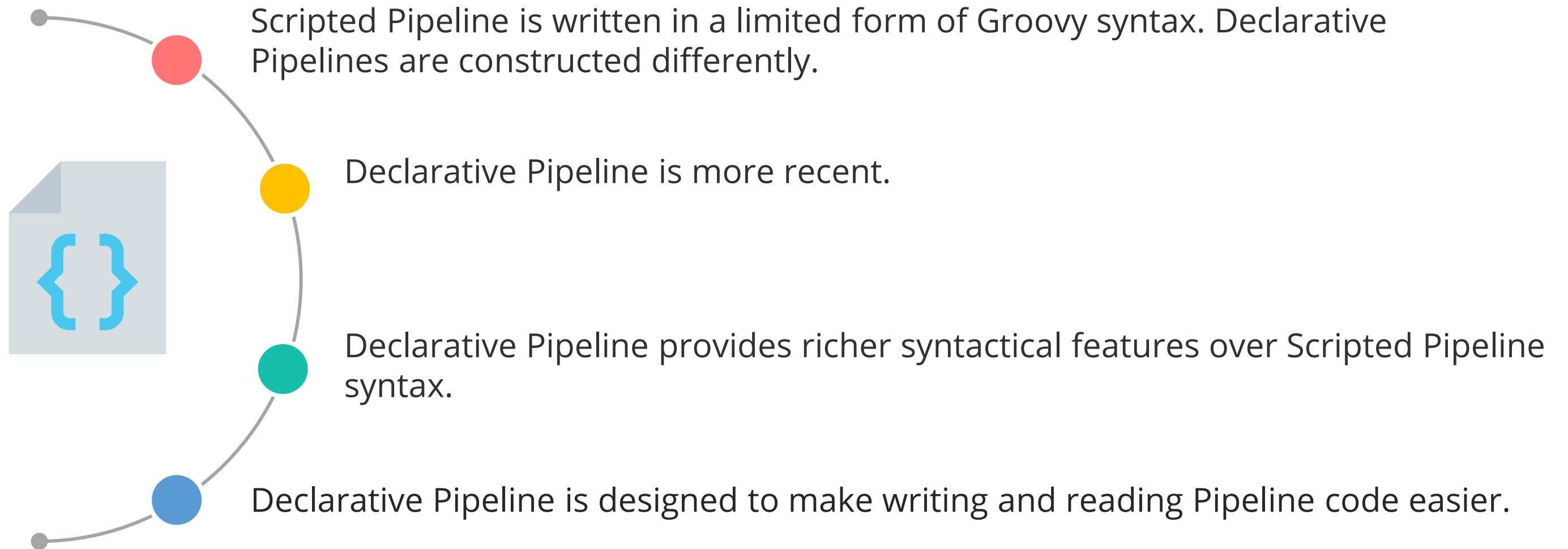
Automatically creates a Pipeline build process for all branches and pull requests.



Leaves an audit trail for the Pipeline.

Undergoes code review/iteration on the Pipeline along with the remaining source code.

Scripted vs. Declarative Syntax



Declarative Pipeline Syntax

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any ←  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Execute this Pipeline or any of its stages, on any available agent.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Defines the Build stage.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Perform some steps related to the Build stage.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Defines the Test stage.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Perform some steps related to the Test stage.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Defines the Deploy stage.

Declarative Pipeline Syntax

In Declarative Pipeline syntax, the Pipeline block defines all the work done throughout your entire Pipeline. The box below shows the Declarative Pipeline syntax:

Jenkinsfile (Declarative Pipeline)

```
Pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        //  
      }  
    }  
    stage('Test') {  
      steps {  
        //  
      }  
    }  
    stage('Deploy') {  
      steps {  
        //  
      }  
    }  
  }  
}
```

Perform some steps related to the Deploy stage.

Declarative Pipeline Syntax

- The declarative syntax was designed to make Pipeline building simple. This is why the syntax is limited only to some important keywords.
- A declarative Pipeline is always specified inside the ***Pipeline*** block and contains:



The complete Pipeline syntax description can be found on the official Jenkins page at <https://jenkins.io/doc/book/Pipeline/syntax/>.

Declarative Pipeline Syntax

The code block below shows a sample Pipeline script:

```
Pipeline {
  agent any
  triggers { cron('* * * * *') }
  options { timeout(time: 5) }
  parameters {
    booleanParam(name: 'DEBUG_BUILD', defaultValue: true,
      description: 'Is it the debug build?')
  }
  stages {
    stage('Example') {
      environment { NAME = 'Simplilearn' }
      when { expression { return params.DEBUG_BUILD } }
      steps {
        echo "Hello from $NAME"
        script {
          def browsers = ['chrome', 'firefox']
          for (int i = 0; i < browsers.size(); ++i) {
            echo "Testing the ${browsers[i]} browser."
          }
        }
      }
    }
  }
  post { always { echo 'Hello again!' } }
}
```

Pipeline Keywords

Here are the important keywords in the Pipeline syntax and their meanings:

Sections:

- Sections define the Pipeline structure and usually contain one or more directives or steps.
- They are defined with the following keywords:
 - **Stages:** This defines a series of one or more stage directives
 - **Steps:** This defines a series of one or more step instructions
 - **Post:** This defines a series of one or more step instructions that are run at the end of the Pipeline build. It is marked with a condition and is usually used to send notifications after the Pipeline build.

Pipeline Keywords

Directives express the configuration of a Pipeline or its parts. They are:

- **Agent:** This specifies where the execution takes place. It can define the label to match the equally labeled agents or docker to specify a container that is dynamically provisioned to provide an environment for the Pipeline execution.
- **Triggers:** This defines automated ways to trigger the Pipeline. It can use cron to set the time-based scheduling or poll SCM to check the repository for changes.
- **Options:** This specifies Pipeline-specific options. For example: timeout or retry
- **Environment:** This defines a set of key values used as environment variables during the build.
- **Parameters:** This defines a list of user-input parameters.
- **Stage:** This allows for logical grouping of steps.
- **When:** This determines whether the stage should be executed depending on the given condition.

Pipeline Keywords

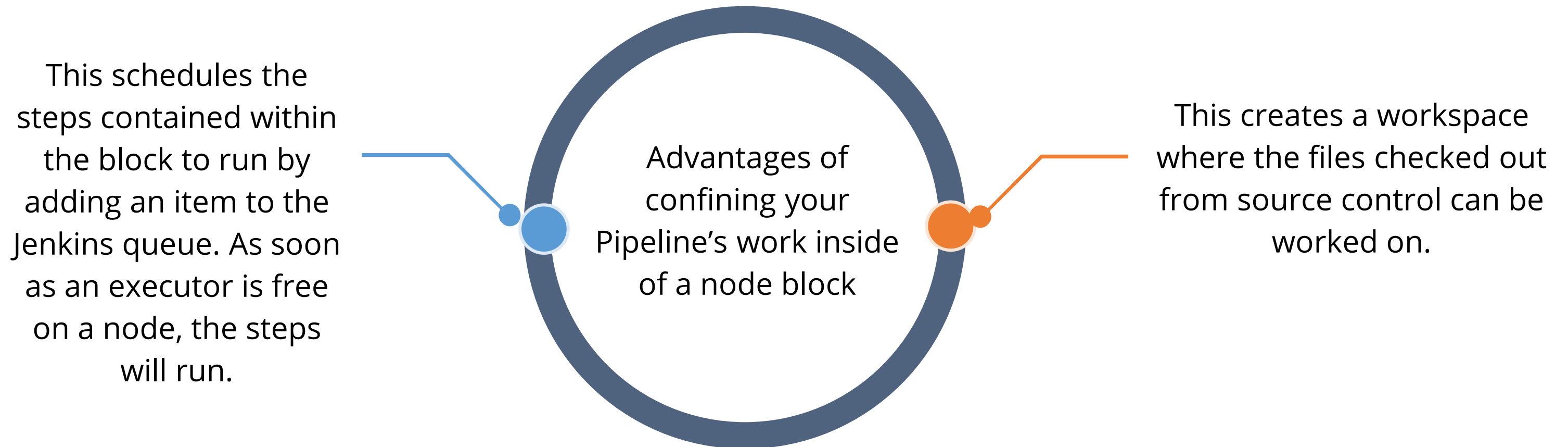
Steps: are the most fundamental part of the Pipeline. The keywords used in steps are:

- **sh:**
 - This executes the shell commands.
 - It's possible to define almost every operation using *sh*.
- **custom:**
 - Jenkins offers a lot of operations that can be used as steps.
 - Many of them are simple wrappers over the *sh* command used for convenience.
 - Plugins can also define their own operations.
- **script:**
 - This executes a block of the Groovy-based code that can be used for some non-trivial scenarios, where flow control is needed.

Scripted Pipeline Syntax

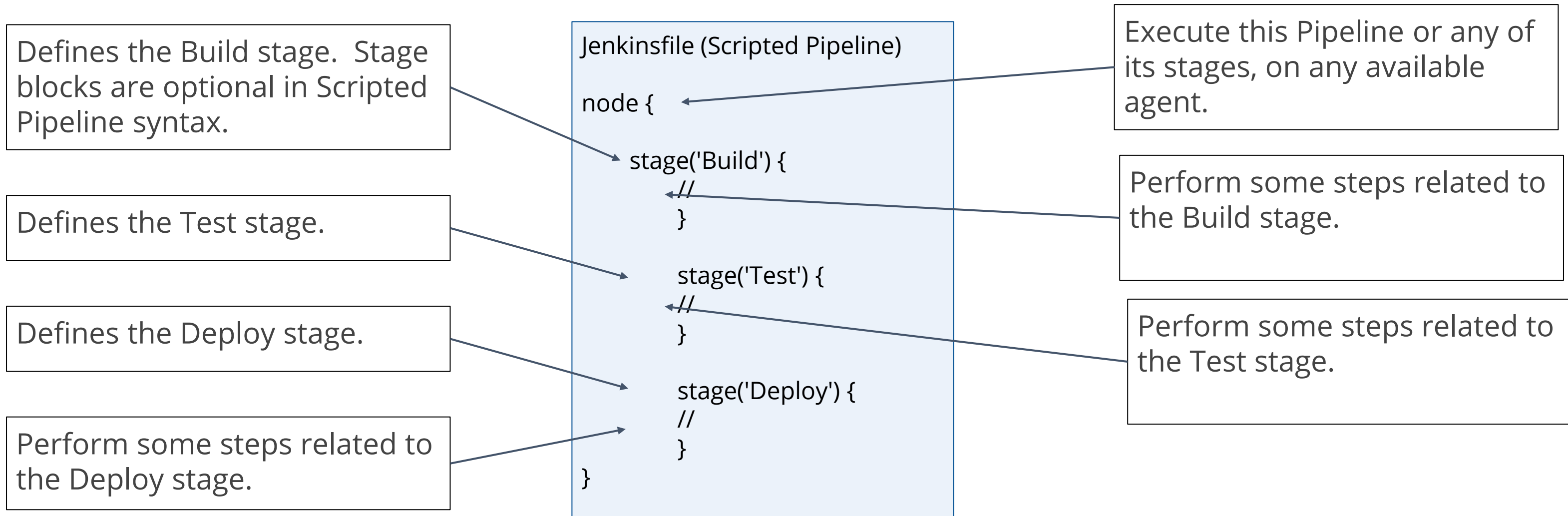
Scripted Pipeline Syntax

- In Scripted Pipeline syntax, one or more node blocks do the core work throughout the entire Pipeline.
- A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.
- This is not a mandatory requirement of Scripted Pipeline syntax.



Scripted Pipeline Syntax

The box below shows the Scripted Pipeline syntax:



Assisted Practice

Pipeline

Problem Statement: You have been asked to configure the Jenkins instance to include the Pipeline functionality and test it.

Steps to perform:

1. Configure the Pipeline plugin in Jenkins
2. Create a Pipeline
3. Test run the Pipeline

Building a Pipeline

Building a Pipeline

There are three ways to create a Pipeline:



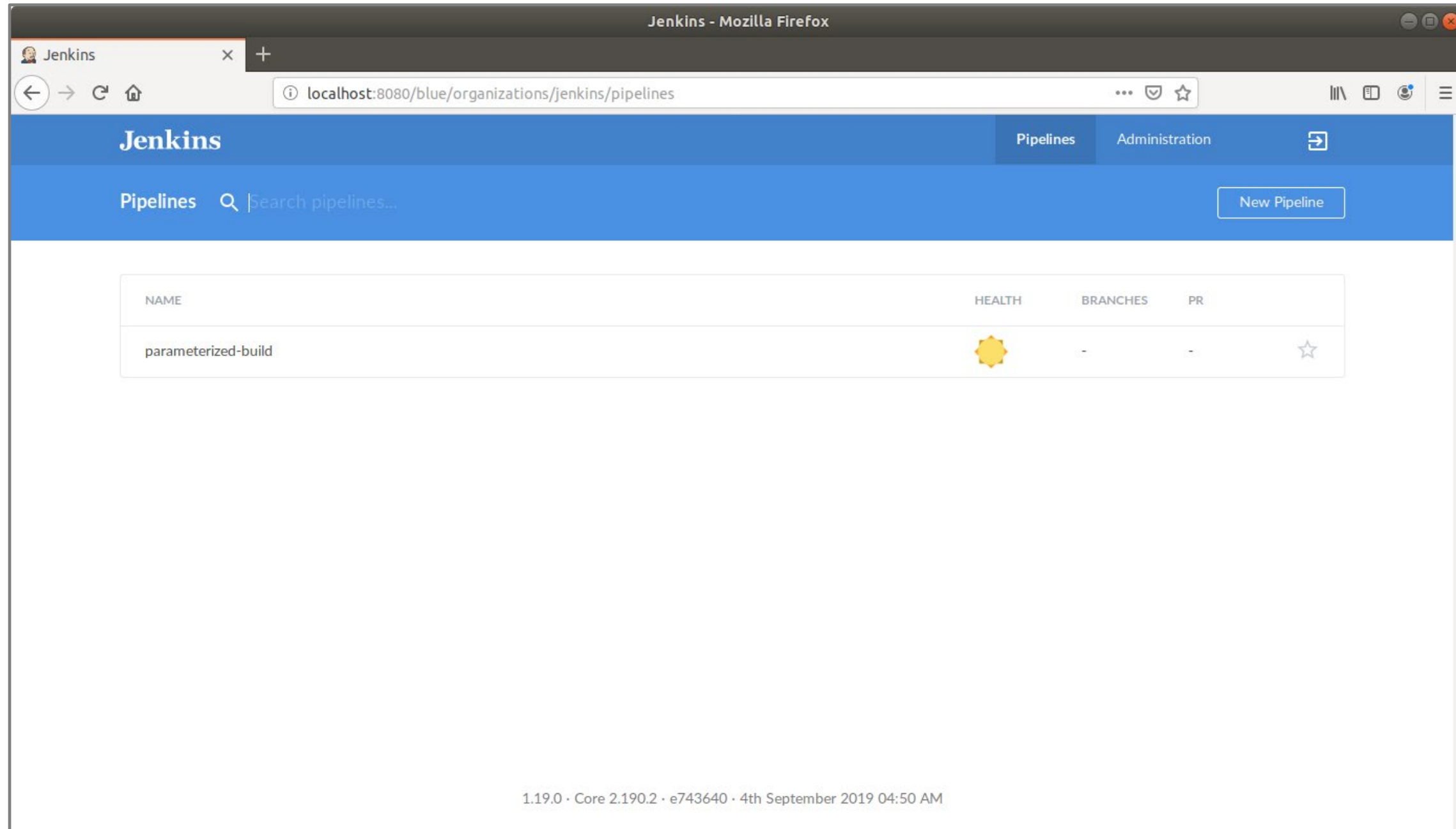
Through
Blue Ocean

Through the
classic UI

Through
Jenkinsfile

Building Pipelines with Blue Ocean

The screenshot below shows the Blue Ocean dashboard:



Building Pipelines with Blue Ocean


- The Blue Ocean UI helps you set up your Pipeline project, and automatically creates your Pipeline, that is, the Jenkinsfile through the graphical Pipeline editor.
- Jenkins configures a secure and appropriately authenticated connection to your project's source control repository.
- Any changes you make to the Jenkinsfile via Blue Ocean's Pipeline editor are automatically saved and committed to source control.
- Click on the *New Pipeline* button in the Blue Ocean Dashboard to create a new Pipeline.

Building Pipelines with the Classic UI


The screenshot below shows the Classic UI:

Enter an item name


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**


Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Bitbucket Team/Project**

Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

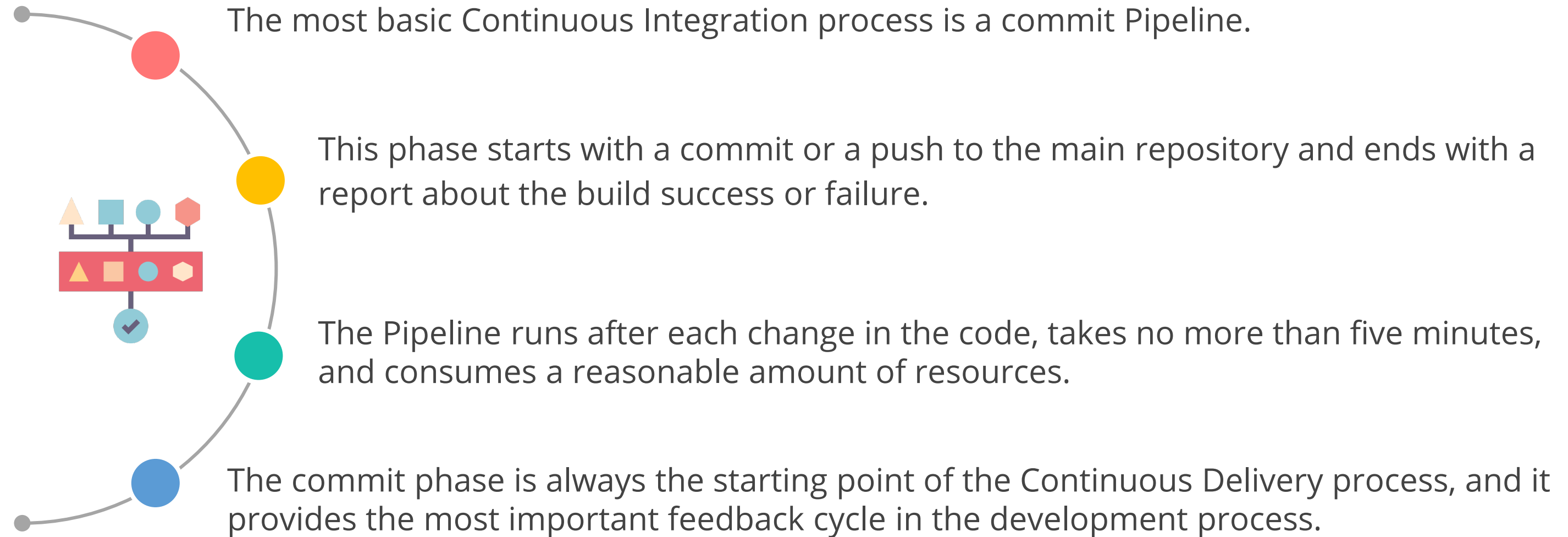
Building Pipelines with the Classic UI

A Jenkinsfile created using the classic UI is stored by Jenkins itself, within the Jenkins home directory. To create a basic Pipeline through the Jenkins classic UI:

- Click *New Item* at the top left in the Jenkins Dashboard.
- Specify the name for your new Pipeline project in the *Enter an item name* field.
- Scroll down and click *Pipeline*, click *OK* at the end of the page to open the Pipeline configuration page.
- Click the *Pipeline* tab at the top of the page to scroll down to the Pipeline section.
- Ensure that the *Definition* field indicates the *Pipeline script* option in the Pipeline section.
- Enter your Pipeline code into the Script text area.
- Click *Save* to open the Pipeline project/item view page.
- Click *Build Now* on the left to run the Pipeline.

Continuous Integration Pipeline

Commit Pipeline



Commit Pipeline

The commit phase works as follow:

- A developer checks in the code to the repository.
- The Continuous Integration server detects the change.
- The build starts.

The commit Pipeline contains three stages:

- Checkout: This stage downloads the source code from the repository.
- Compile: This stage compiles the source code.
- Unit test: This stage runs a suite of unit tests.

Assisted Practice

Continuous Integration Pipelines

Problem Statement: You are given a Java project. Build a Continuous Integration Pipeline in Jenkins for the commit.

Steps to perform:

1. Create a GitHub repository
2. Define a checkout stage
3. Create a Java Spring Boot project
4. Push code to GitHub
5. Define a compile stage
6. Write a unit test
7. Define a unit test stage
8. Run and test the Pipeline

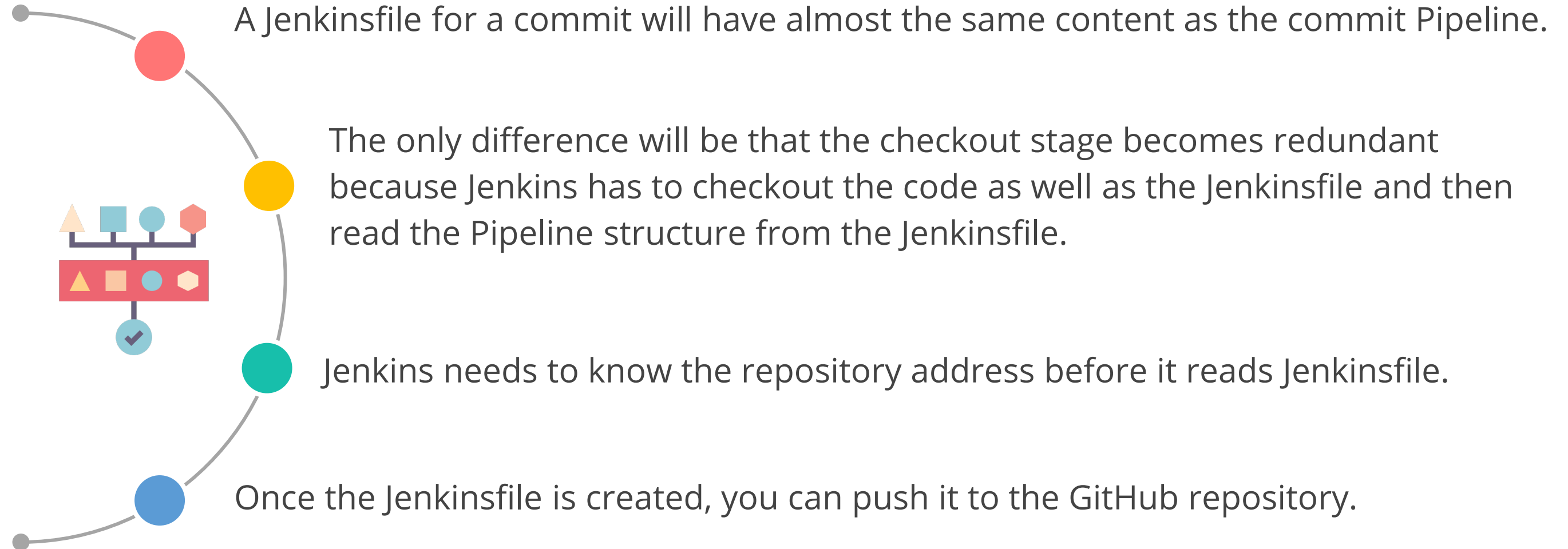
Building Pipelines from Jenkinsfile

Jenkinsfile

- The Pipeline definition in the Jenkinsfile can be committed to the repository along with the source code.
- This method is more consistent as the Pipeline structure becomes strongly tailored to the project.

For example: if you don't need the code compilation because your programming language is interpreted, then you won't have the Compile stage.

Creating Jenkinsfile



Creating Jenkinsfile

Here is a sample Jenkinsfile for a commit Pipeline:

```
Pipeline {  
  agent any  
  stages {  
    stage("Compile") {  
      steps {  
        sh "./gradlew compileJava"  
      }  
    }  
    stage("Unit test") {  
      steps {  
        sh "./gradlew test"  
      }  
    }  
  }  
}
```

Running Pipeline from Jenkinsfile

After the Jenkinsfile is pushed, we can run the Pipeline as given below:

Open the Pipeline configuration.

Change *Definition* from *Pipeline script* to *Pipeline script from SCM*.

Select *Git* in SCM.

Enter Repository URL.

Save the Pipeline.

After saving, the build will always run from the current version of Jenkinsfile in the repository.

Running Pipeline from Jenkinsfile

The screenshot below shows how to run a Pipeline from a Jenkinsfile:

The screenshot displays the 'Pipeline script from SCM' configuration interface in Jenkins. The 'SCM' dropdown is set to 'Git'. Under the 'Repositories' section, the 'Repository URL' is 'https://github.com/leszko/calculator.git' and 'Credentials' is set to '- none -'. There are buttons for 'Advanced...', 'Add Repository', and 'Add'. The 'Branches to build' section has a 'Branch Specifier (blank for 'any')' set to '*/master' and an 'Add Branch' button. The 'Repository browser' is set to '(Auto)'. The 'Additional Behaviours' section has an 'Add' button. At the bottom, the 'Script Path' is set to 'Jenkinsfile'.

Pipeline script from SCM	
SCM	Git
Repositories	<div>Repository URL: <input type="text" value="https://github.com/leszko/calculator.git"/></div> <div>Credentials: <input type="text" value="- none -"/> <button>Add</button></div> <div><button>Advanced...</button></div> <div><button>Add Repository</button></div>
Branches to build	<div>Branch Specifier (blank for 'any'): <input type="text" value="*/master"/></div> <div><button>Add Branch</button></div>
Repository browser	<input type="text" value="(Auto)"/>
Additional Behaviours	<button>Add</button>
Script Path	Jenkinsfile

Build with Jenkinsfile

- The Build stage of the Pipeline will be where source code is assembled, compiled, or packaged.
- The Jenkinsfile is not a replacement for an existing build tool such as GNU/Make, Maven, Gradle, etc.
- It can be considered a glue layer to bind the multiple phases of a project's development lifecycle like build, test, and deploy together.
- Jenkins has plugins for invoking practically any build tool.

Test with Jenkinsfile

- Running automated tests is a crucial component of any successful continuous delivery process.
- Jenkins has a number of test recording, reporting, and visualization facilities provided by a number of plugins.
- The example below uses the JUnit step, provided by the JUnit plugin. If tests fail, the Pipeline is marked ***unstable***.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any

    stages {
        stage('Test') {
            steps {
                /* `make check` returns non-zero on test failures,
                 * using `true` to allow the Pipeline to continue nonetheless
                 */
                sh 'make check || true'
                junit '**/target/*.xml'
            }
        }
    }
}
```

Deploy with Jenkinsfile

- Deployment can be anything from publishing built artifacts to an Artifactory server, to pushing code to a production system.
- The example below shows a Deploy Pipeline. At this stage of the example Pipeline, both the *Build* and *Test* stages have been successfully executed.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any

    stages {
        stage('Deploy') {
            when {
                expression {
                    currentBuild.result == null || currentBuild.result == 'SUCCESS'
                }
            }
            steps {
                sh 'make publish'
            }
        }
    }
}
```

Assisted Practice

Jenkinsfile

Problem Statement: You are given a Java project. Create a Jenkinsfile to build a CI Pipeline.

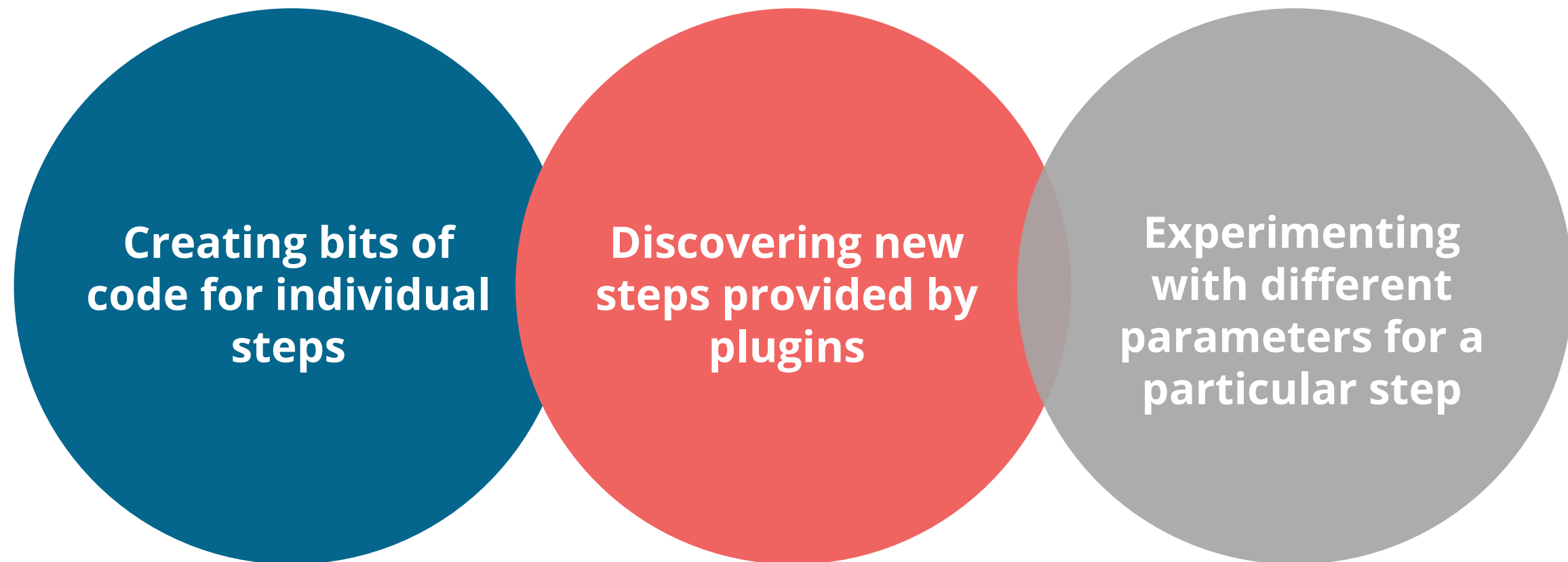
Steps to perform:

1. Create a Maven project
2. Create a Jenkinsfile with stages for checkout, compile, and test
3. Push the code and Jenkinsfile to GitHub
4. Define a Pipeline to build the project

Snippet Generator

Snippet Generator

- The *Snippet Generator* helps create Pipeline script by auto-generating steps.
- The Snippet Generator is dynamically populated with a list of the steps available to the Jenkins instance.
- It is useful for:



Snippet Generator

The screenshot below shows the Snippet Generator:

The screenshot shows the Jenkins Snippet Generator interface. The top navigation bar includes the Jenkins logo, a red tab labeled '1', and a search bar. The left sidebar contains a 'Back' link and a list of links: 'Snippet Generator', 'Declarative Directive Generator', 'Declarative Online Documentation', 'Steps Reference', 'Global Variables Reference', 'Online Documentation', 'Examples Reference', and 'IntelliJ IDEA GDSL'. The main content area is titled 'Overview' and contains an introductory paragraph about the Snippet Generator. Below this is a 'Steps' section with a 'Sample Step' dropdown menu set to 'archiveArtifacts: Archive the artifacts'. A 'Files to archive' text input field is present, followed by an 'Advanced...' button. A 'Generate Pipeline Script' button is located below the input field. At the bottom, a 'Global Variables' section contains a paragraph of text and a link to the 'Global Variables Reference'.

Jenkins

1 search

Jenkins

[Back](#)

- [Snippet Generator](#)
- [Declarative Directive Generator](#)
- [Declarative Online Documentation](#)
- [Steps Reference](#)
- [Global Variables Reference](#)
- [Online Documentation](#)
- [Examples Reference](#)
- [IntelliJ IDEA GDSL](#)

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step archiveArtifacts: Archive the artifacts

Files to archive

[Advanced...](#)

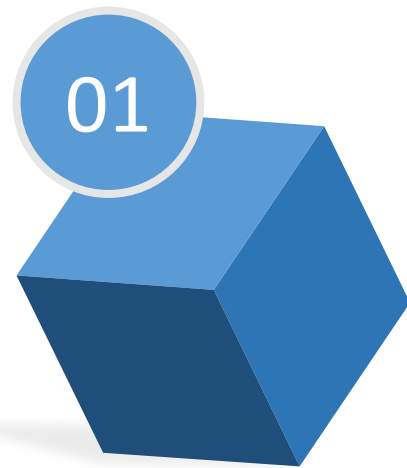
[Generate Pipeline Script](#)

Global Variables

There are many features of the Pipeline that are not steps. These are often exposed via global variables, which are not supported by the snippet generator. See the [Global Variables Reference](#) for details.

Snippet Generator

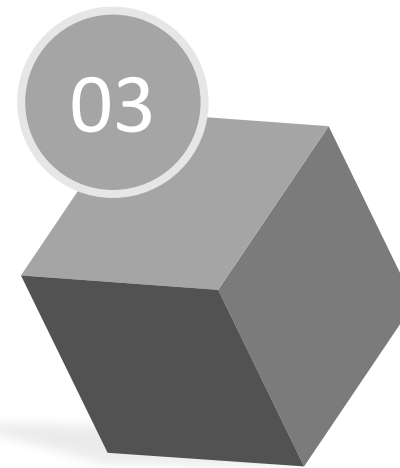
Steps to generate a step snippet with the Snippet Generator:



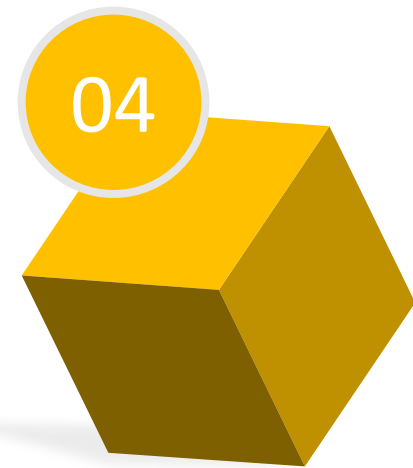
Navigate to the Pipeline Syntax link at `${YOUR_JENKINS_URL}/pipeline-syntax`.



Select the desired step in the *Sample Step* dropdown menu.



Use the dynamically populated area below the *Sample Step* dropdown to configure the selected step.



Click *Generate Pipeline Script* to create a snippet of Pipeline which can be copied and pasted into a Pipeline.

Assisted Practice

Snippet Generator

Problem Statement: Define a Pipeline using the Snippet Generator

Steps to perform:

1. Navigate to `${YOUR_JENKINS_URL}/pipeline-syntax`.
2. Select a step in the Sample Step dropdown menu.
3. Use the dynamically populated area to configure the selected step.
4. Create a snippet of Pipeline.

Global Variable Reference

Global Variable Reference

- Pipeline also provides a built-in *Global Variable Reference*.
- It is also dynamically populated by plugins.
- The Global Variable Reference only contains documentation for variables provided by Pipeline or plugins, which are available for Pipelines.

The built-in global variable reference at **`${YOUR_JENKINS_URL}/pipeline-syntax/globals`** gives a complete list of environment variables available in Pipeline.

Global Variable Reference

The variables provided by default in Pipeline are:

env:

- Environment variables accessible from Scripted Pipeline, for example: env.PATH or env.BUILD_ID.

params:

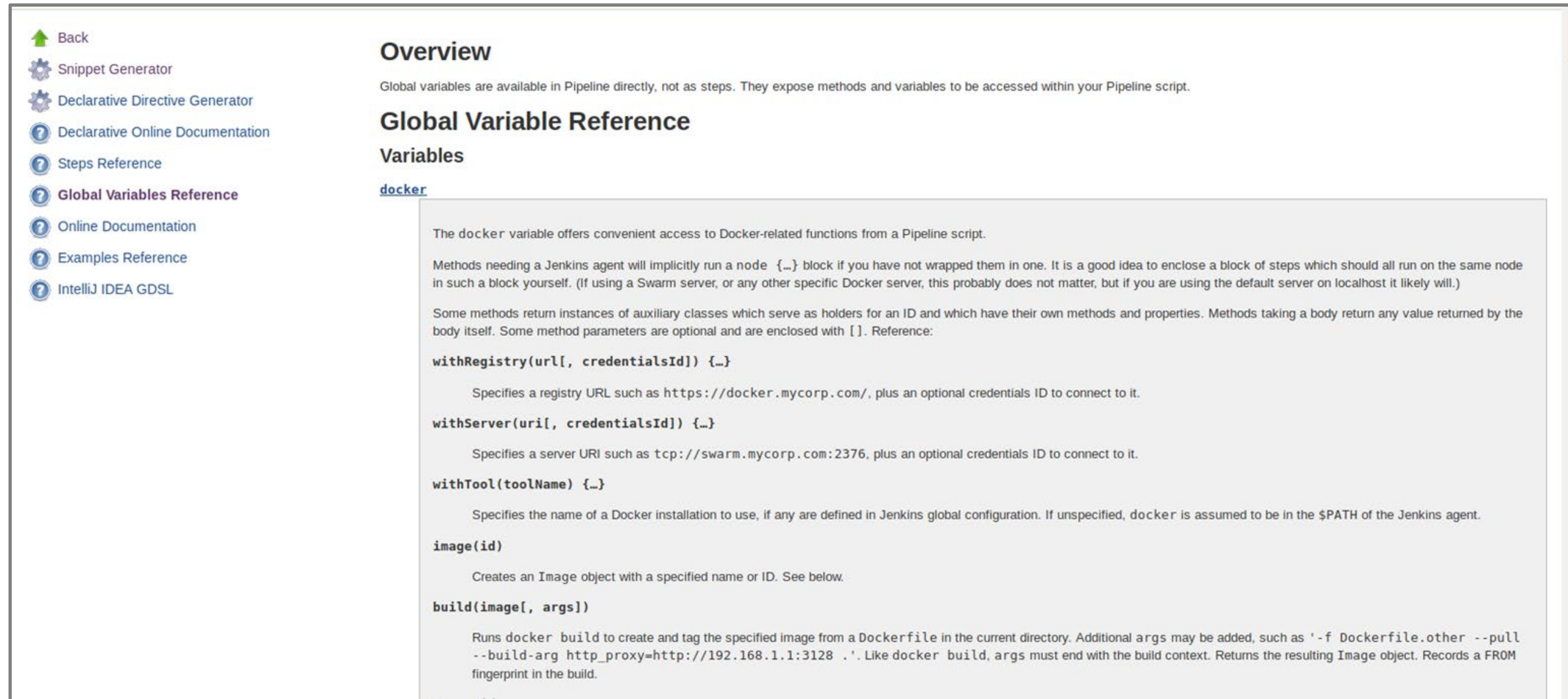
- Exposes all parameters defined for the Pipeline as a read-only Map.
- For example: params.MY_PARAM_NAME.

currentBuild:

- May be used to discover information about the currently executing Pipeline, with properties such as currentBuild.result, currentBuild.displayName, etc.

Global Variable Reference

The screenshot below shows the Global Variable Reference page:



The screenshot displays the Jenkins Global Variable Reference page for the `docker` variable. On the left is a sidebar with navigation links: Back, Snippet Generator, Declarative Directive Generator, Declarative Online Documentation, Steps Reference, Global Variables Reference (highlighted), Online Documentation, Examples Reference, and IntelliJ IDEA GDSL. The main content area is titled "Overview" and "Global Variable Reference Variables". It includes a description of the `docker` variable and its methods: `withRegistry`, `withServer`, `withTool`, `image`, and `build`.

Overview

Global variables are available in Pipeline directly, not as steps. They expose methods and variables to be accessed within your Pipeline script.

Global Variable Reference

Variables

[docker](#)

The `docker` variable offers convenient access to Docker-related functions from a Pipeline script.

Methods needing a Jenkins agent will implicitly run a node `{...}` block if you have not wrapped them in one. It is a good idea to enclose a block of steps which should all run on the same node in such a block yourself. (If using a Swarm server, or any other specific Docker server, this probably does not matter, but if you are using the default server on localhost it likely will.)

Some methods return instances of auxiliary classes which serve as holders for an ID and which have their own methods and properties. Methods taking a body return any value returned by the body itself. Some method parameters are optional and are enclosed with `[]`. Reference:

`withRegistry(url[, credentialsId]) {...}`

Specifies a registry URL such as `https://docker.mycorp.com/`, plus an optional credentials ID to connect to it.

`withServer(uri[, credentialsId]) {...}`

Specifies a server URI such as `tcp://swarm.mycorp.com:2376`, plus an optional credentials ID to connect to it.

`withTool(toolName) {...}`

Specifies the name of a Docker installation to use, if any are defined in Jenkins global configuration. If unspecified, `docker` is assumed to be in the `$PATH` of the Jenkins agent.

`image(id)`

Creates an Image object with a specified name or ID. See below.

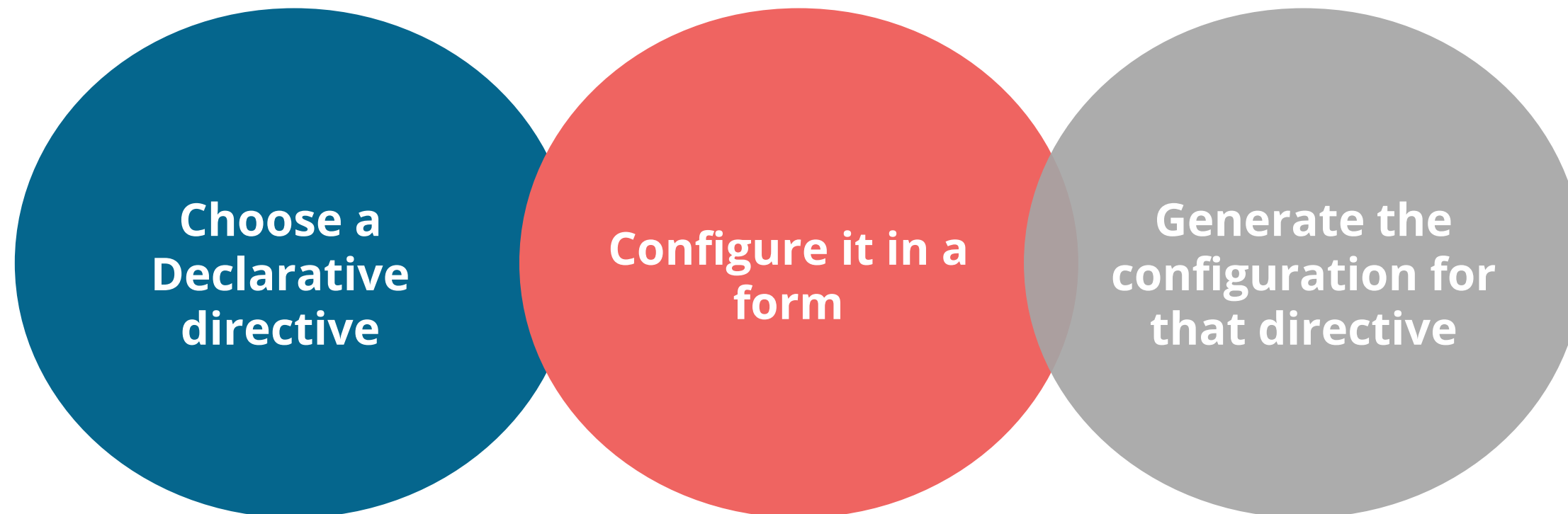
`build(image[, args])`

Runs `docker build` to create and tag the specified image from a Dockerfile in the current directory. Additional args may be added, such as `'-f Dockerfile.other --pull --build-arg http_proxy=http://192.168.1.1:3128 .'`. Like `docker build`, args must end with the build context. Returns the resulting Image object. Records a FROM fingerprint in the build.

Declarative Directive Generator

Declarative Directive Generator

- The Snippet Generator does not cover the *sections* and *directives* used to define a Declarative Pipeline.
- The *Declarative Directive Generator* utility helps generate *sections* and *directives*.
- The Directive Generator allows you to:



- You can then use the generated script in your Declarative Pipeline.

Declarative Directive Generator

The screenshot below shows the Declarative Directive Generator page:

[Back](#)
[Snippet Generator](#)
[Declarative Directive Generator](#)
[Declarative Online Documentation](#)
[Steps Reference](#)
[Global Variables Reference](#)
[Online Documentation](#)
[Examples Reference](#)
[IntelliJ IDEA GDSDL](#)

Overview

The **Directive Generator** allows you to generate the Pipeline code for a Declarative Pipeline directive, such as agent, options, when, and more. Choose the directive you're interested in from the dropdown, and then choose the contents of the directive from the new form. Once you've filled out the form with the choices and values you want for your directive, click **Generate Declarative Directive** and the Pipeline code will appear in the box below. You can copy that code directly into the pipeline block in your Jenkinsfile, for top-level directives, or into a stage block for stage directives. See [the online syntax documentation](#) for more information on directives.

Directives

Sample Directive

agent: Agent

See [the online documentation](#) for more information on the agent directive.

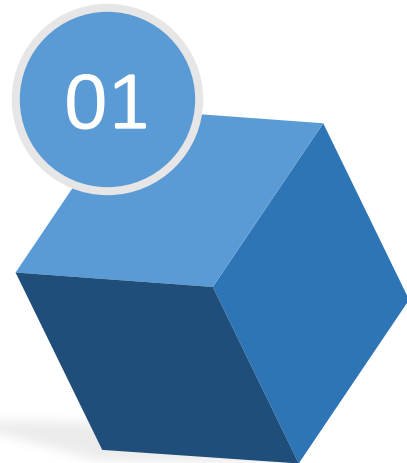
Agent

any: Run on any agent

Generate Declarative Directive

Declarative Directive Generator

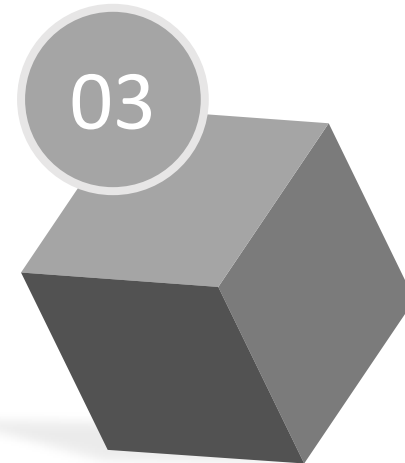
Steps to generate a Declarative directive using the Declarative Directive Generator:



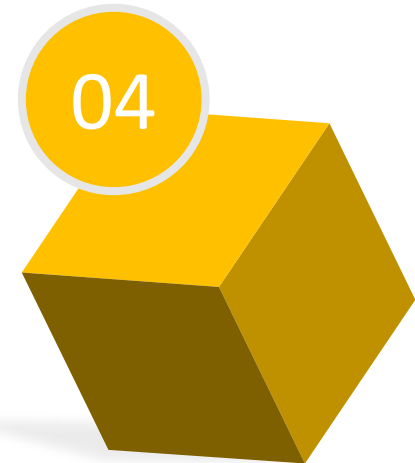
Navigate to the Pipeline Syntax link at `${YOUR_JENKINS_URL}/declarative-generator`.



Select the desired directive in the dropdown menu.



Use the dynamically populated area to configure the selected directive.



Click Generate Directive to create the directive's configuration to copy into your Pipeline.

Restarting a Pipeline

There are multiple ways to rerun or restart a completed Pipeline:

Replay:

- The Replay feature allows for quick modifications and execution of an existing Pipeline without changing the Pipeline configuration or creating a new commit.

Restart from a stage:

- You can restart any completed Declarative Pipeline from any top-level stage which ran in the Pipeline. This allows you to rerun a Pipeline from a stage which failed due to transient or environmental considerations.
- All inputs to the Pipeline, such as SCM information or build parameters will be the same.

Restarting a Pipeline

The screenshot below shows the *Replay* option:



Assisted Practice

Restarting a Pipeline

Problem Statement: You have been asked to restart two Pipelines:

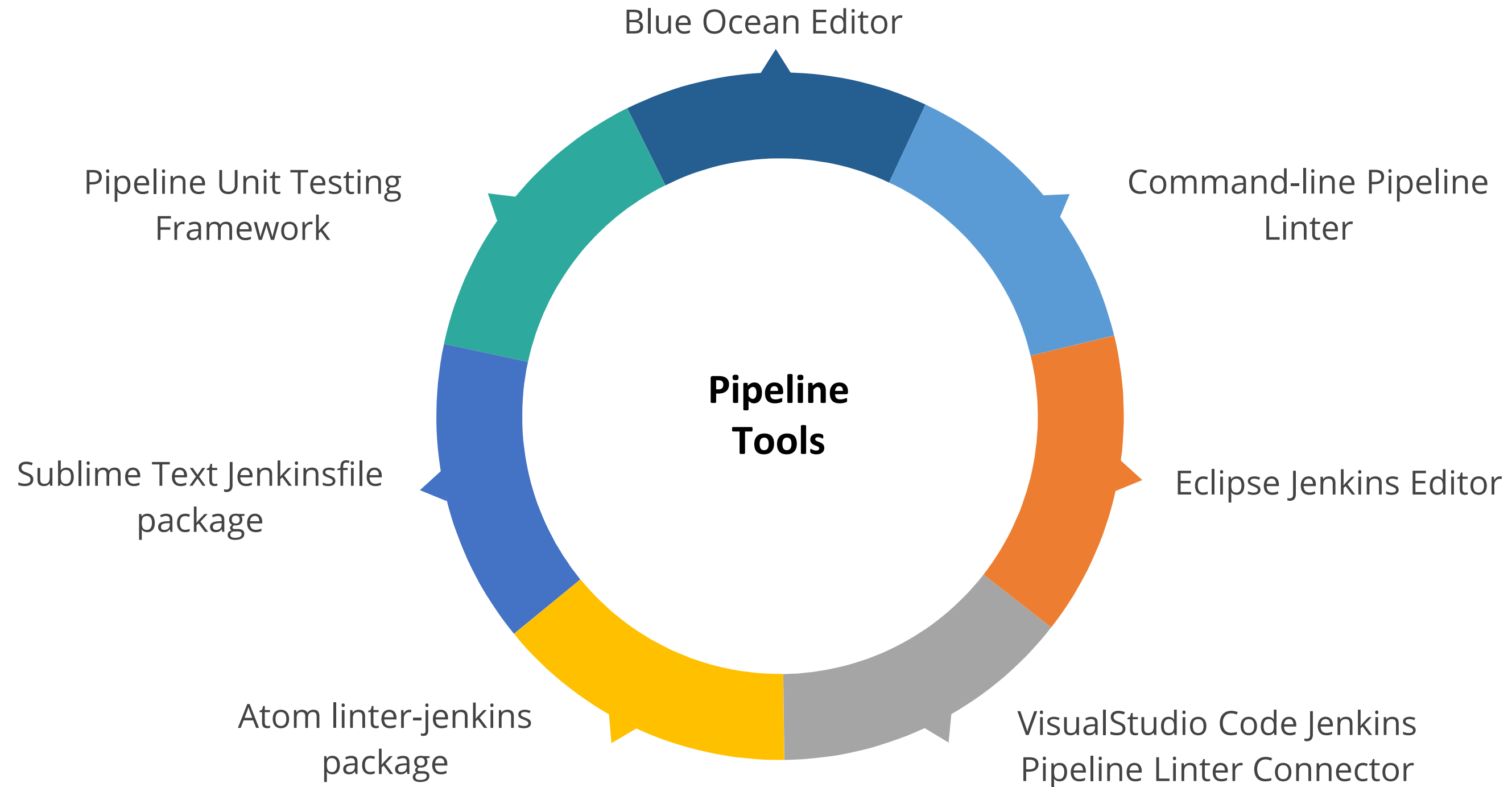
- one with some modifications
- one with no modifications

Steps to perform:

1. Use the Replay option in a previously completed run from the build history to restart a build with some modifications.
2. Use the Restart from Stage option in the Blue Ocean UI to restart a build with no modifications.

Pipeline Development Tools

There are many tools and resources that help with development of Jenkins Pipelines. A few are:





Knowledge Check

Knowledge Check

1

Which of the following is NOT a difference between build jobs and Pipelines?

- A. Pipelines are defined using script whereas jobs are defined via the UI.
- B. Freestyle jobs do not create a persistent record of execution whereas Pipelines do.
- C. Pipelines let you store the build results but build jobs do not.
- D. Pipelines let you break jobs into different stages whereas build jobs do not.



Knowledge Check

1

Which of the following is NOT a difference between build jobs and Pipelines?

- A. Pipelines are defined using script whereas jobs are defined via the UI.
- B. Freestyle jobs do not create a persistent record of execution whereas Pipelines do.
- C. Pipelines let you store the build results but build jobs do not.
- D. Pipelines let you break jobs into different stages whereas build jobs do not.



The correct answer is **C**

Pipelines and build jobs both let you store the generated artifacts and archive the test results.

Knowledge Check

2

Which of the following statement defines a node?

- A. A node is a single task in the Pipeline.
- B. A node refers to a distinct subset of tasks performed through the entire Pipeline.
- C. A node is a user-defined model of a Pipeline.
- D. A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.



Knowledge Check

2

Which of the following statement defines a node?

- A. A node is a single task in the Pipeline.
- B. A node refers to a distinct subset of tasks performed through the entire Pipeline.
- C. A node is a user-defined model of a Pipeline.
- D. A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.



The correct answer is **D**

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.

Knowledge Check

3

Scripted Pipelines and Declarative Pipelines are both constructed using a Groovy syntax. Is the statement true or false?

- A. True
- B. False



Knowledge Check

3

Scripted Pipelines and Declarative Pipelines are both constructed using a Groovy syntax. Is the statement true or false?

- A. True
- B. False



The correct answer is **B**

Scripted Pipeline is written in a limited form of Groovy syntax. Declarative Pipelines are constructed differently.

Knowledge Check

4

Which of the following methods should you use to run a Pipeline without a new commit but with some modifications?

- A. Replay
- B. Restart from a stage



Knowledge Check

4

Which of the following methods should you use to run a Pipeline without a new commit but with some modifications?

- A. Replay
- B. Restart from a stage



The correct answer is **A**

The Replay feature allows for quick modifications and execution of an existing Pipeline without changing the Pipeline configuration or creating a new commit.

Key Takeaways

- Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery Pipelines into Jenkins.
- A Pipeline has two main elements: Steps and Stages
- A Jenkinsfile can be written using two types of syntax: Declarative and Scripted.
- A Pipeline can be built from the Jenkinsfile in the UI or from the committed Jenkinsfile in the SCM.



Lesson-End Project

Multistage Pipeline

Problem Statement:

You're a DevOps engineer at BookMyEvents, a web- and mobile-based app development company that helps users buy tickets for various events in their cities. The company is moving from a call center-based customer support to a chat and email-based customer support system. The backend developers are developing a Maven project for the new ticketing system which generates a random ticket number for each registered complaint. You're required to set up a Jenkins pipeline to compile and test the ticket generator. The job has to read the code and the Jenkinsfile from the SCM and then compile and test the code.

Requirements:

- The pipeline should be defined as a Jenkinsfile and committed to the SCM.
- The pipeline should compile and test the code.
- The program should generate a ticket number higher than 1000.

