

# Jenkins Pipeline



# Jenkins Pipeline

Jenkins Pipeline is a suite of **plugins** which supports implementing and integrating continuous delivery pipelines into Jenkins.

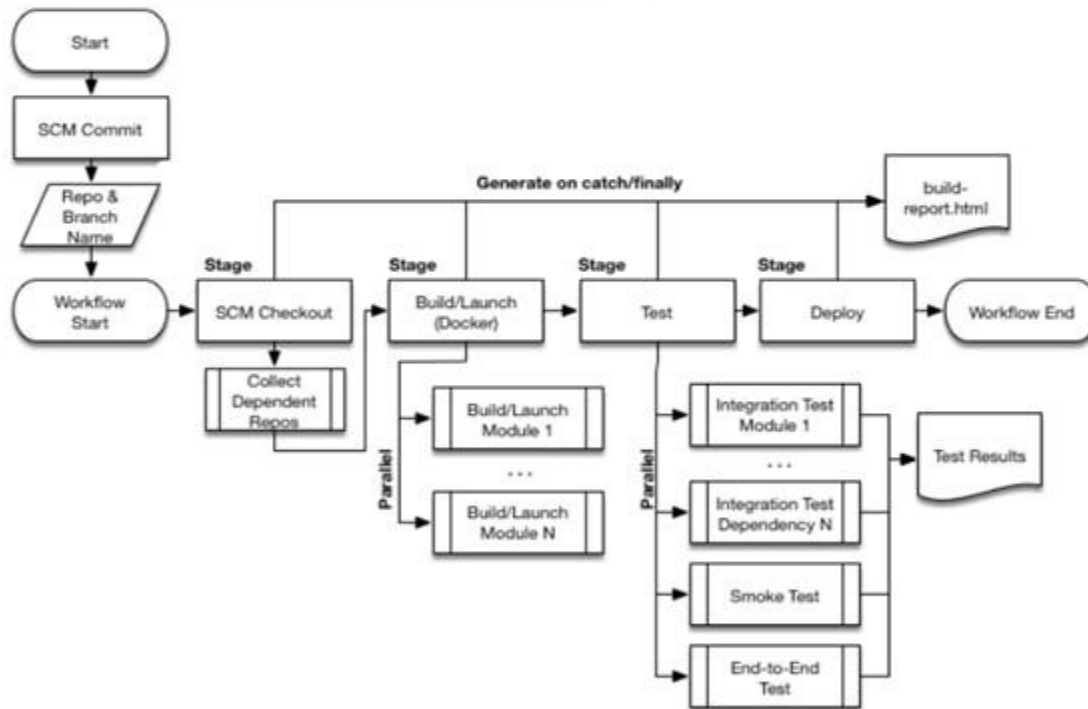
The definition of a Jenkins Pipeline is written into a text file (called a **Jenkinsfile**) which in turn can be committed to a project's **source control repository**. This is the foundation of "**Pipeline-as-code**"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

## Benefits of Jenkins Pipeline

Creating a Jenkinsfile and committing it to source control provides a number of immediate benefits:

- Automatically creates a Pipeline build process for all branches and pull requests.
- Code review/iteration on the Pipeline (along with the remaining source code).
- Audit trail for the Pipeline.
- Single source of truth [3] for the Pipeline, which can be viewed and edited by multiple members of the project.

# Pipeline Flowchart



## Types of Pipeline

A Jenkinsfile can be written using two types of syntax

- Declarative Pipeline (Recent)
- Scripted Pipeline



## Pipeline Concepts

**Pipeline** – This defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.

Also, a pipeline block is a key part of **Declarative Pipeline syntax**.

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

Also, a node block is a key part of **Scripted Pipeline syntax**.

## Pipeline Concepts

**Stage** - A stage block defines a conceptually distinct subset of tasks performed through the entire Pipeline (e.g. "Build", "Test" and "Deploy" stages)

**Step** - A single task. Fundamentally, a step tells Jenkins what to do at a particular point in time (or "step" in the process). For example, to execute the shell command make use the sh step: sh 'make'.

# Scripted Pipeline

*Jenkinsfile (Scripted Pipeline)*

```
node { ❶  
    stage('Build') { ❷  
        // ❸  
    }  
    stage('Test') { ❹  
        // ❺  
    }  
    stage('Deploy') { ❻  
        // ❼  
    }  
}
```

- ❶ Execute this Pipeline or any of its stages, on any available agent.  
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.
- ❷ Defines the "Test" stage.
- ❸ Perform some steps related to the "Build" stage.
- ❹ Defines the "Deploy" stage.
- ❺ Perform some steps related to the "Test" stage.
- ❻ Defines the "Deploy" stage.
- ❼ Perform some steps related to the "Deploy" stage.



# Declarative Pipeline

*Jenkinsfile (Declarative Pipeline)*

```
pipeline {  
    agent any ❶  
    stages {  
        stage('Build') { ❷  
            steps {  
                // ❸  
            }  
        }  
        stage('Test') { ❹  
            steps {  
                // ❺  
            }  
        }  
        stage('Deploy') { ❻  
            steps {  
                // ❼  
            }  
        }  
    }  
}
```

- ❶ Execute this Pipeline or any of its stages, on any available agent.
- ❷ Defines the "Build" stage.
- ❸ Perform some steps related to the "Build" stage.
- ❹ Defines the "Test" stage.
- ❺ Perform some steps related to the "Test" stage.
- ❻ Defines the "Deploy" stage.
- ❼ Perform some steps related to the "Deploy" stage.

# Pipeline Job

# Pipeline Job


- From the Jenkins home page, click New Item at the top left.




## Pipeline Job

- In the Enter an item name field, specify the name for your new Pipeline project.
- Scroll down and click Pipeline, then click OK


**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.



**GitHub Organization**  
Scans a GitHub organization (or user account) for all repositories matching some defined markers.



**Multibranch Pipeline**  
Creates a set of Pipeline projects automatically in detected branches in one SCM repository.





## Pipeline Job

- Scroll down to the Pipeline section.
- Select “Pipeline script” under “Definition” field.
- Enter your Pipeline code into the Script text area.

```
pipeline {  
  agent any 1  
  stages {  
    stage('Stage 1') {  
      steps {  
        echo 'Hello world!' 2  
      }  
    }  
  }  
}
```

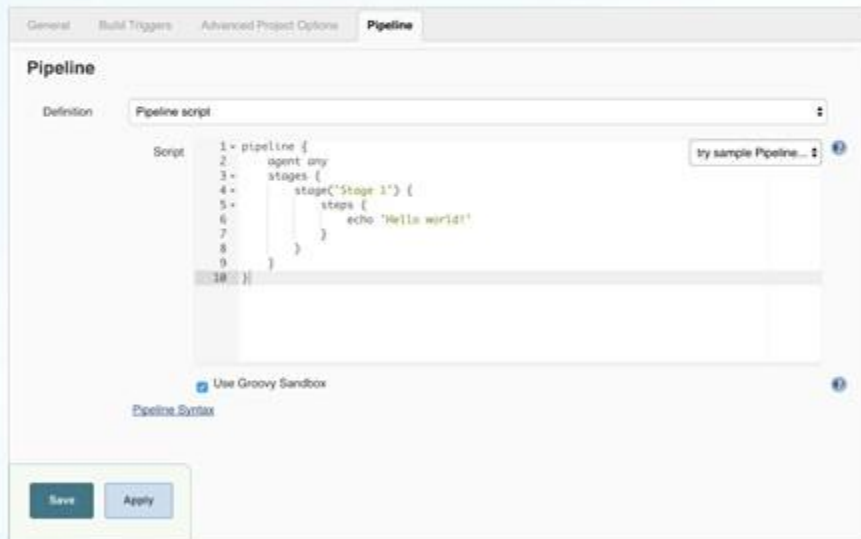
`agent` instructs Jenkins to allocate an executor (on any available

**1** `agent/node` in the Jenkins environment) and workspace for the entire Pipeline.

**2** `echo` writes simple string in the console output.

# Pipeline Job

- Click Save to open the Pipeline project/item view page.
- On this page, click Build Now on the left to run the Pipeline.



General Build Triggers Advanced Project Options **Pipeline**

Pipeline

Definition Pipeline script

Script

```
1- pipeline {
2-   agent any
3-   stages {
4-     stage('Stage 1') {
5-       steps {
6-         echo 'Hello world!'
7-       }
8-     }
9-   }
10- }
```

try sample Pipeline...

Use Groovy Sandbox

Pipeline Syntax

Save Apply



## Console Output

```
Started by user Alex
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/example-pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Stage 1)
[Pipeline] echo
Hello world!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

# Pipeline Job - SCM

## Pipeline Job - SCM

- Complex Pipelines are difficult to write and maintain.
- You can write your pipeline in **Jenkinsfile** using text editor or IDE.
- Commit your Jenkinsfile to SCM.
- Jenkins check out your Jenkinsfile from SCM as part of your Pipeline project's build process.
- Then it proceed to execute your Pipeline.



## Pipeline Job - SCM

### Create a project in GitHub and check-in Jenkinsfile

- Follow the procedure of creating pipeline job.
- From the Definition field, choose the “Pipeline script from SCM”
- From the SCM field, choose the type of source control system of the repository containing your Jenkinsfile.
- Complete the fields specific to your repository’s source control system.
- In the Script Path field, specify the location (and name) of your Jenkinsfile.

# Pipeline Job - SCM

## Pipeline

Definition Pipeline script from SCM

SCM Git

Repositories

Repository URL https://github.com/prabhavagrawal/jenkins.git

Credentials prabhavagrawal/\*\*\*\*\* Add

Advanced...

Add Repository

Branches to build

Branch Specifier (blank for 'any') \*/master Add Branch

Repository browser (Auto)

Additional Behaviours Add

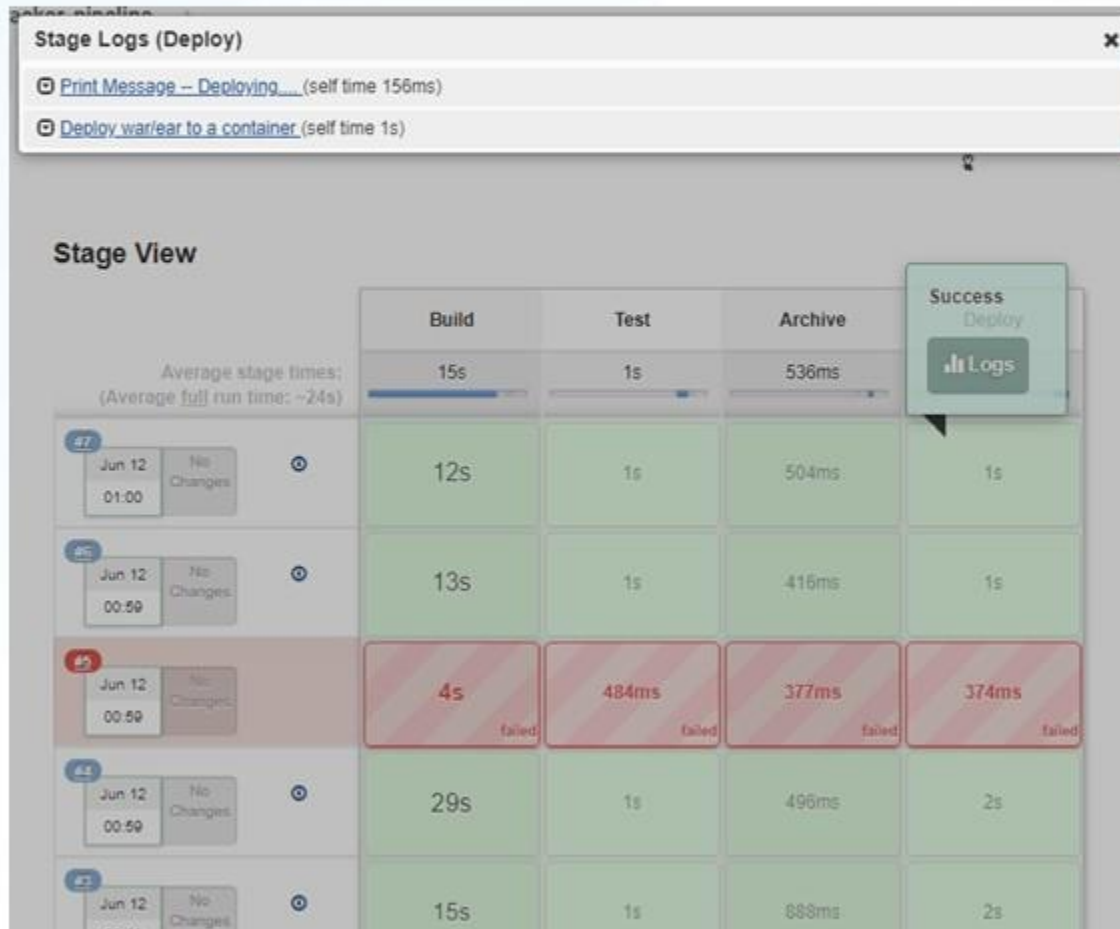
Script Path Jenkinsfile

Lightweight checkout ☒

[Pipeline Syntax](#)

# Pipeline Job - Extras

# Stage View





# Pipeline Job - Snippet Generator

- Navigate to the **Pipeline Syntax** link (referenced above) from a configured Pipeline, or 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.

## 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

Stage Name

## Generate Pipeline Script

```
stage('Deploy') {  
    // some block  
}
```

# Global Variable Reference

- Navigate to the **Global Variable Reference** link at **`${YOUR_JENKINS_URL}/pipeline-syntax/globals`**

Environment variables are accessible from Groovy code as `env.VARNAME` or simply as `VARNAME`. You can write to such properties as well (only using the `env.` prefix):

```
env.MYTOOL_VERSION = '1.33'
node {
  sh '/usr/local/mytool-${MYTOOL_VERSION}/bin/start'
}
```

These definitions will also be available via the REST API during the build or after its completion, and from upstream Pipeline builds using the `build` step.

However any variables set this way are global to the Pipeline build. For variables with node-specific content (such as file paths), you should instead use the `withEnv` step, to bind the variable only within a node block.

A set of environment variables are made available to all Jenkins projects, including Pipelines. The following is a general list of variable names that are available.

## BRANCH\_NAME

For a multibranch project, this will be set to the name of the branch being built, for example in case you wish to deploy to production from `master` but not from feature branches; if corresponding to some kind of change request, the name is generally arbitrary (refer to `CHANGE_ID` and `CHANGE_TARGET`).

## CHANGE\_ID

For a multibranch project corresponding to some kind of change request, this will be set to the change ID, such as a pull request number, if supported; else unset.

## CHANGE\_URL

For a multibranch project corresponding to some kind of change request, this will be set to the change URL, if supported; else unset.

## CHANGE\_TITLE

For a multibranch project corresponding to some kind of change request, this will be set to the title of the change, if supported; else unset.

## CHANGE\_AUTHOR

For a multibranch project corresponding to some kind of change request, this will be set to the username of the author of the proposed change, if supported; else unset.

## CHANGE\_AUTHOR\_DISPLAY\_NAME

For a multibranch project corresponding to some kind of change request, this will be set to the human name of the author, if supported; else unset.

## CHANGE\_AUTHOR\_EMAIL

For a multibranch project corresponding to some kind of change request, this will be set to the email address of the author, if supported; else unset.

## CHANGE\_TARGET

For a multibranch project corresponding to some kind of change request, this will be set to the target or base branch to which the change could be merged, if supported; else unset.

## BUILD\_NUMBER

The current build number, such as "153"

## BUILD\_ID

# Using Jenkinsfile

# Jenkinsfile

## String interpolation

### Input

```
def username = 'Jenkins'  
echo 'Hello Mr. ${username}'  
echo "I said, Hello Mr. ${username}"
```

### Output

```
Hello Mr. ${username}  
I said, Hello Mr. Jenkins
```



# Jenkinsfile

## Using environment variables

```
steps {  
    echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"  
}
```

# Jenkinsfile

## Using environment variables

```
pipeline {  
  agent any  
  environment { ❶  
    CC = 'clang'  
  }  
  stages {  
    stage('Example') {  
      environment { ❷  
        DEBUG_FLAGS = '-g'  
      }  
      steps {  
        sh 'printenv'  
      }  
    }  
  }  
}
```

- ❶ An `environment` directive used in the top-level `pipeline` block will apply to all steps within the Pipeline.
- ❷ An `environment` directive defined within a `stage` will only apply the given environment variables to steps within the `stage`.

# Jenkinsfile

## Handling credentials - Secret text

```
pipeline {
  agent {
    // Define agent details here
  }
  environment {
    AWS_ACCESS_KEY_ID      = credentials('jenkins-aws-secret-key-id')
    AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
  }
  stages {
    stage('Example stage 1') {
      steps {
        // ❶
      }
    }
    stage('Example stage 2') {
      steps {
        // ❷
      }
    }
  }
}
```

1. To maintain the security and anonymity of these credentials, if the job displays the value of these credential variables from within the Pipeline (e.g. `echo $AWS_SECRET_ACCESS_KEY`), Jenkins only returns the value `*****` to reduce the risk of secret information being disclosed to the console output and any logs.
2. Environment variables are scoped globally for the entire Pipeline. This is not recommended when these variables are required only in a stage. In that case define credentials as stage variables

# Jenkinsfile

## Handling credentials - Usernames and passwords

```
pipeline {
  agent {
    // Define agent details here
  }
  stages {
    stage('Example stage 1') {
      environment {
        BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
      }
      steps {
        // ❶
      }
    }
    stage('Example stage 2') {
      steps {
        // ❷
      }
    }
  }
}
```

# Jenkinsfile

## Handling credentials - Secret files

```
pipeline {
  agent {
    // Define agent details here
  }
  environment {
    // The MY_KUBECONFIG environment variable will be assigned
    // the value of a temporary file. For example:
    // /home/user/.jenkins/workspace/cred_test@tmp/secretFiles/546a5cf3-9b56-4165-a0fd-19e2afe6b31f/kubeconfig.txt
    MY_KUBECONFIG = credentials('my-kubeconfig')
  }
  stages {
    stage('Example stage 1') {
      steps {
        sh("kubectl --kubeconfig $MY_KUBECONFIG get pods")
      }
    }
  }
}
```



# Jenkinsfile

## Handling credentials - SSH

```
withCredentials(bindings: [sshUserPrivateKey(credentialsId: 'jenkins-ssh-key-for-abc', \
                                          keyFileVariable: 'SSH_KEY_FOR_ABC', \
                                          passphraseVariable: '', \
                                          usernameVariable: '')[0]]) {

    // some block
}
```

# Handling failure

Declarative Pipeline supports robust failure handling by default via its post section which allows declaring a number of different "post conditions" such as: **always**, **unstable**, **success**, **failure**, and **changed**.

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh 'make check'
            }
        }
    }
    post {
        always {
            junit '**/target/*.xml'
        }
        failure {
            mail to: team@example.com, subject: 'The Pipeline failed :('
        }
    }
}
```

# Jenkinsfile

## Handling parameters

*Jenkinsfile (Declarative Pipeline)*

```
pipeline {
  agent any
  parameters {
    string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I greet the world?')
  }
  stages {
    stage('Example') {
      steps {
        echo "${params.Greeting} World!"
      }
    }
  }
}
```

# Jenkinsfile

## Using multiple agents

```
pipeline {
  agent none
  stages {
    stage('Build') {
      agent any
      steps {
        checkout scm
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app' ❶
      }
    }
    stage('Test on Linux') {
      agent { ❷
        label 'linux'
      }
      steps {
        unstash 'app' ❸
        sh 'make check'
      }
      post {
        always {
          junit '**/target/*.xml'
        }
      }
    }
  }
}
```

# Jenkinsfile

## Parallel execution

Pipeline has built-in functionality for executing portions of Scripted Pipeline in parallel, implemented in the aptly named **parallel** step.

```
stage('Build') {
    /* .. snip .. */
}




stage('Test') {
    parallel linux: {
        node('linux') {
            checkout scm
            try {
                unstash 'app'
                sh 'make check'
            }
            finally {
                junit '**/target/*.xml'
            }
        }
    },
    windows: {
        node('windows') {
            /* .. snip .. */
        }
    }
}
```



# Restart & Replay

# Restart

- You can restart any completed Declarative Pipeline from any top-level stage which ran in that Pipeline.
- Rerun a Pipeline from a stage which failed.
- All inputs to the Pipeline will be the same. This includes SCM information, build parameters, etc.
- **Restarting from UI** - Once your Pipeline has completed, whether it succeeds or fails, you can go to the side panel for the run in the classic UI and click on "Restart from Stage".
- You will be prompted to choose from a list of top-level stages

-  Back to Project
-  Status
-  Changes
-  Console Output
-  Edit Build Information
-  Delete Build
-  Restart from Stage
-  Replay
-  Pipeline Steps

## Restart #1 from Stage

Stage Name

Run

# Replay

- The "Replay" feature allows for quick modifications and execution of an existing Pipeline without changing the Pipeline configuration or creating a new commit.
- Select a previously completed run in the build history.
- Click "Replay" in the left menu
- Make modifications and click "Run". In this example, we changed "ruby-2.3" to "ruby-2.4".
- Check the results of changes

-  Back to Project
-  Status
-  Changes
-  Console Output
-  Edit Build Information
-  Delete Build
-  Restart from Stage
-  **Replay**
-  Pipeline Steps

declarative.html #4 Replay

## Replay #4

Allows you to replay a Pipeline build with a modified script. If any load steps were run, you can also modify the scripts they loaded.

```
Main Script 1 #!groovy
2
3 pipeline {
4   agent {
5     // Use docker container
6     docker {
7       image 'ruby:2.4'
8     }
9   }
10  options {
11    // Only keep the 10 most recent builds
12    buildDiscarder(logRotator(numToKeepSters: '10'))
13  }
14  stages {
15    stage ('Install') {
```

[Pipeline Syntax](#)

Run

# Thank You