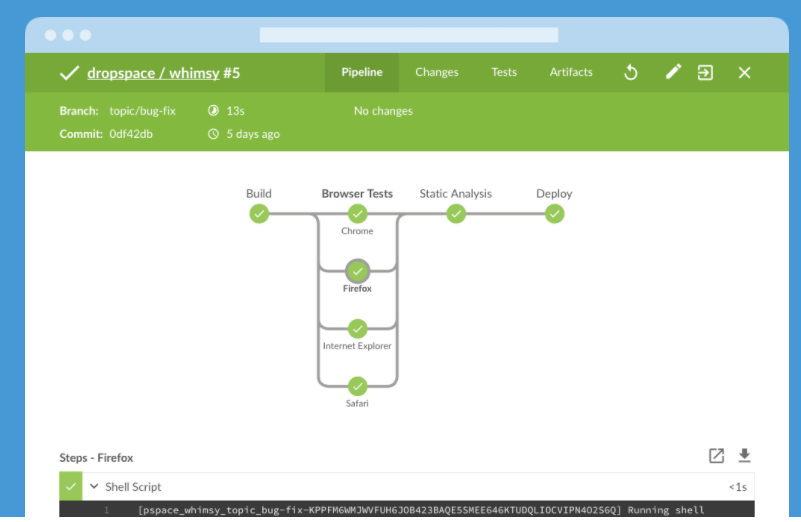
## What is Jenkins?

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed



##### **Continuous Integration and Continuous Delivery**

As an extensible automation server, Jenkins can be used as a simple CI server or turned into the continuous delivery hub for any project.

##### **Easy installation**

Jenkins is a self-contained Java-based program, ready to run out-of-the-box, with packages for Windows, Mac OS X and other Unix-like operating systems.

##### **Easy configuration**

Jenkins can be easily set up and configured via its web interface, which includes on-the-fly error checks and built-in help.

##### **Plugins**

With hundreds of plugins in the Update Center, Jenkins integrates with practically every tool in the continuous integration and continuous delivery toolchain.

##### **Extensible**

Jenkins can be extended via its plugin architecture, providing nearly infinite possibilities for what Jenkins can do.

##### **Distributed**

Jenkins can easily distribute work across multiple machines, helping drive builds, tests and deployments across multiple platforms faster.

Getting started with Jenkins

The Jenkins project produces two release lines, LTS and weekly. Depending on your organization's needs, one may be preferred over the other.

Both release lines are distributed as .war files, native packages, installers, and Docker containers. Packages with the  gear icon are maintained by third parties.

#### Long-term Support (LTS)

LTS (Long-Term Support) releases are chosen every 12 weeks from the stream of regular releases as the stable release for that time period.

#### Weekly

A new release is produced weekly to deliver bug fixes and features to users and plugin developers.

# LTS Release Line

The weekly Jenkins releases deliver bug fixes and new features rapidly to users and plugin developers who need them. But for more conservative users, it’s preferable to stick to a release line which changes less often and only receives important bug fixes, even if such a release line lags behind in terms of features. Several companies maintain their own private branches off of Jenkins for stabilization and internal customizations. We encourage everybody to shift a part of the effort to this release line.

## Model

Every 12 weeks, the community will pick one of the relatively recent releases by consensus and mark it as the baseline for the next "stable but older" release line. Let’s say this was version X. We’ll create a branch from X to produce stable but older patch releases of **X.1**, **X.2**, and **X.3**. Changes to this branch will be restricted to backported cherry-picked bug fixes from the trunk that are "battle-tested" — meaning those commits that have already been a part of a main line release for more than a week. There are 3 minor releases for a baseline published in four week cycles. A release candidate is published two weeks before a minor release.

The following table demonstrates release dates within the 12 week cycle:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Week** | **0** | **2** | **4** | **6** | **8** | **10** | **12** | **14** | **16** | **18** | **20** | **22** | **24** |
| **Release** | W.3 | X.1 RC | X.1 | X.2 RC | X.2 | X.3 RC | X.3 | Y.1 RC | Y.1 | Y.2 RC | Y.2 | Y.3 RC | Y.3 |
| **Baseline Selection** | X chosen |  |  |  |  |  | Y chosen |  |  |  |  |  | Z chosen |

The cycle starts with picking an LTS baseline at week 0. Then, there is a two week period for backporting followed by two weeks for testing the release candidate resulting in the release of X.1. Backporting and RC testing is repeated twice, producing X.2 and X.3. This concludes the cycle for a given baseline and the new one is started immediately.

The baseline release is typically between 2-5 weeks old when it is chosen, so X.1 LTS releases are published about 6-9 weeks after their baseline.

See the event calendar for the specific LTS RC/release dates in the near future.

## Backporting Process

Any user can propose that a bug fix be backported to LTS by labeling with lts-candidate. Backporters use this query to list up issues that need to be attended once resolved.

Aside from the model set out above, backporters apply some subjective selection — for example whether a fix is easy and safe to backport, confidence in the fix, importance/impact of the problem, how much time is left until the end of backporting window and so on.

If backported, a label like 2.46.2-fixed is applied to communicate to the user what LTS version(s) it’s going to be in. If the backport is rejected, a label like 2.46.2-rejected is used to indicate that this ticket will not be backported to that specific release — but it may still be picked up in a later LTS release.

## Switching Between Release Lines

Due to how Jenkins stores data internally, users are generally able to upgrade to newer releases, but not downgrade to older releases. In the context of LTS, the baseline is almost always the deciding factor that determines to which releases of the other line a Jenkins instance can be migrated.

### Switching From LTS to Weekly

Make sure that the weekly release you’re migrating to has been released after the LTS version you’re migrating from.

While almost always any weekly more recent than the LTS baseline release will be compatible, important fixes that are in the LTS release, such as security fixes, may not be present in older weekly releases.

### Switching From Weekly to LTS

Make sure that the LTS baseline you’re migrating to is more recent (compared numerically) than the weekly release you’re migrating from. For example, if you’re using Jenkins 2.5, 2.18, or 2.46, you will be able to upgrade to Jenkins LTS 2.46.x without major problems. However, if you’re using Jenkins 2.47 or Jenkins 2.56, you may have problems downgrading to Jenkins LTS 2.46.x, even if the specific LTS releases were created at a later date than the weekly release you’re using.

### Switching Update Sites

The Jenkins project operates multiple update sites that inform Jenkins about available updates to Jenkins and plugins. As Jenkins sends its current version when requesting new data, the same URL can be used for all releases of Jenkins, and will always serve the most appropriate update information. For example, instances running an LTS release of Jenkins will only be offered LTS versions of Jenkins to upgrade to.

After switching between release lines, it’s recommend to update the update site metadata cached in Jenkins by hitting the "Check now" button in the Plugins Manager (further instructions here). Otherwise, Jenkins may offer to update or install plugins whose newer versions are incompatible with the installed version.

Improve this page | Page history

# Installing Jenkins

The procedures on this page are for new installations of Jenkins on a single/local machine

Jenkins is typically run as a standalone application in its own process with the built-in Java servlet container/application server (Jetty).

Jenkins can also be run as a servlet in different Java servlet containers such as Apache Tomcat or GlassFish. However, instructions for setting up these types of installations are beyond the scope of this page.

**Note:** Although this page focuses on local installations of Jenkins, this content can also be used to help set up Jenkins in production environments.

## Prerequisites

## Prerequisites

Minimum hardware requirements:

* 256 MB of RAM
* 1 GB of drive space (although 10 GB is a recommended minimum if running Jenkins as a Docker container)

Recommended hardware configuration for a small team:

* 1 GB+ of RAM
* 50 GB+ of drive space

Sofware requirements:

* Java 8 - either a Java Runtime Environment (JRE) or a Java Development Kit (JDK) is fine  
  **Note:** This is not a requirement if running Jenkins as a Docker container.

## Installation platforms

This section describes how to install/run Jenkins on different platforms and operating systems.

### Docker

Docker is a platform for running applications in an isolated environment called a "container" (or Docker container). Applications like Jenkins can be downloaded as read-only "images" (or Docker images), each of which is run in Docker as a container. A Docker container is in effect a "running instance" of a Docker image. From this perspective, an image is stored permanently more or less (i.e. insofar as image updates are published), whereas containers are stored temporarily. Read more about these concepts in the Docker documentation’s Getting Started, Part 1: Orientation and setup page.

Docker’s fundamental platform and container design means that a single Docker image (for any given application like Jenkins) can be run on any supported operating system (macOS, Linux and Windows) or cloud service (AWS and Azure) which is also running Docker.

Installation – Setup

### Download and run Jenkins

1. Download Jenkins.
2. Open up a terminal in the download directory.
3. Run java -jar jenkins.war --httpPort=8080.
4. Browse to http://localhost:8080.
5. Follow the instructions to complete the installation.

Path to get initial admin password:

C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword

**User credentials:**

User Id : sauser

Password: Test@123

Name: TestUser

Email: raman@techvisionit.com

Installation Configuration

# Changing boot configuration

By default, your Jenkins runs at http://localhost:8080/. This can be changed by editing jenkins.xml, which is located in your installation directory. This file is also the place to change other boot configuration parameters, such as JVM options, HTTPS setup, etc.

# Starting/stopping the service

Jenkins is installed as a Windows service, and it is configured to start automatically upon boot. To start/stop them manually, use the service manager from the control panel, or the sccommand line tool.

# Inheriting your existing Hudson/Jenkins installation

If you'd like your new installation to take over your existing Jenkins/Hudson data, copy your old data directory into the new JENKINS\_HOME directory.

# Creating your first Pipeline

### What is a Jenkins Pipeline?

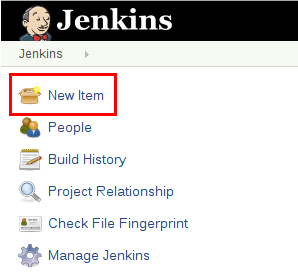
Jenkins Pipeline (or simply "Pipeline") is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.

A continuous delivery pipeline is an automated expression of your process for getting software from version control right through to your users and customers.

Jenkins Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code". The definition of a Jenkins Pipeline is typically written into a text file (called a Jenkinsfile) which in turn is checked into a project’s source control repository. [1]

For more information about Pipeline and what a Jenkinsfile is, refer to the respective Pipeline and Using a Jenkinsfile sections of the User Handbook.

To get started quickly with Pipeline:

1. Copy one of the examples below into your repository and name it Jenkinsfile
2. Click the **New Item** menu within Jenkins 
3. Provide a name for your new item (e.g. **My Pipeline**) and select **Multibranch Pipeline**
4. Click the **Add Source** button, choose the type of repository you want to use and fill in the details.
5. Click the **Save** button and watch your first Pipeline run!

You may need to modify one of the example Jenkinsfile's to make it run with your project. Try modifying the sh command to run the same command you would run on your local machine.

After you have setup your Pipeline, Jenkins will automatically detect any new Branches or Pull Requests that are created in your repository and start running Pipelines for them.

**Continue to "Run multiple steps"**

## Quick Start Examples

Below are some easily copied and pasted examples of a simple Pipeline with various languages.

### Java

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'maven:3.3.3' }

stages {

stage('build') {

steps {

sh 'mvn --version'

sh 'mvn --version'

sh 'mvn --version'

sh 'mvn --version'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

### Node.js / JavaScript

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'node:6.3' }

stages {

stage('build') {

steps {

sh 'npm --version'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

### Ruby

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'ruby' }

stages {

stage('build') {

steps {

sh 'ruby --version'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

### Python

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'python:3.5.1' }

stages {

stage('build') {

steps {

sh 'python --version'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

### PHP

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'php' }

stages {

stage('build') {

steps {

sh 'php --version'

}

}

}

}

# Running multiple steps

Pipelines are made up of multiple steps that allow you to build, test and deploy applications. Jenkins Pipeline allows you to compose multiple steps in an easy way that can help you model any sort of automation process.

Think of a "step" like a single command which performs a single action. When a step succeeds it moves onto the next step. When a step fails to execute correctly the Pipeline will fail.

### Linux, BSD, and Mac OS

On Linux, BSD, and Mac OS (Unix-like) systems, the sh step is used to execute a shell command in a Pipeline.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo "Hello World"'

sh '''

echo "Multiline shell steps works too"

ls -lah

'''

}

}

}

}

Toggle Scripted Pipeline (Advanced)

### Windows

Windows-based systems should use the bat step for executing batch commands.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

bat 'set'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

## Timeouts, retries and more

There are some powerful steps that "wrap" other steps which can easily solve problems like retrying (retry) steps until successful or exiting if a step takes too long (timeout).

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

retry(3) {

sh './flakey-deploy.sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh './health-check.sh'

}

}

}

}

}

Toggle Scripted Pipeline (Advanced)

The "Deploy" stage retries the flakey-deploy.sh script 3 times, and then waits for up to 3 minutes for the health-check.sh script to execute. If the health check script does not complete in 3 minutes, the Pipeline will be marked as having failed in the "Deploy" stage.

"Wrapper" steps such as timeout and retry may contain other steps, including timeout or retry.

We can compose these steps together. For example, if we wanted to retry our deployment 5 times, but never want to spend more than 3 minutes in total before failing the stage:

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh './flakey-deploy.sh'

}

}

}

}

}

}

Toggle Scripted Pipeline (Advanced)

## Finishing up

When the Pipeline has finished executing, you may need to run clean-up steps or perform some actions based on the outcome of the Pipeline. These actions can be performed in the post section.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh 'echo "Fail!"; exit 1'

}

}

}

post {

always {

echo 'This will always run'

}

success {

echo 'This will run only if successful'

}

failure {

echo 'This will run only if failed'

}

unstable {

echo 'This will run only if the run was marked as unstable'

}

changed {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

}

}

# Defining execution environments

In the previous section you may have noticed the agent directive in each of the examples. The agent directive tells Jenkins where and how to execute the Pipeline, or subset thereof. As you might expect, the agent is required for all Pipelines.

Underneath the hood, there are a few things agent causes to happen:

* All the steps contained within the block are queued for execution by Jenkins. As soon as an executor is available, the steps will begin to execute.
* A workspace is allocated which will contain files checked out from source control as well as any additional working files for the Pipeline.

There are several ways to define agents for use in Pipeline, for this tour we will only focus on using an ephemeral Docker container.

Pipeline is designed to easily use Docker images and containers to run inside. This allows the Pipeline to define the environment and tools required without having to configure various system tools and dependencies on agents manually. This approach allows you to use practically any tool which can be packaged in a Docker container.

For more agent specification options, consult the syntax reference.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent {

docker { image 'node:7-alpine' }

}

stages {

stage('Test') {

steps {

sh 'node --version'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

When the Pipeline executes, Jenkins will automatically start the specified container and execute the defined steps within it:

[Pipeline] stage

[Pipeline] { (Test)

[Pipeline] sh

[guided-tour] Running shell script

+ node --version

v7.4.0

[Pipeline] }

[Pipeline] // stage

[Pipeline] }

# Using environment variables

Environment variables can be set globally, like the example below, or per stage. As you might expect, setting environment variables per stage means they will only apply to the stage in which they’re defined.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

environment {

DISABLE\_AUTH = 'true'

DB\_ENGINE = 'sqlite'

}

stages {

stage('Build') {

steps {

sh 'printenv'

}

}

}

}

This approach to defining environment variables from within the Jenkinsfile can be very useful for instructing scripts, such as a Makefile, to configure the build or tests differently to run them inside of Jenkins.

Another common use for environment variables is to set or override "dummy" credentials in build or test scripts. Because it’s (*obviously*) a bad idea to put credentials directly into a Jenkinsfile, Jenkins Pipeline allows users to quickly and safely access pre-defined credentials in the Jenkinsfile without ever needing to know their values

# Recording tests and artifacts

While testing is a critical part of a good continuous delivery pipeline, most people don’t want to sift through thousands of lines of console output to find information about failing tests. To make this easier, Jenkins can record and aggregate test results so long as your test runner can output test result files. Jenkins typically comes bundled with the junit step, but if your test runner cannot output JUnit-style XML reports, there are additional plugins which process practically any widely-used test report format.

To collect our test results and artifacts, we will use the post section.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

junit 'build/reports/\*\*/\*.xml'

}

}

}

Toggle Scripted Pipeline (Advanced)

This will always grab the test results and let Jenkins track them, calculate trends and report on them. A Pipeline that has failing tests will be marked as "UNSTABLE", denoted by yellow in the web UI. That is distinct from the "FAILED" state, denoted by red.

When there are test failures, it is often useful to grab built artifacts from Jenkins for local analysis and investigation. This is made practical by Jenkins’s built-in support for storing "artifacts", files generated during the execution of the Pipeline.

This is easily done with the archiveArtifacts step and a file-globbing expression, as is demonstrated in the example below:

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

sh './gradlew build'

}

}

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

}

Toggle Scripted Pipeline (Advanced)

If more than one parameter is specified in the archiveArtifacts step, then each parameter’s name must explicitly be specified in the step code - i.e. artifacts for the artifact’s path and file name and fingerprint to choose this option. If you only need to specify the artifacts' path and file name/s, then you can omit the parameter name artifacts - e.g.  
archiveArtifacts 'build/libs/\*\*/\*.jar'

Recording tests and artifacts in Jenkins is useful for quickly and easily surfacing information to various members of the team. In the next section we’ll talk about how to **tell** those members of the team what’s been happening in our Pipeline.

# Cleaning up and notifications

Since the post section of a Pipeline is guaranteed to run at the end of a Pipeline’s execution, we can add some notification or other steps to perform finalization, notification, or other end-of-Pipeline tasks.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('No-op') {

steps {

sh 'ls'

}

}

}

post {

always {

echo 'One way or another, I have finished'

deleteDir() /\* clean up our workspace \*/

}

success {

echo 'I succeeeded!'

}

unstable {

echo 'I am unstable :/'

}

failure {

echo 'I failed :('

}

changed {

echo 'Things were different before...'

}

}

}

Toggle Scripted Pipeline (Advanced)

There are plenty of ways to send notifications, below are a few snippets demonstrating how to send notifications about a Pipeline to an email, a Hipchat room, or a Slack channel.

### Email

post {

failure {

mail to: 'team@example.com',

subject: "Failed Pipeline: **${**currentBuild.fullDisplayName**}**",

body: "Something is wrong with **${**env.BUILD\_URL**}**"

}

}

### Hipchat

post {

failure {

hipchatSend message: "Attention @here **${**env.JOB\_NAME**}** #**${**env.BUILD\_NUMBER**}** has failed.",

color: 'RED'

}

}

### Slack

post {

success {

slackSend channel: '#ops-room',

color: 'good',

message: "The pipeline **${**currentBuild.fullDisplayName**}** completed successfully."

}

}

# Deployment

The most basic continuous delivery pipeline will have, at minimum, three stages which should be defined in a Jenkinsfile: Build, Test, and Deploy. For this section we will focus primarily on the Deploy stage, but it should be noted that stable Build and Test stages are an important precursor to any deployment activity.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building'

}

}

stage('Test') {

steps {

echo 'Testing'

}

}

stage('Deploy') {

steps {

echo 'Deploying'

}

}

}

}

Toggle Scripted Pipeline (Advanced)

## Stages as Deployment Environments

One common pattern is to extend the number of stages to capture additional deployment environments, like "staging" or "production", as shown in the following snippet.

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

In this example, we’re assuming that whatever "smoke tests" are run by our ./run-smoke-tests script are sufficient to qualify or validate a release to the production environment. This kind of pipeline that automatically deploys code all the way through to production can be considered an implementation of "continuous deployment." While this is a noble ideal, for many there are good reasons why continuous deployment might not be practical, but those can still enjoy the benefits of continuous delivery. [1] Jenkins Pipeline readily supports both.

## Asking for human input to proceed

Often when passing between stages, especially environment stages, you may want human input before continuing. For example, to judge if the application is in a good enough state to "promote" to the production environment. This can be accomplished with the input step.\In the example below, the "Sanity check" stage actually blocks for input and won’t proceed without a person confirming the progress.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

/\* "Build" and "Test" stages omitted \*/

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Sanity check') {

steps {

input "Does the staging environment look ok?"

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

}

}