This document is no longer maintained. Current Pipeline documentation lives on the jenkins.io documentation zone. Content will be removed from this document as it is verified that the new site has equivalent information, or that it is outdated or unnecessary.

This document is intended for new users of the pipeline feature to learn how to write and understand pipelines.

Why Pipeline?

Pipeline (formerly known as Workflow) was built with the community's requirements for a flexible, extensible, and script-based CD pipeline capability for Jenkins in mind. To that end, Pipeline:

- Can support complex, real-world, CD Pipeline requirements: pipelines can fork/join, loop, *parallel*, to name a few
- Is Resilient: pipeline executions can survive master restarts
- Is Pausable: pipelines can pause and wait for human input/approval
- Is Efficient: pipelines can restart from saved checkpoints
- Is Visualized: Pipeline StageView provides status at-a-glance dashboards including trending

Getting Started

Before you begin, ensure you have the following installed or running:

- You must be running Jenkins 1.580.1 or later (1.642.3+ for latest features).
- Ensure Pipeline is installed: navigate to **Plugin Manager**, install **Pipeline** and restart Jenkins.

Note: If you are running CloudBees Jenkins Enterprise 14.11 or later, you already have Pipeline (plus additional associated features).

If you want to play with Pipeline without installing Jenkins separately (or accessing your production system), try running the Docker demo.

Creating a Pipeline

To create a pipeline, perform the following steps:

1. Click **New Item**, pick a name for your job, select **Pipeline**, and click **OK**.

You will be taken to the configuration screen for the Pipeline. The *Script* text area is important as this is where your Pipeline script is defined. We'll start with a trivial script:

```
echo 'hello from Pipeline'
```

Note: if you are not a Jenkins administrator, click the **Use Groovy Sandbox** option (read here to learn more about this option).

2. **Save** your pipeline when you are done.
3. Click **Build Now** to run it. You should see `#1` under *Build History*.
4. Click ▾ and select **Console Output** to see the output:

```
Started by user anonymous
[Pipeline] echo
hello from Pipeline
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Understanding Pipeline Scripts

A pipeline is a Groovy script that tells Jenkins what to do when your Pipeline is run. You do not need to know much general Groovy to use Pipeline - relevant bits of syntax are introduced as needed.

Example In this example, `echo` is a *step*: a function defined in a Jenkins plugin and made available to all pipelines. Groovy functions can use a C/Java-like syntax such as:

```
echo("hello from Pipeline");
```

You can drop the semicolon (`;`), drop the parentheses (`(` and `)`), and use single quotes (`'`) instead of double (`"`) if you do not need to perform variable substitutions.

Comments in Groovy, as in Java, can use single-line or multiline styles:

```
/*
 * Copyright 2014 Yoyodyne, Inc.
 */
// FIXME write this
```

Creating a Simple Pipeline

The following sections guide you through creating a simple Pipeline.

Setting Up

To set up for creating a Pipeline, ensure you have the following:

1. First, you need a Maven installation available to do builds with. Go to *Jenkins » Manage Jenkins » Configure System*, click **Add Maven**, give it the name **M3** and allow it to install automatically. For Jenkins 2.x and later, this option is under *Jenkins » Manage Jenkins » Global Tool Configuration* instead.
2. Only if you do not have Git installed on your Jenkins server: click **Delete Git** on the default Git installation and *Add Git » JGit* to replace it.
3. Click **Save**.

Checking out and Building Sources

Now, click on your Pipeline and **Configure** it to edit its script.

```
node {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B verify"
}
```

When you run this script:

- it should check out a Git repository and run Maven to build it.
- it will run some tests that might (at random) pass, fail, or be skipped. If they fail, the `mvn` command will fail and your Pipeline run will end with:

```
ERROR: script returned exit code 1
Finished: FAILURE
```

Modifying for Windows Variations

This documentation assumes Jenkins is running on Linux or another Unix-like operating system. If your Jenkins server (or, later, agent node) is running on Windows, try using `bat` in place of `sh`, and use backslashes as the file separator where needed (backslashes do generally need to be escaped inside strings).

Example: rather than:

```
sh "${mvnHome}/bin/mvn -B verify"
```

you could use:

```
bat "${mvnHome}\\bin\\mvn -B verify"
```

Understanding Syntax

A `node` is a step that schedules a task to run by adding it to the Jenkins build queue.

- As soon as an executor slot is available on a **node** (the Jenkins master, or agent), the task is run on that node.
- A `node` also allocates a **workspace** (file directory) on that node for the duration of the task (more on this later).

Groovy functions accept **closures** (blocks of code) and some steps expect a block. In this case, the code between the braces (`{` and `}`) is the body of the `node` step. Many steps (such as: `git` and `sh` in this example) can only run in the context of a node, so trying to run just:

```
sh 'echo oops'
```

as a Pipeline script will not work: Jenkins does not know what system to run commands on.

Unlike user-defined functions, Pipeline steps always take named parameters. Thus:

```
git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
```

is passing one parameter named: `url` (the Git source code repository to check out). This parameter happens to be mandatory; it also takes some other optional parameters such as `branch` . You can pass as many as you need:

```
git url: 'https://github.com/jglick/simple-maven-project-with-tests.git',
```

As before, Groovy lets you omit parentheses around function arguments. The named-parameter syntax is also a shorthand for creating a *map*, which in Groovy uses the syntax `[key1: value1, key2: value2]` , so you could also write:

```
git([url: 'https://github.com/jglick/simple-maven-project-with-tests.git',
```

For convenience, when calling steps taking only one parameter (or only one mandatory parameter) you can omit the parameter name. For example:

```
sh 'echo hello'
```

is really shorthand for:

```
sh([script: 'echo hello'])
```

The `tool` step makes sure a tool with the given name (in this case, a specific version of the Maven build tool) is installed on the current node. But merely running this step does not do much good. The script needs to know *where* it was installed - so the tool can be run later. For this, you need a variable.

The `def` keyword in Groovy is the quickest way to define a new variable (with no specific type).

Here:

```
def mvnHome = tool 'M3'
```

ensures `M3` is installed somewhere accessible to Jenkins and assigns the return value of the step (an installation path) to the `mvnHome` variable. You could also use a more Java-like syntax with a static type:

```
String mvnHome = tool("M3");
```

Finally, you run the Maven build. When Groovy encounters `$` inside a double-quoted string:

```
"${mvnHome}/bin/mvn -B verify"
```

it replaces the `${mvnHome}` part with the value of that expression (here, just the variable value). The more verbose Java-like syntax would be:

```
mvnHome + "/bin/mvn -B verify"
```

In the console output, you see the final command being run.

Example:

```
[Pipeline] Running shell script
+
/path/to/jenkins/tools/hudson.tasks.Maven_MavenInstallation/M3/bin/mvn
-B verify
```

Managing the Environment

One way to use tools by default, is to add them to your executable path - by using the special variable `env` that is defined for all pipelines:

```
node {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  env.PATH = "${mvnHome}/bin:${env.PATH}"
  sh 'mvn -B verify'
}
```

- Properties of this variable are environment variables on the current node.
- You can override certain environment variables and the overrides are seen by subsequent `sh` steps (or anything else that pays attention to environment variables).
- You can run `mvn` without a fully-qualified path.

Setting a variable such as `PATH` in this way is only safe if you are using a single agent for this build. As an alternative, you can use the `withEnv` step to set a variable within a scope:

```
node {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
    sh 'mvn -B verify'
  }
}
```

Some environment variables are defined by Jenkins by default.

Example: `env.BUILD_TAG` can be used to get a tag like `jenkins-projname-1` from Groovy code, or `$BUILD_TAG` can be used from a `sh` script.

See Help in the **Snippet Generator** for the `withEnv` step for more details on this topic.

Build Parameters

If you have configured your pipeline to accept parameters when it is built — **Build with Parameters** — they are accessible as Groovy variables inside `params`. They are also accessible as environment variables.

Example: Using `isFoo` parameter defined as a boolean parameter (checkbox in the UI):

```
node {
  sh "isFoo is ${params.isFoo}"
  sh 'isFoo is ' + params.isFoo
  if (params.isFoo) {
    // do something
  }
```

Recording Test Results and Artifacts

Instead of failing the build if there are test failures, you want Jenkins to record them — and then proceed. If you want it saved, you must capture the JAR that you built.

```
node {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
  step([$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerpr
  step([$class: 'JUnitResultArchiver', testResults: '**/target/surefire-re
}
```

- If tests fail, the Pipeline is marked unstable (yellow ball), and you can browse the **Test Result Trend** to see the history.
- You should see **Last Successful Artifacts** on the Pipeline index page.

Understanding Syntax

The Maven option `-Dmaven.test.failure.ignore` allows the `mvn` command to exit normally (status 0) — so that the Pipeline continues, even when test failures are recorded on disk.

Run the `step` step twice. This step allows you to use certain build (or post-build) steps already defined in Jenkins for use in traditional projects. It takes one parameter (called `delegate` but omitted here) — this parameter value is a standard Jenkins build step.

You could create the delegate using Java constructor/method calls, using Groovy or Java syntax:

```
def aa = new hudson.tasks.ArtifactArchiver('**/target/*.jar')
aa.fingerprint = true // i.e., aa.setFingerprint(true)
step aa
```

but this is cumbersome and does not work well with Groovy sandbox security — so any object-valued argument to a step may instead be given as a map.

The following:

```
[$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerprint: tr
```

- specifies the values of the `artifacts` and `fingerprint` properties (controls what files to save and records fingerprints for them).
- `$class` is used to pick the kind of object to create. It may be a fully-qualified class name (`hudson.tasks.ArtifactArchiver`), but the simple name may be used when unambiguous.

In some cases, part of a step configuration will force an object to be of a fixed class. Thus, `$class` can be omitted entirely.

Newer versions of Pipeline will often allow shorter forms, such as

```
archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true
```

Pipeline: Groovy 2.10 or later is needed for such syntax. Depending on the particular call, you may also need *Pipeline: Basic Steps* 2.1 or later, Jenkins core 2.2 or later, and/or updated versions of various Pipeline-compatible plugins. See the *Pipeline Syntax: Reference* page inside Jenkins for a detailed guide on step configuration syntax.

Example: rather than using the simple `git` step, you can use the more general `checkout` step and specify any complex configuration supported by the Git plugin:

```
checkout scm: [$class: 'GitSCM', branches: [[name: '*/master']], userRemot
```

Here, `[[name: '*/master']]` is an array with one map element, `[name: '*/master']` , which is an object of type `hudson.plugins.git.BranchSpec` , but we can omit `$class: 'BranchSpec'` since `branches` can only hold this kind of object. Similarly, the elements of `userRemoteConfigs` are declared to be of type `UserRemoteConfig` , so this need not be mentioned.

Using Agents

Thus far, pipeline has run only on the "master" agent on your Jenkins server - assuming you had no other agents configured. You can even force it to run on the master by telling the `node` step the following:

```
node('master') {
    // as before
}
```

Here, you pass a value for the optional `label` parameter of the step, as well as a body block.

To create a simple agent:

1. Select *Manage Jenkins » Manage Nodes » New Node* and create a *Permanent Agent*. Leave *# of executors* as 1.
2. Pick a **Remote root directory** such as `/tmp/agent`.
3. Enter `remote` in the **Labels** field and set the *Launch method* to *Launch agents via Java Web Start*.
4. **Save**, then click on the new agent and **Launch**.
5. Now, go back to your Pipeline definition and request this agent's label:

```
node('remote') {
    // as before
}
```

The parameter may be a node name, or a single label, or even a label expression such as:

```
node('unix && 64bit') {
    // as before
}
```

When you **Build Now**, you see:

```
Running on <youragentname> in /<agentroot>/workspace/<jobname>
```

and the `M3` Maven installation being unpacked to this agent root.

Pausing: Flyweight vs. Heavyweight Executors

Pause the script to take a better look at what is happening:

```
node('remote') {
  input 'Ready to go?'
  // rest as before
}
```

The `input` step pauses Pipeline execution. Its default `message` parameter gives a prompt, which is shown to a human. You can, optionally, request information back.

When you run a new build, you see:

```
Running: Input
Ready to go?
Proceed or Abort
```

If you click **Proceed**, the build will proceed as before. First, go to the Jenkins main page and look at the **Build Executor Status** widget.

- You will see an unnumbered entry under **master** named **jobname #10**; executors #1 and #2 on the master are idle.
- You will also see an entry under your agent, in a numbered row (probably #1) called **Building part of jobname #10**.

Why are there two executors consumed by one Pipeline build?

- Every Pipeline build itself runs on the master, using a **flyweight executor** — an uncounted slot that is assumed to not take any significant computational power.
- This executor represents the actual Groovy script, which is almost always idle, waiting for a step to complete.
- Flyweight executors are always available.

When you run a `node` step:

- A regular heavyweight executor is allocated on a node (usually an agent) matching the label expression, as soon as one is available. This executor represents the real work being done on the node.
- If you start a second build of the Pipeline while the first is still paused with the one available executor, you will see both Pipeline builds running on master. But only the first will have grabbed the one available executor on the agent; the other **part of jobname #11** will be shown in **Build Queue (1)**. (shortly after, the console log for the second build will note that it is still waiting for an available executor).

To finish up, click the ▾ beside either executor entry for any running Pipeline and select **Paused for Input**, then click **Proceed** (you can also click the link in the console output).

Allocating Workspaces

In addition to waiting to allocate an executor on a node, the `node` step also automatically allocates a **workspace**: a directory specific to this job — where you can check out sources, run commands, and do other work. Workspaces are locked for the duration of the step: only one build at a time can use a given workspace. If multiple builds need a workspace on the same node, additional workspaces are allocated.

Configure your agent, set **# of executors** to 2 and **Save**. Now start your build twice in a row. The log for the second build will show

```
Running on <youragentname> in /<agentroot>/workspace/<jobname>@2
```

The `@2` shows that the build used a separate workspace from the first one, with which it ran concurrently. You should also have seen

```
Cloning the remote Git repository
```

since this new workspace required a new copy of the project sources.

You can also use the `ws` step to explicitly ask for another workspace on the current agent, *without* grabbing a new executor slot. Inside its body all commands run in the second workspace. The `dir` step can be used to run a block with a different working directory (typically a subdirectory of the workspace) without allocating a new workspace.

Adding More Complex Logic

Your Groovy script can include functions, conditional tests, loops, `try` / `catch` / `finally` blocks, and so on. Save this Pipeline definition:

```
node('remote') {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def v = version()
  if (v) {
    echo "Building version ${v}"
  }
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
```

```
    step([$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerpr
    step([$class: 'JUnitResultArchiver', testResults: '**/target/surefire-re
  }
  def version() {
    def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
    matcher ? matcher[0][1] : null
  }
```

Here, you use:

- `def` keyword to define a function (you can also give a Java type in place of `def` to make it look more like a Java method)
- `=~` is Groovy syntax to match text against a regular expression
- `[0]` looks up the first match
- `[1]` the first `(…)` group within that match
- `readFile` step loads a text file from the workspace and returns its content (do not try to use `java.io.File` methods — these will refer to files on the master where Jenkins is running, not in the current workspace).
- There is also a `writeFile` step to save content to a text file in the workspace
- `fileExists` step to check whether a file exists without loading it.

When you run the Pipeline you see:

```
Building version 1.0-SNAPSHOT
```

Note: Unless your Script Security plugin is version 1.11 or higher, you may see a `RejectedAccessException` error at this point. If so, a Jenkins administrator will need to navigate to **Manage Jenkins » In-process Script Approval** and **Approve** `staticMethod org.codehaus.groovy.runtime.ScriptBytecodeAdapter findRegex java.lang.Object java.lang.Object` . Then try running your script again and it should work.

Serializing Local Variables

If you tried inlining the `version` function as follows:

```
node('remote') {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
  if (matcher) {
    echo "Building version ${matcher[0][1]}"
  }
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
```

```
  step([$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerpr
  step([$class: 'JUnitResultArchiver', testResults: '**/target/surefire-re
}
```

you would have noticed a problem:

```
java.io.NotSerializableException: java.util.regex.Matcher
```

- This occurs because the `matcher` local variable is of a type (`Matcher`) not considered serializable by Java. Since pipelines must survive Jenkins restarts, the state of the running program is periodically saved to disk so it can be resumed later (saves occur after every step or in the middle of steps such as `sh`).
- The “state” includes the whole control flow including: local variables, positions in loops, and so on. As such: any variable values used in your program should be numbers, strings, or other serializable types, not “live” objects such as network connections.
- If you must use a nonserializable value temporarily: discard it before doing anything else. When you keep the matcher only as a local variable inside a function, it is automatically discarded as soon as the function returned.

You can also explicitly discard a reference when you are done with it:

```
node('remote') {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
  if (matcher) {
    echo "Building version ${matcher[0][1]}"
  }
  matcher = null
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
  step([$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerpr
  step([$class: 'JUnitResultArchiver', testResults: '**/target/surefire-re
}
```

However the safest approach is to isolate use of nonserializable state inside a method marked with the annotation `@NonCPS` . Such a method will be treated as “native” by the Pipeline engine, and its local variables never saved. However it may *not* make any calls to Pipeline steps, so the `readFile` call must be pulled out:

```
node('remote') {
  git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'
  def v = version(readFile('pom.xml'))
```

```
  if (v) {
    echo "Building version ${v}"
  }
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
  step([$class: 'ArtifactArchiver', artifacts: '**/target/*.jar', fingerpr
  step([$class: 'JUnitResultArchiver', testResults: '**/target/surefire-re
}
@NonCPS
def version(text) {
  def matcher = text =~ '<version>(.+)</version>'
  matcher ? matcher[0][1] : null
}
```

Here the logic inside the `version` function is run by the normal Groovy runtime, so any local variables are permitted.

The Pipeline: Groovy plugin page has deeper background on `@NonCPS`.

Creating Multiple Threads

Pipelines can use a `parallel` step to perform multiple actions at once. This special step takes a map as its argument; keys are “branch names” (labels for your own benefit), and values are blocks to run.

To see how this can be useful, install a new plugin: **Parallel Test Executor** (version 1.9 or later). This plugin includes a Pipeline step that lets you split apart slow test runs. Also make sure the JUnit plugin is at least version 1.18+.

Now create a new pipeline with the following script:

```
node('remote') {
  git 'https://github.com/jenkinsci/parallel-test-executor-plugin-sample.g
  stash name: 'sources', includes: 'pom.xml,src/'
}
def splits = splitTests count(2)
def branches = [:]
for (int i = 0; i < splits.size(); i++) {
  def index = i // fresh variable per iteration; i will be mutated
  branches["split${i}"] = {
    node('remote') {
      deleteDir()
      unstash 'sources'
      def exclusions = splits.get(index);
      writeFile file: 'exclusions.txt', text: exclusions.join("\n")
      sh "${tool 'M3'}/bin/mvn -B -Dmaven.test.failure.ignore test"
      junit 'target/surefire-reports/*.xml'
    }
  }
```

```
    }
    parallel branches
```

When you run this Pipeline for the first time, it will check out a project and run all of its tests in sequence. The second and subsequent times you run it, the `splitTests` task will partition your tests into two sets of roughly equal runtime. The rest of the Pipeline then runs these in parallel — so if you look at **trend** (in the **Build History** widget) you will see the second and subsequent builds taking roughly half the time of the first. If you only have the one agent configured with its two executors, this won't save as much time, but you may have multiple agents on different hardware matching the same label expression.

This script is more complex than the previous ones so it bears some examination. You start by grabbing an agent, checking out sources, and making a copy of them using the `stash` step:

```
    stash name: 'sources', includes: 'pom.xml,src/'
```

Later, you `unstash` these same files back into **other** workspaces. You could have just run `git` anew in each agent's workspace, but this would result in duplicated changelog entries, as well as contacting the Git server twice.

- A Pipeline build is permitted to run as many SCM checkouts as it needs to, which is useful for projects working with multiple repositories, but not what we want here.
- More importantly, if anyone pushes a new Git commit at the wrong time, you might be testing different sources in some branches - which is prevented when you do the checkout just once and distribute sources to agents yourself.

The command `splitTests` returns a list of lists of strings. From each (list) entry, you construct one branch to run; the label (map key) is akin to a thread name, and will appear in the build log. The Maven project is set up to expect a file `exclusions.txt` at its root, and it will run all tests *not* mentioned there, which we set up via the `writeFile` step. When you run the `parallel` step, each branch is started at the same time, and the overall step completes when all the branches finish: “fork & join”.

There are several new ideas at work here:

- A single Pipeline build allocates several executors, potentially on different agents, at the same time. You can see these starting and finishing in the Jenkins executor widget on the main screen.

- Each call to `node` gets its own workspace. This kind of flexibility is impossible in a freestyle project, each build of which is tied to exactly one workspace.The Parallel Test Executor plugin works around that for its freestyle build step by triggering multiple builds of the project, making the history hard to follow.

Do not use `env` in this case:

```
env.PATH = "${mvnHome}/bin:${env.PATH}"
```

because environment variable overrides are limited to being global to a pipeline run, not local to the current thread (and thus agent). You could, however, use the `withEnv` step as noted above.

You may also have noticed that you are running `JUnitResultArchiver` several times, something that is not possible in a freestyle project. The test results recorded in the build are cumulative.

When you view the log for a build with multiple branches, the output from each will be intermixed. It can be useful to click on the *Pipeline Steps* link on the build's sidebar. This will display a tree-table view of all the steps run so far in the build, grouped by logical block, for example `parallel` branch. You can click on individual steps and get more details, such as the log output for that step in isolation, the workspace associated with a `node` step, and so on.

Creating Stages

A `stage` block lets you label certain sections of a build for use in visualizations. Other steps like `milestone` and `lock` can be used to control concurrency of multiple builds.

Consult the Docker demo for an example of a Pipeline using multiple `stage` s.

Loading Script Text from Version Control

Complex Pipelines would be cumbersome to write and maintain in the textarea provided in the Jenkins job configuration. Therefore it makes sense to load the program from another source, one that you can maintain using version control and standalone Groovy editors.

Building Entire Script from SCM

The easiest way to do this is to select **Pipeline script from SCM** when defining the pipeline.

In that case you do not enter any Groovy code in the Jenkins UI; you just indicate where in source code you want to retrieve the program. If you update this repository, a new build will be triggered, so long as your job is configured with an SCM polling trigger.

Using Libraries

The Shared Groovy Libraries plugin, included in the Pipeline suite, allows multiple jobs to shared common utility code.

Triggering Manual Loading

For some cases, you may prefer to explicitly load Groovy script text from some source. The standard Groovy `evaluate` function can be used, but most likely you will want to load a Pipeline definition from a workspace. For this purpose, you can use the `load` step, which takes a filename in the workspace and runs it as Groovy source text.

The loaded file can contain statements at top level, which are run immediately. That is fine if you only want to use a single executor and workspace, and do not mind hard-coding the agent label in the Jenkins job. For more complex cases, though, you want to leave the external script in full control of agent allocation. In that case the main script defined in the job can just load and run a closure (block of code to be run later):

```
node {
    git '…'
    load 'pipeline.groovy'
}()
```

The subtle part here is that we actually have to do a bit of work with the `node` and `git` steps just to check out a source repository into a workspace so that we can `load` something. Once we have loaded the code, we exit the initial `node` block to release the temporary workspace, so it is not locked for the duration of the build. The return value of the `load` step also becomes the return value of the `node` step, which we run as a closure with the parentheses `()`.

Here `pipeline.groovy` could look like:

```
{ ->
  node('special-agent') {
    hello 'world'
  }
```

```
}
def hello(whom) {
  echo "hello ${whom}"
}
```

Note: While it can contain helper functions, the only code at top level is a Groovy `Closure`, which is the return value of the script, and thus of the main script's `load` step.

The helper script can alternately define functions and return `this`, in which case the result of the `load` step can be used to invoke those functions like object methods. An older version of the Docker demo showed this technique in practice:

```
def pipeline
node('agent') {
    git '…'
    pipeline = load 'pipeline.groovy'
    pipeline.devQAStaging()
}
pipeline.production()
```

where pipeline.groovy defines `devQAStaging` and `production` functions (among others) before ending with

```
return this;
```

In this case `devQAStaging` runs on the same node as the main source code checkout, while `production` runs outside of that block (and in fact allocates a different node).

To reduce the amount of boilerplate needed in the master script, you can try the Workflow Remote File Loader plugin.

Creating Multibranch Projects

The **Pipeline: Multibranch** plugin offers a better way of versioning your Pipeline and managing your project. You need to create a distinct project type, **Multibranch Pipeline**.

When you have a multibranch pipeline, the configuration screen will resemble **Pipeline script from SCM** in that your Pipeline script comes from source control, not the Jenkins job configuration. The difference is that you do not configure a single branch, but a **set** of branches, and Jenkins creates a subproject for each branch it finds in your repository.

For example, if you select **Git** as the branch source (Subversion and Mercurial are also supported already), you will be prompted for the usual connection information, but then rather than a fixed refspec you will enter a branch name pattern (use the defaults to look for any branch). Jenkins expects to find a script named `Jenkinsfile` in branches it can build. From this script, the command `checkout scm` suffices to check out your project's source code inside some `node {}`.

Say you start with just a `master` branch, then you want to experiment with some changes, so you `git checkout -b newfeature` and push some commits. Jenkins automatically detects the new branch in your repository and creates a new subproject for it—with its own build history unrelated to trunk, so no one will mind if it has red/yellow balls for a while. If you choose, you can ask for the subproject to be automatically removed after the branch is merged and deleted.

If you want to change your Pipeline script—for example, to add a new Jenkins publisher step corresponding to reports your `Makefile` / `pom.xml` /etc. is newly creating—you just edit `Jenkinsfile` in your change. The Pipeline script is always synchronized with the rest of the source code you are working on: `checkout scm` checks out the same revision as the script is loaded from.

Exploring the Snippet Generator

There are a number of Pipeline steps not discussed in this document, and plugins can add more. Even steps discussed here can take various special options that can be added from release to release. To browse all available steps and their syntax, a help tool is built into the Pipeline definition screen.

Click **Snippet Generator** beneath your script text area. You see a list of installed steps. Some will have a help icon () at the top which you can click to see general information. There are also UI controls to help you configure the step — in some cases with auto completion and other features found in Jenkins configuration screens. Click help icons to see all.

When you are done, click **Generate Groovy** to see a Groovy snippet that will run the step exactly as you have configured it. This lets you see the function name used for the step, the names of any parameters it takes (if not a default parameter), and their syntax. You can copy and paste the generated code right into your Pipeline, or use it as a starting point (perhaps trimming some unnecessary optional parameters).