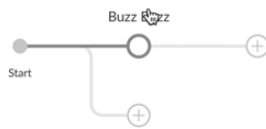


Lab: Create and edit a simple pipeline with branches, artifacts, variables and parallel stages

Strip our Pipeline

We are going to be developing a new pipeline; it is not ready to be put into production, so we will create a new branch and save ("commit") the Pipeline to that branch. To proceed with our simple pipeline using the skeletal pipeline we made in Creating your first (skeletal) Pipeline. Delete all but the the first stage (**Buzz Buzz**) with its single step:



Pipeline Settings

Agent

any

Environment

Name	Value

Save Pipeline

Saving the pipeline will commit a Jenkinsfile to the repository.

Description

What changed?

☐ Commit to *master*

☒ Commit to new branch

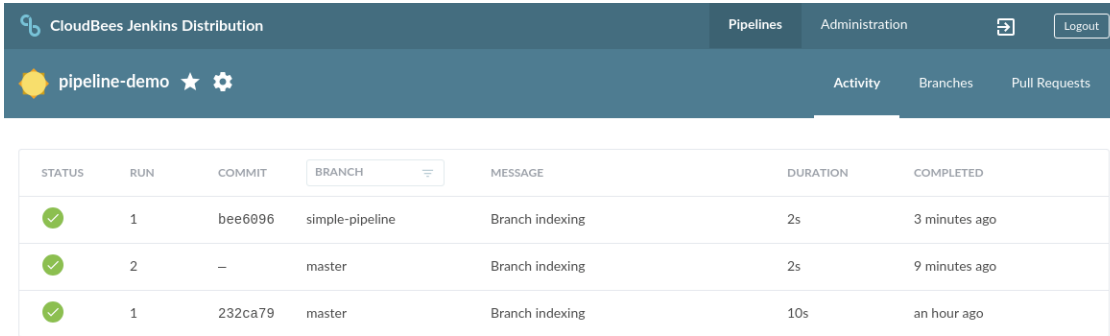
simple-pipeline

Save & run Cancel

To save this modified Pipeline to a branch:

- Click the pencil icon to go back to the Blue Ocean Visual Editor, delete the **Bees Bees** stage, and then select **Save** to save the pipeline
- Type "Start simple pipeline" in the *Description* box
- Select **Commit to new branch**
- Type the name of your new branch in the box; for example, "simple-pipeline"
- Select **Save & run** when the information is complete.

Watch Pipeline Run



STATUS	RUN	COMMIT	BRANCH	MESSAGE	DURATION	COMPLETED
✓	1	bee6096	simple-pipeline	Branch indexing	2s	3 minutes ago
✓	2	—	master	Branch indexing	2s	9 minutes ago
✓	1	232ca79	master	Branch indexing	10s	an hour ago

- Note that the first line is Run #1 for the branch called "simple-pipeline", which is the branch we just created. Because it is a new branch, the "Message" is "Branch indexing".

The Pipeline in the master branch is still there and is unaffected by the presence of this new branch. We are going to work in this branch until our Simple Pipeline is complete and ready for production; then we will merge that back to master.

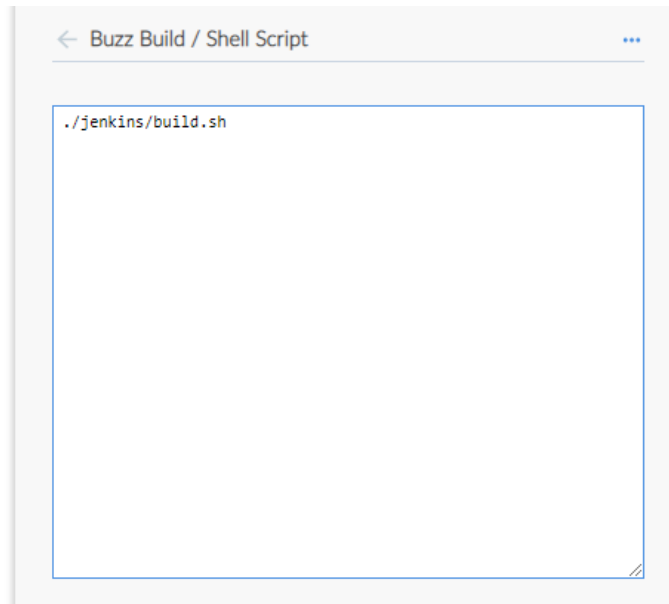
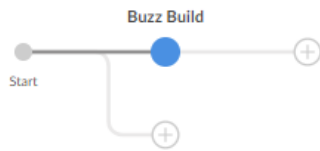
Create simple Pipeline

Now we can add some real functionality to the Pipeline by implementing some `sh` steps that call actual scripts. Your lab includes the scripts that we will use.

Create Buzz Build Stage

To create the `Buzz Build` stage that calls a script to build the software:

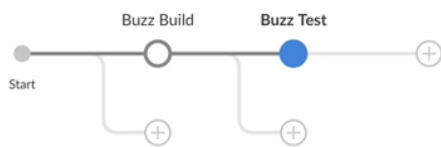
- Click the **simple-pipeline** branch from the **Branches** tab of the Blue Ocean page
- Click the edit icon.
- Rename the "Buzz Buzz" stage to "Buzz Build"
- Delete the "Print Message" step for this stage
- Create a `Shell Script` step
- Type in `./jenkins/build.sh`, which is a Maven script that builds the software.



Add Buzz Test Stage

To create the `Buzz Test` stage that calls a script to build the software:

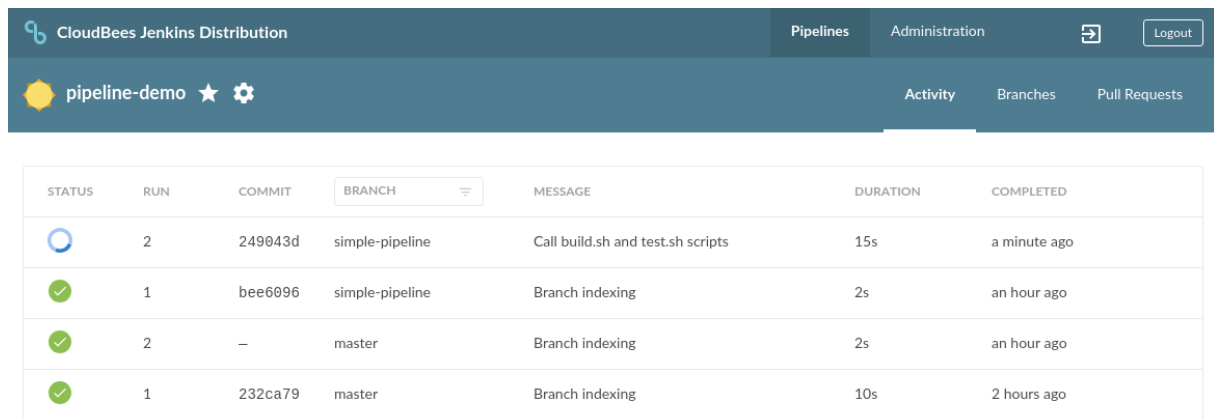
- Add another stage called "Buzz Test"
- Add a step to "Buzz Test" to run the Shell Script, `./jenkins/test-all.sh`
- Save & run the Pipeline, using "Initial build and test" as the Description



Note that, by default, Blue Ocean saves to the branch we last used.

Watch Pipeline Run

Note that our Pipeline is now doing enough work that we can see the swirling blue circle under the status column and the header is still blue. When this run completes successfully, the status switches to a green circle with a checkmark and the header turns green.



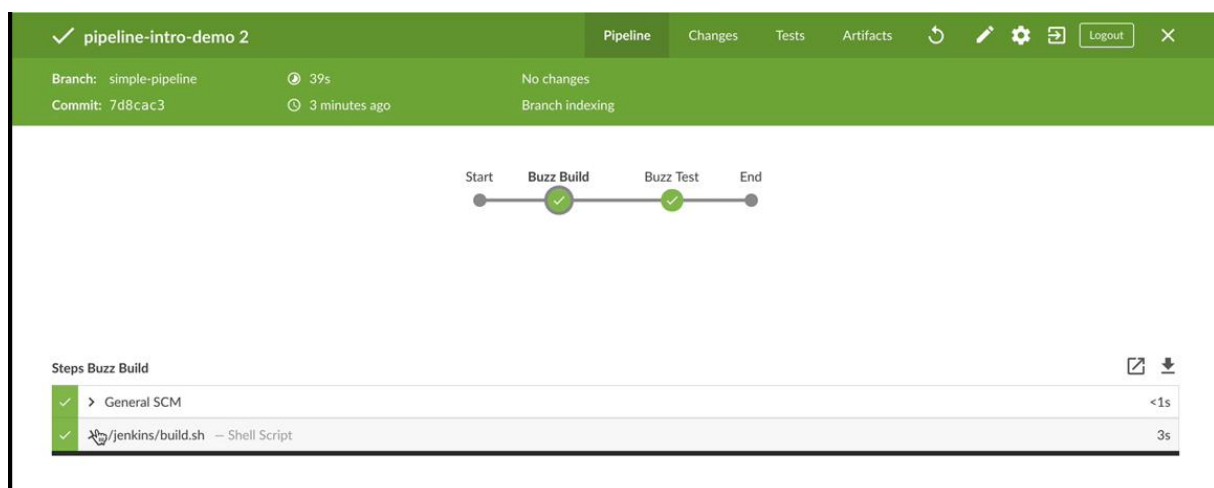
The screenshot shows the Jenkins CloudBees Jenkins Distribution interface. At the top, there's a header with the Jenkins logo, 'CloudBees Jenkins Distribution', and tabs for 'Pipelines' and 'Administration'. Below the header, there's a sub-header with 'pipeline-demo' and a star icon. The main content area displays a table of pipeline runs.

STATUS	RUN	COMMIT	BRANCH	MESSAGE	DURATION	COMPLETED
	2	249043d	simple-pipeline	Call build.sh and test.sh scripts	15s	a minute ago
	1	bee6096	simple-pipeline	Branch indexing	2s	an hour ago
	2	—	master	Branch indexing	2s	an hour ago
	1	232ca79	master	Branch indexing	10s	2 hours ago

View Pipeline run details

The graphical display also allows you to look at details about the Pipeline execution:

- Click on the "Buzz Build" stage in the visual; it shows the SCM step and the step to run the shell script
- Click on the `./jenkins/build.sh` step to see the output of running that script
- Click on the "Buzz Test" stage
- Click on the line for `test.all.sh`; this shows the output of running all the tests



The screenshot shows the Jenkins Pipeline visualizer for a pipeline named 'pipeline-intro-demo 2'. The top bar is green and contains the pipeline name, tabs for 'Pipeline', 'Changes', 'Tests', and 'Artifacts', and a 'Logout' button. Below the bar, there's a summary section with 'Branch: simple-pipeline', 'Commit: 7d8cac3', and 'No changes'. The main visualizer shows a horizontal flow from 'Start' to 'Buzz Build' to 'Buzz Test' to 'End'. The 'Buzz Build' stage is highlighted with a green checkmark. Below the visualizer, there's a section titled 'Steps Buzz Build' with a list of steps: 'General SCM' and './jenkins/build.sh - Shell Script'. The 'General SCM' step is expanded, showing its details.

Artifacts and fingerprints

An **artifact** is a file produced as a result of a Jenkins build. The name comes from Maven naming conventions. A single Jenkins build can produce many artifacts. By default, they are stored on the Jenkins master that ran the Pipeline that created them, not on the agent where the step ran. If they are not archived, they are deleted when the Pipeline completes and the workspace is wiped. Archiving keeps those files in `${JENKINS_HOME}` unless you delete them.

A **fingerprint** is the MD5 checksum of an artifact. Each archived artifact can be fingerprinted; merely check the "Fingerprint" box when you create the archiving step in the Blue Ocean Visual Editor. Jenkins uses fingerprints to keep track of artifacts without any ambiguity.

Most archives should be fingerprinted; the details of how one tracks archives using fingerprints is outside the scope of this class except to say that a *database* of all fingerprints is managed on the master in the `${JENKINS_HOME}/fingerprints` directory.

Archiving artifacts

Projects can be configured to **archive** artifacts based on filename patterns; archived artifacts are available for testing and debugging after the pipeline run finishes. Archived artifacts are kept **forever** unless a **retention** policy is applied to builds to delete them periodically

Patterns for archiving artifacts

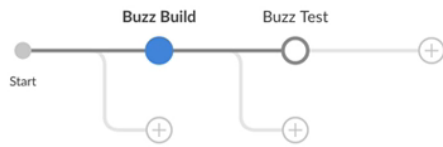
Your Pipeline uses patterns to control which artifacts are archived. For example:

- `my-app.zip`: The file `my-app.zip`, at the workspace's root
- `images/*.png`: All files with the `.png` extension in the `images` folder under the workspace's root
- `target/**/*.jar`: All files with the `.jar` extension, recursively under the `target` folder, under the workspace's root

Archiving artifacts with Blue Ocean

To illustrate how to archive an artifact using the Blue Ocean Visual Editor, we will modify the "Buzz Build" stage:

- From the **simple-pipeline** branch of the demo, click the edit icon.
- In the "Buzz Build" stage, select + **Add step**.
- Search for `archive` to find the `Archive the artifacts` step type.
 - It is much easier to search for the `archive` step type than to scroll through the whole list looking for it.
 - Type `target/*.jar` in the `*Artifacts*` box. to specify the pattern of the artifact to archive.
 - Click the **Fingerprint** box to fingerprint this archive.
- Click **Save** to commit the pipeline to **simple-pipeline** branch with a description of what changed such as `Added artifact support`.



This creates an archive that contains the artifact that is the output of the `build.sh` script. By default, all artifacts are located in the `target` directory and have `.jar` as a suffix.

Jenkinsfile code for archiving artifacts

The Jenkinsfile code that is created includes the `archiveArtifacts` step with `fingerprint: true` specified:

```
pipeline {
  agent any
  stages {
    stage('Buzz Build') {
      steps {
        sh './jenkins/build.sh'
        archiveArtifacts(artifacts: 'target/*.jar', fingerprint: true)
      }
    }
    stage('Buzz Test') {
      steps {
        sh './jenkins/test-all.sh'
      }
    }
  }
}
```

This Pipeline creates a file named `somefile` at the workspace's root, then archives this file. Again, if this file is not archived, it will be deleted when the workspace is deleted after the Pipeline execution completes.

For more details, you can read the [archiveArtifacts](#) step reference page.

Archives — Internal details

Blue Ocean enables you to archive artifacts easily but you may eventually need to understand some of the nitty-gritty details.

Artifacts can be archived in the `post` section of the Pipeline or at the end of the stage that generated the artifact. Jenkins tracks these artifacts forever, across builds, jobs, nodes and folders. Once archived, an archive is attached to the build that produced it.

Artifacts are stored at:

`http://${JENKINS_URL}/job/${YOUR_JOB}/${BUILD_NUMBER}/artifact`

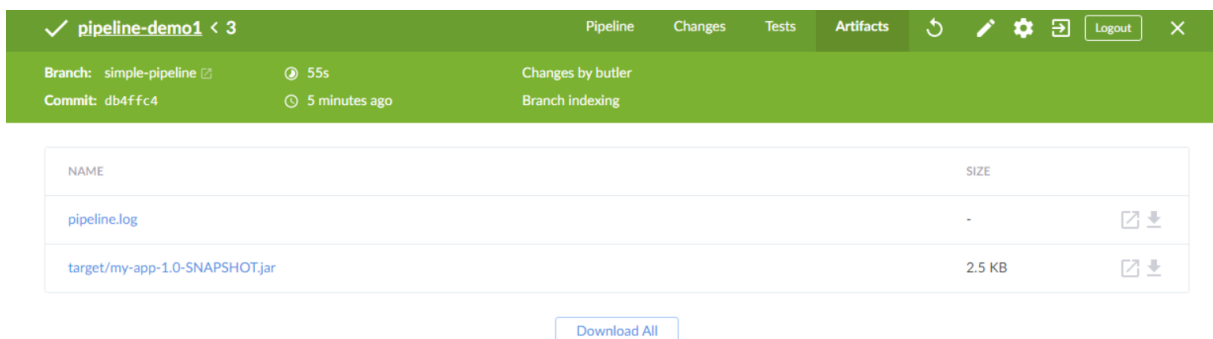
In a production environment, your build chain system (Maven, Gradle, Make, etc.) should publish artifacts to an artifact repository such as Artifactory, Nexus, etc. Teams can also deploy artifacts from Jenkins to test environments. Use the [Copy Artifact Plugin](#) to take artifacts from one project (Pipeline run) and copy them to another Project

Accessing archived artifacts

In Blue Ocean, use the "Artifact" screen for a build to view the artifacts created:



Each Pipeline generates a *pipeline.log* artifact when it runs. Additional artifacts are listed here if the Pipeline is coded to create them. In our example, the artifact is in the `target` directory and is called `my-app-1.0-SNAPSHOT.jar`



Artifacts are also visible on the main Build page in the Classic Jenkins Web UI:

Artifact retention policy

The artifact retention policy is coupled with the build retention policy: deleting a build deletes all attached artifacts.

Important builds can be kept forever but most builds should be deleted regularly. The Jenkins instance can be configured to do this automatically, based on either the age of the build (how many days to keep a build) or number (maximum number of builds to keep).



Discard Old Builds

Strategy: Log Rotation

Days to keep builds: 7

Max # of builds to keep: 20

Advanced...

Add JUnit step

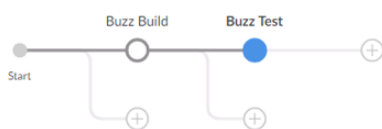
JUnit is a common testing framework for Java programs. The Jenkins [JUnit plugin](#) provides steps that implement JUnit as a publisher which consumes XML test reports and generates some graphical visualization of the historical test results. JUnit provides a web UI for viewing test reports, tracking failures and so forth. It is useful for tracking test result trends and works with all supported build tools. You must specify the appropriate path name for the XML test reports generated by the build tool you are using

In this course, JUnit means the Jenkins publisher that is implemented with the [JUnit plugin](#).

JUnit is very useful for monitoring your test results. Using the JUnit output is a subject for QA and operations people so we do not discuss it in this class, but we can add it to our Pipeline. Because the Pipeline is glue, JUnit works with the XML test reports generated by any build tools.

Implement the JUnit step

JUnit is available as a step for your Pipeline when the JUnit Plugin is installed on your system.



← Buzz Test / Archive JUnit-formatted test results ...

TestResults*

surefire-reports/**/*.xml

☐ AllowEmptyResults

ChecksName

HealthScaleFactor

☐ KeepLongStdio

☐ SkipPublishingChecks

TestDataPublishers

This property type is not supported

To add the `junit` step to your Pipeline with the Blue Ocean Visual Editor:

- From the **simple-pipeline** branch of the demo, click the edit icon.
- Select the **Buzz Test** stage and add a Step called: "Archive JUnit-formatted test results"
- Type the path of the XML files that contain your test reports in the "Test Results" box. For Apache Maven, the path is `**/surefire-reports/**/*.xml`.

- The other configuration options are discussed on the [JUnit plugin](#) page
 - In *most* cases, you can just use the default values
- **Save and Run** the Pipeline, specifying a meaningful description of the changes you made such as "Added JUnit test results configuration".

Error handling in Blue Ocean

This is a good opportunity to see how the Blue Ocean Visual Editor handles an error in the Pipeline. To do this, type `bad_target/**/TEST*.xml` (which is an incorrect path) in the Test Results box then save and Run the Pipeline.

Notice the red header on the Blue Ocean page When the build fails.

- The step to "Archive JUnit-formatted test results" has a red box with an X in it, indicating that it is where the Pipeline failed.
- Click on that step; see that the error message is "No test report files were found. Configuration error?"
 - The problem is that the path for the XML files is invalid, so edit the Pipeline to modify the configuration to have the correct path (`**/surefire-reports/**/*.*.xml`) in the "Test Results" box.
- Now Save and Run the Pipeline, using "Fixed test results" as the "Description"

Some other things to notice:

- The artifacts are being built and show up, even while the Pipeline is running
- Now that the "Buzz Test" stage is doing some work, it takes longer to run the Pipeline
- When the run is complete, click the "Tests" tab in the header and see the "All test are passing" note.
- Look under the "Buzz Build" stage; click on the 2nd step (echo...) and see that it did the replacement on the environment variable to echo "I am a Worker Bee!"

View JUnit step in the Jenkinsfile

The JUnit step is coded in the Pipeline as:

```
steps {  
    sh './jenkins/test-all.sh'  
    junit '**/surefire-reports/**/*.*.xml'  
}
```

For more details, see the [junit](#) step reference.

Set environment variables

Environment variables are set as directives that define key-value pairs and can be used in a Pipeline. They come from various sources:

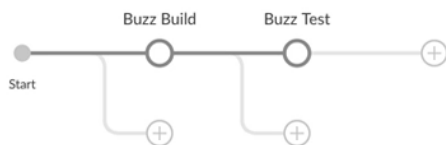
- Jenkins recognizes some environment variables that are set in the shell.
- Jenkins sets its own specific environment variables, such as `BUILD_NUMBER`, `NODE_NAME` and `JOB_NAME`.
- Many plugins define environment variables.

You can see the list of environment variables recognized on your Jenkins instance by following the **Manage Jenkins** → **System information** link on the Jenkins dashboard; we will discuss this more later.

[Jenkins Set Environment Variables](#) lists all environment variables that Jenkins can set. We will discuss using these environment variables in more detail later.

Set your own environment variables

You can set additional environment variables for your Pipeline. They can be set either in the `pipeline` block (applies to the entire Pipeline) or can be set per stage (applies only to that stage).

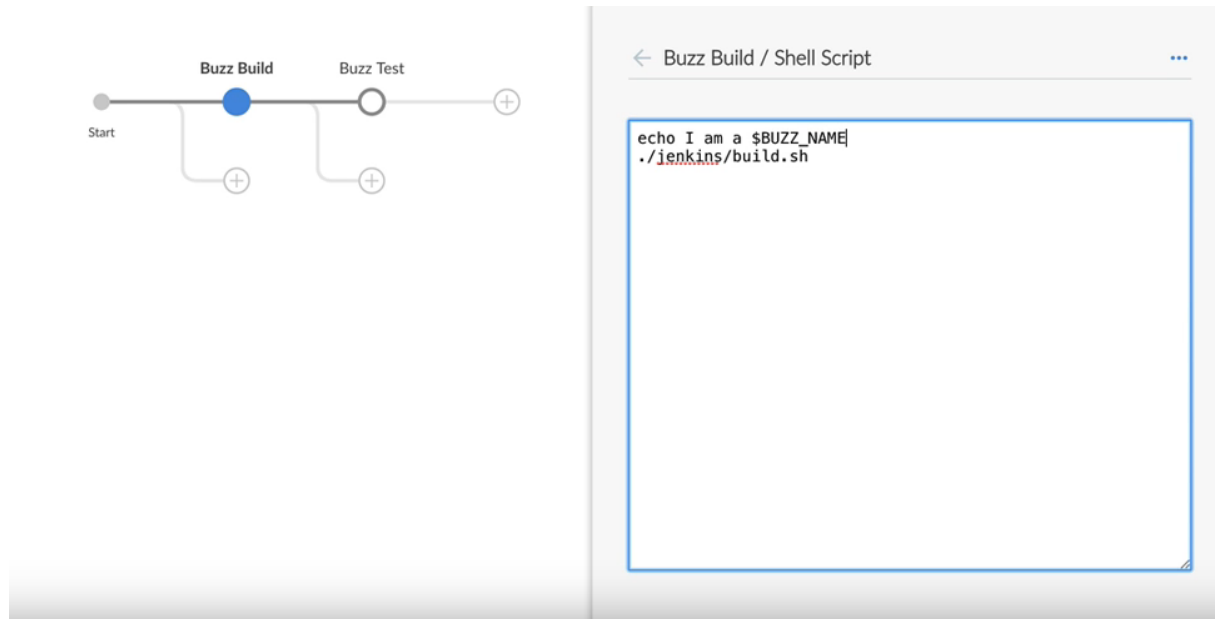


Name	Value
BUZZ_NAME	Worker Bee

To set an environment variable, use the Pipeline Settings screen:

- From the **simple-pipeline** branch of the demo, click the edit icon.
- Click the + sign to the right of the "Name/Value" line under "Environment"
- Type the name of the variable (all capital letters) in the "Name" box
- Type the value of the variable in the "Value" box.
 - For our Demo, set the `BUZZ_NAME` environment variable to have the `Worker Bee` value.

Use environment variables



A simple use of our environment variable:

- Click on the "Buzz Build" stage
- Select the existing shell script for edit
- Add the following line above the existing text in the script:

```
echo "I am a ${BUZZ_NAME}"
```

You can instead use the Blue Ocean Code Editor, a text editor, or the inline editor to edit the Jenkinsfile and add this line above the existing `sh` step.

Now save and run the Pipeline, using the Description: "Added env sample".

We will discuss some more interesting uses of environment variables later in this course.

You can do the following as the build runs to get a better understanding of how Blue Ocean works:

- Go to the Build Details page then to the "Buzz Build" stage; click on the "echo..." line to view the execution of the `echo` line before the `build.sh` script runs
- Go to the "Buzz Test" stage to see the Junit tests running

[View the Jenkinsfile](#)

The Jenkinsfile now contains the following:

```
pipeline {
  agent any
  stages {
    stage('Buzz Build') {
      steps {
        echo "I am a ${BUZZ_NAME}"
        sh './jenkins/build.sh'
        archiveArtifacts(artifacts: 'target/*.jar', fingerprint: true)
      }
    }

    stage('Buzz Test') {
      steps {
        sh './jenkins/test-all.sh'
        junit '**/surefire-reports/**/*.xml'
      }
    }
  }

  environment {
    BUZZ_NAME = 'Worker Bee'
  }
}
```

Define parallel stages

Stages can be run in parallel, which can reduce the execution time for the Pipeline. This is especially useful for long-running stages and builds for different target architectures or operating systems (Debian/Fedora, Android/iOS) or different software versions (JVM8/11), et cetera. Builds and tests of independent modules can be run in parallel.

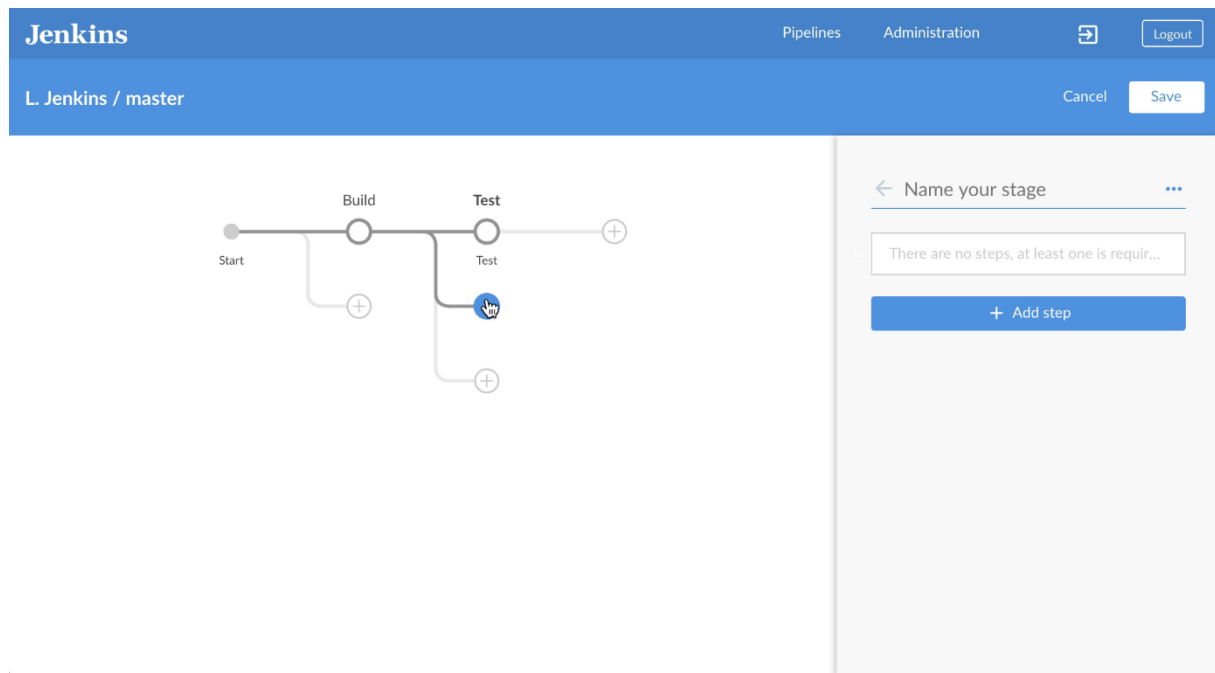
Each "parallelized branch" is a stage. A stage can use either **steps** or **parallel** at the top level; it cannot use both.

Other implementation details for parallel stages are:

- A stage within a **parallel** stage can contain **agent** and **steps** sections.
- A **parallel** stage cannot contain **agent** or **tools** because they are not relevant without steps.
- By default, if one parallel stage fails, the other stages continue to execute. Add `failfast true` to force all parallel processes to abort if one stage fails.

Adding parallel stages in Blue Ocean

Blue Ocean makes it easy to add parallel stages to your pipeline:



- Click on a button under an existing stage to add a parallel stage. In this case, click + under "Buzz Test", and then add steps to that new stage.
- Add as many parallel stages as makes sense.
- Note that Pipeline does not support arbitrary parallel stage depth.

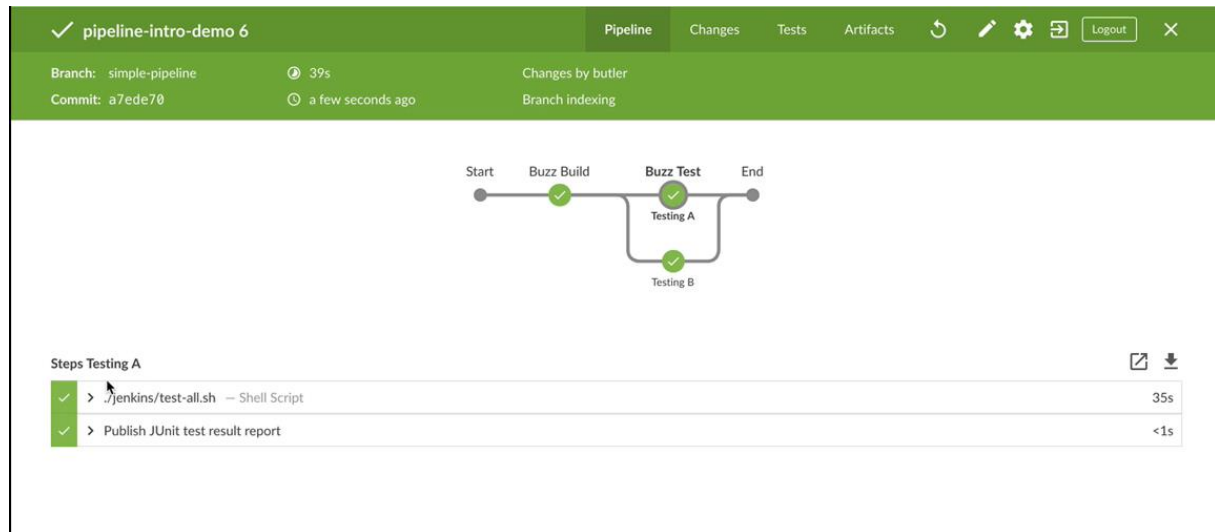
For example, to add Testing A and Testing B stages to the Pipeline:

- From the **simple-pipeline** branch of the demo, click the edit icon.
- Add a parallel stage by clicking + under "Buzz Test"; name it "Testing B".
- Note that when you add this parallel stage, Blue Ocean adds a first stage that is named "Buzz Test"; rename that stage to be "Testing A".
- Add a step to "Testing B"; we're not going to do anything very interesting at this point; just choose "Shell Script" and type in the following:

```
sleep 10
echo done.
```
- Save & run the Pipeline; type "Added Parallel stage" as the description.

Pipeline run details for parallel

Blue Ocean graphically depicts that the stages are running in parallel:



Blue Ocean separates output for steps; this also works across parallel stages. Blue Ocean separates the output for each step in each parallel stage.

While the Pipeline is running, you can click on the details page. You see that "Testing A" is running what it did before. When you click on "Testing B", you see that it now runs the `sleep` and `echo` steps we coded for it.

Note that the Pipeline executed in about the same amount of time as the previous version did because "Testing A" and "Testing B" are running in parallel. Had "Testing B" not started to execute until "Testing A" had completed, the execution time would have increased.

How parallel stages are scheduled

By default, Jenkins tries to allocate a `stage` to the last node on which it (or a similar `stage`) executed. This may mean that multiple parallel steps execute on the same node while other nodes in the cluster are idle.

Pipeline parallel is NOT, by default, a load balancer. The default scheduling is based on:

- SCM updates are more efficient than SCM checkouts
- Some build tools (such as Maven and RVM) use local caches, and they work faster if Jenkins keeps building a job on the same node.

The disadvantage is that some nodes may be left idle while other nodes are overloaded.

Use the [Least Load Plugin](#) to replace the default load balancer with one that schedules a new `stage` to nodes with the least load.

Limits to parallelization

Jenkins does not impose a limit on the number of parallel stages used in a single stage. Using a small number of parallel stages improves the speed of a pipeline but, because each parallel branch uses resources to set up and wait, a very large number of parallel stages may actually degrade the pipeline performance, even if the only statement in each stage is an `echo` statement.

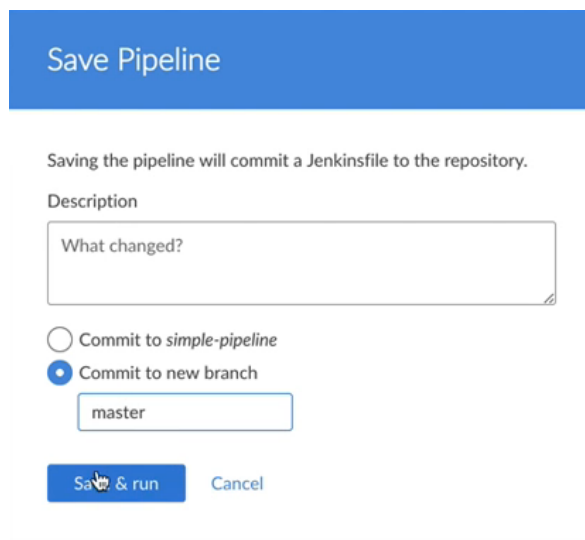
A large number of parallel stages that attempt to pull from a common resource, such as an artifact repository, may also cause performance issues.

Some internal architectural issues between Groovy, the Groovy CPS library, and the JVM `classfile` format occasionally cause even medium-sized pipelines to fail during compilation with errors such as "Method code too large" or "Class too large".

In summary, using a few parallel stages can improve the performance of your Pipeline but you should avoid using too many parallel stages.

Save Pipeline to master

We have completed our Simple Pipeline. To save it to the master branch:



The screenshot shows the 'Save Pipeline' dialog box in Jenkins. At the top, a blue header bar contains the text 'Save Pipeline'. Below this, a message states: 'Saving the pipeline will commit a Jenkinsfile to the repository.' Under the heading 'Description', there is a text input field containing the placeholder text 'What changed?'. Below the input field, there are two radio button options: 'Commit to simple-pipeline' (which is unselected) and 'Commit to new branch' (which is selected). Below these options is a text input field containing the word 'master'. At the bottom of the dialog, there are two buttons: 'Save & run' (highlighted with a mouse cursor) and 'Cancel'.

- From the pipeline edit page, click the "Save" button to save the Pipeline to the master branch; no description is required because we are just pushing all the changes we have made to master
- Click **Commit to new branch** and type "master" in the box.
- Click **Save and Run**.

The common practice is to use Git Pull Requests to commit to master but you can do it through Blue Ocean.

Do it your self Lab: Create and edit a simple pipeline with parallel stages

In this exercise you will:

- Create and run a Pipeline in a feature branch.
- Add artifacts and test results and publish them to a Pipeline.
- Add parallel stages to a Pipeline.
- Save a Pipeline to the master branch.

Before you begin this lab, ensure that your Pipeline matches the solution from the previous lab: Create and edit an example pipeline. Use the Blue Ocean Visual Editor and/or the Blue Ocean Code Editor to make any necessary changes.

Task: Create and run a Pipeline in a feature branch

Edit your pipeline from the `master` branch:

1. In Blue Ocean, from the `pipeline-lab` branch, select the edit icon.
2. Delete all the steps from all three stages that you created. Notice that stages show errors because they do not have steps.
3. Add a step to each of the stages:
 - a. Add a **Shell Script** step to the Fluffy Build stage: `./jenkins/build.sh`.
 - b. Add a **Shell Script** step to the Fluffy Test stage: `./jenkins/test-all.sh`.
 - c. Add a **Shell Script** step to the Fluffy Deploy stage: `./jenkins/deploy.sh staging`.
4. Save the pipeline:
 - a. In the **Save** dialog, provide the description `Create Simple Pipeline`.
 - b. Select **Commit to new branch** and enter `simple-pipeline`.
 - c. Select **Save & run** to commit to the `simple-pipeline` branch.

Task: Add artifacts and test results and publish them to a Pipeline

Edit your pipeline from the `simple-pipeline` branch:

1. From the `simple-pipeline` branch you just created, select the edit icon.
2. Add a step to save the artifacts to the Fluffy Build stage:
 - a. Add **Archive the artifacts** and enter `target/*.jar` in `*Artifacts*`.
3. Add a step to publish test results to the Fluffy Test stage:
 - a. Add **Archive JUnit-formatted test results** and enter `target/**/*.TEST*.xml` in `TestResults`.
4. Save the pipeline. In the **Save** dialog, provide the description `Publish artifacts and test results` and save it to the default branch (`simple-pipeline` branch).

Task: Add parallel stages to a Pipeline

Edit your pipeline from the `simple-pipeline` branch:

1. Add three parallel stages to Fluffy Test to create a total of four parallel stages:
 - a. Name the three new parallel stages Frontend, Performance, and Static.
 - b. Delete all the steps from the Fluffy Test stage and rename the stage Backend.
 - c. See that there are four parallel stages under Fluffy Test.
2. Make each stage only have one step:
 - a. Add a **Shell Script** step to the Backend stage: `./jenkins/test-backend.sh`.
 - b. Add a **Shell Script** step to the Frontend stage: `./jenkins/test-frontend.sh`.
 - c. Add a **Shell Script** step to the Performance stage: `./jenkins/test-performance.sh`.
 - d. Add a **Shell Script** step to the Static stage: `./jenkins/test-static.sh`.
3. Add a second step to the Backend and Frontend stages:
 - a. Add an **Archive JUnit-formatted test results** step to the Backend stage. Enter `target/surefire-reports/**/TEST*.xml` in **TestResults**.
 - b. Add an **Archive JUnit-formatted test results** step to the Frontend stage. Enter `target/test-results/**/TEST*.xml` in **TestResults**.
4. Save the pipeline. In the **Save** dialog, provide the description Add parallel test stages and save to the default branch (simple-pipeline branch).
5. Review the pipeline run results.

Notice that the pipeline completed much more quickly using parallel stages.

Task: Save a Pipeline to the master branch

Edit your pipeline from the `simple-pipeline` branch:

1. Save the pipeline. In the **Save** dialog, provide the description Change to Simple Pipeline in master.
2. Select **Commit to new branch** and enter `master`.
3. Select **Save & run** to commit to the `master` branch.

Solution

Jenkinsfile (final)

```
pipeline {
  agent any
  stages {
    stage('Fluffy Build') {
      steps {
        sh './jenkins/build.sh'
        archiveArtifacts 'target/*.jar'
      }
    }
    stage('Fluffy Test') {
      parallel {
        stage('Backend') {
          steps {
            sh './jenkins/test-backend.sh'
            junit 'target/surefire-reports/**/TEST*.xml'
          }
        }
        stage('Frontend') {
          steps {
            sh './jenkins/test-frontend.sh'
            junit 'target/test-results/**/TEST*.xml'
          }
        }
        stage('Performance') {
          steps {
            sh './jenkins/test-performance.sh'
          }
        }
        stage('Static') {
          steps {
            sh './jenkins/test-static.sh'
          }
        }
      }
    }
    stage('Fluffy Deploy') {
      steps {
        sh './jenkins/deploy.sh staging'
      }
    }
  }
}
```