

MULTIBRANCH PIPELINE

If you are looking for a well-automated Pull Request based or branch-based Jenkins **Continuous Integration & Delivery** (CI/CD) pipeline, this guide will help you get the overall picture of how to achieve it using the Jenkins multibranch pipeline.

Jenkins's multi-branch pipeline is one of the best ways to design CI/CD workflows as it is entirely a git-based (source control) pipeline as code. This guide will talk about all the key concepts involved in a Jenkins multi-branch pipeline setup.

Jenkins Multibranch Pipeline Fundamentals

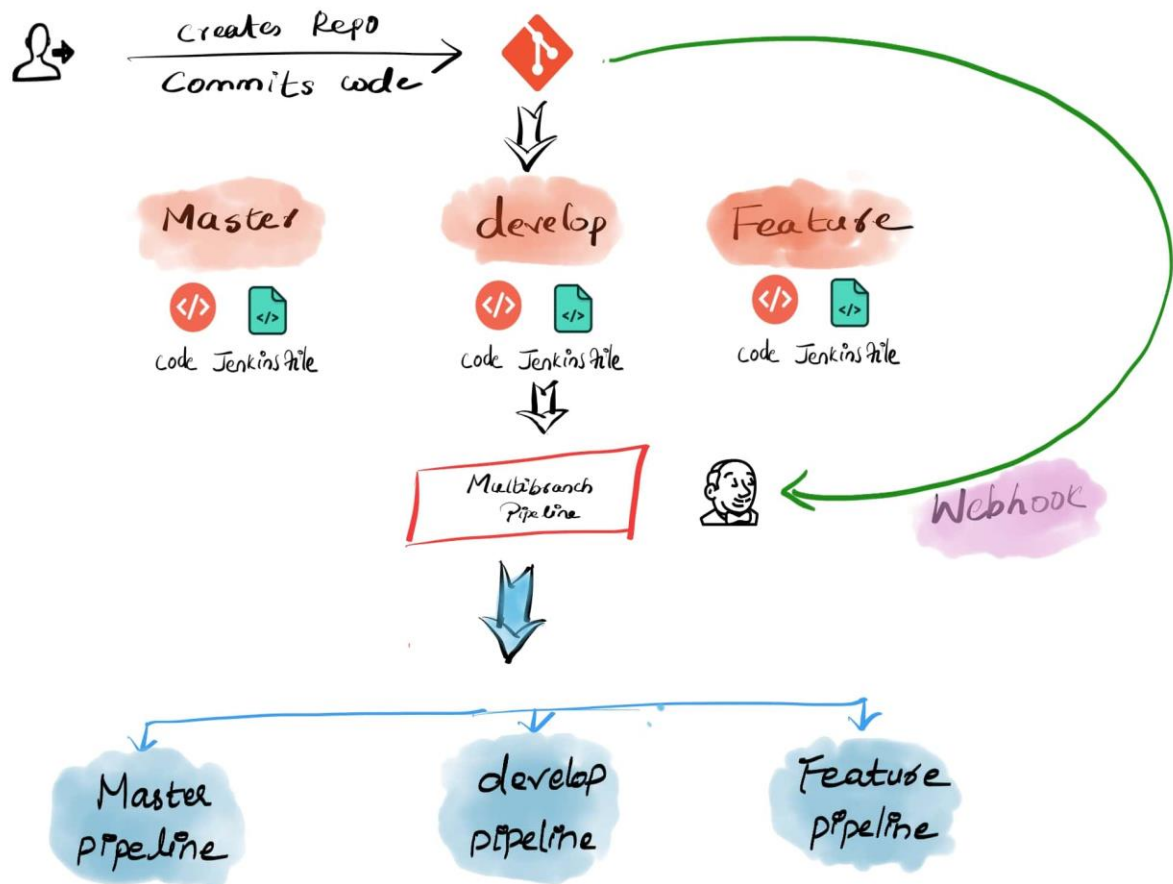
Let's start with the multi-branch pipeline basics. Specifically, in this section, I will cover the concept of a multi-branch pipeline and why it is essential to use it for all Jenkins CI/CD pipelines. I'll also show you how a multi-branch pipeline works with a detailed workflow diagram.

What is a Multi-branch Pipeline?

A multi-branch pipeline is a concept of automatically creating Jenkins pipelines based on Git branches. It can automatically discover new branches in the source control (Github) and automatically create a pipeline for that branch. When the pipeline

build starts, Jenkins uses the Jenkinsfile in that branch for build stages.

SCM (Source Control) can be Github, Bitbucket, or a Gitlab repo.



You can choose to exclude selected branches if you don't want them to be in the automated pipeline with Java regular expressions.

Multi-branch pipeline supports PR based branch discovery. Meaning, branches get discovered automatically in the pipeline if someone raises a PR (pull request) from a branch. If you have this configuration enabled, builds will get triggered only if a PR is raised. So if you are looking for a PR based Jenkins build workflow, this is a great option.

You can add conditional logic to the Jenkinsfile to build jobs based on the branch requirement.

For example, if you want the feature branch to run only unit testing and sonar analysis, you can have a condition to skip the deployment stage with a when a condition, as shown below.

```
stage('Deploy for production') {  
    when {  
        branch 'production'  
    }  
    steps {  
        ----  
    }  
}
```

So whenever the developer raises the PR from the feature branch to some other branch, the pipeline will run the unit testing and sonar analysis stages skipping the deployment stage.

Also, multi-branch pipelines are not limited to the continuous delivery of applications. You can use it to manage your infrastructure code as well.

One such example is having a continuous delivery pipeline for Docker image or a VM image patching, building, and upgrade process.

How Does a Multi-Branch Pipeline work?

I will walk you through a basic build and deployment workflow to understand how a multi-branch pipeline work.

Let's say. I want a Jenkins pipeline to build and deploy an application with the following conditions

1. Development starts with a feature branch by developers committing code to the feature branch.
2. Whenever a developer raises a PR from the feature branch to develop a branch, a Jenkins pipeline should trigger to run a unit test and static code analysis.
3. After testing the code successfully in the feature branch, the developer merges the PR to the develop branch.
4. When the code is ready for release, developers raise a PR from the develop branch to the master. It should trigger a build pipeline that will run the unit test cases, code analysis, push artifact, and deploys it to dev/QA environments.

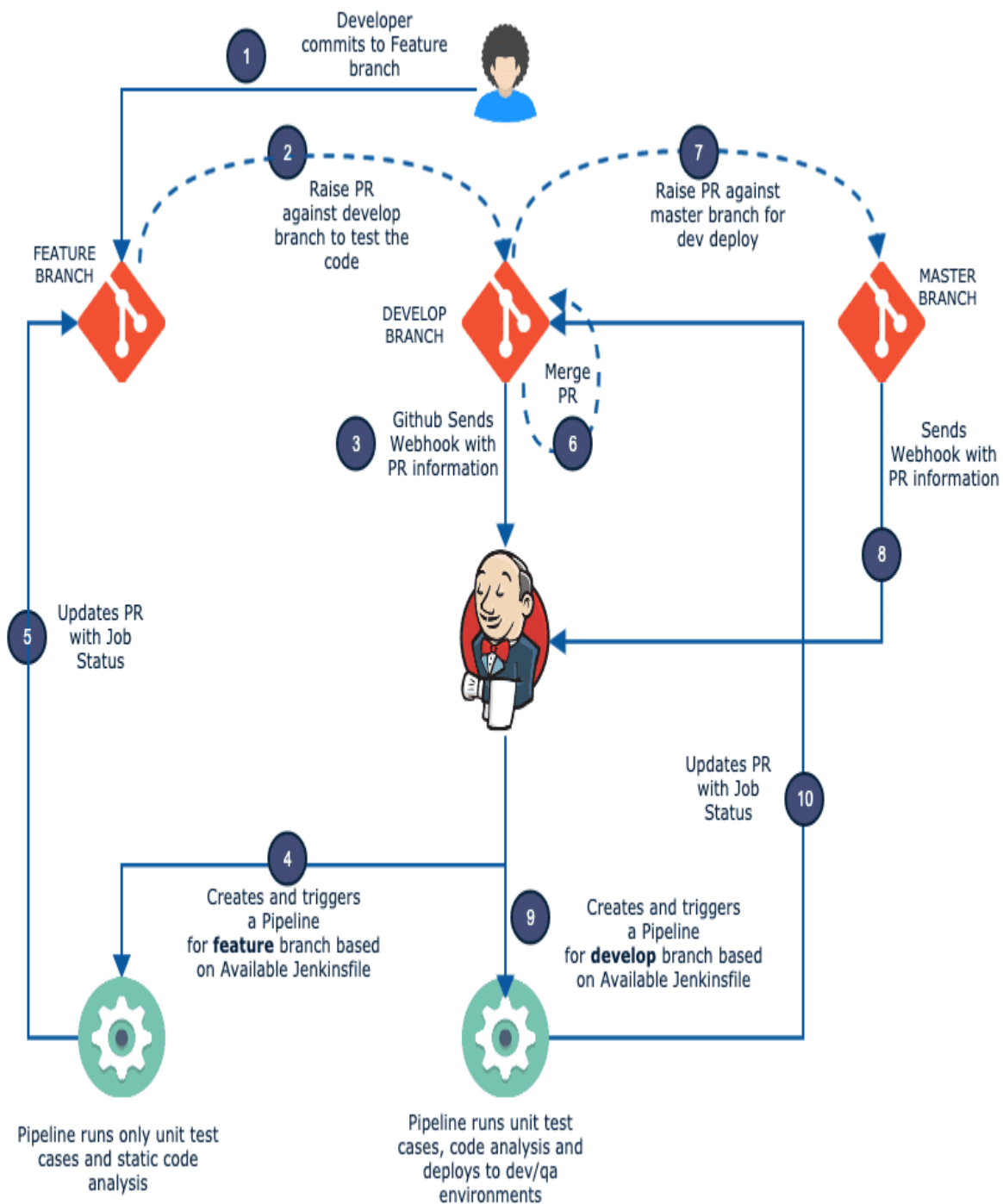
From the above conditions, you can see that there is no manual trigger of Jenkins jobs, and whenever there is a pull request for a branch, the pipeline needs to be triggered automatically and run the required steps for that branch.

This workflow builds a great feedback loop for engineers and avoids dependence on the DevOps team to build and deploy in non-prod environments.

Developer can check the build status on Github and take decisions on what to do next.

This workflow can be achieved easily through a Jenkins multi-branch pipeline.

The following image shows how a multi-branch pipeline workflow would look like for the above example build process



Here is how the multi-branch pipeline works.

1. When a developer creates a PR from a feature branch to develop a branch, Github sends a webhook with the PR information to Jenkins.
2. Jenkins receives the PR and finds the relevant multibranch pipeline, and creates a feature branch pipeline automatically. It then runs the jobs with the steps mentioned in the Jenkinsfile from the feature branch. During checkout, the source and target branches in the PR gets merged. The PR merge will be blocked on Github until a build status from Jenkins is returned.
3. Once the build finishes, Jenkins will update the status to Github PR. Now you will be able to merge the code. If you want to check the Jenkins build logs, you can find the Jenkins build log link in the PR status.

Multibranch Pipeline Jenkinsfile

Before jumping into implementation, let's look at multibranch pipeline Jenkins example Jenkinsfile that can be used in the pipeline.

For the multibranch pipeline to work, you need to have the Jenkinsfile in the SCM repo.

If you are learning/testing, you can use the multibranch pipeline Jenkinsfile given below. It has a checkout stage and other dummy stages, which echoes the message.

Also, you can clone and use [this Github repo](#) which has this Jenkinsfile

Note: Replace the agent label "master" with your Jenkins agent name. master will also work but wouldn't advise it running in actual project environments.

```
pipeline {
```

```
    agent {
```

```
        node {
```

```
            label 'master'
```

```
        }
```

```
    }
```

```
    options {
```

```
    buildDiscarder logRotator(  
        daysToKeepStr: '16',  
        numToKeepStr: '10'  
    )  
}
```

```
stages {
```

```
    stage('Cleanup Workspace') {
```

```
        steps {
```

```
            cleanWs()
```

```
            sh """
```

```
            echo "Cleaned Up Workspace For Project"
```

```
            """
```

```
        }
```

```
}
```

```
stage('Code Checkout') {
```

```
  steps {
```

```
    checkout([
```

```
      $class: 'GitSCM',
```

```
      branches: [[name: '*/main']],
```

```
      userRemoteConfigs: [[url:  
'https://github.com/spring-projects/spring-petclinic.git']]
```

```
    ])
```

```
  }
```

```
}
```

```
stage(' Unit Testing') {
```

```
  steps {
```

```
    sh """  
  
    echo "Running Unit Tests"  
  
    """  
  
}  
  
}
```

```
stage('Code Analysis') {  
  
    steps {  
  
        sh """  
  
        echo "Running Code Analysis"  
  
        """  
  
    }  
  
}
```

```
stage('Build Deploy Code') {
```

```
when {  
    branch 'develop'  
}  
  
steps {  
    sh ""  
  
    echo "Building Artifact"  
  
    ""  
  
    sh ""  
  
    echo "Deploying Code"  
  
    ""  
  
}  
}
```

}

Setup Jenkins Multi-branch Pipeline

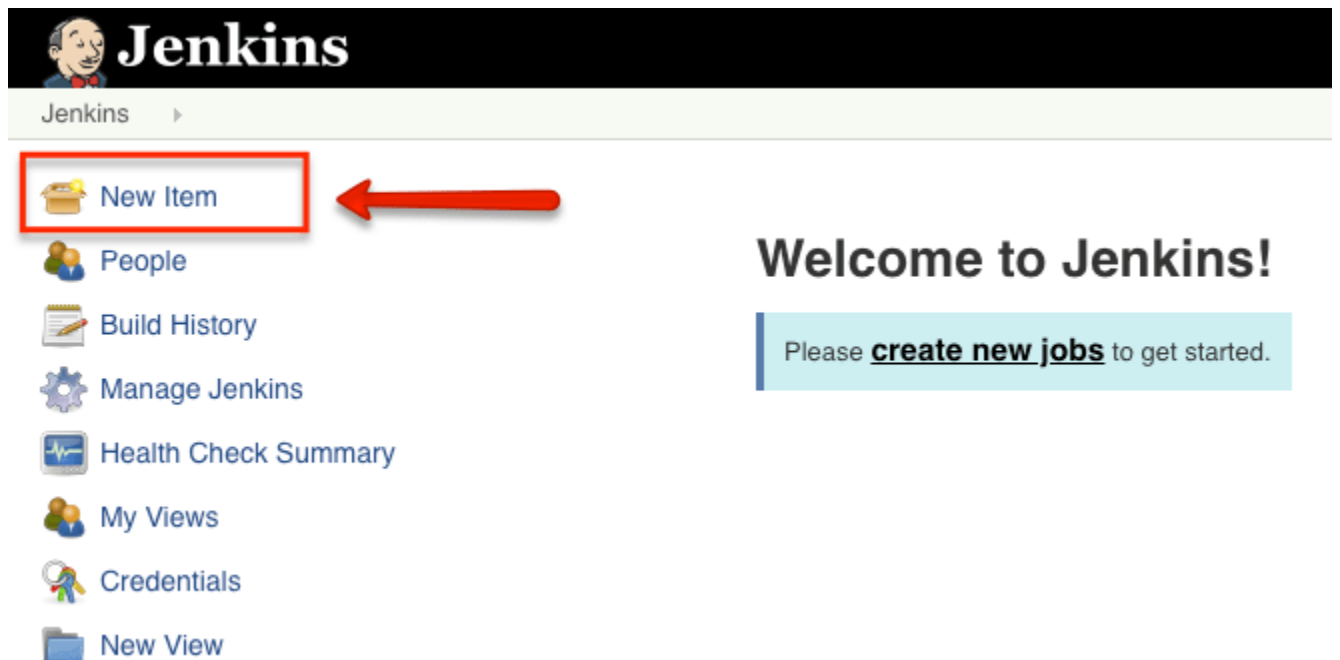
Here I will walk you through the step by step process of setting up a multi-branch pipeline on Jenkins.

This setup will be based on Github and latest Jenkins 2.x version.

You can also use Bitbucket or Gitlab as SCM source for a multi-branch pipeline

Create Multibranch Pipeline on Jenkins (Step by Step Guide)

Step 1: From the Jenkins home page create a "new item".



Step 2: Select the “Multibranch pipeline” from the option and 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 agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.



External Job

This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.



Multi-configuration project

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



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.

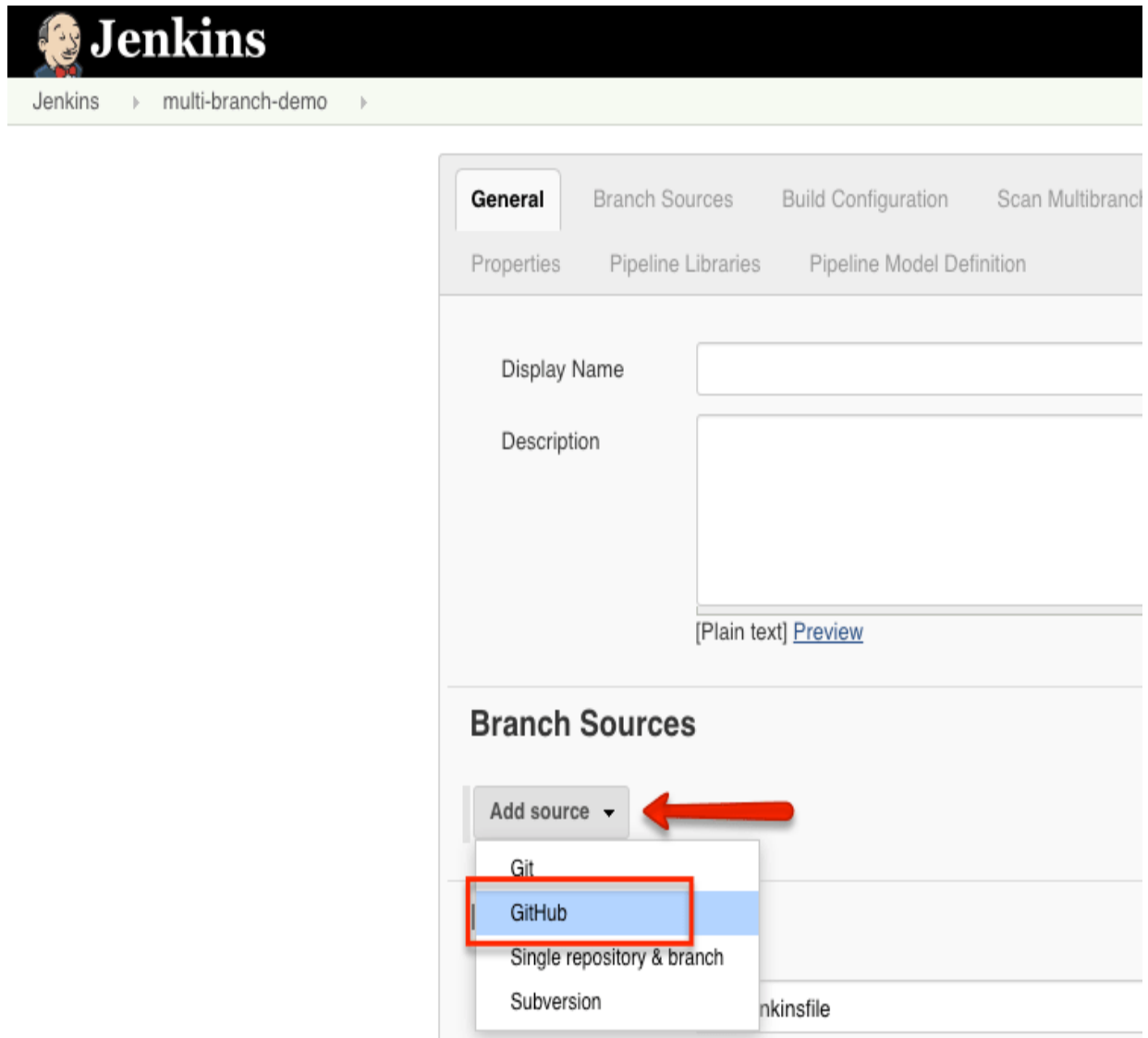


Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.

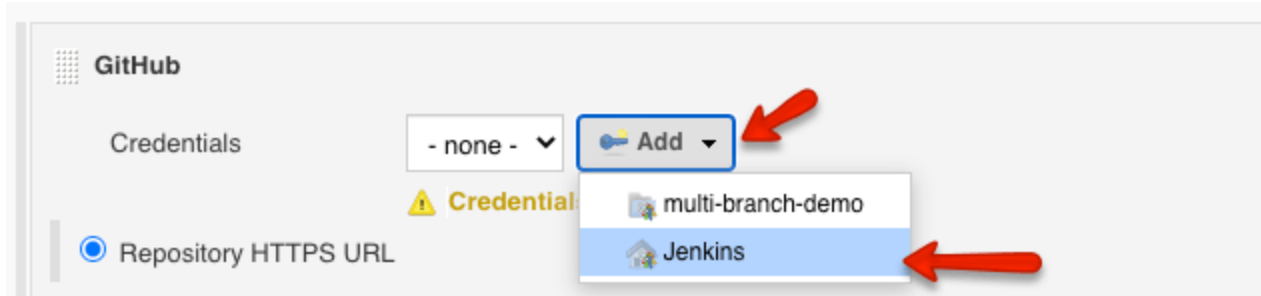


Step 3: Click “Add a Source” and select Github.



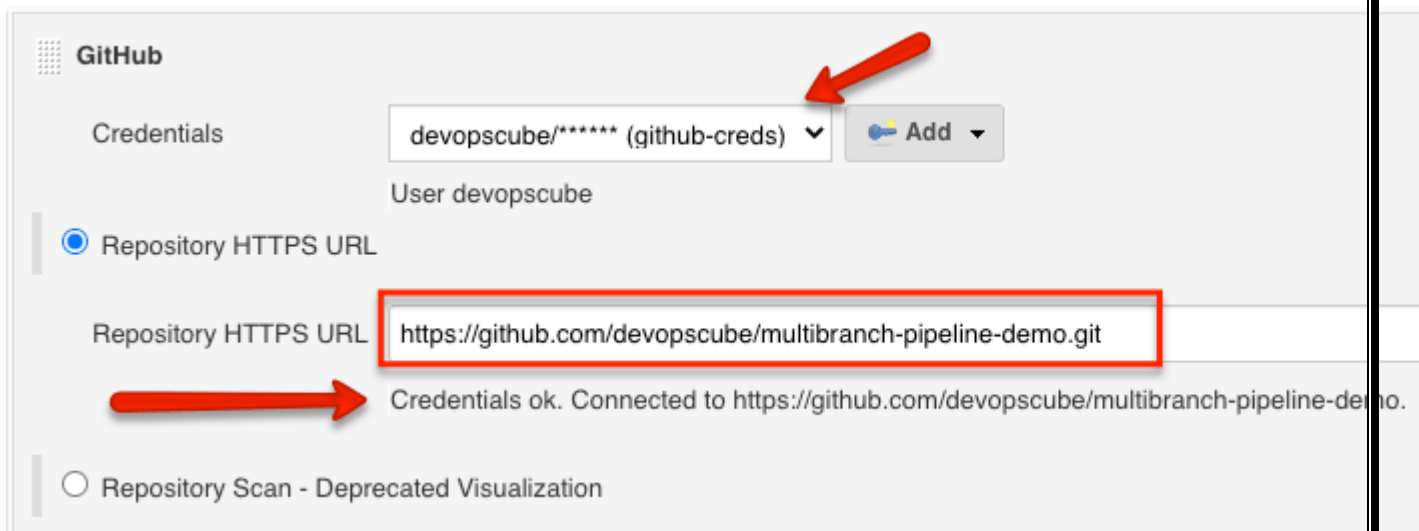
The image shows the Jenkins web interface for configuring a multi-branch project. At the top, the Jenkins logo and the breadcrumb "Jenkins > multi-branch-demo" are visible. Below this, a series of tabs are shown: "General" (selected), "Branch Sources", "Build Configuration", "Scan Multibranch", "Properties", "Pipeline Libraries", and "Pipeline Model Definition". In the "General" tab, there are input fields for "Display Name" and "Description". Below these is a "[Plain text] Preview" link. The "Branch Sources" section contains an "Add source" dropdown menu. A red arrow points to this menu, and a red box highlights the "GitHub" option in the dropdown list. Other options in the list include "Git", "Single repository & branch", and "Subversion".

Step 4: Under the credentials field, select Jenkins, and create a credential with your Github username and password.



Step 5: Select the created credentials and provide your Github repo to validate the credentials as shown below.

If you are testing multi-branch pipeline you can clone the demo Github repo and use it. <https://github.com/devopscube/multibranch-pipeline-demo>.



Step 6: Under "Behaviours" select the required option matches your requirement. You can either choose to discover all the branches in the repo or only branches with a Pull Request.

The pipeline can discover branches with a PR from a forked repo as well.

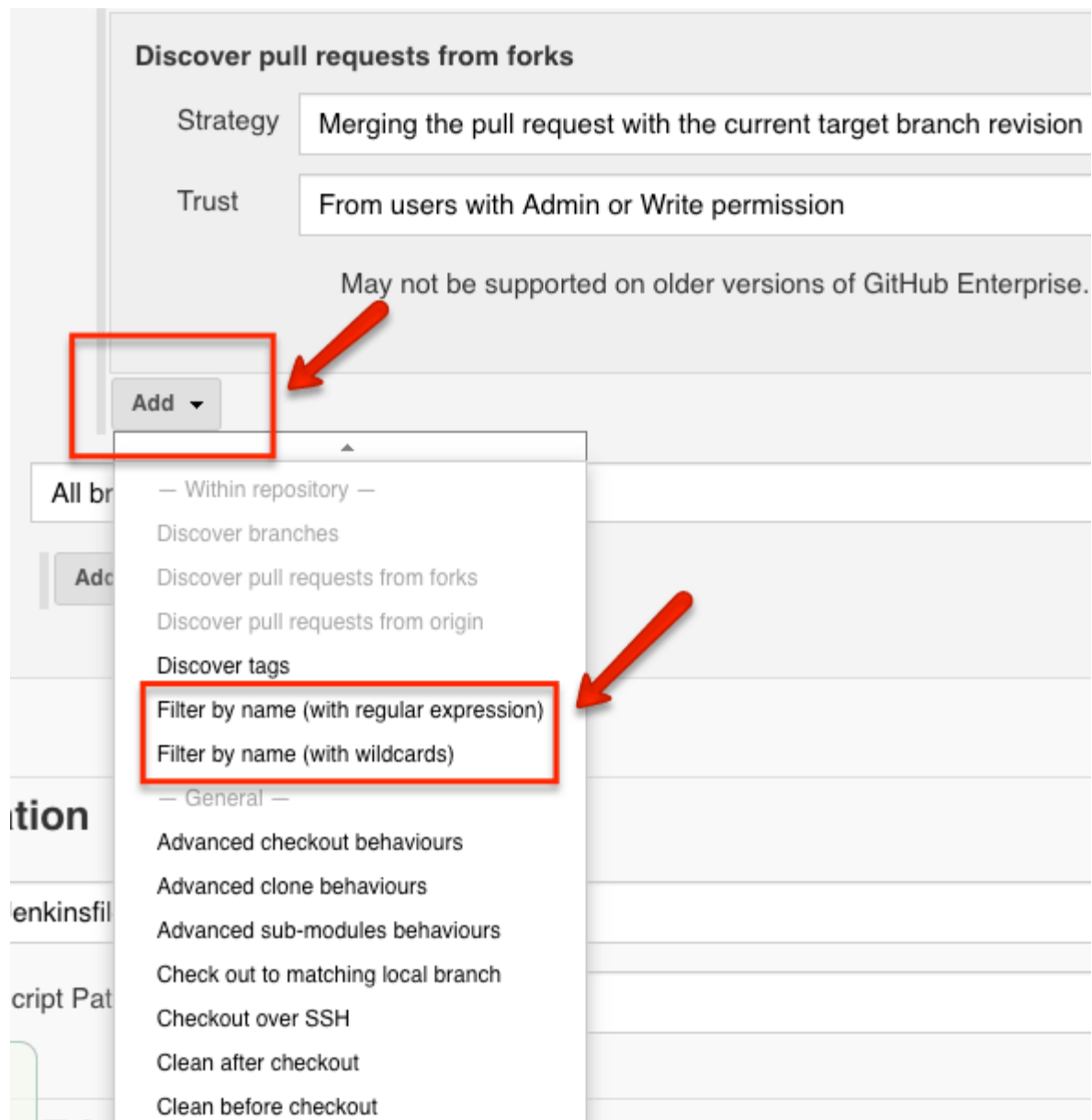
Choosing these options depends on your required workflow.

The screenshot displays the GitHub Actions configuration interface with three main sections, each featuring a red 'X' button and a help icon (question mark):

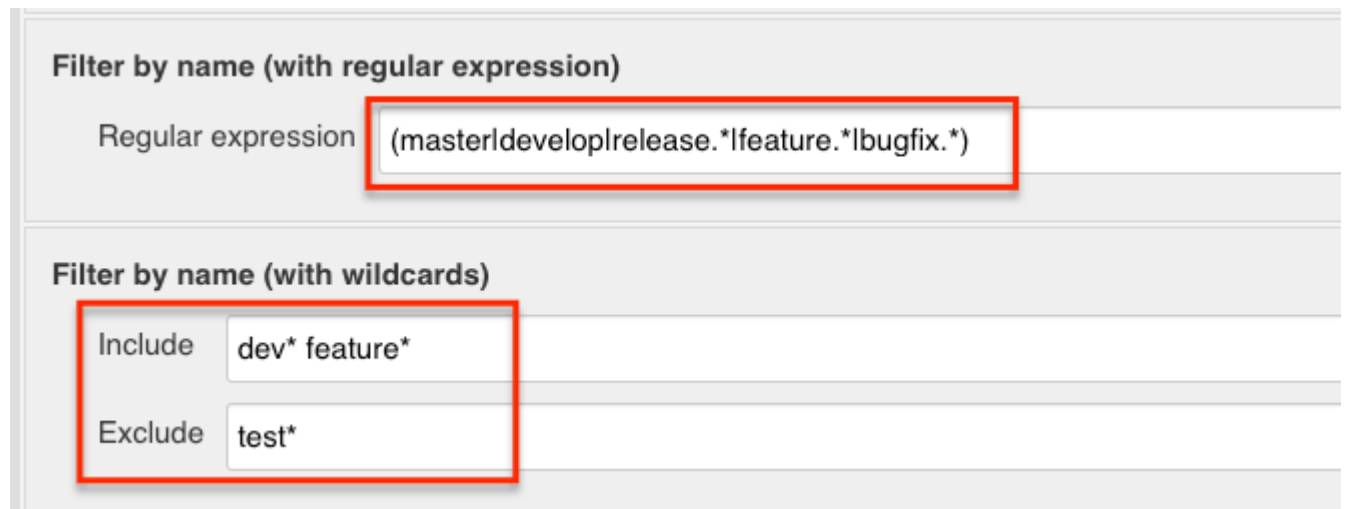
- Discover branches**: A dropdown menu is open, showing three options: 'Exclude branches that are also filed as PRs', '✓ Only branches that are also filed as PRs' (selected), and 'All branches'.
- Discover pull requests from origin**: A dropdown menu is set to 'Merging the pull request with the current target branch revision'.
- Discover pull requests from forks**: This section includes two dropdown menus: 'Strategy' (set to 'Merging the pull request with the current target branch revision') and 'Trust' (set to 'From users with Admin or Write permission'). Below these is a note: 'May not be supported on older versions of GitHub Enterprise. See help button.'

There are additional behavior you can choose from the "add" button.

For example, If you choose not to discover all the branches from the repo, you can opt for the regular expression or wildcard method to discover branches from the repo as shown below.



Here is a regex and wildcard example.

A screenshot of the Jenkins build configuration interface. It shows two sections: 'Filter by name (with regular expression)' and 'Filter by name (with wildcards)'. The first section has a text input field for 'Regular expression' containing the value '(master|develop|release.*|feature.*|bugfix.*)'. The second section has two text input fields: 'Include' with the value 'dev* feature*' and 'Exclude' with the value 'test*'. Red rectangular boxes highlight the 'Regular expression' field and the 'Include'/'Exclude' fields in the second section.

Filter by name (with regular expression)

Regular expression (master|develop|release.*|feature.*|bugfix.*)

Filter by name (with wildcards)

Include dev* feature*

Exclude test*

Step 7: If you choose to have a different name for Jenkinsfile, you can specify it in the build configuration. In the "Script Path" option, you can provide the required name. Ensure the Jenkinsfile is present in the repo with the same name you provide in the pipeline configuration.

Also, Enable "Discard old builds" to keep only required build logs as shown below.

Build Configuration

Mode

by Jenkinsfile

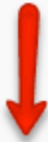
Script Path

Jenkinsfile

Scan Multibranch Pipeline Triggers

☐ Periodically if not otherwise run

Orphaned Item Strategy



☒ Discard old items

Jobs for removed SCM heads (i.e. deleted branches) can be removed immediately or kept based on a desired retention strategy. Jenkins removes them as soon as Jenkins determines their associated SCM head no longer exists. As an example, it may be useful to have a retention strategy to be able to examine build results of a branch after it has been removed.

Days to keep old items

365

if not empty, old items are only kept up to this number of days

Max # of old items to keep

15

if not empty, only up to this number of old items are kept

Step 8: Save all the job configurations. Jenkins scans the configured Github repo for all the branches which has a PR raised.

The following image shows the job scanning the three branches, and since I haven't raised any pull request, Jenkins won't create any branch-based pipeline. I will show how to test the automatic pipeline creation after the webhook setup.

Scan Repository Log

```
Started
[Sat Jun 13 14:02:05 UTC 2020] Starting branch indexing...
14:02:06 Connecting to https://api.github.com using devopscube/***** (jenkins-g:
Examining devopscube/multibranch-pipeline-demo

Checking branches...

Getting remote branches...

Checking branch master

Getting remote pull requests...

Checking branch develop


Checking branch feature

3 branches were processed

Checking pull-requests...
0 pull requests were processed

Finished examining devopscube/multibranch-pipeline-demo

[Sat Jun 13 14:02:07 UTC 2020] Finished branch indexing. Indexing took 1.4 sec
Finished: SUCCESS
```



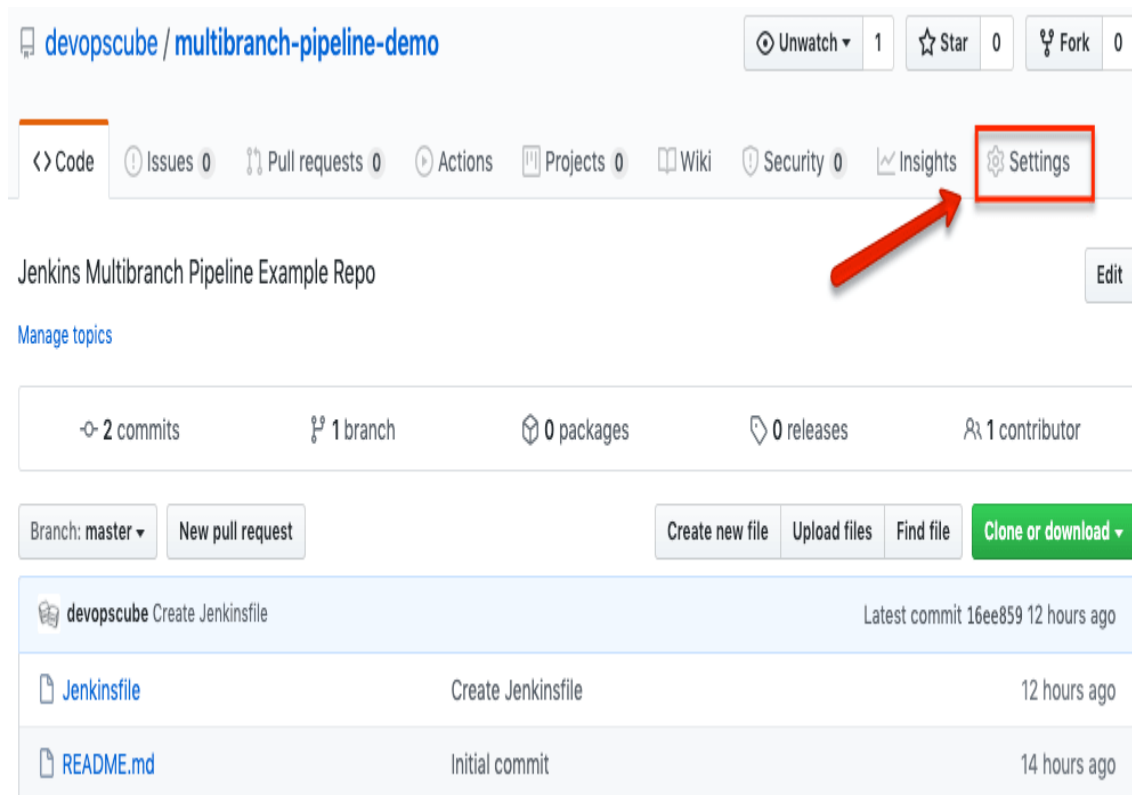
Till now, we have done configurations on the Jenkins side to scan branches based on the PR requests.

To have a complete workflow, we need to have a webhook configured in Github to send all the repo events (commits, PR etc) to Jenkins as the pipelines can be triggered automatically.

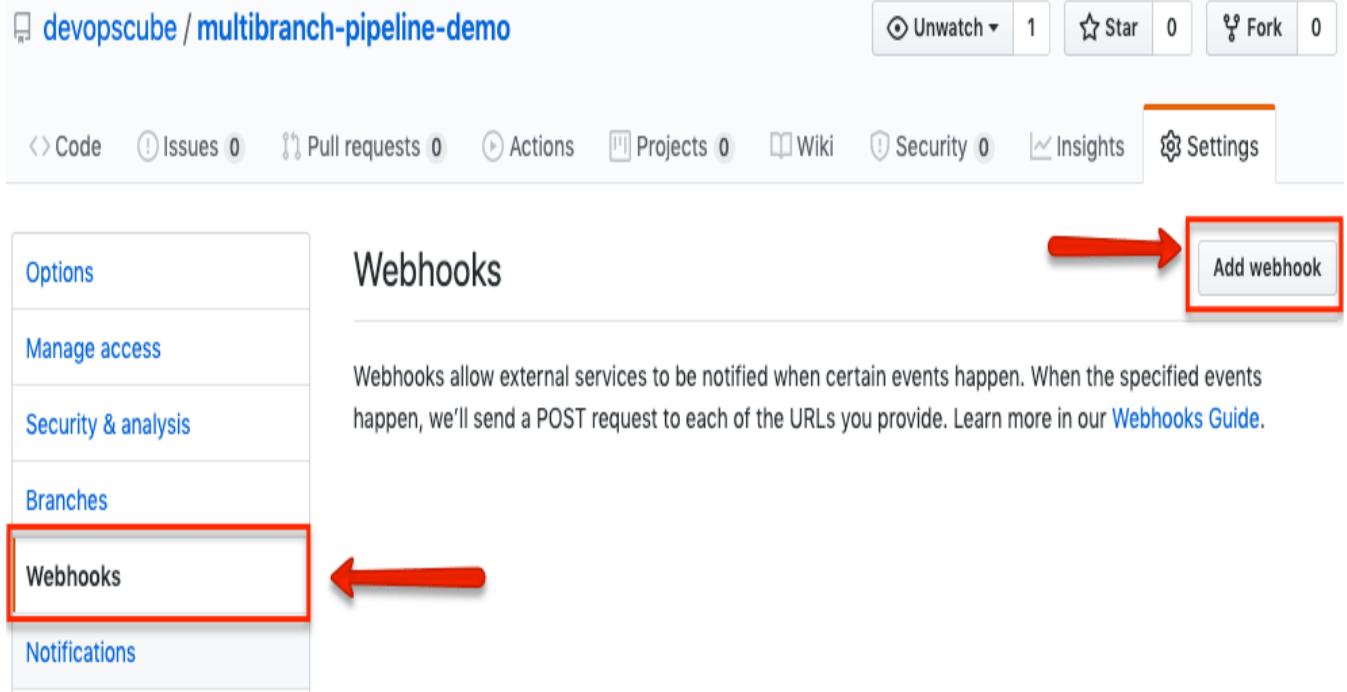
Configure Webhook For Multibranch Pipeline

Follow the steps given below to setup the Jenkins webhook on the repo.

Step 1: Head over to the Github repo and click on the settings.



Step 2: Select the **webhook** option at the left and click “Add Webhook” button.



Step 3: Add your Jenkins URL followed by “/github-webhook/” under payload URL. Select the content type as “application/json” and click “Add Webhook”

Note: You can choose what type of webhook you want to receive in Jenkins. For example, you want to trigger the pipeline only during PR; then, you can select just the PR event from the “Let me select individual events” option.

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found [documentation](#).

Payload URL *

http://35.239.76.164:8080/github-webhook/

Content type

application/json

Secret

Which events would you like to trigger this webhook?

☐ Just the push event.

☒ Send me everything.

☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.


Add webhook

You should see a green tick mark on a successful webhook configuration as shown below.

Webhooks

Add webhook



Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).

 <http://35.239.76.164:8080/github-webhook/> (push)

EditDelete

If you don't see a green tick or see a warning sign, click on the webhook link, scroll down to "Recent Deliveries," and click on the last webhook. You should be able to view why the webhook delivery failed with the status code.

Recent Deliveries

  f70ee380-ad92-11ea-8ad0-59bc4bcf75bd

Request

Response **200**

Redeliver

Headers

```
Request URL: http://35.239.76.164:8080/github-webhook/
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/25c8cea
X-GitHub-Delivery: f70ee380-ad92-11ea-8ad0-59bc4bcf75bd
X-GitHub-Event: ping
```

Now we are done with all the required configurations for the multi-branch pipeline. The next step is to test the multi-branch pipeline workflow triggers.

Test Multi-branch Pipeline


For demo purpose, I have chosen the option "Only Branches that are file as PR". With this option, only the branches with a PR request gets discovered.


To play around with a multi-branch pipeline, you can use this repo with a sample Jenkinsfile -> [Multibranch Pipeline Demo Repo](#)

This repo has three branches. master, develop and feature.


Update some content in the readme file in the feature branch and raise a PR to develop. It will send a webhook to Jenkins and Jenkins will send back the Jenkins job details and the PR will go to check state as shown below.


Add more commits by pushing to the **feature** branch on **devopscube/multibranch-pipeline-demo**.





 **Some checks haven't completed yet** [Hide all checks](#)


2 pending checks



 **continuous-integration/jenkins/branch** Pending — This commit is being... [Details](#)




 **continuous-integration/jenkins/pr-merge** Pending — This commit is bei... [Details](#)



This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request

 You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

If you click the "Details" it will take you to the Jenkins build log. You can write custom check in your Jenkinsfile that can be used for the build reviews.

Now, if you check Jenkins you will find a pipeline for feature branch in Jenkins as shown below.



multi-branch-demo

Jenkins Multibranch Pipeline Example Repo

Branches (1)		Pull Requests (1)		
S	W	Name ↓	Last Success	Last Failure
		feature	12 sec - #3	6 min 30 sec - #2

If the build fails, you can commit the changes to the feature branch and as long as the PR is open, it will trigger the feature pipeline.

In the `Jenkinsfile` I have added a condition to skip the deploy stage if the branch is not develop. You can check that in the Jenkins build log. Also, If you check the build flow in the blue ocean dashboard you can clearly see the skipped deployment stage as shown below.



Now merge the feature branch PR and raise a new PR from develop to the master branch.

Jenkins will receive the webhook from Github for the new PR and the develop pipeline gets created as shown below.

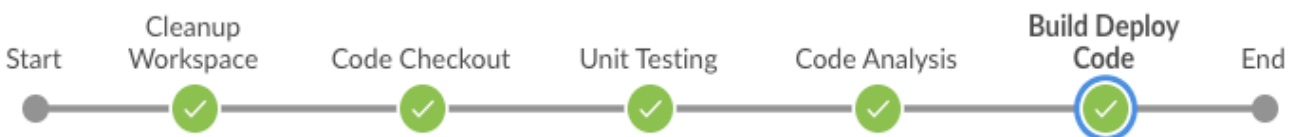


multi-branch-demo

Jenkins Multibranch Pipeline Example Repo

Branches (2)		Pull Requests (1)		
S	W	Name ↓	Last Success	Last Failure
		develop	26 sec - #1	N/A
		feature	22 min - #3	28 min - #2

For develop branch, the deploy stage is enabled and if you check the Blue Ocean build flow you can see all the stages successfully triggered.



Troubleshooting Multibranch Pipelines

I will talk about a few possible errors in a multibranch pipeline that you might encounter and how to troubleshoot them.

Branch Discovery Issue

Sometimes even after creating new branches in the SCM, it might not reflect in the Jenkins pipeline. You can try running the "Scan Repository Now" option to scan the repo again. Also, check the repository scan configurations in the pipeline.

PR Webhooks Not Triggering the Pipelines

When a webhook is not triggering the pipeline, check the webhook delivery in Github for status code and error. Also, check if the Jenkins URL is correct. Also, check Jenkins logs from Manage Jenkins --> System Logs --> All Jenkins logs. If Jenkins is able to receive the webhook, the log should show the reason why the jobs are not getting triggered.

Commits Not Triggering Pipeline

If you want each commit to trigger the branch pipeline, then you should select the "**Discover All Branches**" option in the branch discovery configuration. So whenever you commit a change to the discoverable branches or raise a PR, the pipeline will automatically get triggered.

Multibranch Pipeline Best Practices

Let's have a look at some of the best practices for a multibranch pipeline.

Repo Branching – Have a Standard Structure

It is essential to have the standard branching structure for your repositories. Whether it is your application code or infra code, having a standard branching will reduce the inconsistent configurations across different pipelines.

Shared Libraries – Reusable Pipeline Code

Make use of **shared libraries** for all your multi-branch pipelines. Reusable libraries make it easy to manage all the pipeline stages in a single place.

Pull Request Vs Commit Triggers

Try to use a PR based pipeline rather than commit based. If a code repo gets continuous commits it might overwhelm Jenkins with many builds.

Commit based triggers are supported in PR based discovery as well. Here the commit trigger happens only when the PR is still open.

Jenkins Pipeline Vs. Multibranch Pipeline

A normal pipeline job is meant for building a single branch from the SCM and deploy to a single environment. However, you can

A multibranch pipeline is meant for building multiple branches from a repository and deploy to multiple environments if required.

A pipeline job supports both pipeline steps to be added in Jenkins configuration and from SCM.

Use pipeline job for adhoc jobs, parameterised job executions and to debug pipeline as code.

Do not use multibranch pipeline if you do not have a standard branching and CI/CD strategy.

Multibranch Pipeline Vs. Github Organization Job

Like a multi-branch pipeline, the Github organization folder is one of the Jenkins project types.

Note: The organization folder is not just limited to Github. It can be used for [Gitlab](#), [Bitbucket teams](#), or [Gitea](#) organization.

Multibranch pipeline can only configure pipelines for a single Git repository. Whereas a Jenkins Github organization project can automatically configure multi-branch pipelines for all the repos in a Github organization.

It can discover all the repositories in the configured Github organization, with a Jenkinsfile.

Also, you can configure a generic webhook in the organizational level to avoid having webhooks in each repo.

The only difference between multi-branch and organization project is that organizations can configure multi-branch pipelines for multiple repos.

So which one should I use?

This totally depends on the workflow you need. If you have a standard pipeline and process of deploying applications or infra code, Github organization is great. Or else, configuring multibranch pipeline separately will be a good option.

Parallael pipeline

```
pipeline {
    agent any

    stages {
        stage('One') {
            steps {
                echo 'Hi, this is Zulaikha from edureka'
            }
        }
    }
}
```

```
    stage('Two') {  
        steps {  
            input('Do you want to proceed?')  
        }  
    }  
  
    stage('Three') {  
        when {  
            not {  
                branch "master"  
            }  
        }  
        steps {  
            echo "Hello"  
        }  
    }  
  
    stage('Four') {  
        parallel {  
            stage('Unit Test') {  
                steps {  
                    echo "Running the unit  
test..."  
                }  
            }  
        }  
        stage('Integration test') {  
            agent {  
                docker {  
                    reuseNode true  
                    image 'ubuntu'  
                }  
            }  
        }  
    }  
}
```

```

    }

    }

    steps {
        echo "Running the integration
        test..."

    }

    }

    }

    }

}

}

```

- Stage four runs a parallel directive. This directive allows you to run nested stages in parallel. Here, I'm running two nested stages in parallel, namely, 'Unit test' and 'Integration test'. Within the integration test stage, I'm defining a stage specific docker agent.
-
- This docker agent will execute the 'Integration test' stage.
- Within the stage are two commands.
-
- The **reuseNode** is a Boolean and on returning true, the docker container would run on the agent specified at the top-level of the pipeline,
- in this case the agent specified at the top-level is 'any' which means that the container would be executed on any available node.
- By default this Boolean returns false.

- There are some restrictions while using the parallel directive:
 - A stage can either have a parallel or steps block, **but not both**
 - Within a parallel directive you cannot nest another parallel directive
 - If a stage has a parallel directive then you cannot define 'agent' or 'tool' directives

How can I run a single job parallelly on multiple Jenkins slave?

There are two ways to accomplish this -

1. Using **Node Label Parameter plug-in**

2. Using **Multi- Configuration project + Axis plug-in**

1. Node Label Parameter plug-in : Install the plug-in, under '**this build is parameterized option**', you will find an option for specifying a list of **Nodes as a drop down**, and then to execute them serially or execute them on **multiple nodes parallelly**, when you select Build option.

NodeLabel Parameter Plugin

2. Multi-Configuration and Axis Plug-in : Install both these plug-ins. **Create Multi-configuration Jenkins job** and then you can

create a virtual matrix where the job can run the job on all the nodes identified by a common label. (You need to specify the common label name in all the nodes created (Manage nodes --> Node name --> Configure))

[Elastic Axis - Jenkins - Jenkins Wiki](#)

Install Node parameter plugin in jenkins,

You will get option "Node" under "This build is parameterized", Under Node option Select following.

"Allow multi node selection for concurrent builds."

Now Select following option too

"Execute concurrent builds if necessary".

That's about configuration, If you try to run the build you will get following option.

Run Single Job Parallely in Multiple slaves

There are two ways to accomplish this

1. Using Node Label Parameter plug-in

2. Using Multi- Configuration project + Axis plug-in

3. **Node Label Parameter plug-in** : Install the plug-in, under 'this build is parameterized option', you will find an option for specifying a list of Nodes as a drop down, and then to execute

them serially or execute them on multiple nodes parallelly, when you select Build option.

4. Multi-Configuration and Axis Plug-in : Install both these plug-ins. Create Multi-configuration Jenkins job and then you can create a virtual matrix where the job can run the job on all the nodes identified by a common label. (You need to specify the common label name in all the nodes created (Manage nodes --> Node name --> Configure))

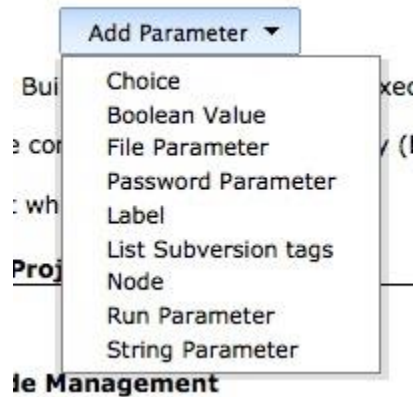
Restrict where this project can be run

If your using a node or label parameter to run your job on a particular node, you should not use the option "Restrict where this project can be run" in the job configuration - it will not have any effect to the selection of your node anymore!

Pre-Requisite Step

Below is the process to run a single job using Node Label Parameter plug-in

1. Add the Node parameter in Job Configuration.



1. Define a list of nodes on which the job should be allowed to be executed on. A default node used for scheduled jobs can be defined. You are able to configure the job to run one after the other or even concurrent.

Delete Project

 Configure

Build History (trend)

#70 (pending - dumb4 is offline)

#69 (pending - dumb2 is offline)

#68 [Sep 18, 2011 5:49:18 PM](#)

#67 [Sep 18, 2011 5:49:17 PM](#)

#66 [Sep 18, 2011 5:49:16 PM](#)

#65 [Sep 18, 2011 5:48:53 PM](#)

#64 [Sep 18, 2011 5:48:52 PM](#)

#63 [Sep 18, 2011 5:48:51 PM](#)

#62 [Sep 18, 2011 5:48:36 PM](#)

#61 [Sep 18, 2011 5:48:35 PM](#)

#60 [Sep 18, 2011 5:48:34 PM](#)

#59 [Sep 18, 2011 5:45:21 PM](#)

RSS for all
 RSS for failures

Logger Console

Expand

☒ Discard Old Builds

 Days to keep builds

if not empty, build records are only kept up to this number of days

 Max # of builds to keep

if not empty, only up to this number of build records are kept

☒ This build is parameterized

Node

Name

Select a default slave

Define possible nodes

ALL (no restriction)

dumb1

dumb2

dumb3

dumb4

☐ Run next build only if build succeeds
 ☐ Run next build only if build suc

☒ Allow multi node selection for concurrent builds
 ☐ Disallow multi node selection

In case of multi node selection, should the next build on the next node be triggered only for succe

Description

Add Parameter ▾

☐ Disable Build (No new builds will be executed until the project is re-enabled.)

☒ Execute concurrent builds if necessary (beta)

☐ Restrict where this project can be run

Advanced Project Options

Save the **Job Configuration and Build the Job according to your requirement.**

42

Step 1– If multi node selection was enabled, you get the chance to select multiple nodes to run the job on. The job will then be executed on each of the nodes, one after the other or concurrent - depending on the configuration.

Jenkins » node

Back to Dashboard

Status

Changes

Workspace

Build Now

Delete Project

Configure

Project node

This build requires parameters:

where to run?

- dumb1
- dumb2
- dumb3
- dumb4
- dumb5
- master

restrict which nodes the job should be executable on

Build

Build History		(trend)
#194	Sep 11, 2011 4:44:34 PM	
#193	Sep 11, 2011 4:44:28 PM	
#192	Sep 11, 2011 4:44:10 PM	

Step 2– Click on the **Build History** link to see the details of the builds running a Single Job simultaneously in Multiple slaves based on the above selection of nodes to run on.

Step 3– Once the build is completed, a status of the build will show if the build was successful or not. In our case, the following build has been executed successfully. Click on the #195 and #196 in the Build history to bring up the details of the build.

How to reset build number in jenkins?

A: Can be easier done from groovy script console . Go to <http://your-jenkins-server/script> In script window enter:

```
item = Jenkins.instance.getItemByFullName("your-job-name-here")
```

//THIS WILL REMOVE ALL BUILD HISTORY

```
item.builds.each() { build ->
```

```
build.delete()
```

```
}
```

```
item.updateNextBuildNumber(1)
```

OR

```
echo "deleting all builds of ${jobName}" item =  
Jenkins.instance.getItemByFullName("${jobName}") for (build in  
item.builds){ build.delete(); } item.updateNextBuildNumber(1)
```

In Jenkins, is it possible to run multiple jobs at the same time on the same build machine? If not, why not?

Go to your job -> configuration and check: **Execute concurrent builds if necessary.**

Doc:

If this option is checked, Jenkins will schedule and execute multiple builds concurrently (provided that you have

sufficient executors and incoming build requests.) This is useful on builds and test jobs that take a long time, as each build will only contain a smaller number of changes, and the total turn-around time decreases due to the shorter time a build request spends waiting for the previous build to complete. It is also very useful with parameterized builds, whose individual executions are independent from each other.

For other kinds of jobs, allowing concurrent executions of multiple builds may be problematic, for example if it assumes a monopoly on a certain resource, like database, or for jobs where you use Jenkins as a cron replacement.

Introduction

[Jenkins World 2017](#) came to a close in late September. Time to revisit the improvements that have been made to the support for build pipelines. I am no stranger to using Jenkins to model a Continuous Delivery pipeline. In the dark ages, you had to construct a pipeline with the help of different Jenkins plugins bit by bit. The approach was highly brittle, inconsistent and full of magical tips and tricks. You can find a discussion of the approach in my book [Gradle in Action](#).

In this blog post, I am going to discuss how to construct and operate a [declarative pipeline with Jenkins](#). The vehicle for the pipeline is a Java-based web application. To emulate a real-world

scenario, every change made to the project travels through the full pipeline and is deployed to Heroku on demand. You can find the [full source code on GitHub](#).

The sample application

The functionality of the sample application is straightforward. It allows the user to record to-do items that can be completed by checking them off the list. A database stores the to-do items and their states. Figure 1 shows a screenshot of the deployed application. You can try out the [deployed application](#) for yourself on Heroku.

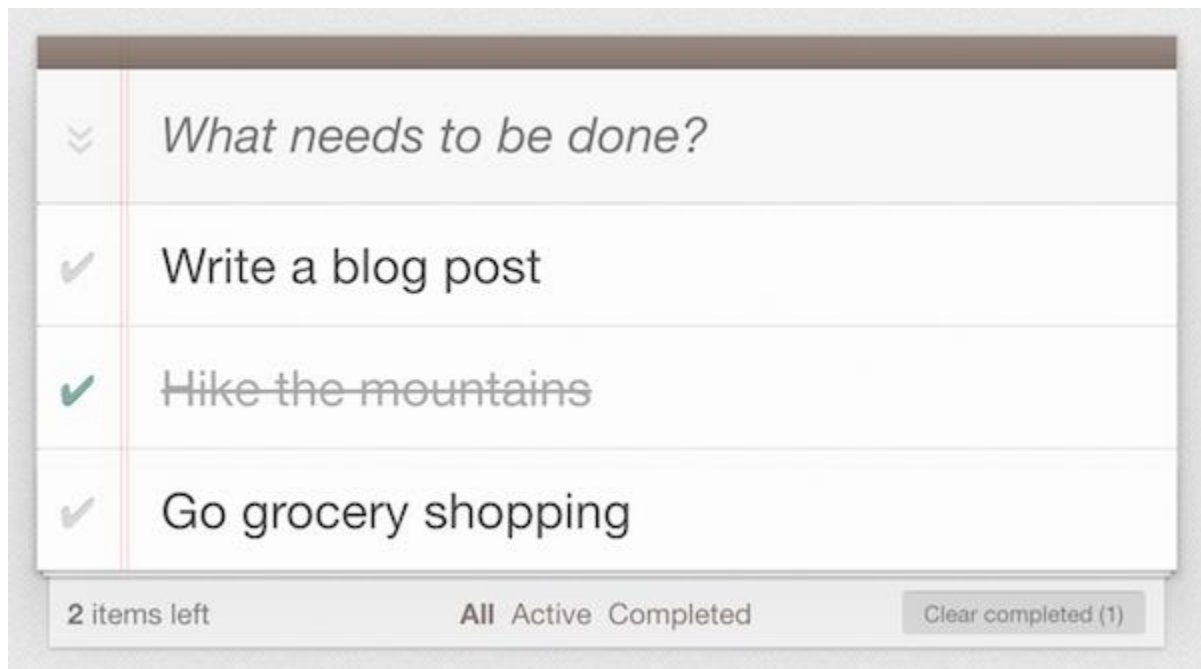


Figure 1. A web-based application for managing to-do items

On the technical side, the web application was built with Java and Spring Boot. Automated tests use JUnit and [Spring Boot's test utilities](#) for integration testing of Spring components. The actual

work performed by the build pipeline is based on Gradle build logic.

The main focus of this post is going to be on modeling pipeline capabilities of Jenkins. I'd encourage you to have a look at the Gradle build scripts available in the source code repository to learn more about the build logic. Before diving into Jenkins right away, let's first have a look at the different stages of build pipeline you are about to model.

Modeling the build pipeline

A build pipeline, also called deployment pipeline, represents the delivery process of a software broken down into individual stages. A change to the code of that software travels through the stages of the pipeline with the goal of producing one or many deliverable artifacts that can be shipped to the customer. The book [Continuous Delivery](#) gives a great introduction to the topic of modeling build pipeline.

The build pipeline for the To Do web application should consist of the following phases:

1. **Compile source code:** Executes the Java compiler to turn production and test source code into byte code.
2. **Execute unit tests:** Runs the unit test suite for fast feedback.
3. **Execute integration tests:** Performs long-running integration tests.

4. **Perform static code analysis:** Analyzes the source code and identifies quality hotspots. For convenience reasons, the project uses the [cloud-based service of Sonarqube](#).
5. **Assemble the WAR file:** Builds the final deliverable, a WAR file containing all resources needed in the runtime environment.
6. **Deploy the deliverable to production:** Pushes the WAR file to the Paas, [Heroku](#), upon request.

Some of the phases should be executable in parallel especially the ones that might take a long time to finish e.g. integration testing (3) and static code analysis (4). While Continuous Deployment might work perfectly for this kind of application, the pipeline should also implement a manual step that only triggers deployment to production (6) if a user pushes a button.

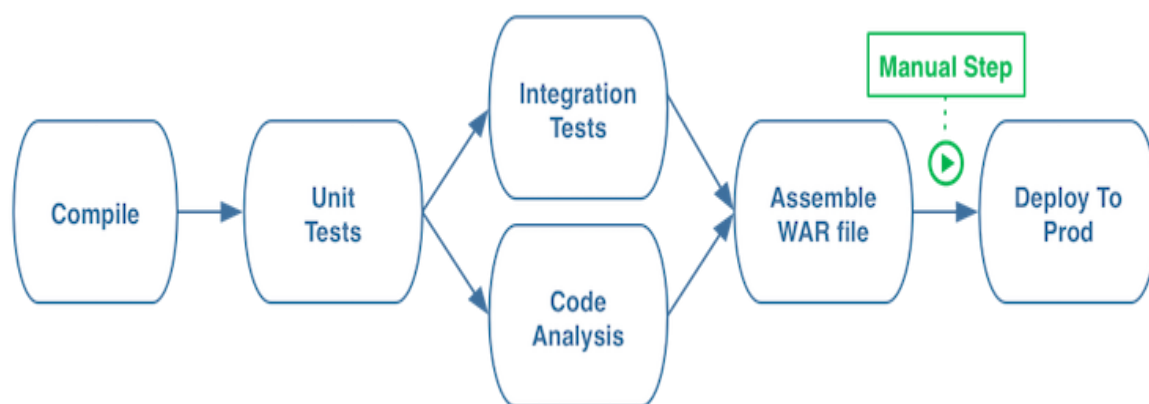


Figure 2. The build pipeline in theory

The next section talks about the necessary prerequisites for implementing the sample pipeline in Jenkins.

Setting up Jenkins

You can literally get up and running with Jenkins in a matter of minutes. Just follow the steps below if you want to set up Jenkins on your local machine and install all relevant plugins.

Installing Jenkins

First, you'll need to install Jenkins on your developer machine or on a server. Download the desired distribution from the ["Getting started with Jenkins" page](#). For the purpose of this blog post, you'll download the WAR file which is runnable on all operating systems.

Navigate to the directory containing the WAR file and execute the command `java -jar jenkins.war` from your console. Bring up Jenkins in the web browser of your choice via the url <http://localhost:8080/>. Log in with the temporary administrator password and simply go with the option "Install suggested plugins". The [pipeline plugin](#), responsible for defining and rendering build pipelines in Jenkins, is a standard plugin and will be installed automatically. In the next screen, set up a new admin user.

In addition to the visualization provided by the standard pipeline plugin, I prefer to use the view introduced by [Blue Ocean plugin](#). Blue Ocean aims to offer a fresh and overhauled UI. Later, you'll

explore the standard pipeline view as well as the visualization provided by Blue Ocean.

Defining credentials

As part of our pipeline, you'll use SonarCloud for static code analysis and Heroku for hosting the application. Both services have free tiers. Sign up for an account if you'd like to take advantage of these services. In this section, you'll configure the credentials for SonarCloud and Heroku in Jenkins. As a result, the pipeline will have access to environment variables that can be passed down to the Gradle build.

SonarCloud and Heroku provide an access token that can be retrieved from the setting page of the corresponding service after logging in. In Jenkins you can declare those access tokens as "Secret text" under "Jenkins > Credentials". For the purpose of this sample pipeline, we'll create the credentials with ID `SONARCLOUD_TOKEN` and `HEROKU_API_KEY`. Figure 3 shows the definition of the SonarCloud credentials.



The image shows the 'Add New Credential' form in Jenkins. The 'Kind' dropdown is set to 'Secret text'. The 'Scope' dropdown is set to 'Global (Jenkins, nodes, items, all child items, etc)'. The 'Secret' field contains a masked token (dots). The 'ID' field is set to 'SONARCLOUD_TOKEN'. The 'Description' field is set to 'The SonarCloud access token.' There is an 'OK' button at the bottom left.

Figure 3. Adding SonarCloud credentials

Next, we'll dive into the creation of the job that defines the build pipeline.

Pipeline definition in Jenkins

Creating a Jenkins pipeline is not fastly different from creating a freestyle project. Select the "Pipeline" option in the job creation screen to get started. We'll use the job name `todo-spring-boot` for now to reflect the project name on GitHub.

For pipeline definitions we have two options:

1. Defining a pipeline script in the Jenkins job that can be edited on-the-fly.
2. Referring to a pipeline script from SCM.

In the light of [Infrastructure as code](#), we'll go with option 2. The Github repository already [contains the file](#) defining the desired layout and configuration of the pipeline for our job. By default, this file is called `Jenkinsfile`. Listing 1 shows pipeline definition in its full beauty.

Jenkinsfile

```
pipeline {  
    agent any  
  
    triggers {  
        pollSCM('*/*5 * * * *')
```

```

    }

    stages {
        stage('Compile') {
            steps {
                gradlew('clean', 'classes')
            }
        }

        stage('Unit Tests') {
            steps {
                gradlew('test')
            }

            post {
                always {
                    junit '**/build/test-results/test/TEST-*.xml'
                }
            }
        }

        stage('Long-running Verification') {
            environment {
                SONAR_LOGIN = credentials('SONARCLOUD_TOKEN')
            }
        }
    }
}

```

```

    }

    parallel {
        stage('Integration Tests') {
            steps {
                gradlew('integrationTest')
            }
            post {
                always {
                    junit '**/build/test-
results/integrationTest/TEST-*.xml'
                }
            }
        }

        stage('Code Analysis') {
            steps {
                gradlew('sonarqube')
            }
        }
    }

    stage('Assemble') {

```

```

    steps {
        gradlew('assemble')
        stash includes: '**/build/libs/*.war', name: 'app'
    }
}

stage('Promotion') {
    steps {
        timeout(time: 1, unit:'DAYS') {
            input 'Deploy to Production?'
        }
    }
}

stage('Deploy to Production') {
    environment {
        HEROKU_API_KEY = credentials('HEROKU_API_KEY')
    }
    steps {
        unstash 'app'
        gradlew('deployHeroku')
    }
}

```

```

    }

    post {
        failure {
            mail to: 'benjamin.muschko@gmail.com', subject: 'Build
failed', body: 'Please fix!'
        }
    }
}

def gradlew(String... args) {
    sh "./gradlew ${args.join(' ')} -s"
}

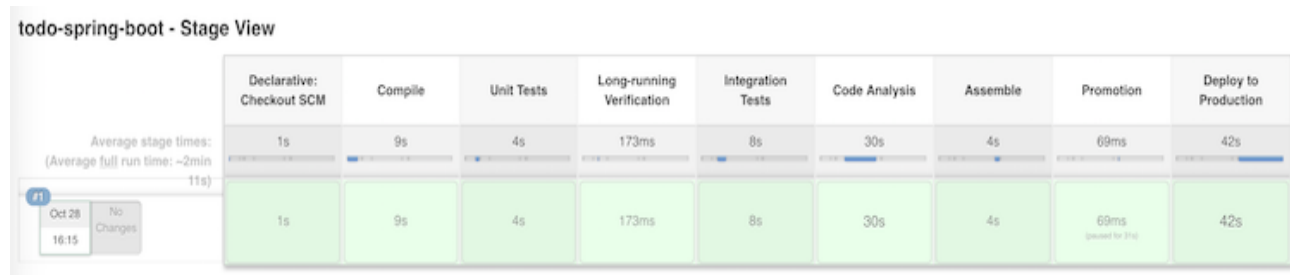
```

Listing 1. The pipeline definition as code

As you can see in the listing, the code is very readable and understandable even if you do not know the syntax in more detail. The script defines each stage of the pipeline in order. Each stage may contain one or many steps. The actual "work" of the pipeline happens under the covers and is performed by Gradle. Jenkins merely acts as orchestration engine.

Let's see what happens when we execute the job by triggering the "Build Now" action. Jenkins runs through all stages of the pipeline up until the point that requires manual execution. In the

console view, you can confirm or abort the manual execution to trigger the deployment to production. Below you can find the standard view of the pipeline in the job.



The screenshot shows the Jenkins 'Stage View' for a job named 'todo-spring-boot'. It displays a table of stages with their names, average times, and progress bars. A sidebar on the left shows a build history with a recent build on Oct 28 at 16:15, marked as 'No Changes'.

Declarative: Checkout SCM	Compile	Unit Tests	Long-running Verification	Integration Tests	Code Analysis	Assemble	Promotion	Deploy to Production
1s	9s	4s	173ms	8s	30s	4s	69ms	42s

Figure 4. Jenkins stage view

You might have noticed that the standard job view of the pipeline does not render parallel stages or manual triggers. Blue Ocean makes a much better attempt at rendering those features. In the next section, we'll have a brief look at what Blue Ocean pipeline views have to offer.

A look at the Blue Ocean pipeline view

In an earlier section, you installed the Blue Ocean plugin. On the left hand side, of your job screen you should see a "Open Blue Ocean" option. Executing the job shows a much richer representation of the pipeline. You can literally see how your change travels through each stage of the pipeline in real-time. The following screenshot shows the waiting state of the pipeline expecting user input on the promotion stage. The UI even provides an option to proceed with the next stage or abort the flow of the pipeline.

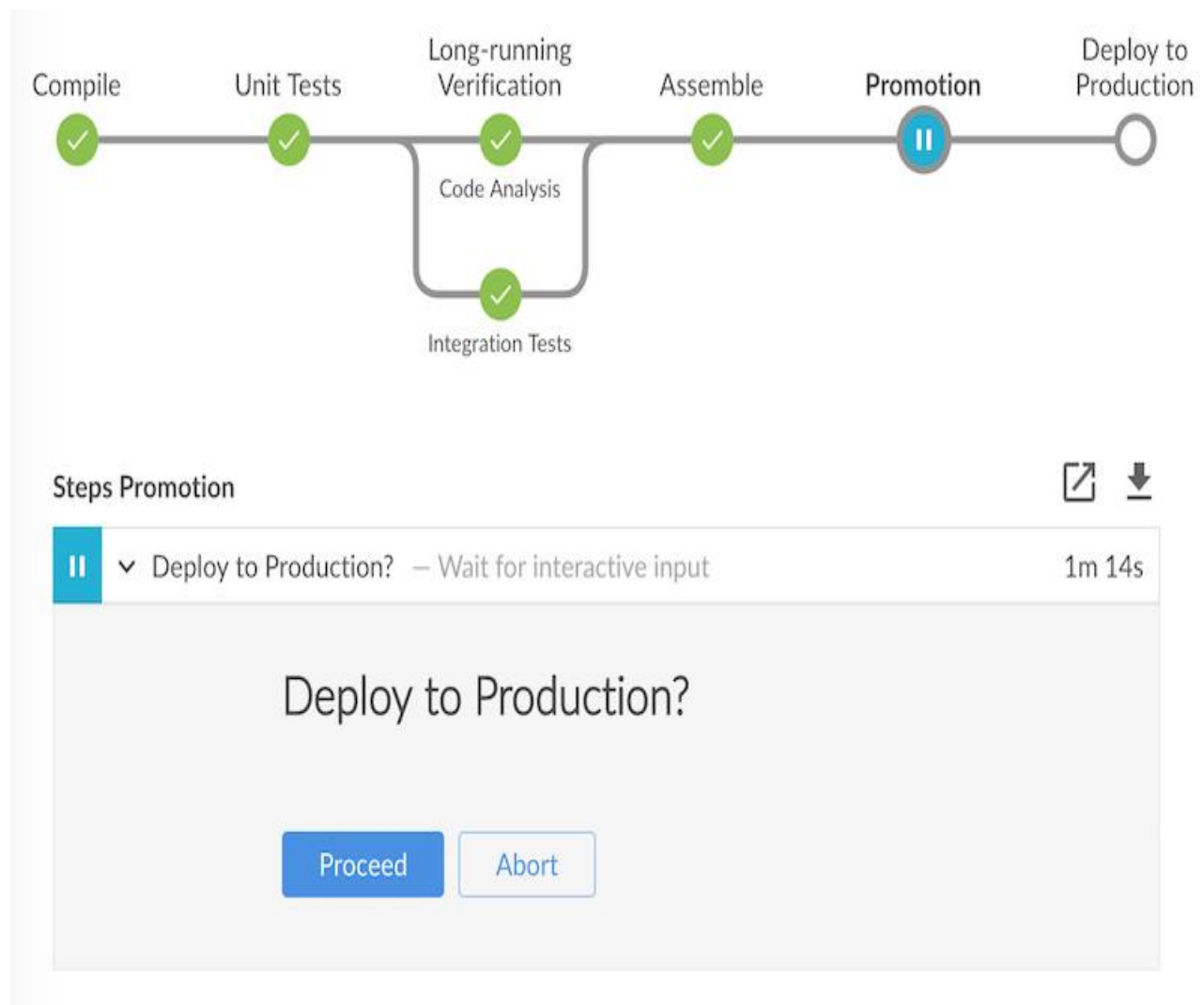


Figure 5. Blue Ocean view of the pipeline waiting for user input

In the pipeline script above, you configured to wait for interactive input during the promotion step for one day. Jenkins automatically times out the manual operation after the defined waiting period. If you don't want to wait this long, then you can always abort the operation manually.

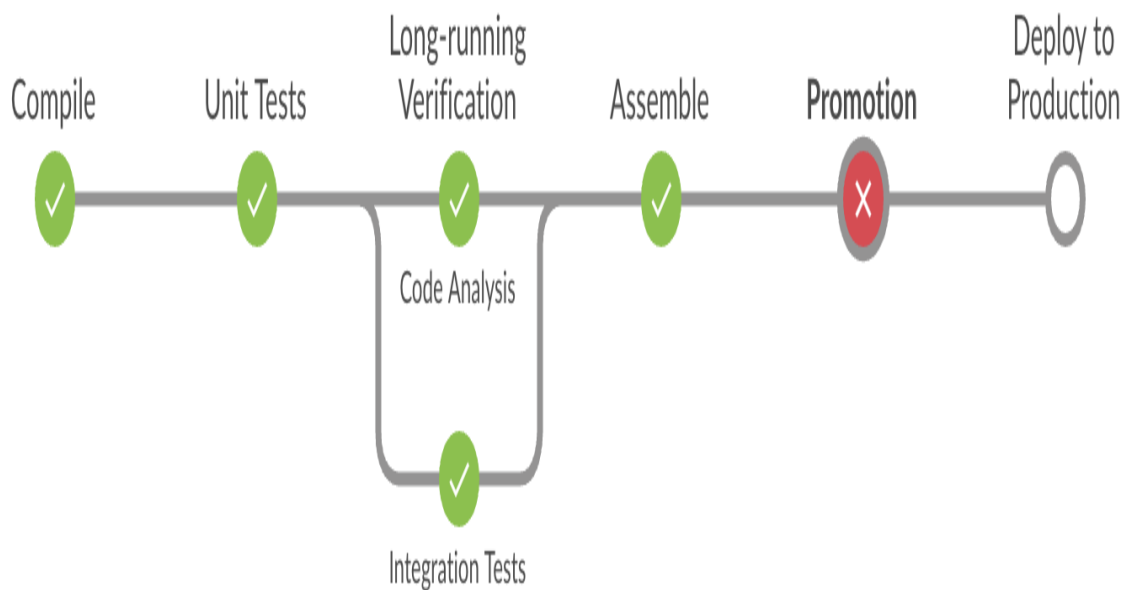


Figure 6. Aborting a manual step in Blue Ocean pipeline view

If you happen to actually want to deploy the WAR file to production, then all you need to do is to press the "Proceed" button. The pipeline will continue its operation immediately and move on with the next stage in line.

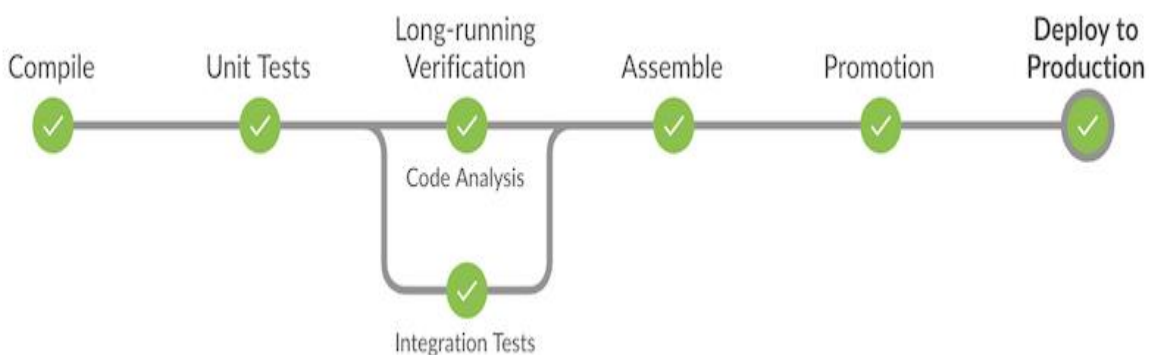


Figure 7. Triggering the deployment via manual step in Blue Ocean

