Getting Started Guide: Using the LabVIEW Command Line Interface to connect LabVIEW and Jenkins

Table of Contents

Introduction

Continuous Integration and Continuous Delivery

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. By integrating regularly, errors are detected and located quickly.[1] Continuous Delivery is the natural extension of Continuous Integration, an approach in which teams ensure that every change to the system is releasable, and release any version with the push of a button.[2]

Continuous Integration is not a new concept; it is however somewhat new within the LabVIEW ecosystem. There have been several CI approaches over the last couple years, and the Command Line Interface extension for LabVIEW is an attempt to merge the best components we have seen into a single extensible framework.

Continuous Integration Workflow

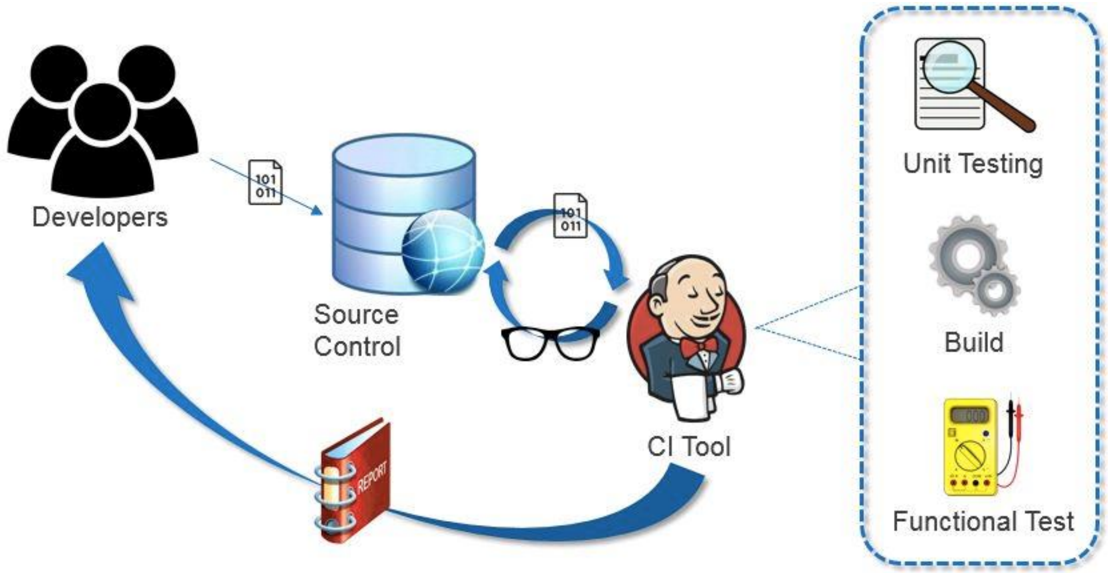


Figure 1: Continuous Integration Workflow

As highlighted in the figure above, the Continuous Integration workflow begins with the developers. After modifying a branch, the developers commit their work back to the source code control repository. Source Code Control is a necessity for Continuous Integration.

[1] https://www.thoughtworks.com/continuous-integration
[2] https://www.thoughtworks.com/continuous-delivery

A CI server that can automate the build is watching to the source code repository and can initiate a test when the code is committed. The test can include stages to unit test each VI in the repository, perform function testing and to build the latest version of the code. The server will then send a final report with the results of the test back to the developers.

Jenkins serves as the CI server. It is open source, and it is the most widely used CI tool. It is installed as a service on a server host and is interfaced with through a Web GUI. For these reasons, it has an extensive plugin collection that can be used to customize the build system. Jenkins can easily communicate with any source code management software with the appropriate plugin. The Server is also configuration-based, meaning that you can configure Jenkins to send your email a notification with the latest build results. One of the easiest ways to have Jenkins execute a build is through batch scripts.

LabVIEW Command Line Interface

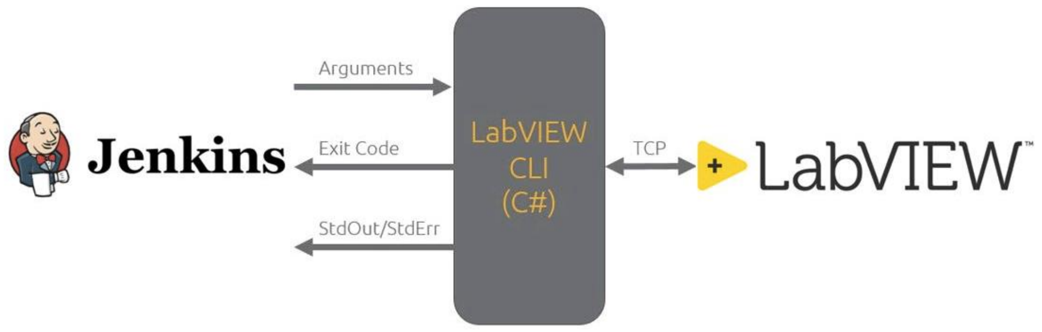


Figure 2: Missing Link Between LabVIEW and Jenkins

Historically, LabVIEW can only receive inputs from a batch file or the command prompt in the form of a string. It has not been able to pass out an Exit Code, StdOut or StdErr, which is how Jenkins knows whether a build has succeeded or failed. With the LabVIEW CLI extension, developed by James McNally of Wiresmith Technology, LabVIEW can now communicate back with the command prompt with an Exit Code and StdErr. For more information on the tool, see https://devs.wiresmithtech.com/blog/bringing-command-line-interface-labview/.

With this tool, LabVIEW and Jenkins can communicate with each other and provide valuable information to the user about the build test and results. However, to execute a build, LabVIEW must know what tests the developers need done. Functions have already been developed that wrap many tests that the developers may need including:

- Invoke Build Specifications
- Build VI Packages
- Run VI Analyzer

- Run UTF test in a project with JUnit Results

For example, the wrapper VI that invokes build specifications calls the LabVIEW Application Builder API. The only parameters needed from Jenkins are the target, the project, the build specification's name and the LabVIEW version.

From the Jenkins side, the scripts that call these functions have already been written, as well. The proper notations have been already been implemented and are in use for LabVIEW DCAF's build system. These scripts are all open-source and are available on the Tools Network as a VI Package knowns as LV-CLI Common Steps. The source is hosted on github.com/LabVIEW-DCAF/buildsystem. Feel free to collaborate, improve and implement this in your own system. How to implement these are explained in later sections.

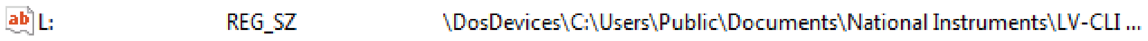
Installing and Implementing the LabVIEW CLI

The first part of this how-to will cover the LabVIEW part and the tools required in Jenkins. These are steps that are necessary for each build system. The second part goes into the Jenkins implementation. This is just one approach to setting up a build server. There are many options, and the way that the is highlighted here is one of many options.

Installing the LabVIEW CLI

1. Navigate to https://github.com/JamesMc86/LabVIEW-CLI/releases and install LabVIEW_CLI.msi.
    a. This installs the proxy mechanism that allows LabVIEW programs to easily write out to the command line.
2. In VI Package Manager, find and **LV-CLI Common Steps**.
    a. The package install any dependencies that you may be missing for the package, such as JUnit reporting for UTF tests. All dependencies are required for proper execution.
3. The package installs the VI's and Jenkins scripts to an OS public documents folder (`C:\Users\Public\Documents\National Instruments\LV-CLI Common Steps`) to allow scripts that use many LabVIEW versions to work properly. The scripts assume that you have mapped this location to the L:\ drive. Use the following steps to do that:
    a. Open regedit.exe.
    b. Navigate to `KEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ DOS Devices`.
    c. Add a new REG_SZ value by tight-clicking to adding a string value. Name it L:

d. Right click the new REG_SZ and select **Modify…**
e. In Value data, insert \DosDevices\C:\Users\Public\Documents\National Instruments\LV-CLI Common Steps\steps.
f. Click **OK.** You should see a registry value, seen in the following figure, added to the list.



```
L:        REG_SZ        \DosDevices\C:\Users\Public\Documents\National Instruments\LV-CLI ...
```

g. Restart your computer. Open up a file explorer, and ensure that your L: drive maps properly.

Note: If you do not want to map the L: drive to the public location, please go into each script and modify every location where you see L:\\ with "C:\\Users\\Public\\Documents\\National Instruments\\LV-CLI Common Steps\\steps." Jenkins scripts require two slashes, \\.

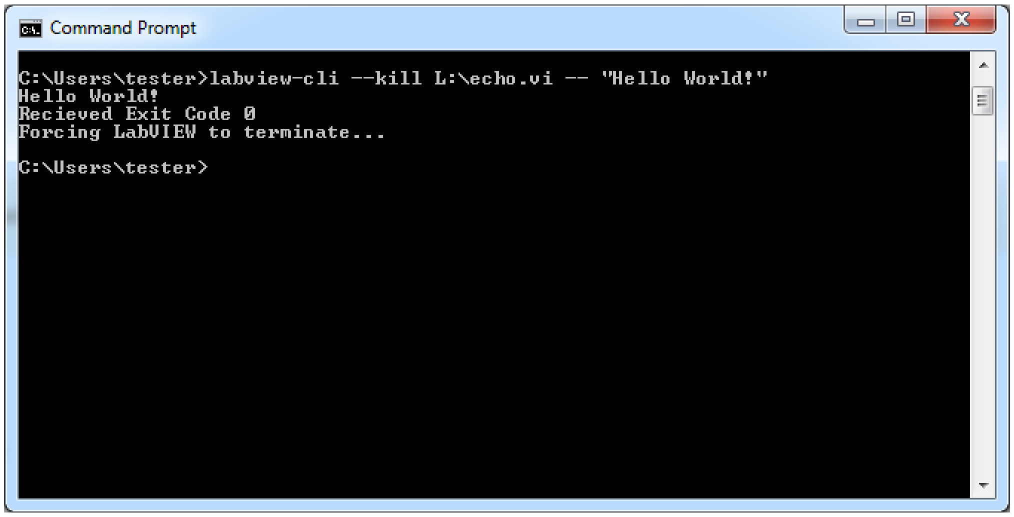
Echo Test

To test whether this installation and the mapping was successful, a test can be done with an echo VI that ships with the package.

1. Open a command prompt.
2. To invoke the LabVIEW CLI, each command will begin with "labview-cli." You can enter `labview-cli -h` to see all the commands or go to https://github.com/JamesMc86/LabVIEW-CLI for the entire list.
3. In command prompt enter `Labview-cli L:\echo.vi -kill -- "Hello World!"` If you want to run this code in a specific LabVIEW version, you can add "lv-ver" followed by the four digits that represent the specific version.

   The "--kill" command will close out LabVIEW when the echo.vi has run. Without the kill command, command prompt will not proceed to the next command and wait for LabVIEW to exit.

   Note: LabVIEW can only receive arguments upon launch. If LabVIEW is open, and you attempt to call the echo.vi, the echo.vi will open and run and output error 1. The command prompt will display "LabVIEW terminated unexpectedly!"
4. Once you have entered the command to the echo.vi, you should see something similar to the following output:



Command Prompt

```
C:\Users\tester>labview-cli --kill L:\echo.vi -- "Hello World!"
Hello World!
Recieved Exit Code 0
Forcing LabVIEW to terminate...

C:\Users\tester>
```

If you see this screen, you have successfully installed the LabVIEW CLI and the LV-CLI common steps. You can do more experimentation with the LV-CLI. If you want to test out a simple build, you can take a look at an example script (a groovy text file used by Jenkins) in the vars folder of the LV-CLI directory to see what inputs are necessary for each LV-CLI function.

Jenkins Setup

Before Jenkins is used to call the echo function and test out the CLI functionality, these steps walk through the Jenkins Configuration necessary to implement that functionality.

Jenkins Plugins

After installing Jenkins, you will be prompted to install plugins to extend Jenkins functionality. Similarly, if you navigate to **Manage Jenkins>>Manage Plugins** and go to the **Available** tab, you will be able to install additional plugins. For the repository in this tutorial, the following plugins were used:

1. SSH Slaves Plugin
2. Run Condition Extras Plugin
3. Windows Slaves Plugin*
4. Workspace Cleanup Plugin
5. Plugin for Source Code Control. For Github:
    a. Git plugin
    b. GitHub Organization
6. Junit Plugin
7. Build Pipeline Plugin
8. Pipeline, and including:
    a. Pipeline: Basic Steps

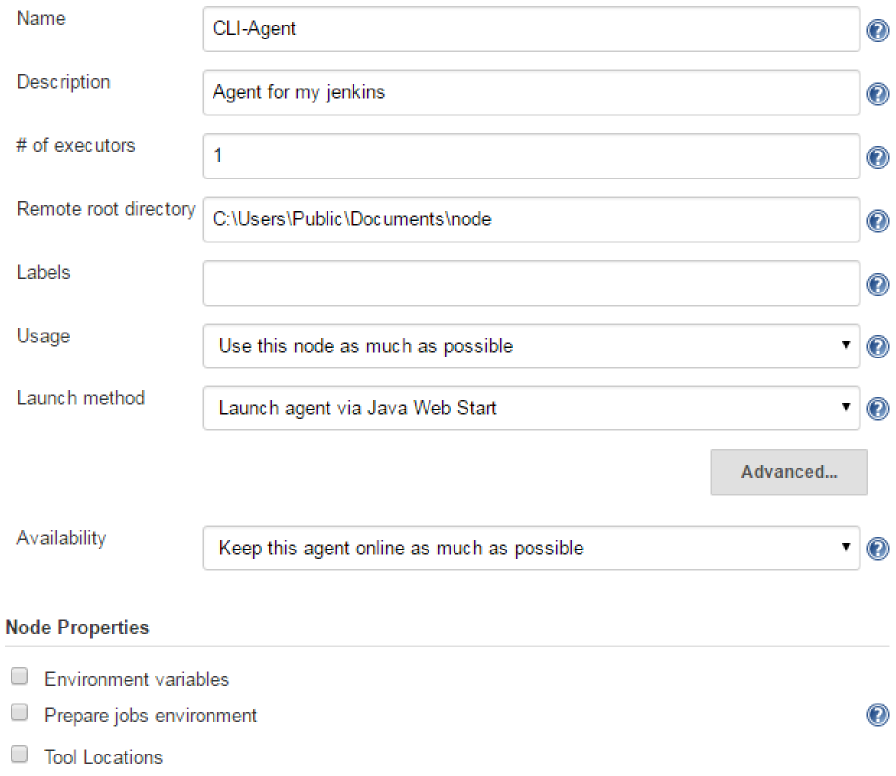
   b. Pipeline: Groovy
   c. Pipeline: Job
   d. Pipeline: SCM Step
   e. Pipeline: Shared Groovy Library
   f. Pipeline Stage View Plugin
9. Groovy

   *Note: The Windows Slaves Plugin depends on one or more of these other plugins. You will not be able to find or select this plugin until you have installed the others, and restarted Jenkins.

Jenkins Node

Jenkins, by default, only has a master node, the computer that you installed Jenkins on. The master node calls the Jenkins processes and programs required to run the builds in the background. Therefore, LabVIEW will not open interactively, but will open in the background to perform tests, with no user interface visible. For debugging purposes, it is recommended that a Jenkins node is set up either on the master computer or any other computer on the network so that LabVIEW development environment is visible during a build. This makes it much easier to debug your build errors. To do this:

1. Navigate to **Manage Jenkins>>Manage Nodes.**
2. Click **New Node.**
3. Name the node. Select Permanent Agent and click **OK.**
4. Configure your node, similar to the following configuration:



Name	CLI-Agent
Description	Agent for my jenkins
# of executors	1
Remote root directory	C:\Users\Public\Documents\node
Labels	
Usage	Use this node as much as possible
Launch method	Launch agent via Java Web Start
Availability	Keep this agent online as much as possible

Node Properties

- [ ] Environment variables
- [ ] Prepare jobs environment
- [ ] Tool Locations

The node can be on the master computer or any other computer as mentioned above. The remote root directory will be local on the slave machine. For more information on the Jenkins node set up, go here.

It is recommended that the Launch Method for your node be "Launch agent via Java Web Start." If this option is not present, use the following steps to enable it:

1. Navigate to **Manage Jenkins>>Configure Global Security.**
2. Check the **Enable Security** option and select either Random or Fixed for the TCP port for Java Network Launch Protocol (JNLP) agents.
3. Navigate back to your node configuration. There should be an option for "Launch agent via Java Web Start."

Note: Java installation is required for this option.

If you want the Jenkins nodes to exclusively do the testing, one of the ways to manually do this is to set the executor number for the Jenkins Master to 0. You can do this by:

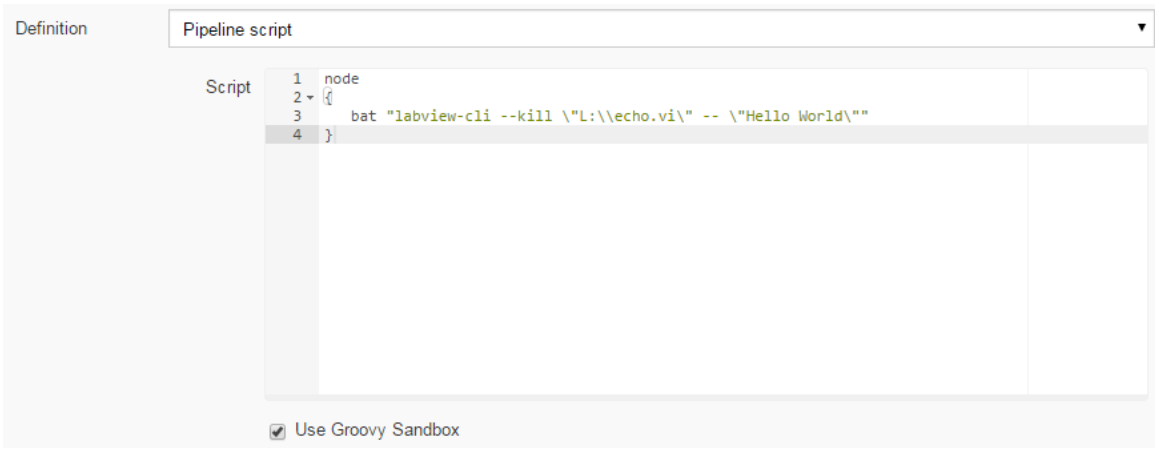
1. Navigate to Manage Jenkins>>Configure System.
2. Set the **# of executors** to 0.

Echo Test in Jenkins

If you are using a Jenkins node, ensure that the node has been launched. Go to **Manage Jenkins>>Manage Nodes** and click on the node that you will be using. Click on the **Launch** button, and open the JNLP file.

For the Echo test in Jenkins, a pipeline job will be used. Pipelines will be discussed more heavily in the following sections. To create a pipeline Job:

1. Click on **New Item.**
2. Enter a name and select Pipeline job.
3. You can configure the job, with any option. In the pipeline section, choose the option, "Pipeline script." You will enter something very similar to what you entered in the console test. Jenkins uses the 'bat', or the batch command to write to the command prompt. The command will look like the following dialogue:



```
node
{
    bat "labview-cli --kill \"L:\\echo.vi\" -- \"Hello World\""
}
```

The following is printed in the pipeline script:

```
node
{
  bat "labview-cli --kill \"L:\\echo.vi\" -- Hello"
}
```

4. Save the configuration.
5. Hit **Build Now.** Navigate to the console output to see the results of the Jenkins build. Make sure that LabVIEW is not open on the node machine before you build. Your console output should look similar to the following:

```
[Pipeline] node
Running on Test in C:\Users\Public\Documents\node\workspace\cli test
[Pipeline] {
[Pipeline] bat
[cli test] Running batch script

C:\Users\Public\Documents\node\workspace\cli test>labview-cli --kill "L:\echo.vi" -- "Hello World"
Hello World
Recieved Exit Code 0
Forcing LabVIEW to terminate...
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

6. You have successfully built your first pipeline job using the LabVIEW Command Line Interface.

Implementing a Jenkins Build System with CLI

The “Hello World” example was a simple introduction into pipelines. All the code necessary for execution was in the pipeline job configuration. We saw that the pipeline example echoed back “Hello World” to the console when the user ran the build. This method can be cumbersome because the user must manually update the configuration on the Jenkins server each time the test code would require maintenance.

Instead the test developer can opt to define a pipeline from an SCM. This pipeline definition is known as a Jenkinsfile. A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. This is the foundation of "Pipeline-as-Code"; treating the continuous delivery pipeline a part of the application to be version and reviewed like any other code.[3] Therefore, the Jenkinsfile is treated like any other piece of code.

A Jenkinsfile is contained in the root of each repository. In this tutorial’s example, it is simple and it does not contain the low-level code for a pipeline build and execution. It calls into pipeline function, which contains the stages for the job (There will be more detail provided on this in the next sections). Each stage then calls the low-level code to execute an action, such as checkout from source code manager, invoke a build specification or run unit framework tests. This approach is seen in figure 9. There are three distinct levels in a pipeline job – a Jenkinsfile than calls the pipeline configuration for the repository, the pipeline code that separates the job into multiple steps, and the low-level code to execute the job’s action.

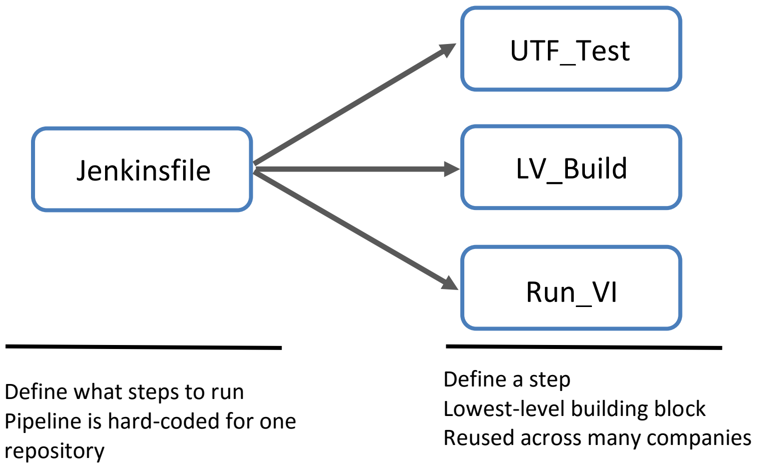

Jenkinsfile
UTF_Test
LV_Build
Run_VI
Define what steps to run
Pipeline is hard-coded for one repository
Define a step
Lowest-level building block
Reused across many companies

Figure 3: Single Repository CI Architecture

However, if there is only one repository in the organization, it is not necessary to have the three different levels. The Jenkinsfile can define the stages and call the low-level code, seen in figure 8. For the multiple repository case, the purpose of separating the stages into a separate script from the Jenkinsfile is to make it reusable among many Jenkinsfiles and repositories. It also becomes easier to edit and update; the user will only have to modify one location and have the change propagate to all repositories, rather than modify multiple instance of the same code.

[3] Jenkins reference

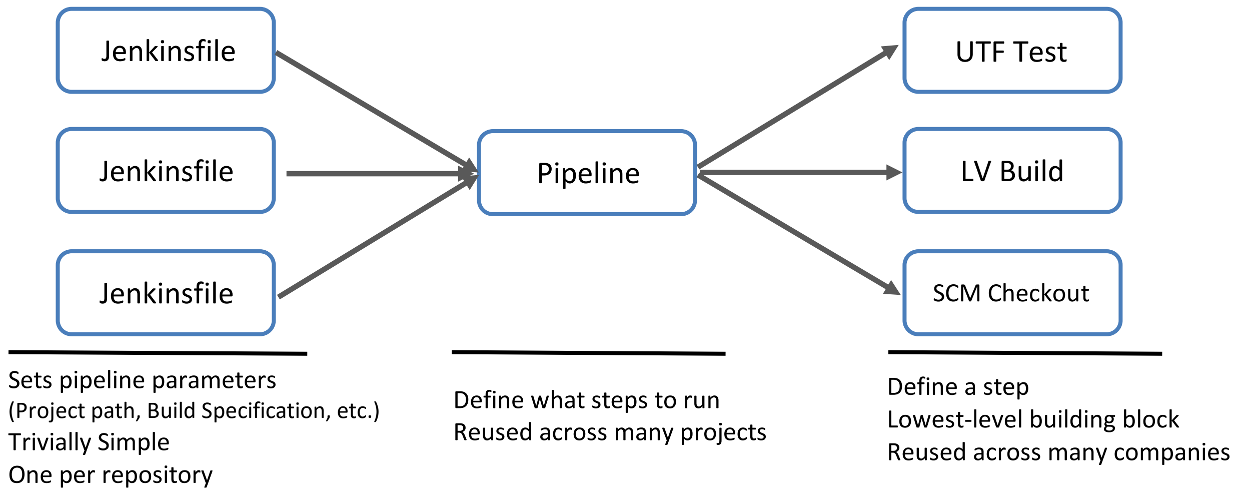


Figure 4: Multiple Repository CI Architecture

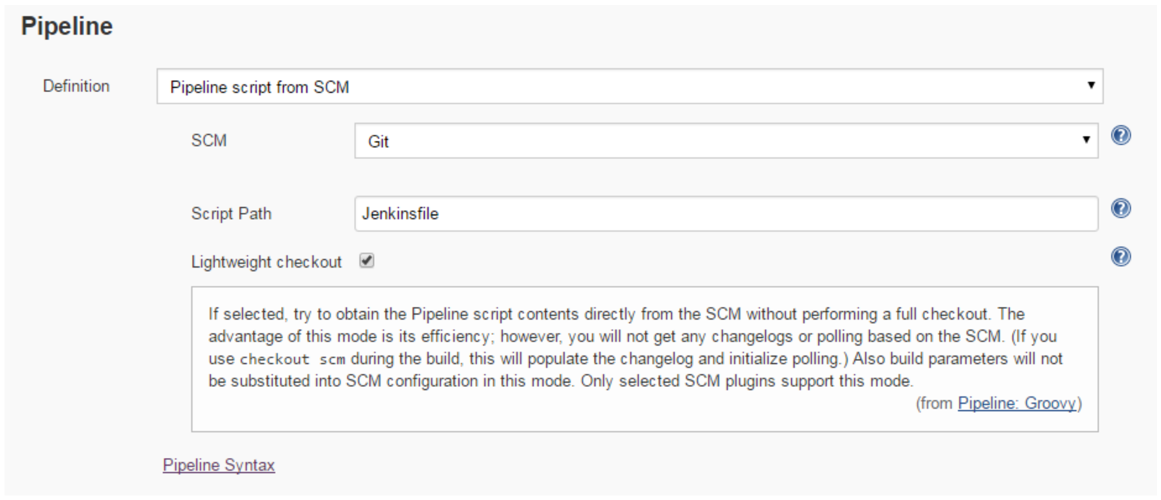
Building Pipeline Job from Jenkinsfile

As discussed earlier, a Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. While the syntax for defining a Pipeline, either in the web UI or with a Jenkinsfile, is the same, it's generally considered best practice to define the Pipeline in a Jenkinsfile and check that in to source control.[4]

For the projects developed with LabVIEW CLI, GitHub was used as a source code management. However, a Jenkins Pipeline is configurable with any source code control. For a more in depth look at Jenkinsfile, please visit here. Use the following steps to configure a pipeline job with a Jenkinsfile:

1. Create a new Pipeline job by choosing **New Item** from the main Jenkins dashboard. Choose Pipeline and name the job accordingly.
2. When configuring a pipeline job, under the Pipeline definition, instead of writing a Groovy script in the script editor, we can choose to use a Jenkinsfile script from a source code management, seen in the following figure:

[4] https://jenkins.io/doc/



Pipeline

Definition: Pipeline script from SCM

SCM: Git

Script Path: Jenkinsfile

Lightweight checkout ✓

If selected, try to obtain the Pipeline script contents directly from the SCM without performing a full checkout. The advantage of this mode is its efficiency; however, you will not get any changelogs or polling based on the SCM. (If you use `checkout scm` during the build, this will populate the changelog and initialize polling.) Also build parameters will not be substituted into SCM configuration in this mode. Only selected SCM plugins support this mode.
(from Pipeline: Groovy)

Pipeline Syntax

Figure 5: Jenkinsfile Configuration for a Pipeline Jobs

Note: This Jenkins file must be in the root of the SCM repository.

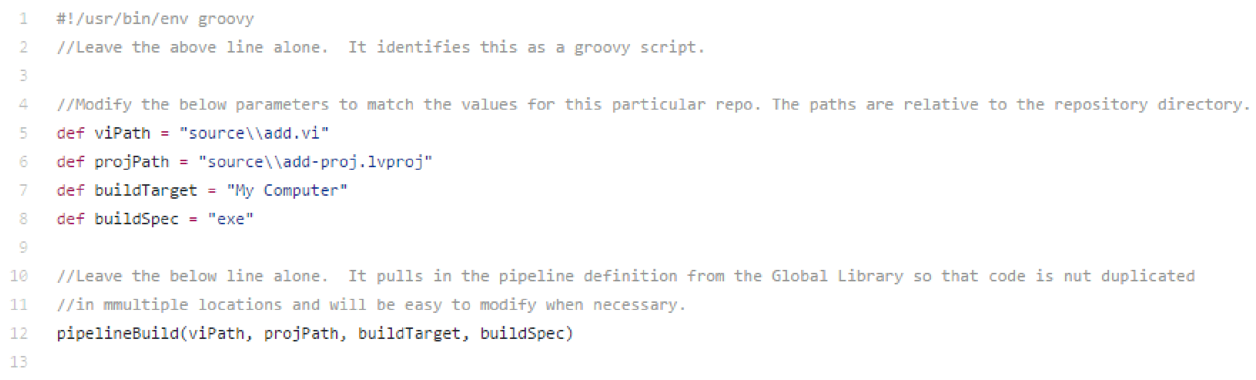
3. The pipeline job created will use the Jenkinsfile located in your SCM repository to carry out the build.

Scripting Jenkinsfile

The Jenkinsfile and all other scripts for the pipeline jobs are written in Apache groovy, an object-oriented programming language for the Java platform. Therefore, every Jenkinsfile must have "#!/usr/bin/env groovy" in the first line.

The Jenkinsfile can be easily be customized for each different repository and project. The method that was used for the tutorial example and the LabVIEW DCAF project.

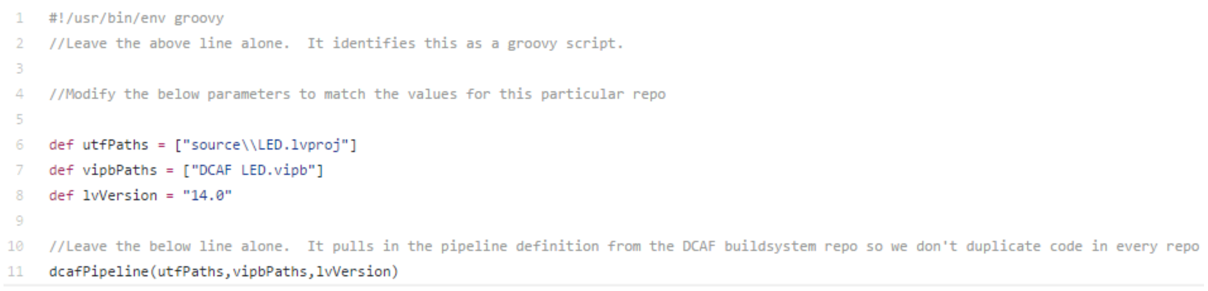
The Jenkinsfile we have created calls the appropriate functions that can perform the builds, rather than the code for the pipeline job. For example, the following Jenkinsfile makes a call to the pipelineBuild method with defined parameters seen in the following figure:



```
#!/usr/bin/env groovy
//Leave the above line alone.  It identifies this as a groovy script.

//Modify the below parameters to match the values for this particular repo. The paths are relative to the repository directory.
def viPath = "source\\add.vi"
def projPath = "source\\add-proj.lvproj"
def buildTarget = "My Computer"
def buildSpec = "exe"

//Leave the below line alone.  It pulls in the pipeline definition from the Global Library so that code is nut duplicated
//in mmultiple locations and will be easy to modify when necessary.
pipelineBuild(viPath, projPath, buildTarget, buildSpec)

```

The pipelineBuild() method, which is contained in the global library, will be called by Jenkins when this pipeline job is initiated. The defined parameters will be unique for each Jenkinsfile, i.e. each repository. When the pipeline job begins, Jenkins will pass in the parameters to the pipelineBuild() method.

The purpose of reserving the Jenkinsfile exclusively for function calls is to avoid unnecessary code duplication and minimize developer efforts. When the code for the pipeline job exists within the global library, any changes to it will propagate to any Jenkinsfile that relies on that function. The test developer will not have to duplicate efforts by modifying the code in all Jenkins files, but rather the code contained in the global library. Another example of a Jenkinsfile is the following used in the LabVIEW DCAF LED repository, seen in the following figure:

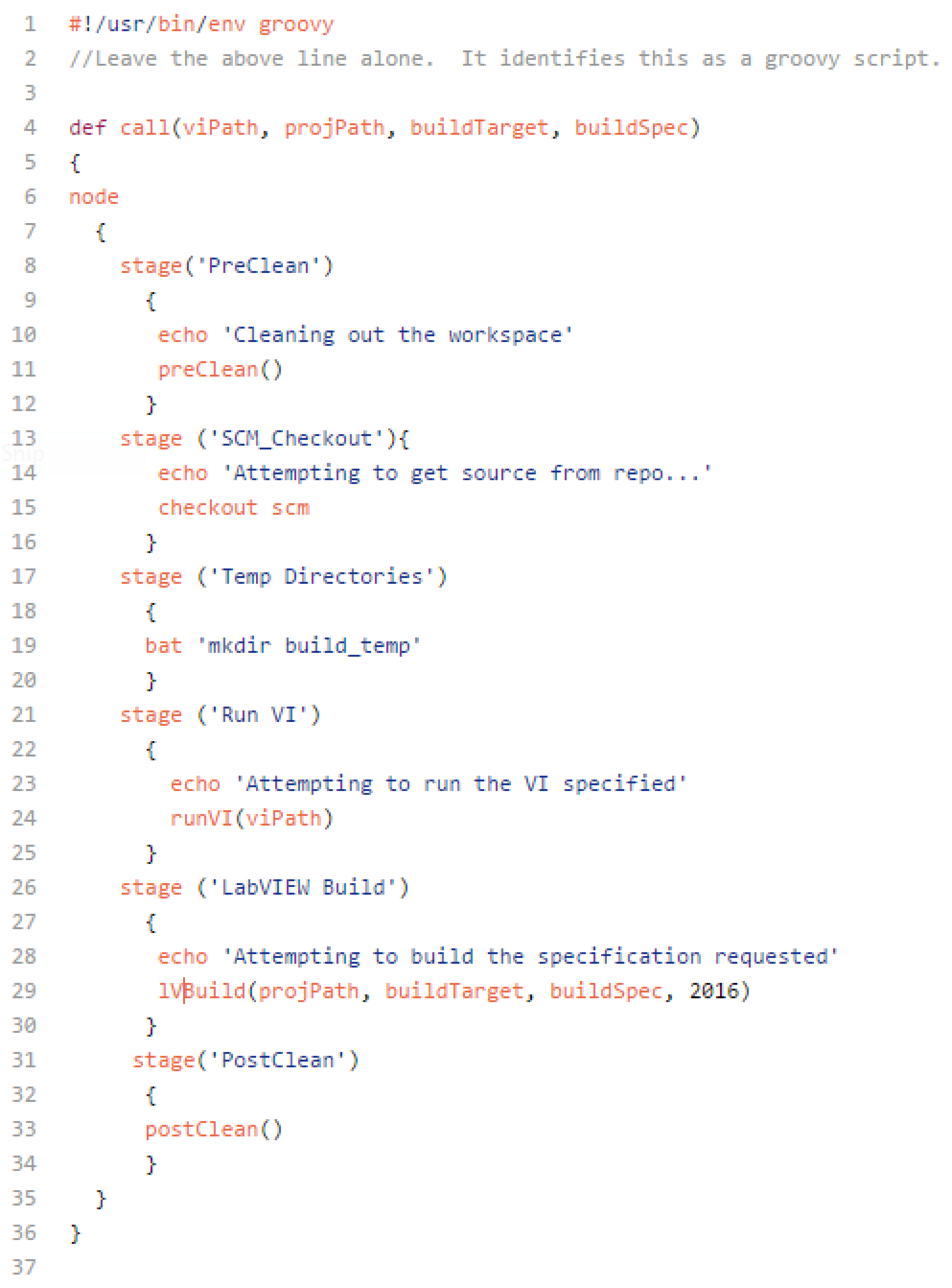


```
#!/usr/bin/env groovy
//Leave the above line alone.  It identifies this as a groovy script.

//Modify the below parameters to match the values for this particular repo

def utfPaths = ["source\\LED.lvproj"]
def vipbPaths = ["DCAF LED.vipb"]
def lvVersion = "14.0"

//Leave the below line alone.  It pulls in the pipeline definition from the DCAF buildsystem repo so we don't duplicate code in every repo
dcafPipeline(utfPaths,vipbPaths,lvVersion)
```

Again, the same principle was used here. This Jenkinsfile calls into the dcafPipeline() method with parameters specific to the LED repository.

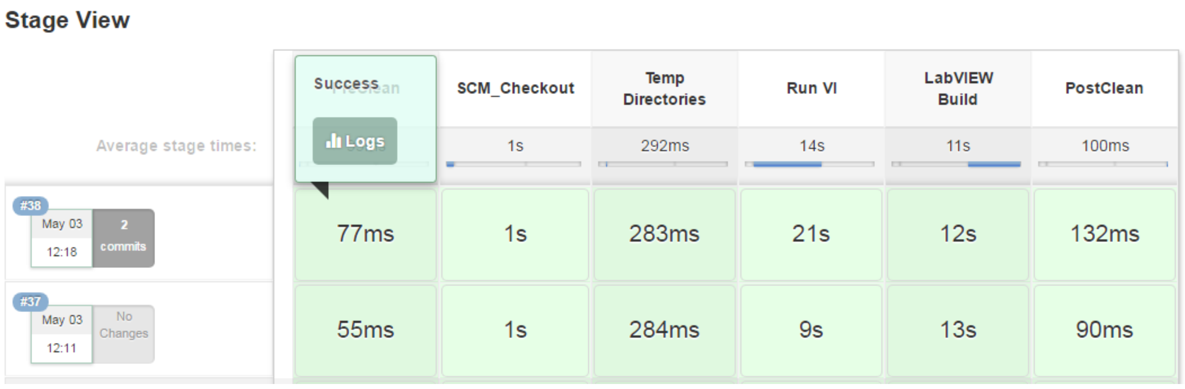
Pipeline Stages and Steps

As seen above, the Jenkinsfile invokes the pipelineBuild() method. The code is seen in the following figure:



```groovy
#!/usr/bin/env groovy
//Leave the above line alone.  It identifies this as a groovy script.

def call(viPath, projPath, buildTarget, buildSpec)
{
node
  {
    stage('PreClean')
      {
       echo 'Cleaning out the workspace'
       preClean()
      }
    stage ('SCM_Checkout'){
       echo 'Attempting to get source from repo...'
       checkout scm
      }
    stage ('Temp Directories')
      {
      bat 'mkdir build_temp'
      }
    stage ('Run VI')
      {
        echo 'Attempting to run the VI specified'
        runVI(viPath)
      }
    stage ('LabVIEW Build')
      {
       echo 'Attempting to build the specification requested'
       lVBuild(projPath, buildTarget, buildSpec, 2016)
      }
     stage('PostClean')
      {
      postClean()
      }
  }
}

```

The pipelineBuild script does not contain the code for the build either – the build is defined in three layers. The second phase defines the stages for the Jenkins build job and calls the methods for the stage to execute. A stage step is a primary building block in Pipeline, dividing the steps of a Pipeline into different sections and creates a visualization of the progress using the "Stage View" plugin. When the pipeline builds, it will divide your test into the different stages as seen in the figure below:



Stage View

		SCM_Checkout	Temp Directories	Run VI	LabVIEW Build	PostClean
Average stage times:		1s	292ms	14s	11s	100ms
#38 May 03 12:18 2 commits	77ms	1s	283ms	21s	12s	132ms
#37 May 03 12:11 No Changes	55ms	1s	284ms	9s	13s	90ms

In the Stage View, seen above, each block corresponds to a stage defined in the piplinebuild() script.

Again, note that each step contains another function call rather than containing the code that performs each step. This is like the Jekinsfile structure. Each stage calls into the necessary separate functions, making the pipeline job easy to maintain and avoids code duplications. Multiple jobs could access these functions that exist within the global library. It is possible to include all the code within this pipeline job as well. However, for the reasons stated above, we chose to only include function calls in the pipeline build script. The separate functions, such as RunVI and LVBuild, are all different scripts that exist in the global library.

These functions are wrappers for the functions of the LabVIEW Command Line interface can. The following sections cover these methods. Again, for the DCAF pipeline, a similar methodology was used.

Configuring Global Library for Groovy Scripts

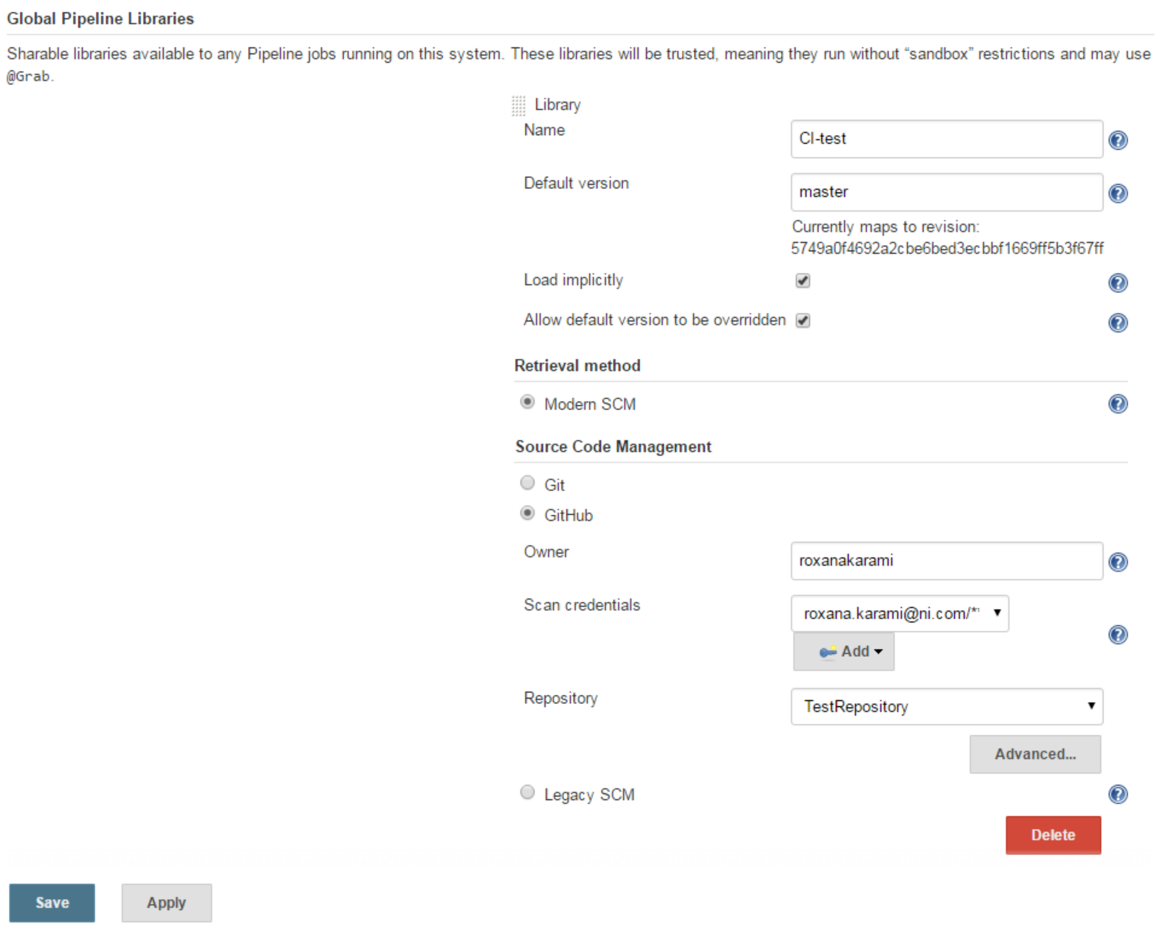
Note: “Pipeline: Shared Groovy Libraries” is the required plugin for this functionality.

The global library contains all the groovy scripts necessary for each separate method that a Jenkinsfile or any other groovy scripts may call. These are methods to Run VIs or Build any pre-defined items in a project’s build specification. Multiple Jenkinsfiles or other functions can invoke these scripts if Jenkins has access to these files from a source code repository.

Jenkins will need to be configured so that it can load these scripts automatically and use them for any Jenkinsfile:

1. From the main Jenkins Dashboard, navigate to **Manage Jenkins>>Configure System.**

2. Underneath the heading, Global Pipeline Libraries, you can point to the library of these functions. The following figure shows the setup in Jenkins for the example repository located at github.com/roxanakarami/TestRepository.



Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library

Name	CI-test
Default version	master Currently maps to revision: 5749a0f4692a2cbe6bed3ecbbf1669ff5b3f67ff
Load implicitly	✓
Allow default version to be overridden	✓

Retrieval method

(●) Modern SCM

Source Code Management

( ) Git
(●) GitHub

Owner	roxanakarami
Scan credentials	roxana.karami@ni.com/**
Repository	TestRepository

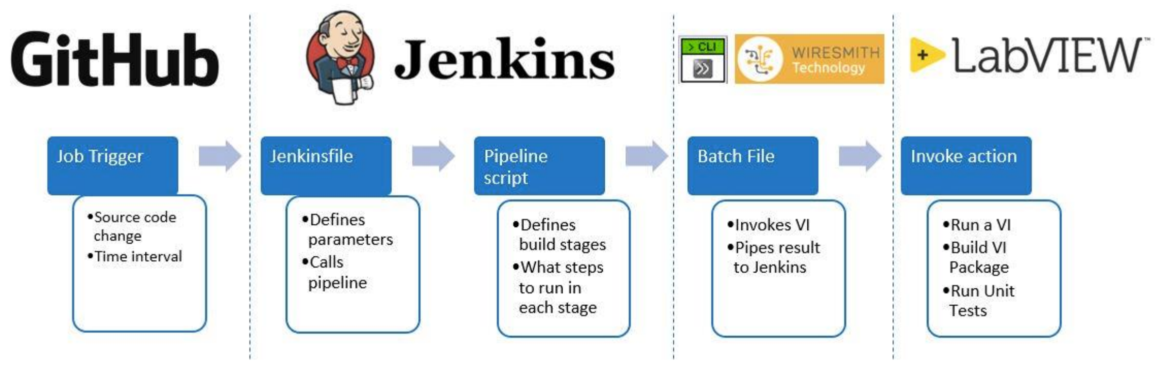
( ) Legacy SCM

3. It is recommended that you leave the Retrieval method as Modern SCM.
4. Jenkins will autopopulate with your SCM if not using Git or Github with the correct plugins. Configure these settings so that Jenkins points correctly to your repository.
5. You may add multiple sets of global libraries, but beware of namespace conflicts.

Note: that these global scripts must be contained in a folder entitled "vars" or "src" in a source code repository. You will receive a build error otherwise.

Examples of this can be seen in https://github.com/LabVIEW-DCAF/buildsystem.

Pulling It Together



The figure above outlines the entire the continuous integration process. Github, or any SCM, triggers a pipeline job a change is committed. A job can also be manually triggered or trigged at a specified time interval. The Jenkinsfile defines the build parameters an invokes pipeline scripts when a build is initiated. The low-level code uses batch scripts to invoke the LabVIEW Command Line Interface, which invokes the LabVIEW development environment to run VIs, run unit tests, etc.

Once you have your code written, use the following steps to run your builds.

1. Configure a pipeline job in Jenkins and create a Jenkinsfile in your repository.
2. Configure a shared library and use any groovy scripts that are necessary for your build from the scripts that install with the LabVIEW-CLI package.
3. Initiate the job in Jenkins, and look at the console output for the build results.