



## Pipelines - A better approach to automated build jobs?

Erik Johansson - erikjo9@kth.se

Felix Eder - felixed@kth.se

DD2482

Automated Software Testing and DevOps

VT2019

### Abstract

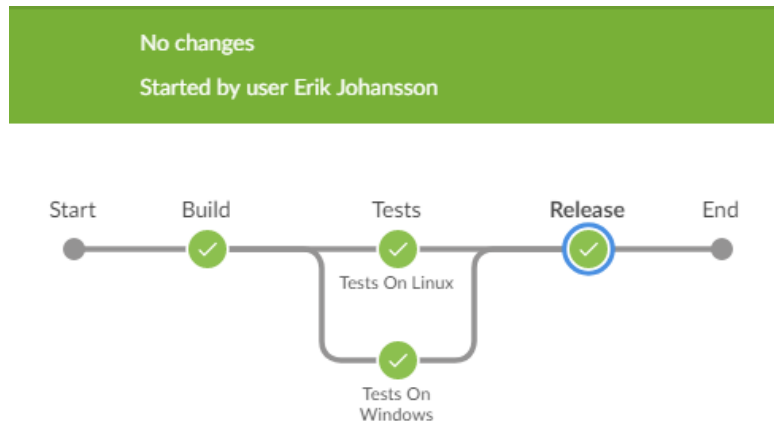


Figure 1: Example of a build pipeline.

This paper explains and compares two common approaches when creating automated or semi-automated build jobs for continuous integration and continuous deployment (CI/CD). The approach of manually configuring build jobs using a graphical user interface (GUI) is compared with the more modern approach of creating a pipeline (figure 1) using a domain specific language (DSL). The advantages and disadvantages in the categories; configuration, functionality, maintainability, security and scalability are discussed. The conclusion is that while the traditional GUI approach could do well in a minimal and simplistic development environment, pipelines are more well suited to the complex and modern CI/CD development environment of software projects today.

**Keywords:**

Pipeline, Jenkins, Continuous Integration, Continuous Delivery, Continuous Deployment, DevOps.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>1</b>
2.1	Software development . . . . .	1
2.2	Continuous Integration and Delivery . . . . .	1
2.3	Build jobs . . . . .	2
2.3.1	Graphical configuration and maintenance . . . . .	3
2.3.2	Pipeline code and maintenance . . . . .	4
<b>3</b>	<b>Comparison</b>	<b>5</b>
3.1	Configuration . . . . .	5
3.2	Functionality . . . . .	6
3.3	Maintainability . . . . .	6
3.4	Security . . . . .	7
3.5	Scalability . . . . .	8
<b>4</b>	<b>Conclusion</b>	<b>9</b>
<b>5</b>	<b>References</b>	<b>10</b>

# 1 Introduction

One integral, but often overlooked, part of working with software, is deployment. The term software deployment comprises all the steps needed to make a software program ready to use. Historically, mainly regarding large mainframes, deployment meant a visit by a system architect to set everything up. With the rise of personal computers the end-users themselves took care of the deployment. This was first done with cartridges that enabled the users to install the software themselves and later with floppy disks, CD:s and flash drives. Because of the offline nature of this type of deployment, it was infrequent and static. With the advent of internet technology software deployment changed radically. Software could now be released to millions of users at the same time while the scheduling of the deployments were in the hands of the developers. This enabled strategies such as continuous delivery to be a valid option for deployment.

Software deployment today is an established practice that has a wide variety of roles. It handles the release, the installation, uninstallation and various types of update handling. There are multiple different tools that handle deployment and as a result it has become easier and easier to release software. However, new deployment technologies, such as pipelines, are gaining popularity. How does traditional software deployment strategies compare to new techniques, like pipelines?

# 2 Background

This section aims at describing the different concepts used in this paper.

## 2.1 Software development

The term "Software development" has been widely used for many years and defines the work process for developing different types of software. There exists many variations of Software Development, however they all have in common that they aim to create and maintain some sort of software (Pfleeger & Atlee 1987).

## 2.2 Continuous Integration and Delivery

Within Software Development there is the aspect of updating or further developing the target software. Continuous Integration and Continuous Deployment (CI/CD for short) is a concept for how software can be integrated,

checked and deployed in a safe way. Continuous Integration (CI) is the process of continually merging branches to the master branch and thus avoiding merge conflicts. Each time the branches are merged to the trunk, a number of automated checks and tests are run. These checks can include building the project, running the test suite, calculate test coverage and so on. These checks enable the developers to focus on writing good code and don't worry about running the build and test suite themselves every time they finish working on something (Fowler & Foemmel 2006).

Continuous Deployment (CD) is the practice of automating the deployment of the software. It can be configured in various ways, but usually it is set up with a CI service (like Travis or Jenkins). When a new commit is added to a specific branch, the CI runs the checks and if everything is okay, it will automatically be pushed to the hosting platform of the software. Continuous Deployment should not be confused with Continuous Delivery, which is differentiated by the fact that software being delivered can be deployed at any time. The software still needs to manually be deployed, which is automated in Continuous Deployment (Leppänen et al. 2015).

## 2.3 Build jobs

To achieve CI/CD the concept of automated or semi-automated build jobs is used. A build job consists of a set of steps or instructions that are run on or by the software being developed. These steps usually include performing checks and tests on the software or even deploying the software to production. The purpose of build jobs is to automate steps that would otherwise have to be run manually by the developers of the software.

Another aspect of build jobs is that they may be triggered by different types of events (see figure 2). An update to the source

code in the form of a patch or a new version are examples of events that could trigger a build job to execute. Another example is scheduled executions of the build job. A common practise is to schedule build jobs of the software

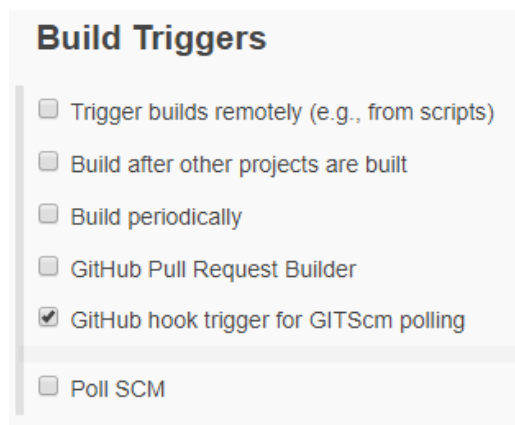


Figure 2: Example of build triggers in a Jenkins build job.

at the end of each working day and these build jobs are usually referred to as "daily" or "nightly" build jobs (for example Firefox Nightly).

The final aspect of a fundamental build job is the notion of build files. Each build job commonly produces files that include binaries of the software, log output, test and other checks results or even large compressed file structures (or "tarballs"). Many of these files are seen as "artifacts" of the build job that should be saved for development purposes.

While these aspects form the basis for almost all build jobs, there are many differences in how to implement a system capable of defining and executing such build jobs. A simple build job could just be a Linux Cron job scheduled to run in some interval. However, for functionality above this, a more complex CI server is used. Today there are many available CI server solutions with the most common being Jenkins (formerly Hudson), Circle CI or Travis CI. While there are many differences in the CI servers, they usually share the concepts of how to define and store the configuration of build jobs. Today there exists two main approaches for this with the more modern approach being pipelines. Both of these approaches are described in further detail below and the aim of this essay is to present and compare their benefits and drawbacks (*Jenkins Job Builder — jenkins-job-builder 2.10.1.dev1 documentation* 2019).

### 2.3.1 Graphical configuration and maintenance

The more traditional way to configure build jobs in various CI server software is to use a graphical interface. Depending on the CI server this will involve creating a build job either from a template or from a blank configuration. The configuration usually includes settings for the source control (branch selection and fetch strategy), triggers (pull requests, merges or scheduled) as well as the different steps the job has to perform. When using the graphical UI this will include filling in the configuration page top-to-bottom as well as selecting and filling in each build step from a set of available steps (see figure 3). For this type of configuration there usually exists much information and documentation within the GUI itself about what configuration is needed and how it should be

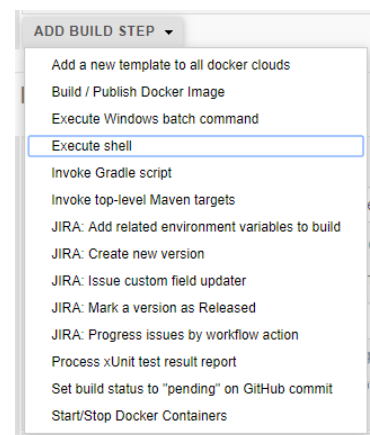
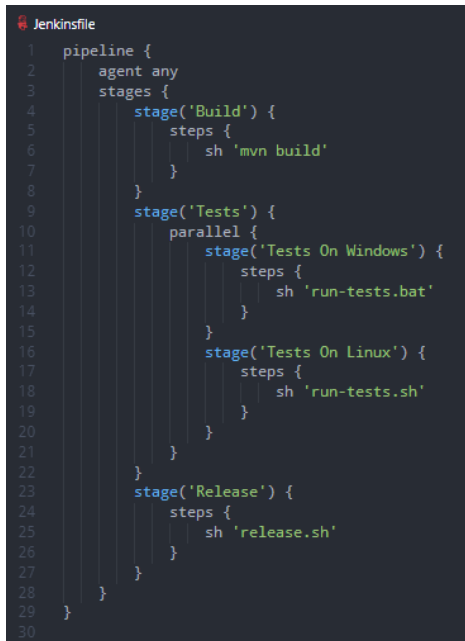


Figure 3: Selection of available build steps

done. The job configuration then is stored internally on the CI server unless backed up in some other fashion. While it usually is possible to avoid the GUI by using an API, we chose to regard this as manual configuration since the job still needs to be created by an admin user on the CI-server (using the API) that is still stored internally on the server (Smart 2011).

### 2.3.2 Pipeline code and maintenance

An alternative way to define a build job is to use pipelines. Pipelines is a more modern concept of defining jobs using only code that defines how and when the build job should run. The code is usually in a domain specific language (DSL) that makes it easier to define the steps necessary to the build job.



```
1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         sh 'mvn build'
7       }
8     }
9     stage('Tests') {
10      parallel {
11        stage('Tests On Windows') {
12          steps {
13            sh 'run-tests.bat'
14          }
15        }
16        stage('Tests On Linux') {
17          steps {
18            sh 'run-tests.sh'
19          }
20        }
21      }
22    }
23    stage('Release') {
24      steps {
25        sh 'release.sh'
26      }
27    }
28  }
29 }
30
```

Figure 4: A pipeline build job in Jenkins declarative pipeline syntax (DSL).

Another important part of pipelines is to also store this build job configuration code together with the software source code in the version control system. The idea behind this is that the build job(s) usually directly relate to the functionality of the software. And since the software functionality continuously changes the build job might also need to be changed at the same time. This way there may be as many versions of the build job as there are versions of the software itself.

A final part of pipelines that the more traditional approach usually lacks is concurrency. While most graphical configurations just include a list of steps to be performed sequentially, the pipeline DSL syntax allows for splitting steps into any number of sections that may run either sequentially or concurrently, see figure 4 (Armenise 2015).

### 3 Comparison

While both of the mentioned approaches to build jobs achieve the same goal, that is, defining and configuring build jobs, there still exists many differences between them that may be compared in much further detail. This is the aim of this section of the essay. We have chosen to compare the two approaches within different categories. These categories are; configuration, functionality, maintainability, security and finally scalability.

#### 3.1 Configuration

Configuration of a job refers to all efforts by the developer required for a build job to be created and configured.

If we first look at the more traditional approach we see that all configuration is done on the CI server itself using some graphical user interface (GUI). In this case the configuration will usually consists of sections defining the source code access and selection, the triggers (see figure 5), the build steps and finally any post build steps. All elements in the GUI are mainly self explanatory since they include descriptive names, help

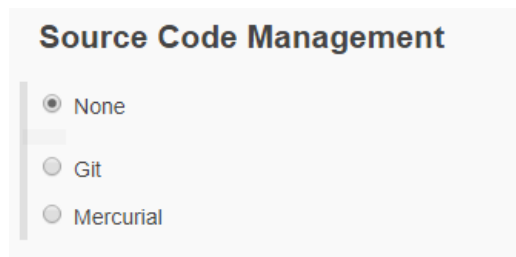


Figure 5: Example of source code management for a Jenkins build job.

annotations as well as drop down lists or selection boxes. This means that the developer needs minimal knowledge of the CI system itself in order to quickly configure a build job. An important downside however is that there usually exists security measures in place on the CI server that prevent any developer from creating a job. This is because the CI server stores many sensitive credentials and since access to job configuration allows leaking such credentials the configuration is usually restricted to admins.

If we instead look at the concept of pipelines we first see that the configuration needed on the CI server itself is very minimal. Essentially the only required configuration is the source control and not any specific build job. Instead the configuration of the build job(s) exists in the source control system itself. There the configuration is described using a domain specific language (DSL) that is tailored to describe a build job. This means that the syntax of the DSL needs to be learned using external documentation in



order to get started with pipelines. Also, since the configuration exists in the source control, it may be created or edited by any developer with write access to the software source code.

## 3.2 Functionality

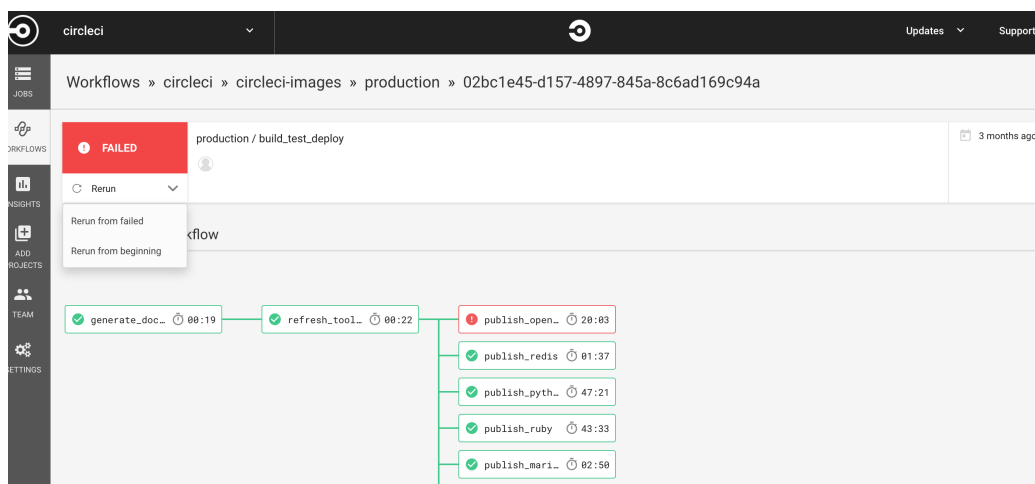


Figure 6: A Circle CI pipeline with concurrent build steps

While the functionality of the created build jobs is essentially the same for the two approaches, there are some possible differences when looking at specific CI server implementation of pipelines. When looking at different GUI configurations a common theme is a sequential list style definition of the build process. This means that the build job only has support for build steps that are executed sequentially. But, since many build steps usually are independent of each other, this may be seen as a major drawback. Using a DSL we are however able to order, group and nest build steps in such a way that we can define which steps may be run concurrently. While this type of concurrency is not supported by all pipeline implementations, it is still supported by two of the most commonly used CI servers (Circle CI and Jenkins), see figure 6.

## 3.3 Maintainability

When it comes to maintainability there are some major differences between pipelines and the more traditional GUI job configuration. Build jobs in general closely relate to the functionality of the software in question. This means that as the software develops, so must their build jobs.

Using GUI configuration to maintain build jobs requires extra efforts by the developers. When a change is made to the software that breaks the build job(s), the developer has to make sure that the jobs are updated to support the new code. As not all developers of a software usually have access to the configuration of build jobs, this approach might need additional work performed by an administrator of the CI-server. For complex changes in software made by a single developer, administrator privileges might even need to be granted to the developer in order to complete the maintenance of the build job correctly.

For pipelines, a main aspect is to utilize the concept known as "configuration as code" (CASC). This means that the pipelines are maintained simply by changing the code in the source control system (that is the pipeline configuration code). This means that the developers are able to maintain the pipeline at the same time as they are updating the code for the software. Another major benefit of CASC is that the job configuration from before the update to the software also is available in the history of the source control. This means that when releasing new versions of a software the old versions may still be build without any extra maintenance from the developers.

### 3.4 Security

The aspect of security in build jobs is very essential. As previously stated, CI servers generally hold a substantial amount of sensitive credentials to different services related to the development and deployment of the software. Besides this, CI servers usually have network access to any connected companies internal networks in order to utilize internal services for build processes. This in combination with the fact that build jobs can execute any arbitrary code on the host system creates a major security risk that should not be ignored.

For GUI configuration the creation and maintenance of jobs is performed directly on the CI server. Because of the mentioned security risks the CI server is most likely also configured with user authentication and authorization in order to limit job configurations to a set of trusted CI server users (usually admins). With this approach there will thereby be a tightly restricted access to the configuration of jobs and thereby also sensitive aspects of the CI server itself.

With pipelines the control of job creation and maintenance is mainly left to the developers with write access to the software source code. Depending on the configuration of the CI server this means that there could be security risks when using pipelines. The server could for example be configured to trust every change made to a pipeline in the source code. However, this way,

the security vulnerabilities previously discussed are all exposed for exploits by malicious developers of the code. Another solution is to not trust any changes to the pipelines in the source code. While this drastically improves security, developers are limited since any changes then have to be audited and approved by a CI-server administrator. A final solution is to let pipelines run in a sand-boxed environment where sensitive parts of the server can not be accessed. These sandbox environments are extremely complex in underlying nature so they can most likely never guarantee that exploits do not exist.

### 3.5 Scalability

When developing larger and more complex software applications it is very common to have the application software split into different sub-modules that each have their own versions and development life cycle. In this aspect there are some major drawbacks when configuring build jobs using the GUI of the CI-server. Since the job configurations are stored on the CI-server and CI-servers lacks support for multi-version jobs, this means that new build jobs need to be created and configured for each version of each software module. While it is possible to create templates for creating new and similar jobs, this process may be regarded as quite time consuming.

As previously mentioned one of the advantages of pipelines is that a multi-version job configuration may be created, as it simply is code on the source control for the software. This also allows for fast scaling since code just has to be copied and modified for new sub-modules of the software that follow the build process of other sub-modules. Also, an additional advantage that comes from the fact that pipelines are written in code is that pipeline libraries may be created and used across any other pipeline configuration. This drastically reduces the cost of scaling the application into new versions and modules since there needs to be much less code repetition when creating the new pipeline files.

## 4 Conclusion

	Graphical User Interface	Pipeline Code
Configuration	+ Simple to get started - Config stored on server - Requires server admin	+ Config stored in source control + Minimal configuration needed - Requires learning DSL syntax
Functionality	- Usually no concurrency	+ Usually supports concurrency
Maintainability	- Requires server admin	+ Can also be done by developers
Security	+ Tight control over jobs + Avoids many exploits	- Unless configured, no job control + Sandbox mode improves security
Scalability	- One job per module/branch + Templates may help a bit	+ Automatic jobs for many modules/branches + Support for libraries to reduce config size

To conclude this comparison there are some important differences between configuring jobs using a GUI or using pipeline code. There are both advantages and disadvantages that are highlighted in Table 1.

The GUI approach is simple to set up for a minimal project or application and it could be regarded as more secure, since access to job configuration (and thereby remote code execution) is more tightly controlled. However, there are major drawbacks in contrast to pipelines when it comes to concurrency in build jobs, maintainability of the build jobs, and scalability of the build jobs. Pipelines are more adjusted to an environment where there are many developers working on a complex multi-module and multi-version application. This is because the build job configuration becomes a part of the source code itself, meaning that the build job configuration may be shared or modified across application modules or versions. And even though the learning curve is steeper for pipelines they still outweigh the traditional GUI approach in most modern DevOps use cases.

## 5 References

- Armenise, V. (2015), Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery, in ‘Proceedings of the Third International Workshop on Release Engineering’, IEEE Press, pp. 24–27.
- Fowler, M. & Foemmel, M. (2006), ‘Continuous integration’, *Thought-Works*) [http://www.thoughtworks.com/Continuous Integration.pdf](http://www.thoughtworks.com/Continuous%20Integration.pdf) **122**, 14.
- Jenkins Job Builder — jenkins-job-builder 2.10.1.dev1 documentation* (2019). [Online; accessed 26. Apr. 2019].  
**URL:** <https://docs.openstack.org/infra/jenkins-job-builder>
- Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V. & Männistö, T. (2015), ‘The highways and country roads to continuous deployment’, *Ieee software* **32**(2), 64–72.
- Pfleeger, S. L. & Atlee, J. M. (1987), *Software engineering*, Macmillan New York.
- Smart, J. F. (2011), *Jenkins: The Definitive Guide: Continuous Integration for the Masses*, " O'Reilly Media, Inc."