

*|* Posted by [**Andrew Phillips**](https://www.infoq.com/profile/Andrew-Phillips)[**Kohsuke Kawaguchi**](https://www.infoq.com/profile/Kohsuke-Kawaguchi) on Mar 29, 2014. Estimated reading time: 20 minutes *|* [3**Discuss**](https://www.infoq.com/articles/orch-pipelines-jenkins#theCommentsSection)

* Share
* **[Share](https://www.addthis.com/bookmark.php?v=250&username=infoq)**
* |
* Orchestrating Your Delivery Pipelines with Jenkins
* [**Read later**](https://www.infoq.com/articles/orch-pipelines-jenkins)
* [**Reading List**](https://www.infoq.com/showbookmarks.action)

In a [previous article](http://www.infoq.com/articles/preparing-for-cd-in-the-enterprise), we covered useful preparation steps for your Continuous Delivery implementation, including defining your pipeline phases, preconditions and required approvals, owners and access control requirements, resource requirements such as number of concurrent build machines, identifying which phases can be run in parallel for speedup, and more.

Here, we will discuss how to put a number of these recommendations into practice in a concrete setting, namely setting up a delivery pipeline in [Jenkins](http://jenkins-ci.org/). Many of the steps we will present carry over to other Continuous Integration (CI) and orchestration tools, and there are analogous extensions or core features for many of the plugins we will introduce, too.

We are focussing here on Jenkins, however, because it is the most widely-used Continuous Integration server out there. Even if you are using different CI servers or services in your environment, it should be relatively easy to experiment with the steps we will cover in a “sandbox” Jenkins installation, before carrying them over to your own CI environment.

Prerequisites

**Related Vendor Content**

[**Top 10 Agile Techniques Every Team Should Know**](https://www.infoq.com/vendorcontent/show.action?vcr=4244&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Top 10 Java Performance Problems**](https://www.infoq.com/vendorcontent/show.action?vcr=4398&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Top 10 AWS Cloud Security Risks**](https://www.infoq.com/vendorcontent/show.action?vcr=4323&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Verticles, EventBus, Reactive systems from Red Hat Developers. Download Vert.x Cheat Sheet now.**](https://www.infoq.com/vendorcontent/show.action?vcr=4384&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Addressing Threats to your Infrastructure with Threat Modeling, Incident Command & DDoS Mitigation**](https://www.infoq.com/infoq/url.action?i=16153&t=f)

**Related Sponsor**

[](https://www.infoq.com/infoq/url.action?i=16144&t=f)

Fastly’s edge cloud platform powers secure, fast and reliable online experiences for the world’s most popular digital businesses. [**See for yourself.**](https://www.infoq.com/infoq/url.action?i=16158&t=f)

Before diving into building our delivery pipeline with Jenkins, we need to discuss two important prerequisites: that our pipeline, or at least the part of the pipeline that we are looking to implement here (going all the way to Production may not be the most sensible initial goal), is

* Predictable and standardized, i.e. that the steps and phases we want to run each time the pipeline is triggered are the same
* Largely automated. We will cover ways to handle manual approvals to “bless” a certain build, but that is about it.

If the current release process does not display these characteristics, i.e. every release ends up being a little bit different, or there are still many manual steps (reviewing test plans, preparing target environments) that are required, building a pipeline via a Continuous Integration tool or generic automation orchestrator may not be the most appropriate step at this point.

It is probably advisable to first increase the level of standardization and automation, and to look at tools such as [XL Release](http://xebialabs.com/products/xl-release) in the “release coordination” or “Continuous Delivery release management” category to help with that.

What We Will Cover

We will cover the following topics to help build your delivery pipeline:

1. [Ensuring reproducible builds](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc1)
2. [Sharing build artifacts throughout the pipeline](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc2)
3. [Choosing the right granularity for each job](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc3)
4. [Parallelizing and joining jobs](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc4)
5. [Gates and approvals](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc5)
6. [Visualizing the pipeline](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc6)
7. [Organizing and securing jobs](https://www.infoq.com/articles/orch-pipelines-jenkins#_Toc7)

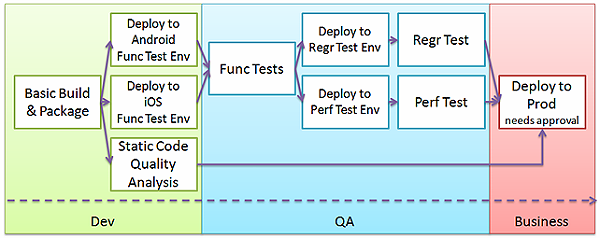
Our example project

In order to make some of the scenarios and approaches we will outline more tangible, we’ll base this discussion around a sample development project. Let’s assume we’re working on the server-side component of a mobile app for Android and iOS. Our delivery process for our application is as follows:

1. Whenever a code change is committed, we build the code and, if successful, package up the current version as a candidate version for release (*Basic Build & Package* job).
2. Now that we know that the code compiles and passes our unit tests, we trigger a code quality build that carries out a bunch of static analysis to verify code quality (*Static Code Quality Analysis* job).
3. The static analysis can take quite some time, so in parallel we deploy the candidate version to two functional testing environments, one for the Android and one for the iOS app version, in preparation for testing (jobs *Deploy to Android Func Test Env*and *Deploy to iOS Func Test Env*). We use two test environments so we can easily identify differences in how the backend behaves when talking to either version of the app.
4. When both deployments have completed, we trigger functional tests, with the iOS and Android apps talking to their respective backend (*Func Tests* job).
5. If the functional tests pass, we carry out parallel deployments of our release candidate to a regression and a performance test environment (jobs *Deploy to Regr Test Env* and *Deploy to Perf Test Env*). The completion of each deployment triggers the appropriate tests (jobs *Regr Test* and *Perf Test*).
6. If the regression and performance tests and our static code analysis build from earlier complete successfully, the candidate is made available for business approval, and the business owner is notified.
7. The business owner can approve, in a manual step, the candidate build.
8. Approval triggers and automated deployment to production (*Deploy to Prod* job).

Schematically, our delivery pipeline looks like this:

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig1-large.png)

**Figure 1: Our example project’s delivery pipeline**

It’s worth noting that this is not intended to be interpreted as a “good”, “bad” or “recommended” pipeline structure. The pipeline that works best for you will not be a direct copy of this example, but will depend on *your* applications and *your* process!

Ensuring reproducible builds

One of the key principles of our pipeline is that we produce *one* set of build artifacts which we will pass through the various pipeline stages for testing, verification and, ultimately, release. We want to be sure that this is a reliable process and that this initial build is carried out in a reproducible way that does not somehow depend on the local dependency cache of the slave we happen to be building on, for example. In our project, we’ve taken the following steps to help achieve this:

Using clean repositories local to the workspace

We’ve configured the build system to use a clean repository local to the build job’s *workspace*, rather than one that is shared by all builds on that slave. This will ensure the build does not happen to succeed because of an old dependency that is no longer available from your standard repositories, but that happened to be published to that slave’s repo at some point. Consider clearing your build job’s workspace regularly (most [SCM plugins](https://wiki.jenkins-ci.org/display/JENKINS/Plugins#Plugins-Sourcecodemanagement) have a “clean build” option, and for things like partial cleanup the [Workspace Cleanup plugin](https://wiki.jenkins-ci.org/display/JENKINS/Workspace+Cleanup+Plugin) can help), or at least wiping its local repo. For Maven builds, the build repository location can easily be configured via the main Jenkins settings, and overridden per job where necessary.

Using clean slaves based on a known template

We can take this a step further by running our builds on “clean” slaves created on demand and initialized to a known, reproducible state where possible. Plugins such as the [Amazon EC2 plugin](https://wiki.jenkins-ci.org/display/JENKINS/Amazon+EC2+Plugin), [Docker plugin](https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin) or [jclouds plugin](https://wiki.jenkins-ci.org/display/JENKINS/JClouds+Plugin) can be used for this purpose, and there are also hosted services such as CloudBees [DEV@cloud](http://www.cloudbees.com/dev) that provide this functionality. Spinning up build slaves on demand also has the substantial advantage of helping avoid long build queue times if you only have a limited pool of slaves and a growing number of pipeline runs.

Using a central, shared repository for build dependencies

We’re using a centralized artifact repository across all our projects, rather than allowing each project to decide from where to download build dependencies. This ensures that two projects that reference the same dependency will actually get the *same* binary, and allows us to enforce dependency policies (such as banning certain dependencies) in a central location. If you are using a build system that supports Maven-based dependency management, a Maven proxy such as [Nexus](http://www.sonatype.org/nexus/) or [Artifactory](http://www.jfrog.com/home/v_artifactory_opensource_overview) is ideal.

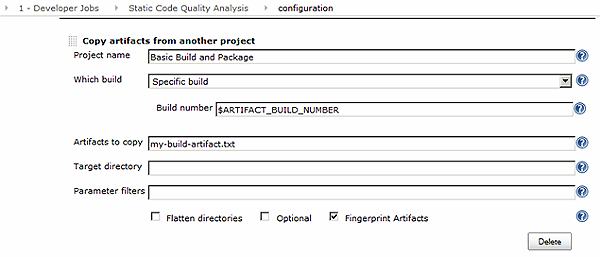
Sharing build artifacts throughout the pipeline

Once we have built our candidate artifact in our initial build job, we need to find a way to ensure exactly *this* artifact is used by all the subsequent builds in our pipeline.

Retrieving build artifacts from upstream jobs

Jenkins provides a couple of ways to share artifacts produced by an upstream job with subsequent downstream jobs. We are using the [Copy Artifact plugin](https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin), which allows us to retrieve build artifacts from another job with a convenient build step. We’re copying from a *fixed* build (i.e. specified by build number or build parameter), which is preferable to referring to a *variable* upstream build (such as the “Last successful build” option). In the latter case, we cannot be sure that we will be referencing the artifacts that triggered *this* pipeline run, rather than those produced by a subsequent commit.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig2-large.png)

**Figure 2: Copying pipeline artifacts using the Copy Artifact plugin**

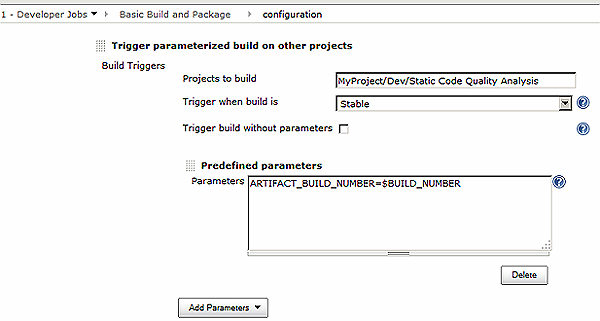
*Alternatives*

* If you also want to access the artifact outside Jenkins, you can save the candidate artifact as a “Build Artifact” of the initial job, then use the Jenkins APIs to download it (e.g. using wget or cURL) in downstream jobs
* If you want to treat candidate artifacts as build dependencies, the [Jenkins Maven Repository Server plugin](https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Maven+Repository+Server) makes build artifacts available via a Maven repo-compliant interface, which can be used by Maven, Gradle, Ant and other build tools to retrieve artifacts. It also provides additional options for referencing artifacts via the SHA1 ID of the Git commit that produced the artifacts (especially useful if the Git commit ID is your unique build identifier), as well as for accessing artifacts of a chain of linked builds.
* If you already maintain a Definitive Software Library *outside* Jenkins, you can create a setup similar to that offered by the Maven Repo Server plugin with an external Maven repo. In that case, you would publish the artifacts to the repo using a Maven identifier that includes the build number, commit ID or whatever is considered a stable identifier or unique ID.

Identifying the correct upstream build throughout the pipeline

Whichever alternative we choose, we need to pass a stable identifier to downstream builds so we can pick the right candidate artifact for our pipeline run. In our pipeline, we have made most of the downstream builds parameterized and are using the [Parameterized Trigger plugin](https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin) to pass the identifier.

**(Click on the image to enlarge it)**

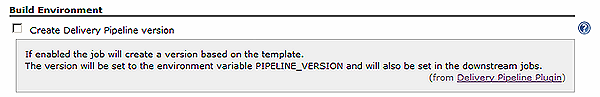
[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig3-large.png)

**Figure 3: Passing the unique pipeline identifier to downstream builds**

*Alternatives*

* We can also use the [Delivery Pipeline plugin](https://wiki.jenkins-ci.org/display/JENKINS/Delivery+Pipeline+Plugin) (which we will also meet later), which optionally creates an environment variable that is available in all downstream jobs.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig4-large.png)

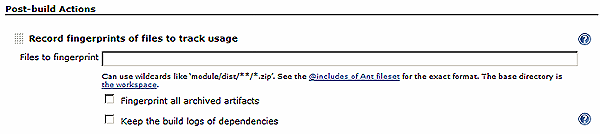
**Figure 4: The pipeline version environment variable option of the Delivery Pipeline plugin**

Using Fingerprints to track artifact usage

However you end up passing the stable pipeline identifier to downstream pipeline phases, setting up all jobs in the pipeline to use [Fingerprints](https://wiki.jenkins-ci.org/display/JENKINS/Fingerprint) is almost always a good idea. Jenkins “fingerprints” artifacts by storing their MD5 checksums, and using these to track use of an artifact across jobs. It allows us to check, at the end of a pipeline run, which artifacts have been used in which builds, and so to verify that our pipeline has indeed consistently been testing and releasing the correct artifact. Jenkins provides a post-build task that allows us to explicitly record fingerprints for files in the workspace. Certain plugins, such as the Copy Artifact plugin, automatically fingerprint artifacts when copying them from an upstream build, in which case we can omit the post-build step

**(Click on the image to enlarge it)**

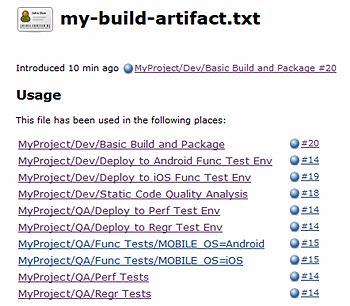
[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig5-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig6-large.png)

**Figure 5: Fingerprinting build artifacts using the Copy Artifact plugin and via a post-build action**

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig7-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig8-large.png)

**Figure 6: Tracking the usage of build artifacts via Fingerprints**

Choosing the right granularity for each job

This may seem a rather obvious point, but choosing the correct granularity for each job, i.e. how to distribute all the steps in our pipeline across multiple jobs, will help us make our pipeline more efficient and allow us to identify bottlenecks more easily. As a rough rule of thumb, every stage in your pipeline can be represented by a separate job or, in the case of multi-dimensional tests, a matrix job. This is why, for instance, we have not combined build and deployment to the test environments, or deployment to the regression test environment and the regression tests themselves, into a single job in our pipeline. If, for instance, we merged *Deploy to Regr Test*and *Regr Test* into one “multi-stage” job and it has failed 10 times recently, we would need to spend time analysing the failures to figure out if the deployment or the tests themselves are the real problem.

The flipside of avoiding “multi-stage” jobs is, of course, that there are more jobs that we need to manage and visualize: 10 already, in our relatively simple example.

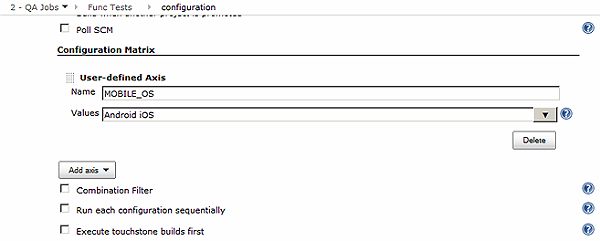
Parallelizing and joining jobs

Especially when running multi-platform tests, but also if we are building artifacts for different target platforms, we want to make our pipeline as efficient as possible by running builds in parallel. In our case, we want to parallelize our functional tests for Android and iOS, as well as running the performance and regression tests in parallel. We’re using a couple of Jenkins mechanisms for this:

Running parallel instances of the same job with different parameters

For the functional tests, which are “variants” of the same build (same steps, but different configuration parameters), we’re using the standard [Multi-Configuration project](https://wiki.jenkins-ci.org/display/JENKINS/Building+a+matrix+project) type (often referred to as a “matrix build”). If we needed to handle potentially spurious failures for some of the matrix builds, we could also add the [Matrix Reloaded plugin](https://wiki.jenkins-ci.org/display/JENKINS/Matrix+Reloaded+Plugin).

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig9-large.png)

**Figure 7: *Func Tests* in our sample pipeline is a multi-configuration (“matrix”) project**

Running different jobs in parallel

For the deployments to the two functional test environments, where we need to run different jobs, we’re using the standard option is simply to use build triggers to kick off multiple downstream jobs in parallel once the upstream job (*Basic Build and Package*, in our case) completes

*Alternatives*

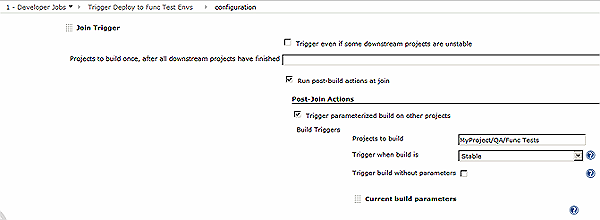
* If you want to coordinate sets of parallel jobs, you might also consider the [Multijob plugin](https://wiki.jenkins-ci.org/display/JENKINS/Multijob+Plugin), which adds a new project type that allows multiple jobs to run in parallel (in fact, it could potentially be used to orchestrate multiple pipeline phases).

Joining parallel sections of the build pipeline

For our matrix job *Func Tests* build, “joining”, i.e. waiting until all the parallel builds have been completed before continuing to the downstream phases *Deploy to Regr Test Env*and *Deploy to Perf Test Env*, is handled automatically by the matrix job type. We have configured *Func Tests* to trigger the downstream builds on success, and these will only be triggered if both the Android and iOS builds in the matrix complete successfully.

For the deployment to the two functional test environments, where we simply trigger multiple jobs to run in parallel, we end up facing the “diamond problem”: how to join the parallel jobs *Deploy to Android Func TestEnv* and *Deploy to iOS Func Test Env*back together to trigger *one* subsequent job, *Func Tests*. Here, we’re using the [Join plugin](https://wiki.jenkins-ci.org/display/JENKINS/Join+Plugin), which we’ve configured in the job “at the top” of the diamond to trigger the job “at the bottom” of the diamond once the parallel deployment jobs have completed successfully. We do not need specify the deployment jobs explicitly – the plugin kicks off the *Func Tests* job once all direct downstream jobs have finished. The Join plugin also supports passing of build parameters, which we need to identify the build artifacts for this pipeline run.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig10-large.png)

**Figure 8: Triggering *Func Tests* in our sample pipeline by using the Join plugin to wait for the direct downstream jobs *Deploy to Android Func Test Env* and *Deploy to iOS Func Test Env* to complete**

Handling more complex job graphs

If you have more complicated “job graphs”, you may also want to have a look at the [Build Flow plugin](https://wiki.jenkins-ci.org/display/JENKINS/Build+Flow+Plugin), which allows you to define job graphs, including parallel sections and joins, programmatically.

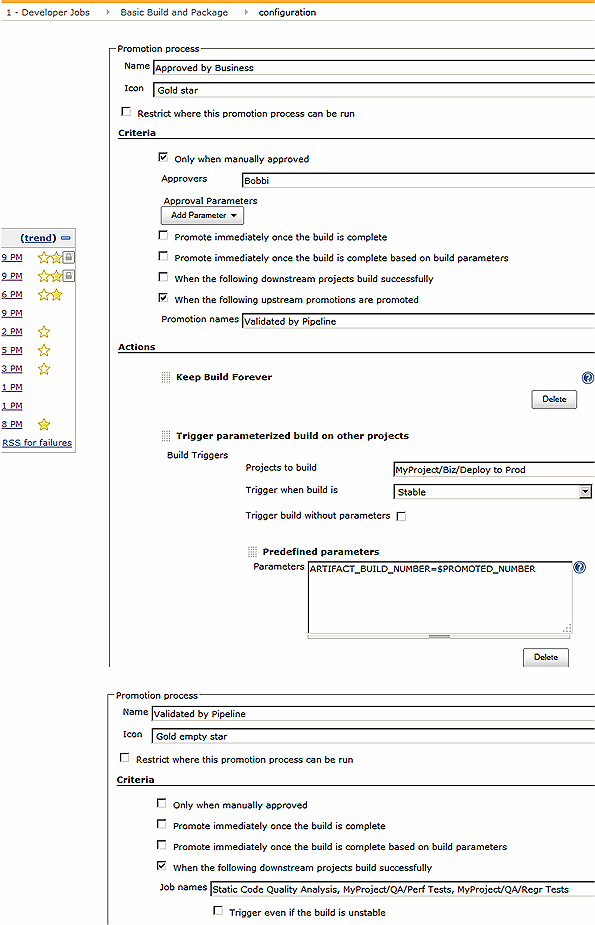
Gates and approvals

As the pipeline stages get closer to the QA and Production environments, many organizations require some form of sign-off or approval before tasks can be carried out. In our case, we require a manual sign-off from the business owner before kicking off the *Deploy to Prod* job, for instance.

As previously noted, Jenkins and other CI tools and generic orchestrators do not offer very comprehensive support for manual pipeline tasks, but there *are* a couple of options to handle approvals.

Supporting approvals based on multiple conditions

We’re using the [Promoted Builds plugin](https://wiki.jenkins-ci.org/display/JENKINS/Promoted+Builds+Plugin), which offers manual approval (and a corresponding email notification to the approver) as one of a number of possible ways to “promote” a build. It also supports a variety of “on promotion” actions, including triggering downstream jobs.



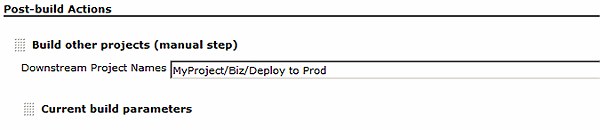
**Figure 9: The *Basic Build and Package*job triggers a production deployment after manual approval by the business owner and confirmation that all downstream jobs have successfully completed**

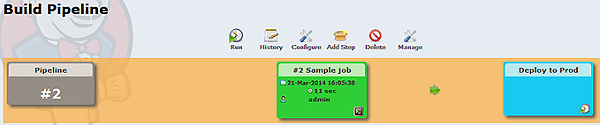
*Alternatives*

* A simple option to create a very basic gate can be just to ensure that the “gated” downstream job is only triggered manually and can only be executed by a limited number of approvers. In this case, the simple fact of triggering a build constitutes approval. This pattern can also be automated, using e.g. the [ScriptTrigger plugin](https://wiki.jenkins-ci.org/display/JENKINS/ScriptTrigger+Plugin) to search for an approval in an external system.  
  However, this breaks the approach of using parameterized triggers to pass on required information, such as the unique artifact ID. If we adopt this pattern, we need to find another way to ensure the appropriate information is passed, e.g. by prompting the approver to enter the appropriate parameters manually, or by having the trigger script retrieve them from the approval record (e.g. a JIRA ticket).
* If you only want to ensure that a task is manually triggered but do not need to track multiple conditions, you might want to look at the [Build Pipeline plugin](https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin), which provides a post-build step to manually execute downstream projects.

This step also allows parameters, such as our build identifier, to be passed to the manually triggered downstream job.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig12-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig13-large.png)

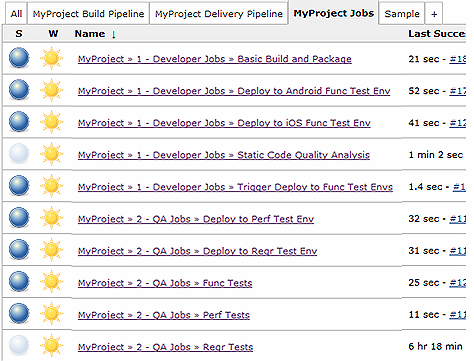
**Figure 10: The Build Pipeline plugin’s post-build step and manual trigger in the pipeline view**

Visualizing the pipeline

Being able to provide a clear, highly accessible visualization of our build pipelines is very important for a successful Continuous Delivery implementation. Not just to ensure the team is always aware of the current pipeline state, but also as a mechanism to communicate to the business and other stakeholders.

Using standard views

Views are standard Jenkins features we are using to collect the jobs that constitute our pipeline in one overview. The Multijob plugin, which we briefly mentioned above, provides a similar “list-style” view. A drawback of both alternatives, however, is that these views show the *currently executing* builds for each job in the pipeline, which may be working on *different* release candidates. For example, the *Perf Tests* and *Regr Tests* jobs may be testing one particular candidate version while the *Basic* *Build and Package* job is already busy with the next commit.

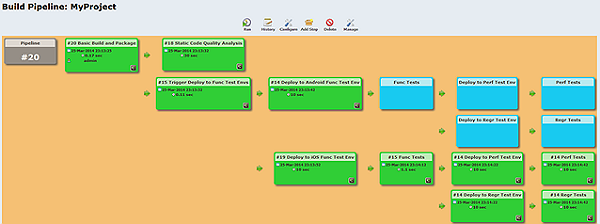


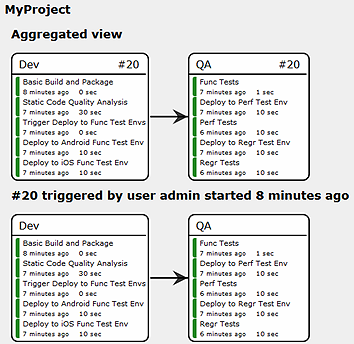
**Figure 11: A standard list view showing active jobs working on *different* release candidates**

Specialized Delivery Pipeline views

From a Continuous Delivery perspective, however, we want to see all the builds that make up a particular *instance* of a pipeline run, i.e. all the builds related to one candidate application version. Two dedicated plugins that support this kind of view are the Build Pipeline plugin and the Delivery Pipeline plugin. Note that both plugins fail to capture the link to the *Deploy to Prod* job, which is not an immediate downstream build, but triggered by the Promoted Builds plugin.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig15-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig16-large.png)

**Figure 12: Build Pipeline and Delivery Pipeline plugin views of our sample pipeline**

Organizing and securing jobs

Handling many jobs

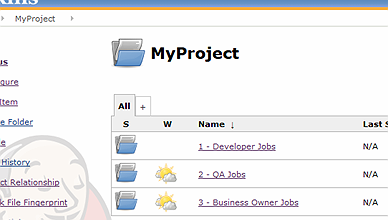
Even if each of our pipelines only consists of a handful of jobs, once we start setting up pipelines for multiple projects or versions, we’ll soon have many Jenkins jobs to manage. In our case, with 10 phases per pipeline, we’d quickly be looking at 100 or more jobs to manage! Creating one or multiple views per pipeline is an obvious approach, but it still leaves us with an incredibly large “All jobs” view in Jenkins – not fun to navigate and manage (in fact, it starts to get so big that you may want to consider [replacing it entirely](https://wiki.jenkins-ci.org/display/JENKINS/Editing+or+Replacing+the+All+View)). It generally also requires us to adopt job naming conventions along the lines of *myProject-myVersion-pipelinePhase*, so that all jobs for a pipeline are listed together, and so that we can use regular expressions when defining views, rather than having to select pipeline jobs for a view individually.

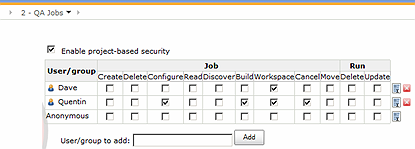
Configuring access control

Where this approach starts to create challenges is when we start to implement access control policies for our pipelines. We need to ensure that different phases of the pipeline have *different* access control policies (in our example, developers are not authorized to trigger the QA jobs or the deployment to production), and setting these policies on each job individually is very maintenance-intensive and error-prone.

In our example, we’re using the [CloudBees Folders plugin](https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Folders+Plugin) in combination with the [Matrix Authorization Strategy plugin](https://wiki.jenkins-ci.org/display/JENKINS/Matrix+Authorization+Strategy+Plugin). The combination allows both for convenient job organization *and* efficient access control configuration. We’ve organized our pipeline jobs in three folders, *MyProject/1 – Developer Jobs*, *My Project/2 – QA Jobs* and *MyProject/3 – Business Owner Jobs*, with each pipeline job in the appropriate folder. Folders are compatible with standard list views, so we can keep our existing *MyProject Jobs* view. Importantly, we can define access control policies at the folder level, which is *much* more convenient than having to secure individual jobs.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig17-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig18-large.png)

**Figure 13: The CloudBees Folders plugin in action, with folder-level security configured using the Matrix Authorization Strategy plugin**

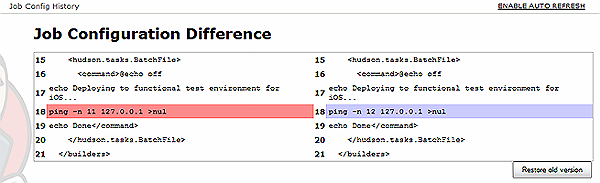
*Alternatives*

* If you want to apply permissions based on the job name, an option you can consider is the [Role Strategy plugin](https://wiki.jenkins-ci.org/display/JENKINS/Role+Strategy+Plugin), which allows you to define different roles that are scoped to different parts of a pipeline. One drawback is that the jobs to which a role definition applies are determined by a regular expression. This can lead to additional complexity in the job naming scheme though (*myProject-myVersion-owningGroup-pipelinePhase*, anyone?), and can be brittle if jobs are renamed.

Good practice: versioning your Jenkins configuration

One additional thing we’re doing in our example, a good Jenkins practice in pretty much all circumstances, is to version out job configurations. This allows us to easily track any changes and revert to earlier configurations, if necessary. We’re using both the [JobConfigHistory plugin](https://wiki.jenkins-ci.org/display/JENKINS/JobConfigHistory+Plugin) (which provides a nice diff view) and [SCM Sync Configuration plugin](https://wiki.jenkins-ci.org/display/JENKINS/SCM+Sync+configuration+plugin) (which stores the configuration off-disk in a repository), but depending on your needs typically one or the other will suffice.

**(Click on the image to enlarge it)**

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig19-large.png)

[](https://cdn.infoq.com/statics_s1_20170510-0410/resource/articles/orch-pipelines-jenkins/en/resources/Fig20-large.png)

**Figure 14: The JobConfigHistory plugin’s diff view and the configuration settings for the SCM Sync Configuration plugin**

Conclusion

Setting up Continuous Delivery pipelines in Jenkins that are secure, efficient, and easy to use and manage can quickly become challenging. We’ve discussed important prerequisites, made a number of recommendations and introduced a set of freely available plugins which can make the process a whole lot easier. Hopefully, you are now in a better position to identify whether Jenkins is the right orchestrator for your current process, to build pipelines painlessly and to quickly start making life better for your teams and delivering business value to your customers!

About the Authors

**Andrew Phillips** is VP of Products for XebiaLabs, providers of application delivery automation solutions. Andrew is a cloud, service delivery and automation expert and has been part of the shift to more automated application delivery platforms.  In his spare time as a developer, he worked on Multiverse, the open-source STM implementation, contributes to Apache jclouds, the leading cloud library and co-maintains the Scala Puzzlers site.

**Kohsuke Kawaguchi** is Cloudblees CTO and the creator of Jenkins. He is a well-respected developer and popular speaker at industry and Jenkins community events. He's often asked to speak about his experience and approach in creating Jenkins; a CI platform that has become a widely adopted and successful community-driven open source project. The principles behind the Jenkins community - extensibility, inclusiveness, low barriers to participation - have been the keys to its success. Kawaguchi's sensibilities in creating Jenkins and his deep understanding of how to translate its capabilities into usable software have also had a major impact on CloudBees' strategy as a company. Before joining CloudBees, Kawaguchi was with Sun Microsystems and Oracle, where he worked on a variety of projects and initiated the open source work that led to Jenkins.

* Personas

* **[DevOps](https://www.infoq.com/Devops)**
* Topics

* [IT Service Management](https://www.infoq.com/ITSM)

* [Continuous Delivery](https://www.infoq.com/Continuous_Delivery)

* [Jenkins](https://www.infoq.com/Jenkins)

* [Automated Deployment](https://www.infoq.com/automated-deployment)

* [Agile Techniques](https://www.infoq.com/agile_techniques)

* [Build systems](https://www.infoq.com/builds)

* [Continuous Integration](https://www.infoq.com/continuous_integration)

* [Agile](https://www.infoq.com/agile)

* [Infrastructure](https://www.infoq.com/Infrastructure)

* [Cloud Computing](https://www.infoq.com/cloud-computing)

**Related Editorial**

[Jenkins Gets a Facelift with Release of Blue Ocean 1.0](http://www.infoq.com/news/2017/04/jenkins-blue-ocean?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk)

[Kohsuke Kawaguchi, Creator of Jenkins Discusses its History, Present and Future](http://www.infoq.com/interviews/kohsuke-kawaguchi?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk)

[Microsoft and Jenkins Partner to Run Project Infrastructure on Azure](http://www.infoq.com/news/2016/05/microsoft-jenkins-azure?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk)

[Serverless Takes DevOps to the Next Level](http://www.infoq.com/articles/serverless-takes-devops-next-level?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk)

[Automated Acceptance Testing Supports Continuous Delivery](http://www.infoq.com/news/2017/04/acceptance-testing-delivery?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk)

**Tell us what you think**

Top of Form



Bottom of Form

**Community comments**[Watch Thread](javascript:void(0))

[**Artifactory Help** by kishore nadendla Posted Dec 26, 2014 09:01](https://www.infoq.com/articles/orch-pipelines-jenkins#anch117752)

[**how to get build failure reason in email content of Jenkins jobs** by Manoj chhablani Posted Aug 13, 2015 02:14](https://www.infoq.com/articles/orch-pipelines-jenkins#anch126590)

[**How to do SVN difference between two revisions using jenkins?** by Manoj chhablani Posted Aug 13, 2015 02:15](https://www.infoq.com/articles/orch-pipelines-jenkins#anch126591)

**Artifactory Help**Dec 26, 2014 09:01 by kishore nadendla

Hi   
  
i would like to start my process from downloading version from artifactory .. so i want to skip build process and static analysis. becuase in my organization that is taken care by different group..  
so in my delivery pipeline i am planning to have 2 stages  
1. CI .. which will contain a) download artifact from artifactory with dependencies b) Deploy using Jenkins to CI server c) run selenium scripts  
  
2. QA (same as 1 steps)  
  
Please let me know how to do with examples