Evolution of Jenkins Jobs and How We Got the YAML-Based Format

This lesson discusses the evolution of Jenkins jobs from the freestyle jobs in traditional Jenkins to jenkins-x.yml jobs in Jenkins X.

We'll cover the following

> ⚠ The examples in this chapter work only with serverless Jenkins X.

The Jenkins X pipeline extension model is, in my opinion, one of the most exciting and innovative improvements we recieved with Jenkins X. It allows us to focus on what really matters in our projects and ignore the steps that are common to others. Understanding the evolution of Jenkins pipelines is vital if we are to adopt the extension model. Before we dive into extensions, we need to understand how pipelines evolved over time.

Freestyle jobs

When Jenkins appeared, its pipelines were called **Freestyle jobs**. There was no way to describe them in code, and they were not kept in version control. We were

creating and maintaining those jobs through Jenkins UI by filling input fields, marking checkboxes, and selecting values from drop-down lists. The results were impossible-to-read `XML` files stored in the Jenkins home directory. Nevertheless, that approach was so great (compared to what existed at the time) that Jenkins become widely adopted overnight.

But, that was many years ago and what was great over a decade ago is not necessarily as good today. As a matter of fact, Freestyle jobs are the *antithesis* of the types of jobs we should be writing today. Tools that create code through drag-and-drop methods are *extinct*. Not having code in version control is a *cardinal sin*. Not being able to use our favorite IDE or code editor is unacceptable. Hence, the Jenkins community created Jenkins pipelines.

Jenkins pipelines

Jenkins pipelines are an attempt to rethink how we define Jenkins jobs. Instead of the click-type-select approach of Freestyle jobs, pipelines are defined in Jenkins-specific **Groovy-based** domain-specific language (DSL) written in `Jenkinsfile` and stored in code repositories together with the rest of the code of our applications. That approach managed to accomplish many improvements when compared to Freestyle jobs.

A single job could define the entire pipeline, the definition could be stored in the code repository, and we could avoid part of the repetition through the usage of Jenkins Shared Libraries. But, for some users, there was a big problem with Jenkins pipelines.

The problem with Jenkins pipelines

Using Groovy-based DSL was too big of a jump for some Jenkins users. Switching from click-type-select with Freestyle jobs into Groovy code was too much. We had a significant number of Jenkins users with years of experience in accomplishing results by clicking and selecting options without knowing how to write code.

I'll leave aside the discussion in which I'd argue that anyone working in the software industry should be able to write code no matter their role. Instead, I'll admit that having a non-coders transition from UI-based into the code-only type of Jenkins jobs posed too large a challenge. Even if we would agree that everyone in the software industry should know how to write code, there is still the issue of Groovy.

Chances are that Groovy is not your preferred programming language. You might be working with **NodeJS**, **Go**, **Python**, **.Net**, **C**, **C++**, **Scala**, or one of the myriads of other languages. Even if you are a **Java** developer, Groovy might not be your favorite. Given that the syntax is somehow similar to Java and the fact that both use JVM does not diminish the fact that Groovy and Java are different languages. Jenkins DSL tried to "hide" Groovy, but that did not remove the fact that you had to know (at least basic) Groovy to write Jenkins pipelines, meaning many had to choose whether to learn Groovy or to switch to something else. So, even though Jenkins pipelines were a significant improvement over Freestyle jobs, there was still work to be done to make them useful to everyone no matter their language preference. Hence, the community came up with a declarative pipeline.

Declarative pipelines

> 🔍 Declarative pipelines are not really declarative. No matter the format, pipelines are by their nature sequential.

The **Declarative pipeline** format is a simplified way to write `Jenkinsfile` definitions. To distinguish one from the other, we call the older pipeline syntax *scripted*, and the newer declarative pipeline. We got much-needed simplicity; there was no Groovy to learn unless we employed shared libraries, the new pipeline format was simple to learn and easy to write, it served us well and was adopted by static Jenkins X. And yet, serverless Jenkins X introduced another change to the format of pipeline definitions. Today, we can think of static Jenkins X with declarative pipelines as a transition towards serverless Jenkins X.

Pipelines defined in `YAML`

With serverless Jenkins X, we moved into pipelines defined in `YAML`. *Why did we do that?*

- One argument could be that `YAML` is easier to understand and manage.
- Another could claim that `YAML` is the golden-standard for any type of definition, Kubernetes resources being one example.

Most of the other tools, especially newer ones, switched to `YAML` definitions. While

Most of the other tools, especially newer ones, switched to `YAML` definitions. While those and many other explanations are valid and certainly played a role in making the decision to switch to `YAML`, I believe that we should look at the change from a different angle.

Why were `XML` and Groovy DSL not suitable for Jenkins?

All Jenkins formats for defining pipelines were based on the fact that it will be Jenkins would execute them. Freestyle jobs used `XML` because Jenkins uses `XML` to store all sorts of information. A long time ago, when Jenkins was created, `XML` was all the rage. Scripted pipelines use Groovy DSL because pipelines need to interact with Jenkins. Since it is written in Java, Groovy was a natural choice. It is more dynamic than Java, it compiles at runtime allowing us to use it as a scripting mechanism. It can access Jenkins libraries written in Java, and it can access Jenkins itself at runtime. Then we added declarative pipeline that is something between Groovy DSL and `YAML`. It is a wrapper around the scripted pipeline.

What all those formats have in common is that they are all limited by Jenkins' architecture. And now, with serverless Jenkins X, there is no Jenkins any more.

What is left of traditional Jenkins in Jenkins X?

Saying that Jenkins is gone is not entirely correct. Jenkins lives in Jenkins X. The foundation that served us well is there. The experience from many years of being the leading CI/CD platform was combined with the need to solve challenges that did not exist before. We had to come up with a platform that is:

- **Kubernetes-first**,
- **cloud-native**,
- **fault-tolerant**,
- **highly-available**,
- **lightweight**,
- with **API-first design**,
- and so on and so forth.

The result is Jenkins X or, to be more precise, its serverless flavor. It combines some of the best tools on the market with custom code written specifically for Jenkins X resulting in what we call serverless or next-generation Jenkins X.

The first generation of Jenkins X (static flavor) reduced the "traditional" Jenkins to a bare minimum. That allowed the community to focus on building all the new tools needed to bring Jenkins to the next level. It reduced Jenkins' surface and added a lot of new code around it. At the same time, static Jenkins X maintains compatibility with the "traditional" Jenkins. Teams can move to Jenkins X without having to rewrite everything they had before while, at the same time, keeping some level of familiarity.

Serverless Jenkins X is the next stage in the evolution. While static flavor reduced the role of the "traditional" Jenkins, serverless eradicated it. The end result is a combination of Prow, Jenkins Pipeline Operator, Tekton, and quite a few other tools and processes. Some of them (e.g., Prow, Tekton) are open-source projects bundled into Jenkins X while others (e.g., Jenkins Pipeline Operator) are written from scratch. On top of those, we have `jx` as the CLI that allows us to control any aspect of Jenkins X.

Given that there is no "traditional" Jenkins in the serverless flavor of Jenkins X, there is no need to stick with the old formats to define pipelines. Those that do need to continue using `Jenkinsfile` can do so by using static Jenkins X. Those who want to get the most benefit from the new platform will appreciate the benefits of the new `YAML`-based format defined in `jenkins-x.yml`. More often than not, organizations will combine both. There are use cases when static Jenkins with the support for `Jenkinsfile` is a good choice, especially in cases when projects already have pipelines running in the "traditional" Jenkins. On the other hand, new projects can be created directly in serverless Jenkins X and use `jenkins-x.yml` to define pipelines.

When would we prefer Jenkins or static Jenkins X over serverless Jenkins X?

Unless you just started a new company, there are all sorts of situations, and some of them might be better fulfilled with static Jenkins X and `Jenkinsfile`, others with serverless Jenkins X and `jenkins-x.yml`. There will be projects that were started a long time ago and do not see enough benefit to change. Those can stay in "traditional" Jenkins running outside Kubernetes or in any other tool they might be using.

A recap and the crux of the discussion

To summarize, static Jenkins is a transition between the "traditional" Jenkins and

the serverless flavor. The former has reduced Jenkins to a bare minimum, while

the latter consists of entirely new code written specifically to solve problems we face today while leveraging all the latest and greatest technology.

So, Jenkins served us well, and it will continue to live for a long time since many applications were written a while ago and may not be good candidates to embrace Kubernetes. Static Jenkins X is for all those who want to transition to Kubernetes without losing all their investment (e.g., `Jenkinsfile`). Serverless Jenkins X is for all those who seek the full power of Kubernetes and want to be genuinely cloud-native.

> 🔍 As of now, serverless Jenkins X works only with GitHub. Until that is corrected, using any other Git platform might be yet another reason to stick with static Jenkins X. The rest of the text will assume that you do use GitHub or that you can wait for a while longer until the support for other Git platforms is added.

Long story short, Serverless Jenkins X uses `YAML` in `jenkins-x.yml` to describe pipelines, while more traditional Jenkins, as well as static Jenkins X, relies on Groovy DSL defined in `Jenkinsfile`. Freestyle jobs are deprecated for quite a long time, so we'll ignore their existence. Whether you prefer `Jenkinsfile` or the `jenkins-x.yml` format will depend on quite a few factors, so let's break down those that matter the most.

If you already use Jenkins, you are likely used to `Jenkinsfile` and might want to keep it for a while longer.

If you have complex pipelines, you might want to stick with the scripted pipeline in `Jenkinsfile`. That being said, I do not believe that anyone should have complex pipelines. Those that do usually tried to solve problems in the wrong place. Pipelines (of any kind) are not supposed to have complex logic. Instead, they should define orchestration of automated tasks defined somewhere else. For example, instead of having tens or hundreds of lines of pipeline code that defines how to deploy our application, we should move that logic into a script and simply invoke it from the pipeline. That logic is similar to what `jx promote` does. It performs semi-complex logic, but from the pipeline point of view it is a simple step

with a single command. If we do adopt that approach, there is no need for complex

pipelines and, therefore, there is no need for Jenkins' scripted pipeline. Declarative is more than enough when using `Jenkinsfile`.

If you do want to leverage all the latest and greatest that Jenkins X (serverless flavor) brings, you should switch to the `YAML` format defined in the `jenkins-x.yml` file.

Therefore, use `Jenkinsfile` with static Jenkins X if you already have pipelines and you do not want to rewrite them. Stop using scripted pipelines as an excuse for misplaced complexity. Adopt serverless Jenkins X with `YAML`-based format for all other cases.

But you probably know all that by now. You might even be wondering why we went through that history lesson. We reminisced because I want to talk about code repetition next, and I had to set the scene for what's coming.