CloudBees®

eBook

A guide to streamlining CI with declarative pipelines

Automating as many repetitive tasks as possible has always been an ideal within software development, and a variety of tools are available to help with this realization.

At the center of this automation is continuous integration and continuous delivery (CI/CD), the ongoing, cyclical processes by which a software team writes, tests, and delivers code to customers.

Pipelines further help DevOps define what they want a CI/CD process to accomplish and the tasks needed to achieve that goal in a streamlined manner—a crucial element of modern software delivery.

For example, to test and build recent changes made to code, the basic steps might be as follows:

1. Check the repository
2. Run automated tests

- If the tests pass successfully, build the code and notify interested parties with medium priority.
- If the tests fail, notify interested parties with high priority.

Traditionally, you may have used the scripted pipeline approach in Jenkins®, which offers a fully featured programming language (Groovy for Jenkins CI) inside the Jenkinsfile. This approach offers a lot of flexibility, but with it comes a need for developers to write and maintain pipelines as well as several other associated problems, such as incompatibility, security and performance concerns, and difficulties sharing steps—many of which can be alleviated with a newer approach: declarative pipelines.

This eBook will explore how to get started with declarative pipelines and the technical and business benefits of doing so.

Bring standards and flexibility to your teams

The flexibility and power of a scripted pipeline may initially feel like an advantage, but the lack of standardized syntax can lead to a complex web of logic and brittleness that becomes difficult to maintain over time.

The development and maintenance of pipelines often end up in the hands of a few knowledgeable developers. If they switch teams or leave your company, that knowledge is lost, and someone else has to try and understand how they are written and work. The standardized format of declarative pipelines results in a predictable syntax, which still requires some learning, but there are common patterns to support that.

Scripted pipelines are also often tied to using the Jenkins UI to generate them. This is a great starting point for people to create pipelines, but it isn't expressive enough to offer features for complex pipelines, such as logic and running steps on multiple agents or in parallel—plus, they're hard to audit.

“The standardized format of declarative pipelines results in a predictable syntax, which still requires some learning.

Many scripted pipelines make use of shared libraries and other dependencies. Not only are these hard to maintain in their own right, but they can introduce further instability into pipelines as well as the CI/CD servers that run the pipelines.

On the other hand, declarative pipelines use pre-defined syntax. They offer the same level of flexibility as scripted pipelines, but with increased reliability and stability. Using this approach makes it easier for other team members to write pipelines, and you can manage them directly with version control. Most version control systems have a core feature that allows you to see who made what change and when, giving you a near-perfect audit log.

The building blocks of a declarative pipeline

Let's dive in further with an example.

To replicate the test process mentioned earlier, you'll first need to wrap everything in a pipeline block and then inside of that, an agent block, which specifies where the tasks run. It can include environment variables, specific nodes, and values from Docker or Kubernetes. You can also further specify an agent at each stage. In this case, the tasks run on any available agent:

```
pipeline {
   agent any
}
```

Next, a declarative pipeline must have exactly one stages block with at least one stage (up to as many as you need) inside of it. A stage represents logical groupings of tasks—for example: build, test, and deploy tasks.

```
pipeline {
    agent any
    stages {
        stage('Check out code') {
        }
   stage('Run tests') {
    }
  }
}
```

Then, each stage block must have exactly one steps block, and inside those are where you can add functional code for the pipeline, which typically runs in a shell environment.

```
pipeline {
    agent any
    stages {
        stage('Check out code') {
        steps {
        // Check out branch and repository
        // Send notification of completion
        }
    }
    stage('Run tests') {
    steps {
    // Run tests // Notify of results
    }
   }
  }
 }
```

You can find a full command reference for declarative pipeline syntax in our documentation.

Adding logic to declarative pipelines

You're likely thinking that the code above doesn't fully represent the logic from the earlier example, as it has conditional logic in it. But declarative pipelines can handle logic, too. There are two ways to handle logic directly with pipelines, but you can also have logic embedded in the commands to handle more nuanced tasks.

The first option is when, which you can add inside a stage and determines when to run that stage based on a repository branch, environment variable, or Groovy expression. For this example, you could set an environment variable with the test state, or use Groovy to check.

```
pipeline {
    agent any
    stages {
…
        stage('Run tests') {
        steps {
        // Run tests // Notify of results
        }
    }
    stage('Tests successful') {
    when {
    // Evaluate condition
    }
    stages {
        stage('Build code') {
        steps {
        // Build code
        }
    }
```

```
stage('Notify of successful build') {
    steps {
    // Notify
    }
   }
  }
 }
    // Stage for when tests fail
    }
  }
```

Each stage can have substages within them, too, allowing for complex multistep tasks.

You can also use post, which defines actions to run at the end of a pipeline depending on the resulting end state. For example:

```
pipeline {
    agent any
    stages {
        stage('Run tests') {
…
        }
    }
    post {
        failure {
        // Tests failed
    }
    success {
    // Tests succeeded
    }
  }
}
```

Although the syntax helps declarative pipelines from becoming too unwieldy to understand how they function, you should always keep your main focus on making them easy to maintain. You can keep pipelines simpler by breaking out as many steps as possible to external scripts and specialized command-line tools. If you need to work with external application programming interfaces, then you can also use fully fledged programming languages—again, moving that functionality into external files where you can also manage version control.

4 best practices to optimize your declarative pipelines

So, we've covered how declarative pipelines offer a flexible way to create CI/CD pipelines while keeping configuration and logic in a consistent and reliable format. However, to fully realize their potential, you'll want to follow these four best practices to ensure a streamlined workflow across users and teams.

1. Make smart use of versioning

Versioning offers an audit trail to see who contributed what and when to the configuration.

Using version control to manage your pipeline configurations makes it easier for team members to collaborate when creating the tasks they need. Teams and individual members can contribute to the parts of the pipeline they are responsible for with minimal impact on others. As team members propose changes, they can see them in the context of the existing configuration and also test the impact of the proposed changes by using branches in the repository.

Versioning offers an audit trail to see who contributed what and when to the configuration. This is useful for identifying issues and changes in performance, but also for regulatory reasons. If a team member or customer notices a problem, versioning means you can roll back changes to previous states and trigger CI/CD processes again based on the changed state. This rollback can happen quickly, restoring systems to a previous state in minutes.

Most version control providers integrate with hundreds of other tools, meaning that you can connect your pipeline configuration to many other services and processes, triggering other steps during pipeline creation or when it's running. This can help keep pipelines cleaner and more relevant to test and build tasks, and nothing else. For example, CI/CD pipelines might not be where you want to run code quality checks, and you could run them before triggering CI/CD pipelines.

2. Focus on reproducible templates

An increasing number of companies now focus on internal developer experience and productivity to retain developers and help them become as efficient as possible. This has led to many following the concept of "platform teams" who manage the infrastructure and processes development teams use. Increasingly, they manage the "X as code" artifacts used to create the infrastructure and processes.

Declarative pipelines help you create reproducible templates for pipelines that you can break into components to share within teams and companies. You can use templates to enforce certain standards, provide access to resources, and allow developers to further customize certain parts of the selected template.

For example, you could create one template for a development team to access shared test infrastructure that gives a lot of leeway and flexibility, while another template for production applications has more restrictions and fewer customization options.

Many companies with large development teams now publish these templates in something like a catalog, allowing developers to select what they're looking for and build on top of it. A catalog is especially useful for onboarding. Instead of new hires needing to search across disparate knowledge silos to find what they're looking for, they can browse a single location. This means they onboard quicker, need to ask fewer questions of fewer people, and will possibly discover helpful resources they didn't even know they were looking for.

3. Allow for dynamic inputs

Complementing templates, declarative pipelines can capture input as a pipeline runs, allowing for dynamic variables during the process.

For example, a pipeline could ask which branch(es) to pull code from, overriding default values. Or, it could ask which environment to deploy code to. In order to enhance security and stability, a pipeline could prompt a user to confirm they have read certain guidelines and, using declarative pipeline logic, trigger different stages depending on the input value. This best practice overlaps with the previous point, as allowing for variable input adds reproducibility, too.

4. Take advantage of parallelization

Developers can spend a lot of time waiting for automated processes to complete, blocking them from moving on to other tasks or requiring them to spend too much time "context switching" between tasks. Declarative pipelines offer two pre-defined ways to take full advantage of CI/CD to run multiple tasks in parallel, and repeatedly.

You can define a parallel section inside a stage that defines multiple substages to run simultaneously—for example, to run similar tasks across multiple branches in a repository:

```
…
stages {
    stage('Parallel branch stages')
    {
    parallel {
         stage('main branch') {
         // Steps
     }
         stage('Dev branch') {
         // Steps
     }
…
```

Another option is the matrix feature, which allows for the specification of a list of stages you want to run in parallel on multiple configurations. Running tests on multiple browsers across multiple platforms is a perfect use case.

To do so, you would add a matrix block into a stage that defines the cells, followed by the stages to run in each cell. For example, to represent the use case mentioned previously:

```
pipeline {
    agent none
    stages {
        stage("Browser tests") {
            matrix { axes { axis { name
           'OS_VALUE' values "linux",
           "windows", "mac" } axis { name
          'BROWSER_VALUE' values "firefox",
           "chrome", "safari", "ie"
           }
        }
     stages {
         stage('Run browser tests') {
            // Steps
}
…
```

Each axis must have a unique name, with the values defining the "cells" of each axis. The example above creates a 3x4 matrix, with each of the four browsers for each of the three operating systems, running the stages a total of 12 times.

A note on converting freestyle projects

While declarative pipelines offer many advantages, a lot of work went into creating scripted pipelines, and many are still used in production setups. To help with the process, you can use the CloudBees Declarative Pipeline Migration Assistant.

The Declarative Pipeline Migration Assistant plugin is a tool to convert scripted pipelines into declarative pipelines. This plugin creates a Jenkinsfile that includes predictable configurations, structure, policies, rules, and conditions for supported plugins, with placeholder stages for plugins not yet supported. This plugin helps alleviate the tedious manual effort of translating pipelines, which is prone to human error and time-consuming to troubleshoot.

“Declarative pipelines offer complexity, but as the syntax is strict, that complexity remains easier to understand and maintain.

Streamline your CI with control and speed

This eBook covered the basic concepts and building blocks of declarative pipelines, and some best practices to represent your CI/CD processes with them. We looked at how to best integrate declarative pipelines into your team workflows to increase flexibility and reproducibility, thus saving developers time on repetitive admin tasks that can take away from innovation.

In summary, the consistency and stability offered by the syntax of declarative pipelines position the approach firmly in the center of modern software delivery practices. This evolution of CI/CD allows development teams to have more control over releases and integrates well with the use of feature flags to increase speed and decrease deployment risk.

Combined with an enterprise CI solution, developers can easily create, maintain, and even automate multiple pipelines; rapidly onboard new projects and teams with preconfigured templates and best practices; and ensure security, governance, and compliance across teams at scale.

With declarative pipelines, instead of fixing behind-the-scenes infrastructure issues, your developers can focus on delivering more meaningful value to customers. It's a win-win for your employees and your business.

Learn how CloudBees Continuous Integration provides flexible, scalable, and governed CI/CD you can trust.

CloudBees®

CloudBees, Inc., 4 North 2nd Street, Suite 1270 San José, CA 95113 United States
www.cloudbees.com • info@cloudbees.com