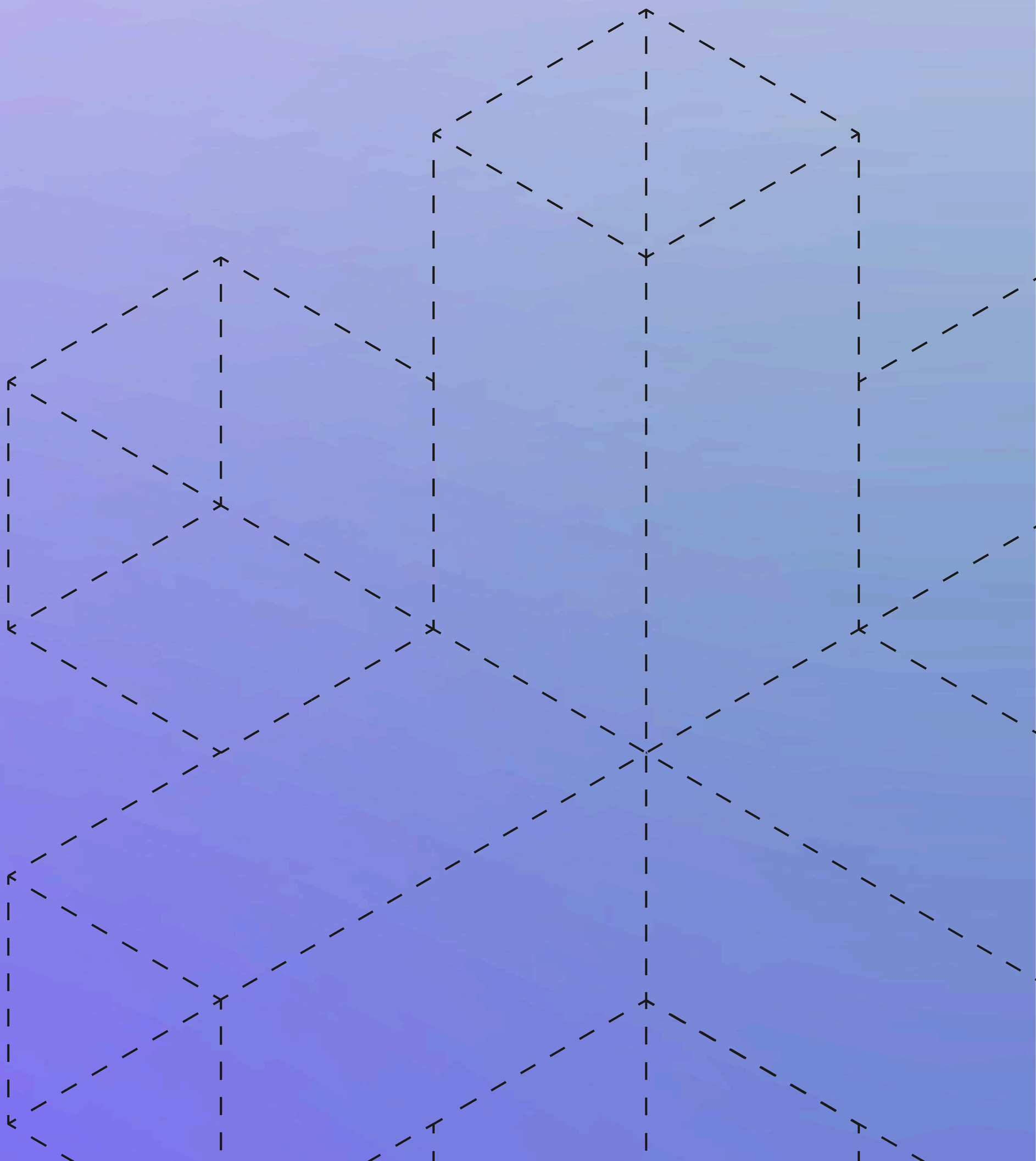
eBook

The Guide to Advanced Jenkins

With over a decade in the DevOps Space, CloudBees is the leading contributor of code commits for Jenkins.



CloudBees

Table of Contents

Introduction 3

Use Just Enough Pipeline 4

Just Say No to Pipelines in Programming Languages 5

Use Just Enough Pipeline to Keep Your Pipeline Scalable 6

Use Declarative Pipeline 8

Avoid Creating a Monolithic Controller 10

Don’t Let Your Plugins Manage You 12

Consider Your Plugin Management Strategy 12

Native Plugin Management Methods for Jenkins 14

Advanced Plugin Management Methods in CloudBees CI 15

Use Containers for Build Agents 17

Flexible, Governed, and Secure CI at Scale 18

Introduction

Jenkins®, is arguably one of the most popular development tools on the planet, with the active Jenkins installations growing year over year. In fact, the number if jobs run on Jenkins has risen over 45% from June 2021 to June 2023! It's great at helping small, agile development teams build, test, and deploy multiple times a day.

However, as teams, environments, projects and market pressures increase, so do the administrative tasks associated with maintaining and administering Jenkins. Scaling Jenkins usage across a growing enterprise, without affecting the security or stability of its software development lifecycle (SDLC) is challenging, particularly for highly regulated industries.

CloudBees is the leading contributor of code commits for Jenkins and with over a decade of field experience in the DevOps space, we understand the inner workings of Jenkins and most importantly, how Jenkins is used by enterprises to deliver and create exceptional software. Based on this experience, we recommend the following best practices for setting up efficient, secure Jenkins pipelines.

Use Just Enough Pipeline

Jenkins Pipeline (or simply Pipeline with a capital P) is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins. This allows you to automate the SDLC and deliver important changes more efficiently to your users and customers.

Pipeline code works beautifully for its intended role of automating build, test, deploy, and administration tasks. But, as it is pressed into more complex roles and unexpected uses, some users have run into snags.

Using best practices – and avoiding common mistakes – can help you design a pipeline that is more robust, scalable, and high-performing.

We see a lot of customers making basic mistakes that can sabotage their pipeline. (Yes, you can sabotage yourself when you're creating a pipeline.) In fact, it's easy to spot someone who is going down this dangerous path – and it's usually because they don't understand some key technical concepts about Pipeline. This invariably leads to scalability mistakes that you'll pay dearly for down the line.

Just Say No to Pipelines in Programming Languages

Perhaps the biggest misstep people make is deciding that they need to write their entire pipeline in a programming language. After all, Pipeline is a domain specific language (DSL). However, that does not mean that it is a general-purpose programming language.

If you treat the DSL as a general-purpose programming language, you are making a serious architectural blunder by doing the wrong work in the wrong place. Remember that the core of Pipeline code runs on the controller.

So, you should be mindful that everything you express in the Pipeline domain specific language (DSL) will compete with every other Jenkins job running on the controller.

For example, it's easy to include a lot of conditionals, flow control logic, and requests using scripted syntax in the pipeline job. Experience tells us this is not a good idea and can result in serious damage to pipeline performance. We've actually seen organizations with poorly written Pipeline jobs bring a controller to its knees, while only running a few concurrent builds.

Wait a minute, you might ask, "Isn't the controller supposed to handle code?" Yes, the controller certainly is there to execute pipelines. But it's much better to assign individual steps of the pipeline to command line calls that execute on an agent. So, instead of running a lot of conditionals inside the pipeline DSL, it's better to put those conditionals inside a shell script or batch file and call that script from the pipeline.

However, this begs another question: “What if I don’t have any agents connected to my controller?” If this is the case, then you’ve just made another bad mistake in scaling Jenkins pipelines. Why? Because the first rule of building an effective pipeline is to make sure you use agents. If you’re using a Jenkins controller and haven’t defined any agents, then your first step should be to define at least one agent and use that agent instead of executing on the controller.

For the sake of maintaining scalability in your pipeline, the general rule is to avoid processing any workload on your controller. If you’re running Jenkins jobs on the controller, you are sacrificing controller performance. So, try to avoid using Jenkins controller capacity for things that should be passed off to an agent. Then, as you grow and develop, all of your work should be running agents. This is why we always recommend setting the number of executors on the controller to zero.

Use Just Enough Pipeline to Keep Your Pipeline Scalable

All of this serves to highlight our overarching theme of “using just enough pipeline.” Simply put, you want to use enough code to connect the pipeline steps and integrate tools – but no more than that. Limit the amount of complex logic embedded in the Pipeline itself (similarly to a shell script), and avoid treating it as a general-purpose programming language. This makes the pipeline easier to maintain, protects against bugs, and reduces the load on controllers.

Another best practice for keeping your pipeline lean, fast, and scalable is to use declarative syntax instead of scripted syntax for your Pipeline. Declarative naturally leads you away from the kinds of mistakes we've just described.

It is a simpler expression of code and an easier way to define your job. It's computed at the startup of the pipeline instead of executing continually during the pipeline.

Therefore, when creating a pipeline, start with declarative, and keep it simple for as long as possible. Anytime a script block shows up inside of a declarative pipeline, you should extract that block and put it in a shared library step. That way, the declarative pipeline is still clean. Joining together the declarative and the shared library will take care of the vast majority of use cases you'll experience.

That being said, you cannot assume that declarative plus a shared library will solve every problem. There are cases where scripted is the right solution. However, declarative is a great starting point until you discover that you absolutely must use scripted. Just remember, at the end of the day, you'll do well to follow the adage: "Use just enough pipeline and no more."

Use Declarative Pipeline

Maintaining multiple configurations of larger, more complex pipelines is another pain point commonly experienced by enterprise Jenkins users. Declarative Pipelines provide a more modern, opinionated approach to building software.

The resulting Jenkins file can then be committed to a Git repo in a “pipeline as code” fashion. It is the recommended way of building Pipelines at scale as it makes Pipeline easier to manage.

A declarative Pipeline provides a structured hierarchical syntax to simplify the creation of Pipelines and the associated Jenkins files. In its simplest form, a Pipeline runs on an agent and contains stages, while each stage contains steps that define specific actions. Here is an example:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn install'
            }
        }
    }
}
```

Figure 1: Declarative Pipeline Stages

Let’s define a few terms before we go any further.

agent	An agent section specifies where tasks in a Pipeline run. An agent must be defined at the top level inside the pipeline block to define the default agent for all stages in the Pipeline. An agent section may optionally be specified at the stage level, overriding the default for this stage and any child stages. In this example, the agent is specified with any, meaning this Pipeline will run on any available agent.
stages	A stage represents a logical grouping of tasks in the Pipeline. Each stage may contain a steps section with steps to be executed, or stages to be executed sequentially, in parallel, or expanded into a parallel matrix. It contains commands that run processes like Build, Deploy, or Tests. These commands are included in the steps section in the stage. It is advisable to have descriptive names for a stage as these names are displayed in the UI and logs. There can be one or more stage sections inside a stages section. At least one stage must be defined in the stages section.
steps	The `steps` section contains a set of commands. Each command performs a specific action and is executed one-by-one.

Figure 2: Declarative Pipeline Terms

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                sh 'mvn compile'
            }
        }
    }
}
```

Figure 3: The ‘Steps’ Section

Avoid Creating a Monolithic Controller

CloudBees has combined a decade of field data and found that any controller that houses more than 5,000 jobs has experienced, or will likely experience, performance and stability issues.

Now, you might say this number is grossly underestimated, and in your case, it may be true. For example, 5,000 “Hello World” jobs are going to perform completely differently than 5,000 complex pipelines with multiple stages, making different types of calls to binaries on an agent.

CloudBees has landed on the 5,000 number as a rough estimate to account for both extremes just cited.

Coupled with field experience, data gathered from our performance team (and shared later in this document) shows that customers who experience performance and stability issues are likely running controllers with more than 5,000 jobs.

The number one question Jenkins administrators have when broaching this subject is, “How many jobs can I run on my controller?” Unfortunately, there is not a simple answer.

The number of variables is too great. The pipeline complexity variable alone is enough to sway a guesstimate into too many unknowns. We do, however, recommend using the 5,000 jobs figure as a high-water mark to help you understand when it is time to horizontally scale.

Employing a monitoring solution to monitor macro metrics (CPU/RAM/DISK I/O) and creating baselines will allow you to properly plan for scaling.

Micro-metrics such as garbage collection logs, object creation rate, and thread counts can be analyzed and baselined so you understand the needs of your jobs and properly plan for additional controllers as needed in your installation.

Don’t Let Your Plugins Manage You

The large ecosystem of plugins that extend the functionality of Jenkins is a critical factor in its popularity and success. Despite all of the benefits that plugins bring to Jenkins, they also bring with them a host of problems. The management of plugins within a given Jenkins controller—and especially across a fleet of Jenkins controllers—has security, stability, and other implications that are compounded at scale.

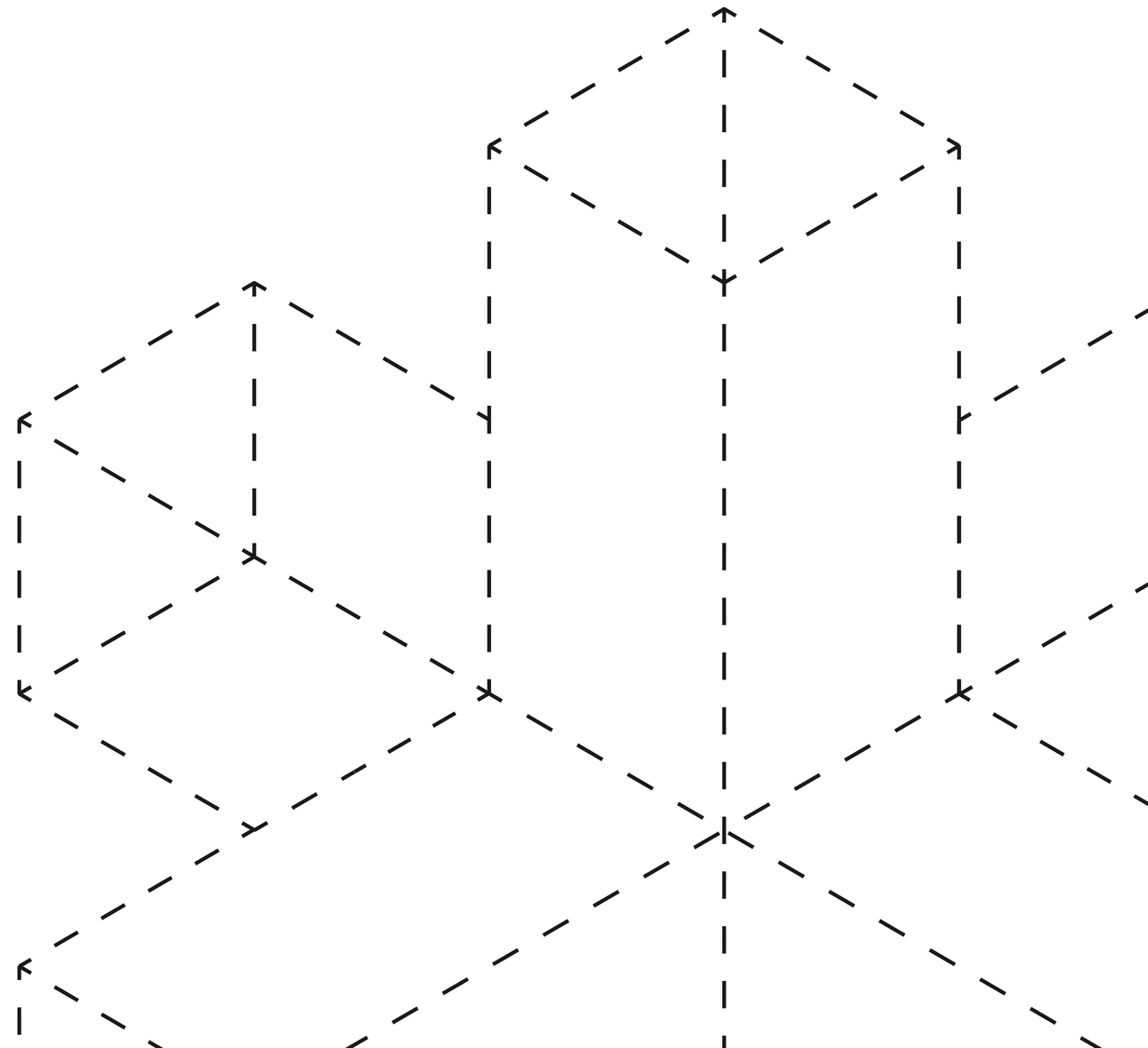
Consider Your Plugin Management Strategy

Jenkins administrators can go about managing their plugins in many different ways. To determine the best plugin management strategy for your organization, consider the following questions.

What is your operational model? For example, do you...

- Have centralized management for all plugin-related changes on all controllers?
- Default to one-time plugin deployment for all controllers, then self-managed afterwards?
- Allow team/project administrators of each controller to manage and install plugins?

1	Will controllers be restricted on the list of plugins available to them?
2	Do you prefer the definition of plugin list to be in code or via UI?
3	Is the installation of plugins per controller optional or mandatory?
4	What are the network restrictions and/or corporate policy restrictions on plugins?
5	Do you have any internally developed plugins?
6	What type of controllers will be provisioned?



CloudBees

Native Plugin Management Methods for Jenkins

Options	Description
The Manage Plugins Page Available to all Jenkins administrators. Plugins can be installed, uninstalled, and updated via the UI. The Advanced tab lets administrators install custom plugins, or plugins that are not available from the update center or the plugin catalog.	This is the simplest and most common way of installing plugins.
Commands with Jenkins CLI Use existing commands that can perform plugin installations, enable, disable, and list plugins. This CLI command can be performed by a script and will have to be triggered on each controller.	From the CLI, plugins can be installed by name (via update center) and by a link to a plugin (via URL or file).
Jenkins Configuration as Code (JCasC) for Controllers	JCasC simplifies the management of a Jenkins installation by capturing the configuration of Jenkins controllers in human-readable declarative configuration files that can then be applied to a controller in a reproducible way. Plugins are managed using two configuration files, plugins.yaml and plugin-catalog.yaml. Plugins are installed automatically upon boot or restart of a controller, and upon a hot reload on the controller. Plugins listed in the JCasC bundle are not optional; they are installed automatically every time.

For smaller organizations, the above methods may cover their plugin management needs. Larger organizations, particularly those in highly regulated industries such as banking, healthcare or insurance, require more robust team management capabilities that tie-in to their RBAC strategy and ensure ongoing performance and security of plug-ins used. F

or those organizations, more advanced plug-in management methods are available by using CloudBees CI, as indicated in the chart below.

Advanced Plugin Management Methods in CloudBees CI

Options	Description
Cluster Operations Perform maintenance operations on various items in the operations center, such as client controllers and update centers. Different operations are applicable to various items such as performing backups or restarts on client controllers, or upgrade/ install plugins in update centers, etc.	Allows an administrator to upgrade, install, enable, or disable plugins on one or more controllers at the same time. This makes it easy to affect multiple controllers and enforce plugin changes across the whole environment.

Team Controller Recipes

These contain plugins and plugin Catalog as configurable parameters. The defined plugins are installed when team controllers are provisioned.

NOTE: Team controller recipes are not compatible with Configuration as Code (CasC) for Controllers. If you try to use both, only the CasC configuration is applied. enable, disable, and list plugins. This CLI command can be performed by a script and will have to be triggered on each controller.

Team controller recipes apply only to environments that use team controllers. A recipe provides a base default team controller installation that teams can build upon.

CloudBees Assurance Program (CAP)

This program, which is designed and maintained by CloudBees, tests plugins, plugin versions, and plugin dependencies to determine their stability. The Beeker Upgrade Assistant in CAP provides a centralized view of the monitored Jenkins plugins, recommended actions, and configuration options.

As software organizations grow, so do the number of plugins that are required to support Jenkins pipelines and application development teams. A Jenkins administrator can end up with hundreds of plugins to manage. Without visibility into which plugins are being used and not used, unused plugins are oftentimes left installed to prevent disruption to the entire system. Over time, this leads to plugin bloat, which can negatively impact system stability and whether or not upgrades are performed.

Use Containers for Build Agents

A continuous integration environment is a mixed bag of machines, platforms, build toolchains, and operating systems. You need the utmost flexibility to manage these machines and build them to be interchangeable. In general, you don’t want to tie builds to a specific build machine.

Make use of container images and Dockerfiles. You won’t have to worry about configuring and managing tool installers or setting up build agent images. With Declarative Pipelines, you can specify that the build runs in a specific container image, or you can have your build environment expressed as a Dockerfile that is stored in source control. You can also use the same behavior at a per-stage level rather than across the entire pipeline.

Developers Can Have Complete Control Over Their Build Environment.

If you need a new version of a tool, update the Dockerfile in source control as part of the pull request to upgrade the tool. If you need to build an old branch, the old branch will have the old build environment.

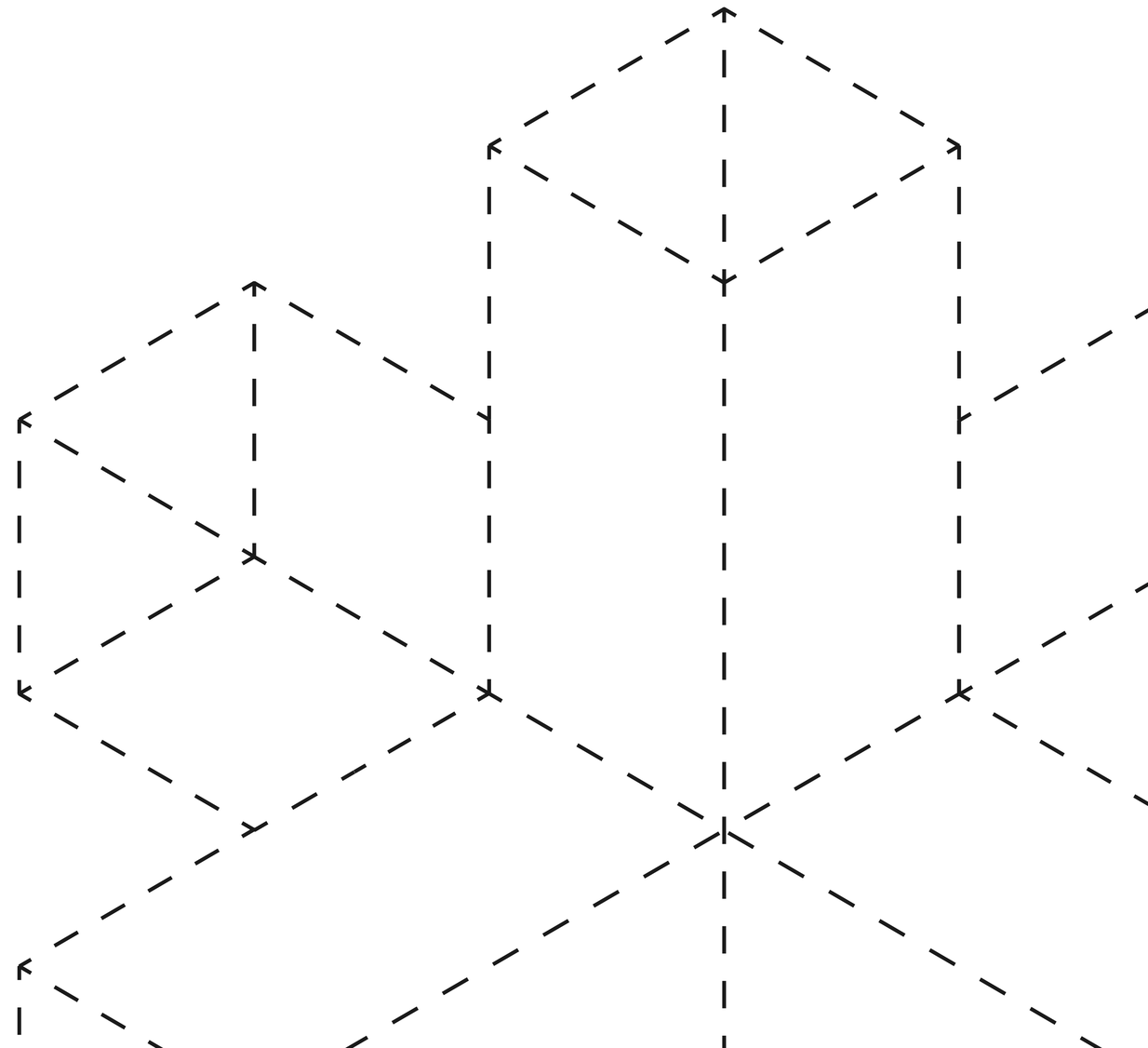
What if I’m Using Windows and Linux Containers?

If you are working in an environment with a mixture of Windows and Linux containers, then you will probably want to use labels to differentiate the build agents configured to run Windows containers from the agents configured to run Linux containers.

Flexible, Governed, and Secure CI at Scale

Jenkins reigns as the top choice for continuous integration (CI), but when it comes to scaling across diverse teams, the game changes. You need efficiency without the extra management weight. Security and compliance? Non negotiable, but without stifling productivity. And as your projects evolve from monoliths to more complex structures, and as hybrid cloud becomes the norm, the challenge of supporting various infrastructures while optimizing software delivery becomes more daunting.

Organizations facing these challenges should consider CloudBees to help with the burden of managing and scaling Jenkins. Want a stable and resilient CI that CloudBees backs directly? It s time to consider making the switch. CloudBees isn t just about keeping Jenkins running smoothly; it s about empowering your teams to push forward, faster and more securely than ever.



CloudBees

Author

Darin Pope

Developer Advocate, CloudBees

Darin is a developer advocate for CloudBees, where he produces technical digital and written content for users. He is also the co-host of a weekly podcast named DevOps Paradox.

@darinpope | cloudbees.com

Learn more cloudbees.com/get-started

CloudBees, Inc.

4 North Second Street, Suite 1270
San Jose, CA 95113
United States
cloudbees.com
info@cloudbees.com