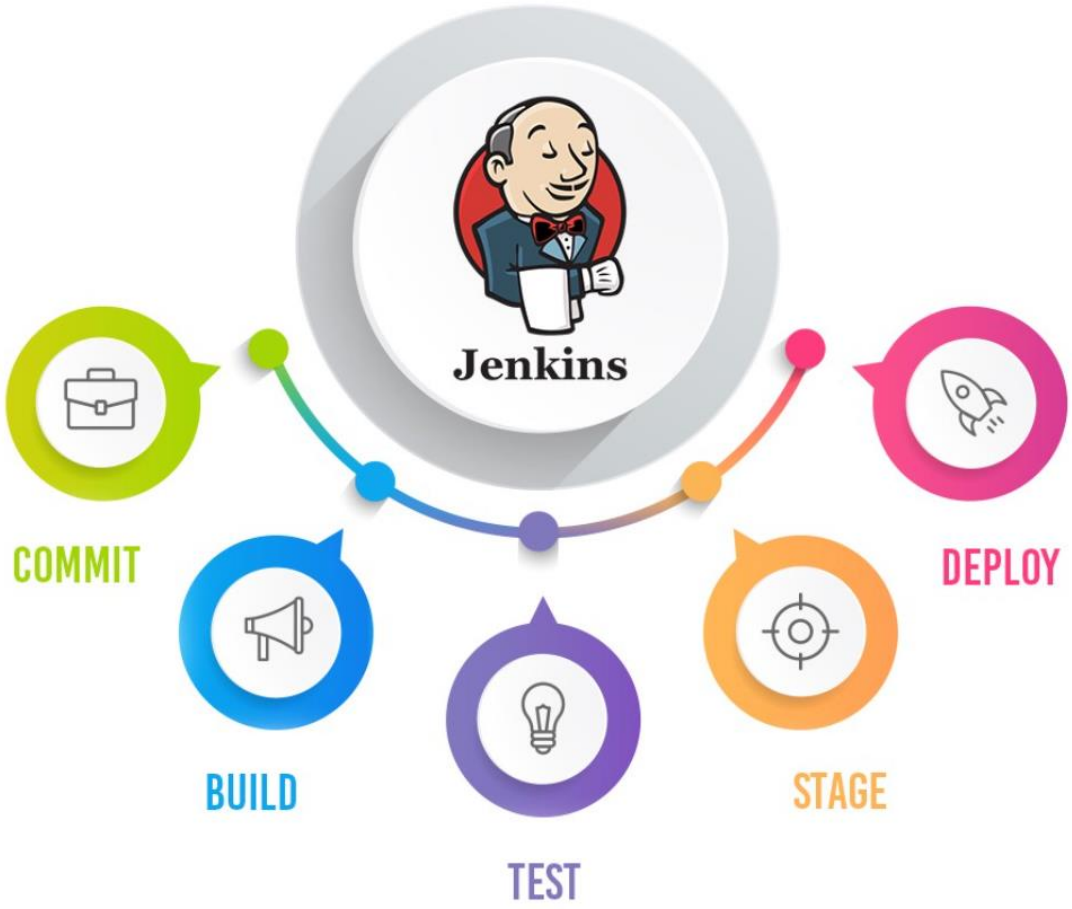
Jenkins- a tool used for CI/CD



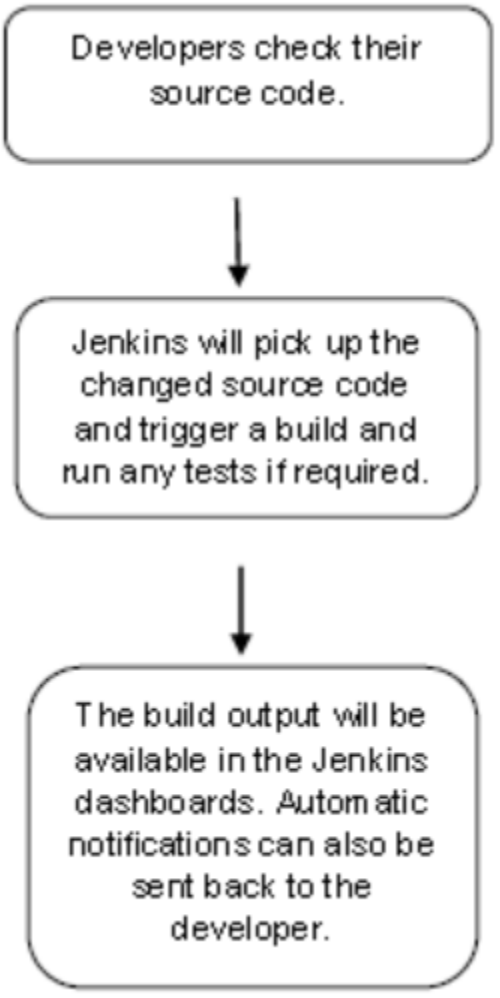
What is Jenkins? Why we need it?

Jenkins is a powerful application that allows *continuous integration and continuous delivery (CI/CD)* of software projects. It is a free source and can be integrated with several testing and deployment technologies.

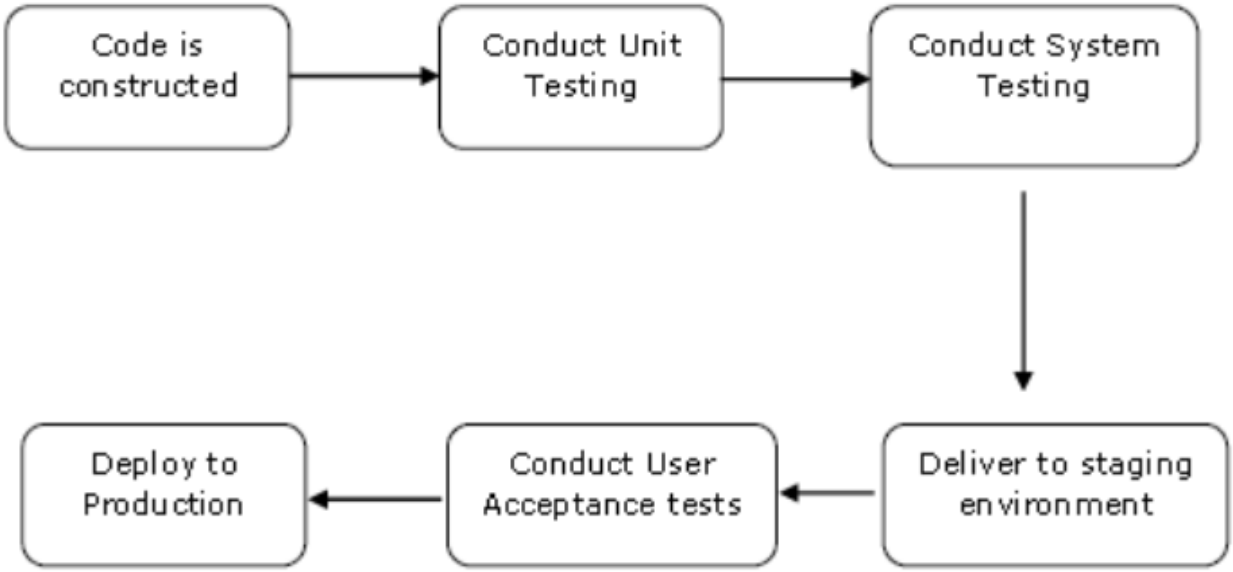
The need for a tool such as Jenkins arises when the developers need to integrate code into a shared repository at regular intervals. This development practice is known as ***Continuous Integration*** (CI). This concept was meant to remove the

problem of finding later occurrences of issues in the build lifecycle. Continuous integration requires the developers to have frequent builds. The common practice is that whenever a code commit occurs, a build should be triggered.

The following flowchart demonstrates a very simple workflow of how Jenkins works.



Jenkins provides good support for providing ***continuous deployment and delivery***. If you look at the flow of any software development through deployment, it will be as shown below.



There are several ways you can install Jenkins in your master node (EC2 instance in our case).

Using docker container

Blueocean container

https://hub.docker.com/r/jenkinsci/blueocean/

or jenkins/jenkins

https://hub.docker.com/_/jenkins

Do not forget to add the volume on Jenkins's workspace and docker socket (do some research on how to add both workspace and docker socket volume to either blue ocean or Jenkins's container) when running the container in your terminal.

For example

```
docker run -p 8080:8080 -p 50000:50000 -v /your/home:/var/jenkins_home jenkins
```

Using a War file

Automate the provisioning of Jenkins using a bash script

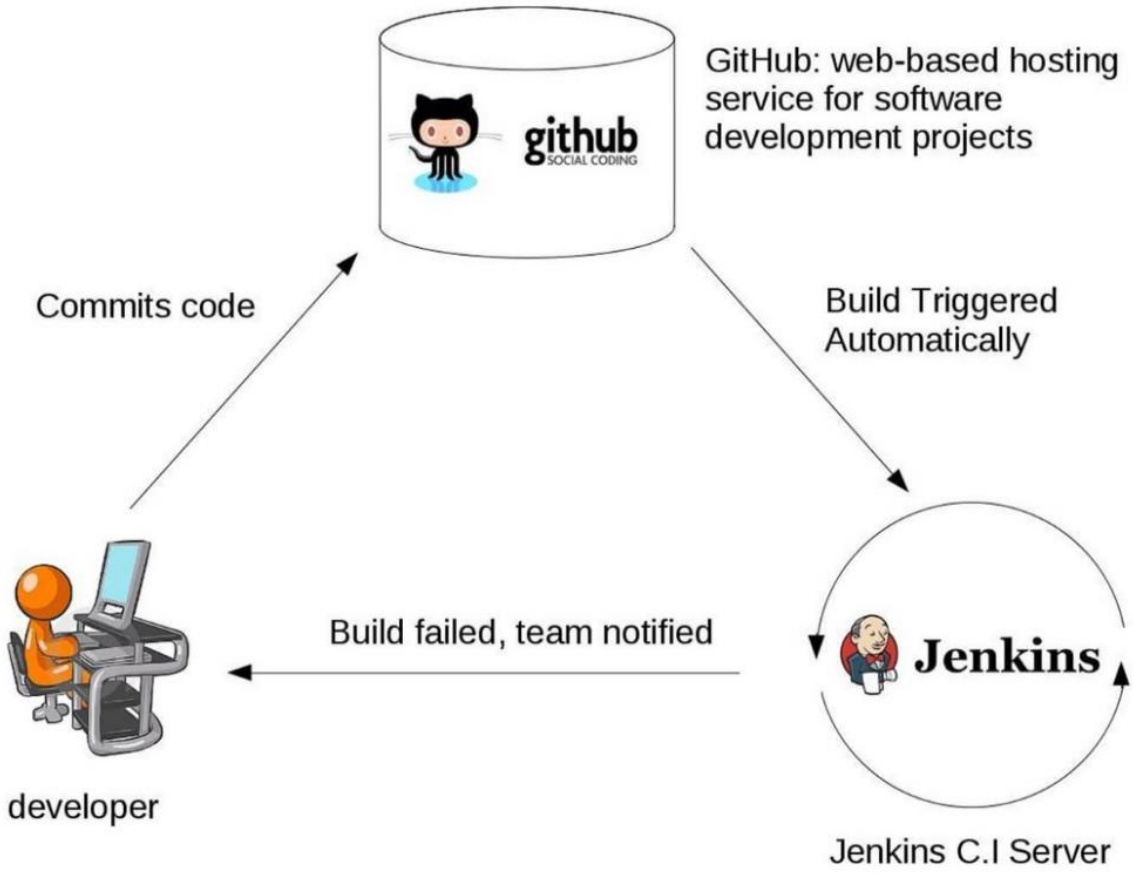
Terraform will provision an AWS EC2 instance and install git, Apache Maven, Docker, Java 8, and Jenkins as shown in the **install_jenkins.sh file:**

```
#!/bin/bash
sudo yum -y update

echo "Install Java JDK 8"
sudo yum remove -y java
sudo yum install -y java-1.8.0-openjdk

echo "Install Maven"
sudo yum install -y maven

echo "Install git"
sudo yum install -y git

echo "Install Docker engine"
sudo yum update -y
sudo yum install docker -y
sudo sudo chkconfig docker on
```

```
echo "Install Jenkins"
sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo
sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
sudo yum install -y Jenkins
sudo amazon-linux-extras install epel -y
sudo usermod -a -G docker jenkins
sudo chkconfig jenkins on

echo "Start Docker & Jenkins services"
sudo service docker start
sudo service jenkins start
```

Using a browser, open the page at http://jenkins_ip_address:8080; the Jenkins admin page will be displayed:

Jenkins with Github



Jenkins, has a number of plugins for integrating into Git and GitHub such as:

1. **GitHub plugin**: integrates Jenkins jobs with GitHub repositories for triggering builds and reporting commit status
2. **GitHub Authentication plugin:** uses GitHub as the authentication provider for your Jenkins instance
3. **Git plugin:** use Git repositories as the source of code for a Jenkins job
4. **GitHub SQS plugin:** trigger builds via the GitHub SQS (AWS) service hook

5. **GitHub Issues plugin:** create a GitHub issue whenever your build fails, and automatically close it once the build starts passing again.

Jenkins and Docker

Docker is a software platform for building applications based on containers — small and lightweight execution environments that make shared use of the operating system kernel but otherwise run-in isolation from one another.

Combining Jenkins and Docker together can bring improved speed and consistency to your automation tasks. A means to isolate different jobs from one another, quickly clean a job's workspace or even dynamically deploy or schedule jobs with Docker containers would increase resource utilization and efficiency.

Docker plugins for Jenkins

1. **Docker plugin**: dynamically provision agents with Docker
2. **Docker Pipeline plugin:** build and use Docker containers in Pipelines
3. **Kubernetes plugin:** dynamically allocate and schedule Docker agents on a Kubernetes cluster
4. **Docker build step plugin:** add Docker commands as build steps
5. **CloudBees Docker Hub Notification plugin:** allows Jenkins to receive webhooks from hub.docker.com to drive pipelines based on Docker in Jenkins.

Pipeline as Code with Jenkins

The default interaction model with Jenkins, historically, has been very web UI driven, requiring users to manually create jobs, then manually fill in the details through a web browser. This requires additional effort to create and manage jobs to test and build multiple projects, it also keeps the configuration of a job to build/test/deploy separately from the actual code being built/tested/deployed. This prevents users from applying their existing CI/CD best practices to the job configurations themselves.

With the introduction of the Pipeline Plugin, users now can implement a project's entire build/test/deploy pipeline in a `Jenkinsfile` and store that alongside their code, treating their pipeline as another piece of code checked into source control.

Continuous Delivery plugins for Jenkins

1. **Pipeline plugin**: allows creating Pipeline scripts for defining build/test/deploy stages of a delivery pipeline
2. **Pipeline Utility Steps plugin:** provides a number of additional, useful, steps to the Pipeline DSL
3. **Job DSL plugin:** creates a DSL to orchestrate job creation

Jenkins Market Share and Competitors in Software Development Tools

Jenkins Case Study: Topdanmark

Topdanmark, a Danish insurance company

Automating the build processes with a highly configurable Jenkins platform

Summary: Topdanmark, a leading Danish insurer chooses Jenkins as the “de facto” product to build their CI/CD platform.

Challenge: Balancing the needs of DevOps modernization and cloud migration while maintaining legacy systems.

Solution: A highly-configurable CI/CD platform that allows for automation and ease of development.

Results:

- 100% automatic creation of Jenkins instances
- the ability to release and deploy an artifact whenever, wherever

- software developers focused on developing software rather than operations
- smaller monoliths and containerization

Jenkins Case Study: Tymit

credit card processing company

Fintech Innovation With A Jenkins Backbone

Solution: Tymit, a revolutionary credit card processing company, leveraged Jenkins to build a compliant, transparent, and secure modern DevOps platform to drive product innovation, handle instant financial transactions and support thousands of users in real-time.

Challenge: Create a solidly reliable CI/CD platform that provides the technology team with the agility and the flexibility needed to innovate while ensuring the security and scalability their fintech service requires.

Results:

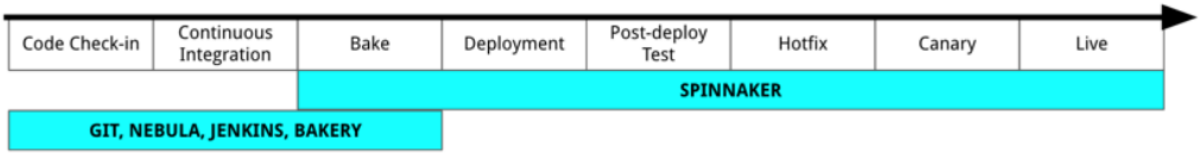
- faster delivery of mobile, microservices, and operational services
- reduced software testing and release cycles by 50%
- ability to support thousands of users for real-time transactions
- created a secure, controlled, and compliant fintech environment

Case Study: Netflix

A streaming service

How does Netflix build code before it's deployed to the cloud?

The tools and techniques used to go from source code to a deployed service serving movies and TV shows to more than 75 million global Netflix members are shown below:



There are several steps that need to happen before a line of code makes its way into Spinnaker, Netflix's global continuous delivery platform:

- Code is built and tested locally using Nebula
- Changes are committed to a central git repository
- A Jenkins job executes Nebula, which builds, tests, and packages the application for deployment
- Builds are "baked" into Amazon Machine Images
- Spinnaker pipelines are used to deploy and promote the code change

What is Jenkins, and why does everyone love it?

Jenkins is a job runner that is primarily used to do **Continuous Integration**.

As a job runner, it can do "anything" technically, but it has three primary use-cases:

1. Build Automation & Automated Testing (the core of CI)
2. Deploying to Pre-Production Servers
3. Deployment to Production Servers

Jenkins has features, delivered by plugins, to help with Continuous Integration (CI), like automated testing.

Ah, the plugins. **Plugins are what make Jenkins, Jenkins.** They are also the crux of the Jenkins love-hate relationship.

The main value of Jenkins is that library of plugins. These usually-community-created, free-and-open-source plugins extend Jenkins's way beyond its job-running capabilities. Need to back up a system with a button click? How about deploy a file to a server with a click? There's a plugin for that.

Without plugins, Jenkins wouldn’t do much and people wouldn’t love it. But with plugins, Jenkins is much more chaotic, a main source of Jenkins hatred.

Why does everyone hate Jenkins?

There are five common complaints we found in our research: expertise confusion, expert bottlenecks, and self-service chaos, lack of visibility, and scaling difficulty. Because of all these, Jenkins’s chaos can happen faster than you expect.

1. The gap between "self-service" and "required expertise" creates complications and chaos.

Problem	Effect	Impact
Jenkins instances are complex and often do more than basic CI	You need a Jenkins expert for even the most basic maintenance	Experts become bottlenecks, slowing new projects, builds, releases, etc.
Tightly secured instances create expertise bottlenecks	Self-service seekers will create their own instances or find other alternatives	Builds/projects are not systematically distributed and are chaotic, increasing risk and slowing things down
Loosely secured instances become chaotic and unstable over time	Non-expert users modify configuration, install/update plugins for their project	Unrelated builds/projects stop working, causing lost productivity and frustration across teams

Teams that use Jenkins as it was originally intended — that is, as a project-specific, expert-managed CI server — don't experience these problems at first. The expert(s) sets up Jenkins quickly and provides basic self-service options for everyone else.

However, things quickly get chaotic after that since Jenkins can do so many things and is so easy to extend with plugins. And that's where the problem comes in: Most teams end up with Jenkins’s configurations much more complex than the average dev can handle.

This turns your Jenkins experts into a bottleneck. To bypass the bottleneck, other team members will try to maintain the server themselves or will just create a separate Jenkins install someplace else. This creates complications and chaos by adding to Jenkins and by distributing Jenkins projects in a non-systematic way.

All of this ends up being very expensive, because when you cannot rely on your tool alone. Your Jenkins people are more of a blocker than Jenkins itself. Sounds a lot like pre-DevOps, manual processes, doesn't it?

2. Jenkins experts can quickly become both a bottleneck and a firewall, which gets very expensive.

Problem	**Effect**	**Impact**
Jenkins is often used for more than CI	You need a Jenkins expert	Experts become bottlenecks, firewalls, and are very expensive
Experts become bottlenecks	Jenkins-related tasks *require* experts	New projects, builds, releases, etc. depend on that one expert (very slow)
Experts become firewalls	Jenkins knowledge lives inside experts, rather than on some tool	Chaos ensues if experts ever leave or are unavailable
Experts are Jenkins-only	Harder to hire; All focus is on one tool	Very costly to the business

Jenkins seems great for small companies because it’s free and open source, and there are so many free tutorials and free plugins! But is it free?

Consider this example: a small team of four decides to use Jenkins for CI and build automation. One of the members, Terry, volunteers to learn Jenkins, follow the basic tutorials, and set it up. Within just a few hours, Terry manages to get some basic projects configured, and everyone can use it.

Over time, other members ask Terry to set up new projects in Jenkins or ask Terry if Jenkins can do this or if Jenkins can do that. And of course, it can, because there's a plugin for that!

Eventually, the team starts relying on Jenkins to do everything from basic operations jobs to continuous delivery (CD). However, Terry is the only one of the four who can configure and set it up. Now all work must go through Terry, and Jenkins tasks become 90+% of Terry's job.

In this common scenario, "Terry" becomes both a Jenkins bottleneck and a firewall, and the team essentially loses a developer to any non-Jenkins work. The way most teams use Jenkins, you end up hiring full-time Jenkins expert to create and manage Jenkins pipelines. Your "free" Jenkins tool costs one (or more) employee salary and talents... expensive!

Hiring engineers (and keeping them) is very costly. It's much harder if that person must be a Jenkins expert, because Jenkins's expertise requires such skills ask:

- Choosing the best plugins
- Developing pipelines
- Managing and organizing projects
- Creating reports and visibility for other users
- Groovy (Jenkins's DSL—it's relatively easy to learn, but like PowerShell, some expertise is needed to use it AND experts will often do too-complicated things with Groovy that a non-expert cannot maintain.)

Software developers cost companies an average of six figures (USD) per year per engineer. Holy mackerel—that's a lot to spend on someone to channel all their talents and time into just your free tool.

To try to counter this, many teams try to distribute the work of managing Jenkins through "self-service," and let different team members manage their Jenkins instances. But this causes problems too.

3. Self-service can be too much of a good thing.

Problem	Effect	Impact

Having a ton of plugins and complications in your Jenkins (especially in the controller/master)	The Jenkins controller slows way down AND becomes difficult to navigate for non-experts	Bottlenecks form, defeating the purpose of faster-paced DevOps tools
Anyone can install any plugin, which don't require automatic testing	Vulnerabilities can be easily introduced	Increased risks of malware, downtime, and other nasties
Lack of visibility means *people* hold Jenkins's knowledge, rather than Jenkins itself	Information is behind human "firewalls"	Slows self-service (and thus work) down, since you're dependent on people rather than just the tool

Jenkins is a by-design "democratic" tool, allowing anyone to create projects, configure agents, install plugins, etc. But like too much of any good thing, too much self-service can cause problems. Namely, these problems are speed, vulnerability, and longevity.

Because anyone can install any plugin, odds are that you will not know who installed a plugin, why they installed it, how often it's being used, or whether it can be removed without breaking anything; Jenkins doesn't indicate any of this. Heck, some plugins get automatically added because *another* plugin requires *that* plugin.

If a less-thorough colleague doesn't seek answers to ALL these questions from the people who installed them and modifies or deletes a plugin, your stuff can break. And if all these plugins are on your controller, you end up with a sluggish Jenkins that may result in more failed builds.

Because anyone can install any plugin—which are not of equal quality—it's too easy to introduce vulnerabilities or even malware into your Jenkins installations.

To avoid this, perhaps you'll require your developers to get approval and advice on installing plugins from Jenkins expert... which makes the Jenkins expert a bottleneck again.

And when your information relies on human memory and manual intervention for updates, your Jenkins pipelines become "legacy" much faster than competing tools' pipelines.

4. Poor visibility into Jenkins installs and projects creates chaos, can disrupt work, and increases risk.

Problem	**Effect**	**Impact**
Jenkins project ownership is unclear	Instances may lock if someone makes a mistake on a project that isn't theirs	All Jenkins work halts until the correct person can remedy the issue
Information you may need for auditing or rollbacks are not centralized	You rely on people's memories rather than on a tool	Risk increases (human memory is much less reliable than software logs)

Unlike dedicated CI/CD tools, Jenkins does not have “applications” or “releases.” Instead, everything must be its own project (what used to be called jobs). Though there is a plugin for folders and views, you cannot categorize or organize Jenkins projects easily or clearly without work.

What ends up happening is that you have no idea who owns what Jenkins’s project. You could accidentally work on someone else's project and mess it up, delete it and mess it up, or someone else can mess with your project and potentially lock the entire Jenkins instance.

Lack of visibility leads to confusion, which slows down work and could risk more than just lost time. And your Jenkins instances become legacy/outmoded much faster when visibility is obscured between users, teams, and installs.

5. Jenkins is very hard to scale efficiently.

Problem	Effect	Impact
Not all plugins are supported by Jenkinsfile	Jenkins is hard to back up without heavy manual intervention	Lots of wasted time doing manual processes on an automation tool
Different departments need different instances	Doing "Jenkins as Code" becomes much harder than similar CI/CD tools	Costs both time and money to stay organized
Jenkins "database" is just XML files on a disk	XML format makes it relatively expensive to read and nearly impossible to index	Needed expertise costs lots of money and time.
Jenkins can only have a single controller	Jenkins’s controller cannot run in high availability mode.	Scaled businesses cannot afford risks of downtime and lost work time due to Jenkins’s outages

Young devs have plenty of time, energy, and passion but have no money. Jenkins is perfect for them because it’s free, but it’s not good for mature organizations with responsibility to customers and stakeholders, because Jenkins is not simply built for scaling.

Because plugins are created by the Jenkins community, they won't always have what you may consider "basic features," and not all plugins are supported by Jenkinsfile (which is exported Jenkins’s configuration). To back up Jenkins, you must manually write down a list of plugins and manually install them, and then configure those plugins manually. Compare this to other tools that allow scheduled, automated backups.

In a similar way, doing "Jenkins as Code" requires extra, manual effort at scale, because different departments will spawn different and often multiple Jenkins instances. Because of the difficulty of back-ups, “Jenkins as Code” means backing up multiple instances in a difficult way every time.

Another thing that slows you down is Jenkins's database, which is basically just XML. This was originally designed with interoperability in mind, not performance. As a result, the verbose and dynamic nature of the xml format makes it relatively expensive to read and more so to index.

And perhaps most importantly: scaled organizations rely on constant up-time. This requires high-availability and load balancing, which Jenkins cannot do with just a single controller. You can "duct tape" solutions together as a standby failover, but (as with any duct-taped solution) these often have performance issues.

Other issues

Jenkins does not have its plugins. It depends on third-party plugins to accomplish its tasks. The problem is that Jenkins does not control regulating the patching protocol for these plugins. This can potentially expose the software to a malicious attack.

Jenkins’s plugins update all the time. Yesterday before leaving the office, your job worked perfectly fine. When you run the same job today, it fails because the plugins you were using yesterday have been updated.

16 Jenkins Alternatives for Continuous Integration in 2021

Use this link to evaluate and read more on Jenkins alternatives.

https://blog.inedo.com/jenkins/alternatives-for-continuous-integration?utm_source=Google&utm_medium=CPC&utm_campaign=Jenkins&utm_term=jenkins%20vs%20github&utm_campaign=jenkins&utm_source=adwords&utm_medium=ppc&hsa_acc=5810695529&hsa_cam=12879955820&hsa_grp=123922410700&hsa_ad=525777807767&hsa_src=g&hsa_tgt=aud-1413957521019:kwd-1621962340252&hsa_kw=jenkins%20vs%20github&hsa_mt=b&hsa_net=adwords&hsa_ver=3&gclid=Cj0KCQjw5-WRBhCKARIsAAId9Fn99IeJwN0JSBjSwn26cdtI3TIt7M_kNpJfgfLVhjEM6AXwiV7Ff8oaAmalEALw_wcB

For this class, I would like to focus on GitHub Actions as the Alternative CI/CD tool.