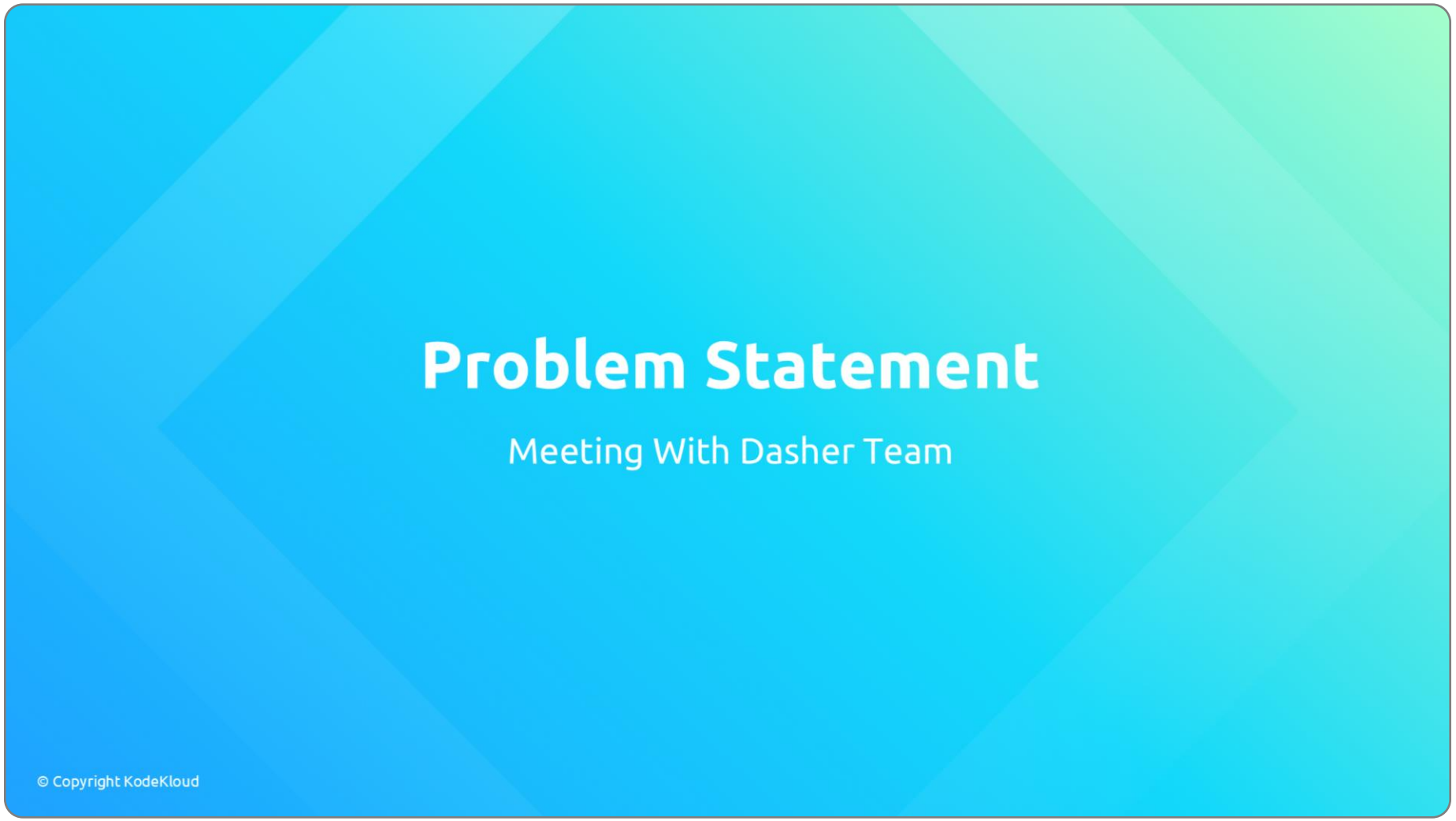
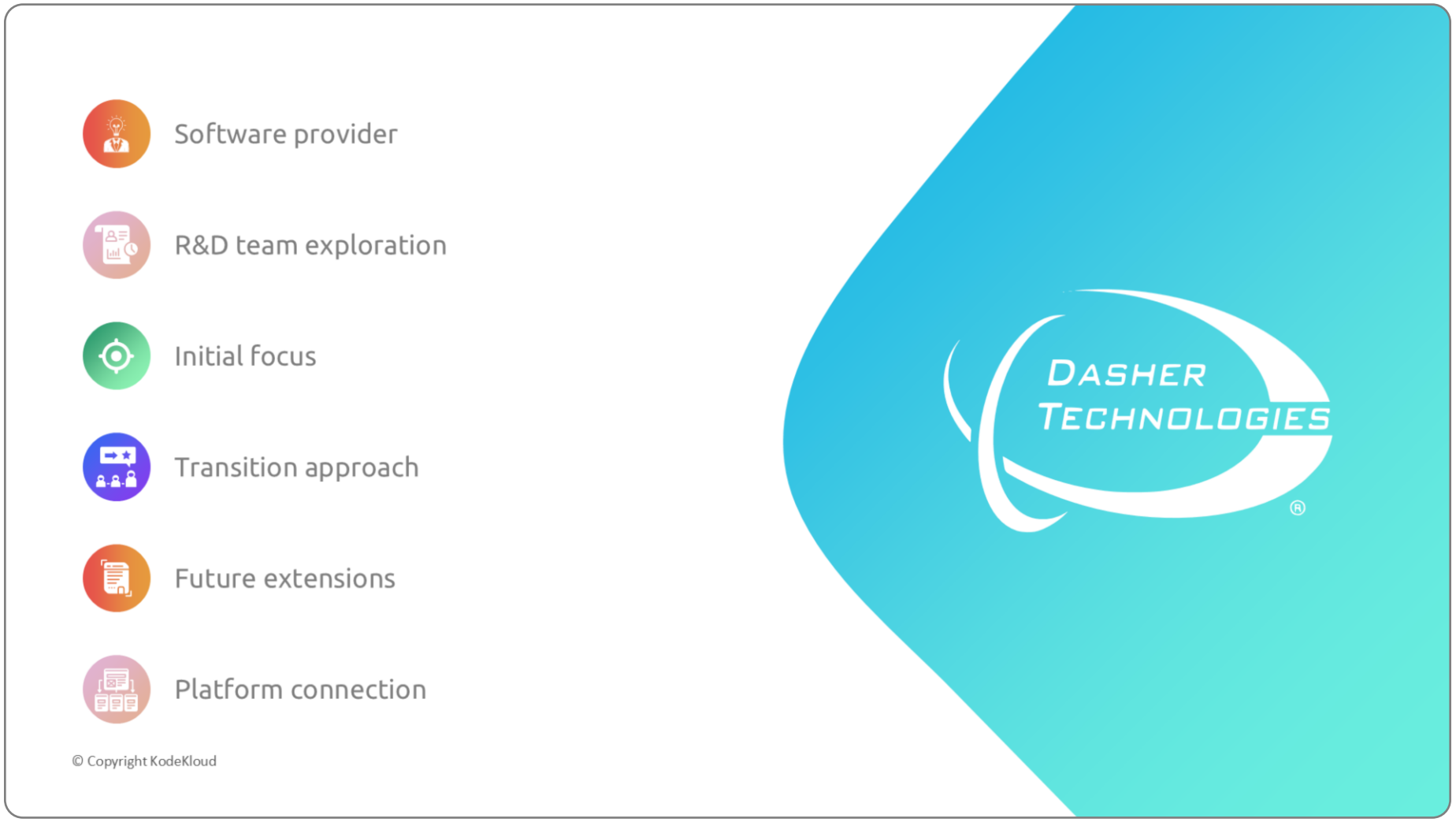
KodeKloud

Visit www.kodekloud.com to learn more.



Let’s start the course by understanding the DevOps prerequisites of a software provider. Over the duration of this course, we will check into and learn about how Jenkins can be used to meet these DevOps requirements.



Dasher technology is a software provider, which offers a platform to facilitate the connection of data, applications, and devices across on-premises environments for businesses. Recently, their Research and Development (R&D) team has been exploring the possibilities of transitioning their services to the Cloud and leveraging container technologies. Initially, they aim to apply this to one of their NodeJS-based projects, with intentions to extend this approach to other projects developed in Java and Python at a later stage.

Task Dash Team DevOps Requirement

Docker for Containerization

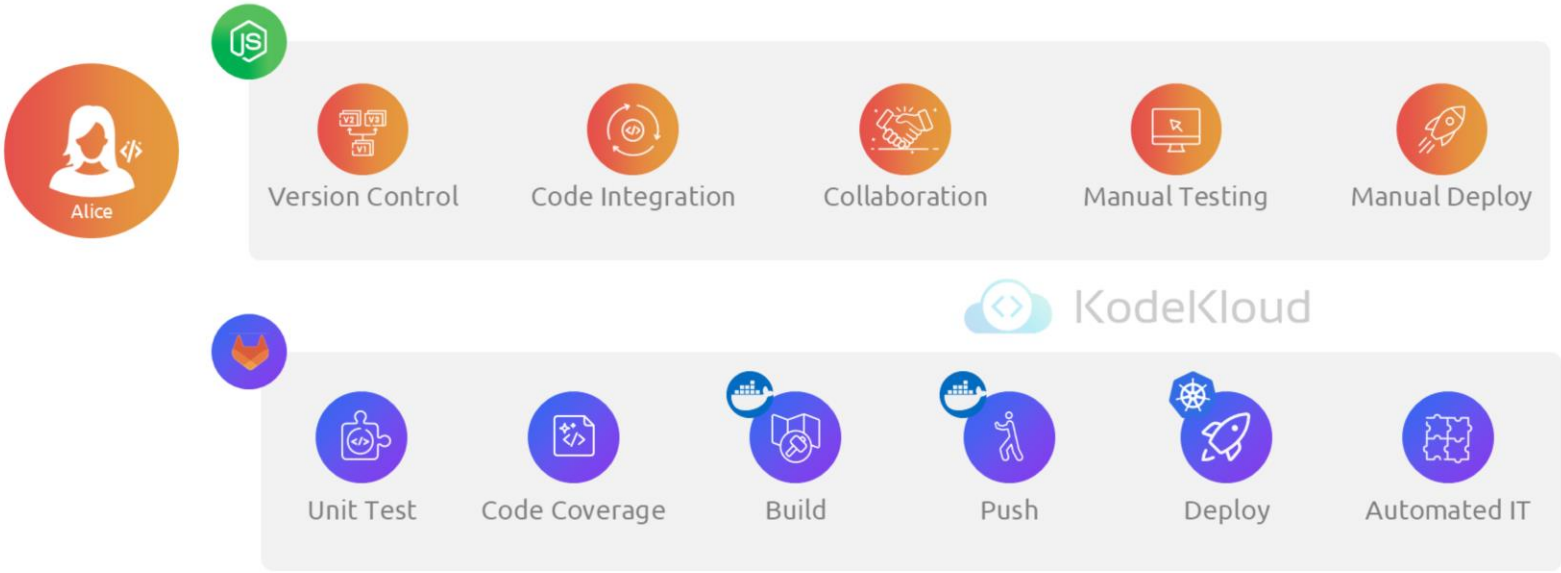
KodeKloud

Kubernetes for Container Orchestration

AWS Lambda Functions

They've introduced a DevOps team for this initiative, led by Alice, who will be responsible for establishing the DevOps pipeline for the project right from its inception, adhering to industry best practices. Their work will revolve around a multi-cloud infrastructure, utilizing Docker for containerization, Kubernetes for container orchestration and Deploying to AWS Lambda functions.

Task Dash Team DevOps Requirement



Alice conducted a swift evaluation and learned that the current requirement pertains to a NodeJS project. The previous team operated without a Version Control System, with developers independently writing and manually integrating code. The testing process was sluggish and ineffective due to manual execution. Collaboration among developers was often hampered as they worked on separate code branches. With infrequent integration and testing, software releases carried more significant risks. Deployment of software to various environments, including development, staging, and production, was primarily a manual procedure.

To tackle these challenges, Alice and her team have opted to implement a Continuous Integration/Continuous Deployment (CI/CD) pipeline and have outlined the following key steps:
Adoption of Gitlab for version control and developer collaboration.
Implementation of unit testing and code coverage measures to expedite testing and minimize bugs.
Utilization of Docker Build and Push processes for containerization, and the application is deployed to Kubernetes.
Incorporation of automated integration testing as a final step.

The successful execution of these steps is expected to resolve the existing issues. Nevertheless, the team now faces the additional hurdle of selecting the most suitable CI/CD tool.

Task Dash Team DevOps Requirement

Jenkins

Travis CI

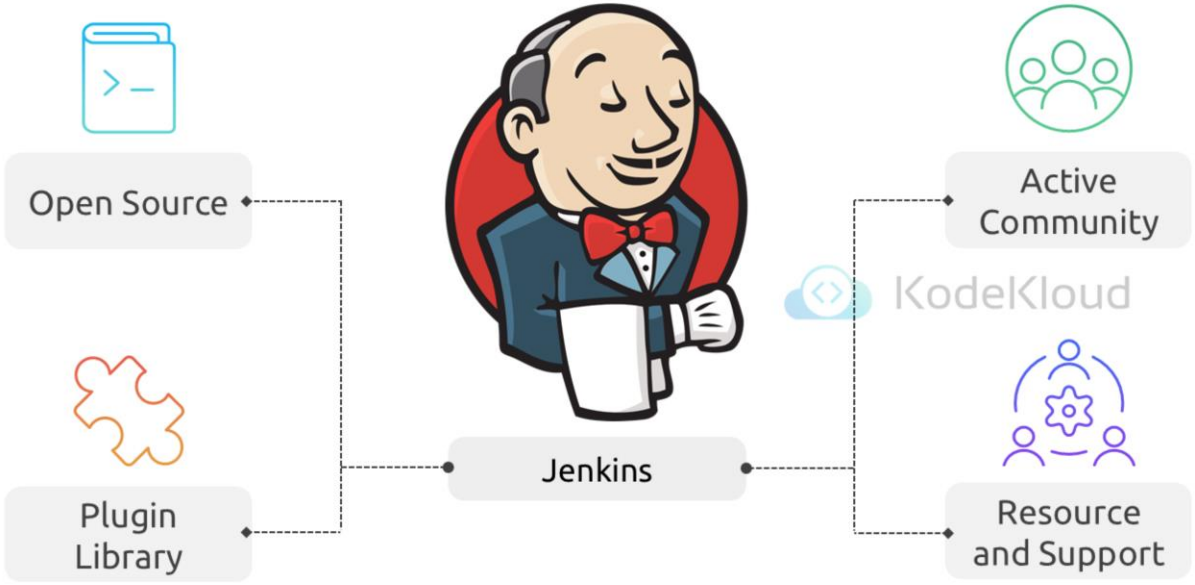
Circle CI

Atlassian Bamboo

Spinnaker

Alice wasn't short on options when it came to choosing a CI/CD tool. There were established players like Jenkins and Bamboo, alongside newer options like Travis CI and CircleCI. To pick the right one for her project, she dug into each tool's features and functionalities.

Jenkins



While exploring, Alice gets to know that Jenkins might not be the newest CI/CD tool on the block, but it holds its own against the competition. Its strength lies in its open-source nature. This allows for high customization through a massive plugin library.

This vast collection of plugins allows for high customization, no matter the project's specific needs. Additionally, the large and active Jenkins community provides ample resources and support.

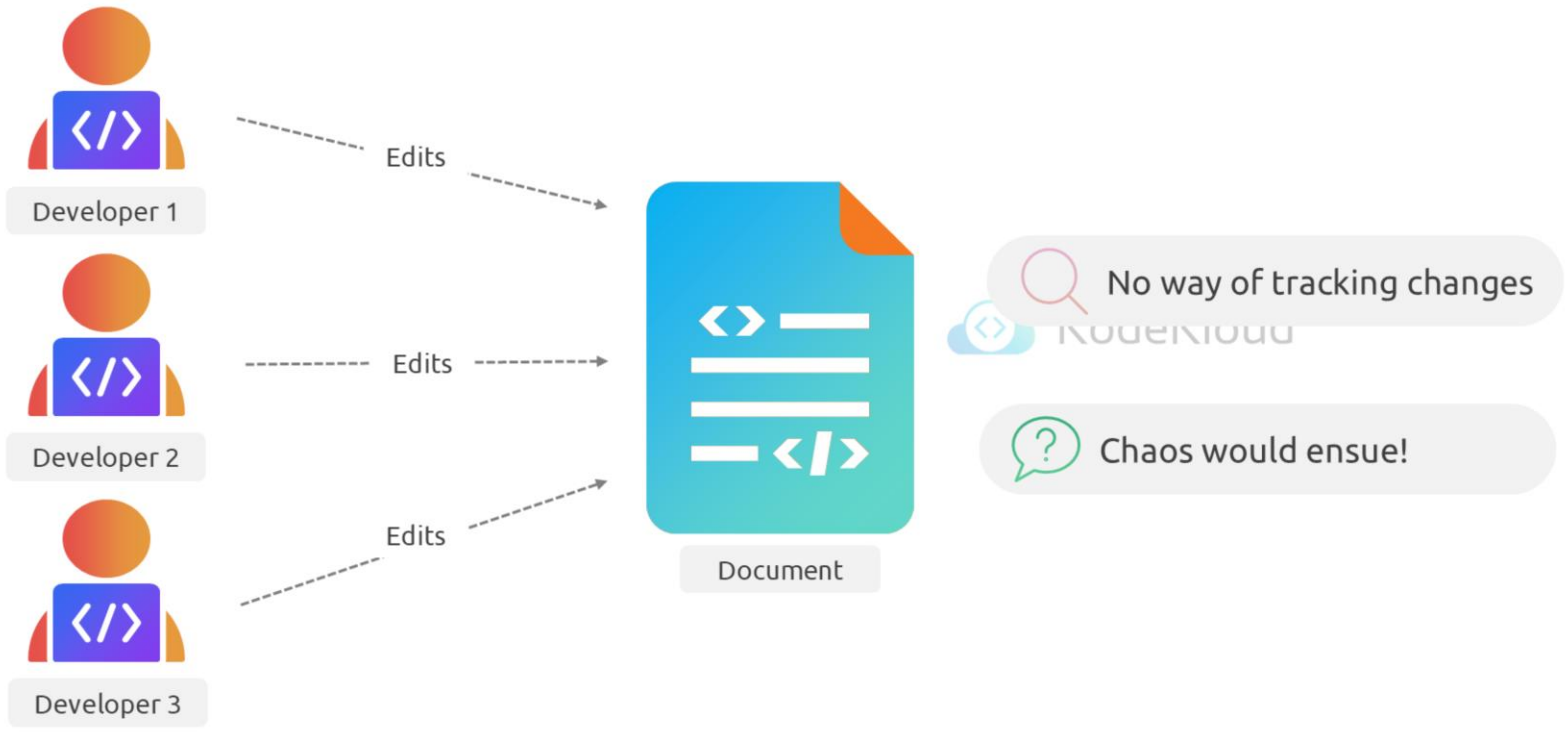
This combination of customizability, extensive features, and strong community support makes Jenkins a powerful and adaptable solution for CI/CD pipelines.

Throughout this training program, we'll explore creating Jenkins pipelines specifically for Node.js application, guiding you through integrating testing, security, and deployment across various platforms.

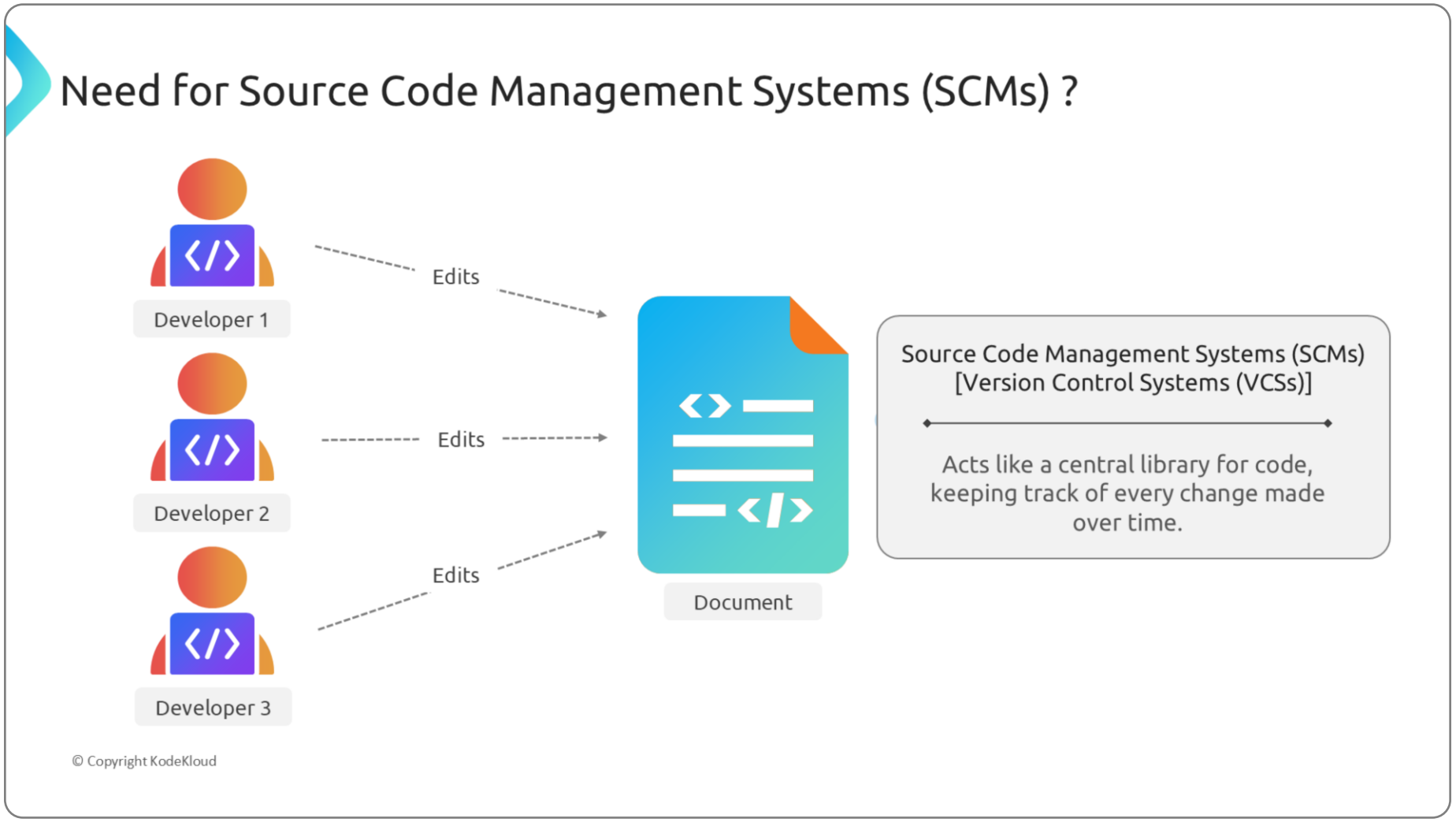
Source Code Management Systems (SCMs)

Whats the need source code management?

Need for Source Code Management Systems (SCMs) ?

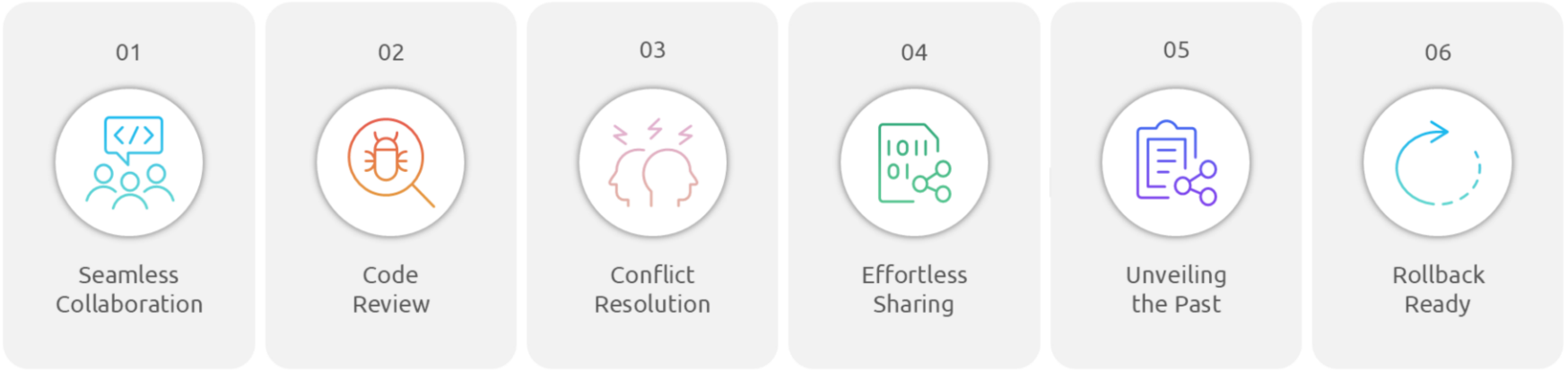


Imagine a project where everyone edits a single document at the same time, with no way of tracking changes. Chaos would ensue! Source code management systems (SCMs) or version control systems (VCSs) exist to prevent exactly that scenario in the world of software development.



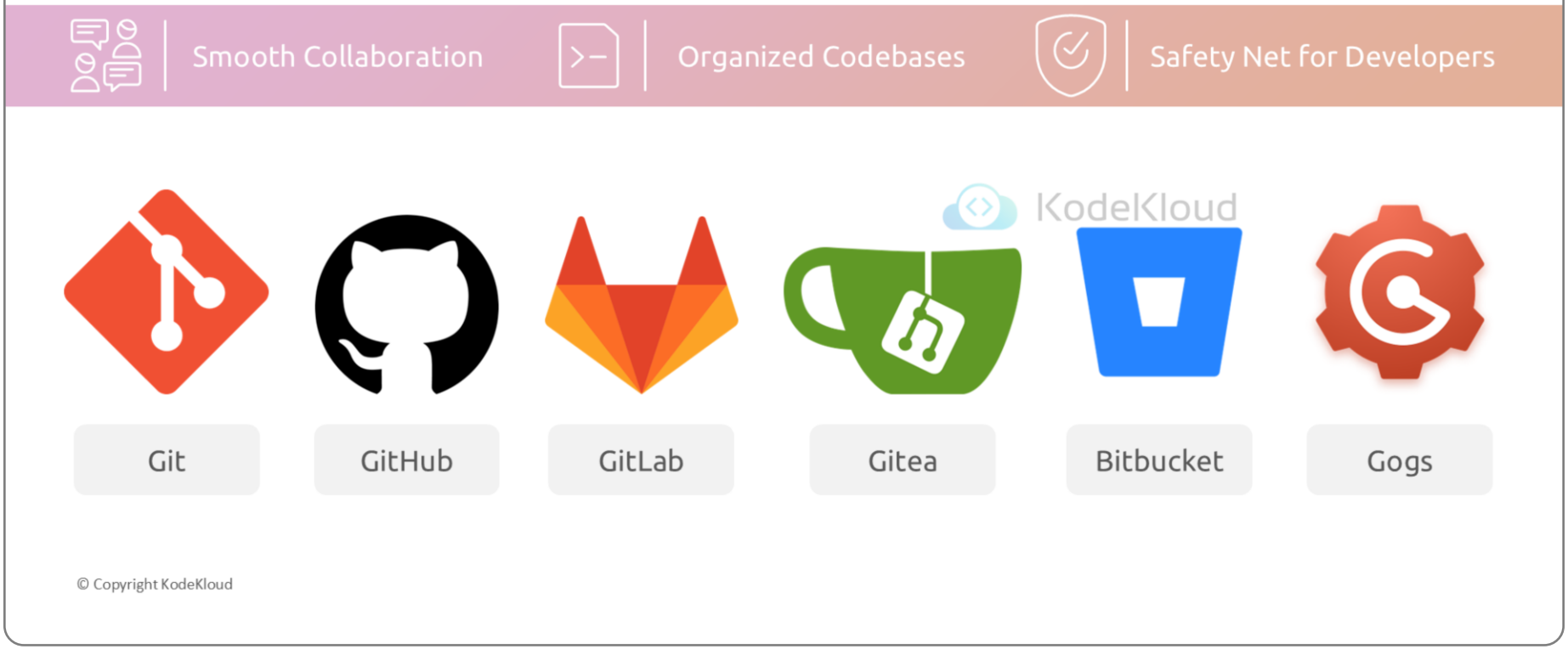
The chaos can be avoided by Source Code Management Systems (SCMs), also known as Version Control Systems (VCSs). These act like a central library for your code, keeping track of every change made over time.

Source Code Management Systems (SCMs)



Using a SCM enables you to:
Seamless Collaboration: Work together on the same codebase without conflicts.
Code Reviews: Teammates can comment on changes before they're merged, ensuring quality.
Conflict Resolution: SCMs help identify and resolve conflicting edits made by different developers.
Effortless Sharing: Share code easily and securely with your team.
Unveiling the Past: Track every change made to the codebase, providing a clear history.
Rollback Ready: Easily revert to a previous version if something goes wrong.

Source Code Management Systems (SCMs)



In essence, SCMs are the cornerstone of modern software development. They ensure smooth collaboration, organized codebases, and a safety net for developers, making them the single source of truth for your project's code.

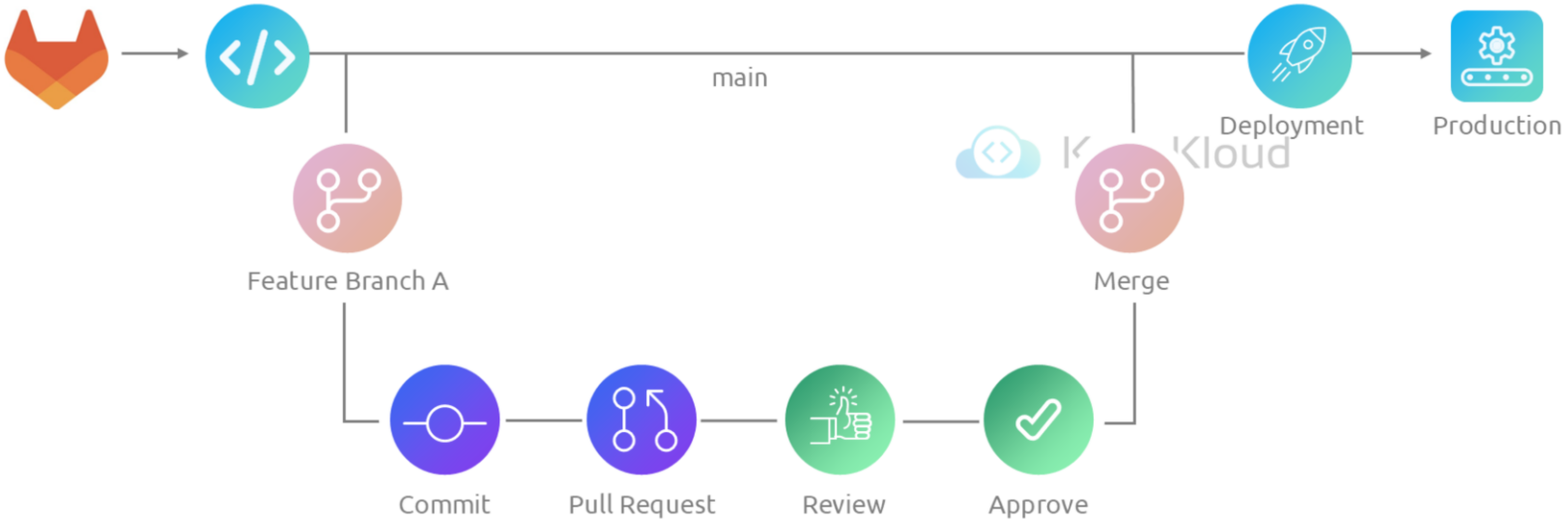
Git is the core technology used in many cloud-based SCM systems, including GitHub, Bitbucket, GitLab, Gitea and Gogs.

Basics of CI/CD

Let's explore the importance of Continuous Integration and Continuous Deployment (CI/CD).

Understanding CI/CD

Code residing on the main or master branch is deployed to production server or environment	Feature branch functions as a clone of the main codebase, enabling developers to work on a new feature until it's fully developed	Before merging, a review process is carried out, and the changes require approval from relevant team members or individuals



In a typical project, source code resides in a Git repository where it's both stored and versioned.

While this course will primarily utilize Gitea as our Git hosting platform, the concepts and pipelines we'll build are applicable to other popular options like Github, Gitlab, or Bitbucket. These platforms all offer central hubs for managing Git repositories and often extend Git's functionality with additional features. So, feel free to use your preferred tool

Typically, all code residing on the main or master branch is frequently deployed to production servers or environments.

When there's a need to introduce new features or simply modify existing code, developers create and collaborate on what's known as a "feature branch." This branch essentially functions as a clone of the main codebase, enabling a team of developers to work on a new feature until it's fully developed.

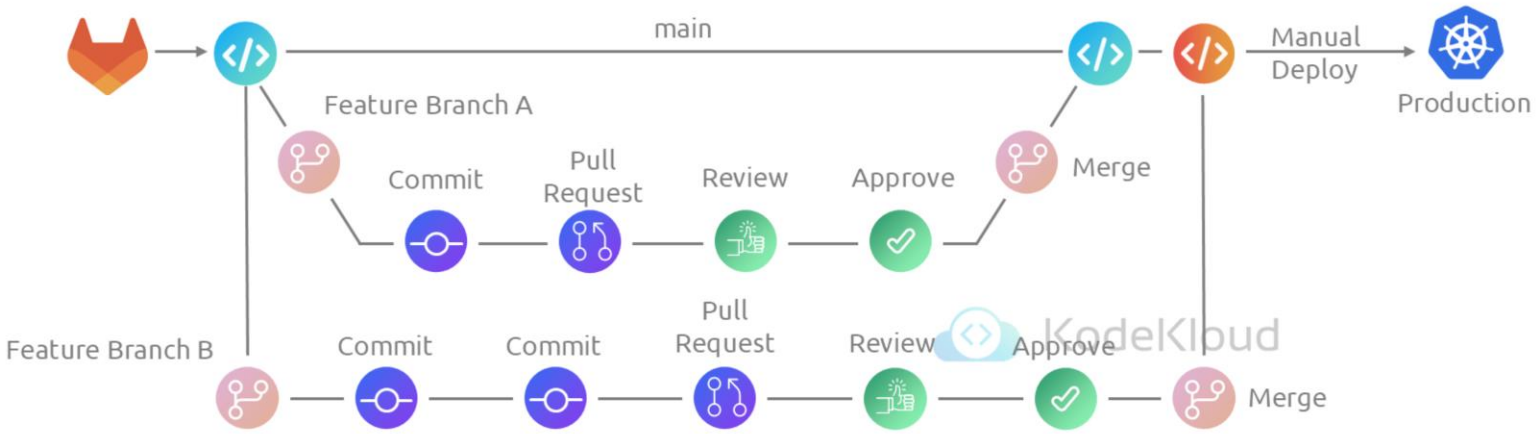
Once the necessary changes are made, the code is committed to the feature branch, and a pull request is initiated to merge this code back into the main branch.

Before the merge takes place, a review process is carried out, and the changes require approval from relevant team members or individuals.

Following a successful merge into the main branch, the code is then deployed to production, either manually or through an automated process.

This situation poses a significant risk to the application's stability, as there's often no testing conducted on the newly merged code before it reaches the production environment.

Need for Continuous Integration



Delayed Testing

Without CI, testing typically occurs late in the development cycle, often after multiple merges have taken place

Inefficient Deployment

In the absence of CI, deploying code to various environments (e.g., staging, production) often relies on manual processes

Quality Assurance Challenges

Without automated testing, ensuring quality becomes more reliant on manual testing, making it prone to human error

In real-time scenarios, you'll often find multiple developers working on different feature branches each focusing on a new enhancement.

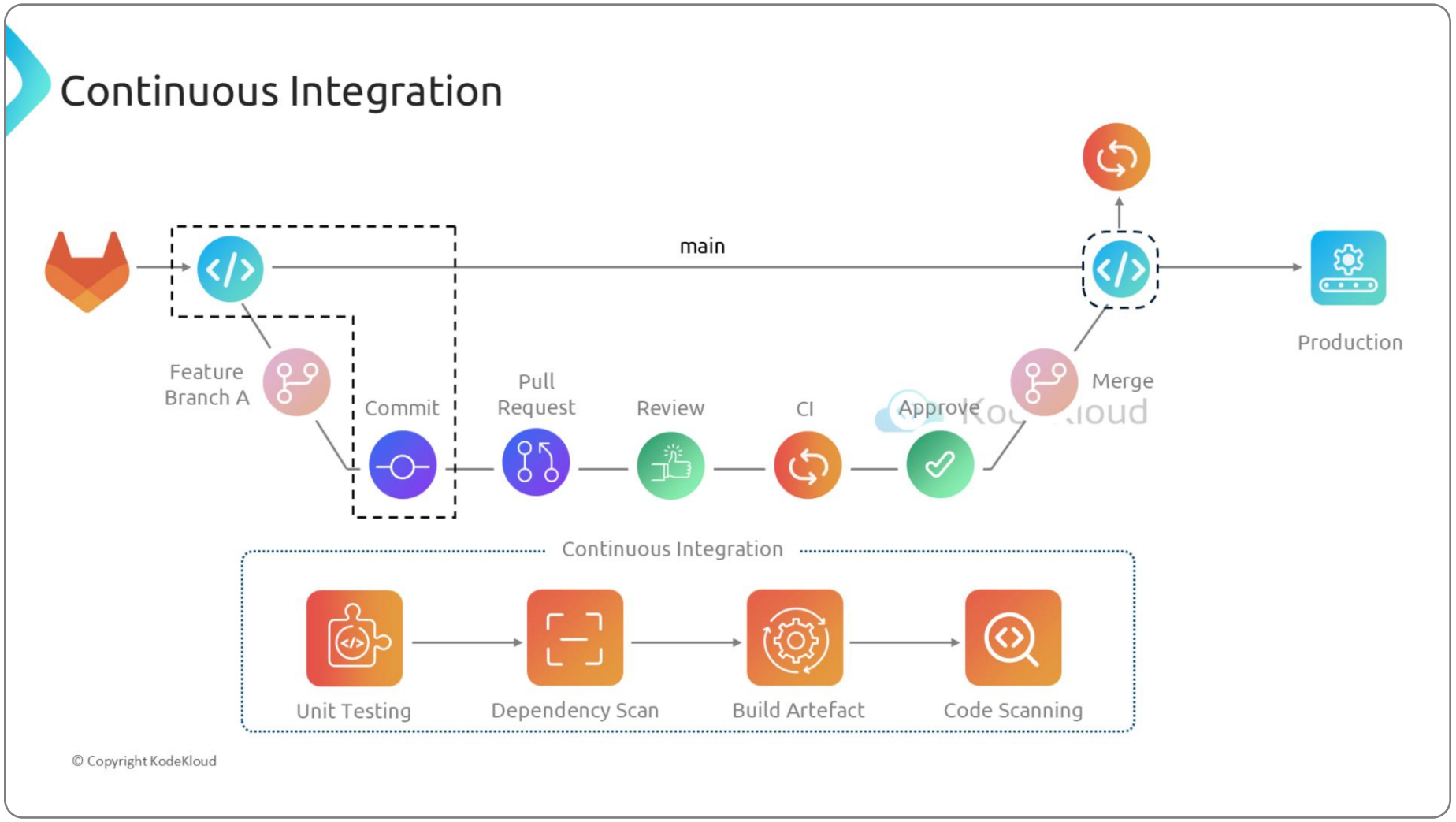
Performing multiple merges without the implementation of Continuous Integration (CI) in a software development workflow can lead to several significant problems and challenges such as:

Delayed Testing: Without CI, testing typically occurs late in the development cycle, often after multiple merges have taken

place. This delay in testing can make it harder to identify and rectify issues early in the development process, increasing the risk of defects making their way into production.

Inefficient Deployment: In the absence of CI, deploying code to various environments (e.g., development, staging, production) often relies on manual processes. This can lead to inconsistencies in deployment and potential configuration errors.

Quality Assurance Challenges: Without automated testing as an integral part of the development process, ensuring software quality becomes more reliant on manual testing, making it prone to human error and subject to resource constraints.



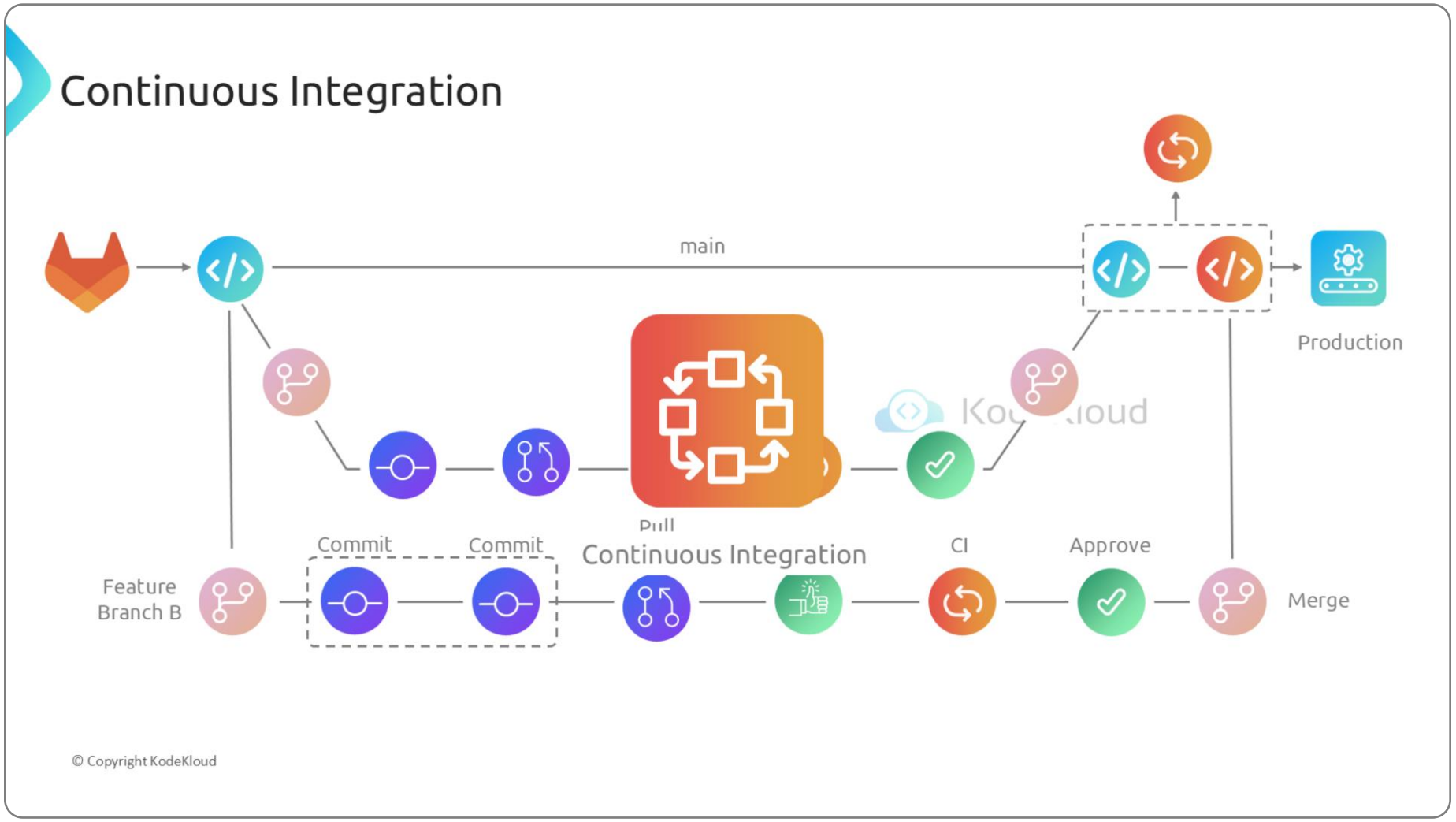
Let's imagine a scenario where Developer 1 creates a feature branch A, makes some modifications, and commits the code to this branch. A pull request is generated to merge these changes into the main branch. Before the merge, a team member reviews the code, and an automated CI pipeline is triggered.

The CI pipeline proceeds through several stages, including unit testing, dependency scanning, artifact building, and vulnerability code scanning. All these assessments are performed on both the newly added code and the existing code from

the main branch.

If any of these tests fail, the developer is asked to make necessary adjustments and commit the changes to the same pull request. This action triggers the CI pipeline once again. If there are no failures this time, the pull request is approved and merged into the main branch.

Upon merging into the main branch, the same CI pipeline, or possibly a different one with additional steps and tests, is automatically executed to verify the merged code. At this point testing the same code again after merging may seem redundant, but I will try to clarify this by the end of this video.



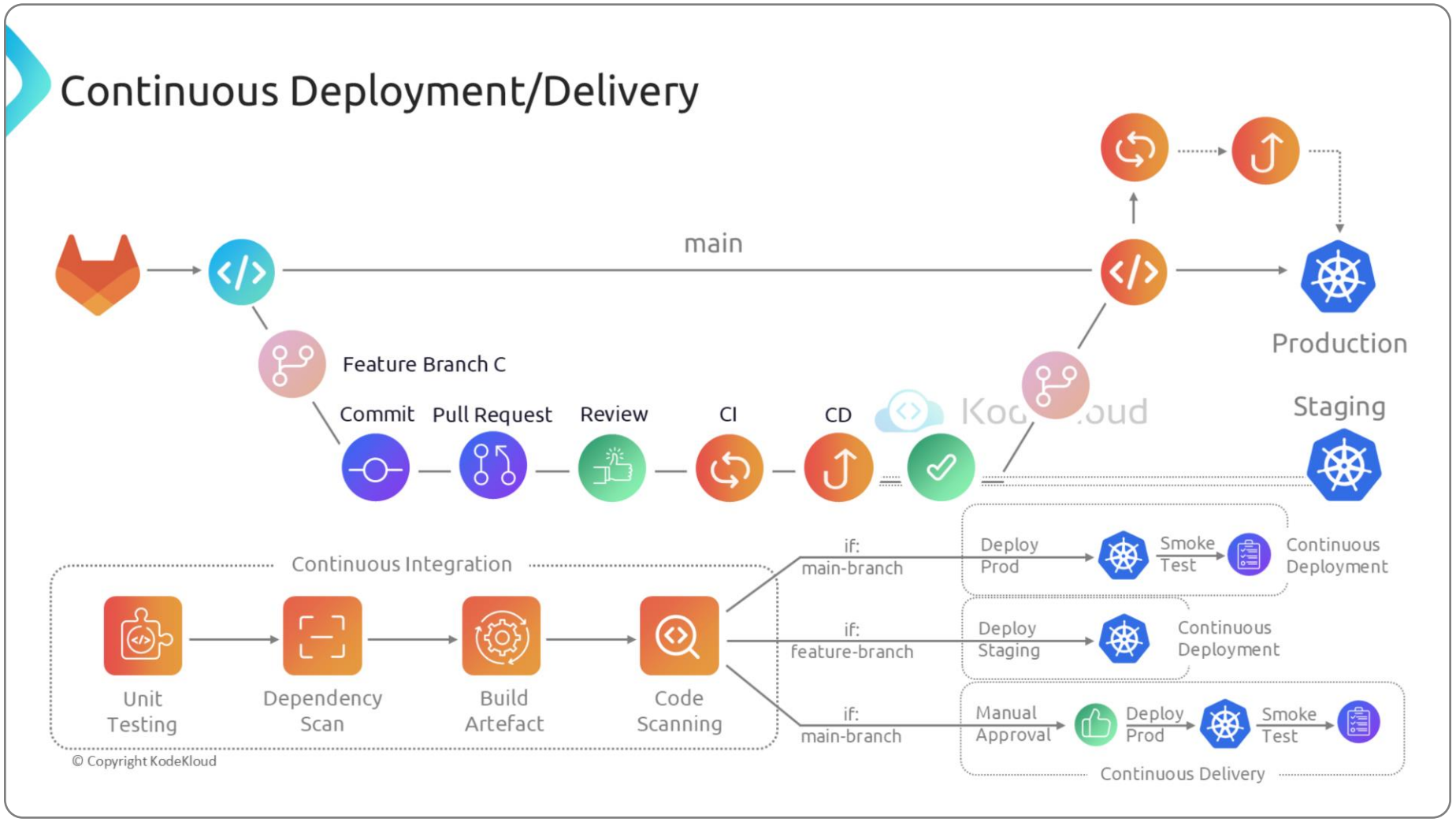
While these developments are taking place in feature-branch A, Developer 2 is progressing on feature branch B. They commit their changes and create a pull request. An automated CI pipeline is initiated, testing the code committed in this branch. Once the CI pipeline successfully completes, the pull request is approved and merged into the main branch.

Following the successful merge, the main branch now contains code changes from both feature-branch A and B.

As mentioned earlier, any merges into the main branch automatically trigger a CI pipeline. In this instance, it once again runs

all the tests, including unit testing, dependency scanning, artifact building, and vulnerability scans. This ensures that the code changes from both feature-branch B and A work seamlessly together.

This entire process, which enables multiple developers to work on the same application while ensuring that these new changes integrate smoothly without introducing any new issues, is known as continuous integration.



So far, we have explored the concept of continuous integration. Now, let's check into the CD aspect, which encompasses continuous deployment and/or continuous delivery.

In previous instances, after code integration into the main branch and successful completion of the CI pipeline, manual deployment to the production environment was performed.

In many scenarios, even after rigorous CI and testing procedures, it is advisable to deploy the modified application to a non-

production environment that closely resembles the live environment. This allows for live testing before proceeding with production deployment.

Within the feature branch, following a successful CI pipeline run, we can establish another continuous deployment pipeline. This pipeline is responsible for deploying the modified code to a staging or development environment. Following deployment, a series of tests are executed to ensure the quality of the application.

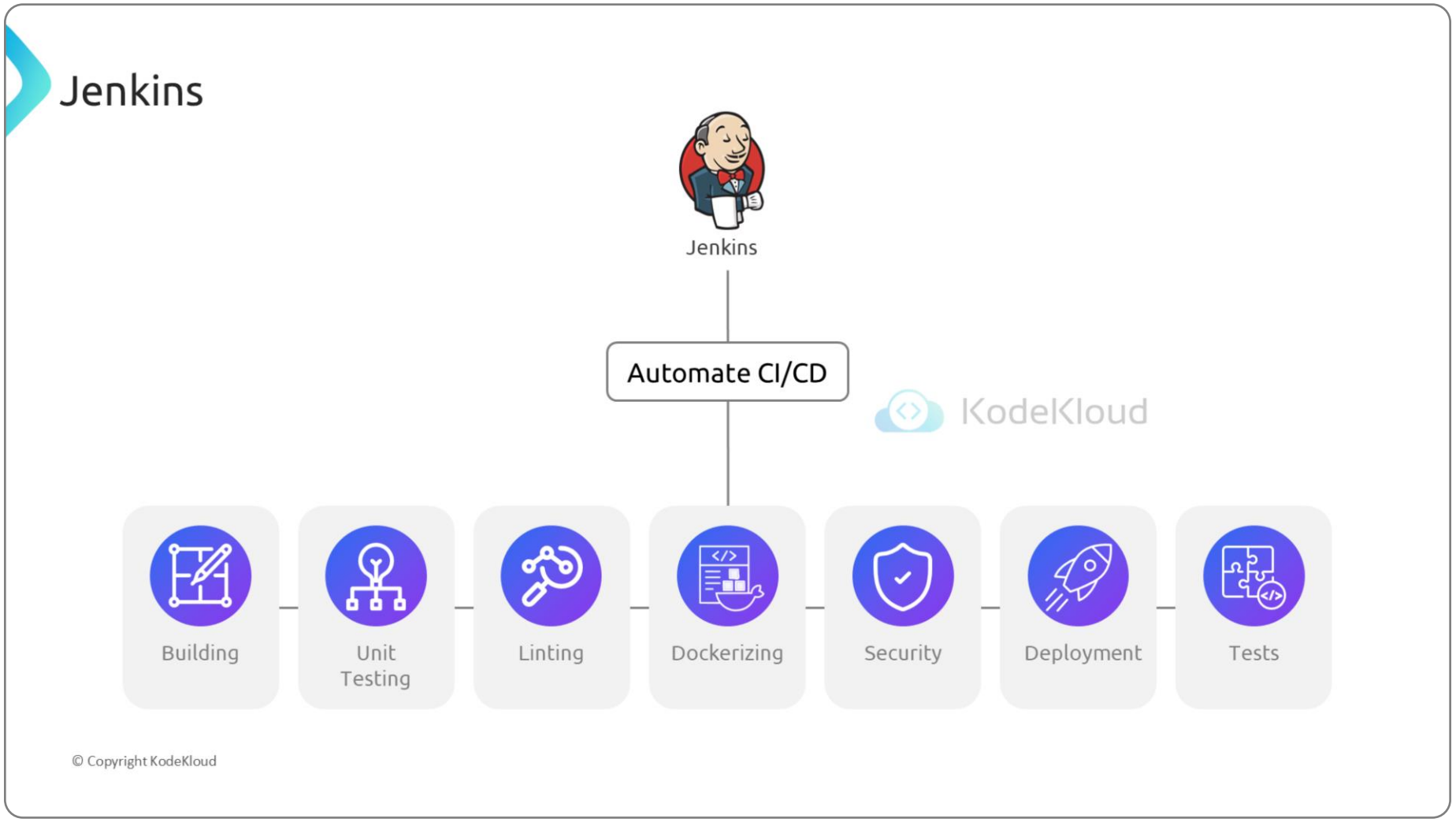
Upon successful completion of CD, the pull request is approved and merged back into the main branch.

Within the main branch, the CI pipeline is triggered, assessing the newly merged changes. If successful, it automatically initiates the CD pipeline, resulting in the deployment of the application to the production environment. This automatic deployment process following successful continuous integration is referred to as continuous deployment.

In certain instances, a manual approval step before production deployment is a critical aspect of the deployment process. This step serves to minimize risks, ensure quality, adhere to compliance requirements, and effectively coordinate changes. It offers a safety net and allows for human judgment and oversight within an otherwise automated process.

In this scenario, following the successful completion of the CI pipeline, the CD pipeline awaits human approval before proceeding with the production deployment. This process of manual approval prior to production deployment is known as continuous delivery.

Introducing Jenkins

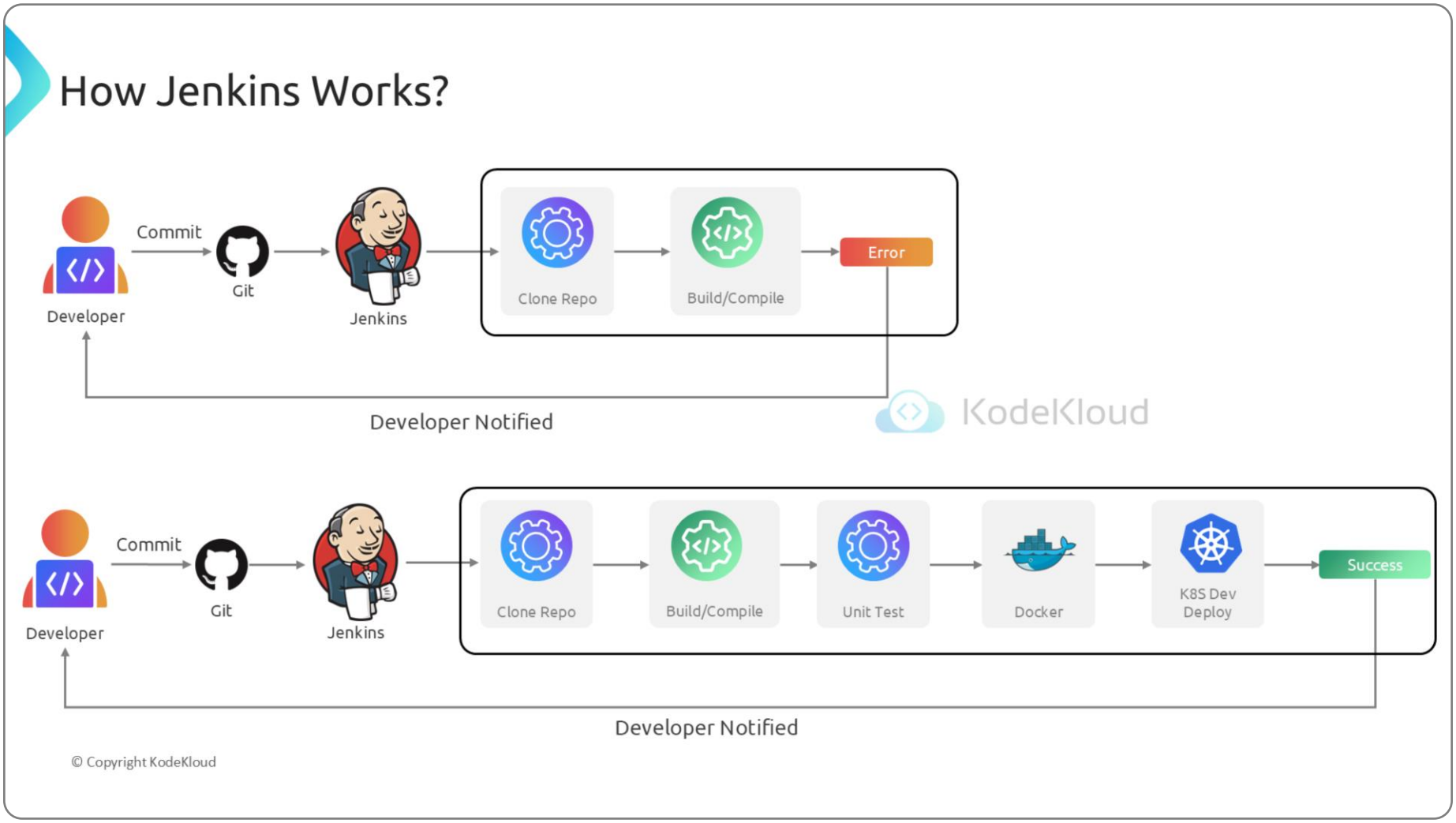


Imagine a world where every code change triggers a flurry of automated tasks: building your application, running tests, and even deploying it. That's the magic of Jenkins in the DevOps world.

Jenkins is a Open-source CI/CD server. It's a free and customizable software that automates the software delivery process.

It orchestrates different stages in your development lifecycle, like building, testing, containerization, and deploying your code.

As a continuous integration tool, Jenkins helps development teams identify errors in the early stages of a project, and automate the integration process of new code.



Let's dive into how Jenkins works with a simple example:

Developers regularly push their code updates to a Git repository
Jenkins keeps an eye on this repository and whenever there's a change, it automatically clones the repo and compiles the code.
If the compilation fails, Jenkins alerts the developer.
Developer makes code changes and again pushes them to Git.

Jenkins detects the update and compiles the code.
If the build succeeds (and any other stages), Jenkins automatically deploys the code to production and the developer gets a notification about the successful deployment.
This loop keeps your code fresh in production with minimal manual intervention.

As you make further changes and push them to Git, the process repeats, ensuring a smooth and automated development workflow.

Jenkins Core Concepts

Jobs
Define specific tasks such as compiling code, running tests, or deploying to an environment.

Builds
Each execution of a job, with Jenkins maintaining a history for troubleshooting.

Freestyle Project
The default project type, offering flexible job definitions

Pipelines
Chain jobs together to automate tasks from development to deployment.

Before we jump in, here are some key terms you'll need to know when working with Jenkins for CI/CD.

Jobs: The building blocks of your pipeline. Each job defines a specific task, like compiling code, running tests, or deploying to a specific environment.
Builds: Each execution of a job is considered a build. Jenkins maintains a build history, allowing you to track past runs (successful or failed) for troubleshooting.

Freestyle Project: The default project type in Jenkins. It offers a flexible way to define build jobs using various steps and configurations.
Pipelines: Workflows created by chaining jobs together. This creates a seamless flow for your code, automating tasks from development to deployment. Pipelines can be written in code for better version control.
Stages (Pipeline Stages): Within a pipeline, you can group related jobs into stages. This provides a clear structure for your workflow, like "Build", "Test", and "Deploy", making it easier to visualize different phases.

Nodes: The machines where Jenkins executes jobs. You can have a single controller node (Jenkins server) coordinating builds or distribute tasks across multiple worker nodes (agents) for increased performance.
Plugins: Extensions that expand Jenkins' functionality. They allow seamless integration with various tools and technologies used throughout your development lifecycle. Need to connect to a specific testing framework or cloud platform? A plugin can handle that!

We've just scratched the surface! Throughout this Jenkins series of courses, we'll dive deeper into these concepts and many more, providing a comprehensive understanding of Jenkins for your CI/CD needs.

Jenkins Core Concepts

Stages

Group related jobs into phases like "Build", "Test", and "Deploy" for clarity.

Nodes

Machines where Jenkins executes jobs, with a single controller or multiple agents.

Plugins

Extend Jenkins' functionality, integrating with various tools and technologies.

Before we jump in, here are some key terms you'll need to know when working with Jenkins for CI/CD.

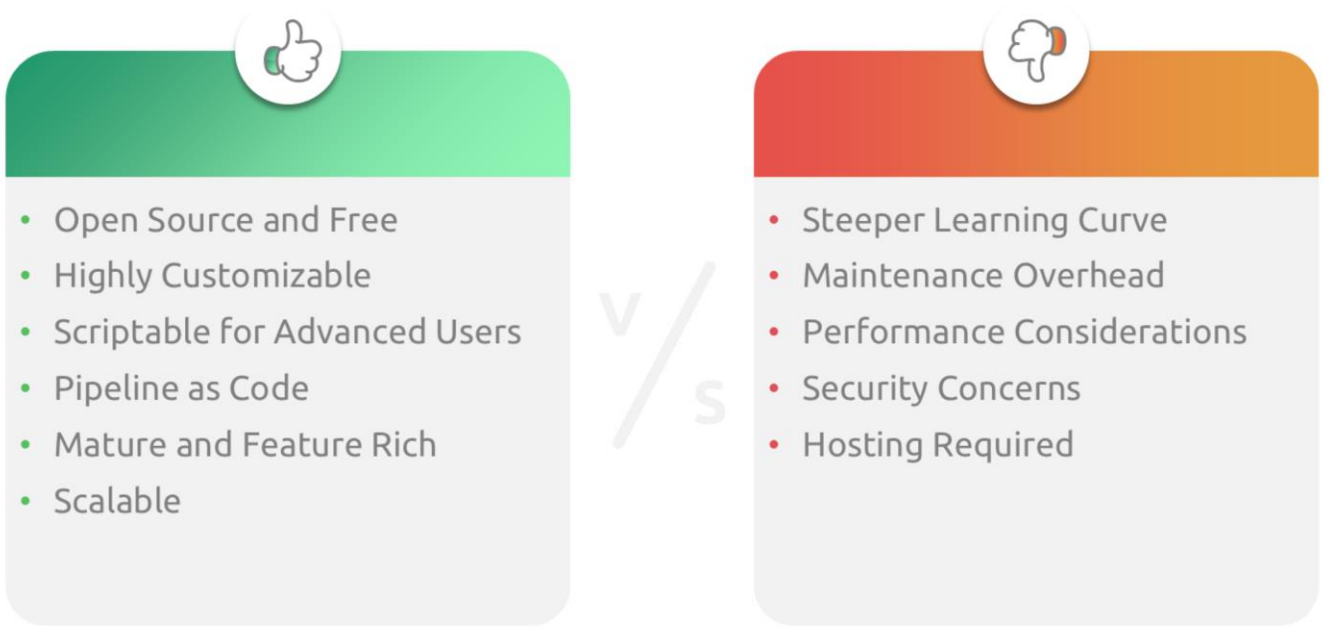
Jobs: The building blocks of your pipeline. Each job defines a specific task, like compiling code, running tests, or deploying to a specific environment.
Builds: Each execution of a job is considered a build. Jenkins maintains a build history, allowing you to track past runs (successful or failed) for troubleshooting.

Freestyle Project: The default project type in Jenkins. It offers a flexible way to define build jobs using various steps and configurations.
Pipelines: Workflows created by chaining jobs together. This creates a seamless flow for your code, automating tasks from development to deployment. Pipelines can be written in code for better version control.
Stages (Pipeline Stages): Within a pipeline, you can group related jobs into stages. This provides a clear structure for your workflow, like "Build", "Test", and "Deploy", making it easier to visualize different phases.

Nodes: The machines where Jenkins executes jobs. You can have a single controller node (Jenkins server) coordinating builds or distribute tasks across multiple worker nodes (agents) for increased performance.
Plugins: Extensions that expand Jenkins' functionality. They allow seamless integration with various tools and technologies used throughout your development lifecycle. Need to connect to a specific testing framework or cloud platform? A plugin can handle that!

We've just scratched the surface! Throughout this Jenkins series of courses, we'll dive deeper into these concepts and many more, providing a comprehensive understanding of Jenkins for your CI/CD needs.

Pro's and Con's

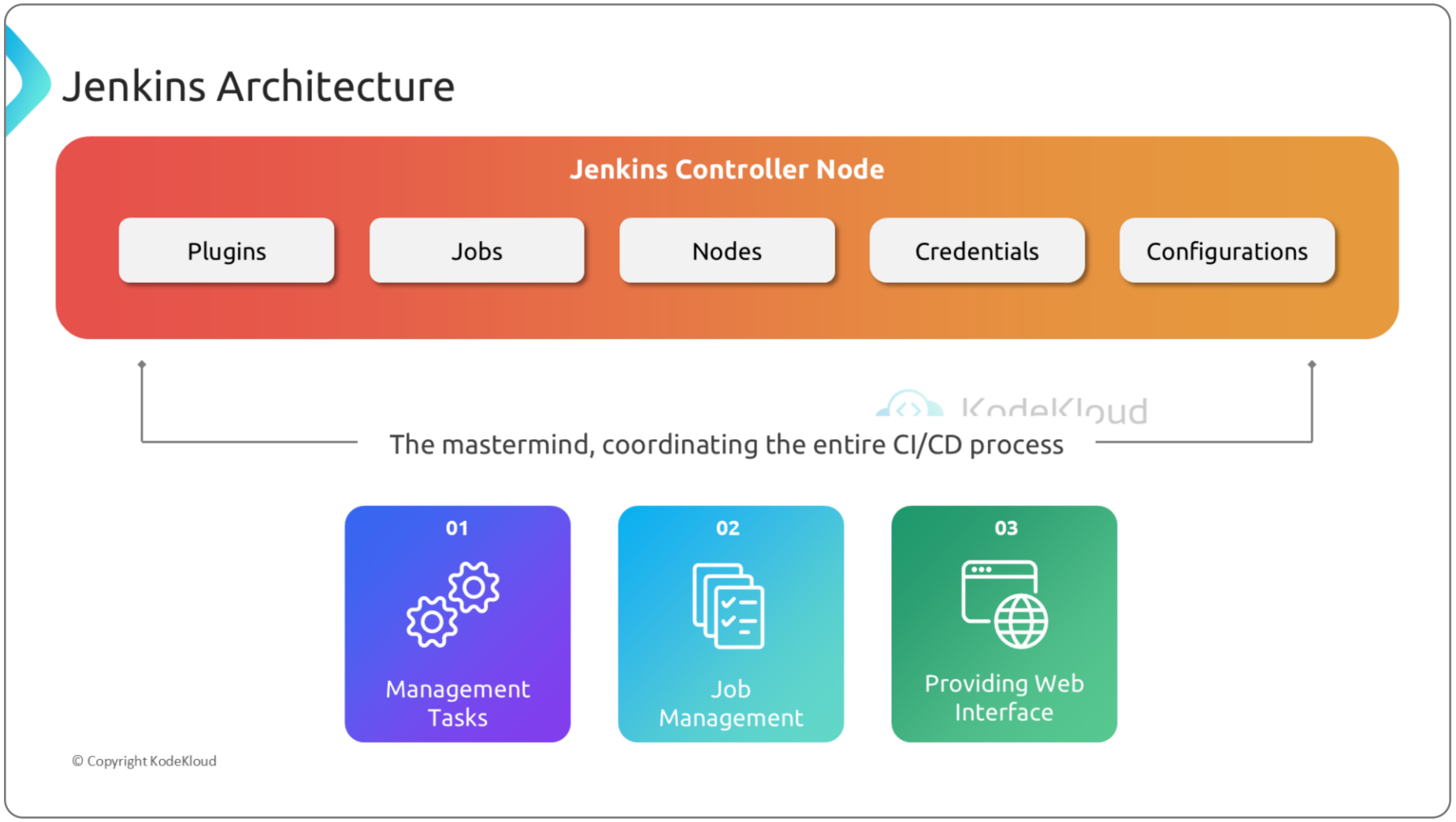


Pros:
Open Source and Free: Jenkins is a free and open-source tool, making it accessible to anyone without licensing costs. This also allows for a large and active community that contributes to its development and provides ample resources.
Highly Customizable: With a vast library of plugins, Jenkins can be tailored to almost any development environment. Need to integrate with a specific programming language, testing framework, or deployment target? There's likely a plugin for that! You can even write custom Groovy scripts for complex workflows.
Scriptable for Advanced Users: For those comfortable with scripting, Jenkins allows for advanced automation using Groovy.

This lets you write custom logic for complex workflows or integrate seamlessly with other tools.
Pipeline as Code: Jenkins supports "Pipeline as Code" which lets you define your CI/CD pipeline in code alongside your project. This promotes version control, collaboration, and easier management of your pipelines.
Mature and Feature Rich: As a well-established platform, Jenkins offers a wide range of features like build execution, parallel testing, artifact management, and detailed build reports for troubleshooting.
Scalable: Jenkins can be scaled horizontally by adding more nodes (agents) to handle increased workloads and support larger projects.

Cons:
Steeper Learning Curve: While Jenkins offers a user interface, it can be complex for beginners compared to some newer CI/CD tools. Understanding its configuration and vast plugin ecosystem can take time.
Maintenance Overhead: Frequent updates and a vast plugin library, while a strength, can also mean additional time is needed to keep Jenkins and its plugins up-to-date.
Performance Considerations: As project size and complexity grow, performance issues can arise with Jenkins, especially on a single server setup. Scaling with additional nodes can help mitigate this.
Security Concerns: Being open-source, security is a responsibility users need to take seriously. Regular updates, proper plugin selection, and following security best practices are crucial.
Hosting Required: Unlike some CI/CD tools like Github Actions or Gitlab CICD that are hosted, you need to host and configure Jenkins before using it.

Jenkins Architecture



Jenkins leverages a distributed architecture to manage your CI/CD pipelines efficiently. This structure offers scalability and power, allowing you to automate complex workflows across multiple machines. Here's a breakdown of the key players:

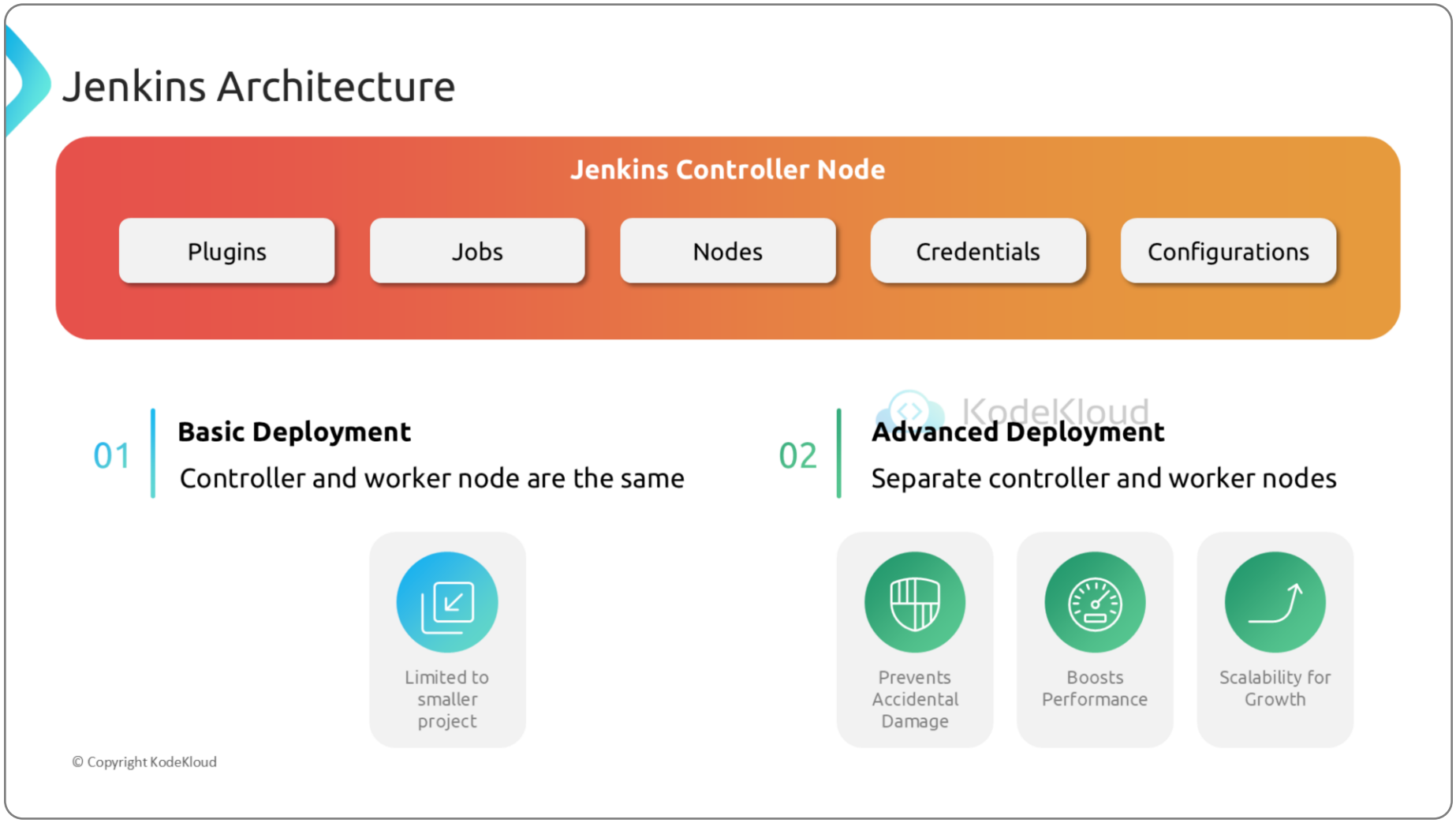
1. Jenkins Controller (formerly known as Master Node):
The central hub of your Jenkins setup, often referred to as the Jenkins server.
Acts as the mastermind, coordinating the entire CI/CD process.
Responsibilities include:

- Management tasks such as user authentication and authorization are executed on the controller,
- Managing jobs and pipelines: Defining, scheduling, and monitoring their execution.
- Providing a web interface: For configuration, setting up plugins, onboarding users, monitoring builds, and managing Jenkins.

In basic deployments, the Jenkins server itself can handle everything, acting as both the controller and the sole worker (node). This can be a good starting point for smaller projects.

However, for most situations, it's recommended to have a dedicated controller and separate worker nodes. This offers several advantages:

- Prevents Accidental Damage: Keeps your controller configuration safe from potential job-related changes.
- Boosts Performance: Distributes build tasks across multiple machines, improving speed and efficiency, especially for complex projects.
- Scalability for Growth: Easily add more nodes as your project and automation needs increase.



Jenkins leverages a distributed architecture to manage your CI/CD pipelines efficiently. This structure offers scalability and power, allowing you to automate complex workflows across multiple machines. Here's a breakdown of the key players:

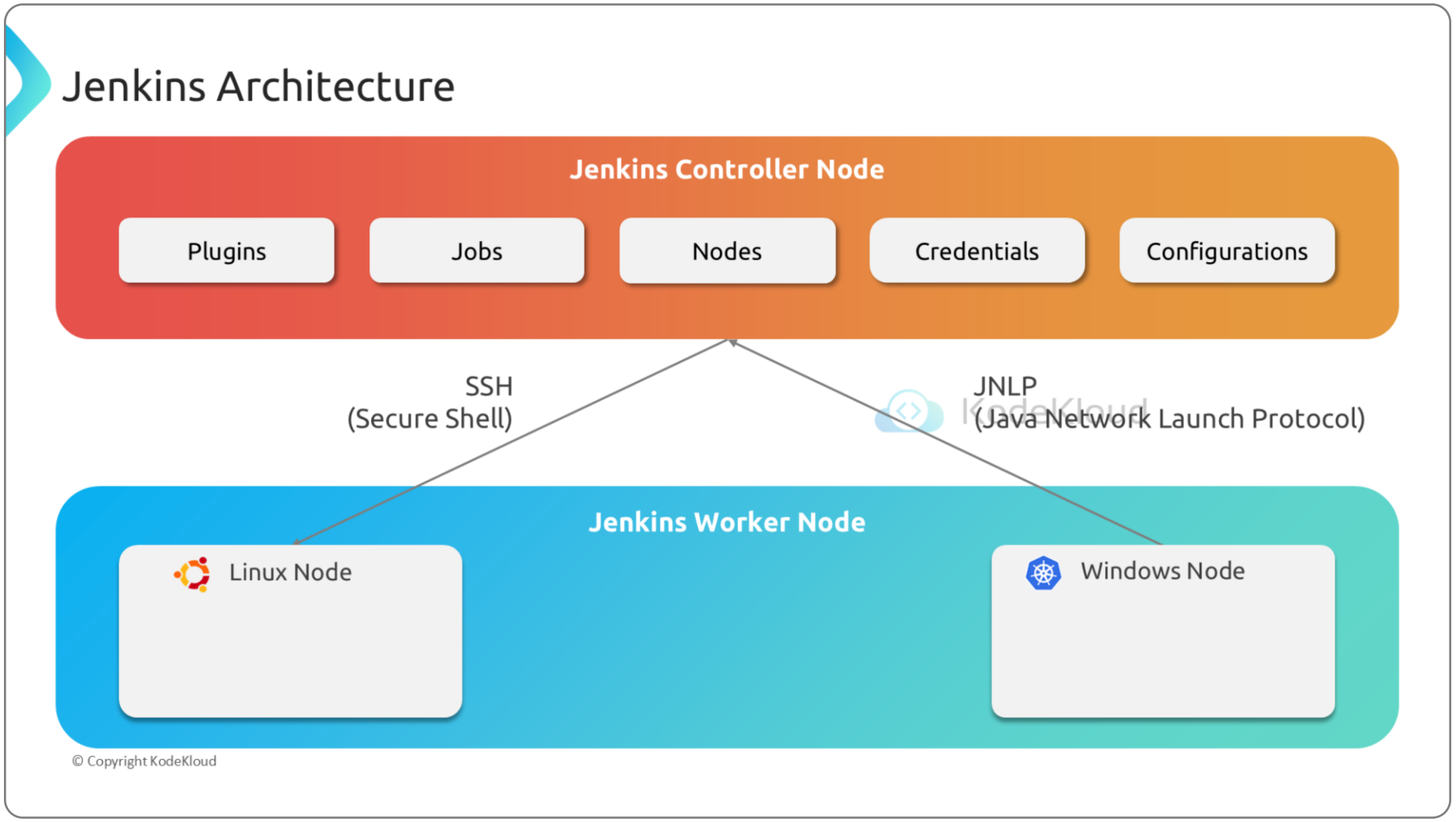
1. Jenkins Controller (formerly known as Master Node):
The central hub of your Jenkins setup, often referred to as the Jenkins server.
Acts as the mastermind, coordinating the entire CI/CD process.
Responsibilities include:

- Management tasks such as user authentication and authorization are executed on the controller,
- Managing jobs and pipelines: Defining, scheduling, and monitoring their execution.
- Providing a web interface: For configuration, setting up plugins, onboarding users, monitoring builds, and managing Jenkins.

In basic deployments, the Jenkins server itself can handle everything, acting as both the controller and the sole worker (node). This can be a good starting point for smaller projects.

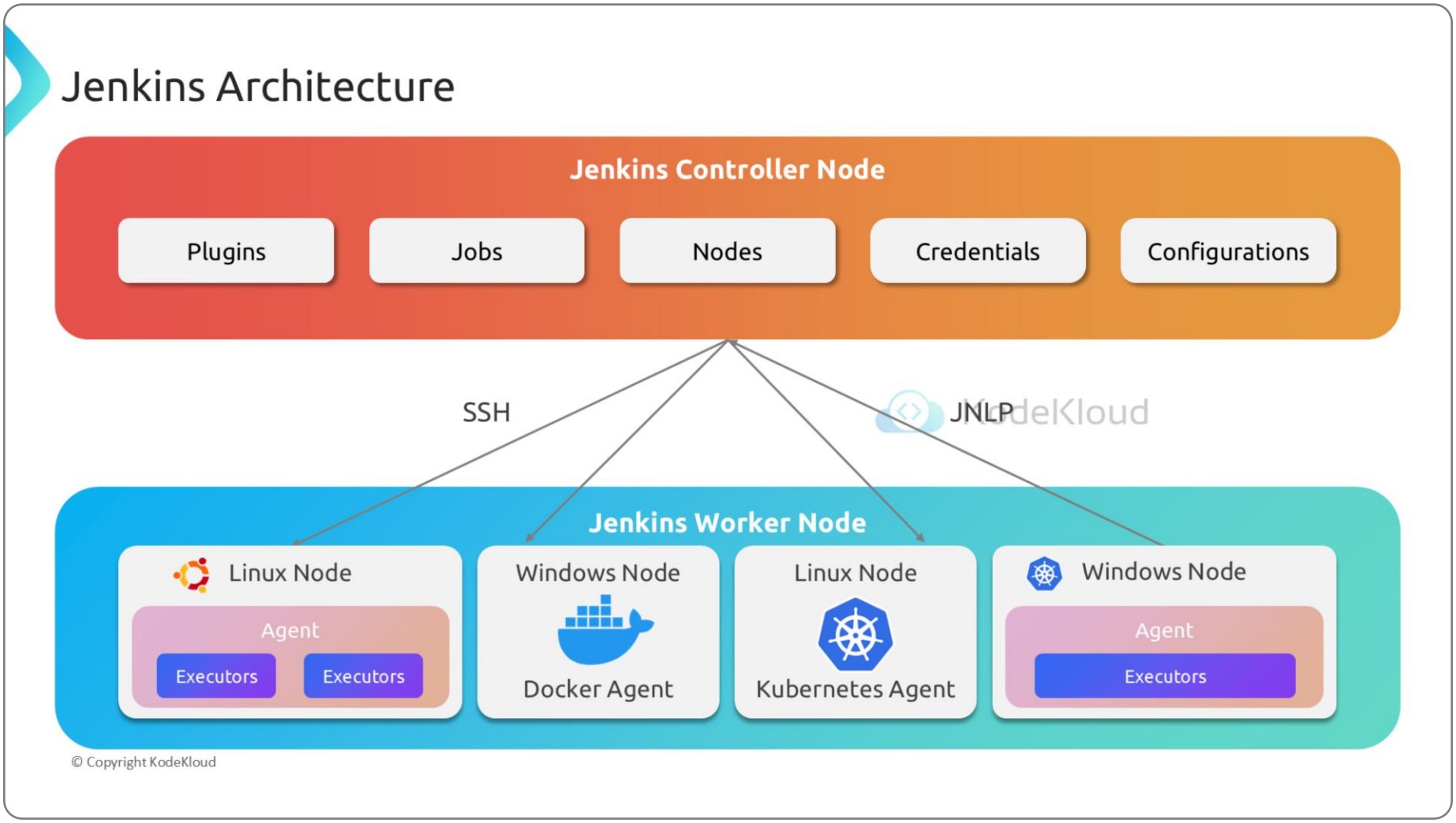
However, for most situations, it's recommended to have a dedicated controller and separate worker nodes. This offers several advantages:

- Prevents Accidental Damage: Keeps your controller configuration safe from potential job-related changes.
- Boosts Performance: Distributes build tasks across multiple machines, improving speed and efficiency, especially for complex projects.
- Scalability for Growth: Easily add more nodes as your project and automation needs increase.



So whats a node?

2. Nodes (Formerly known as Jenkins Slaves):
play a crucial role in your CI/CD workflow by serving as the worker machines that execute your build jobs.
Dedicated machines that handle build execution, allowing for parallel processing and increased performance for larger projects. They connect to the controller using protocols like SSH or JNLP (Java Network Launch Protocol).



3. Executors
Each node has a configurable number of executors, essentially slots for running build jobs concurrently.
An executor is a thread for execution of tasks. More executors allow for parallel processing of multiple jobs, speeding up your builds.
The number of executors assigned to a node is determined by the node's available resources (such as CPU and memory) and the requirements of your build tasks.

Allocating one executor for each node is the most secure setup.

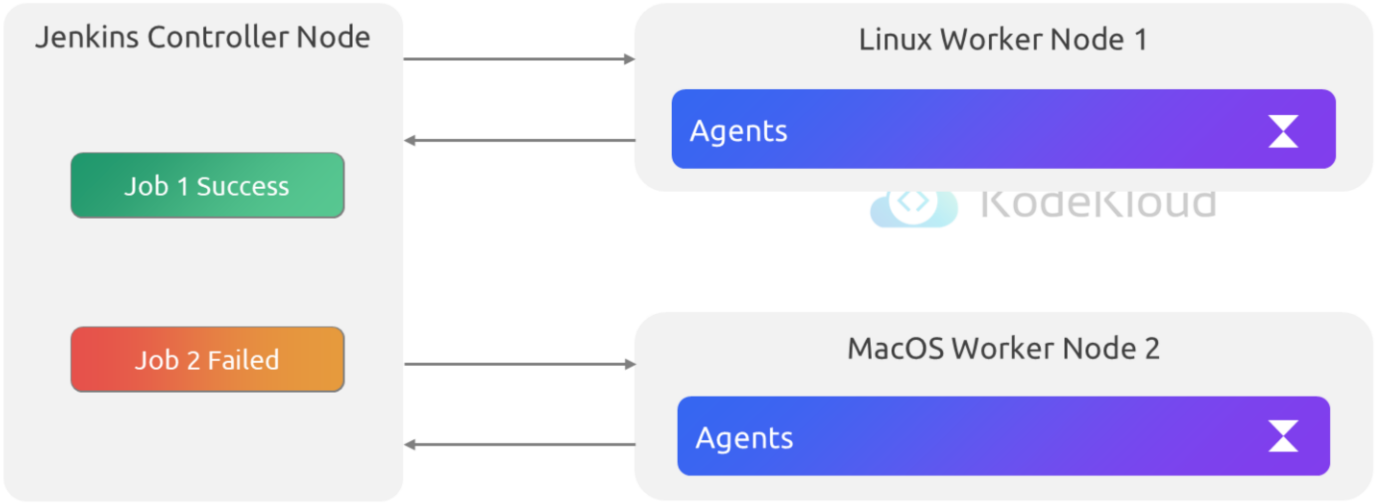
If the tasks to be executed are small, assigning one executor per CPU core could be an effective approach.

4. Agents:
The agent acts as an extension of Jenkins, managing the task execution by using executors on a remote node.
An agent represents a specific way a node connects to the controller. It defines the communication protocol and authentication mechanism used.
For example, you might have a Java agent connecting using JNLP or a SSH agent connecting using the SSH protocol.
Tools required for builds and tests are installed on the node where the agent runs;

Docker agent
We can leverage Docker containers as build agents.
Instead of installing tools and dependencies directly on a dedicated worker node, you define a Docker image containing all the necessary software for your build jobs.
This is Well-suited for scenarios where build jobs require specific software versions or have complex dependencies.
Finally each build job runs in a clean, isolated container, ensuring consistent environments and preventing conflicts between projects.

Similarly we can Utilize Kubernetes clusters to dynamically provision and manage build agents (nodes) as pods. This allows for on-demand scaling and efficient resource utilization.

Putting it All Together



Putting it All Together:
Lets assume we have a controller node connected to two worker nodes.

You define jobs and pipelines within the controller.
The controller identifies available executors on connected nodes (agents).
The controller schedules and distributes jobs to available executors on nodes.
Nodes (agents) execute the jobs using their allocated resources.

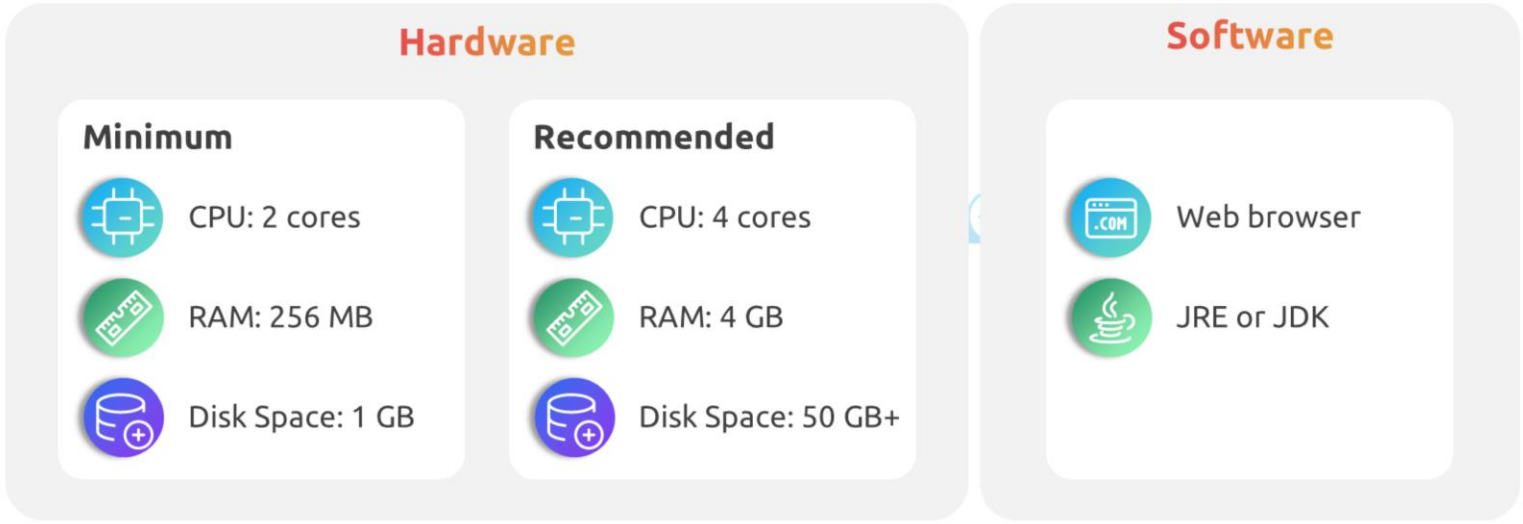
The controller receives results and feedback from the nodes, providing build status and logs.

This distributed architecture allows you to scale your Jenkins setup by adding more nodes with additional executors. This way, you can handle complex workflows and larger projects efficiently, ensuring smooth and automated CI/CD pipelines.

Jenkins Installation

To get started with Jenkins and build your CI/CD pipeline, you'll need to consider installation options, hardware requirements, and software dependencies.

Hardware and Software Requirement



The hardware required for your Jenkins server will depend on the complexity of your CI/CD pipelines and the number of jobs you plan to run concurrently. Here's a general guideline:

Minimum:

- CPU: 2 cores
- RAM: 256 MB (although 1 GB is recommended)
- Disk Space: 1 GB

Recommended (for small teams):

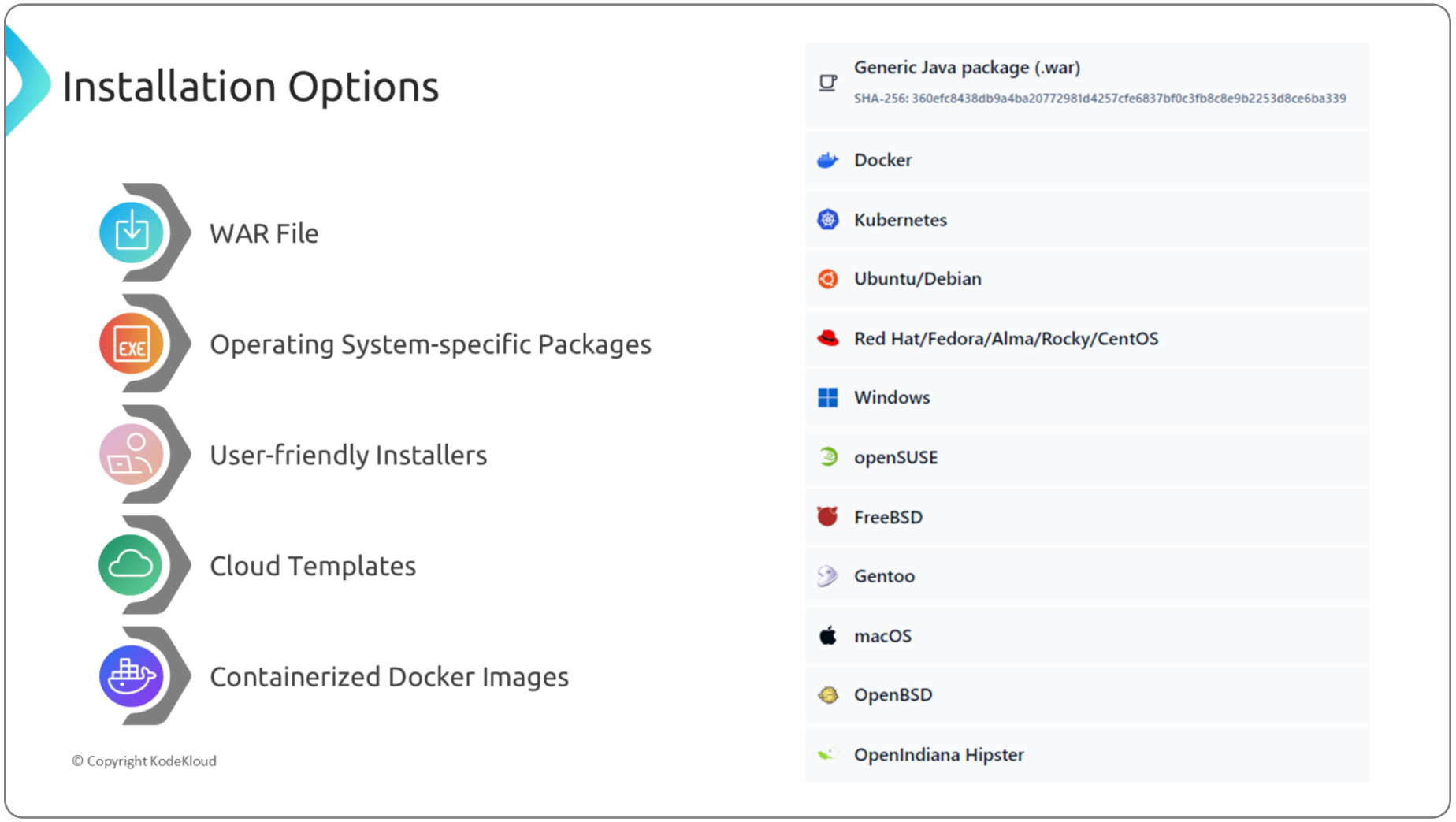
CPU: 4 cores
RAM: 4 GB
Disk Space: 50 GB+

Talking about the Software Requirements:
Jenkins relies on the Java Virtual Machine (JVM) to run. To ensure smooth operation, you'll need either the Java Development Kit (JDK) or the Java Runtime Environment (JRE) installed on your server.

JRE: The basic option for running Jenkins itself.

JDK (Recommended): While JRE is sufficient, the JDK provides additional tools for troubleshooting and developing custom Jenkins plugins. This makes it the preferred choice for production environments.

Web Access: A web browser is all you need to interact with and manage your Jenkins instance once it's up and running.



Jenkins offers several ways to get up and running, catering to different preferences and environments:

You can install Jenkins in various ways, including web application archives (WAR files), operating system-specific packages, user-friendly installers, cloud templates and containerized Docker images.

WAR File: Jenkins can can be started on any operating system or platform with a version of Java supported by Jenkins by downloading the jenkins.war file.

Operating System-specific Packages: Jenkins provides native packages for various operating systems like Windows, macOS, and various Linux distributions (Debian, Ubuntu, Red Hat, etc.). These packages handle dependencies and offer a more integrated system experience.

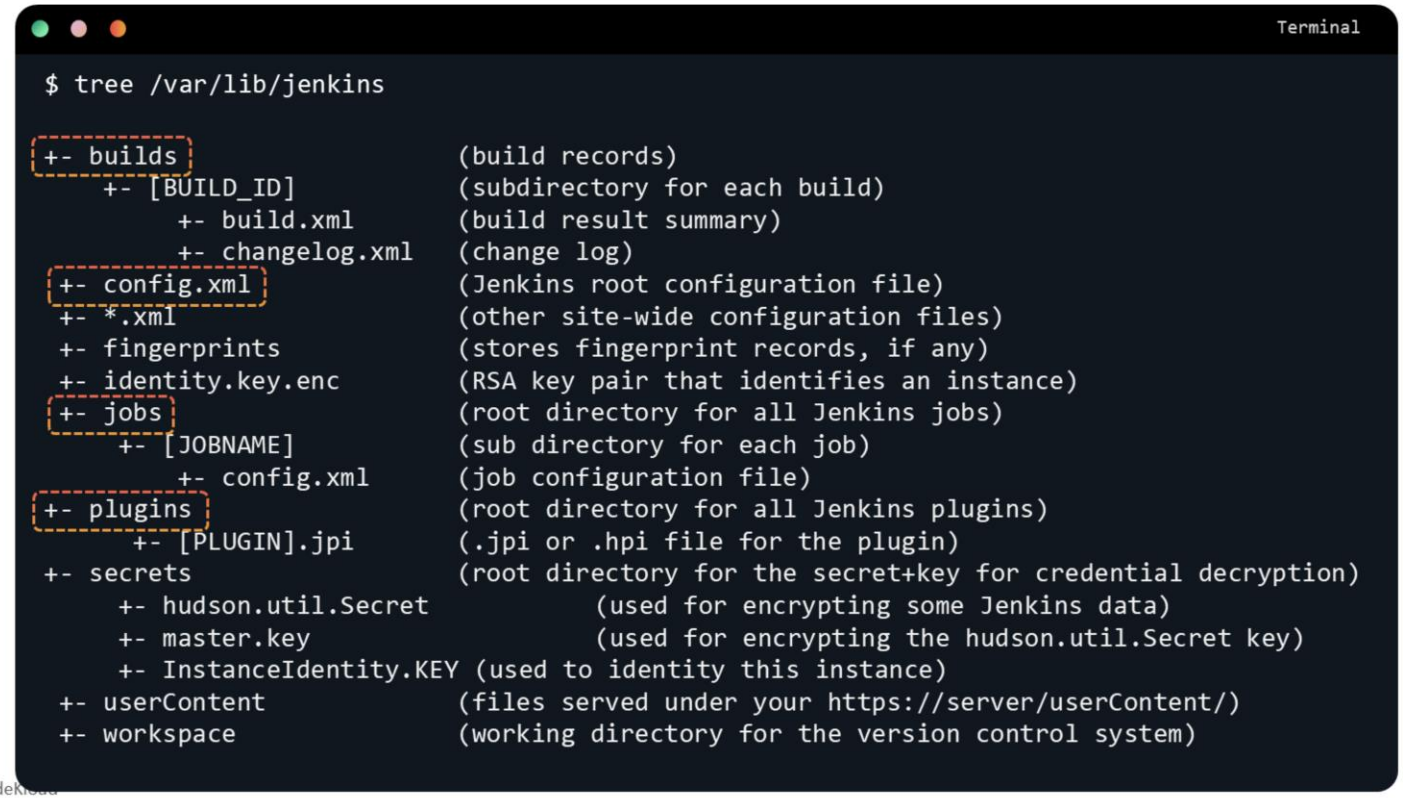
User-friendly Installers: For Windows and macOS, Jenkins provides a graphical installer, which is a quick and intuitive way for setting up Jenkins, especially for those not comfortable with command-line installations.

Cloud Templates: When utilizing cloud services such as AWS, Azure, or GCP, many of them provide Jenkins as a managed service. This enables the rapid setup and administration of Jenkins instances on their cloud platforms. Pre-designed templates are available for deploying Jenkins, often pre-loaded with configurations that adhere to industry best practices.

Containerized Docker Images: Jenkins also provides official Docker images. This is a great way to run Jenkins in a containerized environment, and it’s especially useful for creating reproducible build environments.

Remember, the best installation method depends on your specific needs and environment.

JENKINS_HOME



The $JENKINS_HOME directory is a crucial part of a Jenkins setup, which is where Jenkins stores all its data and configurations. This includes job plugins, configurations, build logs, artifact archives, and other metadata related to the Jenkins server.

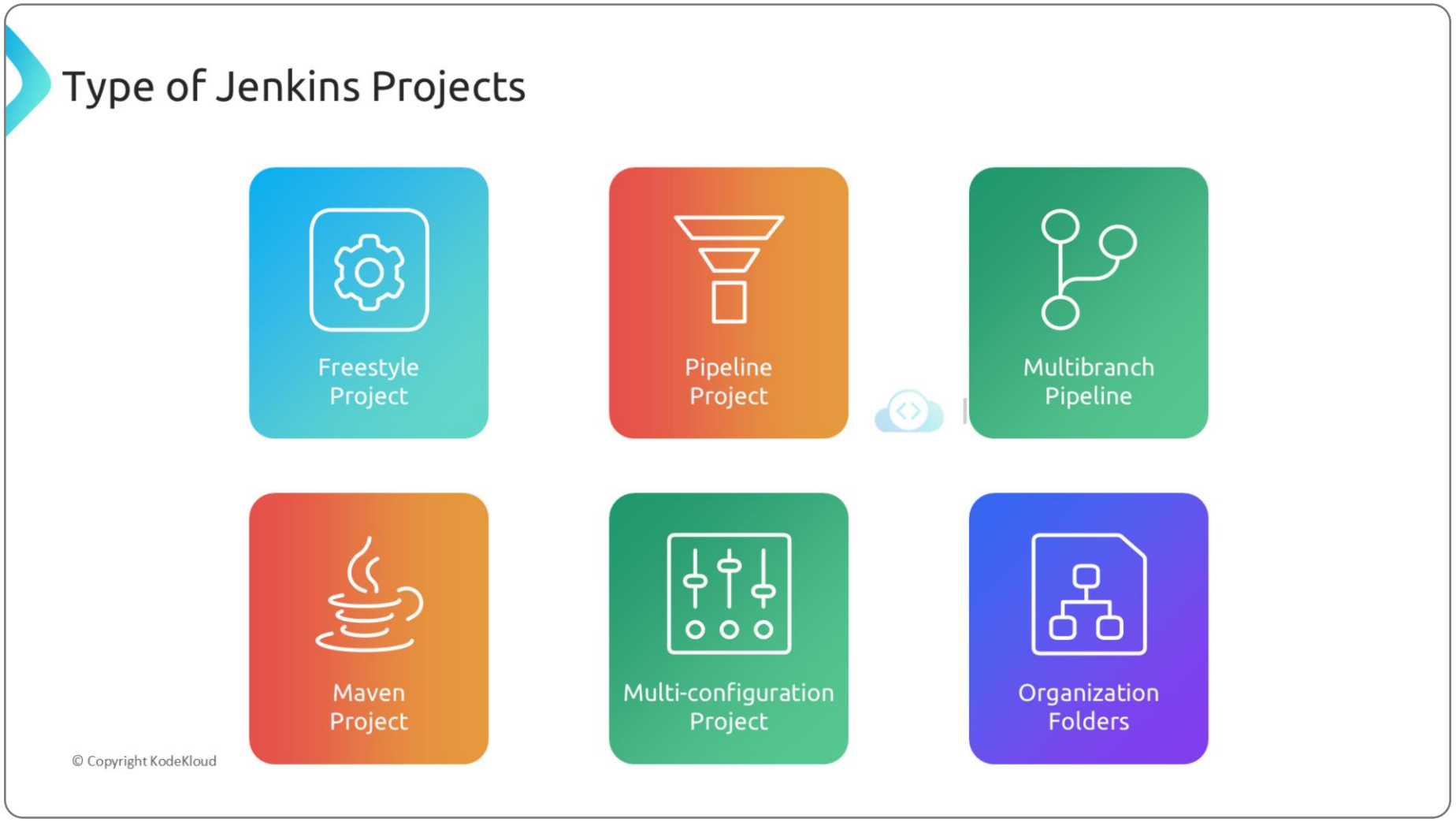
The location of $JENKINS_HOME varies depending on the installation method and the operating system:

If you’re running Jenkins from the .war file, $JENKINS_HOME defaults to the .jenkins directory within the home directory of the user running Jenkins.
If you’ve installed Jenkins using system packages on Linux, $JENKINS_HOME is usually set to /var/lib/jenkins.

You can also customize the location of $JENKINS_HOME by setting the JENKINS_HOME environment variable to your preferred directory path.

Remember, it’s important to regularly back up $JENKINS_HOME because it contains all the data and settings for your Jenkins server. If this directory is lost, you would lose your entire Jenkins setup.

Type of Jenkins Projects



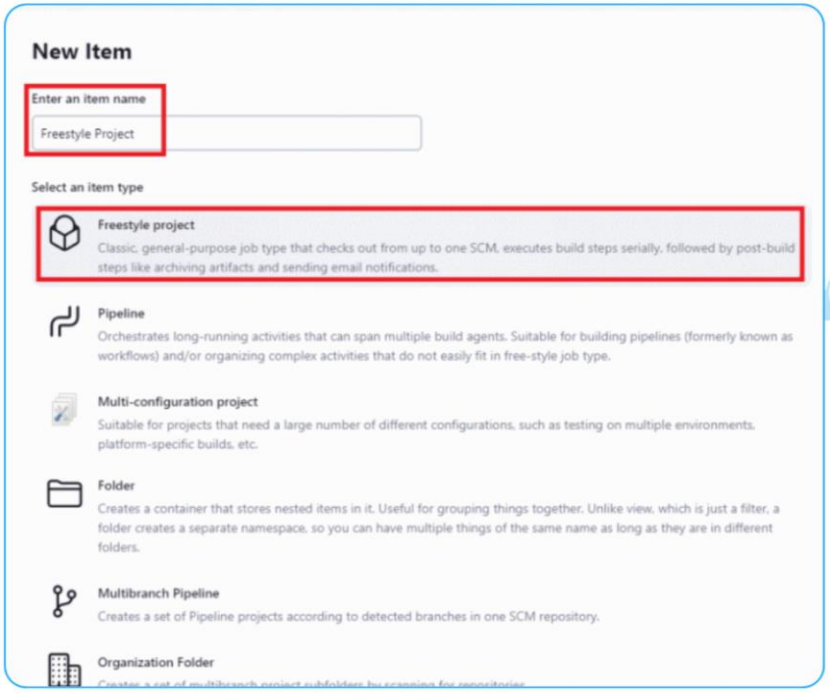
Jenkins offers various project types, also known as jobs, to automate different workflows. Here are the main ones:

Freestyle Project: This is the most basic and flexible option. It allows you to define custom build steps and configurations for any project, making it suitable for simple or unique needs.
Pipeline Project: This powerful approach uses a specific code (domain-specific language) to define the entire build pipeline. It includes stages, steps, and conditions for complex build processes and continuous delivery (CD) workflows.
Multibranch Pipeline: This builds upon Pipeline projects by allowing you to manage builds from multiple branches in a single

code repository. It's ideal for handling projects with frequent branching and development activities.
Maven Project: This streamlines building projects that use Maven, a popular build automation tool for Java projects. Jenkins automatically detects and executes Maven commands based on the project's pom.xml file.
Multi-configuration Project (Matrix): This lets you run the same build process with different configurations. You can define various combinations of parameters and run tests or builds for each combination.
Organization Folders: These aren't project types themselves, but rather a way to organize your Jenkins projects into a hierarchical structure for better management, especially when dealing with a large number of projects.

In addition to these core types, Jenkins may offer other project options depending on the plugins you have installed.

Freestyle Projects



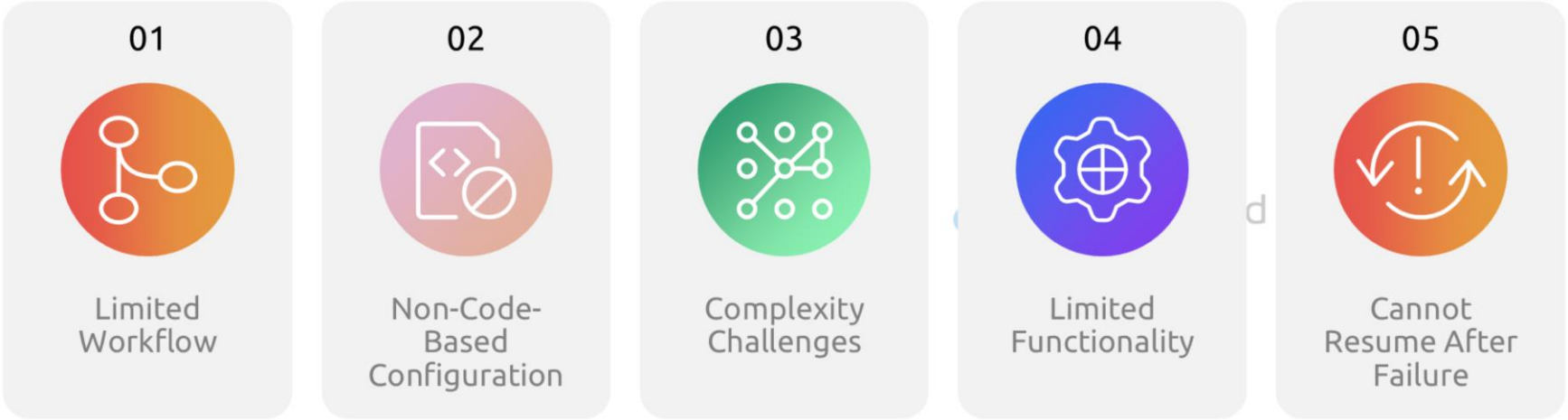
Freestyle projects (also known as jobs) have been a traditional way to define build workflows in Jenkins since its early days. It allows you to define a series of steps to be executed in a sequential order for automating tasks like building, testing, and deploying your project. Here projects are chained together.

Imagine building a mobile app. A Freestyle project in Jenkins would be like a simple to-do list for the build process: clone code, run tests, deploy the app. It's straightforward for basic tasks.

But what if you're building a complex app with multiple features, or need to deploy to different environments?

While still supported, Freestyle projects are generally not recommended due to limitations:

Freestyle Projects



- **Limited Workflow:** Freestyle projects can only define steps in a simple sequence, even though they can be linked together (upstream/downstream jobs).
- **Non-Code-Based Configuration:** Configuration happens entirely through the graphical user interface (GUI), making it less flexible and centrally manageable.
- **Complexity Challenges:** Modern continuous delivery (CD) pipelines often involve complex workflows that Freestyle projects struggle to handle. Even achievable tasks might result in clunky, hard-to-maintain, and unclear configurations.
- **Limited Functionality:** Many CD goals are simply not possible or become overly complex with Freestyle projects.

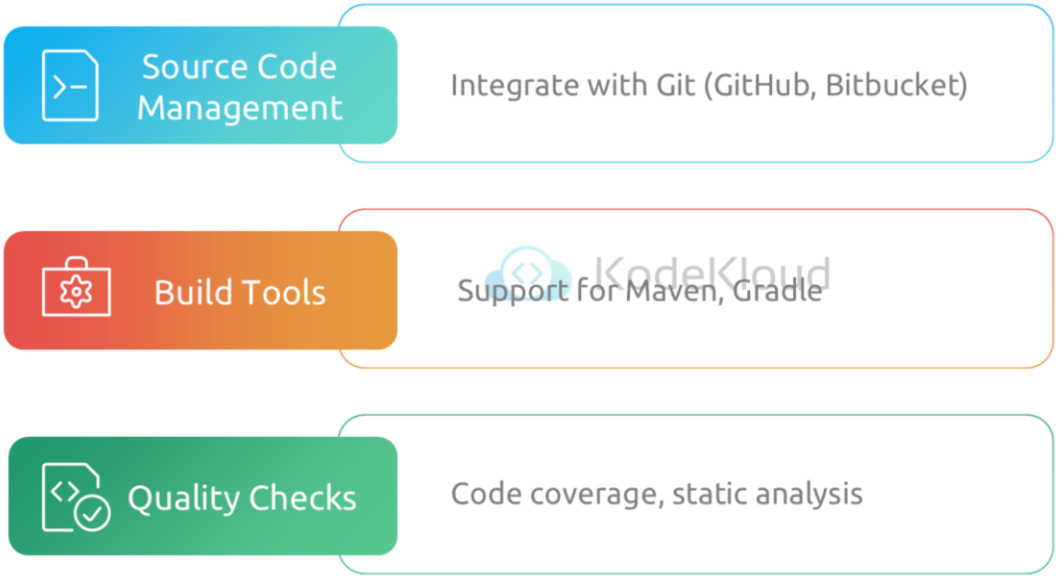
- **Freestyle project cannot be resumed after controller failure:**

In essence, Freestyle projects are a legacy approach. While they might work for basic cases, they lack the flexibility and maintainability needed for most modern CD pipelines.

Plugins

Plugins

Jenkins



Jenkins as a powerful engine, but it needs the right tools to handle specific tasks in your CI/CD pipeline. That's where plugins come in – they're like extensions that supercharge Jenkins with functionalities for various tools and technologies you use in your development process. With over 1900+ community-developed plugins available, Jenkins offers immense flexibility to adapt to your specific development workflow.

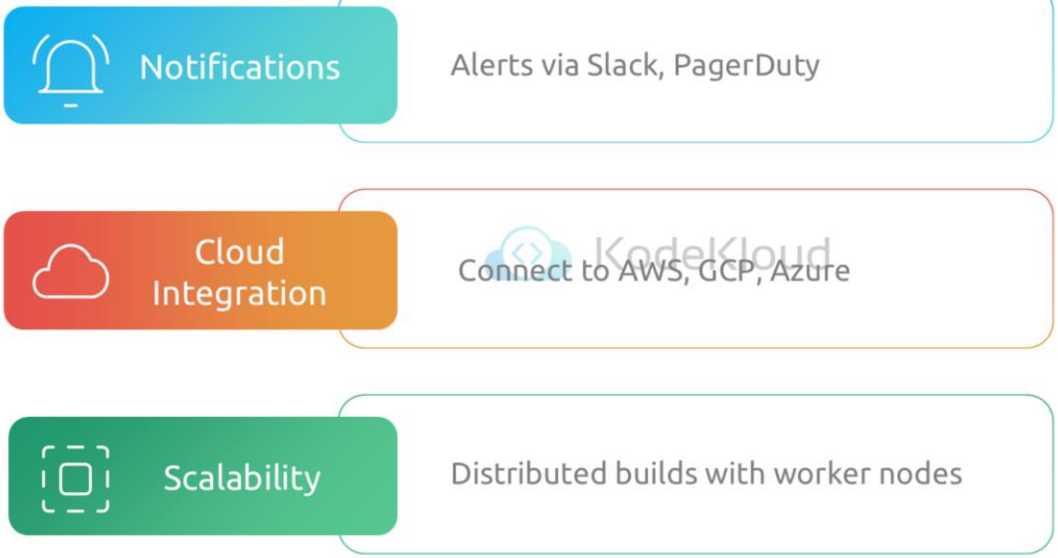
Here are some examples of functionalities that are provided as plugins:
Source code management plugins connects with Git systems like Github, Bitbucket

Build plugins Streamline your build process for popular build tools like Maven and Gradle.
Leverage plugins for code coverage analysis, static code analysis, and other quality checks.
Stay informed with notification tools like Slack or PagerDuty that send alerts upon build completion or failures.
Cloud plugins are used to Connect Jenkins to cloud platforms like AWS, GCP, or Azure for easy deployment and resource management.
Scale your CI/CD by utilizing plugins to connect with worker nodes (slave nodes) for distributed builds

.

Plugins

Jenkins



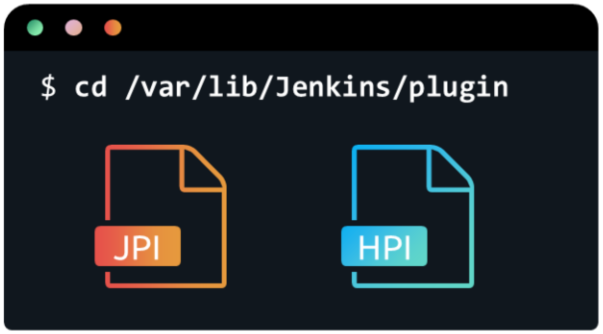
Jenkins as a powerful engine, but it needs the right tools to handle specific tasks in your CI/CD pipeline. That's where plugins come in – they're like extensions that supercharge Jenkins with functionalities for various tools and technologies you use in your development process. With over 1900+ community-developed plugins available, Jenkins offers immense flexibility to adapt to your specific development workflow.

Here are some examples of functionalities that are provided as plugins:
Source code management plugins connects with Git systems like Github, Bitbucket

Build plugins Streamline your build process for popular build tools like Maven and Gradle.
Leverage plugins for code coverage analysis, static code analysis, and other quality checks.
Stay informed with notification tools like Slack or PagerDuty that send alerts upon build completion or failures.
Cloud plugins are used to Connect Jenkins to cloud platforms like AWS, GCP, or Azure for easy deployment and resource management.
Scale your CI/CD by utilizing plugins to connect with worker nodes (slave nodes) for distributed builds

.

Plugins

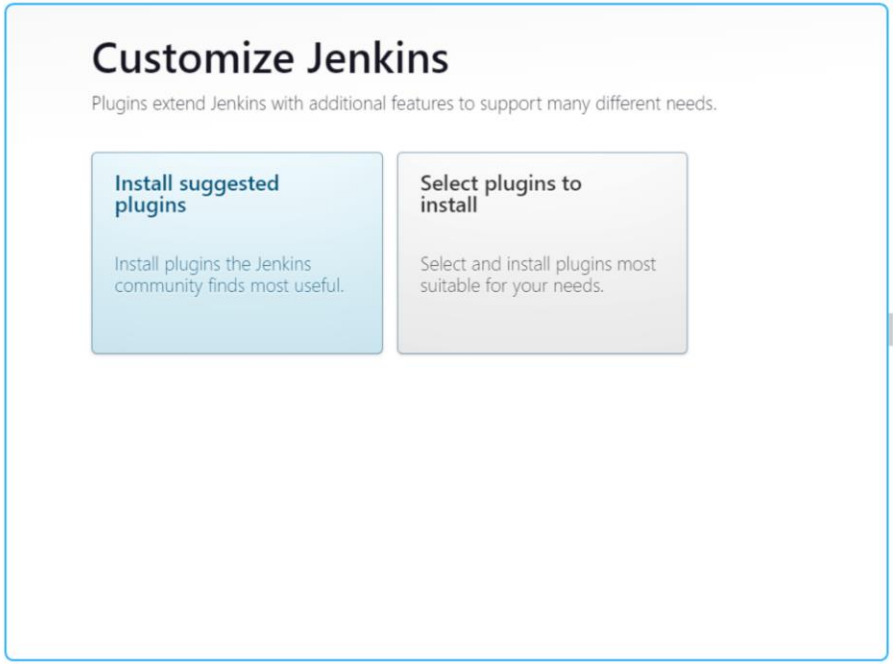


- **Formats:** .hpi (Hudson PlugIn), .jpi (Jenkins PlugIn)
- **Storage:** Located in ${jenkins_home}/plugins directory
- **Content:** Code, resources, configuration
- **Priority:** .jpi takes priority over .hpi if duplicates exist

Jenkins plugins are essentially JAR files. They come in a file with an hpi or jpi extension and stored in the ${jenkins_home}/plugins directory
"hpi" stands for Hudson PlugIn and is historical. "jpi" stands for Jenkins PlugIn.
Both .hpi and .jpi files are archive formats used to distribute Jenkins plugins. They contain the code, resources, and configuration information needed for the plugin to function within Jenkins.
If you have duplicate plugins with different extensions, the .jpi version takes priority.

Suggested Plugins When you first install Jenkins, it suggests a list of plugins that are widely used and provide fundamental capabilities to Jenkins. These include plugins for Git, Gradle, and Maven, among others. It’s a good idea to install these suggested plugins if you’re not sure what you need at the start.

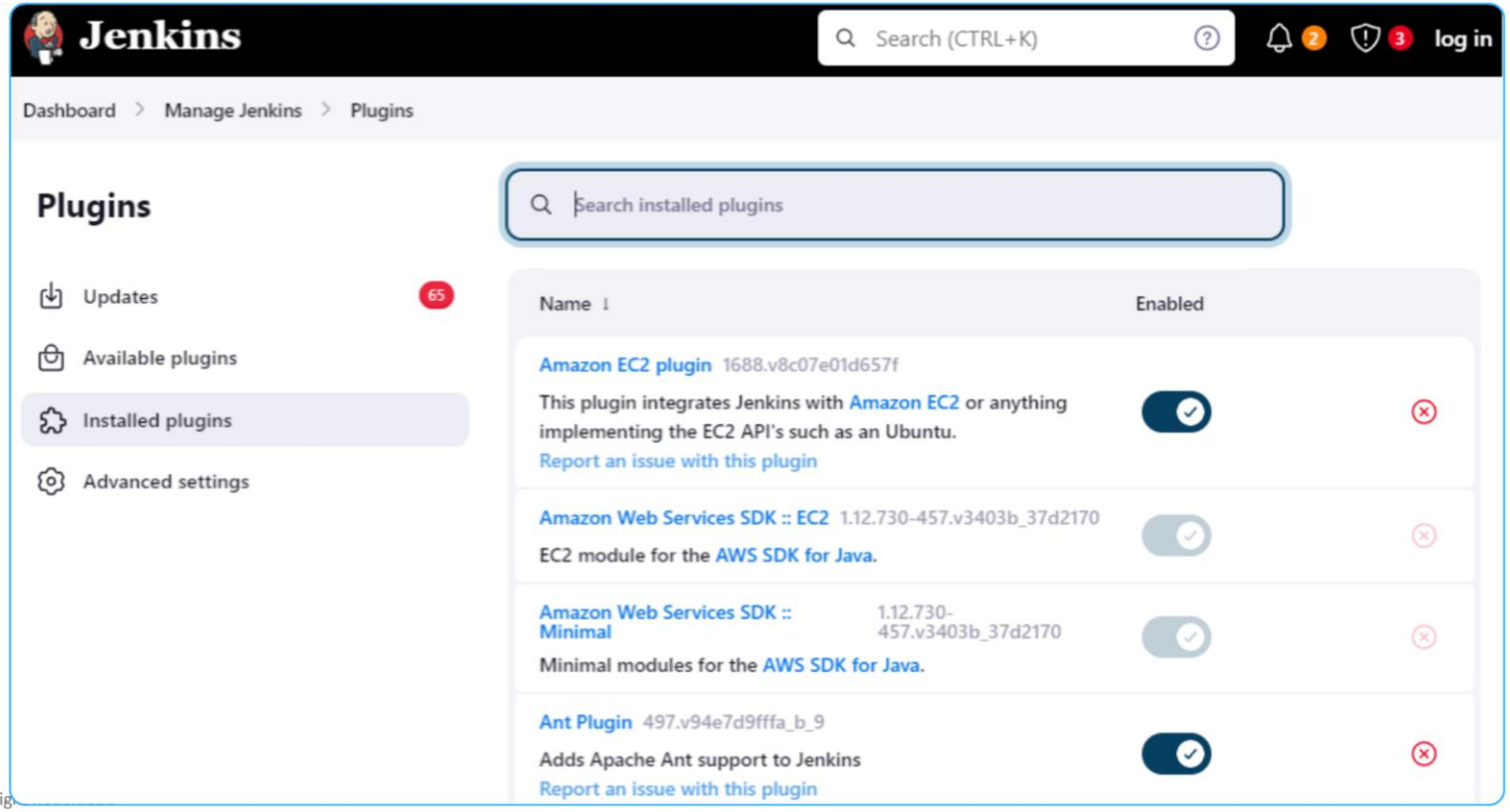
Plugins



Jenkins plugins are essentially JAR files. They come in a file with an hpi or jpi extension and stored in the ${jenkins_home}/plugins directory
"hpi" stands for Hudson PlugIn and is historical. "jpi" stands for Jenkins PlugIn.
Both .hpi and .jpi files are archive formats used to distribute Jenkins plugins. They contain the code, resources, and configuration information needed for the plugin to function within Jenkins.
If you have duplicate plugins with different extensions, the .jpi version takes priority.

Suggested Plugins When you first install Jenkins, it suggests a list of plugins that are widely used and provide fundamental capabilities to Jenkins. These include plugins for Git, Gradle, and Maven, among others. It’s a good idea to install these suggested plugins if you’re not sure what you need at the start.

Managing Plugins

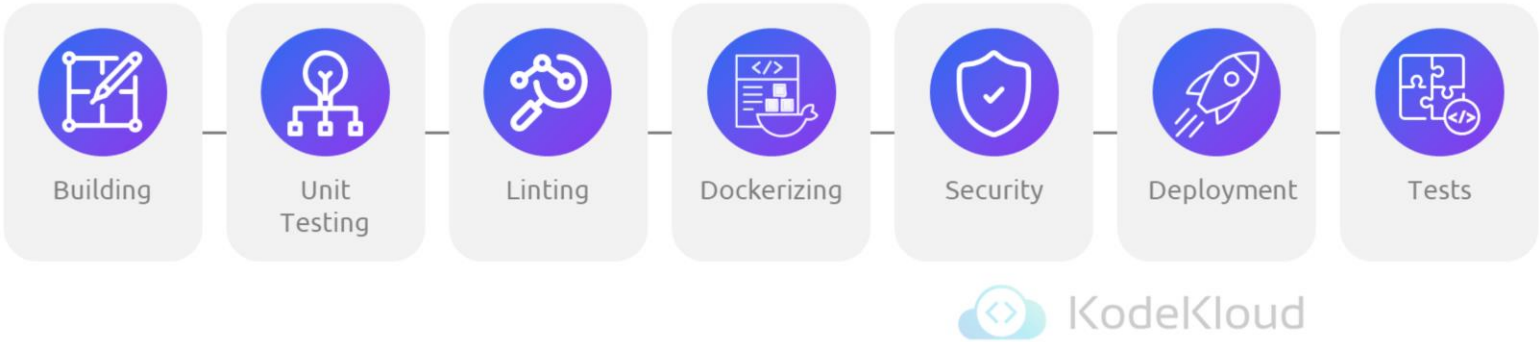


You can manage your installed plugins through the Jenkins management console. Here, you can enable or disable plugins, and uninstall plugins that you no longer need. Remember, some plugins depend on others, so be careful when uninstalling.

We'll look into plugin installation, disabling, and the update center in the upcoming demo.

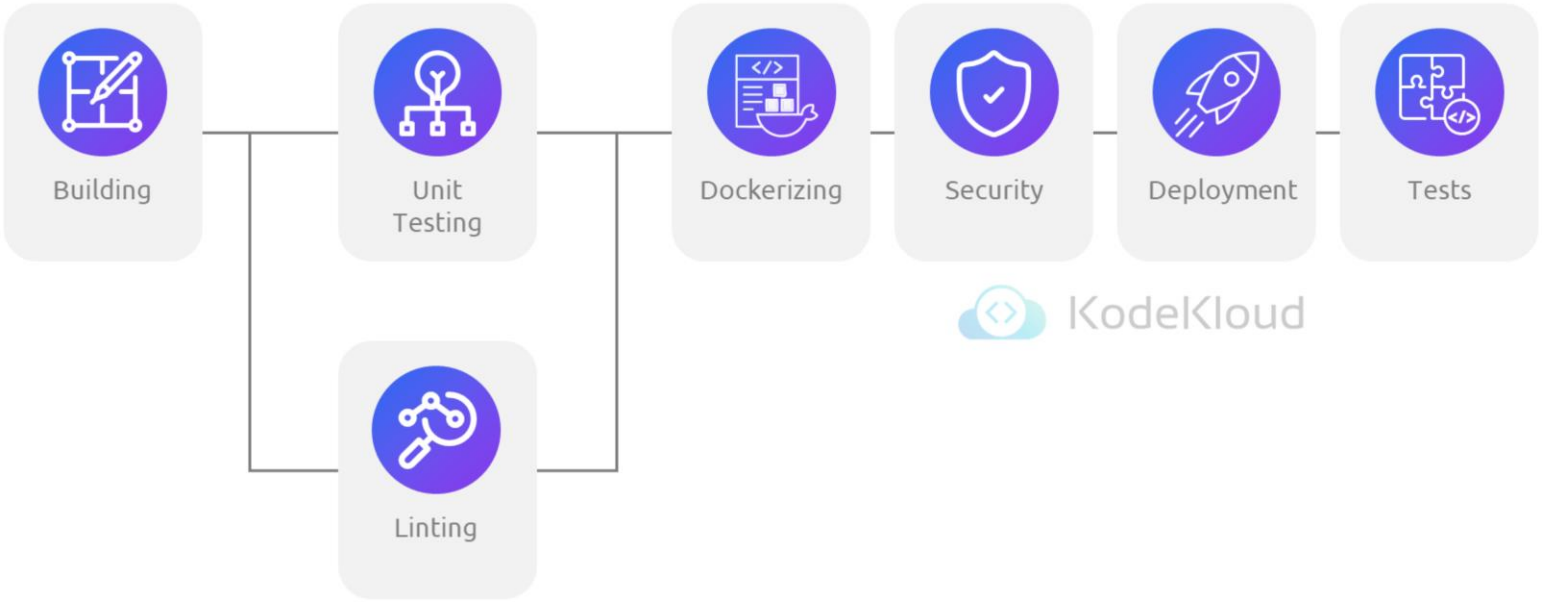
Pipeline and Jenkinsfile

Jenkins Pipeline



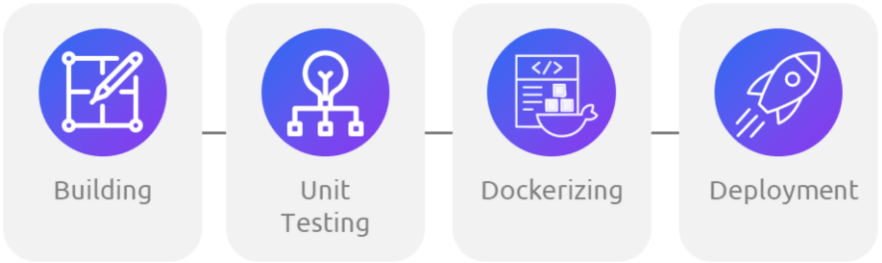
For automating complex software development workflows, particularly in continuous integration and delivery (CI/CD) practices, Jenkins Pipeline projects (often simply called "Pipeline") offer a powerful and versatile solution. Imagine a detailed recipe for building and deploying your software. Unlike a simple list of steps, a Pipeline project allows for a structured, multi-stage approach with control over execution flow. This translates to:
Organized Workflow: Break down your build process into logical stages (e.g., coding, testing, deployment) for improved clarity and easier management. Each stage can group related tasks. For example in the Dockerizing stage we could have steps for building a docker image and pushing it to a container registry

Jenkins Pipeline



Enhanced Efficiency: Leverage parallel execution for independent tasks within stages. This can significantly speed up the build process by allowing tasks like code compilation to run concurrently with unit tests.

Jenkins Pipeline



Jenkinsfile Terminal

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn build'
            }
        }
        stage('Unit Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Dockerize') {
            steps {
                sh 'docker build -t imageName:version .'
                sh 'docker push imageName:version'
            }
        }
        stage('Deploy') {
            steps {
                  sh 'kubectl apply -f deployment.yaml
                }
            }
        }
    }
}
```

Customization Power: Go beyond point-and-click configuration. Pipelines are defined using a script file called the Jenkinsfile. This script, typically written in Groovy-like domain-specific language (DSL) , empowers you to implement complex logic and automation. For instance this is how a basic Jenkinsfile looks like for these 4 stages.

Jenkinsfile offer a clear and concise way to define your build workflows.

Imagine your pipeline as a recipe – stages are like the main sections. Each stage groups related steps that perform a specific

task in your build process.
Within each stage, you define steps – these are the actual commands or scripts executed during the build. Jenkins provides built-in steps for common tasks, and you can extend this functionality with plugins offering even more options.
We'll dive deeper into these components (stages and steps) and explore real-world example.

Types of Pipeline Projects

Pipeline projects can be characterized in two ways: Declarative and Scripted Pipelines. Despite their unique syntaxes, both are denoted as Jenkinsfile and make use of the same pipeline subsystem. They are both reliant on the Apache Groovy syntax and provide support for shared libraries.

Scripted Pipeline: This approach offers the most control and flexibility but requires some coding knowledge (typically Groovy). The Jenkinsfile script allows for defining complex logic and automation tailored to your specific needs.

Code-centric: Scripted pipelines require writing a complete Groovy script (Jenkinsfile) to define all stages and steps of your workflow.
Flexibility: This approach offers the most control and flexibility. You can leverage Groovy's full power for complex logic, conditional execution, and customization.
Learning Curve: Since it involves scripting, there's a steeper learning curve for those unfamiliar with Groovy or programming concepts.

Declarative Pipeline: This option uses a simpler, human-readable format to define the pipeline. While easier to learn and use, it might not be suitable for highly complex workflows that require extensive scripting.

Human-readable: Declarative pipelines use a simpler, human-readable syntax within the Jenkinsfile. This format focuses on defining the desired outcome rather than the specific steps to achieve it.
Easier to Learn: The syntax is more intuitive and requires less coding knowledge than scripted pipelines.
Limited Complexity: While offering good flexibility, declarative pipelines might not be suitable for highly intricate workflows that require extensive scripting for conditional logic or custom actions.

Choosing the Right Approach:
The choice between scripted and declarative pipelines depends on your needs and skill set:
Scripted Pipelines: Ideal for complex workflows requiring precise control and customization. Choose this if you're comfortable with Groovy scripting.
Declarative Pipelines: A good starting point for beginners or when you have a well-defined workflow with moderate complexity.

Pipeline vs Freestyle

We talked about Freestyle projects earlier which were the traditional way of defining build workflows in Jenkins and these have limitations that Pipeline projects address:
Structure: Pipelines offer stages with potentially parallel execution of tasks within each stage. Freestyle projects, on the other hand, have a rigid, sequential order of steps.
Configuration: Pipeline projects leverage code (Jenkinsfile) for configuration, allowing for detailed control and easier version control. Freestyle projects rely on the web interface for configuration, making it less flexible and harder to track changes.

Complexity: Pipelines excel at handling intricate workflows with their scripting capabilities and stage-based structure. Freestyle projects are suitable for basic automation tasks due to their simpler, sequential approach.

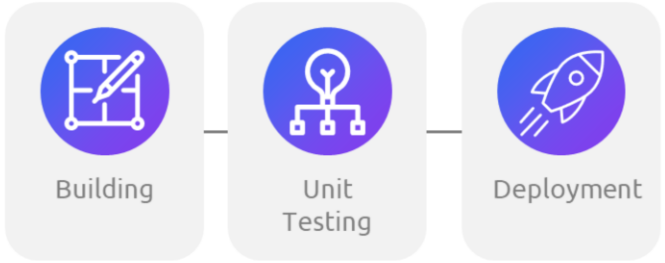
Benefits of Pipelines

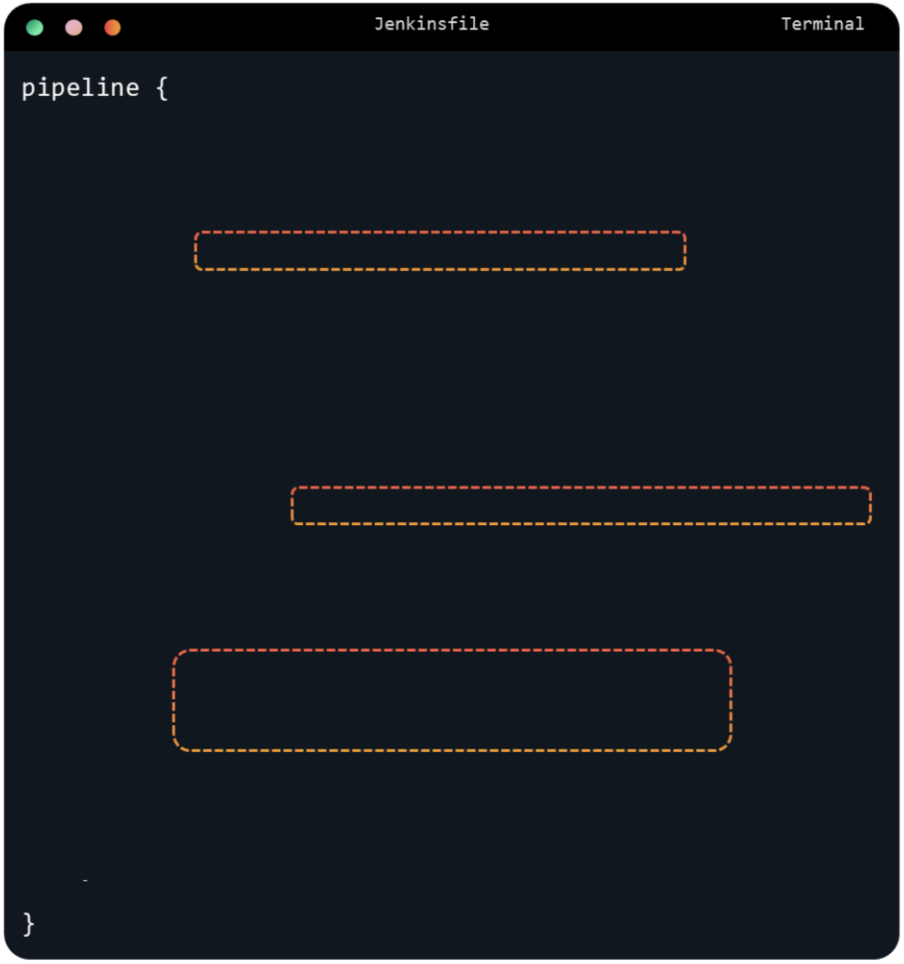
Jenkins pipelines offer a significant leap forward compared to traditional Freestyle projects. Here's how they empower your CI/CD process:
Code as Configuration: Pipelines are defined in code (Jenkinsfile) stored alongside your project code. This enables version control, collaborative editing, and easier tracking of changes.
Resilient by Design: Pipelines can survive unexpected restarts of the Jenkins controller, ensuring your workflow doesn't stall due to technical hiccups.
Human Interaction Integration: Pipelines can be paused at specific points, allowing for manual intervention or approval

before continuing the build process.
Handling Complexity with Ease: Pipelines excel at handling intricate real-world CI/CD requirements. Features like forking/joining stages, looping, and parallel execution enable efficient orchestration of complex workflows.
Extensibility Beyond Limits: The Pipeline plugin allows for custom extensions to its DSL, providing flexibility to tailor it to your specific needs. Additionally, it integrates seamlessly with numerous Jenkins plugins, further expanding its capabilities.
Streamlined Build Management: Pipelines promote efficient build management by reducing the number of jobs needed compared to Freestyle projects. Their ability to define stages with multiple steps within a single pipeline job leads to a cleaner and more concise workflow representation.

Jenkinsfile

Jenkinsfile





Just to recap, Jenkinsfile serves as the cornerstone of your CI/CD pipelines. It's a text-based script written in a Groovy-like domain-specific language (DSL) specifically designed for defining and automating the various stages of your software delivery process.

In this session we will cover the declarative Pipelines. In a later session we will talk about Scripted pipelines.

Your Jenkinsfile is structured into distinct stages, each representing a specific phase in the CI/CD pipeline. Common stages

include:
Source Code Management (SCM): Checking out code from your version control system (e.g., Git, Subversion)
Build: Compiling, building, and packaging your application
Test: Executing unit, integration, and other tests to ensure code quality
Deploy: Pushing your built artifacts to staging or production environments

Here's a breakdown of the essential syntax and components that make up a Jenkinsfile:
Pipeline Definition (pipeline):
The pipeline keyword marks the beginning of your pipeline definition.
Agent (agent):
The agent directive specifies the execution environment for your pipeline stages. Common options include:
any: Runs on any available Jenkins agent including the Jenkins controller
docker { image 'node:16' }: Runs within a Docker container
Stages (stage):
stages (plural): This refers to the entire concept of structuring your pipeline into different phases. It's a general term encompassing the idea of breaking down your workflow into meaningful steps
stage (singular): This is the specific keyword used within the Jenkinsfile syntax to define an individual stage within your pipeline. It's the building block that creates those distinct phases in your workflow..
Steps (steps):
The steps block within a stage lists the individual commands or actions to be executed in that stage.
Steps can involve:
Shell commands (e.g., sh 'mvn clean install')
Jenkins Pipeline Domain Specific Language commands (e.g., script { ... }, when)
Use the script step to include bits of scripted code within a Declarative Pipeline only when you have needs that are beyond the capabilities of Declarative syntax
Third-party plugin integrations (e.g., junit 'testResults/**/*.xml’)

In this example:
The pipeline runs on any available Jenkins agent.
The Build stage executes Maven commands to build the application.

- This stage will run on a Docker container with the image ubuntu:alpine. This ensures the environment has the necessary tools for building your project.
- Notice how the agent directive is nested within the stage('Building') block. This means the ubuntu:alpine Docker container will only be used for the "Building" stage.
- Other stages will be using the any agent defined at the root level.

The Test stage runs unit tests using JUnit plugin integration.
The optional Deploy stage copies the built WAR file to a production server using scp (only if the repo was pushed to the 'main' branch). Here we are conditionally deploy using the when directive.

Jenkinsfile Directives

environment

```
pipeline {
  environment {
    VAR1 = 'value1'
    VAR2 = 'value2'
  }
  // ... pipeline stages
}
```

post

```
pipeline {
  // ... pipeline stages
  post {
    success {
      script {
        // Send success notification
      }
    }
    failure {
      script {
        // Send failure notification
      }
    }
  }
}
```


environment:
Defines key-value pairs to set environment variables for all stages or specific stages (depending on placement).

post:
Executes one or more steps after a pipeline run finishes, regardless of success or failure.
Common use cases: sending notifications, cleaning up temporary files.

Jenkinsfile Directives

notification

```
Terminal
pipeline {
  // ... pipeline stages
  post {
    always {
      slackNotifier(
        channel: '#your-slack-channel',
        message: '${currentBuild.result}'
      )
    }
  }
}
```

script block

```
Terminal
stage('Process Files') {
  steps {
    script {
      files = ['file1.txt', 'file2.txt']
      files.each { file ->
        sh "echo Processing $file"
      }
    }
  }
}
```


notification (using Slack):
Integrates with plugins like SlackNotifier to send notifications to channels.
Requires plugin configuration and specific syntax for the chosen notification plugin.
script:
Encapsulates a block of Groovy code for more complex logic within a stage's steps.
Provides access to pipeline variables, methods, and Jenkins functionalities.
Example (using script block to loop through a list of files):

Jenkinsfile Directives

when

Terminal

```
pipeline {
  // ... pipeline stages
  stage('Deploy') {
    when {
      expression { branch == 'main' }
    }
    // ... deployment steps
  }
}
```


credentials

Terminal

```
pipeline {
 agent any
  stages {
   stage('Example') {
    steps {
     withCredentials([(credentialsId: 'myCredentials')
      echo "Username: $USERNAME"
    ...
```

when:
Defines conditions under which a stage should execute.
Commonly used for conditional branching based on factors like branch name or build result.

credentials:
Provides access to pre-defined credentials stored in Jenkins (e.g., API keys, passwords).
Used with tools or scripts that require authentication.

Jenkinsfile Directives

input

Terminal

```
input {
  message 'Are you sure you want to deploy?'
  ok 'Yes'
  cancel 'No'
}
```


parameters

Terminal

```
pipeline {
  parameters {
    string(name: 'ENV_NAME', defaultValue: 'dev',
description: 'Environment to deploy to')
  }
  // ... pipeline stages
  environment {
    ENV = params.ENV_NAME
  }
}
```

interactive inputs:
Allows users to provide input during pipeline execution through a UI.
Requires plugins like Pipeline Utility Steps plugin.
Example (prompting user for confirmation):

parameters:
Defines parameters that can be passed to the pipeline during execution, allowing for dynamic configuration.

Jenkinsfile Directives

stash/unstash

```
Terminal
stage('Build') {
  // ... build steps
  stash name: 'build-artifacts'
}

stage('Deploy') {
  unstash name: 'build-artifacts'
  // ... deployment steps using stashed
artifacts
}
```

parallel stages

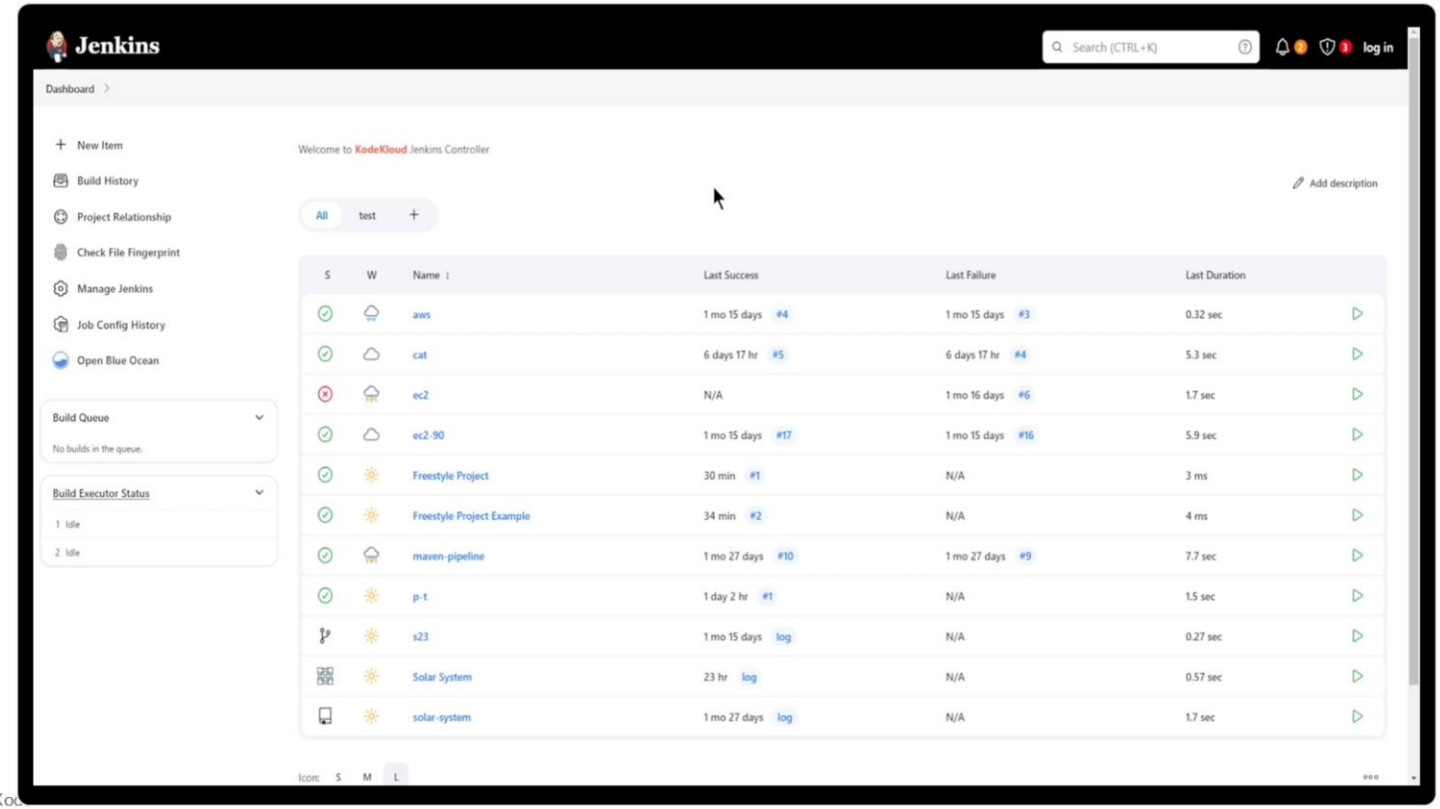
```
Terminal
pipeline {
  parallel {
    stage('Build') {
      // ... build steps
    }
    stage('Test') {
      // ... test steps
    }
  }
  // ... subsequent stages (optional)
}
```


. stash/unstash:
Stores and retrieves artifacts (files, build outputs) between pipeline stages or jobs.
Useful for caching build results or sharing data across jobs.
Example (stashing build artifacts):
parallel stages:
Executes multiple stages concurrently to optimize pipeline execution time.
Requires stages to be independent and not rely on outputs from each other.

Example (running two stages in parallel):

Jenkins CLI, APIs

Automation

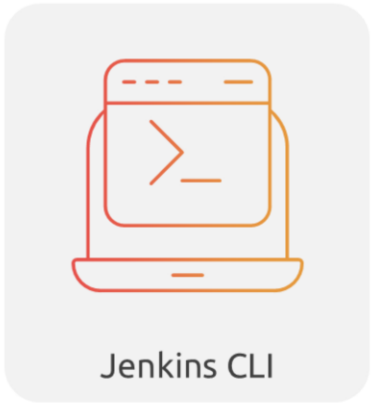


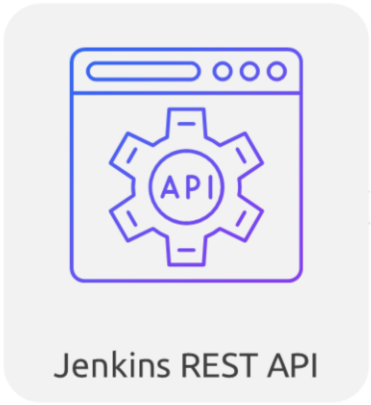
While the graphical Jenkins dashboard is a great way to get started and explore basic features, automating tasks with the command-line interface (CLI) and REST API offers significant efficiency benefits.

Jenkins offers two powerful options for automating tasks: the Jenkins CLI (Command Line Interface) and the Jenkins REST API.

Let’s dive into how each works and how to leverage them for automation.

Automation





Jenkins offers two powerful options for automating tasks: the Jenkins CLI (Command Line Interface) and the Jenkins REST API.

Let’s dive into how each works and how to leverage them for automation.

Jenkins CLI

Jenkins provides a built-in command line interface (CLI) that enables users and administrators to interact with Jenkins from a script or shell environment. You can access the CLI over SSH or by using the Jenkins CLI client, which is distributed as a .jar file along with Jenkins.s

Jenkins CLI - SSH

Terminal

```
curl -Lv https://JENKINS_URL/login 2>&1 | grep -i 'x-ssh-endpoint'
< X-SSH-Endpoint: localhost:53801
```

Terminal

```
ssh -l username -p 53801 localhost help
```

Terminal

```
ssh -l username -p 53801 localhost build hello-kodekloud -f -v
Started hello-kodekloud #1
Started from command line by username
Building in workspace /tmp/jenkins/workspace/hello-kodekloud
[hello-kodekloud] /bin/sh -xe /tmp/hudson56238023482.sh
+ echo Hello KodeKloud
Hello KodeKloud
Finished: SUCCESS
Completed hello-kodekloud #1 : SUCCESS
```


If you choose the SSH option, note that the SSH service is initially disabled in Jenkins. To enable it, you'll need to configure the SSH port. Once you've done that, set up authentication for SSH. Jenkins relies on SSH-based public/private key authentication. To add an SSH public key for the relevant user, navigate to the Jenkins configuration page and paste the key there.

After configuring authentication, you'll have access to several built-in CLI commands in any Jenkins environment. These commands include actions like build or list-jobs. To see the complete list of available commands, execute the

CLI help command.

One of the most common and useful CLI commands is build. It allows users to trigger any job or pipeline for which they have permission. Here’s an example of how the SSH command and response might look for a simple job named ‘hello-kodekloud’.

Jenkins CLI - JAR

Jenkins CLI

You can access various features in Jenkins through a command-line tool. See the documentation for more details of this feature. To get started, download jenkins-cli.jar, and run it as follows:

```
java -jar jenkins-cli.jar -s http://10.0.0.5:8080/ -webSocket help
```

Available Commands

Name ↓	Description
add-job-to-view	Adds jobs to view.
build	Builds a job, and optionally waits until its completion.
cancel-quiet-down	Cancel the effect of the "quiet-down" command.
clear-queue	Clears the build queue.
connect-node	Reconnect to a node(s)
console	Retrieves console output of a build.

While the SSH-based CLI is efficient and meets most requirements, there are scenarios where the Jenkins CLI client distributed with Jenkins is a more suitable choice. For instance, the default transport for the CLI client is HTTP, which eliminates the need to open additional ports in a firewall.

The Jenkins CLI is a Java application provided as a downloadable JAR file. It enables you to interact with Jenkins from your terminal, making automation through scripting possible. To use the CLI, simply download the jenkins-cli.jar file from your Jenkins instance.

Jenkins CLI - JAR

Terminal

```
java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token list-jobs
```

Terminal

```
java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token
job build <job_name>
```

Terminal

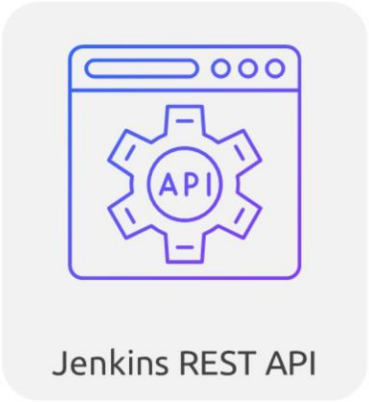
```
java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token
job get-config <job_name>
```


You can use the CLI to perform various actions, including:
Listing jobs:
Building a job:
Getting job details

Authentication: The CLI offers secure access through
Basic Authentication: Use the -auth username:password flag to provide your credentials for connecting to Jenkins.

However, this stores credentials in plain text on the command line, so consider it for quick operations or local environments.
API Tokens: For improved security, create API tokens in Jenkins and use them with the -token <token> flag. This avoids exposing credentials in the command line.

Jenkins REST API



Terminal

```
curl -s POST --data "<jenkins><install plugin='${plugin}' /></jenkins>“ \
 -H 'Content-Type: text/xml’ \
 http://localhost:8080/pluginManager/installNecessaryPlugins \
--user admin:$JENKINS_TOKEN
```


The Jenkins REST API provides a comprehensive set of endpoints for programmatic interaction with Jenkins. This allows you to develop scripts or applications that can control Jenkins remotely.

Actions: The API empowers you to perform a wider range of tasks, including:
Creating, updating, and deleting jobs
Triggering builds
Retrieving job details, logs, and artifacts

Managing nodes and agents

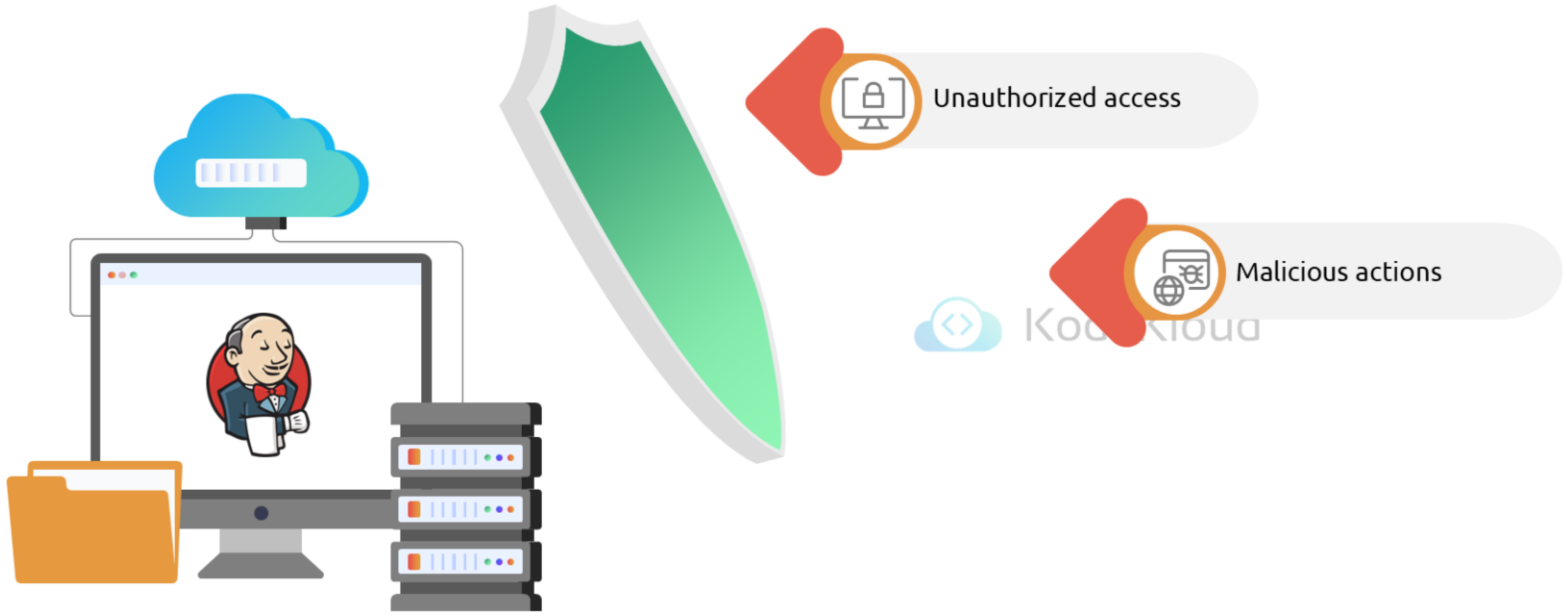
Usage: You can access the API endpoints through HTTP requests using tools like curl or by utilizing libraries in various programming languages (Python, Java, etc.). These libraries simplify API interactions and offer a more structured approach.

Authentication: Similar to the CLI, the REST API also supports authentication methods for secure access:
Basic Authentication: Pass your username and password using HTTP headers for basic authentication. This method is similar to the CLI's approach, but be cautious about storing credentials in code.
<for sid – double check this >API Tokens: The recommended approach is using API tokens generated within Jenkins. Include the token in the HTTP header (Authorization: Bearer <token>) for secure communication.

For example here is a JENKINS RESTT API for installing plugins

Jenkins Security Overview

Security

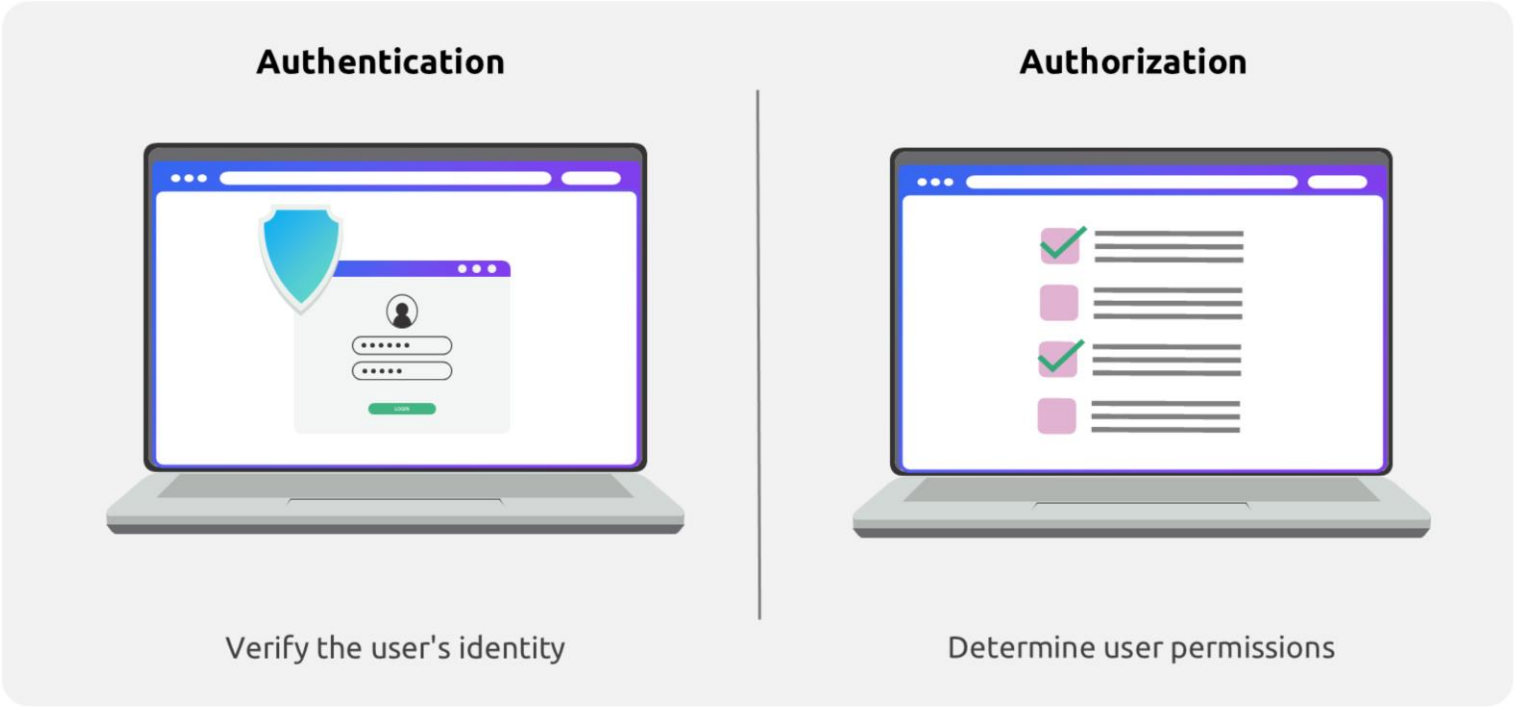


Core principle of jenkins security is the concept of least privilege.

Protecting your Jenkins server is like securing your toolbox. You wouldn't want someone to steal your tools or use them to damage something. The same applies to Jenkins - you want to prevent unauthorized access and malicious actions.

Just like any other system, a core principle of Jenkins security is the concept of least privilege. This means giving users only the permissions they absolutely need to perform their tasks within Jenkins.

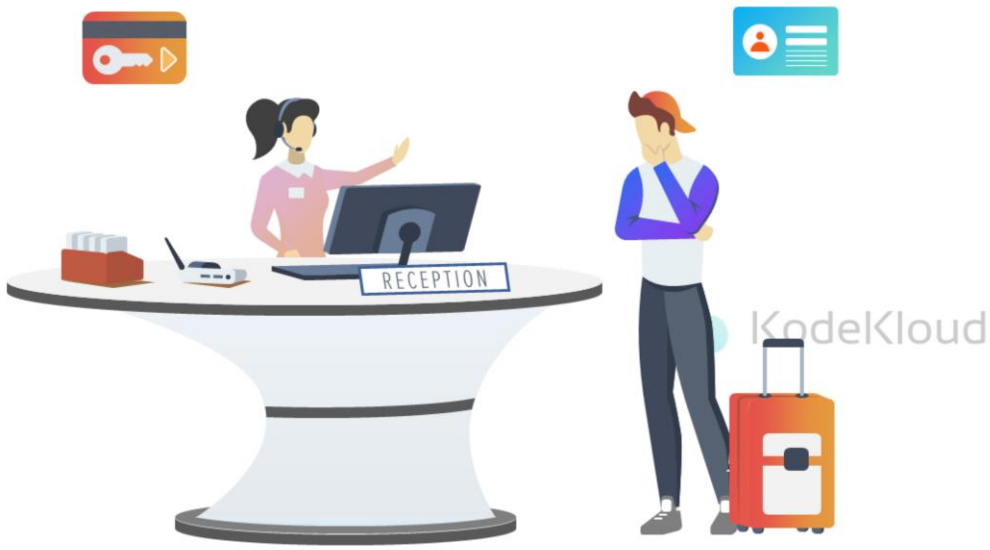
Authentication vs Authorization



Authentication is the process of verifying someone's identity.
How it works: You provide credentials (username/password, API token, etc.), and the system checks if they are valid and belong to a real user.

Authorization is the process of determining what a user is allowed to do after they've been authenticated.
How it works: Based on your identity and assigned permissions, the system determines what actions you can take (read, write, delete, etc.).

Authentication vs Authorization



Verifies identity

Grants access

To give an analogy,
Authentication (Government ID): When you check into a hotel, you present your government ID (passport, driver's license) to the receptionist. This verifies your identity and confirms that you are the person who made the reservation.
Authorization (Room Key): Once your identity is verified (authentication), the receptionist hands you a room key. This key acts as your authorization token. It grants you access to only one specific room (your assigned room) and allows you to perform specific actions within that room (unlock the door, use the amenities).

Authentication in Jenkins

Built-in Choices

- Jenkins User Database
- Unix User/Group Database
- Servlet Container Security
- External LDAP

Expanding Options

- Active Directory
- SAML 2.0 Single Sign-On

Jenkins offers several ways to manage who can access it, providing different levels of flexibility depending on your needs. Here's a breakdown:

Built-in Choices:

Jenkins User Database: This is the default option, suitable for smaller setups where you can manage users directly within Jenkins.

Unix User/Group Database: If your system uses Unix-based user accounts, you can leverage them for Jenkins access

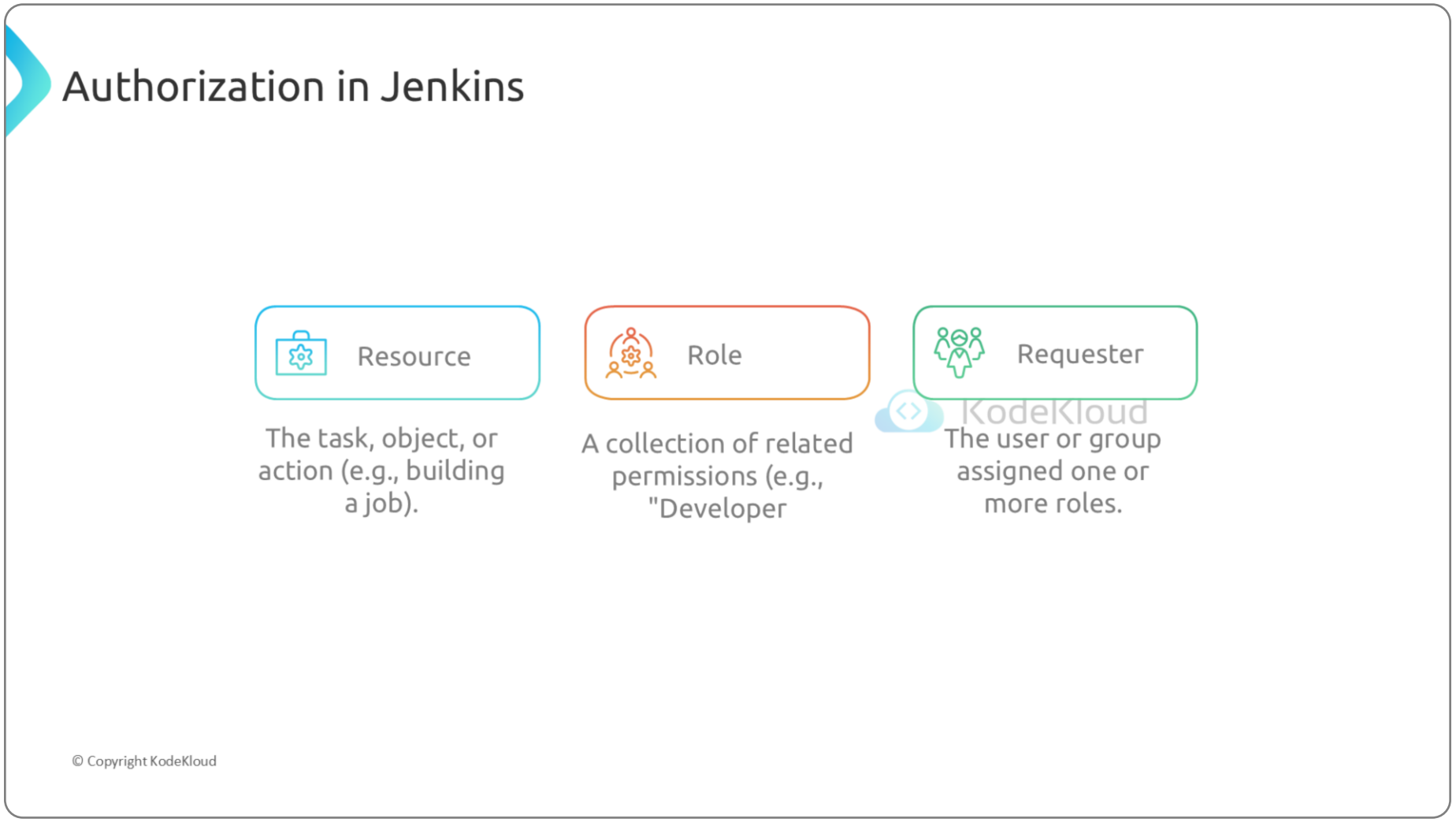
control.
Servlet Container Security: This option integrates with your web server's security model for user authentication.
External LDAP: For larger organizations, Jenkins can connect to existing directory services like LDAP, allowing users to log in with their usual corporate credentials.

Expanding Your Options (Plugins): Jenkins plugins can further extend your security options. Popular choices include:

Active Directory: Integrate with Microsoft's Active Directory for centralized user management.
SAML 2.0 Single Sign-On: Enable users to log in to Jenkins using their existing SSO credentials for a seamless experience.

Authentication in Jenkins

Choosing the Right Approach:
The default internal database works well for smaller setups. However, for enterprise environments, using a corporate directory service (LDAP or Active Directory) is recommended. This simplifies user management and avoids the need for separate credentials within Jenkins.



Authorization in Jenkins determines what actions users can perform. Here's a breakdown:

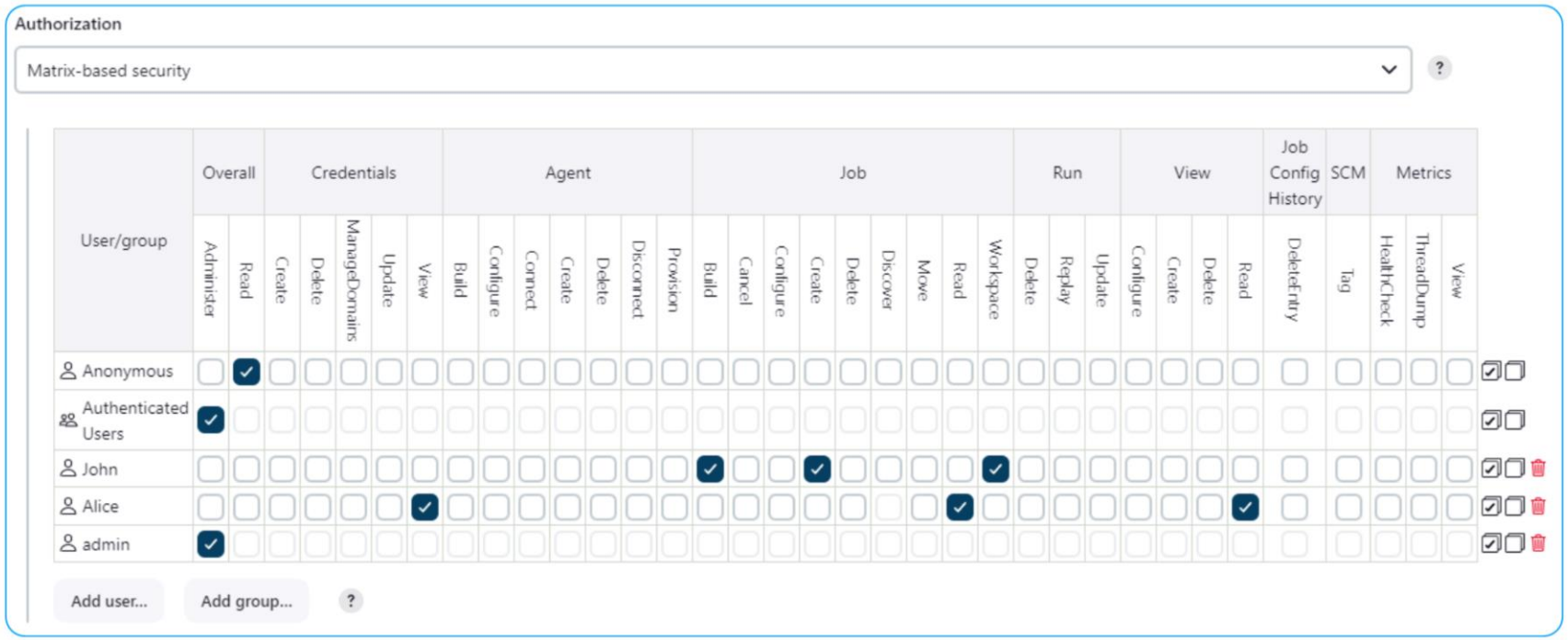
Understanding the Players:

Resource: This represents the "what" - the task, object, or action a user might want to manipulate (e.g., building a job, deleting a credential).

Role: This is a collection of related permissions grouped together by function (e.g., "Developer" with permissions to build and view jobs).

Requester: This is the user or group trying to access a resource, typically assigned one or more roles.

Authorization in Jenkins



Security Matrix: A Visual Guide
Jenkins uses a security matrix to define authorization. It's like a spreadsheet with permissions (grouped by resource) across the top and user/group names down the left side. This allows you to easily see who has access to what.

Authorization Options:

Matrix-Based Security: This is a simpler approach where users and groups are granted permissions globally (across all

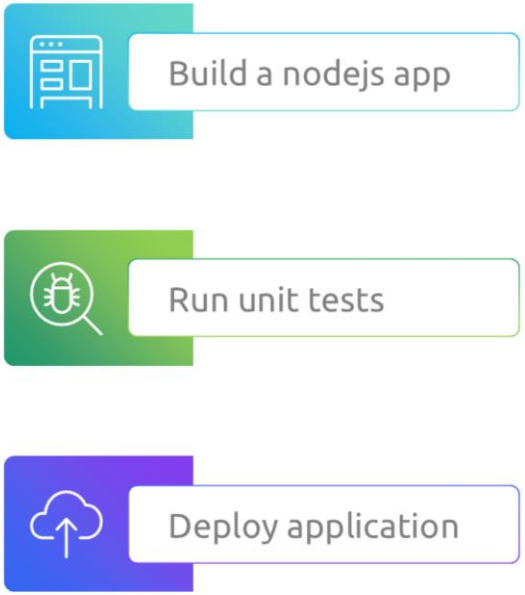
projects).
Project-Based Matrix Authorization Strategy: This offers more granular control by allowing you to assign permissions based on the specific projects a user or group accesses.

Remember, Always Grant the Least Necessary Access:
When assigning permissions, follow the principle of least privilege. Give users only the minimum access required to do their jobs effectively. This avoids creating bottlenecks in your CI/CD pipeline while maintaining good security.

Introduction to Shared Libraries

A shared library is a collection of Groovy scripts that define reusable functions (steps) for your Jenkins pipelines. These functions encapsulate common tasks, making your pipelines more concise and readable.

Why use Shared Library



Terminal

```
$ cat Jenkinsfile
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'npm install'
        sh 'npm run build'
      }
    }
    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
    stage('Deploy') {
      steps {
        script {
          def server = 'user@server.example.com'
          def targetDir = '/path/to/deployment/directory'
          sh "scp -r target/* ${server}:${targetDir}"
        }
      }
    }
  }
}
```

Assume you have a job with 3 stages to build a nodejs app, run unit tests, and deploy application.

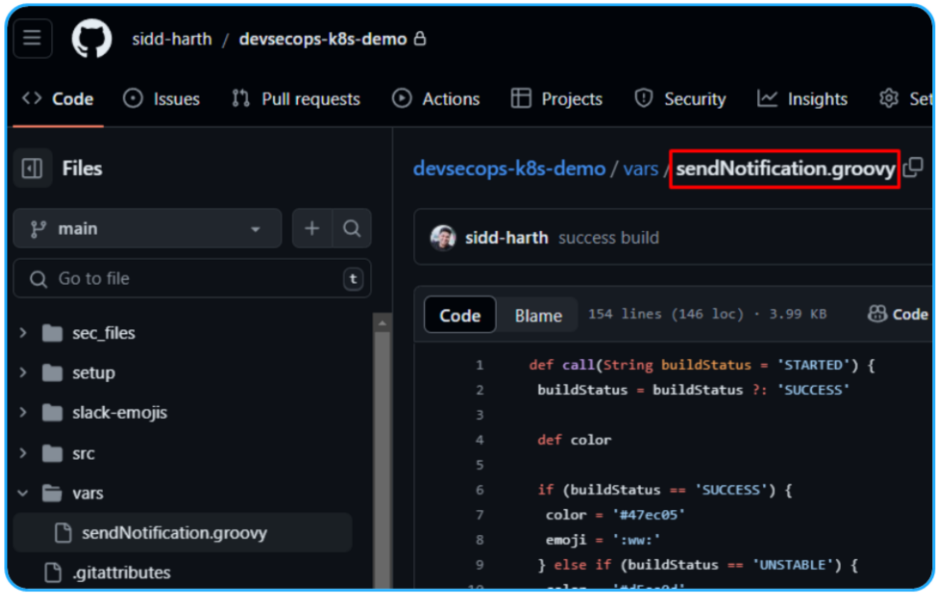
Imagine you have to build multiple Jenkins jobs that all need to perform similar tasks like building and testing using the same commands.

Why use Shared Library

Without shared libraries, you'd copy and paste that code into each Jenkinsfile, leading to:

Duplication: Wasted effort maintaining the same code in multiple places.
Inconsistency: Changes in one place might not be reflected everywhere.
Difficulty in managing complexity: As pipelines grow, managing scattered code becomes cumbersome.

Why use Shared Library



DRY (Don't Repeat Yourself) principle

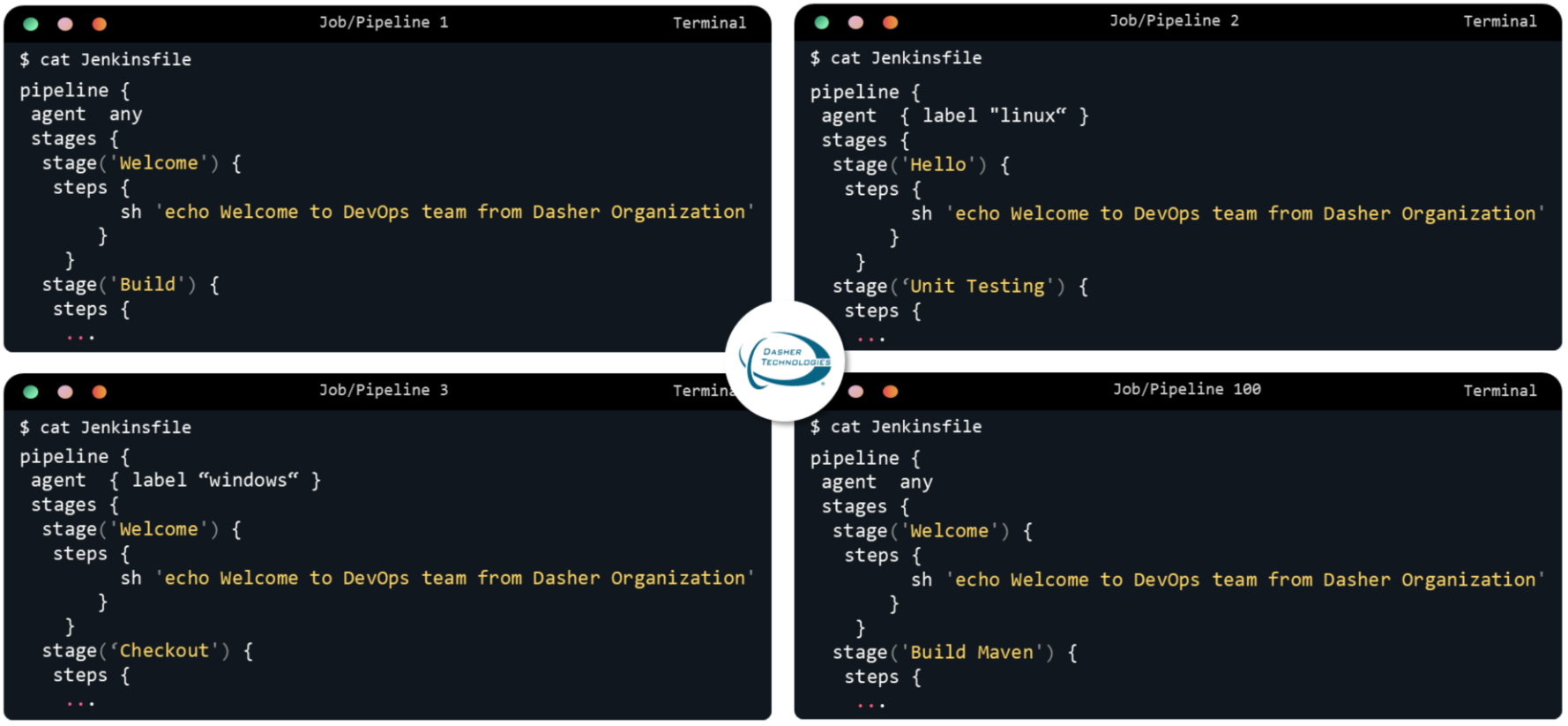
KodeKloud
Consistency

Maintainability

Shared libraries solve these problems by providing a central location to store reusable Groovy scripts that can be incorporated into your Jenkinsfiles. This promotes:
DRY (Don't Repeat Yourself) principle: Code is written once and reused across jobs.
Consistency: Updates in the shared library are reflected everywhere it's used.
Maintainability: Easier to manage and evolve common functionality in one place.

The central location could be anything which is reachable by Jenkins server. In most cases it is a Git repository as seen in this image.

Shared libraries - Example



Lets take an example and understand how it’s done

Assume as an organization just started using Jenkins and has a mandate that each job/pipeline should start with a stage which has a static welcome message. So you start with your first job/pipeline and have hardcoded the welcome message “Welcome to DevOps team from Dasher Organization”

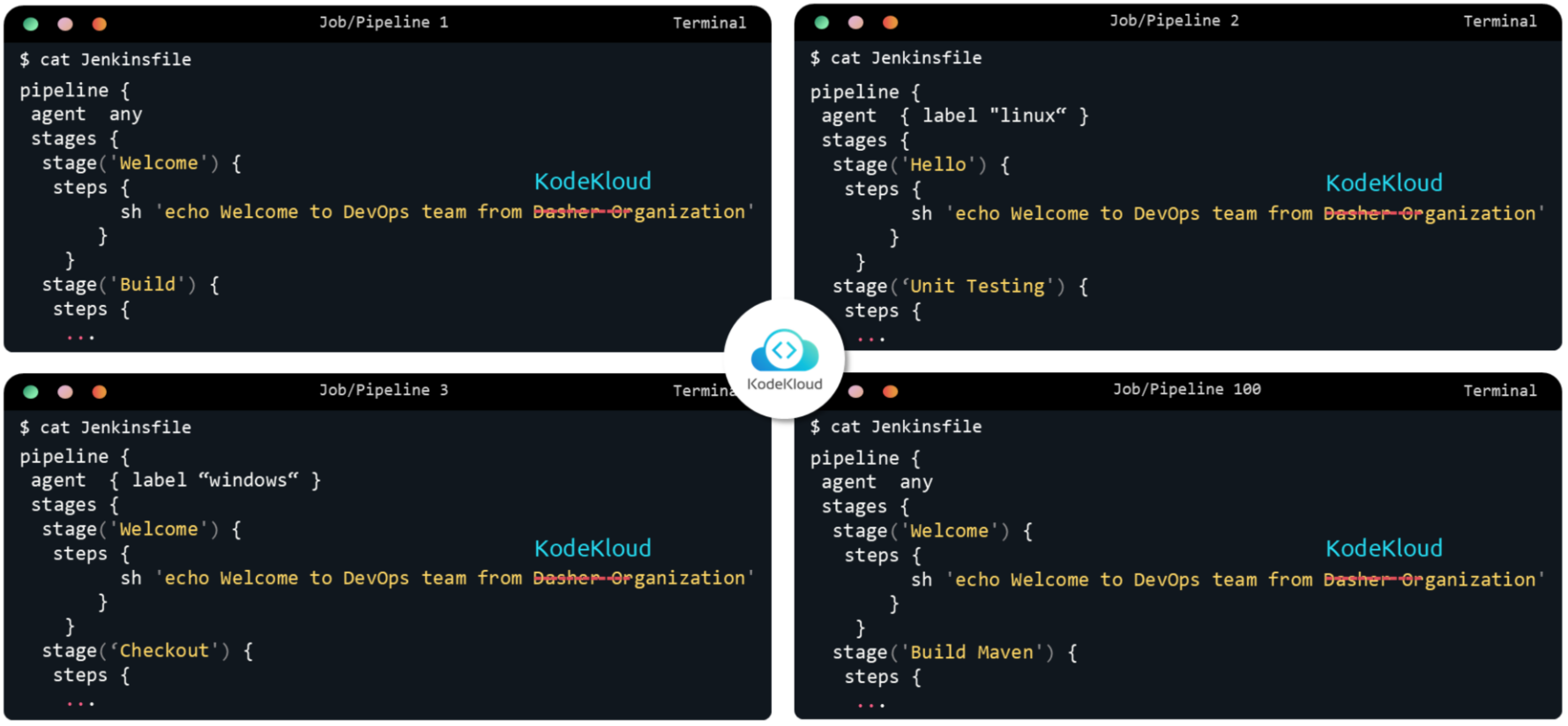
As time progresses the number of pipelines increases exponentially and the same welcome message has been duplicated to

all the hundered pipelines the organization currently handles.

Let's consider a scenario where a new DevOps team is just getting started with Jenkins. They have a requirement that every job or pipeline begins with a stage displaying a welcome message. Initially, they create their first pipeline and directly include a welcome message like "Welcome to DevOps team from Dasher Organization.“

Duplication and Maintenance Issues Arise
As the organization grows, the number of pipelines multiplies rapidly. The same welcome message gets copied and pasted into all the hundreds of pipelines they now manage. This leads to a problem:
Code Duplication: The welcome message code is scattered across many pipelines, making it difficult to maintain and update.
Inconsistency: If the organization needs to change the message, they must manually modify it in every single pipeline, which can be time-consuming and error-prone.

Shared libraries - Example

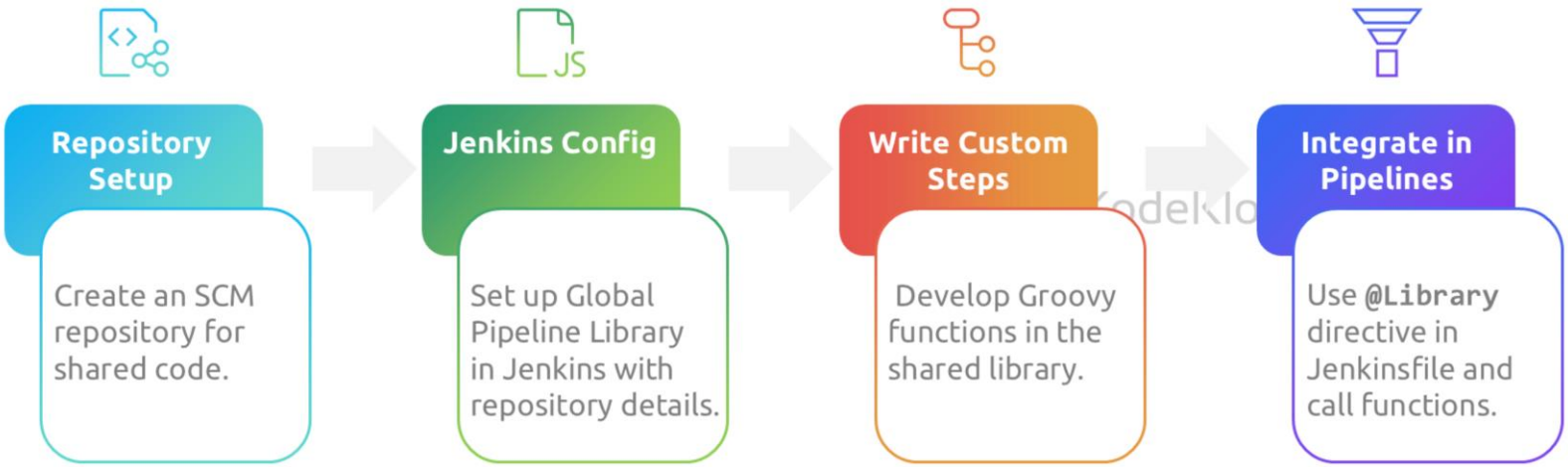


Now fast forward 6 months, the organization name changed from Dasher to Kodekloud and now this change has to updated in not one but hundreds of pipelines.

Shared Libraries: The Solution
This is where shared libraries come in. They offer a way to centralize reusable code, eliminating duplication and promoting consistency:
Centralized Message: Instead of hardcoding the message in each pipeline, you can create a shared library containing a

function that returns the welcome message.
Easy Updates: If the organization's name changes (e.g., from Dasher to Kodekloud), you only need to update the shared library function. This change will automatically be reflected in all pipelines that use the library.

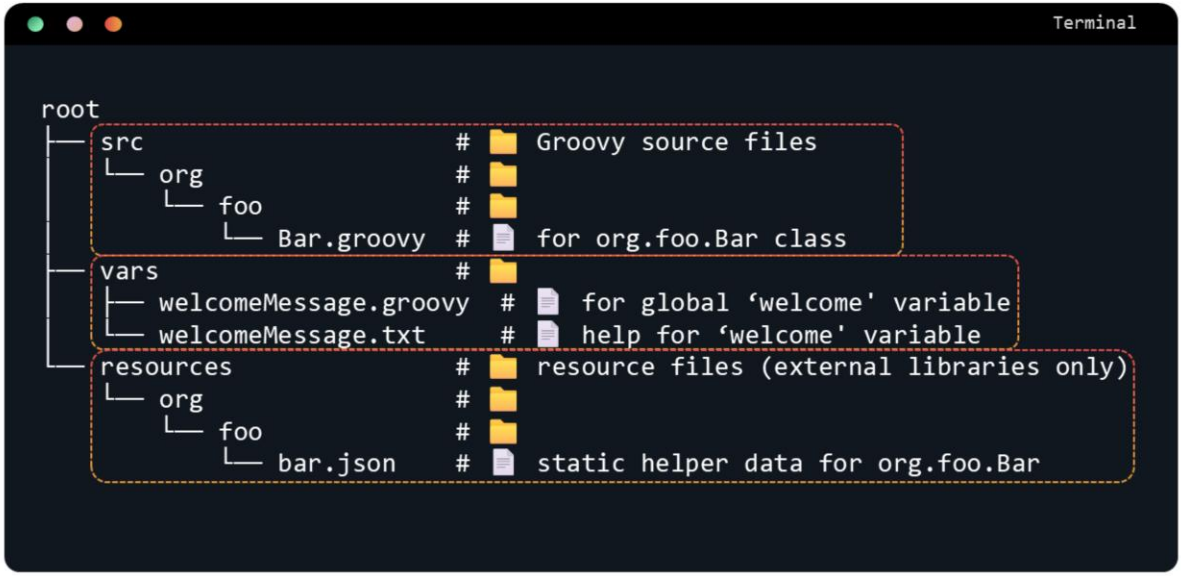
Steps to Shared Library



Four Steps to Shared Library:
Dedicated Repository: Start by setting up a separate SCM repository specifically for your shared library code. This keeps it organized and easy to track.
Jenkins Configuration: Head over to Jenkins and configure a Global Pipeline Library. Here, you'll provide details like the library name and how Jenkins retrieves the code (e.g., the SCM repository URL).
Craft Your Custom Step: Write the Groovy code defining your reusable function (step) within the shared library repository. Think of it as a building block for your pipelines.

Putting It All Together: In your Jenkinsfile, use the @Library directive to load the shared library. Then, you can directly call your custom step from within your pipeline stages, making your code more concise and maintainable.

Shared Library Repository Structure



This is a sample Shared Library Repository Structure

While not mandatory, the shared library offers several directories for organizing your code:
src (Optional): This directory follows a standard Java project structure and gets added to the Jenkins pipeline classpath. It's generally recommended to avoid using src for shared libraries as it's meant for compiled code.
vars (Essential): This is where you define your reusable steps (functions) for pipelines. Here are the key points:

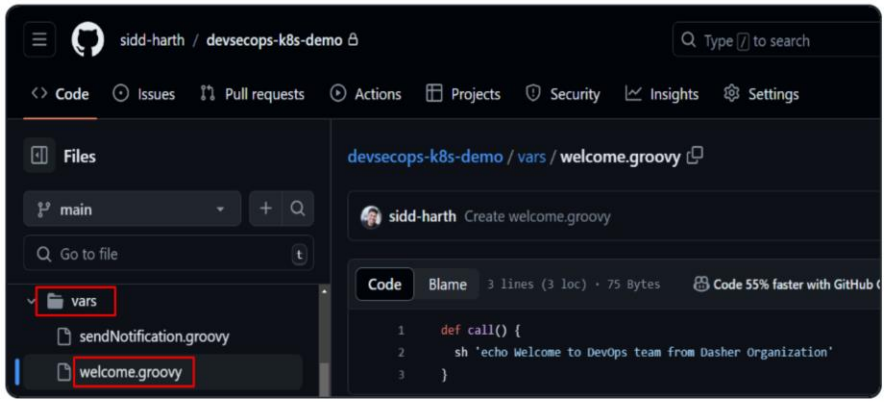
No Subfolders are Allowed here: Keep things organized within the root of vars.

Naming Convention: Use camelCase for file names (e.g., deploy.groovy).
Optional Documentation: Include a matching .txt file (e.g., deploy.txt) for step documentation. Jenkins can process it using your system's configured markup formatter.

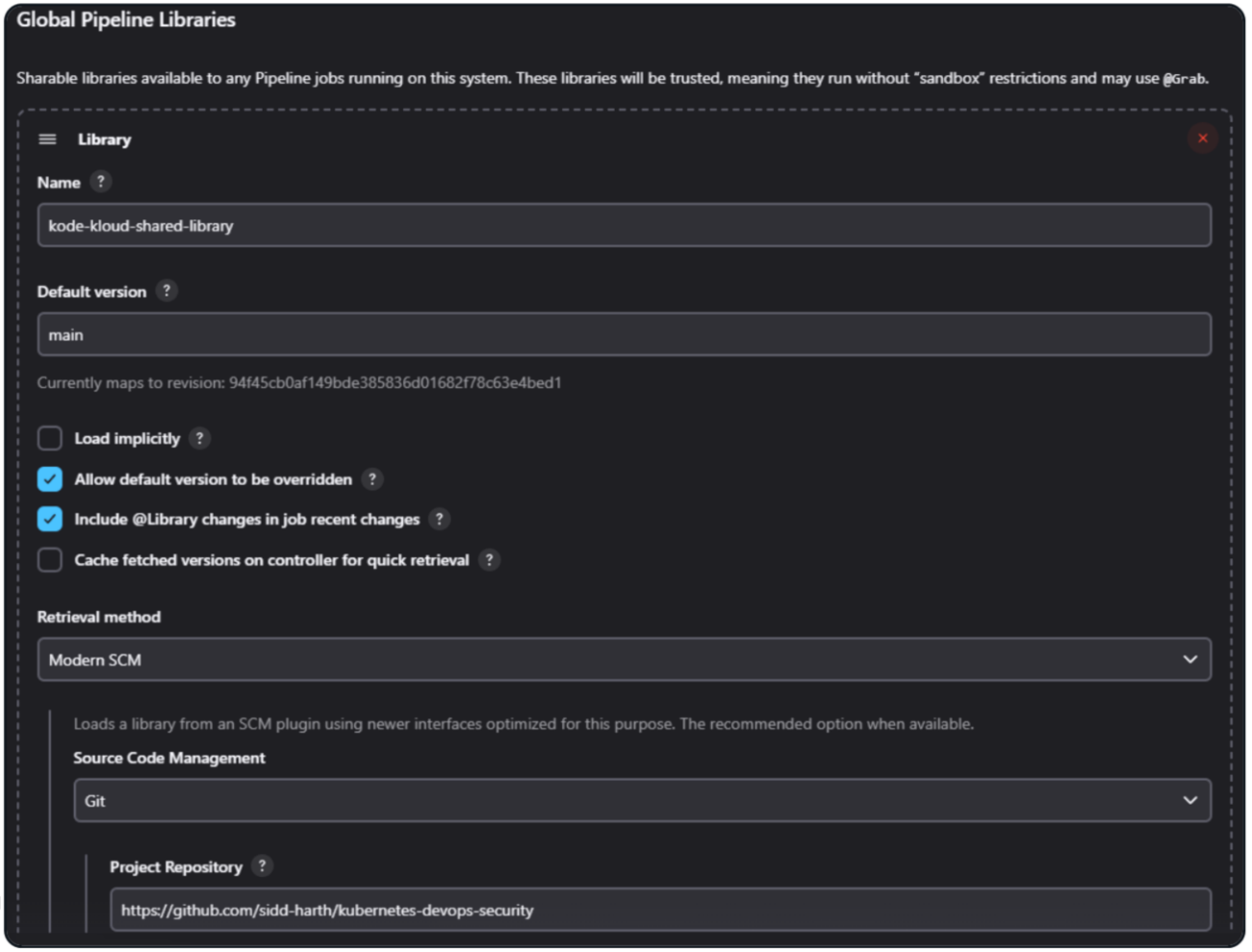
resources (Optional): While libraryResource lets external libraries load non-Groovy files, it's not yet usable for internal shared libraries within Jenkins.

By understanding these directories and their purposes, you can effectively structure your shared libraries for better organization, maintainability, and reuse in your Jenkins pipelines.

Shared libraries - Example



- We'll use GitHub as the version control system (SCM).
- Inside the vars directory of the repository, create a file named welcome.groovy.
- This file will define a reusable function named call() that returns the welcome message.
- Head over to Jenkins and configure a Global Pipeline Library. This tells Jenkins how to access the shared library code.



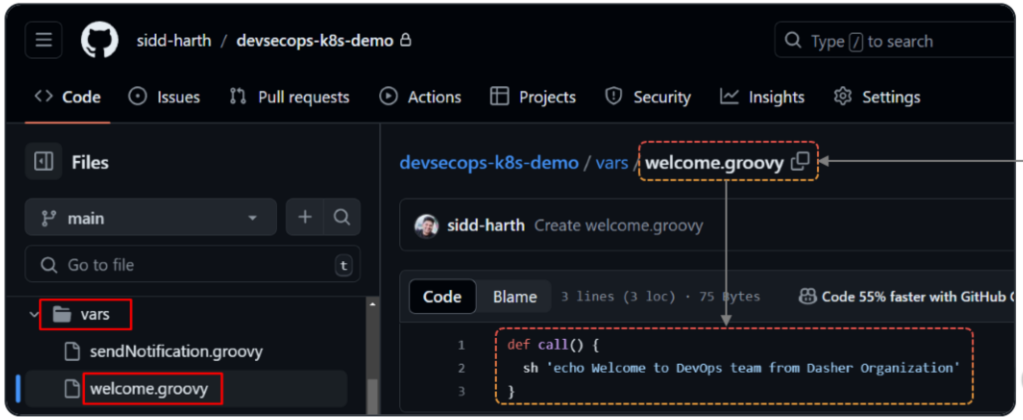
Here, you'll provide details like the library name and how Jenkins retrieves the code (e.g., the SCM repository URL).

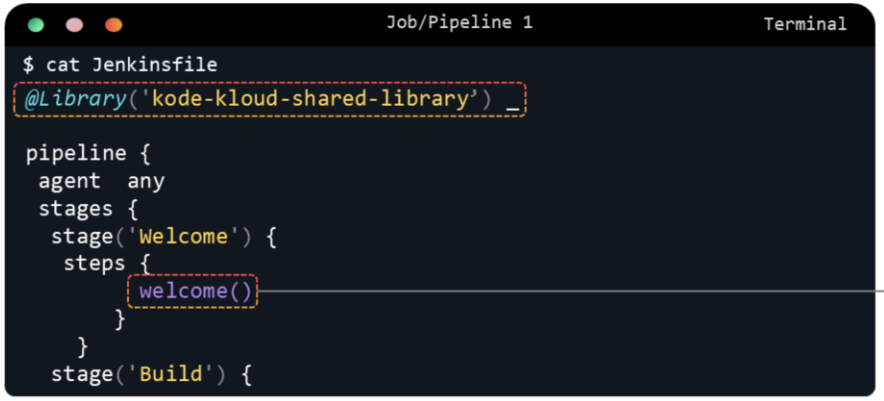
Default Versioning:
Set "Default version" to "main" to have pipelines use the "main" branch by default.
If "Allow default version to be overridden" is enabled, pipelines can use other branches with the @Library annotation.

Implicit Loading and SCM Configuration:

Enable "Load implicitly" to make the default branch automatically available.
Use "Modern SCM" as the Retrieval method and configure the source code management details (repository URL) for Jenkins.

Shared libraries - Example





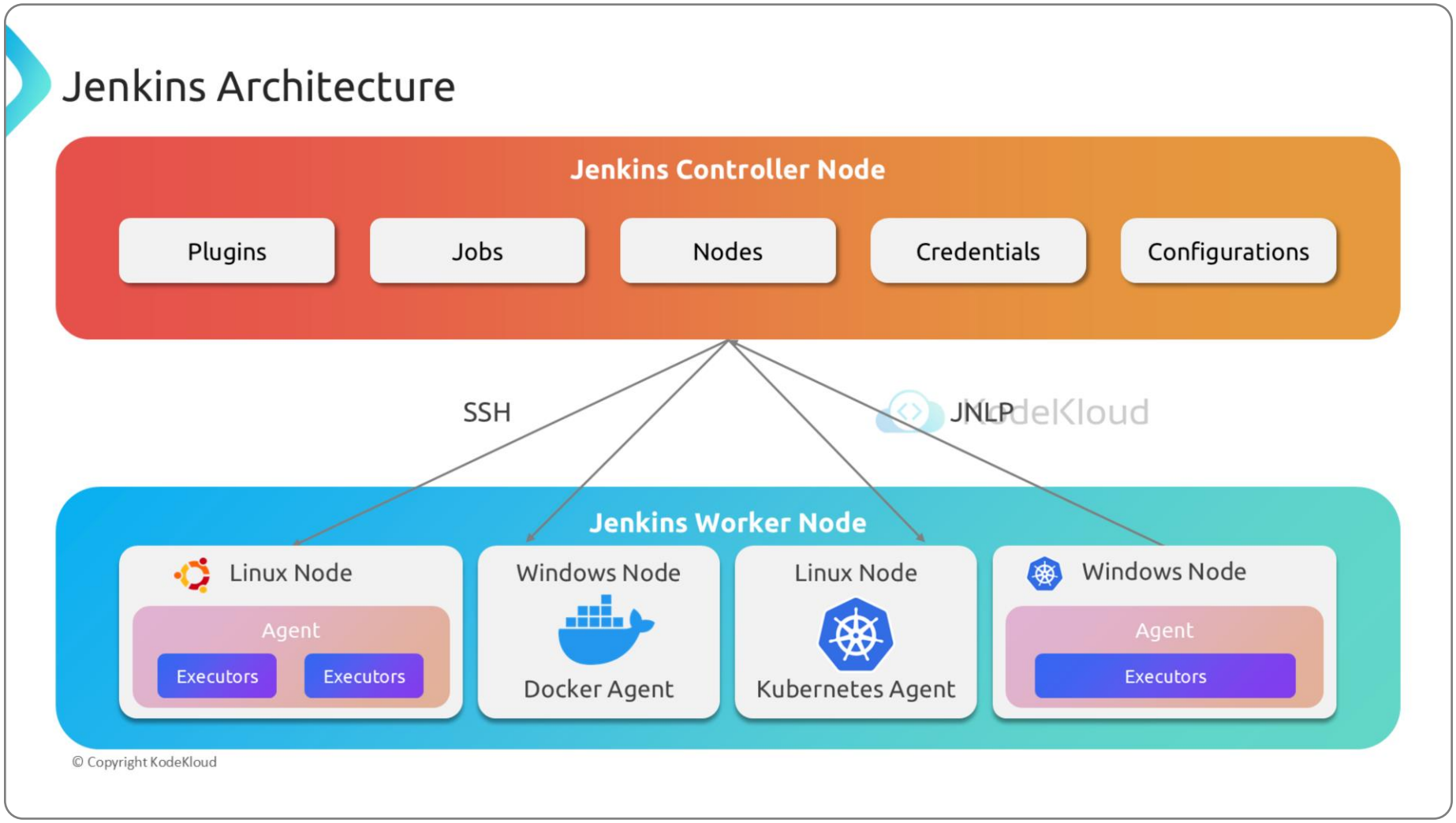
Once you've configured the shared library in Jenkins, it's time to use it in your pipeline. Here's how to load it in your Jenkinsfile:
@Library Directive: Include the @Library directive at the beginning of your Jenkinsfile.
Specify Library Name: Inside the @Library directive, mention the name you gave to the shared library during configuration.

The @Library directive in your Jenkinsfile makes the shared library accessible.
This allows you to replace the old shell command for displaying the welcome message with welcome().

The welcome() call invokes the call method defined within your welcome.groovy file, effectively displaying the welcome message from the shared library.

This approach streamlines your pipeline by removing the need for hardcoded messages and promoting code reuse.

Types of Agents

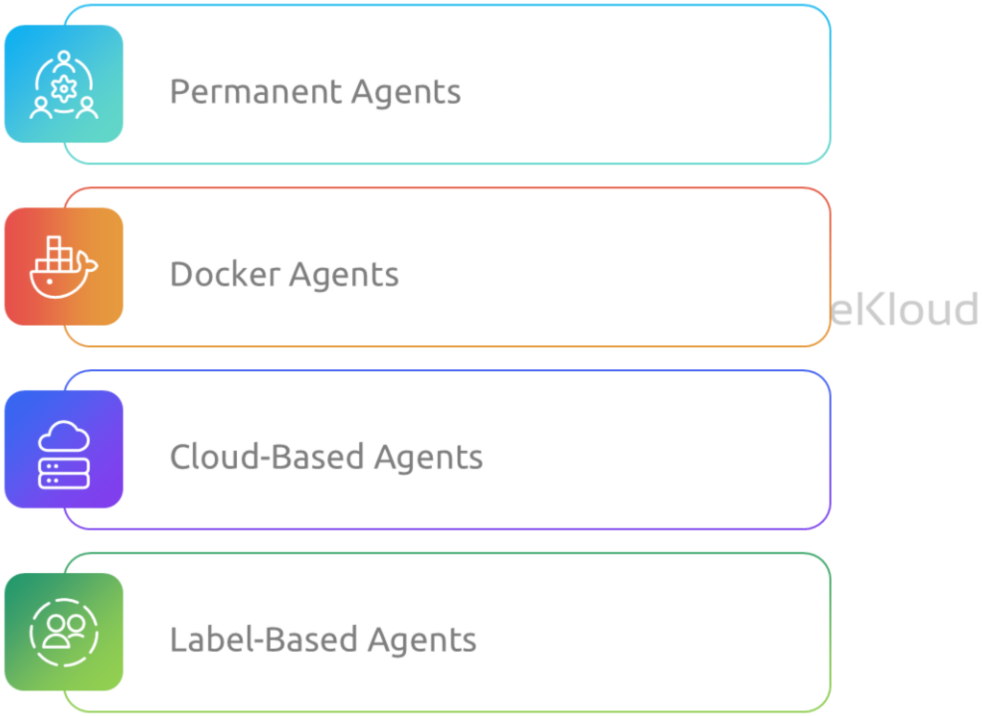


We talked about agents back in Jenkins beginner course. To give a quick recap

Agent acts as an extension of Jenkins, managing the task execution by using executors on a remote node.
An agent represents a specific way a node connects to the controller. It defines the communication protocol and authentication mechanism used.
For example, you might have a Java agent connecting using JNLP or a SSH agent connecting using the SSH protocol.
Tools required for builds and tests are installed on the node where the agent runs;

Docker agent
We can leverage Docker containers as build agents.
Instead of installing tools and dependencies directly on a dedicated worker node, you define a Docker image containing all the necessary software for your build jobs.
This is Well-suited for scenarios where build jobs require specific software versions or have complex dependencies.
Finally each build job runs in a clean, isolated container, ensuring consistent environments and preventing conflicts between projects.

Types of Agents



Jenkins agents are more like worker machines. These machines, whether physical, virtual, or even containers, connect to the Jenkins controller and serve as the execution engines for your build pipelines.

Different types of agents offer unique ways to manage these worker machines, catering to diverse CI/CD needs. Let's delve into the most popular agent types

Permanent Agents : Imagine a dedicated team of specialists, each equipped with a specific skillset (e.g., Java development

tools). These agents are always on standby, connected to the Jenkins master, waiting to be assigned tasks. Perfect for scenarios where consistent environments with certain tools are crucial, but keep in mind they can be resource-intensive if not constantly in use.

Docker Agents: Enter the world of containers! Docker agents are like temporary contractors, spun up from pre-configured Docker images. Each container provides an isolated environment with the exact tools needed for a particular build. Think of it as hiring a specialist for a specific task, then dismissing them when it's done. This approach promotes efficient resource utilization and consistent build environments.

Cloud-Based Agents (AWS, Azure, GCP): Picture a vast pool of on-demand workers. Cloud-based agents leverage the elasticity of cloud providers like AWS or Azure. When a build arrives, Jenkins seamlessly provisions a virtual machine, runs the job, and then terminates the instance. This is ideal for scaling your CI/CD infrastructure to meet fluctuating build demands without managing physical servers. Pay only for what you use!

Label-Based Agents: Let's ditch the rigid hierarchies! Label-based agents break free from the concept of specific machines. These agents are identified by labels that reflect their capabilities (e.g., "java," "windows"). Imagine a flexible workforce where skills are the priority, not physical location. Jenkins can then match the labels defined in your pipeline with available agents that possess those skills, ensuring the right tool is assigned for the job.

Using agents in Pipeline

Terminal

```
$ cat Jenkinsfile
pipeline {
  agent any  // Use any available agent
  stages {
    stage('Build') {
      steps {
        sh 'echo Running on ${NODE_NAME}'        }
    }
  }
}
```

Terminal

```
$ cat Jenkinsfile
pipeline {
  agent {
    docker {
      image 'node:latest'  // Use a Docker image with Node.js
      args '-v $HOME/.npm:/root/.npm'  // Optional: Mount volume
    }
  }
  stages {
    stage('Build') {
      steps {
            ……..
```

Terminal

```
$ cat Jenkinsfile
pipeline {
  agent {
    label 'my-agent'  // Run on agent with label "my-agent"
  }
  stages {
    stage('Build') {
      steps {
        sh 'echo Running on ${NODE_NAME}'
      }
    }
  }
}
```

Terminal

```
$ cat Jenkinsfile
pipeline {
  agent {
    label 'my-agent'  // Run on agent with label "my-agent"
  }
  stages {
    stage('Build') {
      agent { label 'nodejs-agent'}
      steps {
        sh 'echo Running on ${NODE_NAME}'
      }
    }
  }
```

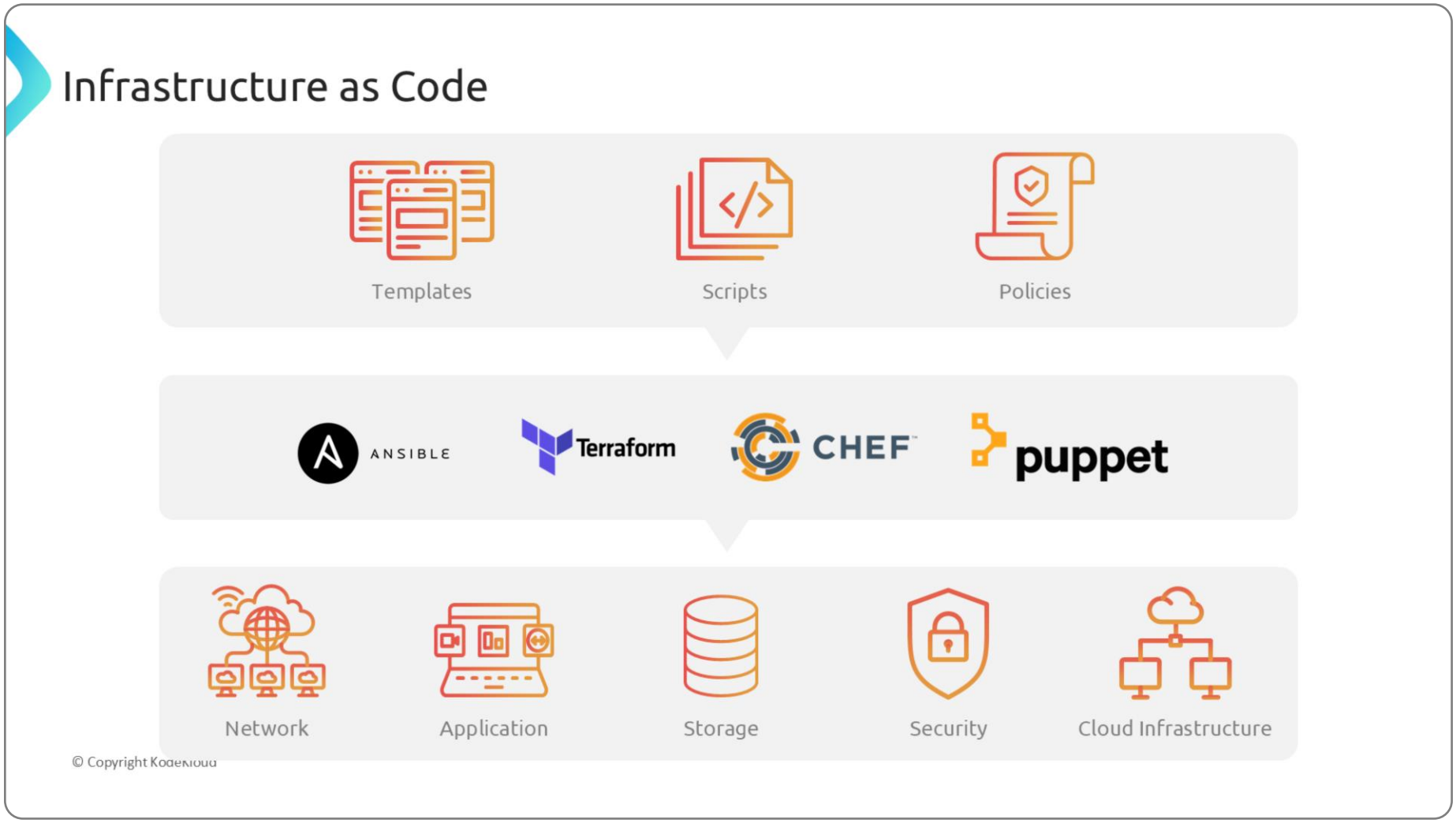
These examples showcase various ways to define agents in your Jenkinsfile using Declarative Pipeline syntax:

This example allows Jenkins to run the pipeline on any available agent within Jenkins.
This example instructs Jenkins to run the pipeline on an agent that has the label "my-agent." Make sure you have an agent configured with that label in your Jenkins environment.
This example defines a Docker agent that uses the node:latest image. You can also optionally specify arguments for the container, such as mounting a volume for your npm cache. This approach ensures a consistent and isolated build

environment with Node.js pre-installed.
This Jenkinsfile employs a unique agent strategy. It sets a default agent with the label "my-agent" at the root level. This means all stages will use this agent unless explicitly overridden. However, the Build stage demonstrates a clever twist. It defines a dedicated agent labeled "nodejs-agent" within the stage itself. As a result, the Build stage will use this specific "nodejs-agent" to run its steps, prioritizing it over the default "my-agent." This approach offers flexibility:

- Default agent for most stages.
- Specific agent tailored for the Build stage, likely with Node.js pre-installed.

Jenkins Configuration as Code (JCasC)



Before we talk about Jenkins Configuration as Code (JCasC), lets review Infrastructure as Code

Building a strong foundation is essential for any organization's IT systems. As your business scales and adapts, you'll need an infrastructure that can keep up - flexible, reliable, and able to handle new demands.
Manually managing this infrastructure can be slow, error-prone, and inefficient. This is where Infrastructure as Code (IaC) comes in. Tools like Ansible, Terraform, and Puppet let you define your infrastructure using code, making it easier to manage, repeatable, and scalable.

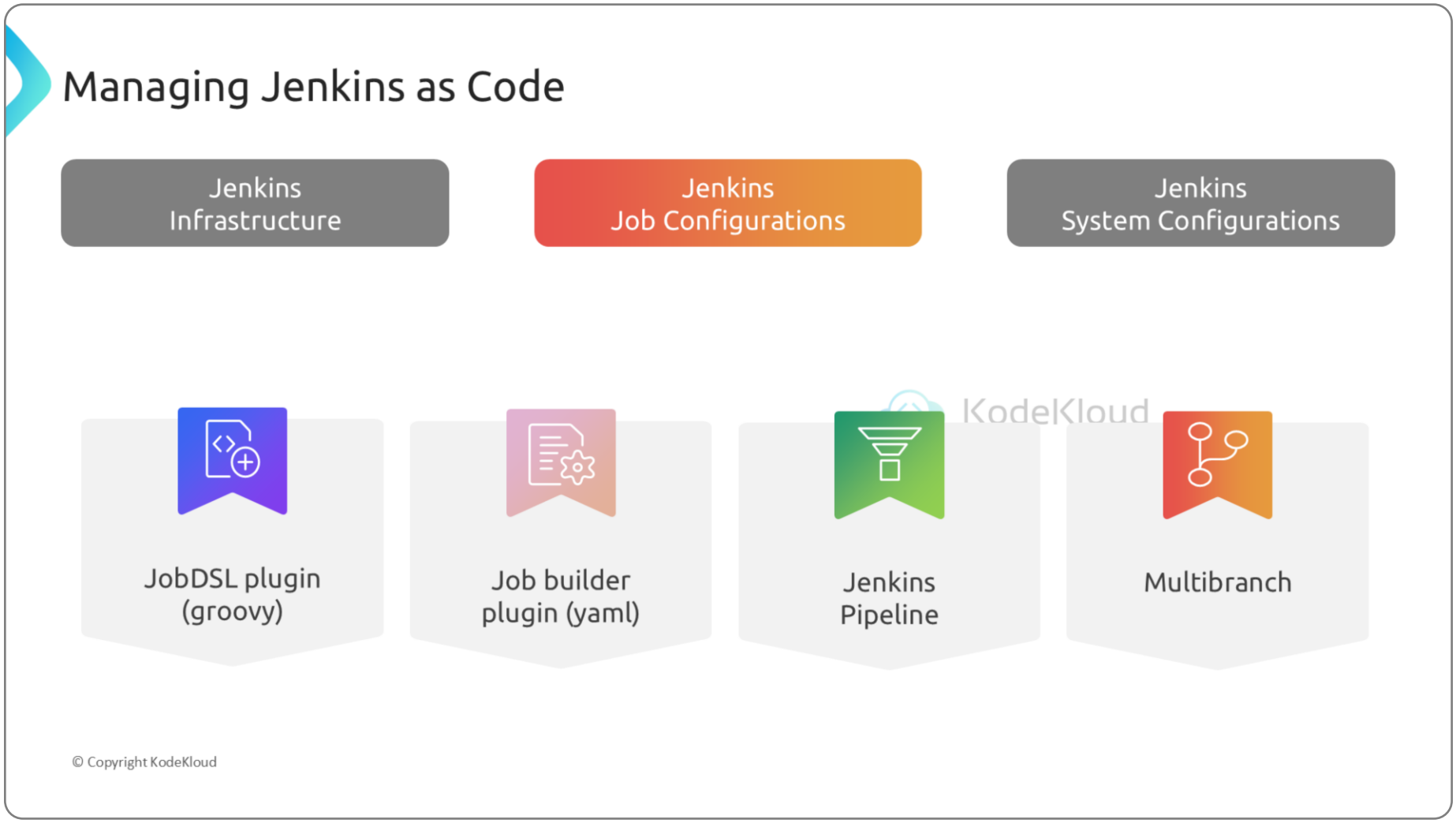
Just like IaC simplifies infrastructure management, there's a similar approach for Jenkins configuration: Jenkins Configuration as Code (JCasC). Let's explore how JCasC streamlines managing your Jenkins setup.

Managing Jenkins as Code

Jenkins offers the ability to manage various aspects of its functionality through code. This includes:
Jenkins Infrastructure: Define and manage the underlying infrastructure for Jenkins, such as agents and resources.
Jenkins Job Configurations: Specify the steps and settings for each automated task (job) within Jenkins using code.
Jenkins System Configurations: Control global settings for Jenkins itself, like security options or user management, through code.

There's a wide range of tools at your disposal to manage Jenkins infrastructure. These options include:
Command-Line Tools: The Jenkins CLI lets you manage your Jenkins server directly from the terminal.
RESTful API: Integrate with Jenkins using its programmatic interface for more flexibility.
Client Libraries: Utilize official client libraries for popular programming languages like Java, Python, and Go to manage Jenkins from your code.
Infrastructure as Code (IaC) Tools: Leverage popular IaC tools like Ansible and Chef to define and provision your Jenkins infrastructure through code.

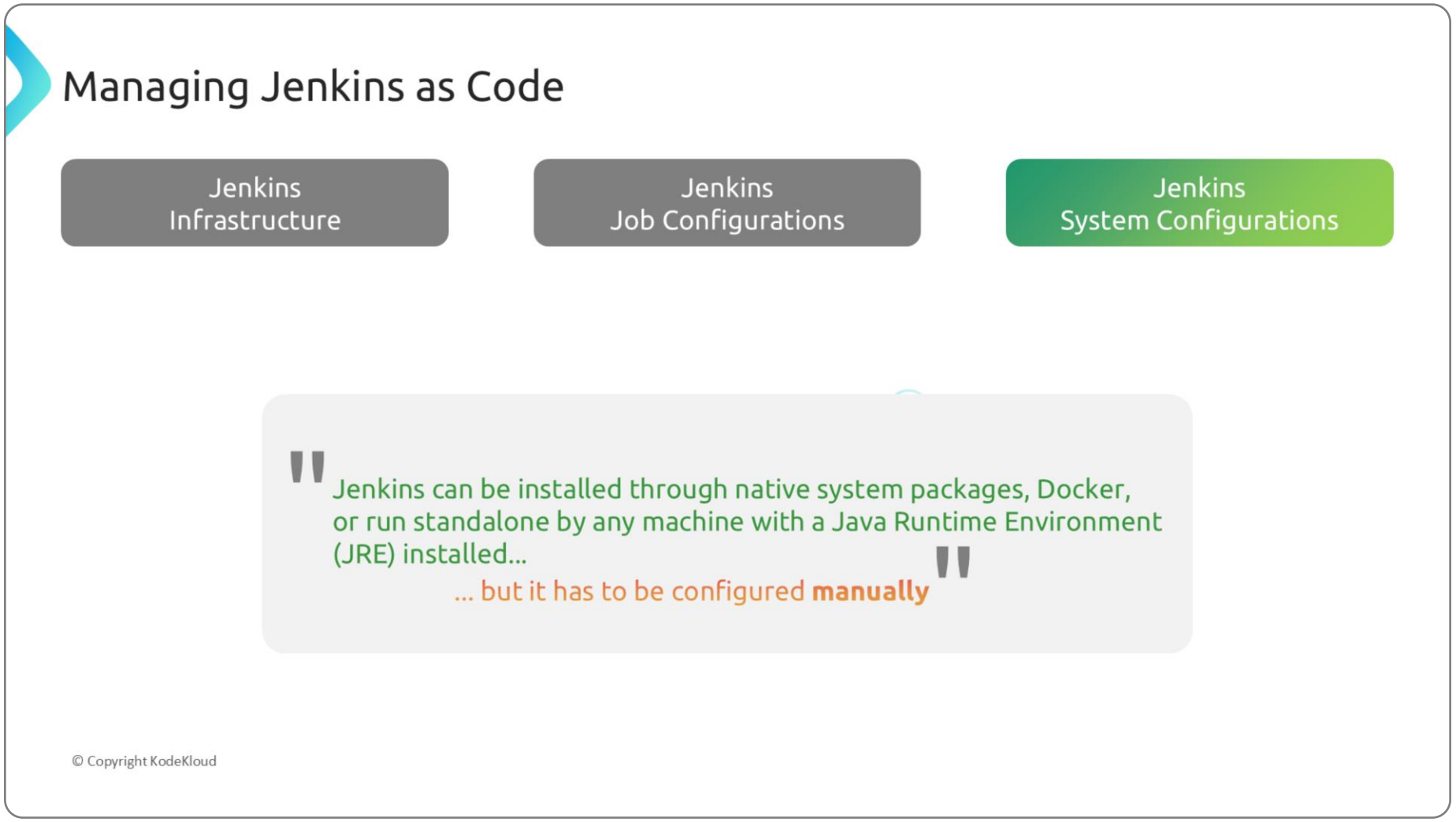
Containerization: Package your Jenkins server as a Docker image for a portable and consistent environment.



Jenkins excels at managing build processes, and its user interface is popular for configuring individual jobs. However, as the number of jobs grows, UI-based management becomes cumbersome and impractical.
Here are some approaches to tackle this challenge:
Job DSL Plugin (Groovy): This plugin allows defining jobs in a human-readable Groovy script. This code-based approach is more manageable for large numbers of jobs and translates intuitively from the web UI configuration. It's a pioneering plugin in managing Jenkins configuration as code, paving the way for others like Jenkins Pipeline and Configuration as Code.
Job Builder Plugin (YAML/JSON): This plugin leverages simpler formats like YAML or JSON to define job configurations.

These files provide a clear and concise way to manage jobs.
Jenkins Pipeline: This built-in feature offers a powerful way to define job workflows using code. It allows for more complex and flexible automation compared to traditional job configurations.
Multibranch Pipeline: This feature extends Jenkins Pipeline by automatically configuring jobs based on changes detected in your source code repository.

This combined approach empowers you to manage a large number of Jenkins jobs effectively, moving beyond the limitations of UI-based configuration.



Talking about Jenkins System Configuration,

I would like to quote a message from Manage Jenkins Configuration as Code a video of the 2018 DevOps World presentation that introduced the JCasC feature.

"Jenkins can be installed through native system packages, Docker, or run standalone by any machine with a Java Runtime

Environment (JRE) installed..."

"... but it has to be configured manually“

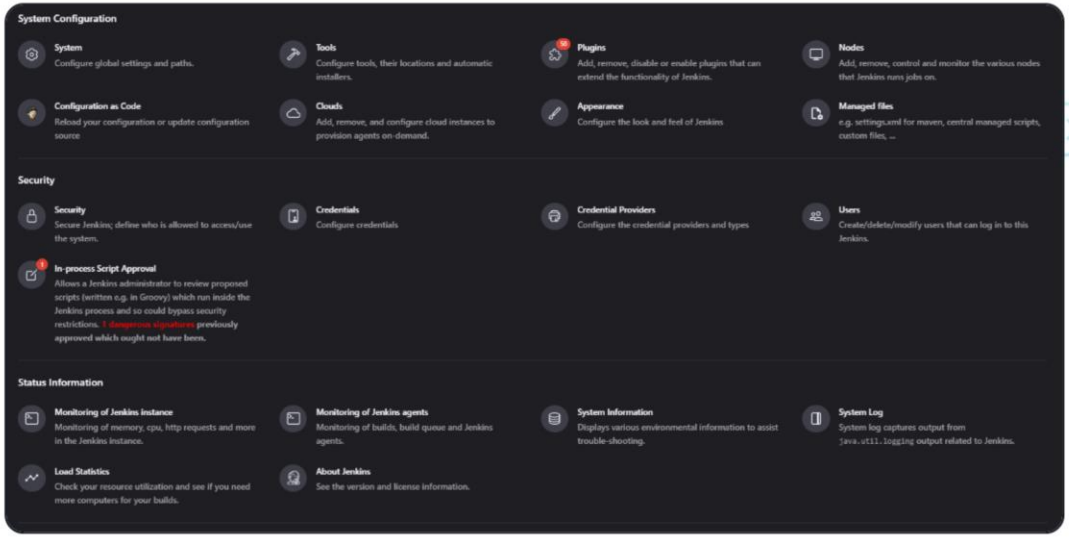
Look Ma! No Hands! — Manage Jenkins Configuration as Code

Managing Jenkins as Code

Jenkins Infrastructure

Jenkins Job Configurations

Jenkins System Configurations



01 | Time-consuming

02 | Error-prone

03 | Inefficient

Talking about Jenkins system configurations, the Jenkins interface offers a wide range of options for configuration, encompassing aspects like credential management, node setup, plugin installation, tool configuration, and system settings. However, navigating through menus, clicking buttons, and manually modifying numerous fields can be:
Time-consuming: The process can be slow, especially for complex configurations.
Error-prone: Repetitive manual tasks increase the risk of mistakes.
Inefficient: Maintaining consistency across multiple Jenkins instances becomes cumbersome.

Before the introduction of Jenkins Configuration as Code (JCasC), Apache Groovy init scripts were a common approach for experienced users. These scripts provided a powerful way to automate configuration by directly interacting with the Jenkins API. While offering great flexibility, they also required:

In-depth knowledge of Jenkins internals: Understanding the intricacies of the API was crucial for effective script development.
Groovy scripting expertise: Scripting fluency was essential to properly interact with the API.

Overall, traditional configuration methods lacked the simplicity and maintainability offered by JCasC.

Managing Jenkins as Code

Jenkins Infrastructure

Jenkins Job Configurations

Jenkins System Configurations

Apache Groovy init scripts

KodeKloud

In-depth knowledge of Jenkins internals

Groovy scripting expertise

Before the introduction of Jenkins Configuration as Code (JCasC), Apache Groovy init scripts were a common approach for experienced users. These scripts provided a powerful way to automate configuration by directly interacting with the Jenkins API. While offering great flexibility, they also required:

In-depth knowledge of Jenkins internals: Understanding the intricacies of the API was crucial for effective script development.

Groovy scripting expertise: Scripting fluency was essential to properly interact with the API.

Overall, traditional configuration methods lacked the simplicity and maintainability offered by JCasC.

Jenkins Configuration as Code (JCasC)

YAML

Effortless Configuration

Enhanced Efficiency

JCasC offers a powerful alternative to the traditional UI-based configuration of Jenkins. It defines configuration parameters in a user-friendly YAML file, mirroring the settings you'd make in the web interface. This essentially captures your configuration choices in code, enabling:
Effortless Configuration: Manage Jenkins controllers without navigating menus or clicking buttons.
Enhanced Efficiency: Streamline configuration management and simplify your workflow.

Jenkins Configuration as Code (JCasC)

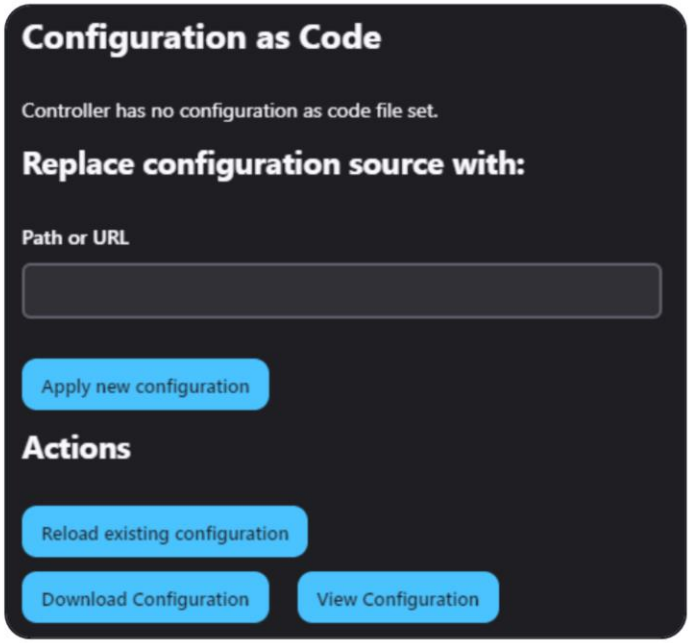
- Version Control Integration
- Repeatable Deployments
- Reduced Errors

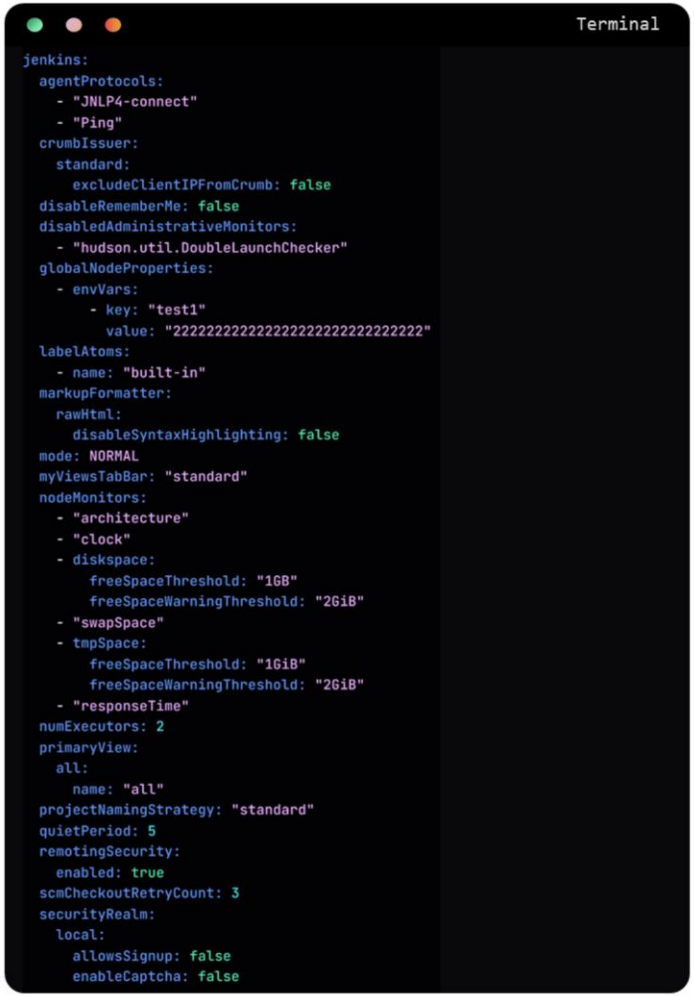
JCasC unlocks several advantages:
Version Control Integration: Store your JCasC files alongside your application code in a Version Control System (VCS) like Git. This allows you to track changes, revert to previous configurations if needed, and collaborate effectively with your team. Repeatable Deployments: Eliminate the need for manual configuration on every new Jenkins instance. Simply deploy your JCasC files, and your Jenkins environment will be automatically configured.
Reduced Errors: Minimize human error by eliminating repetitive manual configuration tasks. YAML syntax errors are also easier to identify and fix compared to UI-based configuration mistakes.

Jenkins Configuration as Code (JCasC)

Getting Started with JCasC
The first step to leverage JCasC is to install the "configuration-as-code" plugin on your Jenkins controller. This plugin bridges the gap between your YAML configuration files and your Jenkins environment.

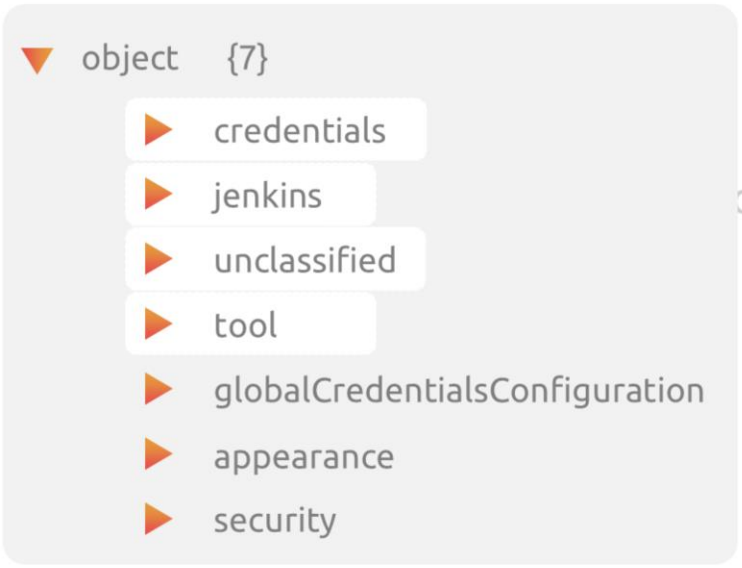
Jenkins Configuration as Code (JCasC)

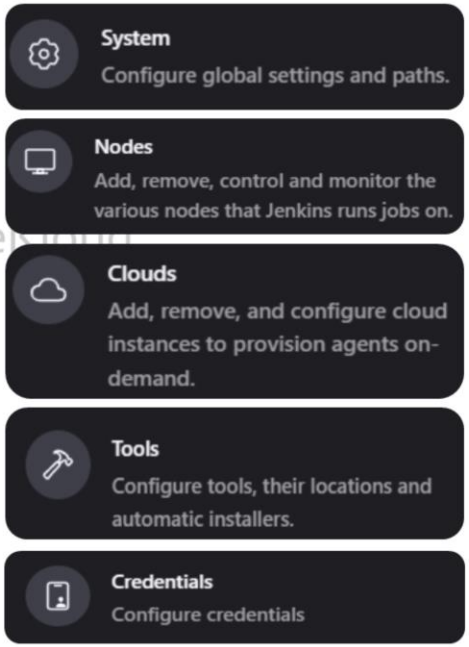




Once you've installed the Configuration as Code plugin, a new section named "Configuration as Code" will appear within the "System Configuration" area of your Jenkins dashboard under "Manage Jenkins." Clicking "View Configuration" generates a comprehensive YAML file representing your current Jenkins controller configuration.
While this file might be quite extensive, it's often usable without modification. However, it's recommended to customize it before deployment to suit your specific needs. Even if you choose not to modify it initially, consider pushing the unmodified version to your Source Control Management (SCM) system to maintain a historical record of your configuration. This can be helpful for future reference or rollbacks.

Jenkins Configuration as Code (JCasC)



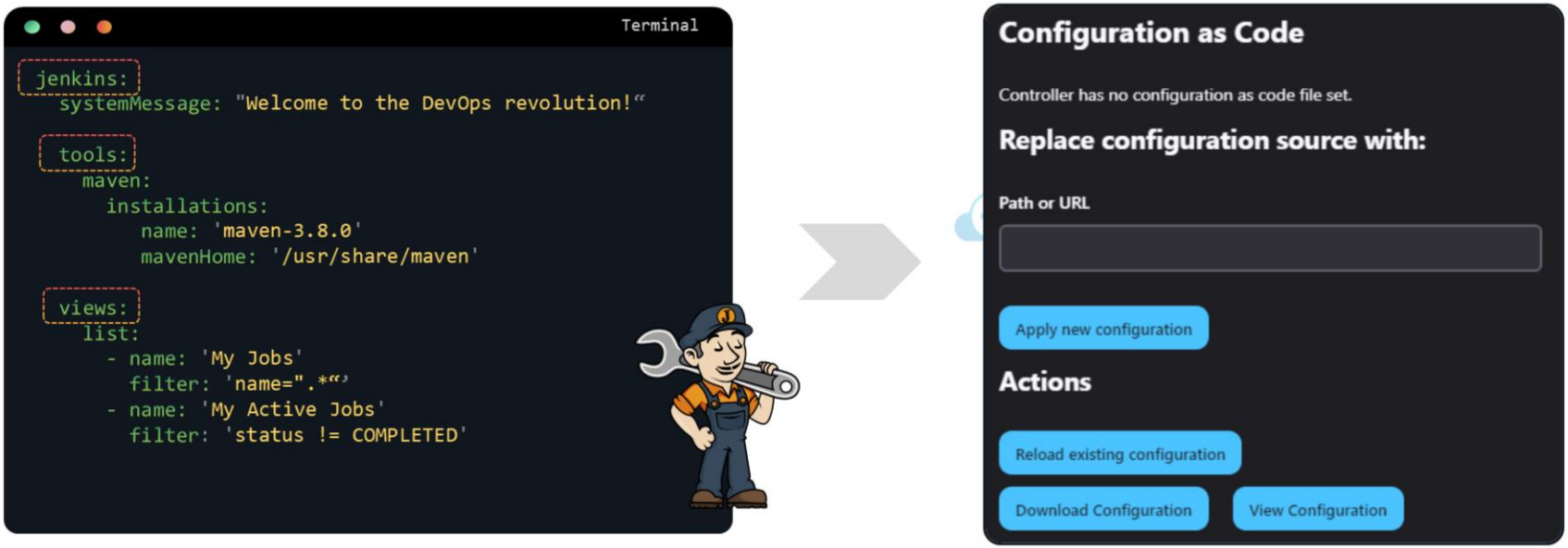


The JCasC YAML file serves as a blueprint for your Jenkins configuration. It typically consists of four main sections:

1. **jenkins:** This section defines the core Jenkins object, encompassing settings you would typically configure through the "System Configuration" and "Configure Nodes and Clouds" screens in the UI.
2. **tool:** This section manages build tools, mirroring configurations typically set via the "Tools" screen in the UI.
3. **unclassified:** This section acts as a catch-all for other configurations, including settings for installed plugins.
4. **credentials:** This section handles credential definitions, similar to what you would configure through the "Manage Credentials" screen in the UI. It's important to note that additional sections might be present in your specific JCasC

YAML file depending on the configuration of your Jenkins controller and any plugins you have installed. These additional sections will reflect the configuration options provided by those plugins.

Jenkins Configuration as Code (JCasC)



The power of JCasC lies in its ability to tailor your Jenkins configuration to your specific needs.

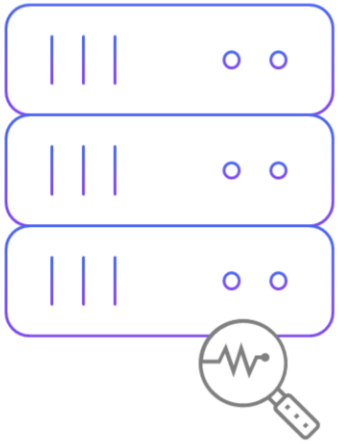
Let's explore some examples of modifications you can make in the JCasC YAML file:
jenkins Section: This top-level section allows you to define global configurations for Jenkins itself. In this example, we've set a custom message to be displayed on the Jenkins dashboard using the systemMessage property.
tools Section: This section lets you manage tools available for use within your pipelines. Here, we've configured a specific Maven installation with a custom name and path.

views Section: This section empowers you to define list views for organizing your jobs. We've created two views in this example: one showcasing active jobs and another displaying all jobs.

This is just a glimpse of JCasC's capabilities. It grants you extensive control over various aspects of Jenkins, including security settings, credential management, plugin configuration, and more.
Once you've finalized your YAML file, you can seamlessly apply the changes through the "Configuration as Code" page within the "Manage Jenkins" section. Notably, applying JCasC modifications doesn't require a Jenkins restart, ensuring a smooth and efficient workflow.

Jenkins Supervision

Jenkins Supervision



Common Monitoring Areas

- System Errors
- Plugin Malfunctions
- Pipeline Code Issues

Key Benefits

- Prevent Disruptions
- Reduce Delays
- Maintain Efficiency

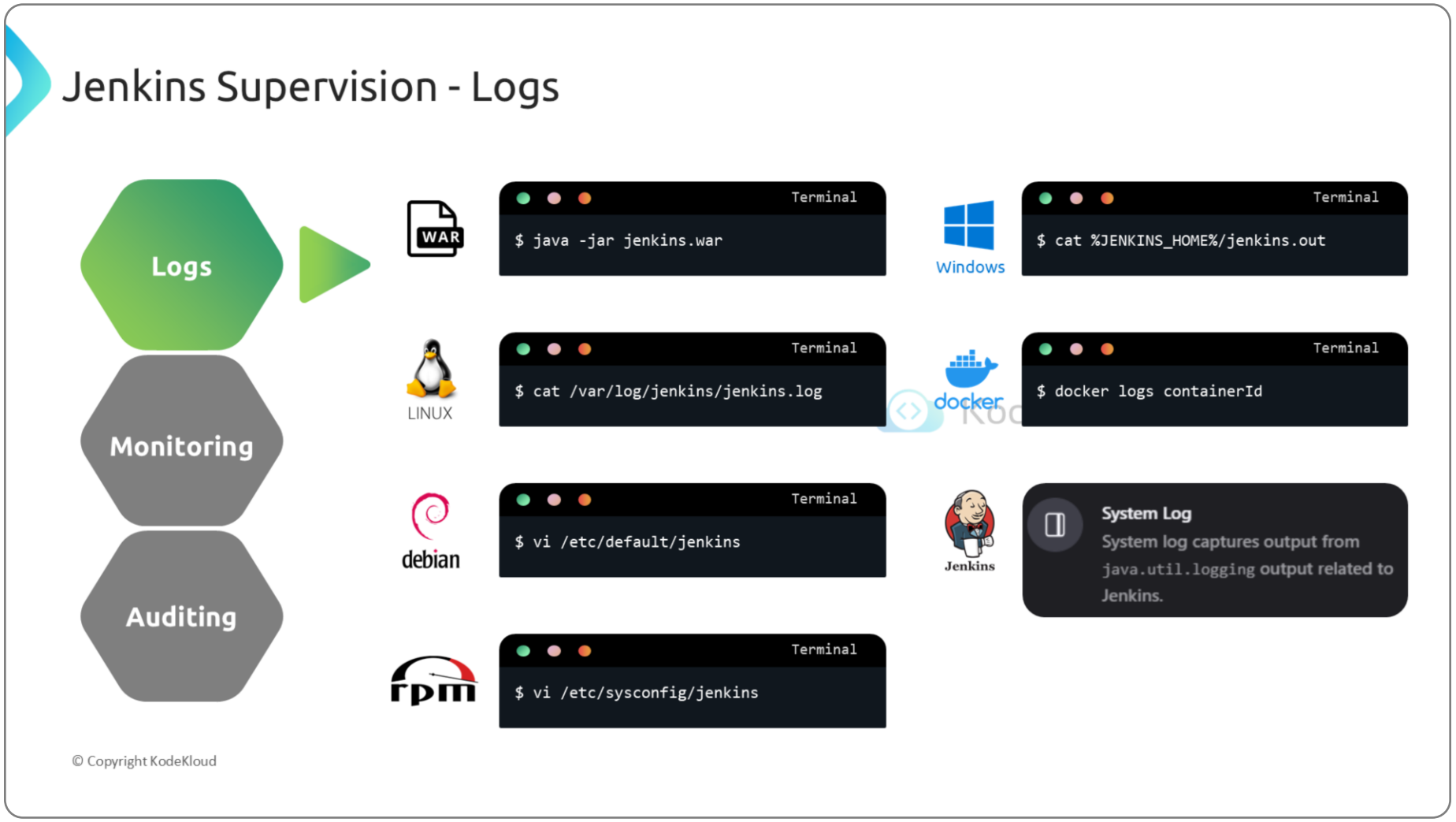
Effective CI/CD pipelines rely heavily on proper Jenkins supervision. Monitoring your Jenkins server proactively helps you anticipate potential problems. This includes identifying system errors, plugin malfunctions, or issues within your pipeline code. By taking a pre-emptive approach, you can prevent disruptions and delays in your testing and deployment stages.

Jenkins Supervision

KodeKloud

In this discussion, we'll explore into three key aspects of Jenkins supervision:

Logs: We'll explore how logs provide valuable insights into your Jenkins activity.
Monitoring Options: We'll discuss various tools and techniques for monitoring Jenkins performance and resource usage.
Auditing Options: We'll cover methods for tracking user activity and configuration changes within Jenkins to maintain security and identify potential misuse.



In the world of software, logs act as a common language, providing crucial information about how any product or application functions.

No matter how you run Jenkins, its logs offer valuable insights into its operation.

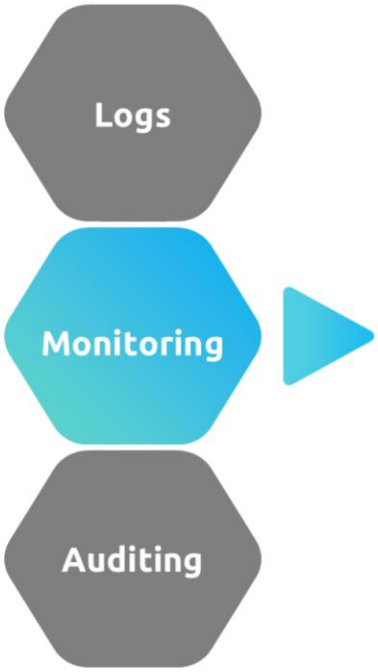
If you are Manually Running Jenkins (war file) using java -jar jenkins.war all Logs are written to the standard console by default.

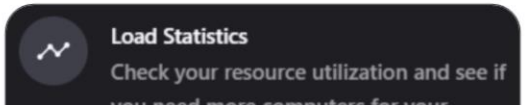
Linux Systems: Look for them in /var/log/jenkins/jenkins.log.
You can customize this location in /etc/default/jenkins (for Debian-based systems) or /etc/sysconfig/jenkins (for RedHat-based systems).

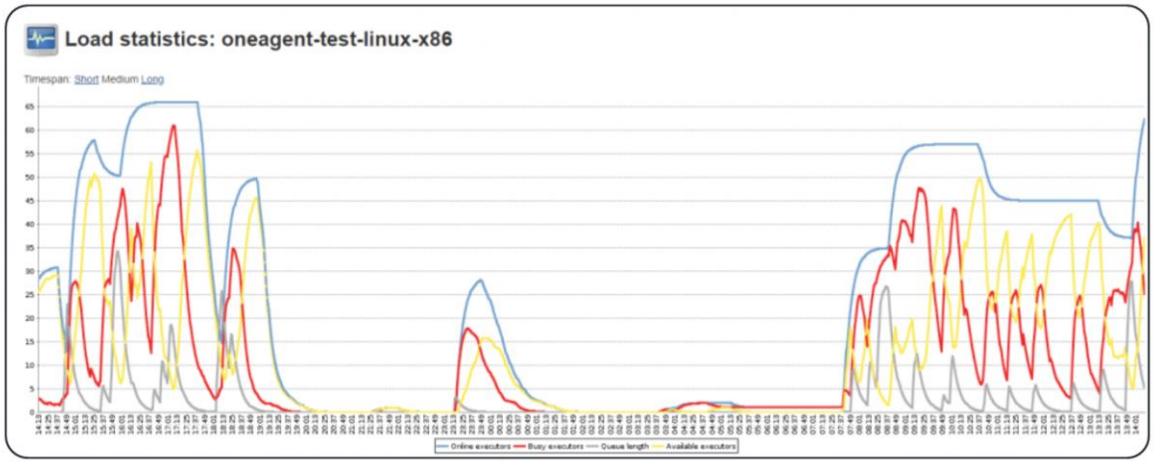
Windows Systems: By default, logs are stored in %JENKINS_HOME%/jenkins.out (standard output) and %JENKINS_HOME%/jenkins.err (standard error). You can adjust this location in the %JENKINS_HOME%/jenkins.xml file.

Jenkins Web Interface: Go to "Manage Jenkins" > "System Log" to view logs directly within the Jenkins UI.

Jenkins Supervision - Monitoring

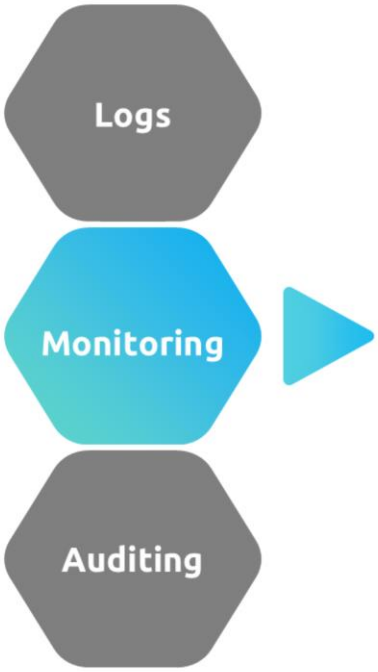


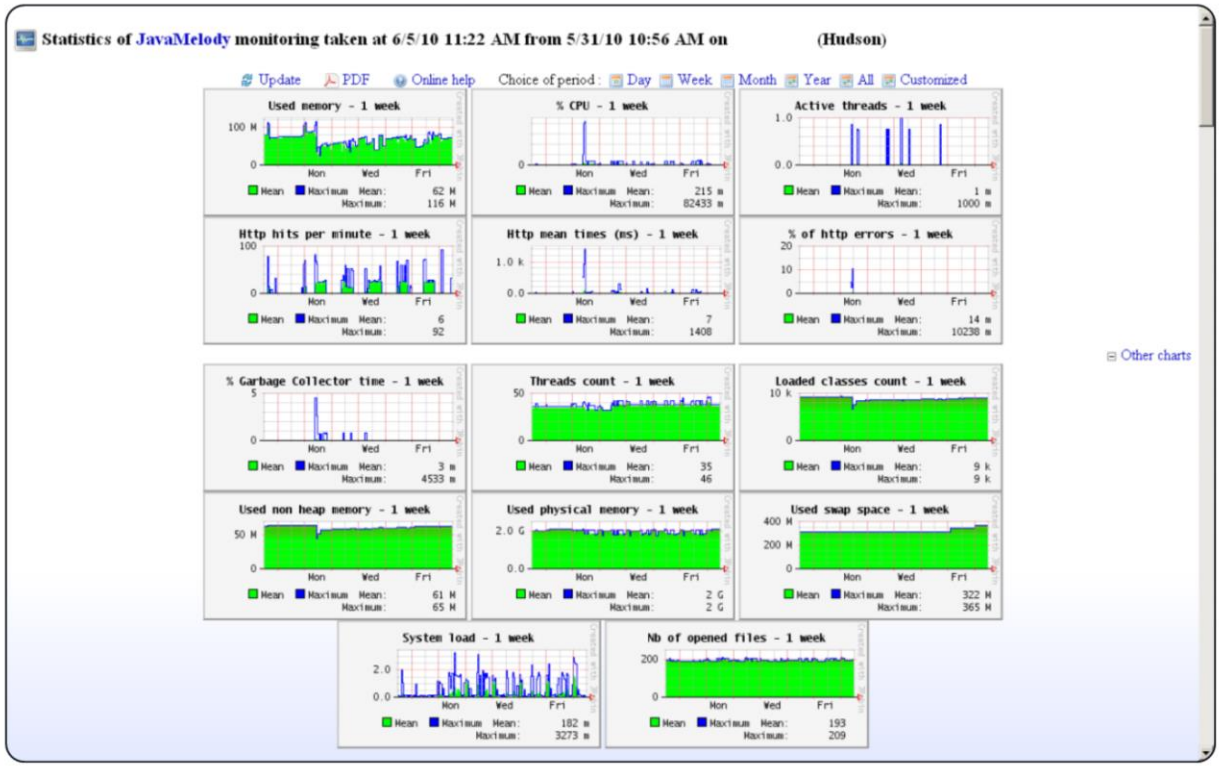




Jenkins offers built-in tools to monitor your server's health. The "Load Statistics" page provides a visual representation of server activity through a graph. This graph tracks four key metrics:
Available Resources: The number of executors ready to handle tasks.
Active Jobs: The number of executors currently running builds.
Queued Jobs: The number of jobs waiting for an available executor.
Overall Server Load: A general indicator of the server's workload.

Jenkins Supervision - Monitoring

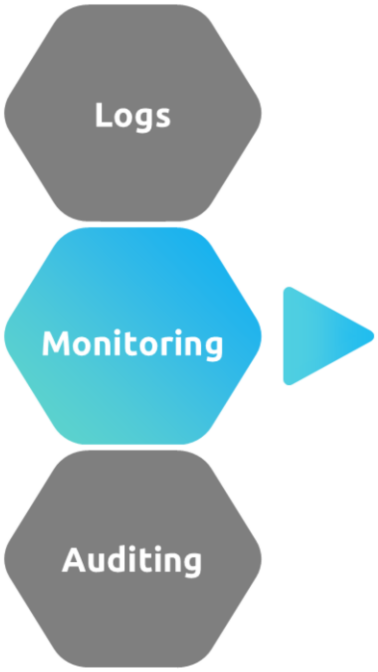




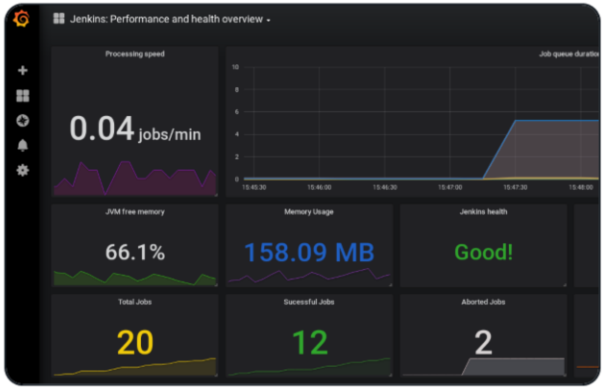
Beyond Built-Ins:
The Jenkins ecosystem offers various open-source plugins to enhance monitoring capabilities. Here are a few popular examples:
Monitoring Plugin: This plugin leverages JavaMelody to provide comprehensive insights. It displays charts for CPU usage, memory consumption, system load, and response times. Additionally, it offers details about HTTP sessions, errors, logs, and even options for garbage collection, heap dumps, and invalid session management.

Disk Usage Plugin: This plugin explores into storage utilization, providing project-wide breakdowns for jobs and workspaces, along with trends in disk usage over time.
Build Monitor Plugin: This plugin focuses on job status, offering a detailed view of selected jobs and even highlighting potential culprits who might have caused build failures (based on user associations with builds).

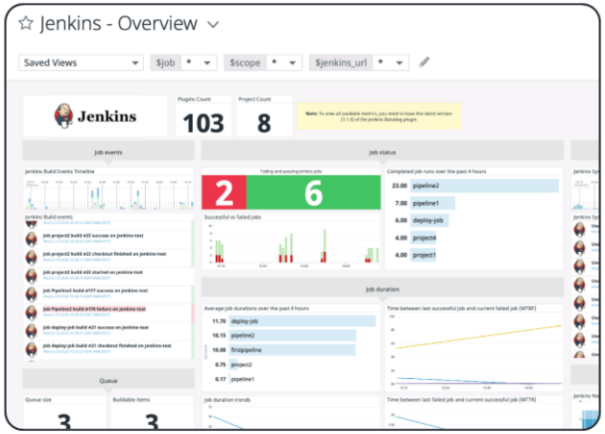
Jenkins Supervision - Monitoring



Grafana



DATADOG

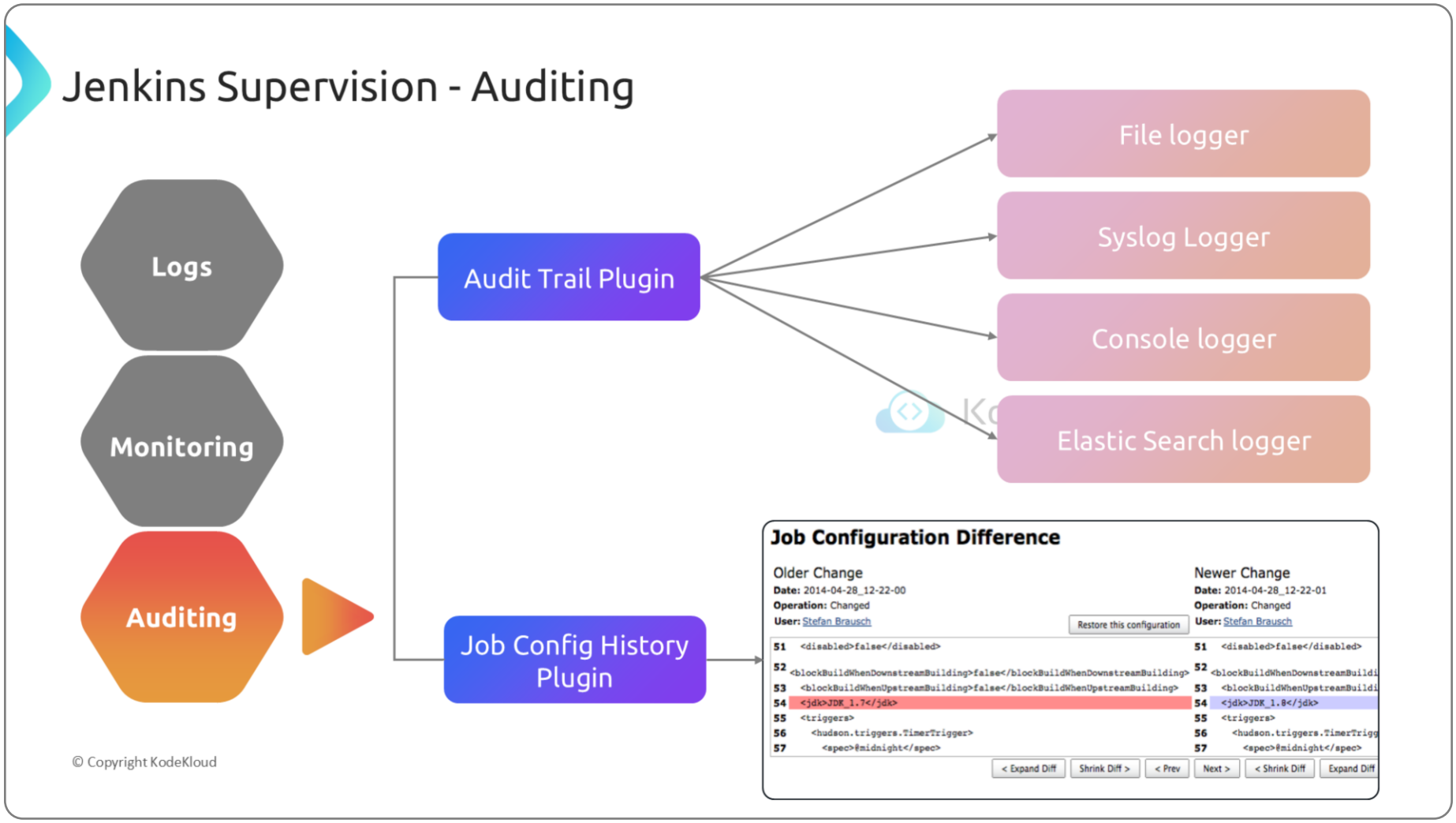


Jenkins goes beyond basic monitoring to offer a range of options for in-depth analysis.

If you already use Datadog or New Relic for monitoring, dedicated plugins integrate Jenkins seamlessly with those platforms.

If you Prefer an open-source approach? The Prometheus plugin for Jenkins, combined with the powerful Grafana visualization tool, provides comprehensive monitoring capabilities.

This flexibility allows you to customize your Jenkins monitoring strategy to perfectly match your existing infrastructure and specific needs.



Jenkins administrators need a robust audit trail to track who did what within the Jenkins server.

We have two open-source plugins for tracking user activity:
Audit Trail Plugin: This plugin adds a dedicated "Audit Trail" section in your main Jenkins configuration. Here, you can define where logs are stored for actions taken on Jenkins, essentially recording "who did what". It focuses on capturing user actions and outputs the information to a file.

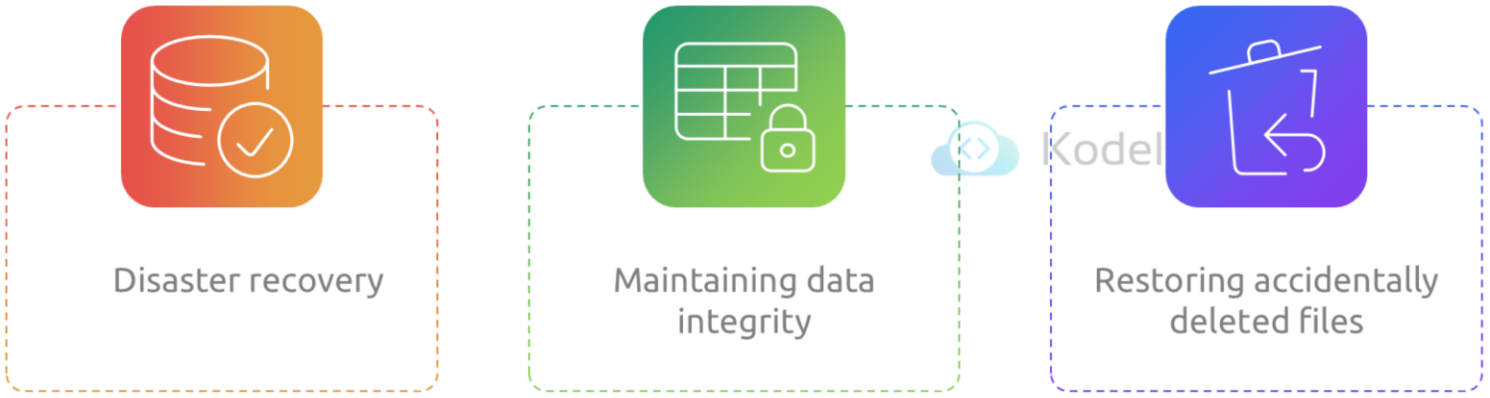
The Audit Trail Plugin offers flexibility in how you capture and store audit logs, providing four distinct logger options:
File Logger: This is the default option, saving audit logs in rotating files on your system. This offers a reliable and easy-to-manage approach.
Syslog Logger: This option transmits audit logs to a dedicated Syslog server. This can be useful for centralizing logs across multiple systems or integrating with existing logging infrastructure.
Console Logger (Debug Only): Primarily for troubleshooting purposes, this option outputs audit logs directly to the console (stdout or stderr). This is not recommended for production environments as it can clutter console output.
Elasticsearch Logger: This option sends audit logs to an Elasticsearch server. This is a powerful choice for large-scale deployments or scenarios where you want to leverage advanced search and analysis capabilities offered by Elasticsearch.

Job Config History Plugin: This plugin acts as a version control system for your Jenkins configurations. It stores all changes made to jobs, folders, and overall system configuration. This includes saving historical versions of config.xml files. The UI offers functionalities to compare differences between versions and even restore previous configurations. While seemingly more comprehensive, it doesn't track job executions or exit statuses.

Due to their complementary nature, many users leverage both plugins. The Audit Trail Plugin provides user activity logs, while the Job Config History Plugin tracks configuration changes. This combined approach ensures you capture both "who did what" and the "when" and "how" of configuration changes and job executions.

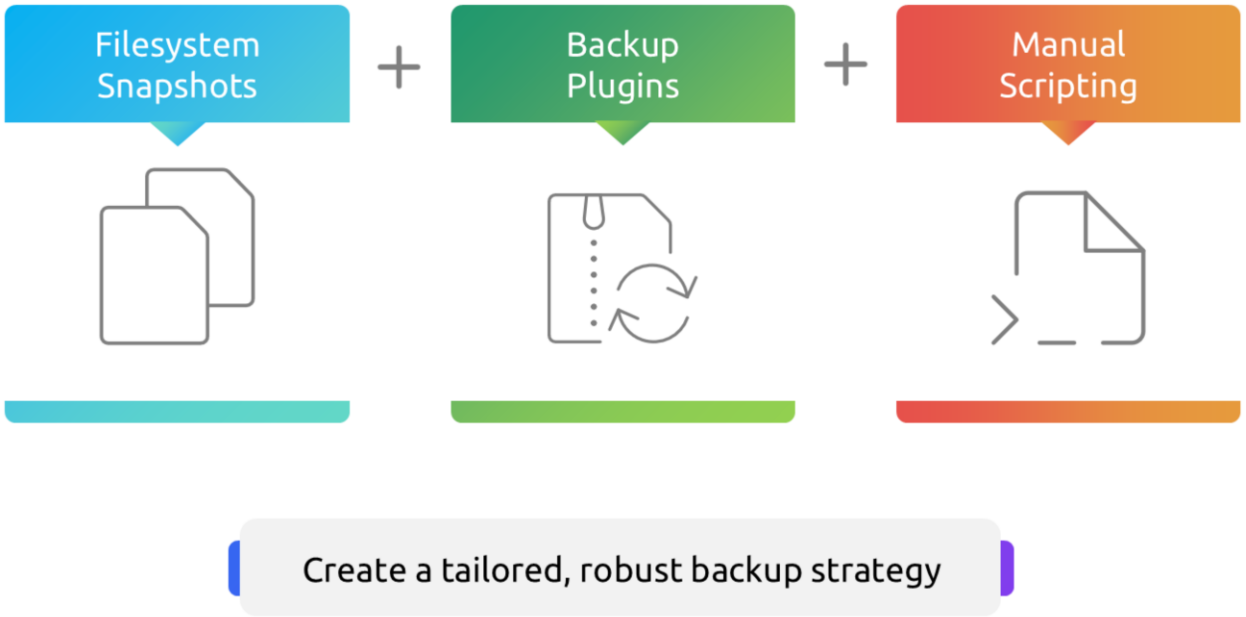
Jenkins backup

Effective Jenkins Backup Strategies



Regularly backing up your Jenkins instance is crucial for disaster recovery, maintaining data integrity and also will be helpful in Restoring a file that's been accidentally deleted.

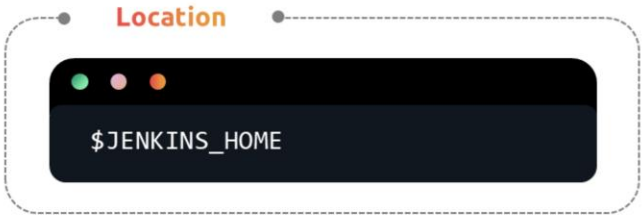
Effective Jenkins Backup Strategies



There are several ways to create backups for your Jenkins instance.
Filesystem Snapshots: These provide the most consistent backups. They're supported by tools like Linux LVM, many cloud platforms, and some storage devices.
Backup Plugins: While various plugins are available, only ThinBackup plugin is actively maintained. Other options might have limitations.
Manual Scripting: You can create a custom script to copy relevant Jenkins files to a backup location. For speed, back up to local storage first, then transfer to a remote location for long-term storage.

By combining these methods, you can create a robust backup strategy tailored to your needs.

Controller Key



Purpose: Encrypts credentials and secrets

The controller key is essential for protecting sensitive information stored in Jenkins. It's used to encrypt credentials and other secrets. Found within the $JENKINS_HOME directory, this key is crucial for restoring your Jenkins instance.

Due to its critical nature, the controller key should be treated like an SSH private key. Never include it in regular backups. Instead, store it separately in a highly secure location. It's a small file rarely modified, but essential for full system recovery. Restore the main Jenkins system first, then apply the controller key backup separately to regain access to encrypted data.

Controller Key

Security Practices

- Treat like an SSH private key
- Store separately in a secure location

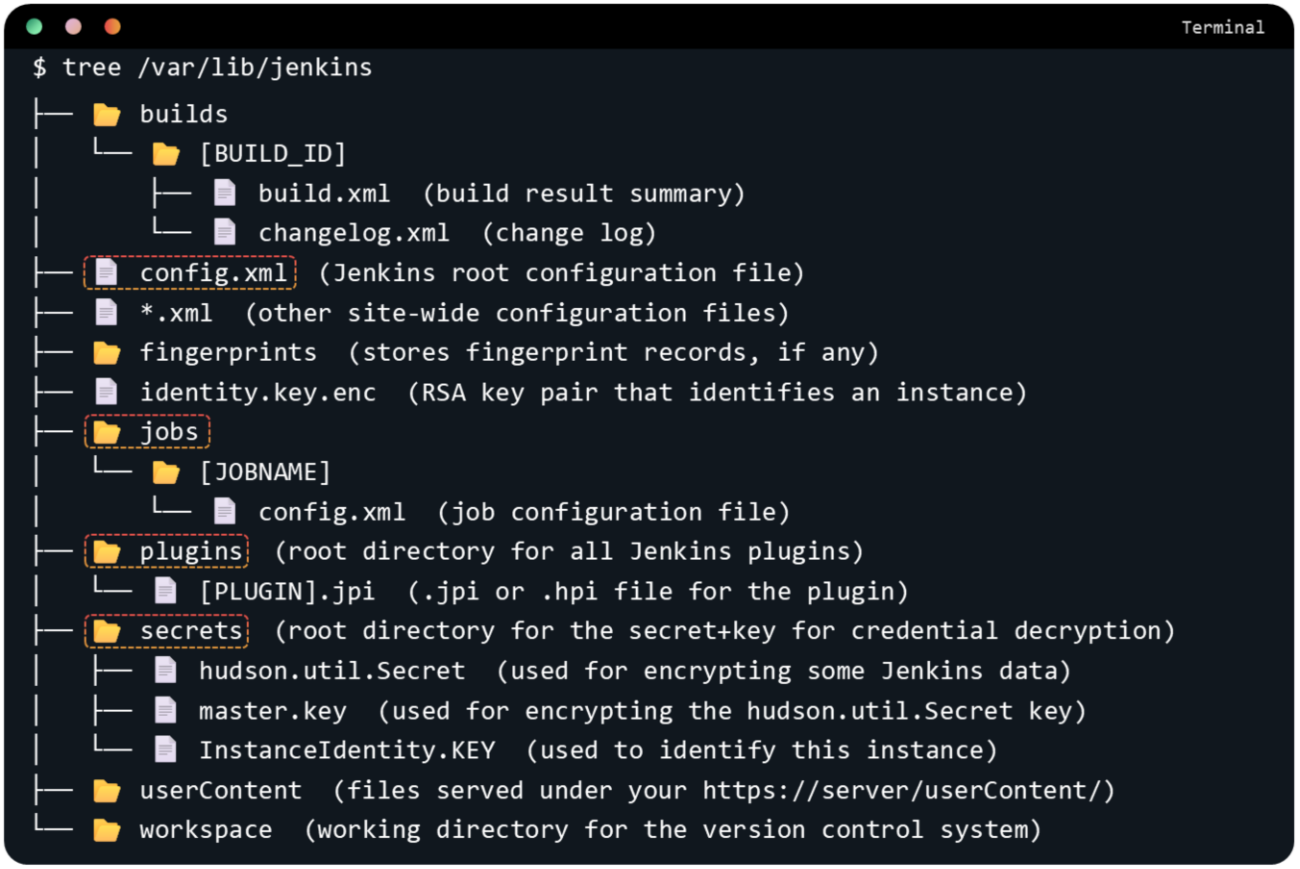
Recovery Process

- Restore Jenkins System
- Apply Controller Key Backup

The controller key is essential for protecting sensitive information stored in Jenkins. It's used to encrypt credentials and other secrets. Found within the $JENKINS_HOME directory, this key is crucial for restoring your Jenkins instance.

Due to its critical nature, the controller key should be treated like an SSH private key. Never include it in regular backups. Instead, store it separately in a highly secure location. It's a small file rarely modified, but essential for full system recovery. Restore the main Jenkins system first, then apply the controller key backup separately to regain access to encrypted data.

JENKINS_HOME



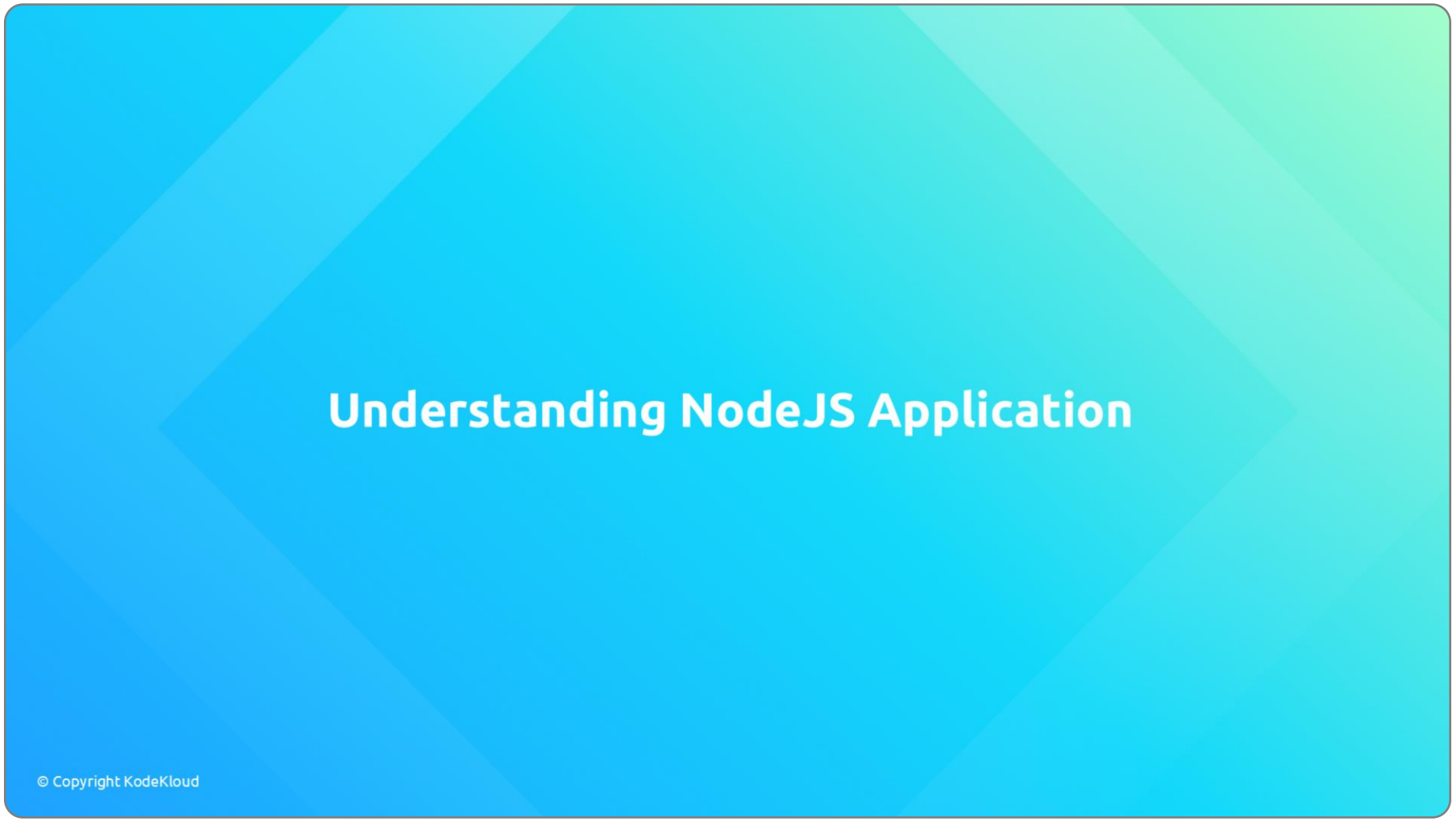
Which Files Should Be Backed Up

The $JENKINS_HOME directory is the heart of your Jenkins instance, housing all essential data and configurations. While backing up the entire directory is possible, it's often unnecessary and can create excessively large backups.

Configuration files: config.xml is the main Jenkins configuration file contains global Jenkins configuration settings. Other configuration files also have the .xml suffix. Backing up $JENKINS_HOME/*.xml to back up all configuration files.

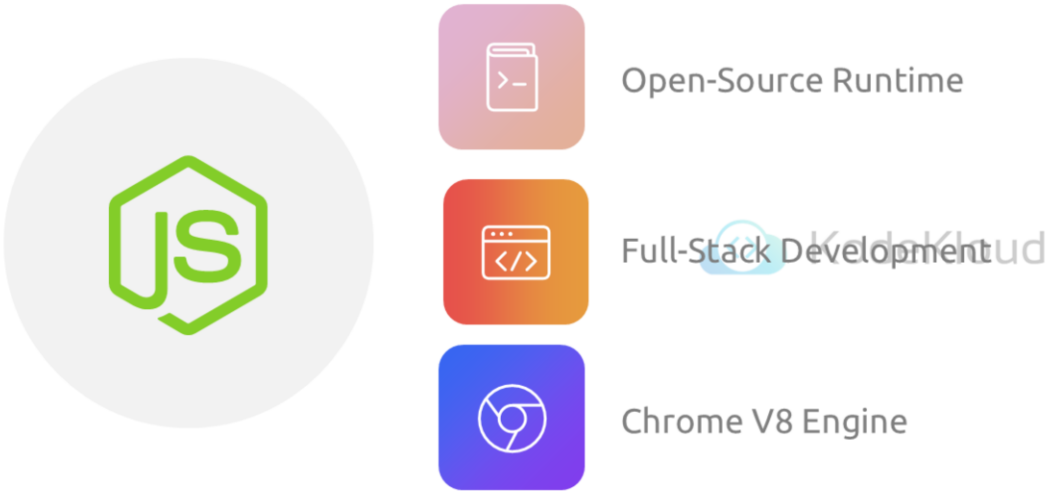
Plugin configuration files (usually with .xml extension): Store plugin-specific settings.
Jobs directory: Contains job definitions and build histories. Consider excluding build artifacts and logs if storage space is a concern.
Users directory: Stores user information and credentials.
Secrets directory: Contains sensitive information like SSH keys and secrets.
Plugins directory: Contains installed plugins. However, plugins can often be reinstalled from the update center if needed.
./builds — Contains build records
./builds/archive — Contains archived artifacts
    Back this up if it is important to retain these artifacts long-term
    These can be very large and may make your backups very large

Avoid backing up temporary data like workspace files, caches, tools and certain plugin components that can be easily redownloaded while restoring Jenkins



Let's take a brief look at Node.js.

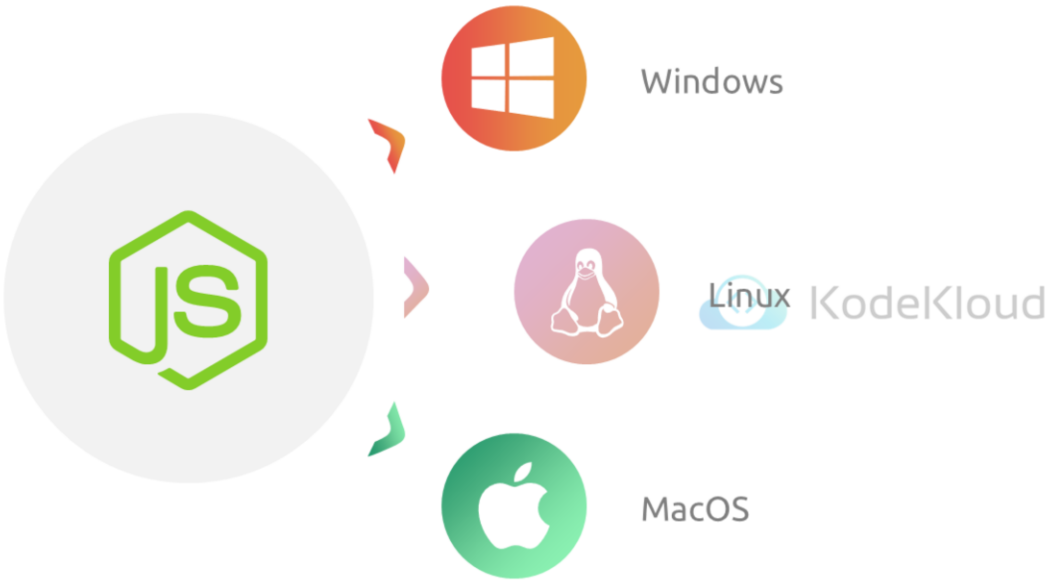
NodeJS



In a later module, we'll utilize the Node.js application to develop a customized GitHub action pipeline.
Node.js is an open-source runtime environment that enables developers to execute JavaScript code outside of web browsers.
Node.js allows developers to create both front-end and back-end applications using JavaScript.
Nodejs is built on the Chrome V8 JavaScript engine. This makes it possible to build back-end applications using the same JavaScript programming language you may be familiar with.

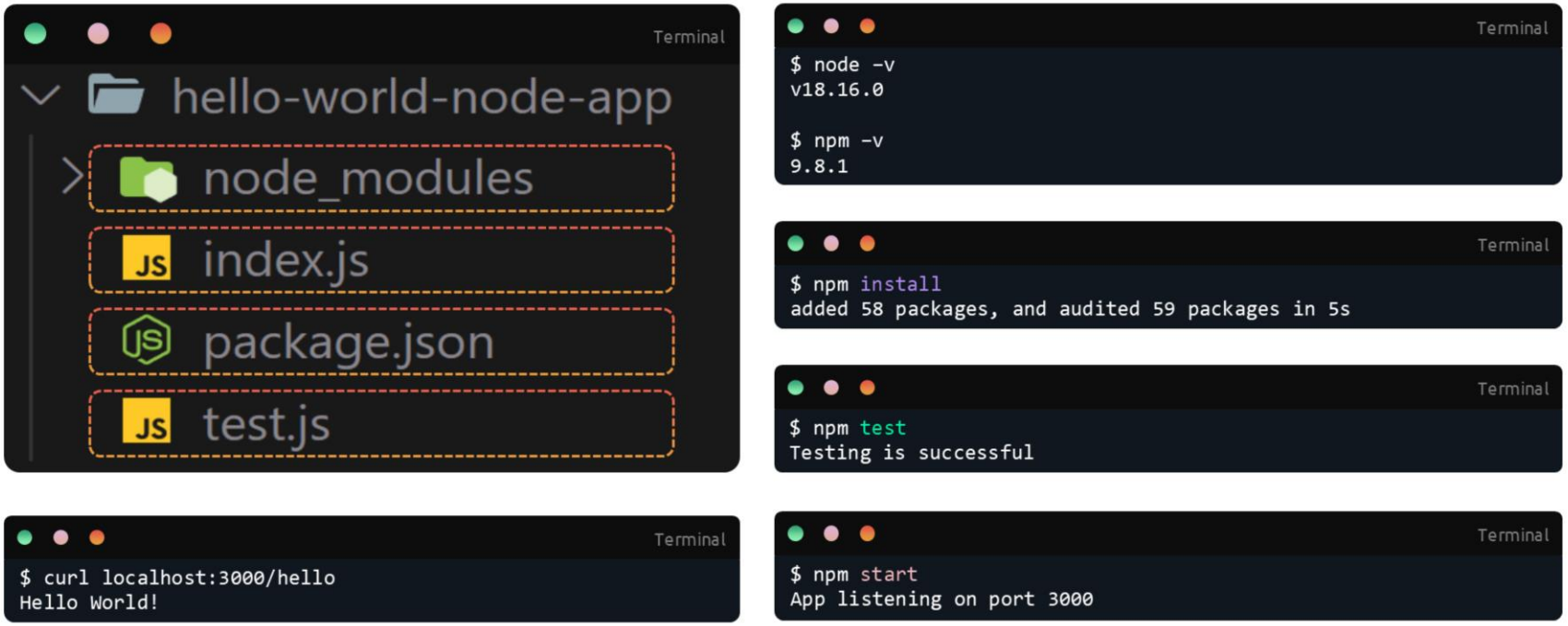
Node.js can be installed on all major platforms, including Windows, macOS, and various Linux distributions.

NodeJS



In a later module, we'll utilize the Node.js application to develop a customized GitHub action pipeline.
Node.js is an open-source runtime environment that enables developers to execute JavaScript code outside of web browsers.
Node.js allows developers to create both front-end and back-end applications using JavaScript.
Nodejs is built on the Chrome V8 JavaScript engine. This makes it possible to build back-end applications using the same JavaScript programming language you may be familiar with.
Node.js can be installed on all major platforms, including Windows, macOS, and various Linux distributions.

NodeJS



What is npm?
npm stands for "Node Package Manager." It is a package manager for JavaScript and Node.js applications. It is used to discover, share, distribute, and manage the various packages, libraries, and dependencies that JavaScript developers use when building web and server-side applications.

Npm gets installed as part of the Node.js installation.

Let's explore a sample Node.js project to learn how to run tests and execute node.js apps.
This is a basic, minimal Node.js application that displays 'Hello World.'

A package.json file in a Node.js project is a metadata file that includes essential information about the project, like its name, version, dependencies, and scripts. It is used for dependency management, enabling you to specify which external packages your project relies on and their versions.

To install the dependencies defined within package.json, we run npm install.

Once install is successful, a node_modules directory gets created, which contains all the external JavaScript modules and packages that your project depends on.

Finally, we have the index.js file where the actual business logic exits and a test.js file is used to define the test cases.

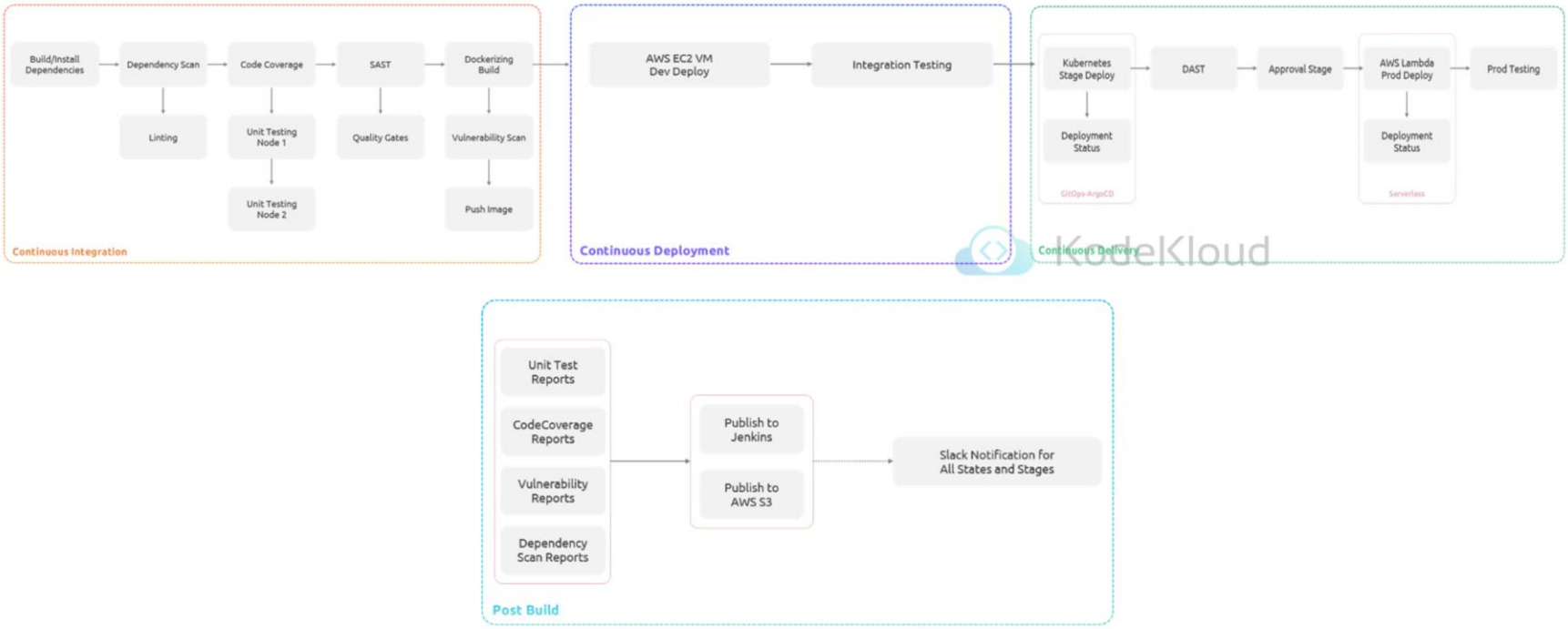
The npm test command is commonly used to run tests for a JavaScript project. This command runs all the tests defined in the test.js file.

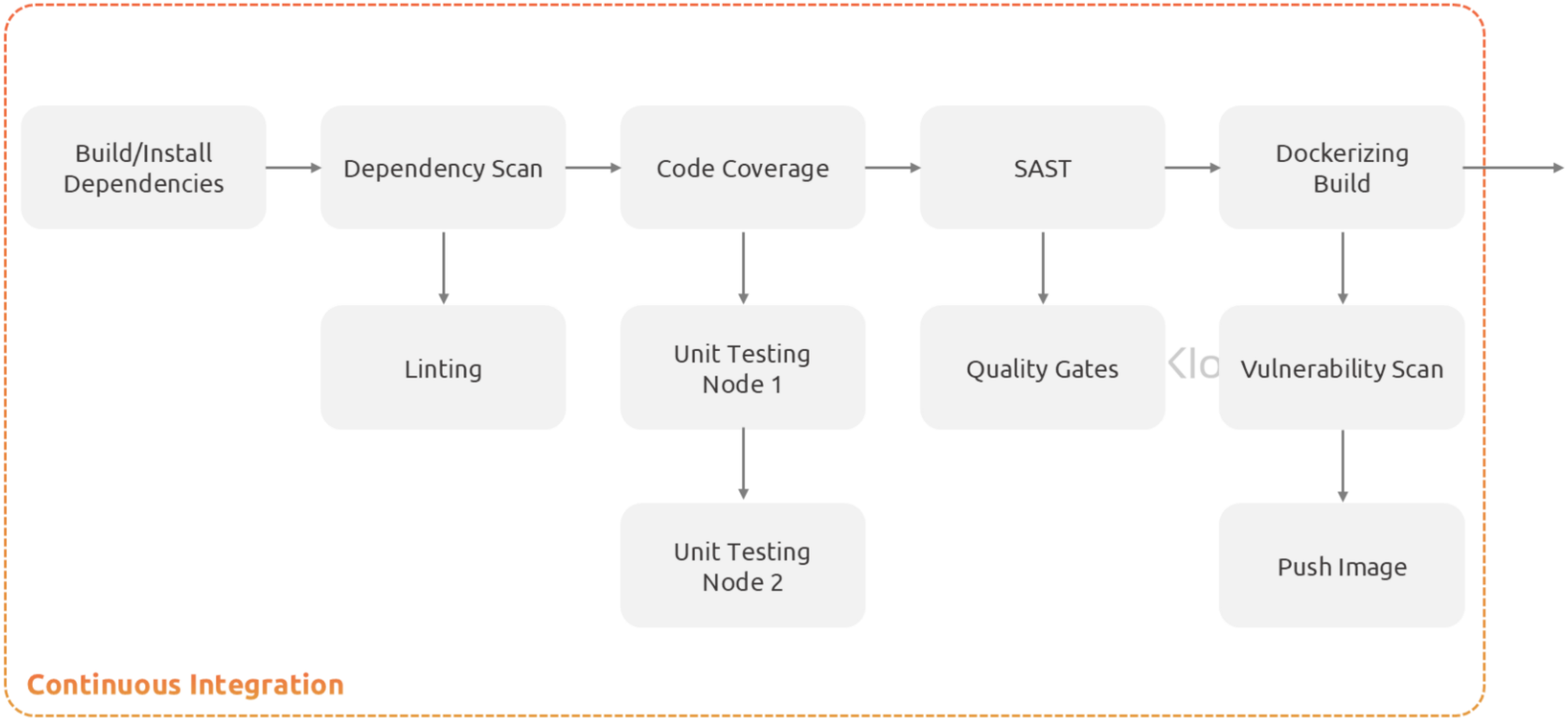
In a Node.js project, npm start is a common command used to start your application.

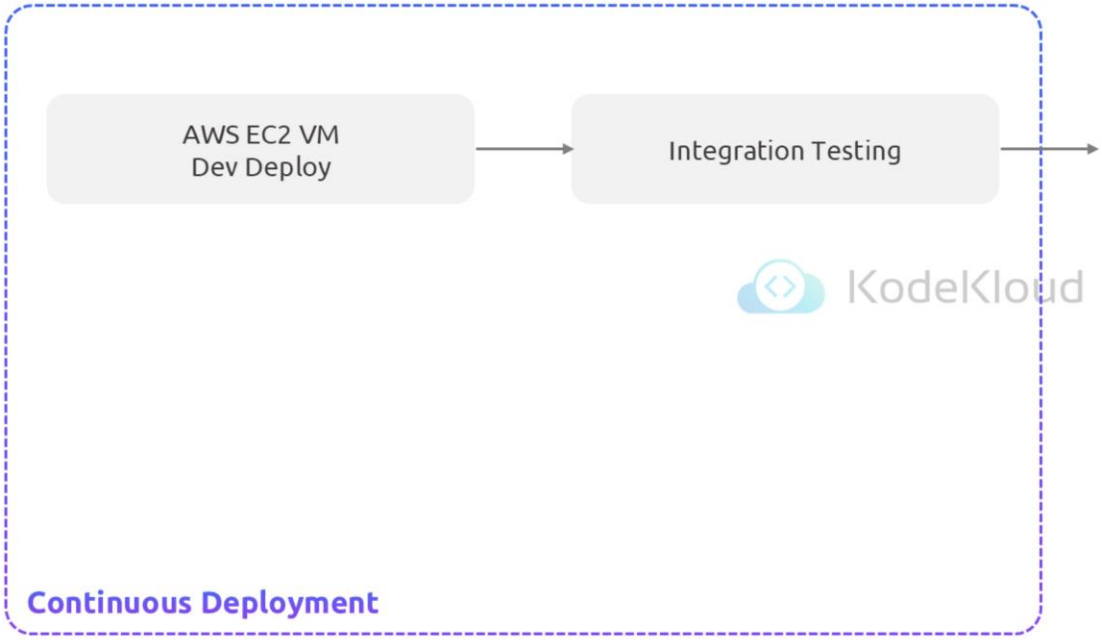
Once the application is started, we can access it on the browser.

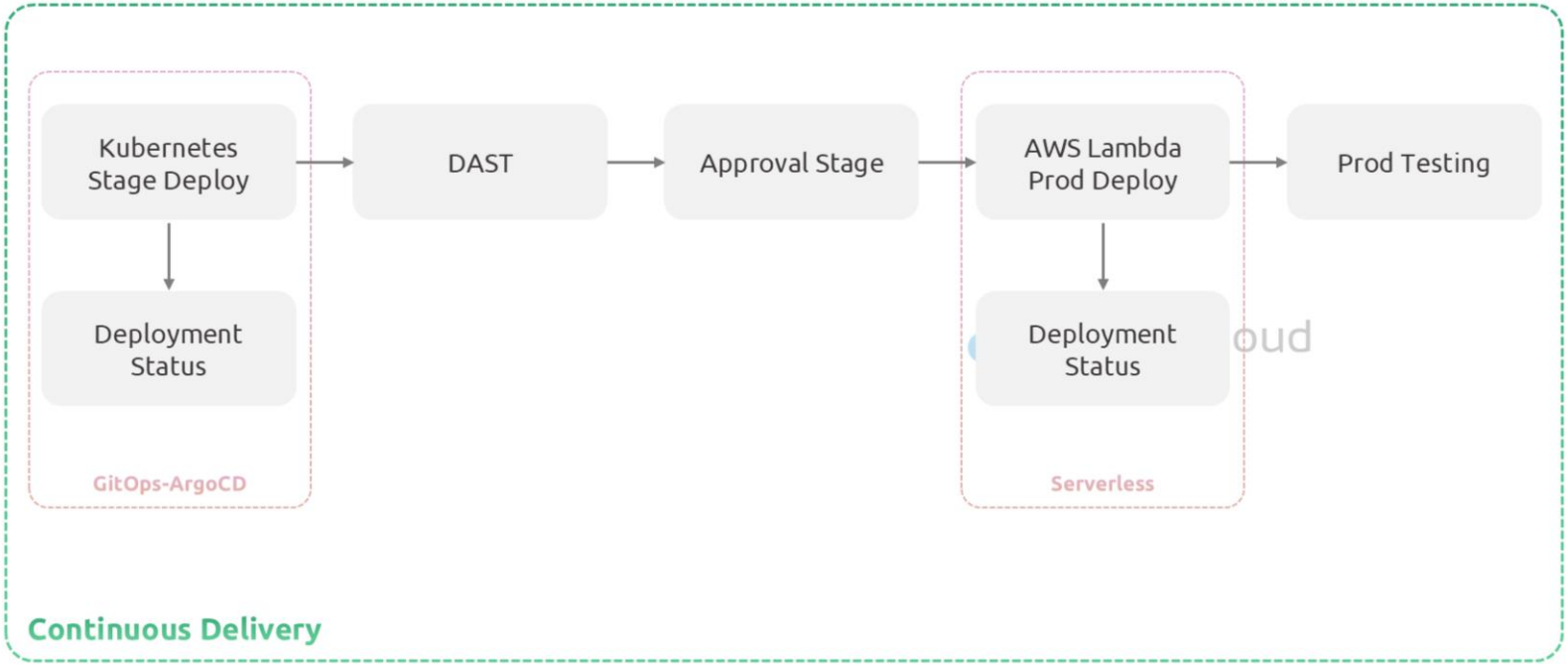
Understanding DevOps Pipeline

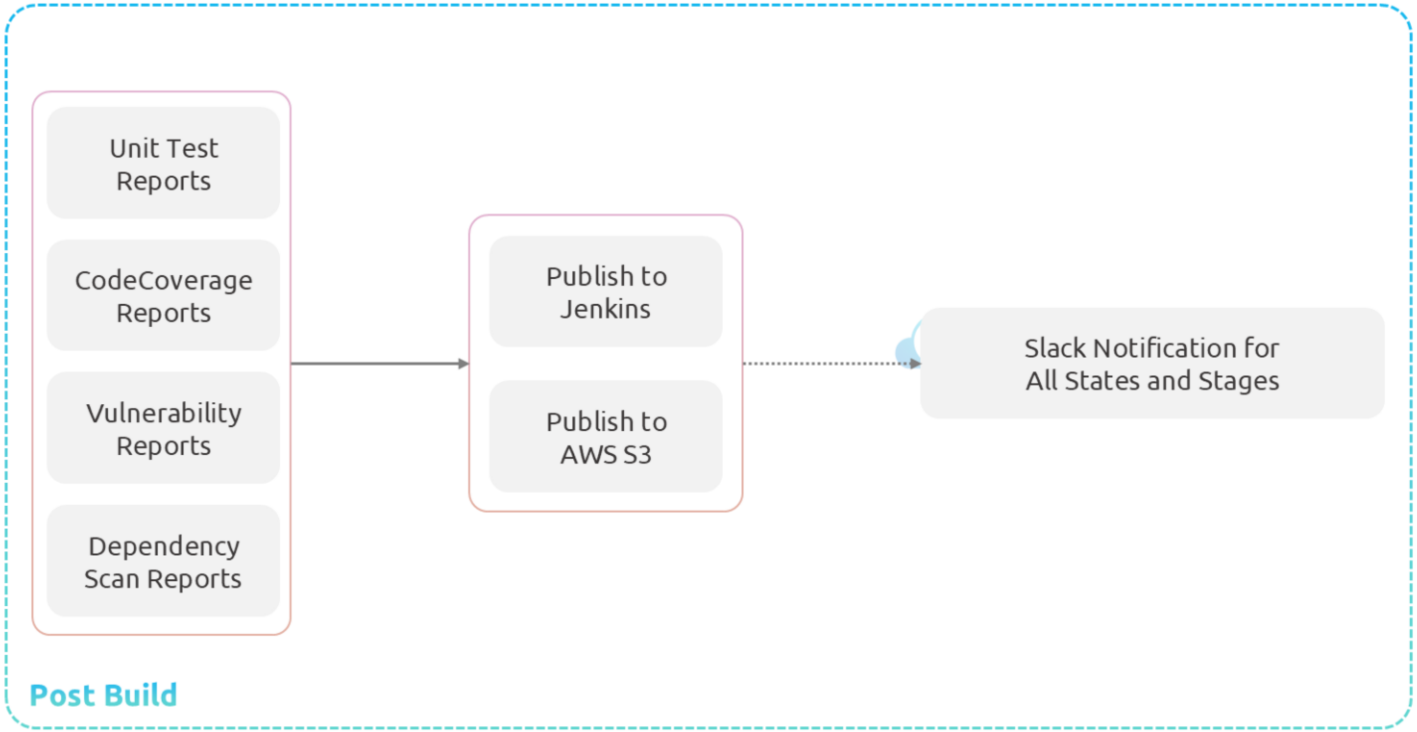
Understanding DevOps Pipeline

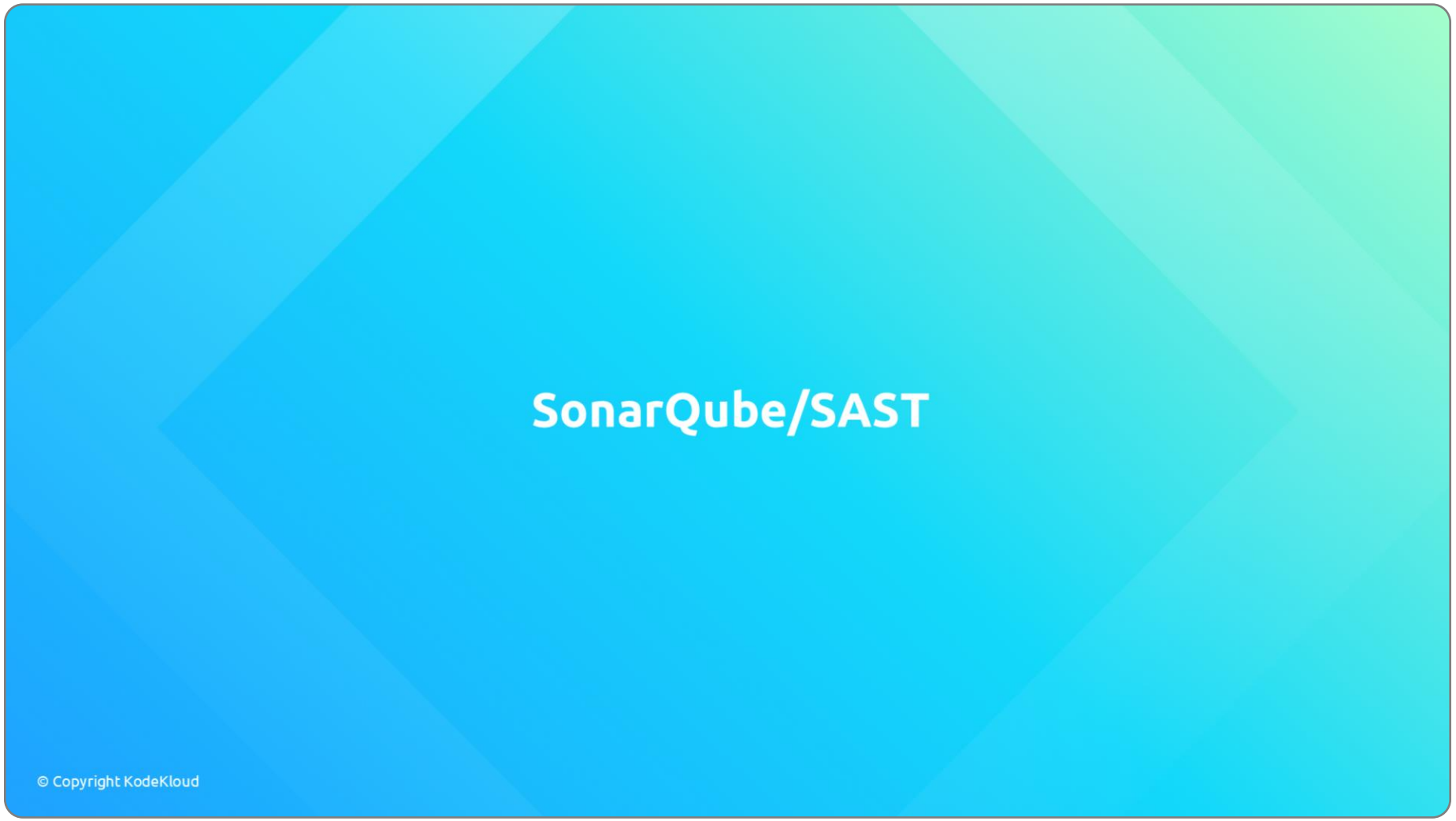












This video dives into Static Application Security Testing (SAST), also known as static analysis. It's a testing method that examines your application's source code to identify security vulnerabilities. Imagine it as a code scanner that flags potential security issues.

We'll be using SonarQube, an open-source platform from SonarSource, for our Static Analysis.

SonarQube helps you continuously monitor your code quality and perform automated code reviews.
Benefits of Static Analysis:

- Early Bug Detection: Identify bugs early in the development lifecycle, saving time and money on fixes.
- Refactoring and Simplification: SAST helps pinpoint areas in your code that might need restructuring or simplification.

Automated Code Rule Enforcement: Define project-specific coding rules and enforce them automatically, promoting consistent coding practices.

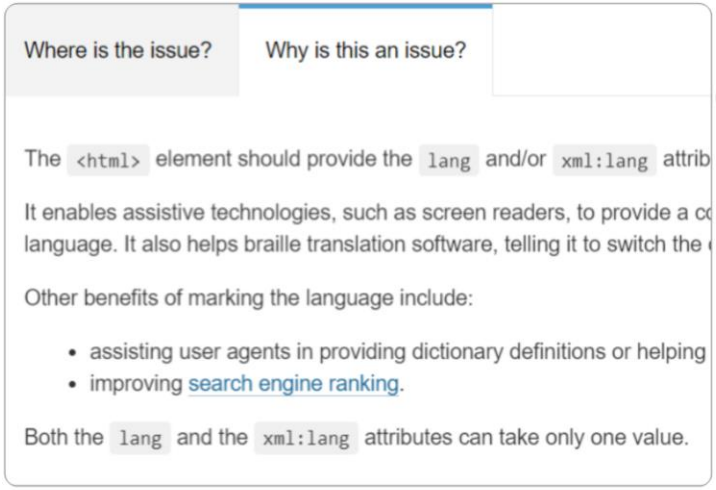
SonarQube

Pinpointing Security Issues

Simply seeing the code isn't enough. To address the issues flagged by SonarQube, we need to leverage the data it provides: Pinpointing Security Issues: SonarQube scans your entire codebase, highlighting specific lines where it detects security vulnerabilities.

SonarQube

Mitigating Risks



Mitigating Risks: It also offers insights on how to address these vulnerabilities, empowering you to fix them effectively.

Enforcing Quality Standards with Quality Gates

Code Smells

Indicators of potentially problematic coding practices.

Security Hotspots

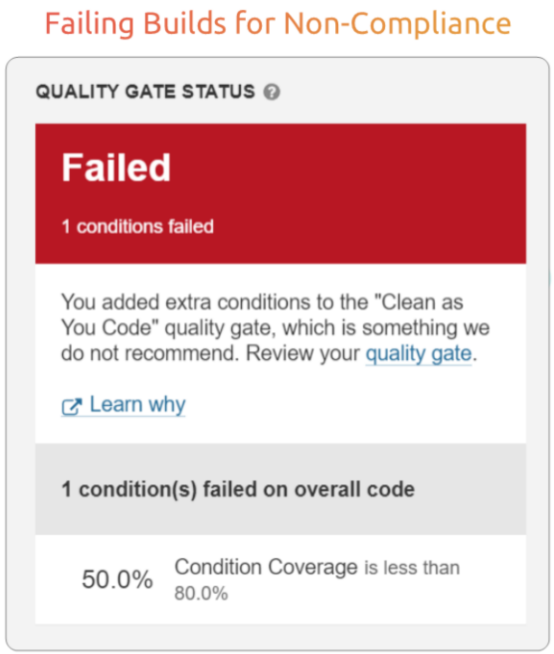
Areas in the code susceptible to security vulnerabilities.

Code Coverage

The percentage of your codebase tested by automated tests.

Quality gates act as your project's security checkpoint, ensuring standards are met. You can define thresholds for various metrics within SonarQube, including:
Code Smells: Indicators of potentially problematic coding practices.
Security Hotspots: Areas in the code susceptible to security vulnerabilities.
Code Coverage: The percentage of your codebase tested by automated tests.

Enforcing Quality Standards with Quality Gates

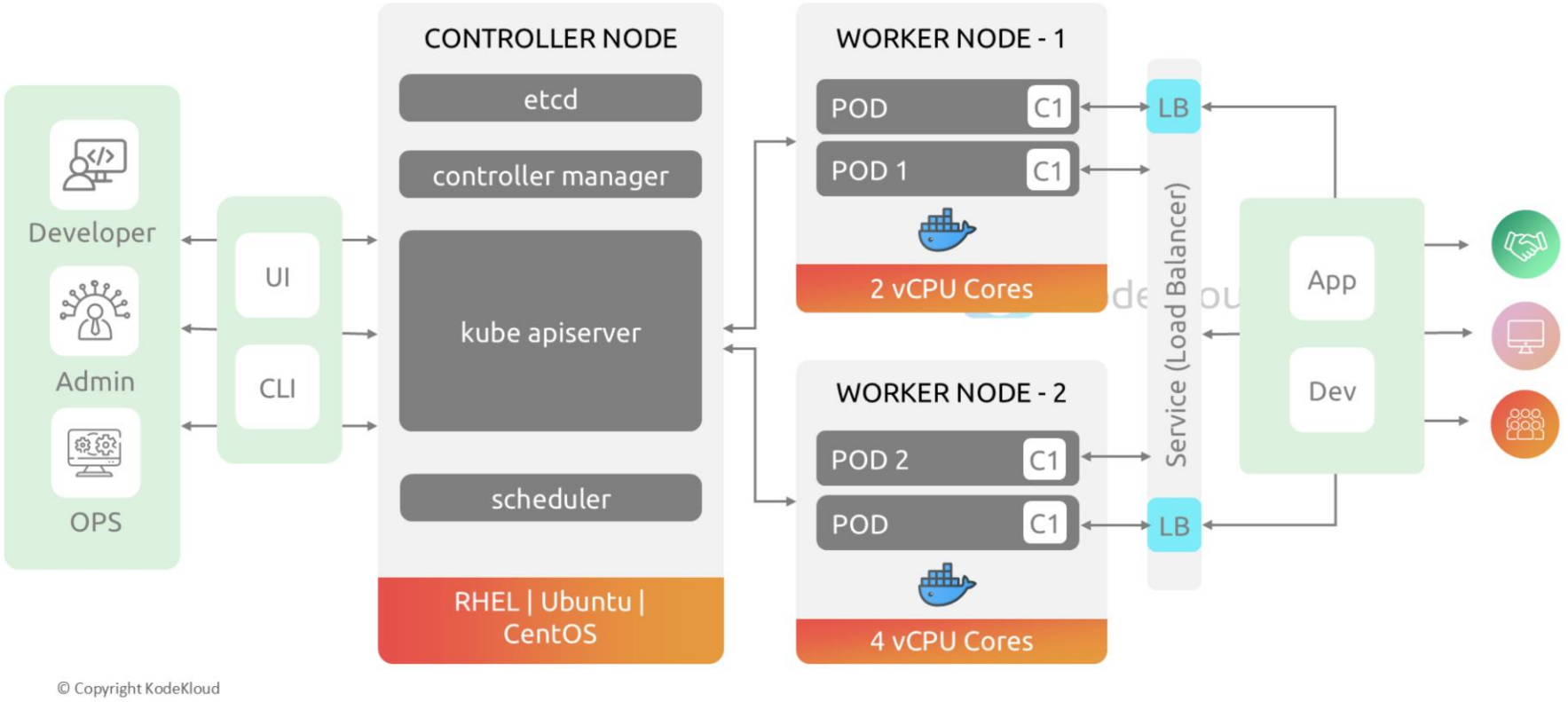


KodeKloud

Failing Builds for Non-Compliance:
If any of these quality gate conditions fail, your entire build fails. This ensures that only code meeting your quality standards progresses through the development pipeline.

Kubernetes – Brief Overview

Kubernetes Basics



Kubernetes is an open-source container orchestration platform originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). It is designed to automate the deployment, scaling, and management of containerized applications.

The first major components of Kubernetes are Nodes.

Nodes are the individual machines (physical or virtual) that make up a Kubernetes cluster. Nodes can be categorized as

either worker nodes or controller nodes. Worker nodes run the containers, while master nodes manage the overall cluster.

The controller node include the API server, controller manager, scheduler, and etcd. The API server is the entry point for managing the cluster and serves the Kubernetes API.

A pod is the smallest deployable unit in Kubernetes. It represents a single instance of a running process in the cluster. Pods can contain one or more containers that share the same network namespace, storage, and IP address. They are often used to group containers that need to work together.
In our example, we have only 1 container in a pod.

If a pod experiences an issue, it is removed and does not automatically restart. To prevent the need for manual intervention, we can utilize replication controllers or deployments, which handle the task of pod recovery.

Deployments are a higher-level abstraction for managing replica sets and pods. They provide declarative updates to applications, allowing you to describe an application's desired state, and Kubernetes will handle the details of updating the actual state to match the desired state.

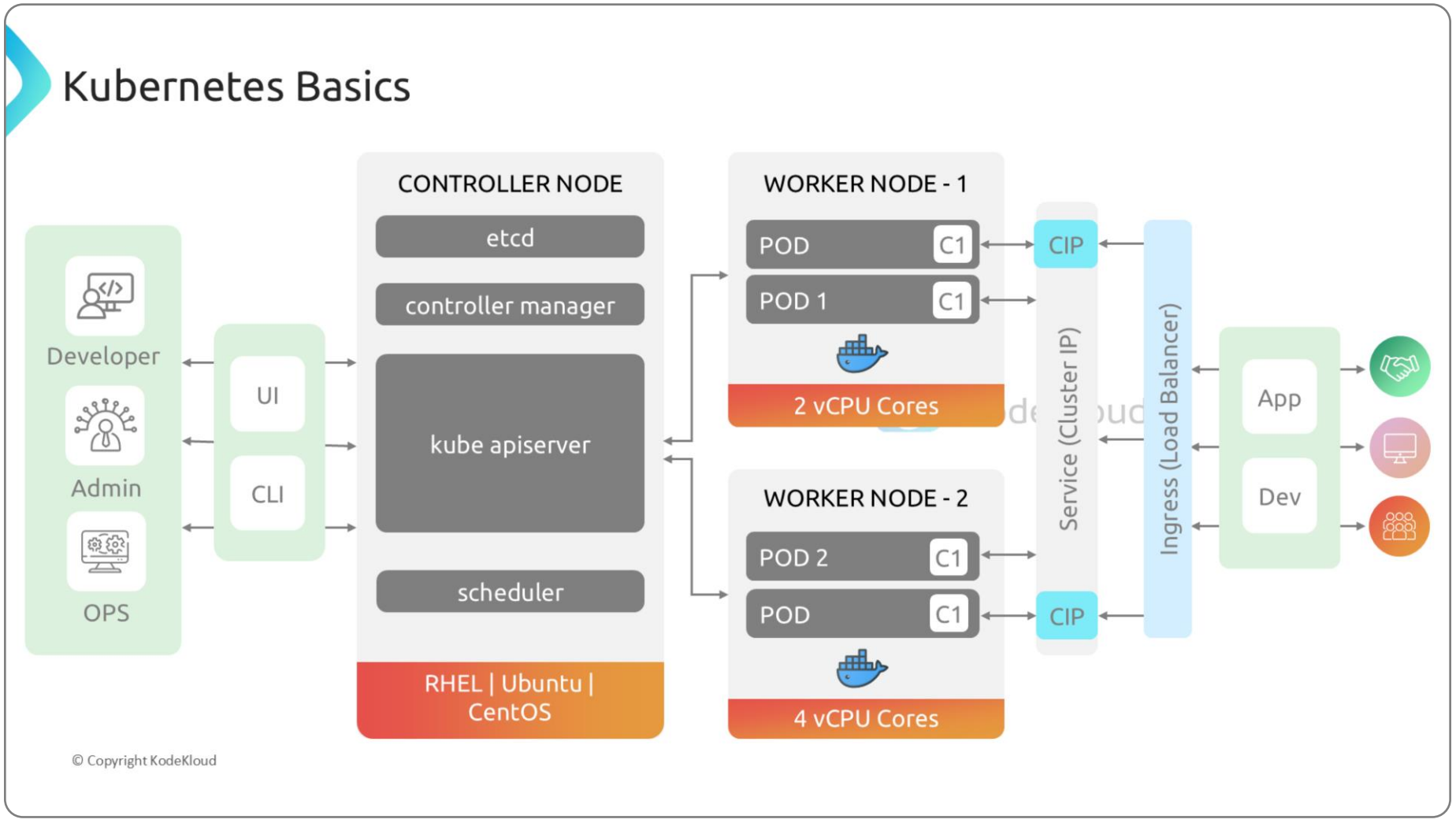
Once the pods are up and running, we can access them using Services.

Services in Kubernetes provide a stable and consistent network endpoint to access one or more pods. They can load balance traffic among multiple pods, allowing applications to be easily scaled without affecting external clients.

Kubernetes offers several types of services to enable network communication between pods. One of these types is the "LoadBalancer" service, which is used to expose your service to the external world, typically in a cloud environment. It creates an external load balancer (e.g., an AWS ELB or GCP Load Balancer) that routes traffic to the service.

It's important to note that the availability and features of LoadBalancer services can vary depending on the cloud provider or on-premises infrastructure you are using. Additionally, using LoadBalancer services may incur additional costs particularly when considering that a distinct load balancer may be required for each individual pod or application you wish to expose.

To avoid using multiple load balancers, one can utilize Ingress as an alternative solution.



Kubernetes Ingress is designed to manage and route HTTP and HTTPS traffic into the cluster. It provides more advanced and flexible traffic routing capabilities than a LoadBalancer service.

It's suitable for exposing multiple services under a single domain or for setting up more complex routing rules. You can route traffic based on paths, domain names, and other HTTP request attributes.

When services are exposed through Ingress, they are often configured with a Service type of "ClusterIP" for several reasons.

One major reason is ClusterIP services provide internal access to pods within the cluster but are not directly exposed externally. This allows you to have fine-grained control over which services are accessible from within the cluster, ensuring that not every service is exposed by default.

However, it's important to note that the choice of service types (e.g., ClusterIP, NodePort, LoadBalancer) and Ingress configuration depends on your specific use case and requirements.

ArgoCD

01 The fundamental concepts of GitOps.

02 Detailed information about ArgoCD, including what it is, why it is important, and how it works.

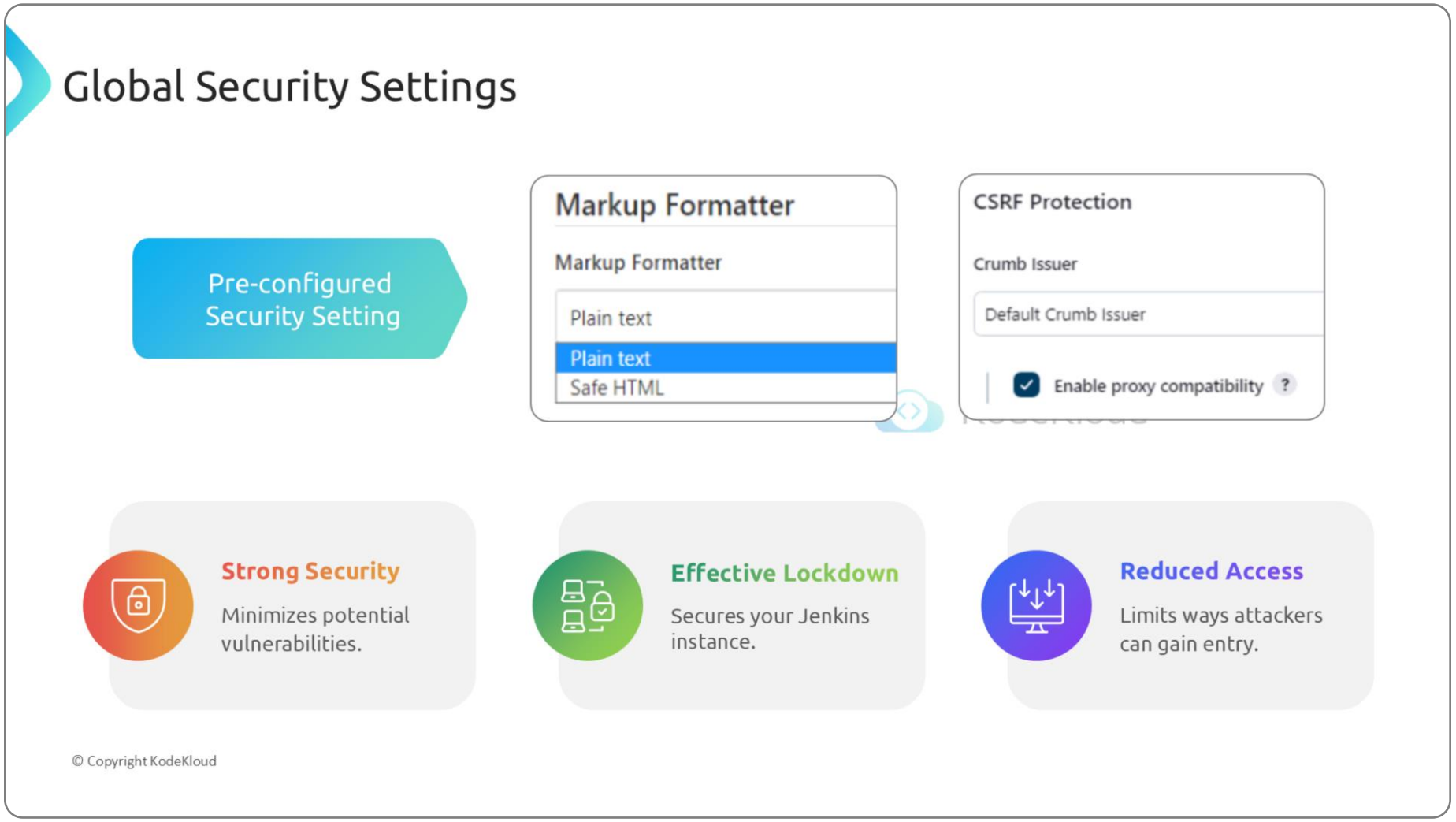
03 Key concepts and terminology related to GitOps and ArgoCD.

04 The architecture of ArgoCD.

AWS Lambda Basics - by Sanjeev

https://www.youtube.com/watch?v=RtiWU1DrMaM

Jenkins Global Security Settings



Jenkins comes pre-configured with strong security measures to minimize potential vulnerabilities. These settings effectively "lock down" your Jenkins instance, reducing the number of ways attackers could gain access.

However, sometimes you might need to adjust these settings to run specific jobs. It's important to remember that loosening security should be done cautiously and only when absolutely necessary.

Think of it like a house. A strong security system (default settings) keeps intruders out. But occasionally, you might need to

unlock a door (adjust settings) to let someone in (run a job). Just remember to re-lock the door (tighten security) as soon as you're done!

We will be talking about Markup formatter and CSRF Protection w.r.t Jenkins

Global Security Settings

However, sometimes you might need to adjust these settings to run specific jobs. It's important to remember that loosening security should be done cautiously and only when absolutely necessary.

Think of it like a house. A strong security system (default settings) keeps intruders out. But occasionally, you might need to unlock a door (adjust settings) to let someone in (run a job). Just remember to re-lock the door (tighten security) as soon as you're done!

We will be talking about Markup formatter and CSRF Protection w.r.t Jenkins

Global Security Settings

However, sometimes you might need to adjust these settings to run specific jobs. It's important to remember that loosening security should be done cautiously and only when absolutely necessary.

Think of it like a house. A strong security system (default settings) keeps intruders out. But occasionally, you might need to unlock a door (adjust settings) to let someone in (run a job). Just remember to re-lock the door (tighten security) as soon as you're done!

We will be talking about Markup formatter and CSRF Protection w.r.t Jenkins

Global Security Settings

Secured House

KodeKloud

However, sometimes you might need to adjust these settings to run specific jobs. It's important to remember that loosening security should be done cautiously and only when absolutely necessary.

Think of it like a house. A strong security system (default settings) keeps intruders out. But occasionally, you might need to unlock a door (adjust settings) to let someone in (run a job). Just remember to re-lock the door (tighten security) as soon as you're done!

We will be talking about Markup formatter and CSRF Protection w.r.t Jenkins

Markup Formatting

Security Concerns

Risk of malicious code injection (XSS attacks)

Format descriptions in Jenkins to enhance readability

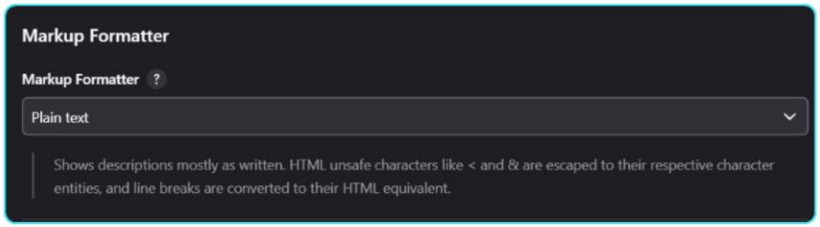
Security Measures

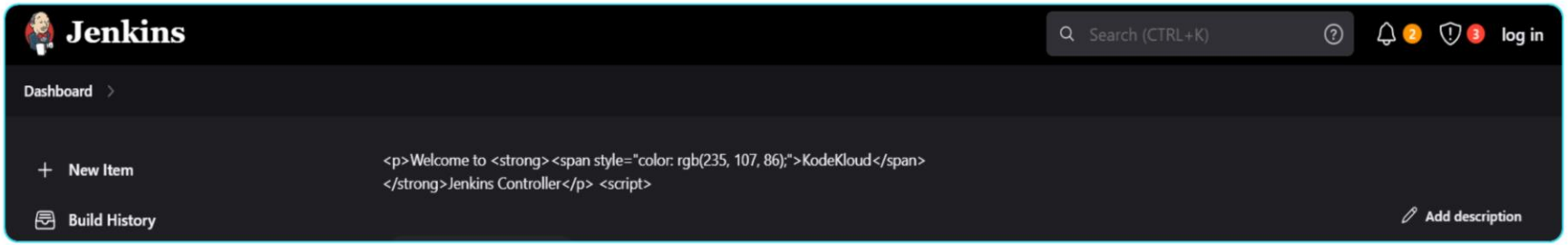
- Plain Text (Default)
- Safe HTML
- Custom Formatters (Plugins)

1. Markup Formatting:
What it is: Markup formatting allows users to format descriptions within Jenkins for better readability. Examples include descriptions for jobs, views, and builds.
Security Concerns: Without proper controls, users could potentially inject malicious code through formatting options. This could lead to security vulnerabilities like XSS (Cross-Site Scripting) attacks.
Security Measures: Jenkins offers various markup formatters:
Plain Text (Default): This is the most secure option, as it treats all input as plain text and escapes any characters that

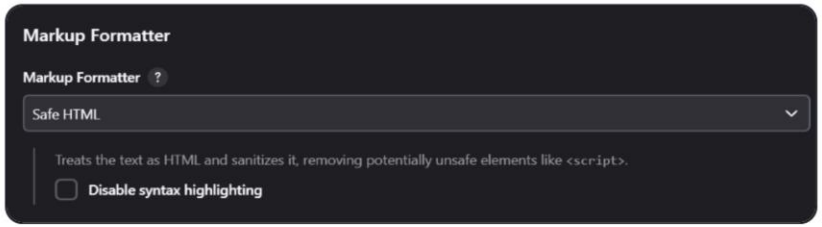
might be interpreted as code.
Safe HTML: This allows basic HTML formatting options but removes potentially risky elements to mitigate XSS vulnerabilities.
Custom Markup Formatters (Plugins): Some plugins provide more advanced formatting options, but require careful configuration to ensure security.

Markup Formatting - 1

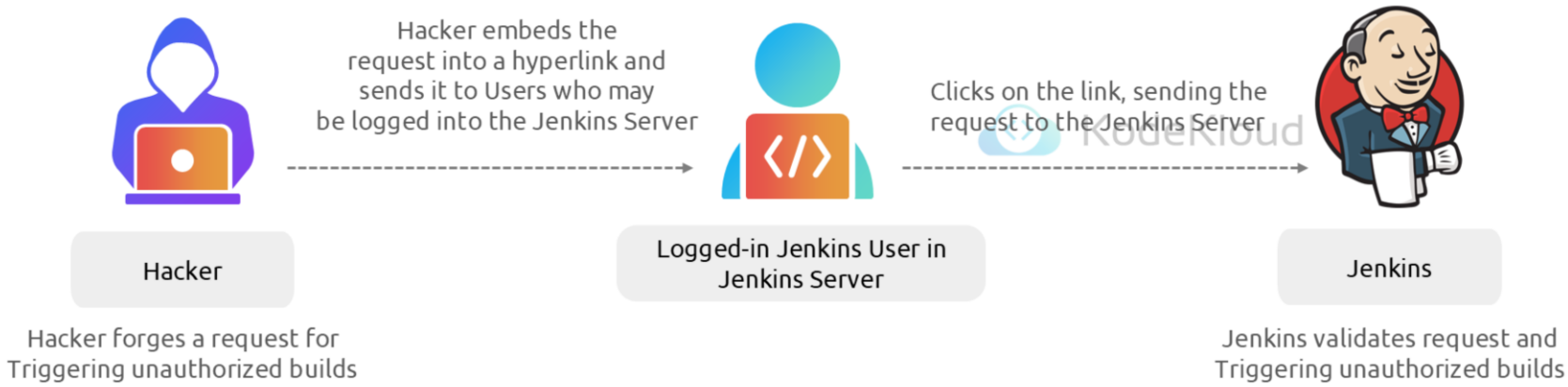




Markup Formatting - 2



CSRF (Cross-Site Request Forgery) Protection



Cross-Site Request Forgery (CSRF):
What it is: CSRF is a web security vulnerability that allows an attacker to trick a logged-in user into performing unintended actions within Jenkins.
How it Works: The attacker sends a malicious link or script that exploits the user's active session to trigger actions in Jenkins without their knowledge.
Examples:

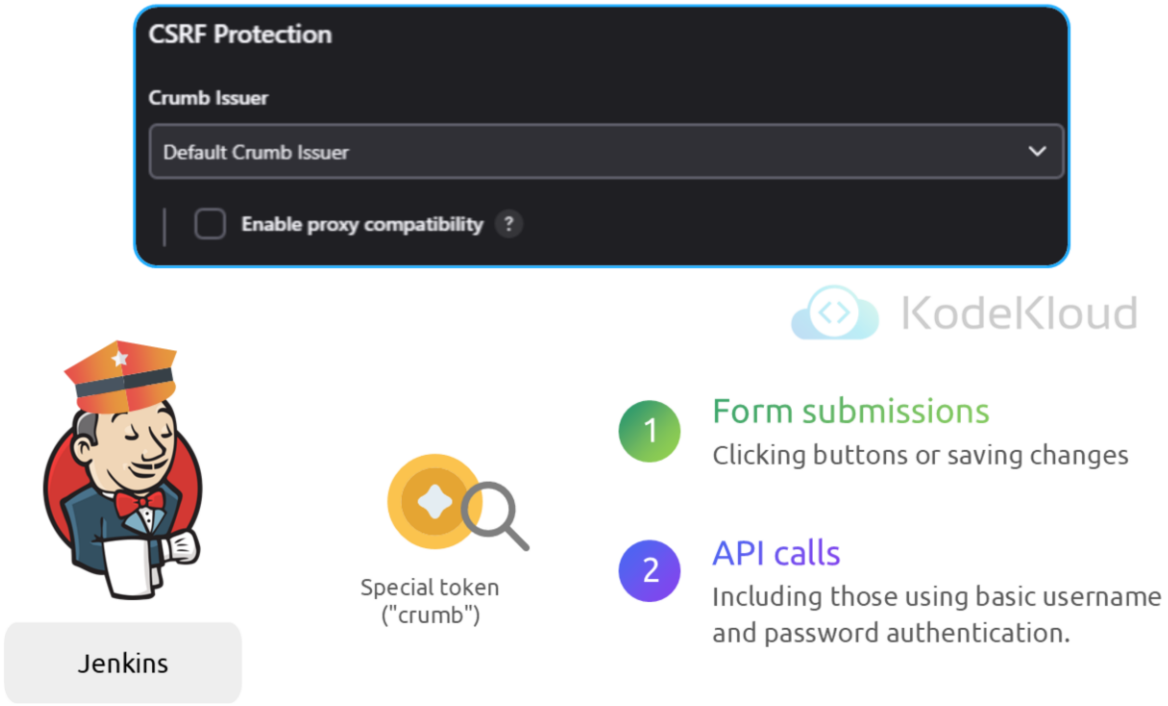
Triggering unauthorized builds.

Deleting builds and artifacts
Modifying configurations.

Prevention:

CSRF Protection: Enable this setting in Jenkins. It adds a hidden token to forms and requests, ensuring they originate from legitimate submissions within Jenkins.
User Awareness: Train users to be cautious about clicking suspicious links, especially when logged in to Jenkins.

CSRF Protection



By default, Jenkins has a built-in security feature called CSRF Protection (sometimes referred to as "crumb protection"). It's highly recommended to leave this feature enabled for maximum security.

How Does It Work?
When CSRF Protection is on, Jenkins acts like a security guard. It checks for a special token (called a "crumb") on every request that could potentially modify data within Jenkins. This includes:
Any form submissions (think clicking buttons or saving changes)

Calls made to the Jenkins API, even those using basic username and password authentication (like "Basic" auth)

Jenkins Scaling

Scaling Jenkins in your organization based on jobs running is quite a common activity.

Horizontal vs Vertical Scaling

As a software engineer, you've probably encountered situations where your application needs more resources to handle increased load. Here's a quick breakdown of two common scaling approaches:

Horizontal Scaling (Adding More Machines):
Imagine adding more worker servers to your software system. Each server works independently, sharing the workload.
Benefits:

Easier to scale up quickly by adding more machines.

Improves fault tolerance - if one machine fails, others can handle the load.

Drawbacks:

Requires managing multiple machines and configurations.

Might not be cost-effective for smaller loads.

Vertical Scaling (Upgrading Existing Machine):

Think of giving your existing server a performance boost (more CPU, memory). It can now handle more work on its own.

Benefits:

Simpler to manage, as you only need to upgrade one machine.

Can be cheaper for smaller workloads.

Drawbacks:

Limited scalability - there's a physical limit to how much you can upgrade a single machine.

A single point of failure - if the machine fails, everything stops.

Choosing the Right Approach:

The best approach depends on your specific needs. Horizontal scaling is generally preferred for large, distributed systems, while vertical scaling might be suitable for simpler applications with predictable workloads.

Horizontal vs Vertical Scaling

Horizontal Scaling

Adding more worker servers to share the workload.

Benefits

- Easy to scale up quickly.
- Improved fault tolerance.

Drawbacks

- Requires managing multiple machines.
- Not cost-effective for smaller loads.

As a software engineer, you've probably encountered situations where your application needs more resources to handle increased load. Here's a quick breakdown of two common scaling approaches:

Horizontal Scaling (Adding More Machines):
Imagine adding more worker servers to your software system. Each server works independently, sharing the workload.
Benefits:

Easier to scale up quickly by adding more machines.

Improves fault tolerance - if one machine fails, others can handle the load.

Drawbacks:

Requires managing multiple machines and configurations.

Might not be cost-effective for smaller loads.

Vertical Scaling (Upgrading Existing Machine):

Think of giving your existing server a performance boost (more CPU, memory). It can now handle more work on its own.

Benefits:

Simpler to manage, as you only need to upgrade one machine.

Can be cheaper for smaller workloads.

Drawbacks:

Limited scalability - there's a physical limit to how much you can upgrade a single machine.

A single point of failure - if the machine fails, everything stops.

Choosing the Right Approach:

The best approach depends on your specific needs. Horizontal scaling is generally preferred for large, distributed systems, while vertical scaling might be suitable for simpler applications with predictable workloads.

Horizontal vs Vertical Scaling

Vertical Scaling

Upgrading the existing server with more CPU and memory.

Benefits

- Simpler management.
- Cheaper for smaller workloads.

Drawbacks

- Limited scalability.
- Single point of failure.

As a software engineer, you've probably encountered situations where your application needs more resources to handle increased load. Here's a quick breakdown of two common scaling approaches:

Horizontal Scaling (Adding More Machines):
Imagine adding more worker servers to your software system. Each server works independently, sharing the workload.
Benefits:

Easier to scale up quickly by adding more machines.

Improves fault tolerance - if one machine fails, others can handle the load.

Drawbacks:

Requires managing multiple machines and configurations.

Might not be cost-effective for smaller loads.

Vertical Scaling (Upgrading Existing Machine):

Think of giving your existing server a performance boost (more CPU, memory). It can now handle more work on its own.

Benefits:

Simpler to manage, as you only need to upgrade one machine.

Can be cheaper for smaller workloads.

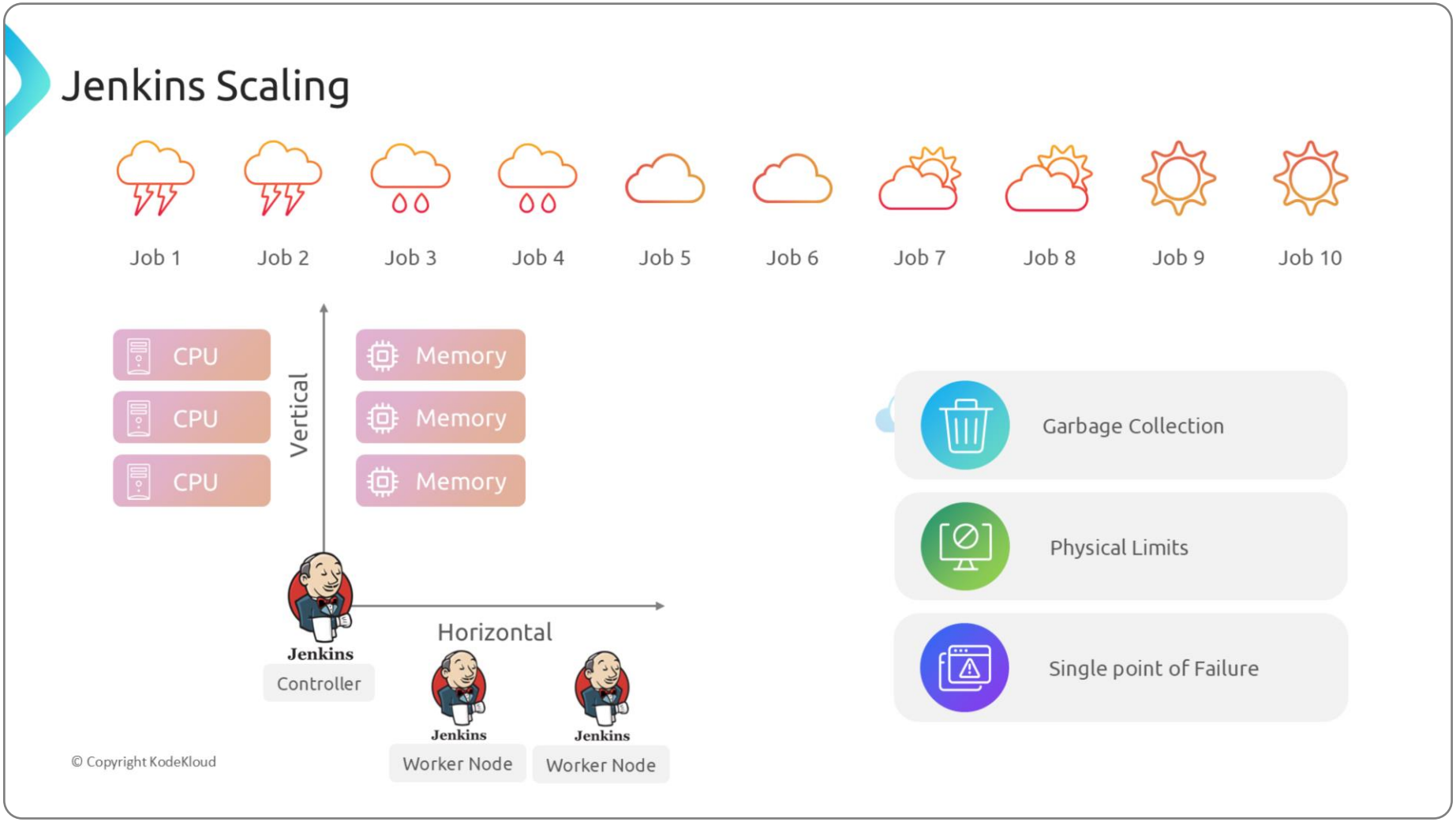
Drawbacks:

Limited scalability - there's a physical limit to how much you can upgrade a single machine.

A single point of failure - if the machine fails, everything stops.

Choosing the Right Approach:

The best approach depends on your specific needs. Horizontal scaling is generally preferred for large, distributed systems, while vertical scaling might be suitable for simpler applications with predictable workloads.



As your team embraces CI/CD across more projects/jobs, you might face a question: can Jenkins handle the increased workload? This is where Jenkins scaling becomes crucial.

The Monolith Challenge:
Let's imagine your organization uses a single, central Jenkins instance (often called a "monolith"). This single controller manages everything, with all possible plugins installed and integrations with numerous pipelines.

UI Load and Vertical Scaling Limits:
With a large team using the Jenkins UI, and executing numerous pipelines/jobs the controller's CPU, memory demands rise. While vertical scaling (adding more CPU and memory to the controller) seems like a solution, it has drawbacks:
Temporary Relief: Increased memory might not solve the problem forever. As you add more memory (heap), garbage collection cycles take longer, impacting performance.
Physical Limits: Upgrading a single machine has limitations. There's a physical cap on how much hardware you can add.
Single Point of Failure: A single controller creates a critical point. If it fails, everything stops.

The Power of Horizontal Scaling:
The best practice here is horizontal scaling. This involves adding worker nodes (agents) to distribute the workload. This reduces the footprint of the controller, making your Jenkins architecture more scalable and resilient.

Scalable Storage:
As your Jenkins environment grows, consider a scalable storage solution to manage build artifacts and logs efficiently.

==========

KodeKloud

Visit www.kodekloud.com to learn more.