KodeKloud

Visit www.kodekloud.com to discover more!

Problem Statement (Meeting With Dasher Team)

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 Gitlab CICD can be used to meet these DevOps requirements.

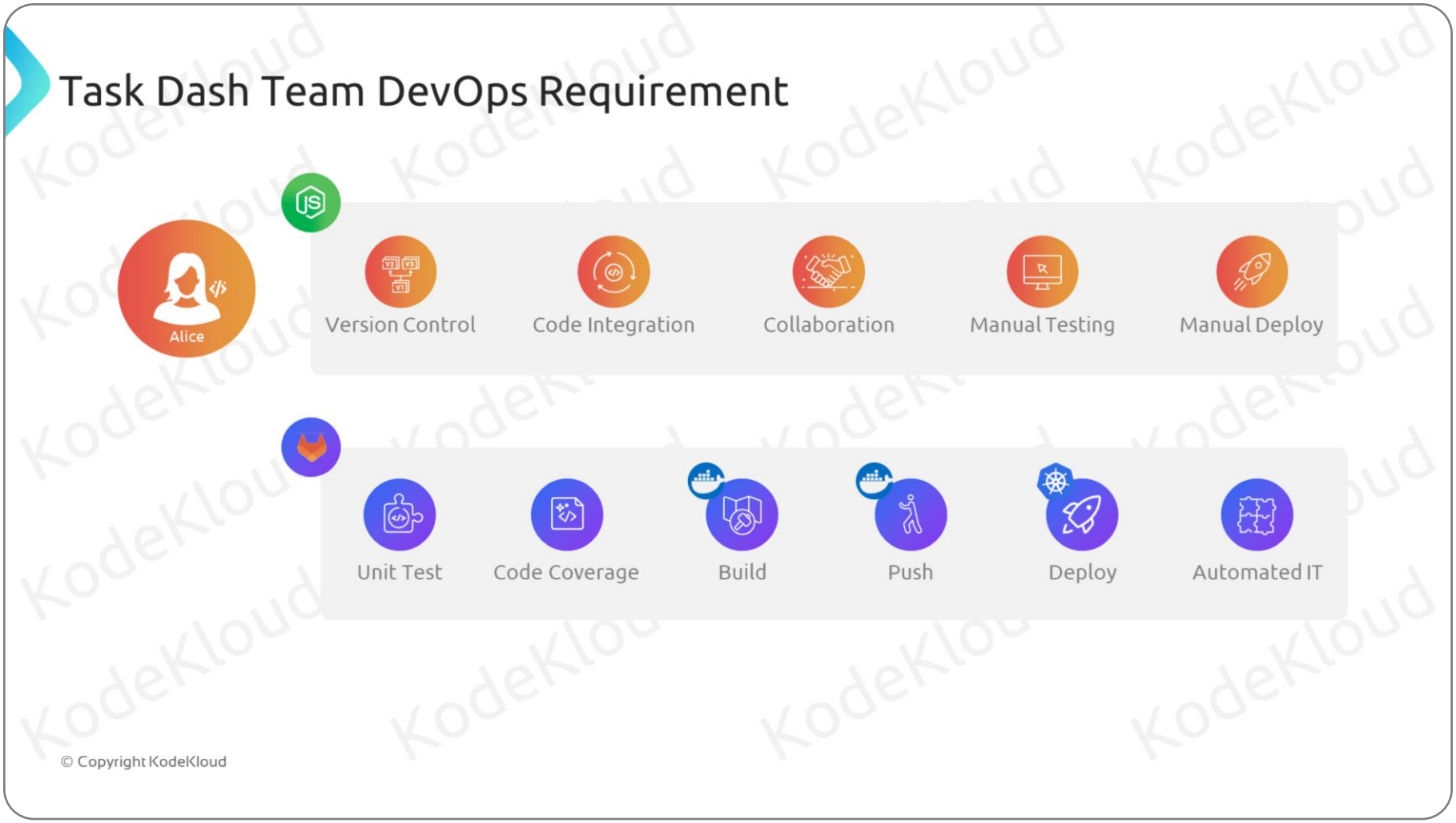
- Software provider
- R&D team exploration
- Initial focus
- Transition approach
- Future extensions
- Platform connection

DASHER TECHNOLOGIES®

Task Dash Team DevOps Requirement

Docker for Containerization

Kubernetes for Container Orchestration



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:
1.Adoption of Gitlab for version control and developer collaboration.
2.Implementation of unit testing and code coverage measures to expedite testing and minimize bugs.
3.Utilization of Docker Build and Push processes for containerization, and the application is deployed to Kubernetes.
4.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

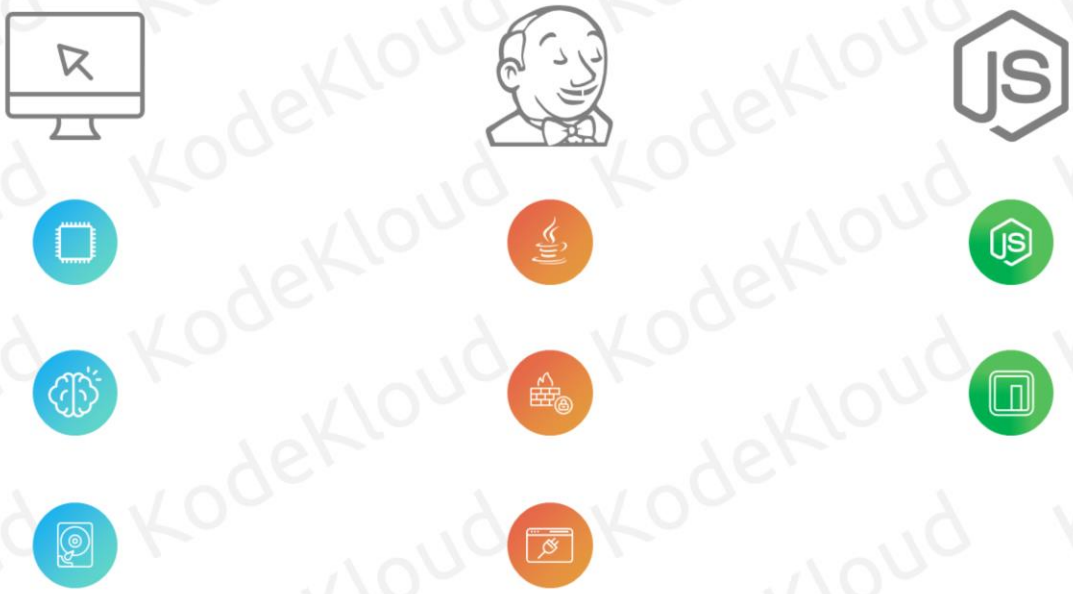
Travis CI

Circle CI

Atlassian Bamboo

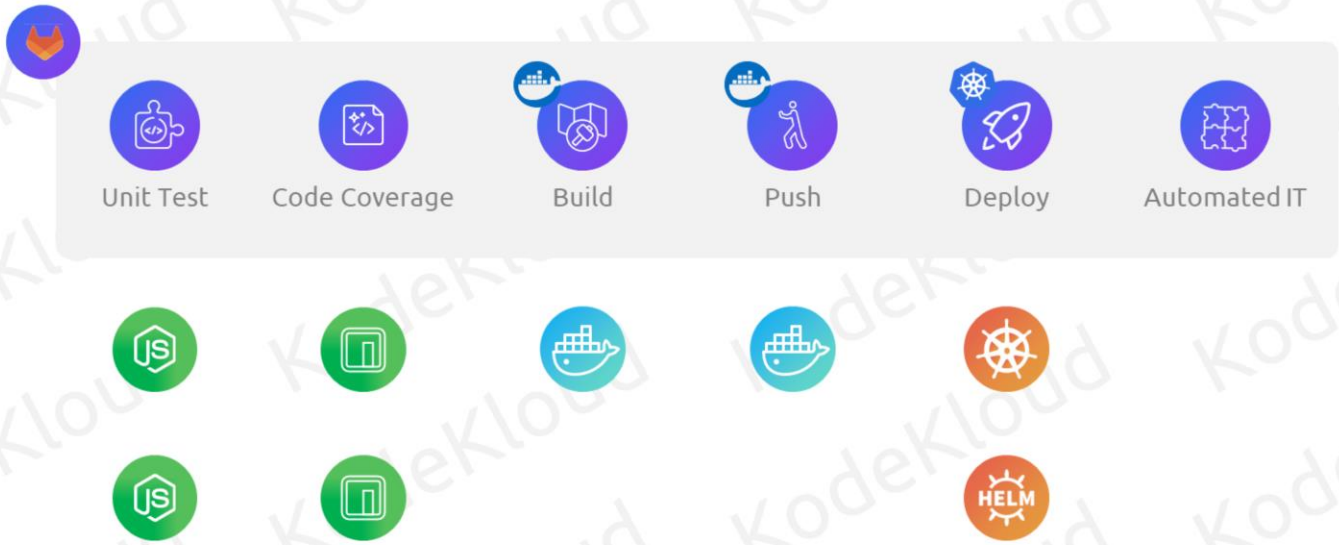
Spinnaker

Traditional CI/CD Tools – Challenges



To employ Jenkins, the team is required to undertake several tasks:
Configuration, operation, and upkeep of a virtual machine with specific CPU, memory, and HDD capacity.
Prior to Jenkins installation, prerequisites include Java JDK installation, setting up firewall rules, and installing Jenkins Plugins.
Given that this project is centered around Node.js, Node.js and Npm must be installed.

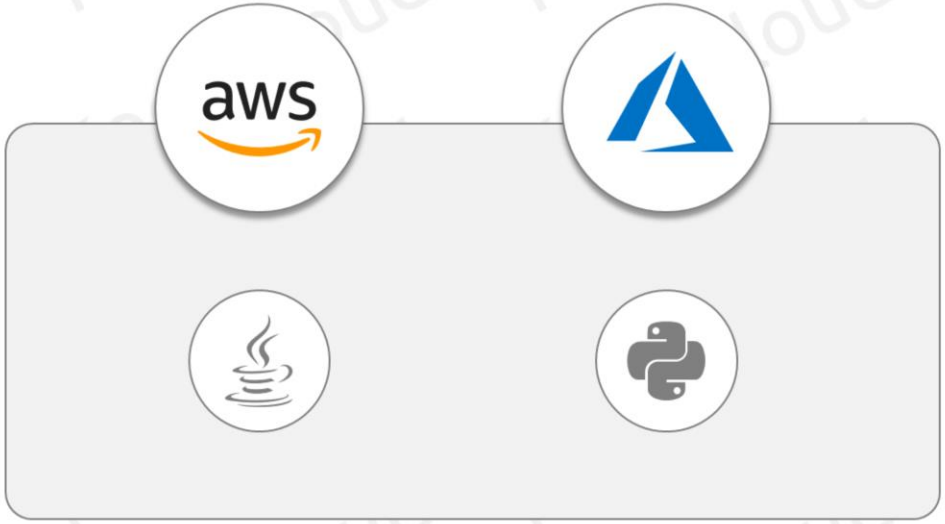
Traditional CI/CD Tools – Challenges



To facilitate unit testing with various Node.js versions, the DevOps engineer must install multiple Node.js/Npm versions, ensuring tests run without version conflicts.
Docker installation is essential for containerization.
For Kubernetes deployment, the installation of kubectl, helm, or other necessary binaries is mandatory.
Teams often employ external tools for integration testing and reporting, necessitating the installation of these tools or their Command-Line Interfaces (CLIs) on the machine where Jenkins will execute them.

It's important to note that these tasks apply to a single Node.js project with high-level steps. The pipeline's complexity may grow with additional steps, proportionally increasing the manual configuration workload.

Traditional CI/CD Tools – Challenges



Furthermore, the organization intends to implement similar methodologies for other Java and Python projects/applications running on cloud platforms like AWS and Azure.

Traditional CI/CD Tools – Challenges

Depending on the specific project, additional software requirements such as Java, Maven, Python, Azure/AWS CLI may be necessary. If the pipeline demands enhanced security in line with DevSecOps practices, more tools like Trivy and kubesec must be configured.

Traditional CI/CD Tools – Challenges

01 Simple setup and initiation, no need for extensive service installations and configurations.

02 Emphasis on pipeline development without managing infrastructure or scalability concerns.

Considering that this DevOps team is recently established, many team members lack familiarity with a broad spectrum of tools and services.

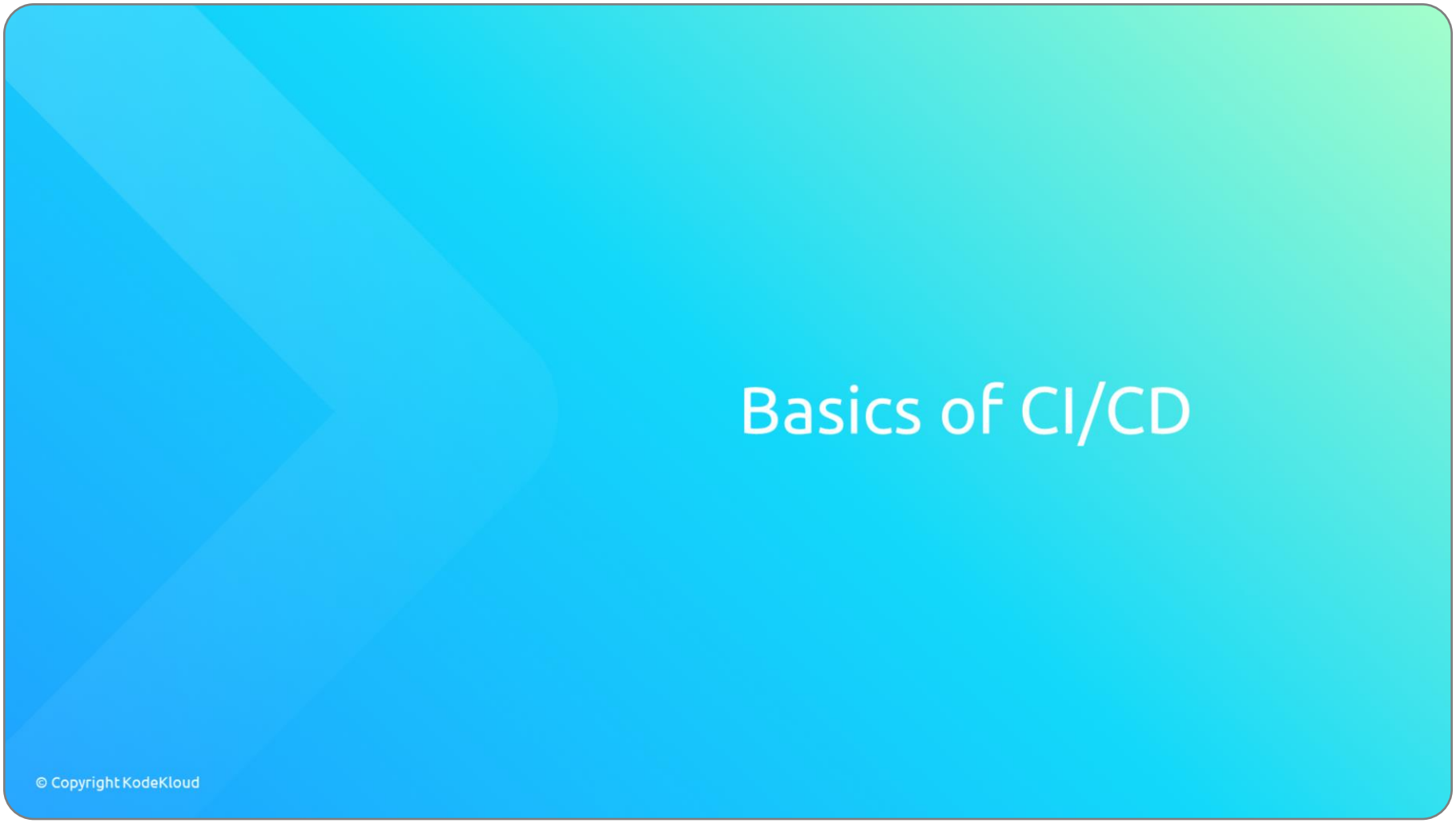
Alice has initiated a search for a tool that satisfies the following criteria:
Simplified setup and initiation without the need to install and configure numerous services.
A focus on building pipelines without the burden of managing infrastructure or concerns regarding scalability.

Traditional CI/CD Tools – Challenges

GitLab CI/CD

Following a thorough evaluation of multiple tools, Alice has chosen to check and implement Gitlab CICD.

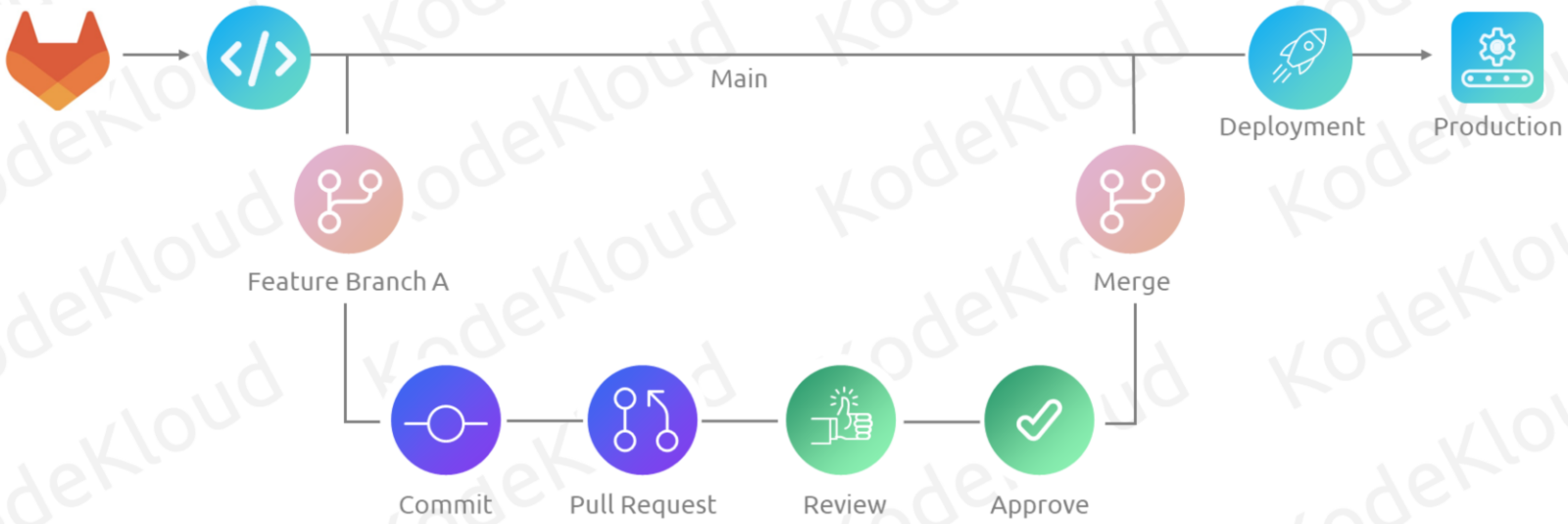
Over the course of this training program, we will check into the creation of Gitlab CICD Workflows for a straightforward yet a real-time NodeJS application.



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. Gitlab, a web-based platform centered around Git, serves as a centralized hub for hosting these repositories and extends Git's capabilities with additional features.

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.

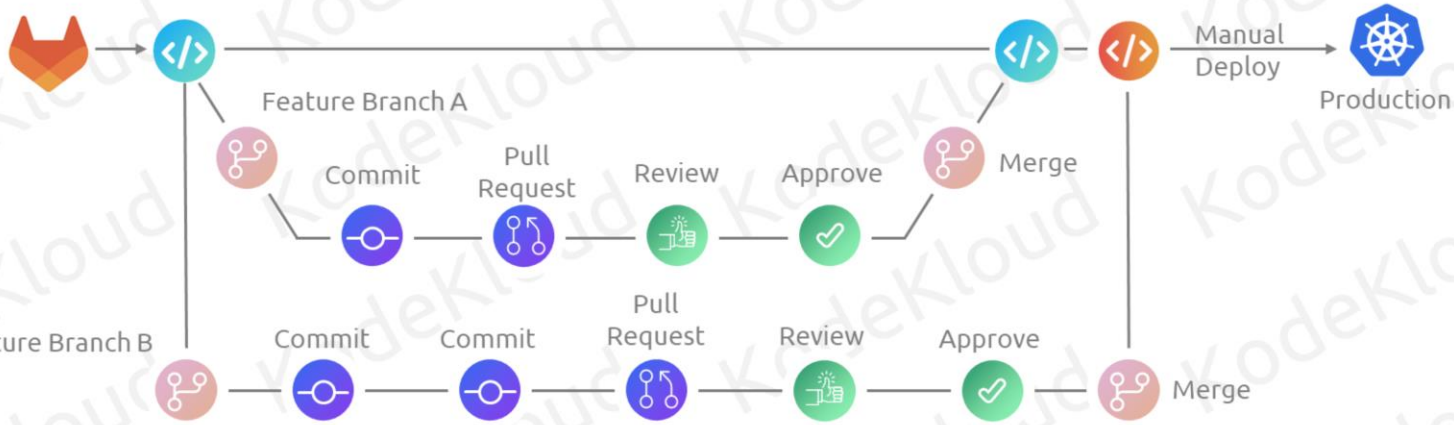
Once the necessary changes are made, the code is committed to the feature branch, and a pull request is initiated to merge this code back into the main branch.

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

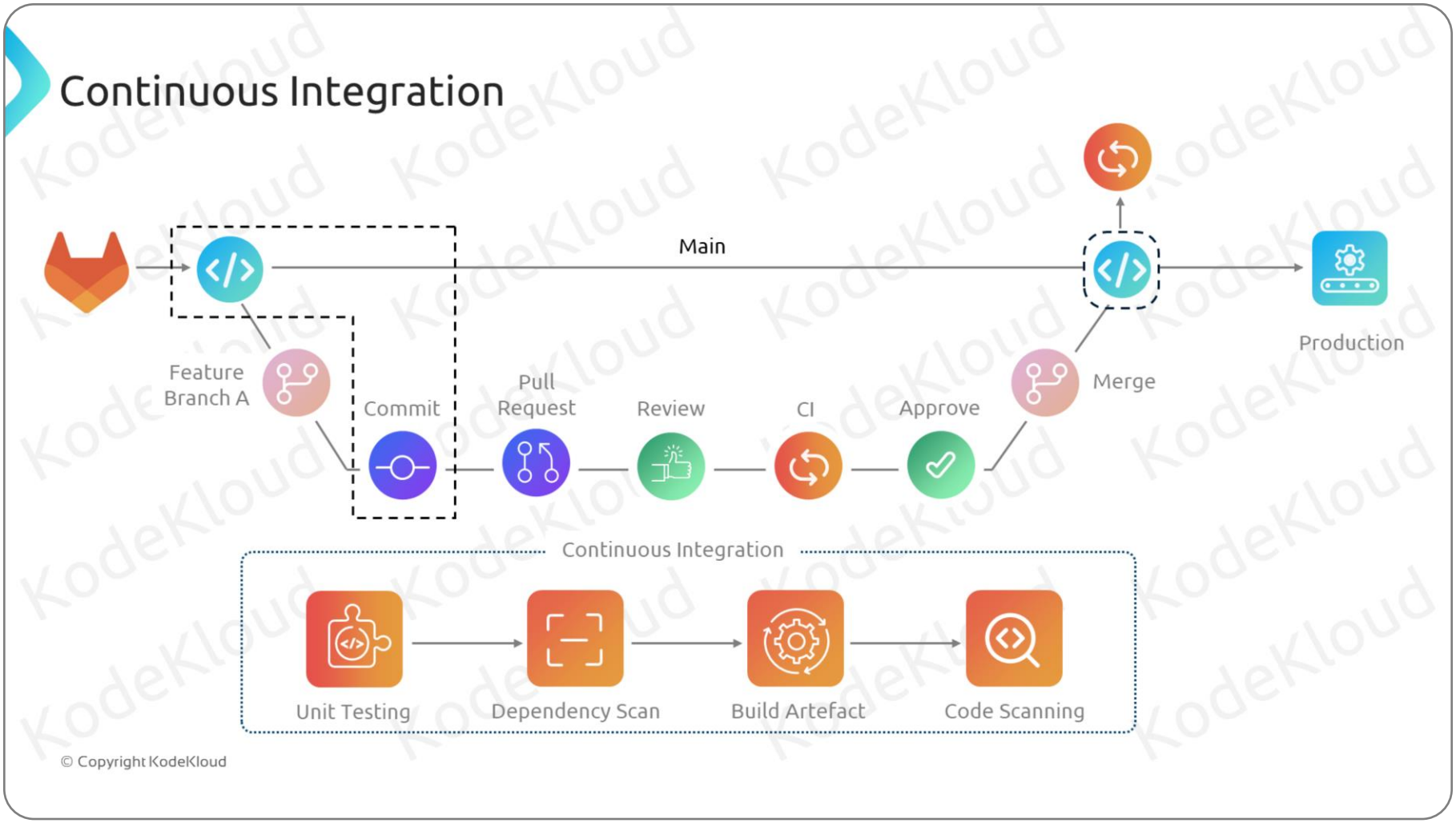
In real-time scenarios, you'll often find multiple developers working on different feature branches each focusing on a new enhancement.

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.



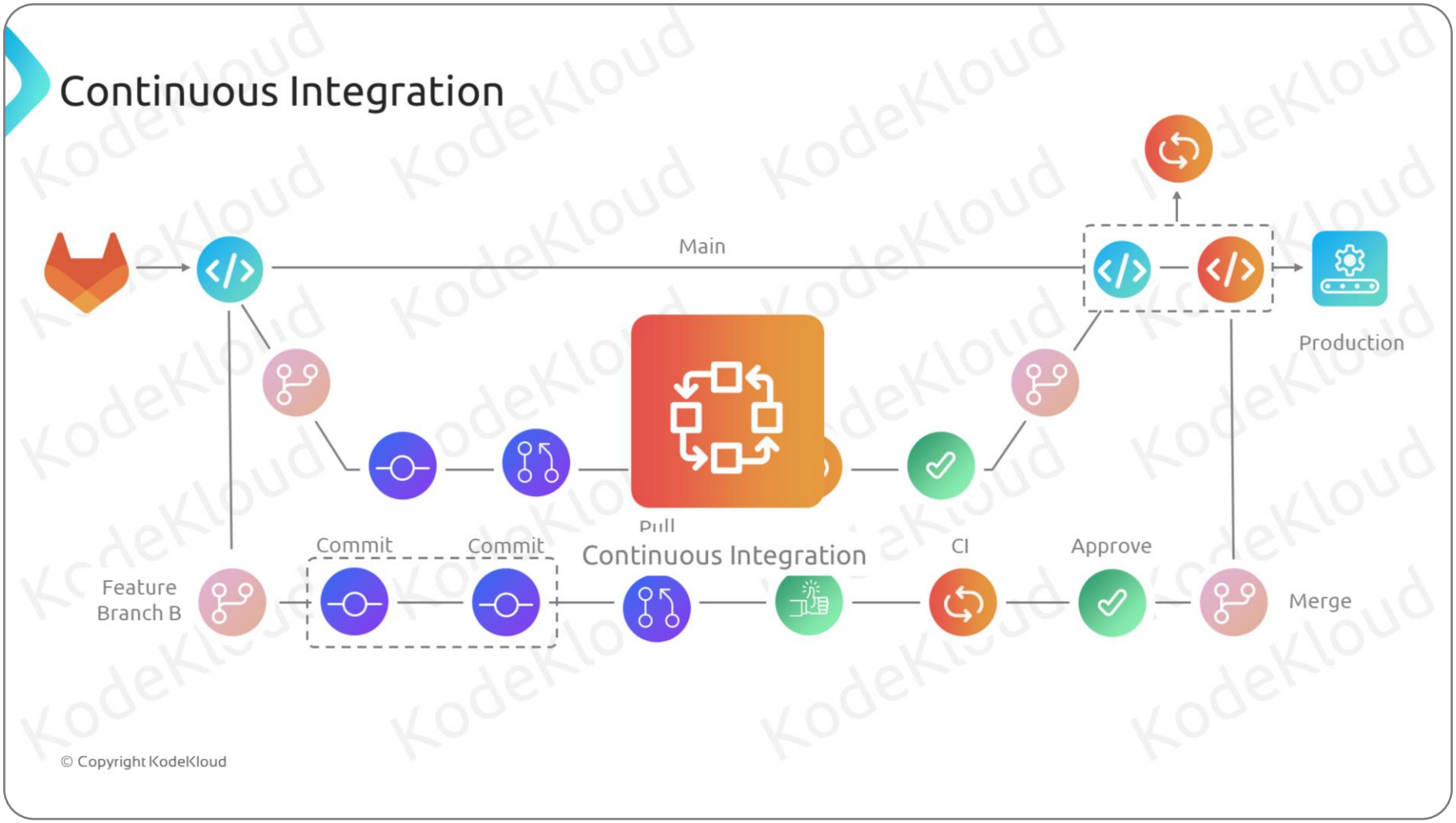
Let's imagine a scenario where Developer 1 creates a feature branch A, makes some modifications, and commits the code to this branch. A pull request is generated to merge these changes into the main branch. Before the merge, a team member reviews the code, and an automated CI pipeline is triggered.

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.

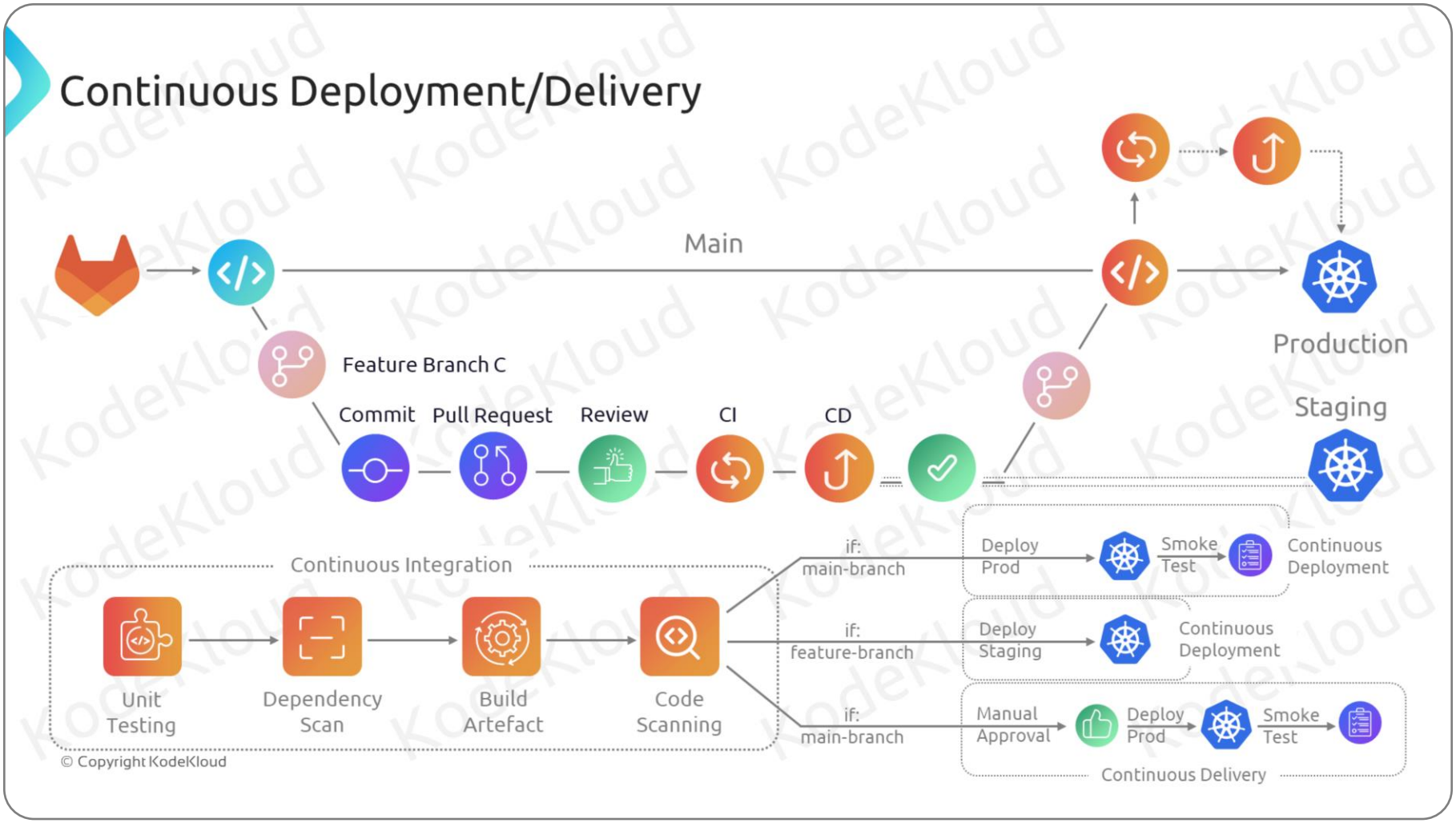


While these developments are taking place in feature-branch A, Developer 2 is progressing on feature branch B. They commit their changes and create a pull request. An automated CI pipeline is initiated, testing the code committed in this branch. Once the CI pipeline successfully completes, the pull request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

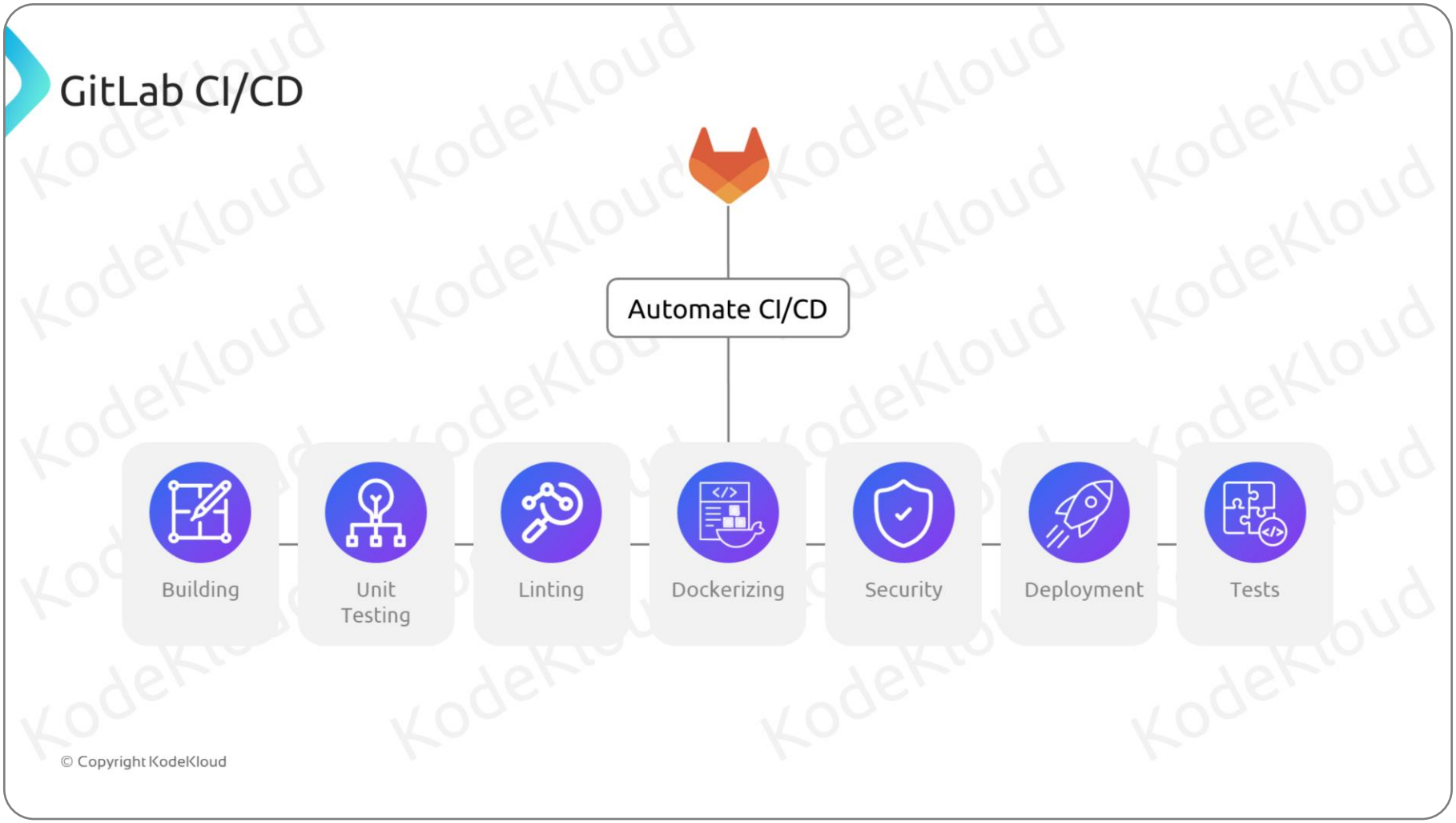
Upon successful completion of CD, the pull request is approved and merged back into the main branch.

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 GitLab CI/CD



Lets assume an organization has decided to use Gitlab as they code repository and and is in search of an CICD automation solution.

GitLab Server is an open-source DevOps platform that allows you to manage your entire software development lifecycle from a single location. It provides features for version control, issue tracking, CI/CD pipelines, package registries, and more

There are numerous tools available in the market to automate CICD pipelines. Nevertheless, given the organization's commitment to Gitlab as the code repository, let's explore how Gitlab CICD offers the automation capabilities.

So what is Gitlab CICD ?

GitLab CI/CD is a built-in tool that helps you automate your software development lifecycle, from code commit to deployment. It allows developers to automate tasks directly from their repositories.

You can quickly create workflows/pipelines to implement a CICD job that build, test on every, commit, pull request, and deploy merged pull requests to production right beside your code base.

So how are these CICD jobs executed?

GitLab Runners

SaaS Runners

Self-Managed Runners

Runners are the agents that run your jobs. GitLab Runner works with GitLab CI/CD to run jobs in a pipeline.

Out of the box, GitLab CI/CD offers two main options SaaS Runners and self-managed runners.

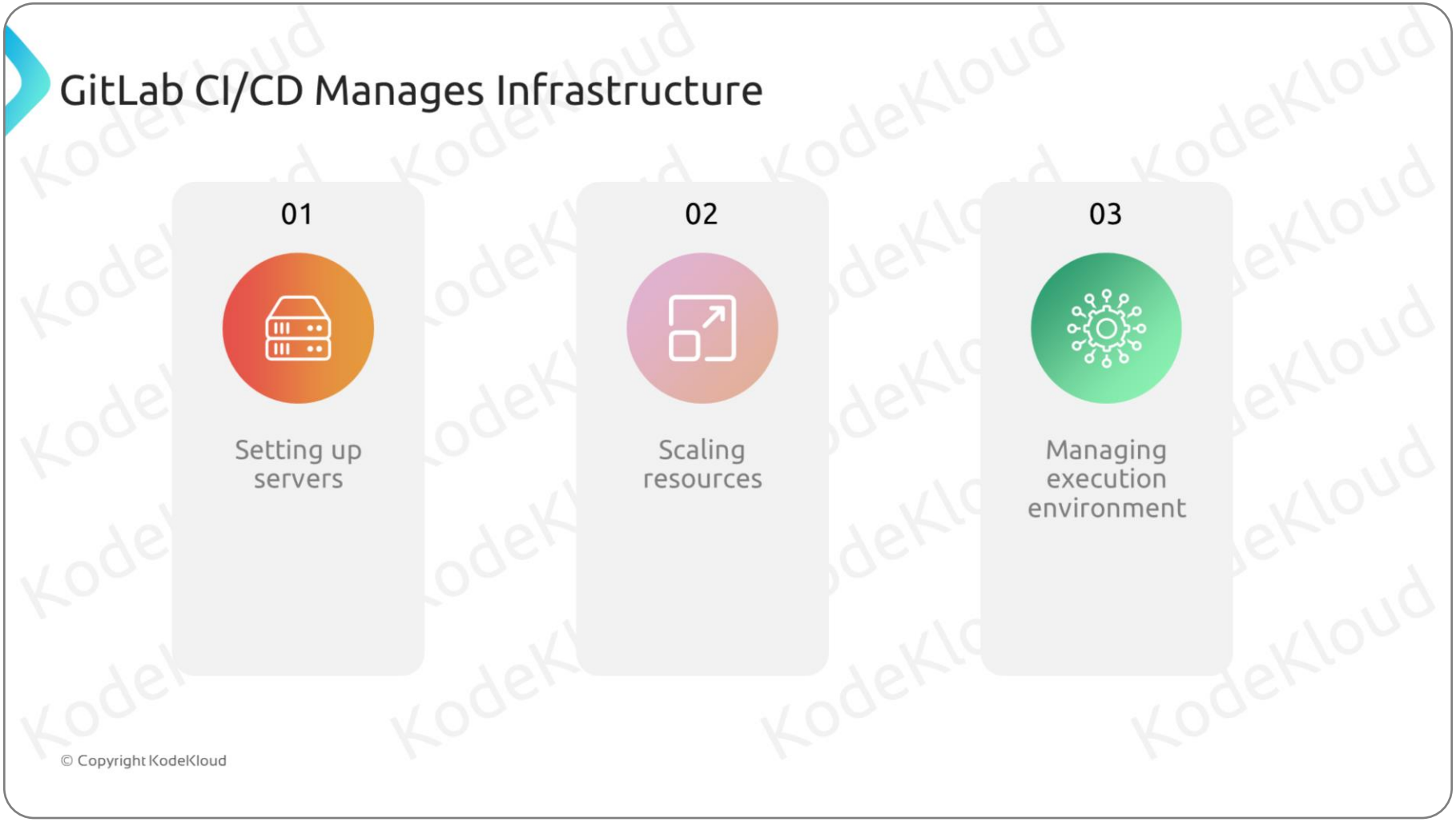
Lets first look at the SaaS Runners. We will explore self-managed runners in a separate segment

SaaS runners are hosted and managed by GitLab. These runners fully integrated with GitLab.com and are enabled by default for all projects, with no configuration required.

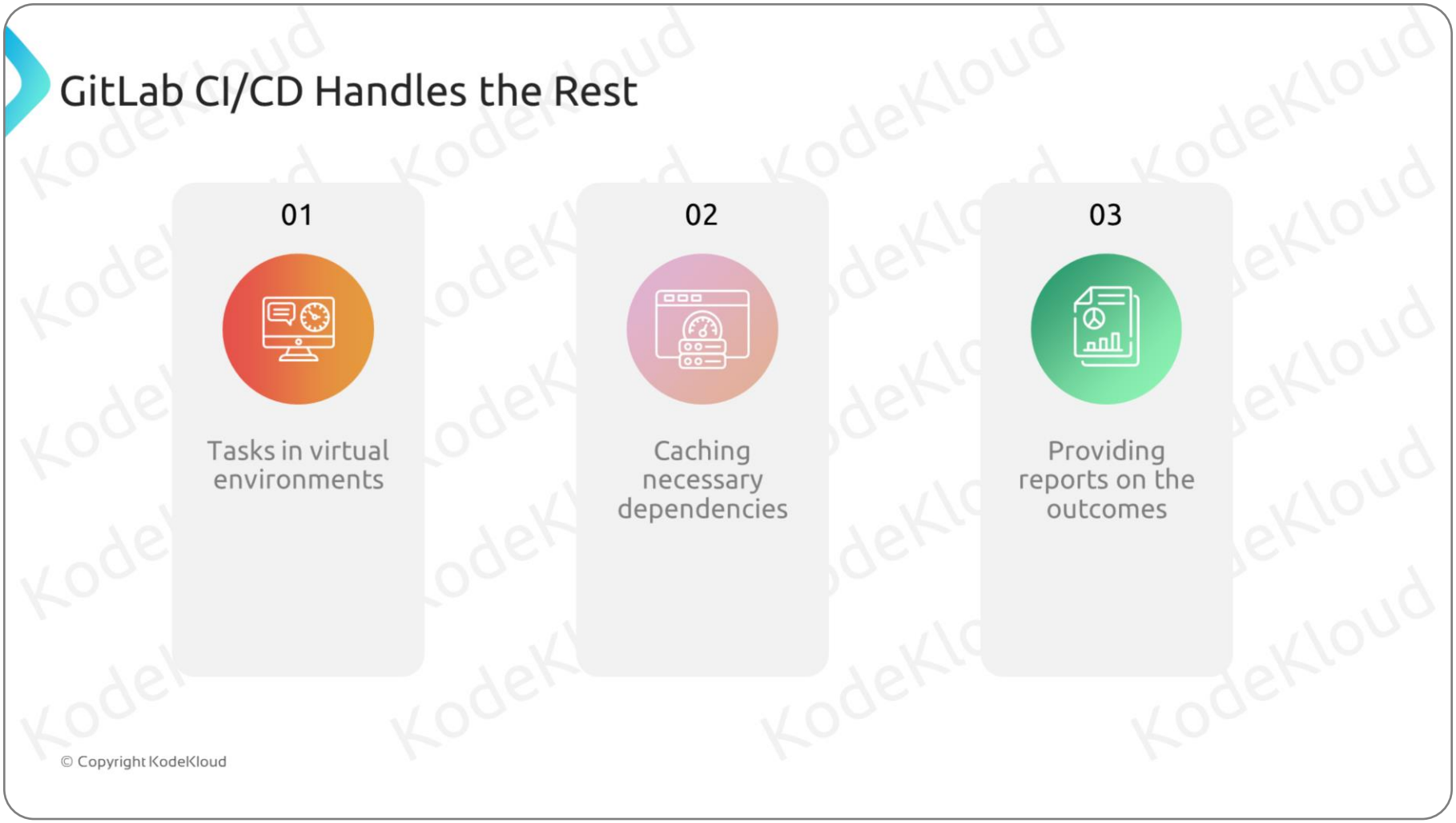
Your jobs can run on:
Linux runners which supports a wide range of languages and tools.
Windows runners (Beta) - Ideal for projects that use Windows-specific tools or require testing on Windows systems

macOS runners (Beta) - A good choice for projects that require macOS-specific tools.
GPU-enabled SaaS runners to accelerate heavy compute workloads for high-performance computing workloads such as the training or deployment of Large Language Models (LLMs) as part of ModelOps workloads.

GitLab CI/CD Manages Infrastructure



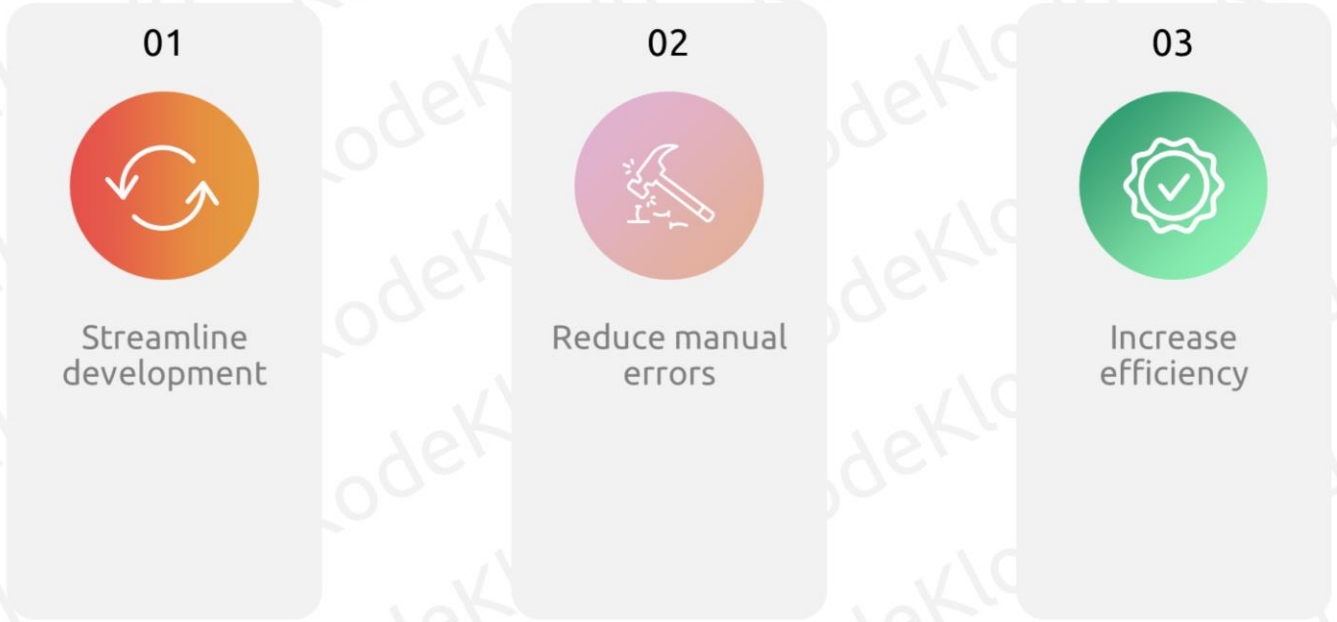
Another advantage of SaaS runners is that GitLab manages the infrastructure for you, which includes setting up servers, scaling resources, and managing the execution environment for your workflows.



Your task is to write workflow configurations in YAML files, and Gitlab CICD handles the rest.

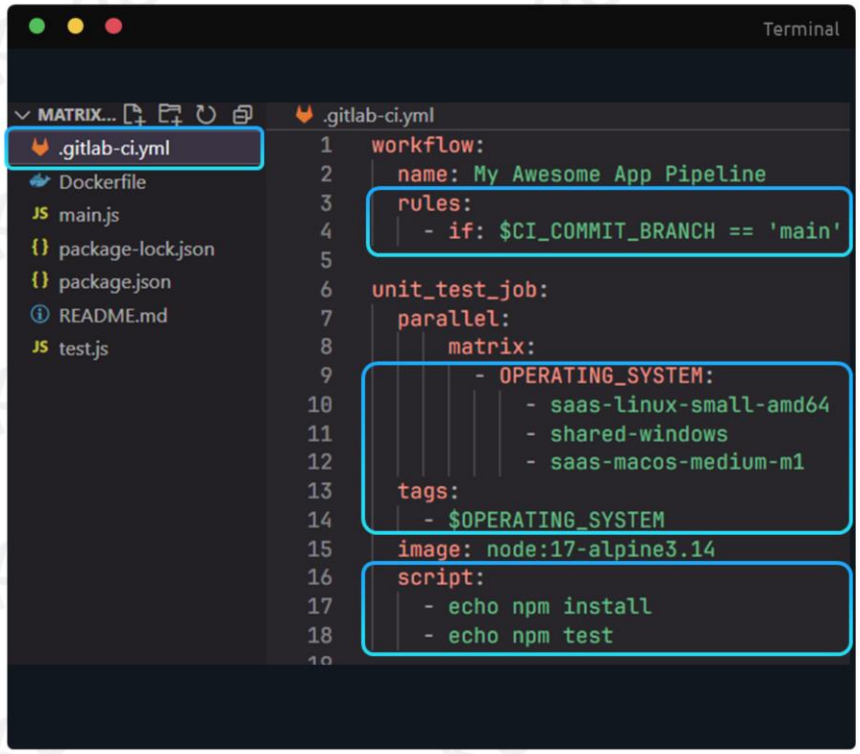
This includes executing your tasks within a newly provisioned virtual environments for each job, caching necessary dependencies, and providing reports on the outcomes.

GitLab CI/CD – Benefits



Finally, this automation helps
streamline development, by releasing new features and bug fixes to your users more quickly.
By automating your deployments you can reduce manual errors, and
increase the efficiency of your software by catching bugs early and often, before they make it to production.

Get Started With GitLab CI/CD



To kickstart your GitLab CI/CD journey, you start with a .gitlab-ci.yml file in your project's root directory which contains the configuration for your CI/CD pipeline. A pipeline is an automated process capable of executing one or more jobs. These pipeline jobs runs in response to specific events occurring in your repository.

For instance, in this example, a simple event like committing code to the main branch of the repository triggers the pipeline.

Within a pipeline, you define one or more jobs. Jobs define what you want to do. For example, test code changes, or deploy to a production environment.

A runner is an virtual machine (VM) agent that is responsible for running your pipeline jobs upon triggering. Gitlab automatically provisions SaaS runners for each job based on the tags specified.

In this example pipeline, there's a job named 'unit-test_job' that runs on three different machines: Windows, Ubuntu, and macOS. Consequently, Gitlab will provision three VMs for these jobs. These VMs are known as GitLab Hosted SaaS Runners.

In this particular workflow example, all the runners are provisioned and operated in parallel.

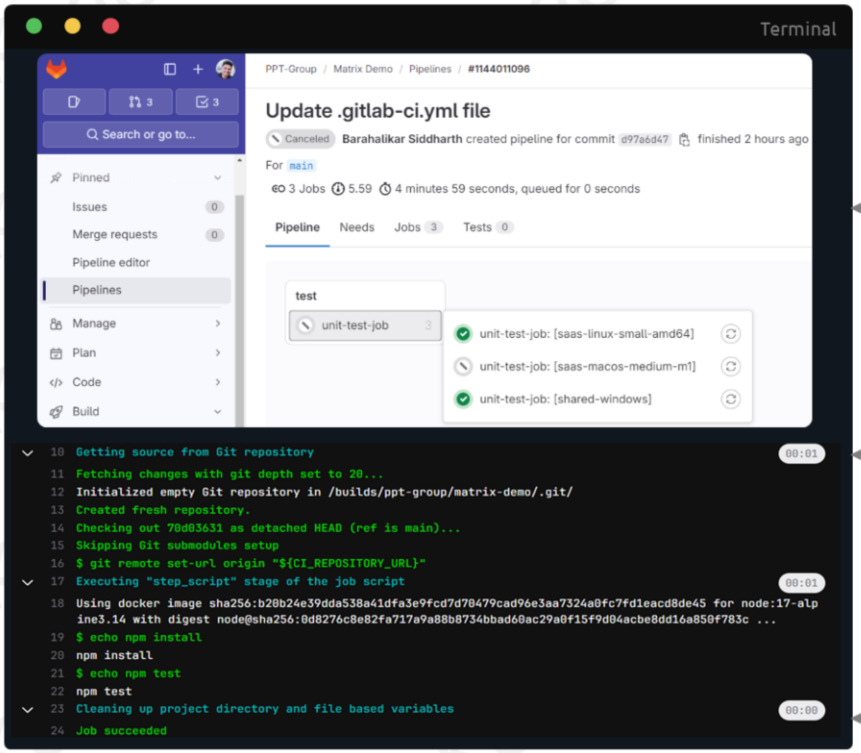
Once the runners spin up, by default, GitLab Runner perform a clone of the repository to fetch the latest commit
.
After cloning the runners start the execution of the steps/commands defined within the script.

In this example workflow, there are 2 commands and they execute sequentially.
First it runs the npm install command and installs the nodejs packages on the runner machine and
Then runs the npm test commands to execute the tests.

These same steps are executed on all three runners concurrently. Consequently, if one runner completes all the step executions, it will be marked as successful, even if the other runners are still in the process of executing their remaining steps.

The pipeline as a whole is marked as successful only once all three runners have completed their tasks successfully.

Logs, Artifacts, Status



During and after the job's execution, you can access the logs, output, and any artifacts for each job via the GitLab User Interface.

This information is received from the runners where the jobs get executed

Within the Gitlab repository, you can simply navigate to the Pipeline Tab to inspect the pipeline status, job logs and

artifacts.

Here, you'll find that the 'unit-test' job has been executed on three different machines, and you can review the logs for each of them individually. This allows for detailed debugging of the workflow's progress and results.

If the job generates any artifacts, you can access and download them as well. GitLab provides a convenient interface for managing and retrieving any artifacts produced during the workflow's execution, enhancing the overall visibility and usability of your automated processes.

Conclusion

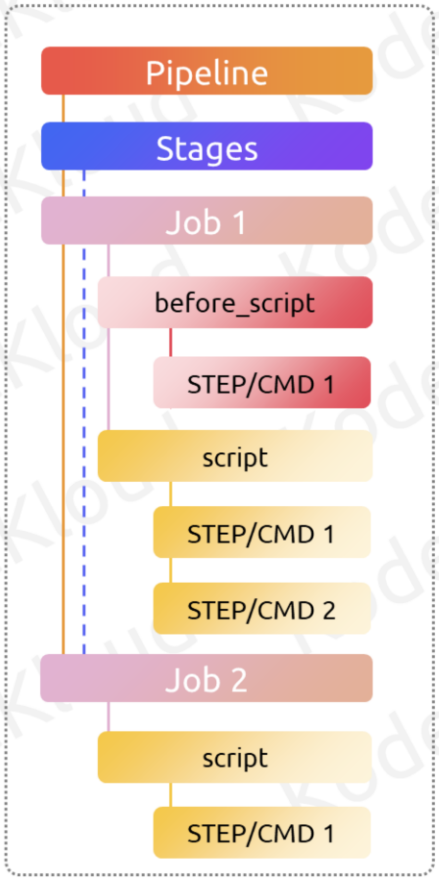
In the video, I've provided a brief overview of pipelines, jobs, scripts, and runners. These concepts will be explored in greater detail in our upcoming sessions.

GitLab CI/CD – Core Components

Core Concepts

```yaml
workflow:
  name: My Awesome App Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'

stages:
  - test
  - deploy

unit_test_job:
  stage: test
  tags:
    - saas-linux-small-amd64
  before_script:
    - echo "Add cmds to install NodeJS"
  script:
    - echo npm install
    - echo npm test
  after_script:
    - echo "Execute after script section"

deploy_job:
  stage: deploy
  script:
    - echo "Deploying ..."
```

Pipeline
Stages
Job 1
Script
Job 2



In this session, we will explore the core components of a Gitlab CI/CD Pipeline.

It consists of 4 key components: Pipeline, Stages, Jobs, and Script. Let's take an example to understand them.

A Pipeline is an automated process capable of executing one or more jobs. It acts as the blueprint for continuous integration and delivery, ensuring that code changes flow smoothly through testing, building, and deployment

phases. The pipeline is defined using YAML files and is located within your repository right beside your code base.

To uniquely identify different pipelines, each pipeline can have an optional workflow name keyword. This name will be visible in the “Pipeline" tab of the GitLab repository.

Pipelines are triggered by events such as code pushes, merge requests, or manual actions. In this example, the rules keyword in workflow trigger’s the pipeline when there is a commit to the main branch. We will talk more about triggers and rules in a later session.

The next core component is a Job. Jobs are the building blocks of a Pipelines, and you can have one or more jobs within a single pipeline. Each job is associated with a runner; it could be a GitLab-hosted or self-hosted runner, which are machines responsible for running the job's instructions. We can optionally define a tags keyword to specify the runner. In this example, the job1 will run on a Linux runner.
If no tag keyword in your job is specified, the jobs will run on a default runner1.
We will talk more about tags, runners, and runner configurations in a later session.

Jobs can be independent or have dependencies on other jobs, creating a network of interconnected tasks.

The core component of a job is a Script, which contains the commands to be executed. It's the heart of the action, defining the specific tasks to be performed.
Scripts can be written as individual commands, as a shell script, or other supported languages, offering flexibility in automation tools.
All the steps/commands within a script are executed sequentially.
Typical actions include building code, running tests, deploying artifacts, performing security scans, and generating documentation.

Each job can also have a before/after scripts, these are optional scripts that run before or after the main script of a job, providing opportunities for setup and teardown activities.

Common uses of before_script includes:

Installing dependencies, Preparing test environments, Setting up database connections

Common uses of after_script includes:

Cleaning up resources

If a pipeline has multiple jobs, by default, the Jobs run in parallel at the same time. To logically group jobs within a pipeline, we use a concept of stages.

Stages provides Logical groupings of jobs within a pipeline. They represent distinct phases of the CI/CD process, often aligned with development milestones.
Stages Execute jobs in a predetermined order, thereby enabling conditional execution, allowing flexibility based on outcomes of previous stages, optimizing resource usage, and preventing unnecessary actions.
Common stages include build (compiling code), test (running automated tests), deploy (publishing code to production environments), and security (scanning for vulnerabilities).
In this example, we have defined 2 stages test and deploy and, hence, the deploy job will start execution only after the successful execution of the unit test job.

Pipelines in GitLab CI/CD offer a variety of flexible configurations to tailor your software delivery process:
Basic Pipelines: Execute jobs within a stage concurrently, moving to the next stage only after all jobs in the current stage are complete.
DAG Pipelines: Optimize efficiency by running jobs based on dependencies, potentially reducing execution time compared to basic pipelines.
Merge Request Pipelines: Trigger specifically for merge requests, ensuring code quality and consistency before merging

branches.
Merged Results Pipelines: Simulate the merged state of a merge request, providing early insights into potential conflicts or issues.
Merge Trains: Queue merge requests for sequential execution, ensuring a controlled and orderly integration process.
Parent-Child Pipelines: Manage complexity by breaking down extensive pipelines into a parent pipeline that orchestrates multiple child pipelines, often used in monorepo environments.
Multi-Project Pipelines: Coordinate pipelines across different projects, fostering collaboration and streamlining cross-project dependencies.

Choose the configuration that best suits your workflow, team structure, and project needs to achieve efficient and reliable software delivery.

GitLab CI/CD Architecture

GitLab CI/CD Architecture – Key Components

In the previous demo, we saw how a pipeline was defined and executed.

In this session, we will understand the Key Components of the GitLab CI/CD Architecture which were responsible for defining and executing a pipeline.

The components are Gitlab Server, Runner, and Executor.

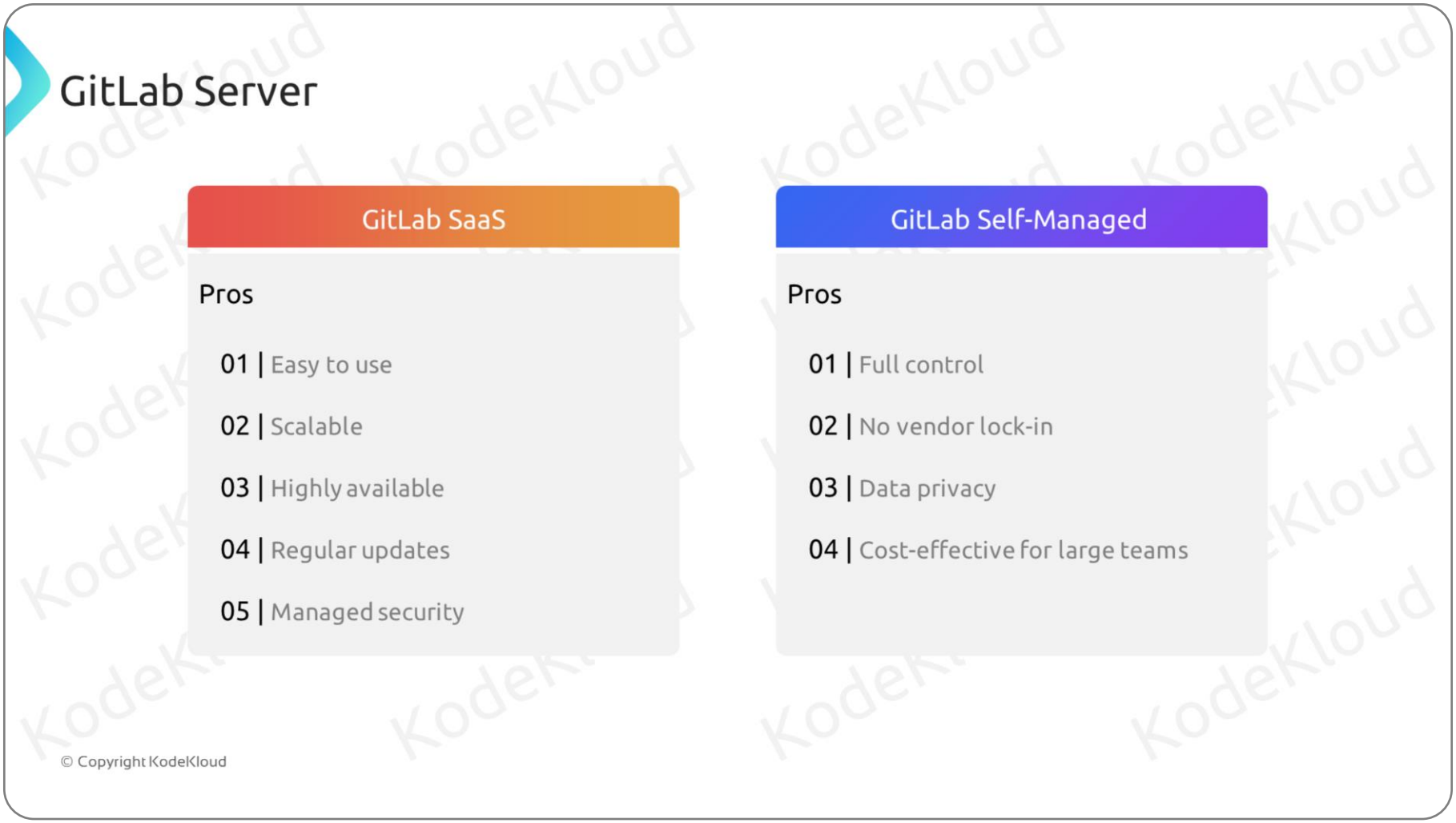
========

Building on our previous demo of pipeline execution, we'll now focus on the architectural elements that orchestrate the entire process.
The three key components of the GitLab CI/CD architecture are Gitlab Server, Runner, and Executor.

GitLab Server

Feature	GitLab SaaS	GitLab Self-Managed
Hosting	Hosted by GitLab	Hosted on your own infrastructure
Setup	Simple, no setup required	Requires installation and configuration
Maintenance	Managed by GitLab	Managed by you
Control	Limited customization	Full control over configuration
Scalability	Scales automatically	Scales within your infrastructure
Security	Managed by GitLab	Managed by you
Cost	Pay-as-you-go pricing	No recurring costs (after initial setup)
Data Privacy	Data stored on GitLab's servers	Data stored on your own servers
Vendor Lock-in	Potential vendor lock-in	No vendor lock-in
Best for	Small teams, simple needs, ease of use	Large teams, complex needs, control, privacy

GitLab Server is an open-source DevOps platform that allows you to manage your entire software development lifecycle from a single location. It provides features for version control, issue tracking, CI/CD pipelines, package registries, and more. It comes in 2 flavors - Gitlab SaaS vs GitLab Self-Managed.

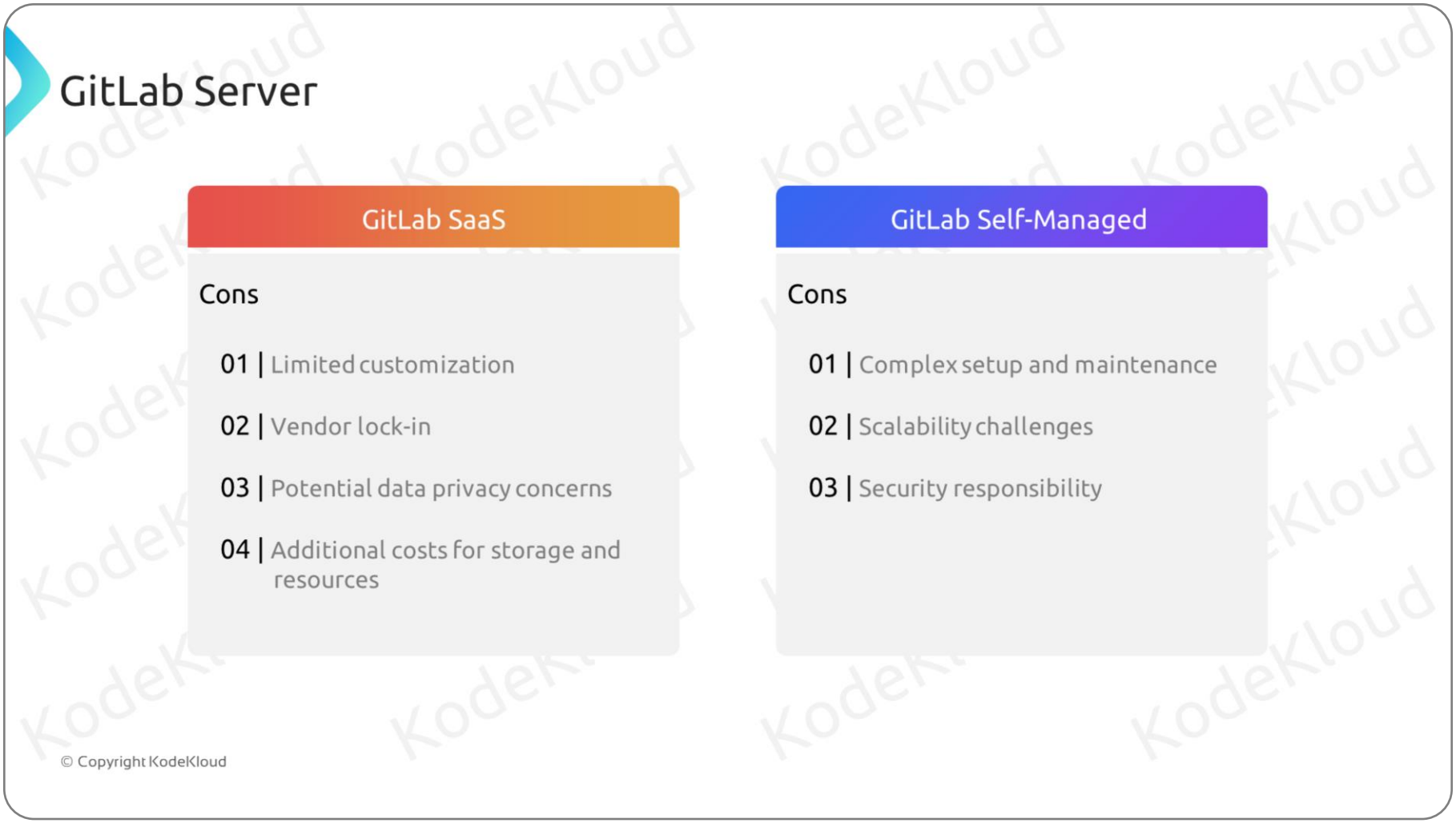


Both GitLab SaaS and GitLab self-managed offers the same capabilities, but they cater to different needs and come with their own set of advantages and disadvantages.

GitLab SaaS:
Pros:
Easy to use: No infrastructure setup or maintenance required. Gets started quickly with minimal technical expertise.

Scalable: Pay-as-you-go pricing scales seamlessly with your needs. No need to worry about managing server capacity.
Highly available: GitLab manages the infrastructure, ensuring high uptime and redundancy.
Regular updates: Automatic updates guarantee access to the latest features and security patches.
Managed security: GitLab handles security patching and vulnerability management.

GitLab Self-Managed:
Pros:
Full control: Customize the platform to your specific needs and security requirements.
No vendor lock-in: Easily migrate to another platform or host on your own infrastructure.
Data privacy: Maintain complete control over your code and data.
Cost-effective for large teams: Potentially cheaper than SaaS for high-volume usage.



GitLab SaaS:
Cons:
Limited customization: Less control over the platform configuration compared to self-hosted.
Vendor lock-in: Migrating to another platform can be complex and costly.
Potential data privacy concerns: Your code and data are hosted on GitLab's infrastructure.
Additional costs for storage and resources: Can become expensive for large teams or projects with high resource

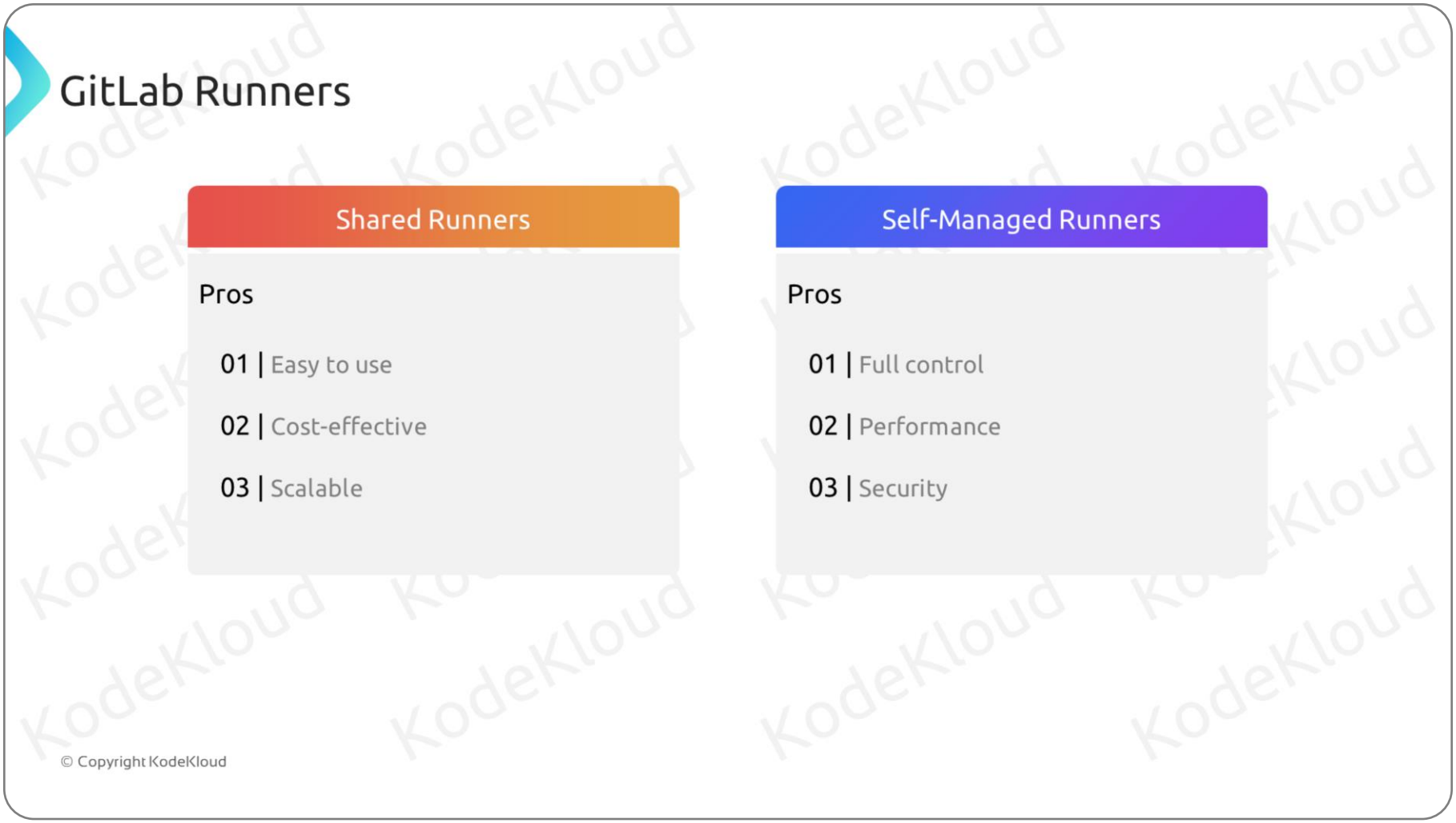
demands.

GitLab Self-Managed:
Cons:
Complex setup and maintenance: Requires technical expertise to manage the infrastructure and keep the platform updated.
Scalability challenges: Scaling beyond your available infrastructure can be expensive and complex.
Security responsibility: You are responsible for applying security updates and managing vulnerabilities.

GitLab Runners

Feature	Shared Runners	Self-Managed Runners
Hosting	Hosted by GitLab	Hosted on your own infrastructure
Setup	No setup required	Requires installation and configuration
Maintenance	Managed by GitLab	Managed by you
Control	Limited control over configuration	Full control over configuration
Scalability	Scales automatically	Scales within your infrastructure
Security	Shares resources with other projects	Isolated from other projects
Cost	Included in some GitLab tiers	May require additional hardware/software costs
Best for	Small teams, simple needs, ease of use, cost-effectiveness	Large teams, complex needs, control, performance, security
Hosting	Hosted by GitLab	Hosted on your own infrastructure
Setup	No setup required	Requires installation and configuration

In GitLab CI/CD, Runners are External machines or virtual machines responsible for executing pipeline jobs.
They Register with GitLab server to receive and process job instructions.



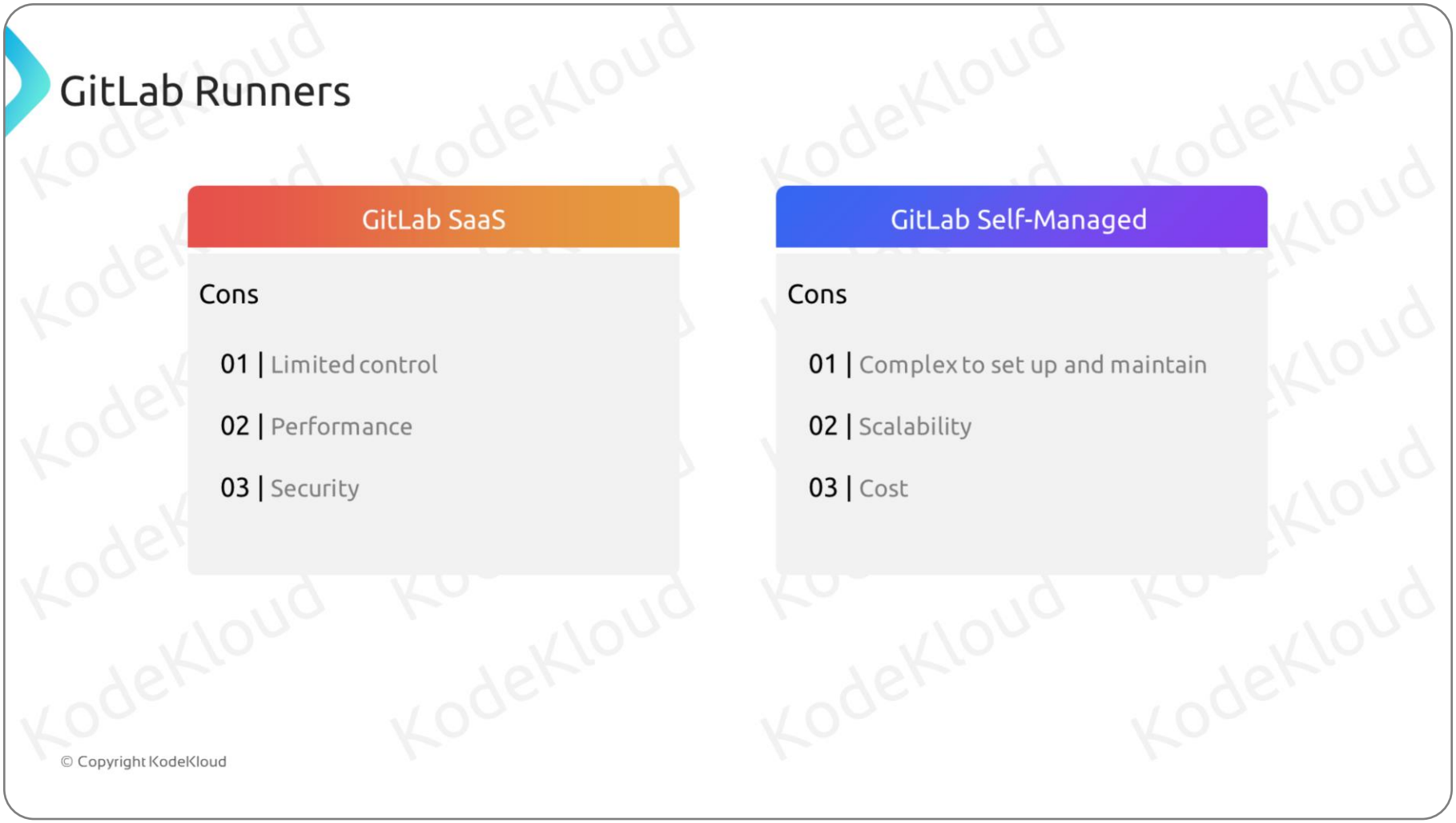
You have two main options: shared runners and self-managed runners. Each has its own set of advantages and disadvantages, so choosing the right one for your needs is important.

Shared Runners are runners provided and managed by GitLab. They are available to all projects on a GitLab SaaS instance and are used on a first-come, first-served basis.
Pros:

Easy to use: No setup or maintenance required. Just start using them!
Cost-effective: Included in your GitLab subscription for some tiers.
Scalable: GitLab automatically scales the number of runners to meet demand.

Self-managed runners are runners that you install and manage yourself on your own infrastructure.
Pros:

Full control: You have complete control over the runner configuration, environment, and security.
Performance: Can be faster and more reliable than shared runners.
Security: Isolated from other projects, potentially reducing security risks.



Shared Runners:
Cons:

Limited control: You don't have control over the runner configuration or environment.
Performance: Runs may be slower due to competition for resources.
Security: Shares resources with other projects, potentially introducing security risks.

Self-managed runners

Cons:

- Complex to set up and maintain: As these require technical expertise to install, configure, and maintain.
- Scalability: Can be difficult to scale as your needs grow.
- Cost: May require additional hardware and software costs.

Runner Executors

In GitLab CI/CD, runners are the workers who execute your pipeline jobs. But under the hood of each runner lies a hidden engine called an executor, determining the environment and tools available for those jobs.

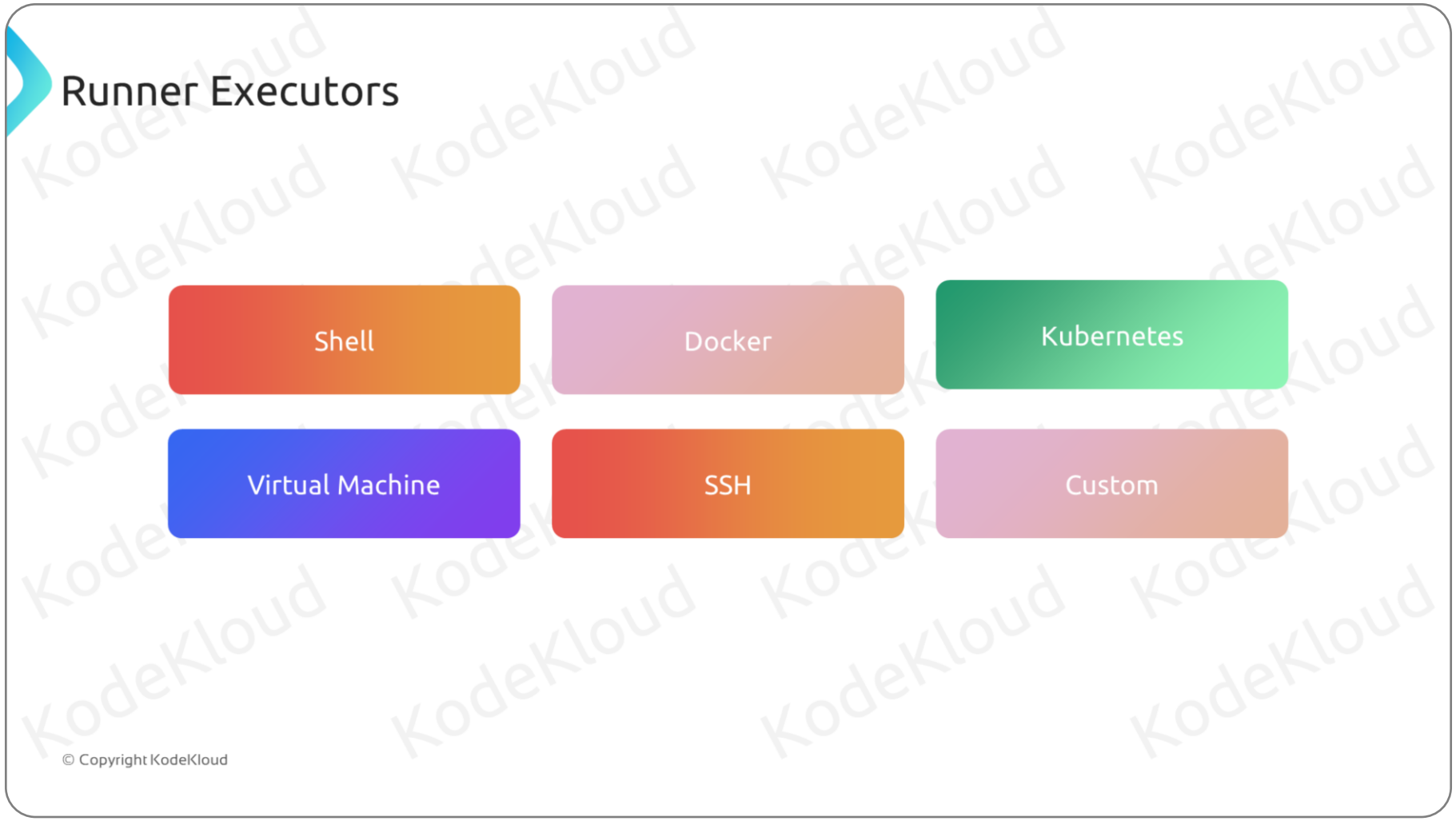
Runner Executors

Choosing the right executor is crucial for ensuring efficient, secure, and reliable pipeline execution.

Runner Executors

Executors, like containers, define the job's environment (OS, tools, resources), ensuring consistent, isolated runs to prevent conflicts and unexpected behavior.

So, what are GitLab Runner Executors?
Think of executors as containers within your runner, defining the operating system, tools, and resources a job has access to. They ensure jobs run in a consistent and isolated environment, preventing conflicts and unexpected behavior.



There are around 10 Types of GitLab Runner Executors:

Runner Executors

- Shell
- Docker
- Kubernetes
- Virtual Machine
- SSH
- Custom

Cons

01 | Lack of isolation

02 | Limited reproducibility

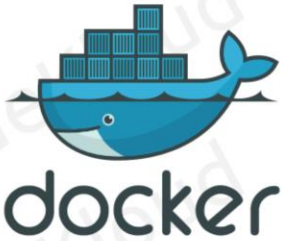
03 | Security concerns

Shell:
This basic executor runs jobs directly on the runner's operating system using its shell (e.g., Bash, PowerShell). Simple and lightweight, it's suitable for quick scripts or testing basic functionality.
Cons:
Lack of isolation: Shares the runner's operating system and resources, risking conflicts and unexpected behavior.
Limited reproducibility: Results may vary depending on the runner's environment and installed tools.

Security concerns: Sharing resources might pose security risks if jobs require different levels of access.

Runner Executors

- Shell
- Docker
- Kubernetes
- Virtual Machine
- SSH
- Custom



Cons

01 | Overhead and complexity

02 | Increased resource usage

03 | Limited access to underlying system

Docker:
For greater isolation and reproducibility, Docker executors run jobs inside Docker containers. These containers encapsulate the specific environment (OS, libraries, tools) needed for the job, guaranteeing consistency across machines. We can Choose from countless pre-built Docker images with specific tools and environments readily available.
Cons:
Overhead and complexity: Requires additional configuration and understanding of Docker concepts.

Increased resource usage: Running multiple containers can consume more system resources on the runner.
Limited access to underlying system: Docker containers often have restricted access to the runner's native resources.

Runner Executors

- Shell
- Docker
- Kubernetes
- Virtual Machine
- SSH
- Custom

Cons

01 | Complex setup and maintenance

02 | Potential dependency on existing infrastructure

03 | Increased learning curve

Kubernetes:
If your project utilizes Kubernetes clusters, the Kubernetes executor lets you seamlessly run jobs as pods within those clusters. This leverages the cluster's resource management and scaling capabilities for efficient execution and parallel processing.
Cons:
Complex setup and maintenance: Requires familiarity with Kubernetes configuration and management.

Potential dependency on existing infrastructure: May not be feasible if you don't have a Kubernetes cluster.
Increased learning curve: Understanding best practices and security considerations for running jobs in Kubernetes is crucial.

Runner Executors

- Shell
- Docker
- Kubernetes
- **Virtual Machine**
- SSH
- Custom

Cons

01 | High resource usage

02 | Slower startup times

03 | Complex management

Virtual Machine (VM):
Need a completely clean slate for each job? The VM executor creates a new virtual machine for each job execution, guaranteeing isolation and eliminating potential contamination from previous runs.
Cons:
High resource usage: VMs require significant disk space, memory, and CPU resources, potentially taxing your infrastructure.

Slower startup times: Creating and initializing VMs takes longer than starting containers, slowing down pipeline execution.
Complex management: Requires expertise in VM management and configuration.

Runner Executors

- Shell
- Docker
- Kubernetes
- Virtual Machine
- SSH
- Custom

Cons

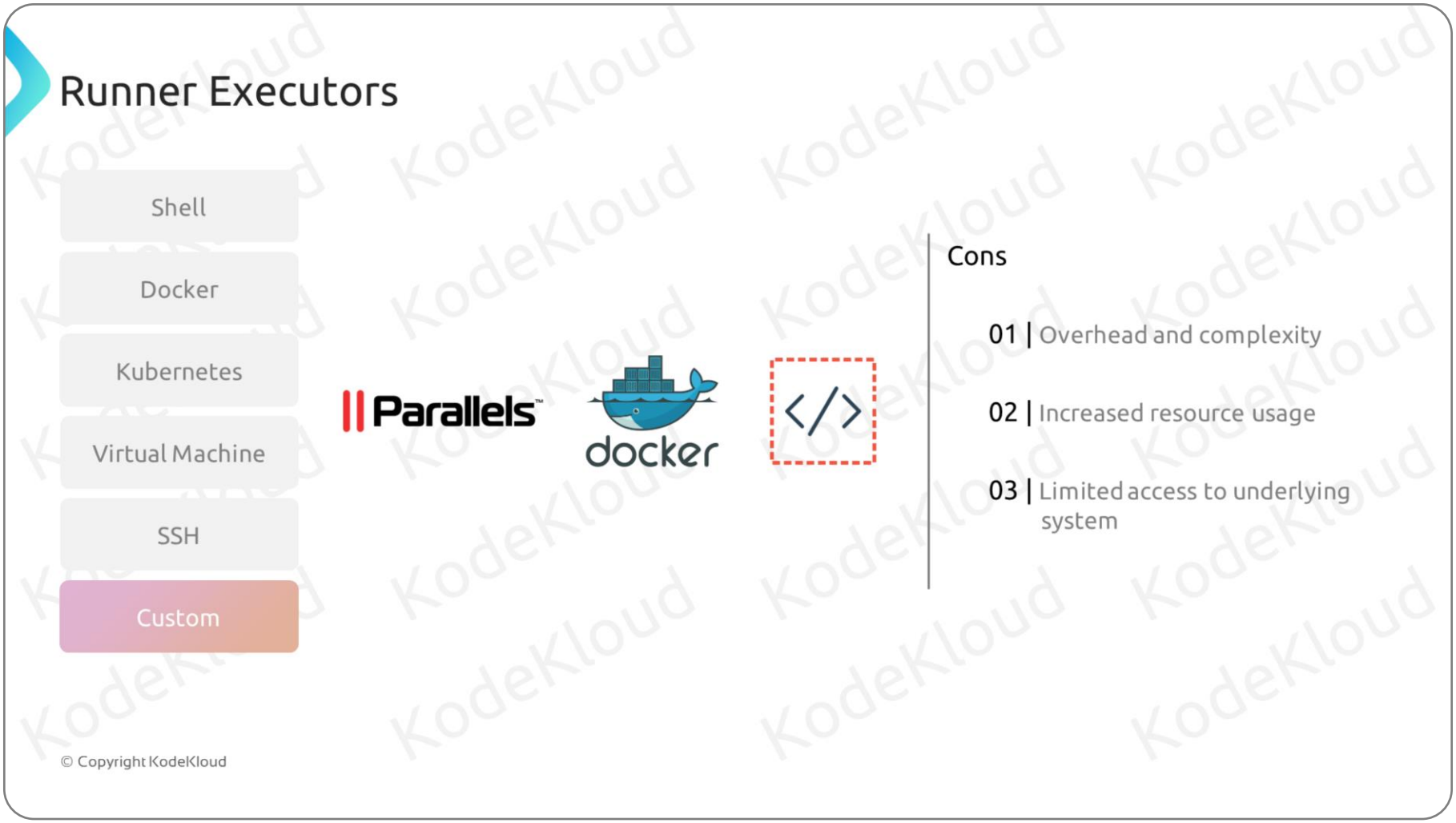
01 | Security concerns

02 | Network dependency

03 | Limited control and visibility

SSH:
This executor allows running jobs remotely on another machine via SSH connection. It's useful for utilizing specific resources available on a different machine or integrating with pre-existing infrastructure.
Cons:
Security concerns: Requires secure SSH access and careful configuration to maintain job and system security.
Network dependency: Relies on reliable network connectivity between the runner and the remote machine.

Limited control and visibility: Managing and monitoring jobs running on a remote machine might be more challenging.



Beyond the Basics:

GitLab offers additional specialized executors like Parallels, Docker Machine, and even custom ones you can develop yourself. Each caters to specific needs and environments, ensuring flexibility and catering to diverse workflows.

Selecting the Executor

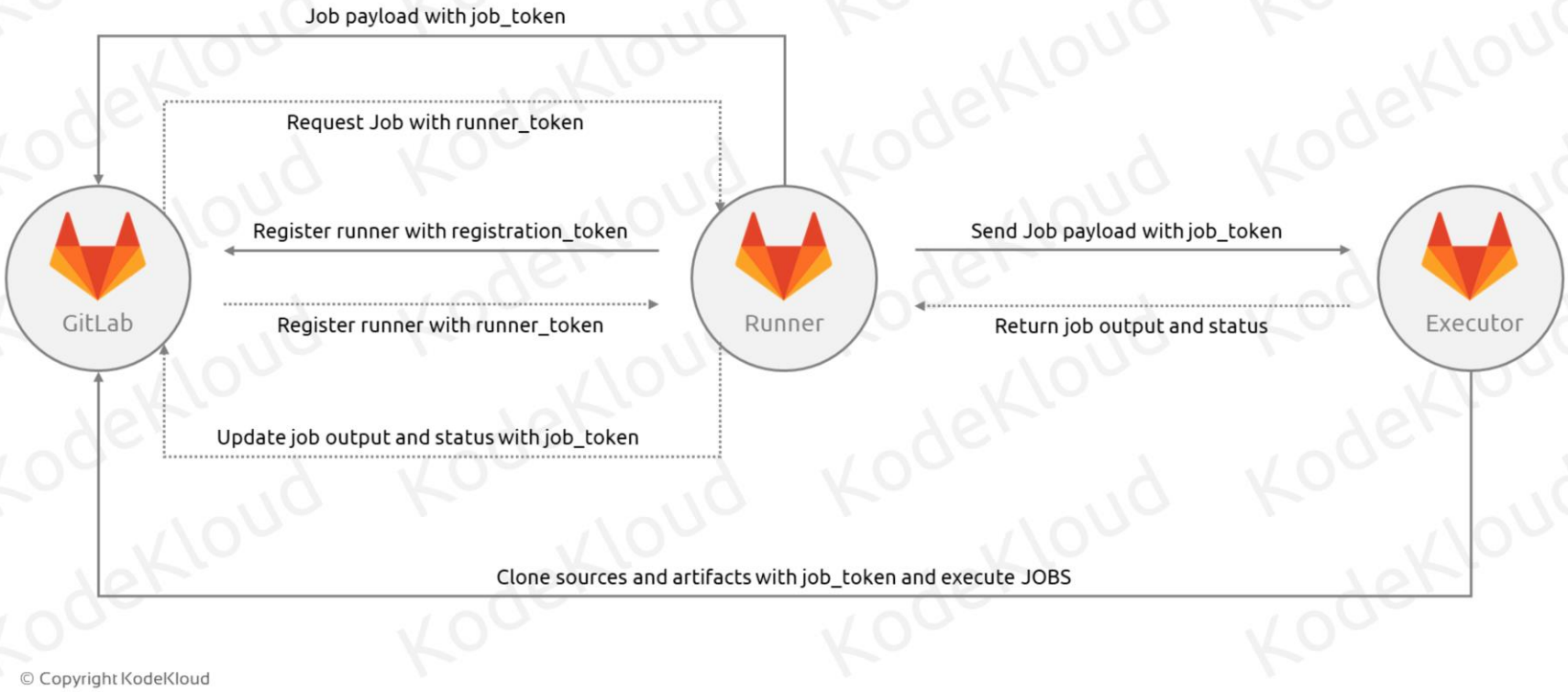
Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Clean build environment for every build	×	×	✓	✓	✓	✓	conditional(4)
Reuse previous clone if it exists	✓	✓	×	×	✓	×	conditional(4)
Runner file system access protected(5)	✓	×	✓	✓	✓	✓	conditional
Migrate runner machine	×	×	partial	partial	✓	✓	✓
Zero-configuration support for concurrent builds	×	× (1)	✓	✓	✓	✓	conditional(4)
Complicated build environments	×	× (2)	✓ (3)	✓ (3)	✓	✓	✓
Debugging build problems	easy	easy	hard	hard	medium	medium	medium

Source - https://docs.gitlab.com/runner/executors/#selecting-the-executor

The executors support different platforms and methodologies for building a project. The table shows the key facts for each executor which will help you decide which executor to use.

Do check out the official docs for more information https://docs.gitlab.com/runner/executors/#selecting-the-executor

GitLab CI/CD Architecture – Key Components



Let's understand how GitLab Server, GitLab Runner, and Executor interact during the CI/CD process.

Registration

The registration process is the first step in setting up a GitLab Runner.
During registration, the runner makes a POST request to the runners endpoint of the GitLab server.
This request includes a registration_token, which is a unique identifier provided by the GitLab server.

- The server responds with a runner_token upon successful registration.
- This runner_token is then used by the runner for future communications with the GitLab server.

Job Requesting and Handling

- Once the runner is registered, it enters a loop where it continuously requests jobs from the GitLab server.
- The runner makes POST requests to the request endpoint, using the runner_token received during registration.
- The server responds with a job payload, which includes a job_token.
- This job_token is used by the runner to authenticate itself when it performs actions related to the job.

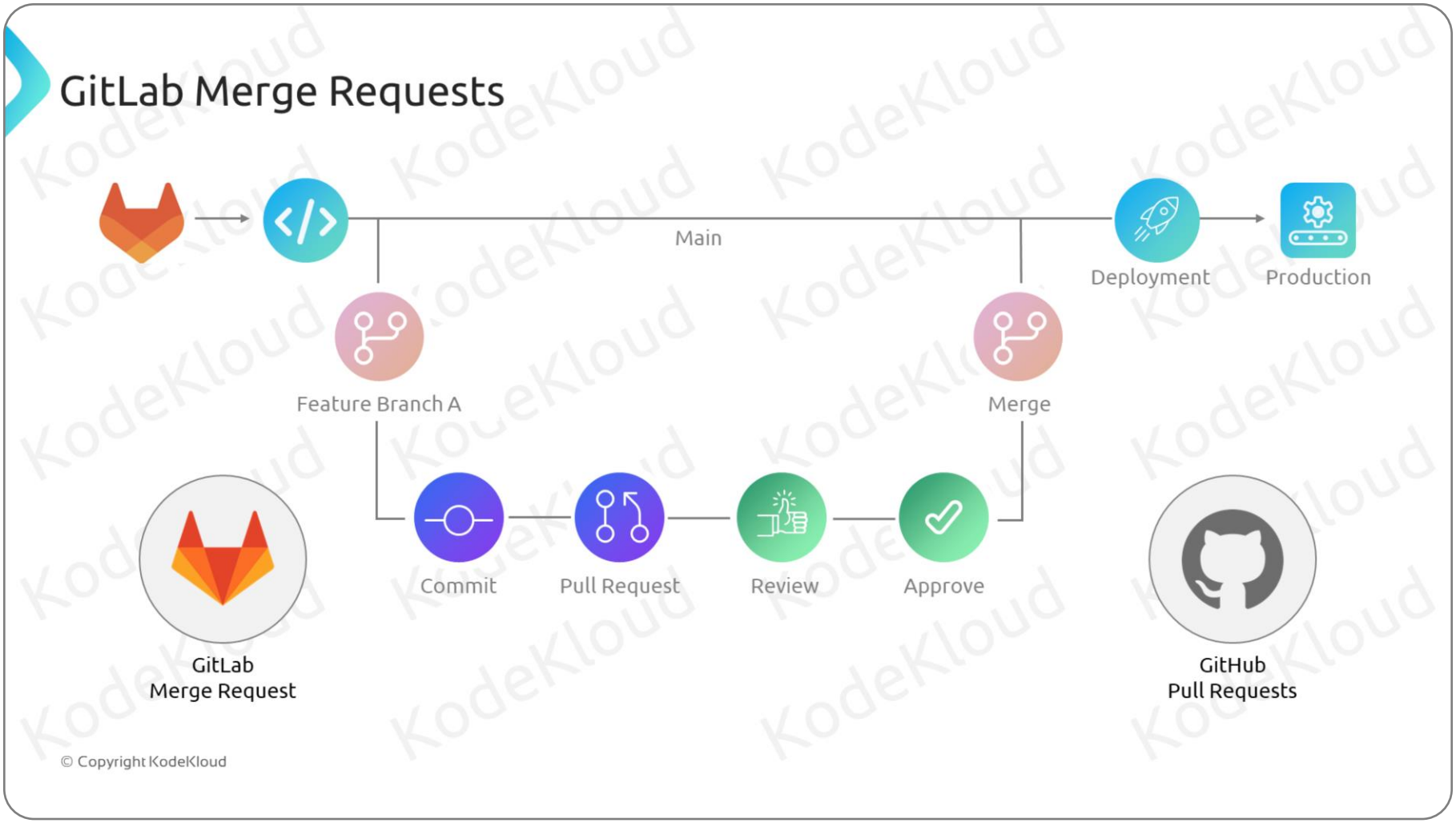
Job Execution

- The job payload received from the server is passed on to the executor.
- The executor is responsible for executing the job.
- It uses the job_token to authenticate itself when it needs to clone sources or download artifacts from the GitLab server.
- The executor runs the job in an environment that is isolated from the runner and other jobs.

Return Job Output and Status

- After the job is executed, the executor returns the job output and status.
- The job_token is still used to authenticate these actions.
- The runner then updates the job output and status on the GitLab server using the job_token.
- This allows the server to track the progress and result of the job.

GitLab
Merge Requests



A merge request lets you propose merging changes from a source branch to a target branch.

Does this remind you of anything?

Both GitLab Merge Requests and GitHub Pull Requests serve the same purpose: they propose changes from a branch to be merged into the main branch of a code repository. However, they differ in terminology and some workflow

implementation details.

Both allow developers to:
Create branching points for their changes.
Commit proposed changes (code, documentation, etc.) for review.
Discuss changes with the team and receive feedback/approvals.
Get changes reviewed and potentially merged into the main codebase.

Project Status Meeting - 1

Project Status Meeting - 1

Priority	Task	Assigned	Status	Comments/Issue
01	Understand Requirement	Alice	Not started	
02	Unit Testing	Alice	Not started	
03	Code Coverage	Alice	Not started	
04	Containerization	Alice	Not started	
05	Dev Integration Testing	Alice	Not started	
06	Manual Approval	Alice	Not started	
07	Kubernetes Prod Deployment	Alice	Not started	
08	Prod Integration Testing	Alice	Not started	

Over the course of this project, we will be developing a GitHub workflow for a Node.js application.

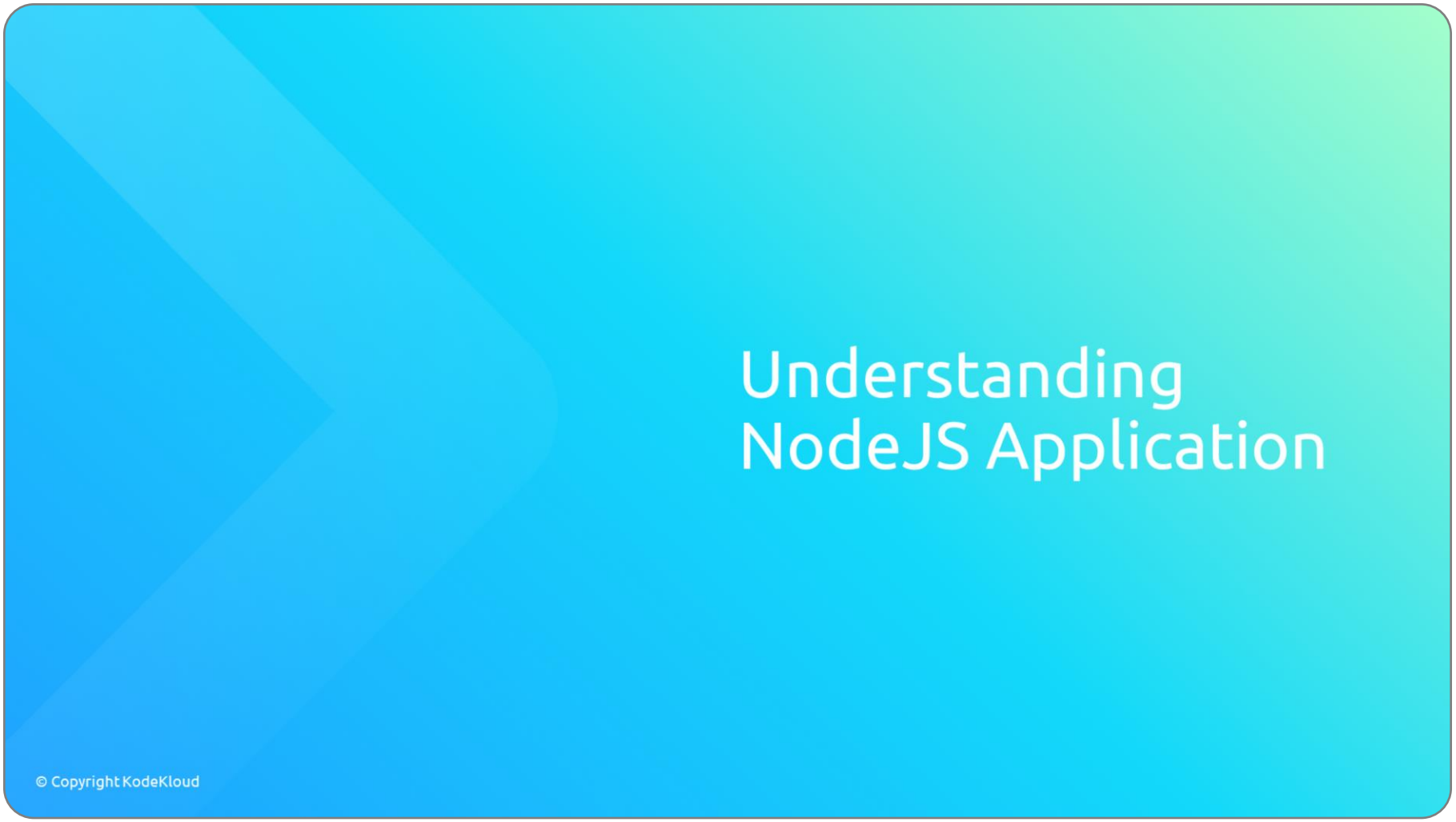
To simplify understanding and make the process more streamlined, we will utilize project status meeting slides. These slides will present an overview of the tasks we will be performing and the reasons behind the changes. Additionally, comments will be provided wherever necessary to further clarify the context.

Project Status Meeting - 1

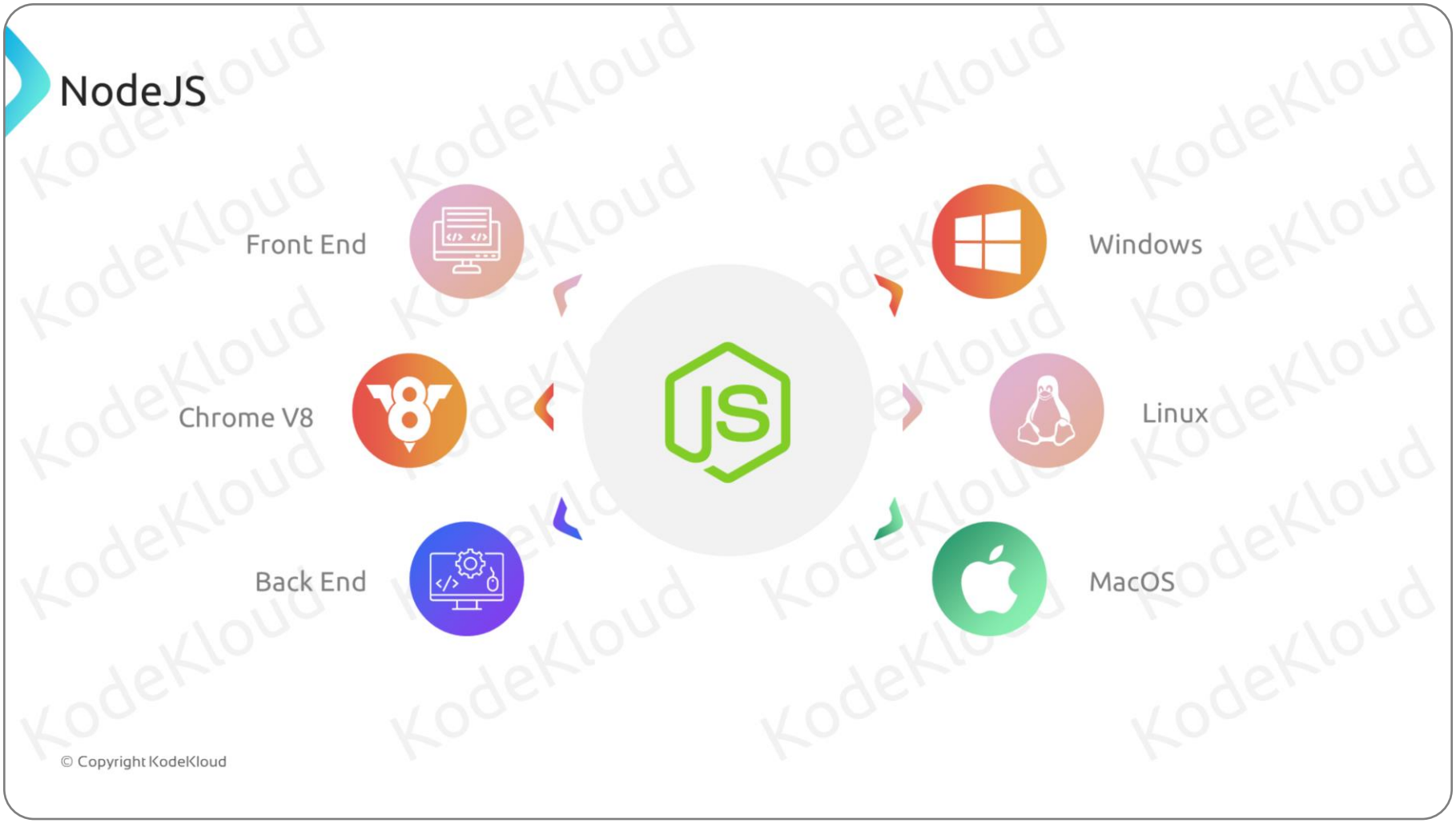
Priority	Task	Assigned	Status	Comments/Issue
01	Understand Requirement	Alice	In progress	
02	Unit Testing	Alice	In progress	
03	Code Coverage	Alice	In progress	
04	Containerization	Alice	In progress	
05	Dev Integration Testing	Alice	Not started	
06	Manual Approval	Alice	Not started	
07	Kubernetes Prod Deployment	Alice	Not started	
08	Prod Integration Testing	Alice	Not started	

We will begin with the first 4 tasks, by understanding the node.js application and DevOps requirement.

Then, we'll start building the workflow by creating jobs for unit testing, code coverage, and containerization.



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

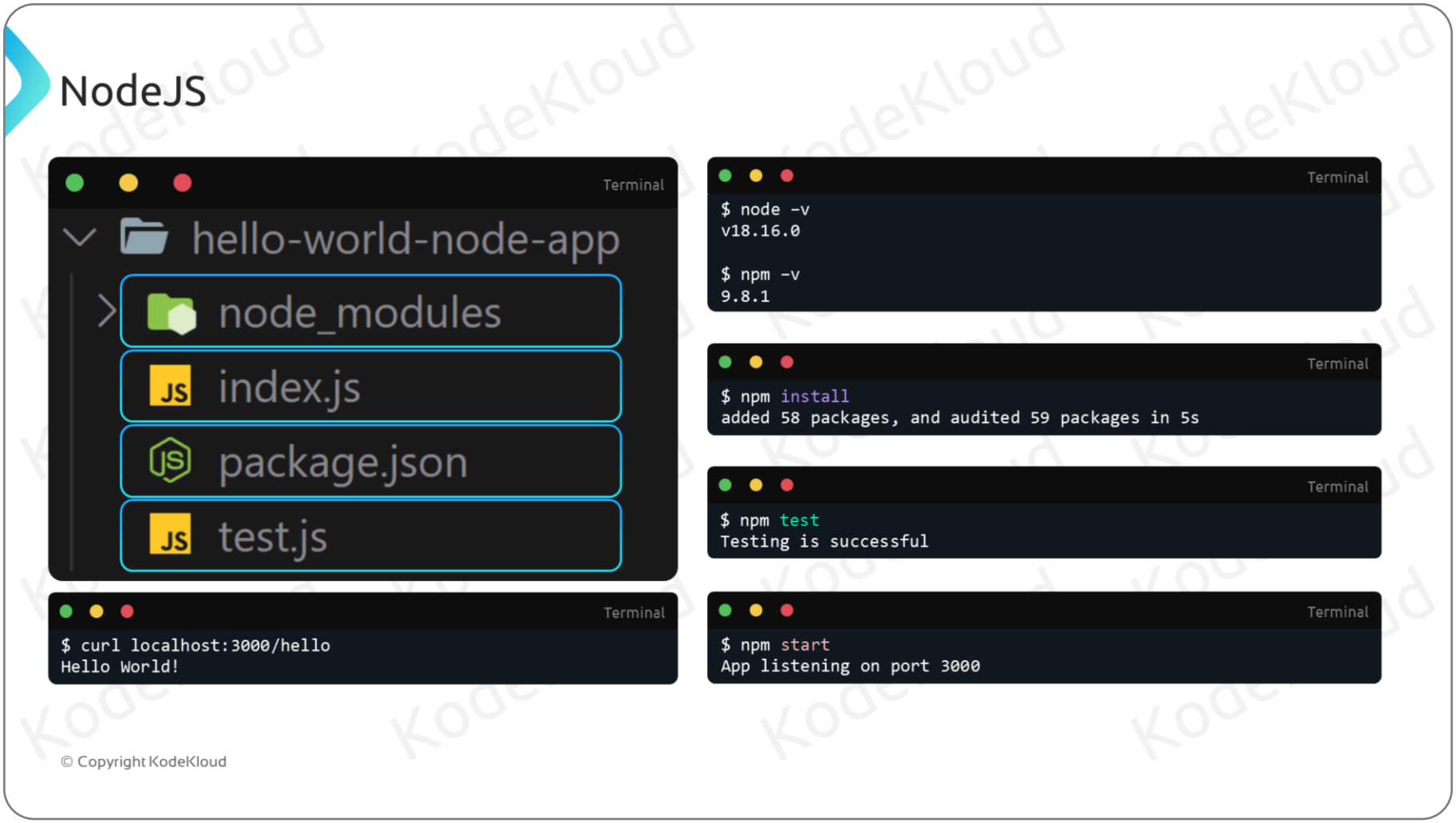


In a later module, we'll utilize the Node.js application to develop a customized GitHub action pipeline.

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.



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.

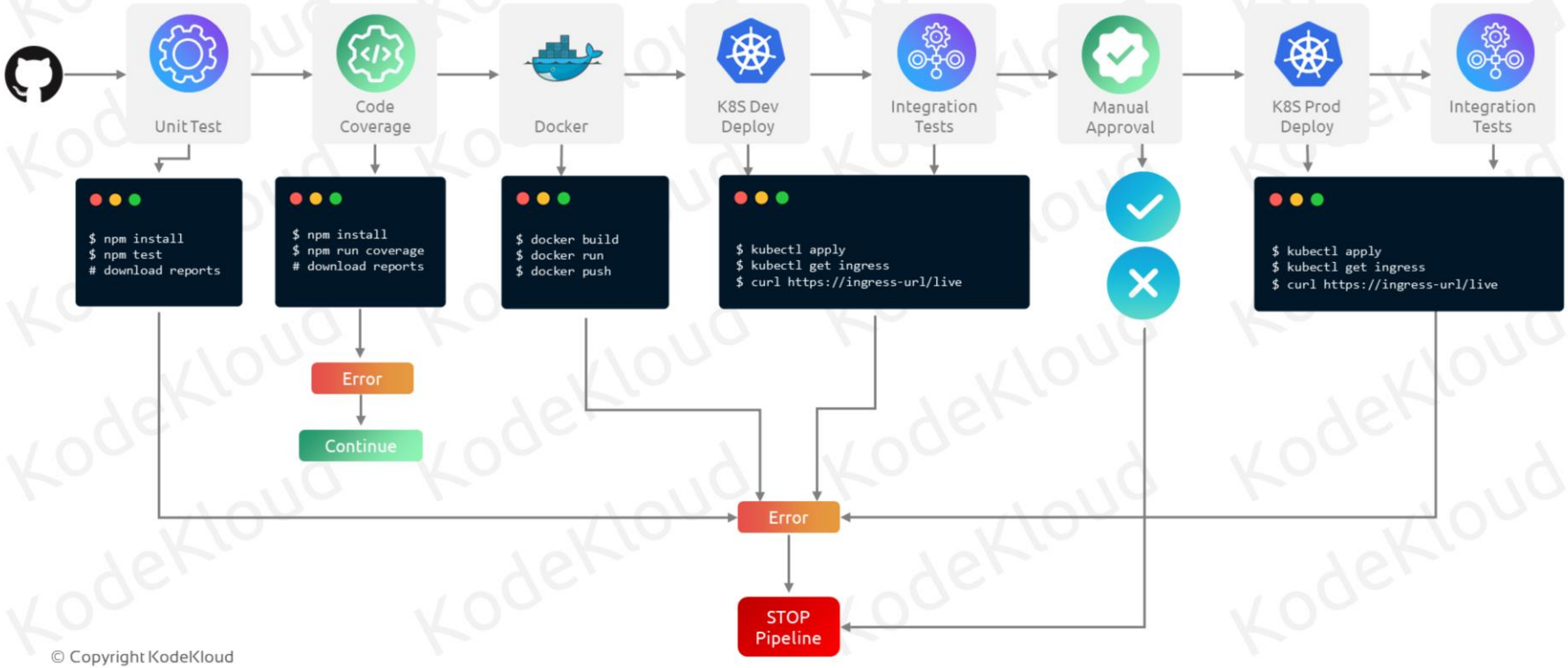
The npm test command is commonly used to run tests for a JavaScript project. This command runs all the tests defined in the test.js file.

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

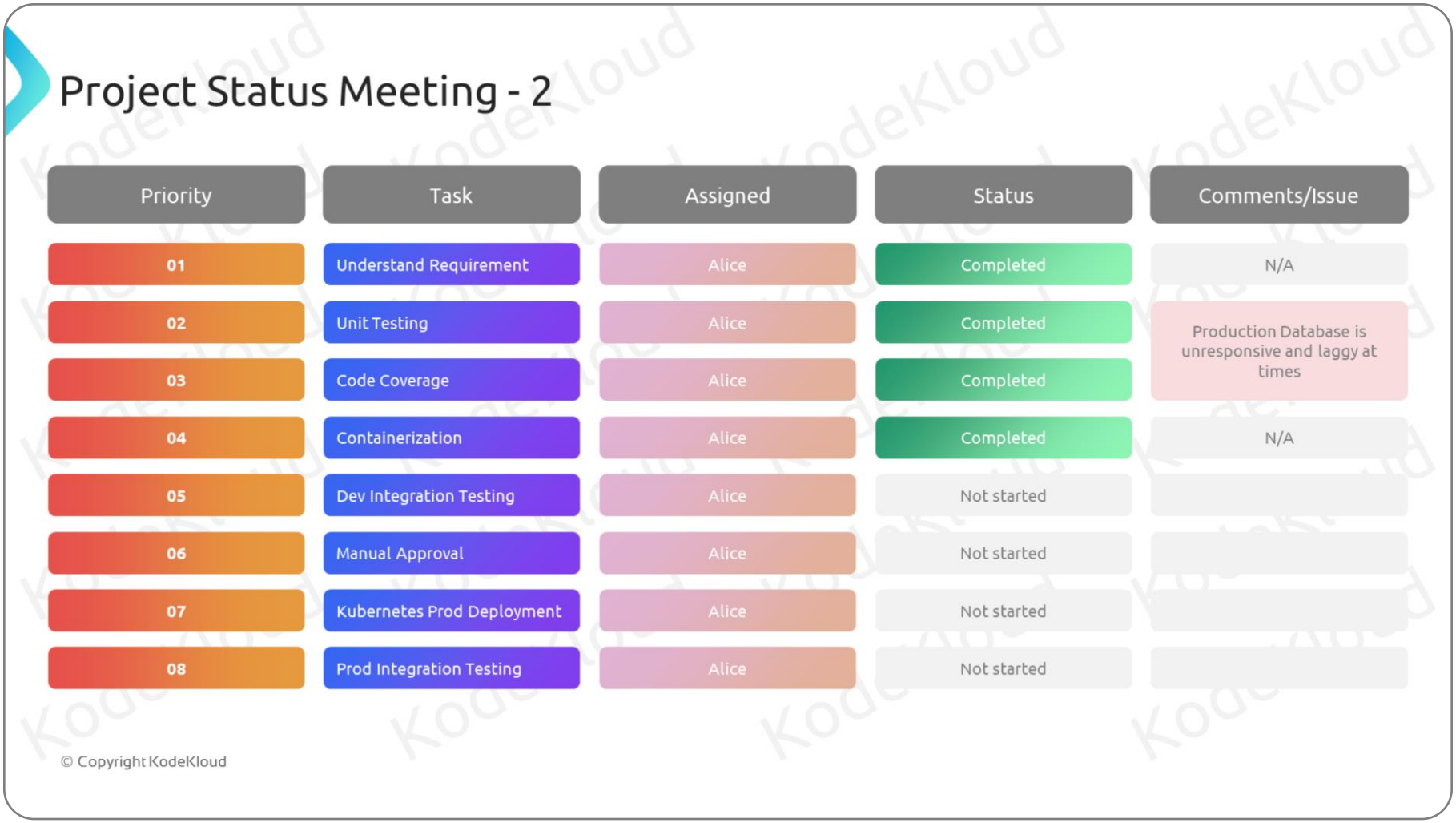
Once the application is started, we can access it on the browser.

Understanding DevOps Pipeline

Understanding DevOps Pipeline



Project Status Meeting - 2



After successfully completing the first four tasks, Alice's satisfaction was short-lived as she was called to an urgent meeting.

During the meeting, she learned that the production database had encountered problems, becoming periodically unresponsive and sluggish since her team began developing the GitHub Actions workflows. She was tasked with investigating the issue and making necessary refactoring to the workflow.

Project Status Meeting - 2

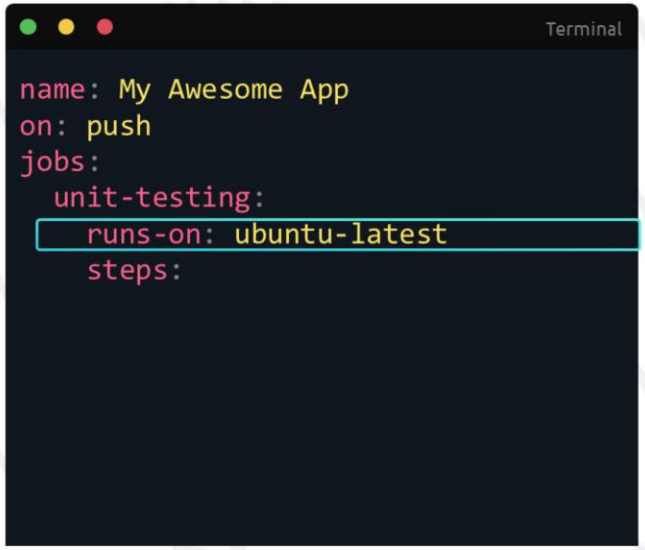
Priority	Task	Assigned	Status	Comments/Issue
01	Understand Requirement	Alice	Completed	N/A
02	Unit Testing	Alice	In progress	Production Database is unresponsive and laggy at times
03	Code Coverage	Alice	In progress	
04	Containerization	Alice	Completed	N/A
05	Dev Integration Testing	Alice	Not started	
06	Manual Approval	Alice	Not started	
07	Kubernetes Prod Deployment	Alice	Not started	
08	Prod Integration Testing	Alice	Not started	

After successfully completing the first four tasks, Alice's satisfaction was short-lived as she was called to an urgent meeting.

During the meeting, she learned that the production database had encountered problems, becoming periodically unresponsive and sluggish since her team began developing the GitHub Actions workflows. She was tasked with investigating the issue and making necessary refactoring to the workflow.

GitLab CI/CD – Images and Services

GitLab Images



Language/ Runtime

- Node.js 18.17.1
- Perl 5.34.0
- Python 3.10.12
- Ruby 3.0.2p107

Package Management

- Miniconda 23.5.2
- Npm 9.6.7
- NuGet 6.6.1.2
- Pip 22.0.2

Tools

- Docker Client 24.0.5
- Terraform 1.5.6
- yamllint 1.32.0
- kubectl 1.28.1

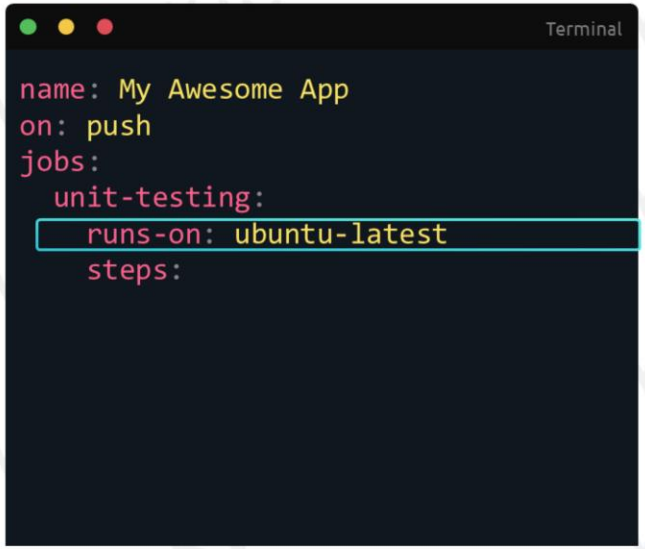
When creating a workflow, we are given the choice to select a runner for each job. Often, we opt for GitHub's hosted virtual machines to execute our workflows. These virtual machines are equipped with a preconfigured environment that includes various tools, packages, and settings designed for GitHub Actions.

For instance, the "ubuntu-latest" operating system on these virtual machines comes with preinstalled software, containing language runtimes, package management tools, command-line utilities, web browsers, cached Docker images,

and much more. Within a job, we can directly leverage these preinstalled software components to carry out our steps. However, there are instances when we need a different version of a runtime or a package that isn't readily available in the preinstalled environment.

In such cases, we can compose additional steps in our workflow to install the required runtime versions and packages.

GitLab Images



CLI Tools

- Azure CLI 2.51.0
- AWS CLI 2.13.15
- GitHub CLI 2.33.0
- Vercel CLI 32.1.0

Browsers and Drivers

- Google Chrome
- Microsoft Edge
- Mozilla Firefox
- Gecko Driver

Cached Docker Image

- alpine:3.16
- debian:10
- node:18-alpine
- ubuntu:22.04

When creating a workflow, we are given the choice to select a runner for each job. Often, we opt for GitHub's hosted virtual machines to execute our workflows. These virtual machines are equipped with a preconfigured environment that includes various tools, packages, and settings designed for GitHub Actions.

For instance, the "ubuntu-latest" operating system on these virtual machines comes with preinstalled software, containing language runtimes, package management tools, command-line utilities, web browsers, cached Docker images,

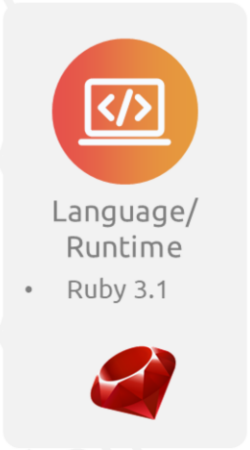
and much more. Within a job, we can directly leverage these preinstalled software components to carry out our steps. However, there are instances when we need a different version of a runtime or a package that isn't readily available in the preinstalled environment.

In such cases, we can compose additional steps in our workflow to install the required runtime versions and packages.

GitLab Images

Terminal

```
workflow:
  name: My Awesome App Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'

unit-testing-job:
  tags:
    - saas-linux-small-amd64
```



When creating a pipeline, we are given the choice to select a runner for each job. When we opt for Gitlab-hosted Linux-based SaaS runner, it defaults to the ruby:3.1 container. This provides a pre-configured environment for Ruby projects out of the box. This reduces complexity for teams primarily working with Ruby, it simplifies managing runner configurations and focuses on writing code.

But what about non-Ruby applications/projects? Accidentally triggering a pipeline in this default container might lead to

confusion and unexpected errors due to the presence of Ruby-specific tools.

In such cases, we can compose additional steps in our job scripts to install the required runtime versions and packages.

GitLab Images

```
workflow:
  name: My Awesome App Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'

unit-testing-job:
  tags:
    - saas-linux-small-amd64
  before_script:
  - apt-get upgrade
  - curl https://node.com/node_20 | bash -
  - apt-get install nodejs -y
  script:
    - npm install
    - npm test
```

In the given example, we are installing Node.js runtime version 20 using before_script for our unit testing process.

When we trigger the pipeline, it will provision a GitLab-hosted runner running the Linux operating system. It will first prepare and start a ruby container, and all the steps are executed within this container.

It's worth noting that, in certain situations, the time it takes to install specific runtimes and packages can significantly

increase the GitLab CI/CD billing costs. To mitigate this, one effective strategy is to use image containers.

GitLab Images

```
Terminal

workflow:
  name: My Awesome App Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'

unit-testing-job:
  tags:
    - saas-linux-small-amd64




  script:
    - npm install
    - npm test
```

In GitLab CI/CD, image containers play a crucial role in defining the environment your job runs in. Image containers are like pre-built sandboxes for your code to run in. Each job gets its own isolated container with the specific tools and environment it needs, like Python libraries for a web app or Node.js for a backend service. These containers provide:
Isolation with co clashing tools or conflicting dependencies between jobs.
Reproducibility where Jobs always run the same way, regardless of the runner machine.
Control on choosing the exact tools and versions for each job for optimal efficiency and security.

To utilize an image container, you must specify the Docker image to be used within the job by configuring the “images" keyword. Additionally, you have the flexibility to define other configurations, including pull-policies, entrypoint, and others.

For instance, this job uses image keyword to execute a container that includes Node.js runtime version 20. By utilizing this container, the need to install the Node.js runtime separately is eliminated.

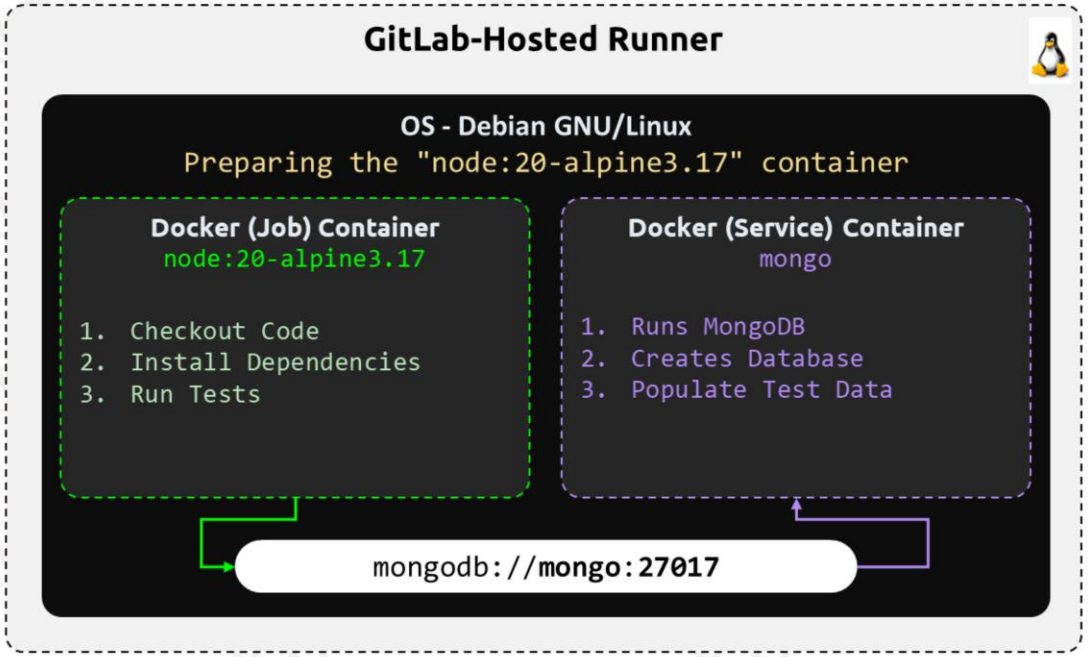
During the execution of the workflow, Gitlab provisions a hosted runner with the Debian/Linux operating system. Within this OS, it creates and runs the Nodejs Docker container, executing all the defined steps within that container.

Lets assume that in this particular job, unit tests are executed while connected to the production database. This practice is not advisable, as it can potentially impact the production system's performance. For tasks like unit testing or code coverage analysis, it is recommended to use dedicated testing databases or services.

Gitlab cicd offers an other solution to address this issue as well.

GitLab Services

```
Terminal
workflow:
  name: My Awesome App Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'

unit-testing-job:
  tags:
    - saas-linux-small-amd64
  image: node:20-alpine3.17




  script:
    - npm install
    - npm test
```



Much like Job image containers, Service containers within GitLab CI/CD are Docker containers that provide a simple and portable way for you to host services. They provide specific resources and capabilities that your jobs might need during execution. Think of them as specialized tools on your workbench, such as:

Providing resources such as a database for testing? A service can spin up a temporary MySQL instance for your job.
Boost capabilities: by offering Docker in Docker (DinD) services.

Increase flexibility by creating custom Services allowing you to tailor them to your specific needs.
Finally, like job containers, services run in their own environment, preventing interference with your main job.

Service images are specified using the services keyword. This additional image is used to create another container, which is available to the first container. The two containers have access to one another and can communicate when running the job.

This pipeline configuration specifies a job that runs all steps within a Node.js-based image container. It utilizes the Debian GNU/Linux" GitLab-hosted runner as the Docker host for this container. Additionally, the job is configured with a service container with a custom mongodb docker image and an alias mongo

When the pipeline is executed, it will run 2 containers,
One is a mongodb service container, which runs mongodb and populates test data,
and
Within the job container, it carries out the following steps:

Checks out the repository on the runner.
Installs dependencies.
Executes unit tests by connecting to the MongoDB service container.

But how does the Unit testing step executing in Job container connect with Mongodb in service container?

Given that job and service Docker containers run within the same user-defined bridge network, all ports are exposed to each other, you'll have seamless access to the service container via the default MongoDB port, which is 27017.

The hostname of the MongoDB service corresponds to the alias configured in the service keyword, which in this case is “mongo"

Now that we have an idea about Job and Service containers, let’s implement them in our workflow.

Project Status Meeting - 3

Project Status Meeting - 3

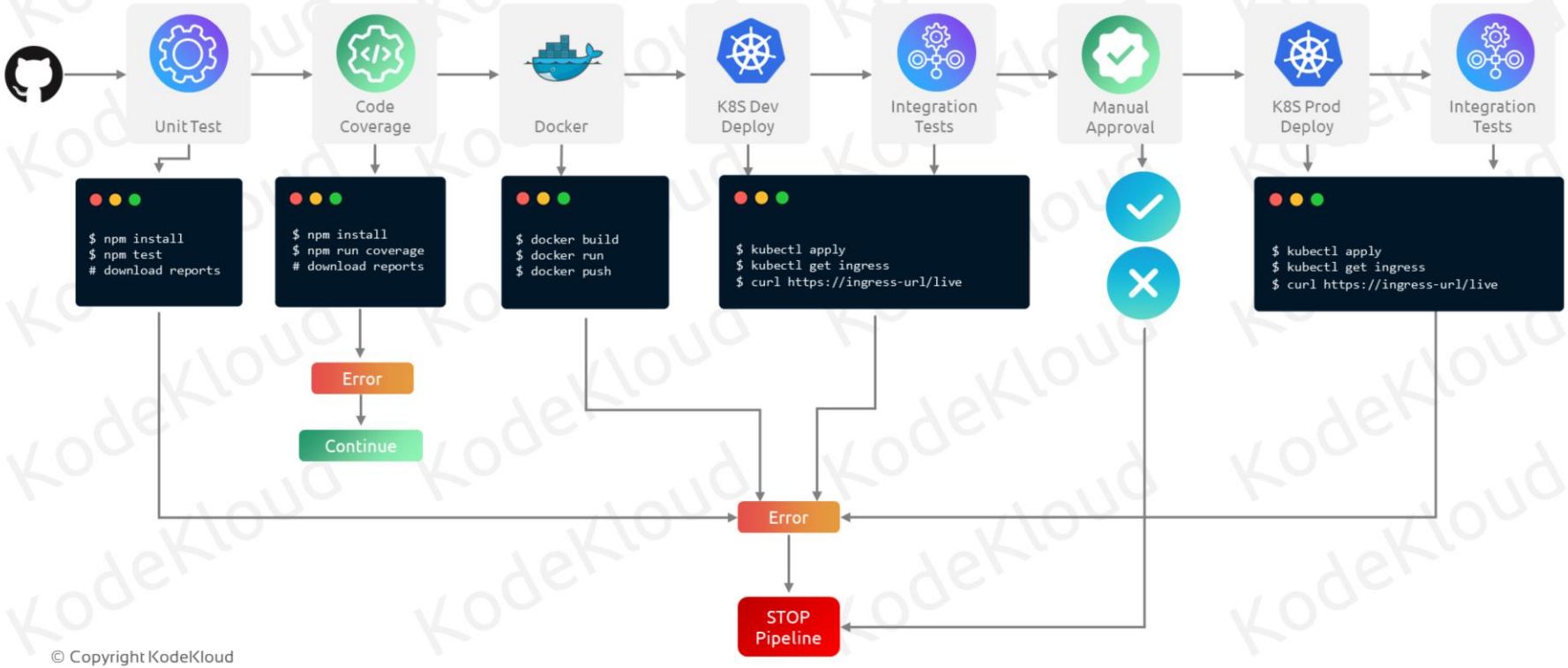
Priority	Task	Assigned	Status	Comments/Issue
01	Understand Requirement	Alice	Completed	N/A
02	Unit Testing	Alice	Completed	N/A
03	Code Coverage	Alice	Completed	
04	Containerization	Alice	Completed	N/A
05	Dev Integration Testing	Alice	In progress	
06	Manual Approval	Alice	In progress	
07	Kubernetes Prod Deployment	Alice	Not started	
08	Prod Integration Testing	Alice	Not started	

By refactoring the workflow with job and service containers, the load on the production database has been reduced, allowing the jobs to be successfully marked as completed.

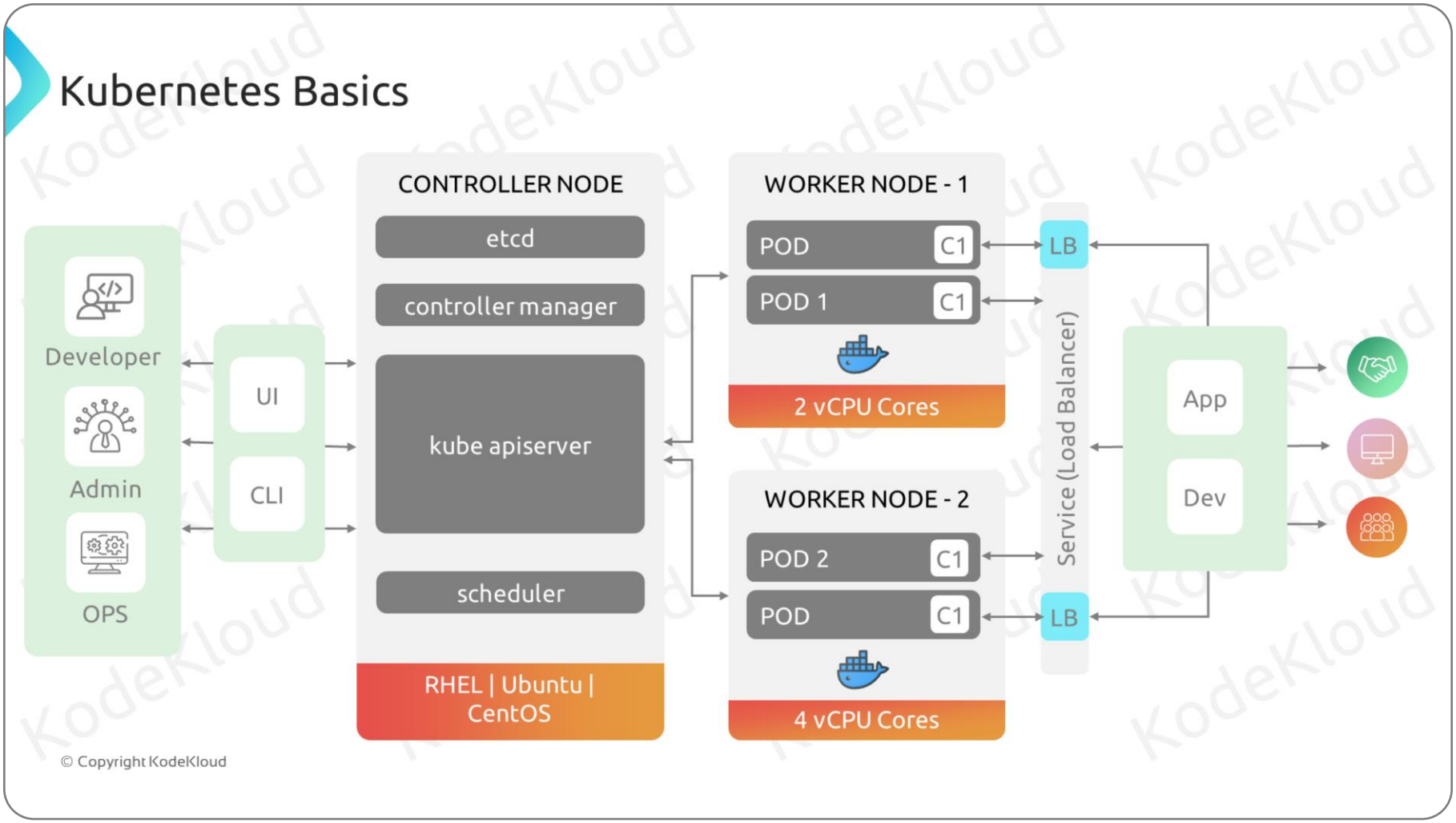
Alice and her team are now preparing for the deployment phase. Before proceeding, let's examine the deployment requirements and look into the fundamentals of Kubernetes.

Understanding Deployment Pipeline

Understanding Deployment Pipeline



Kubernetes – Brief Overview



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.

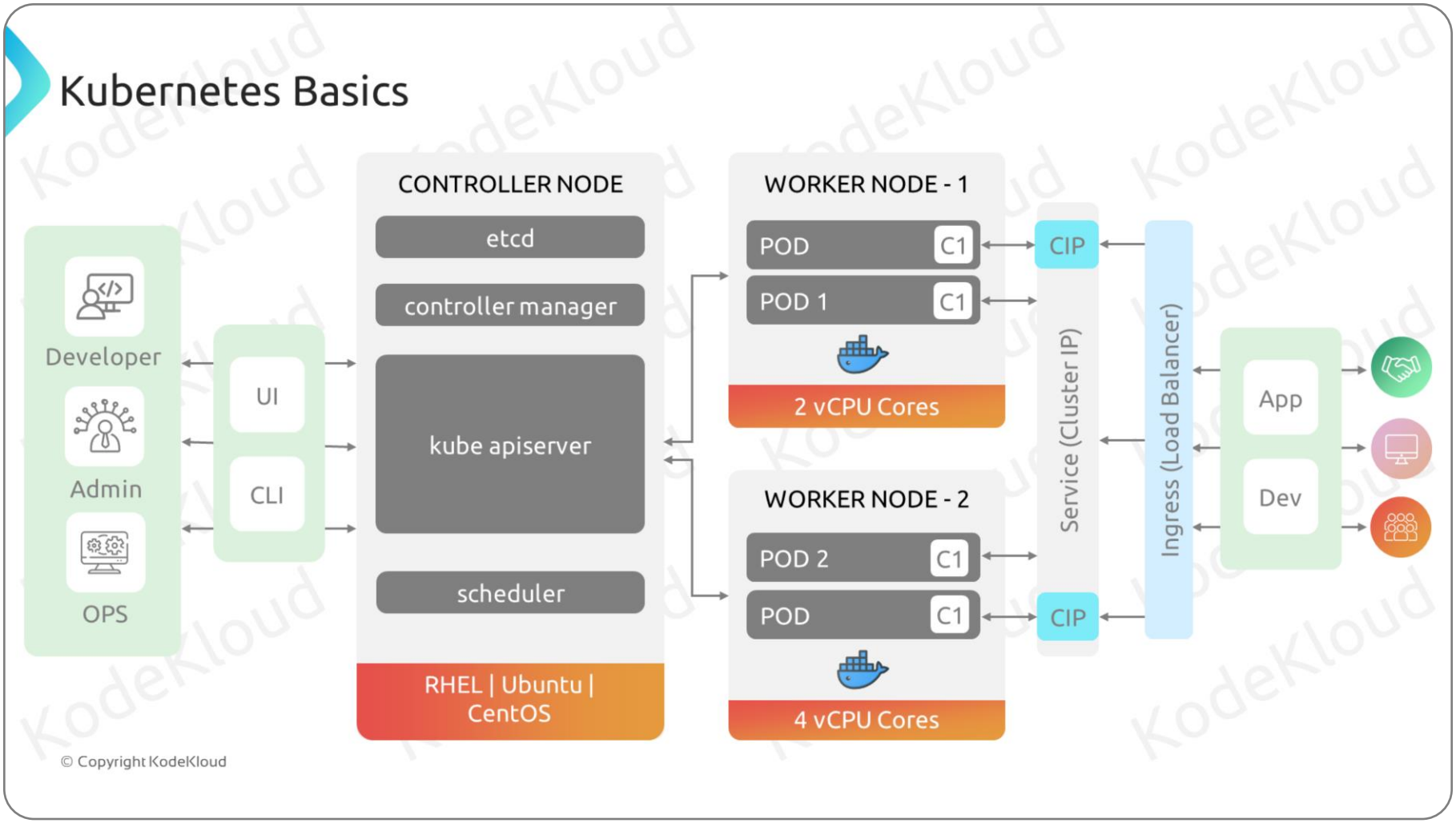
Services in Kubernetes provide a stable and consistent network endpoint to access one or more pods. They can load balance traffic among multiple pods, allowing applications to be easily scaled without affecting external clients.

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.

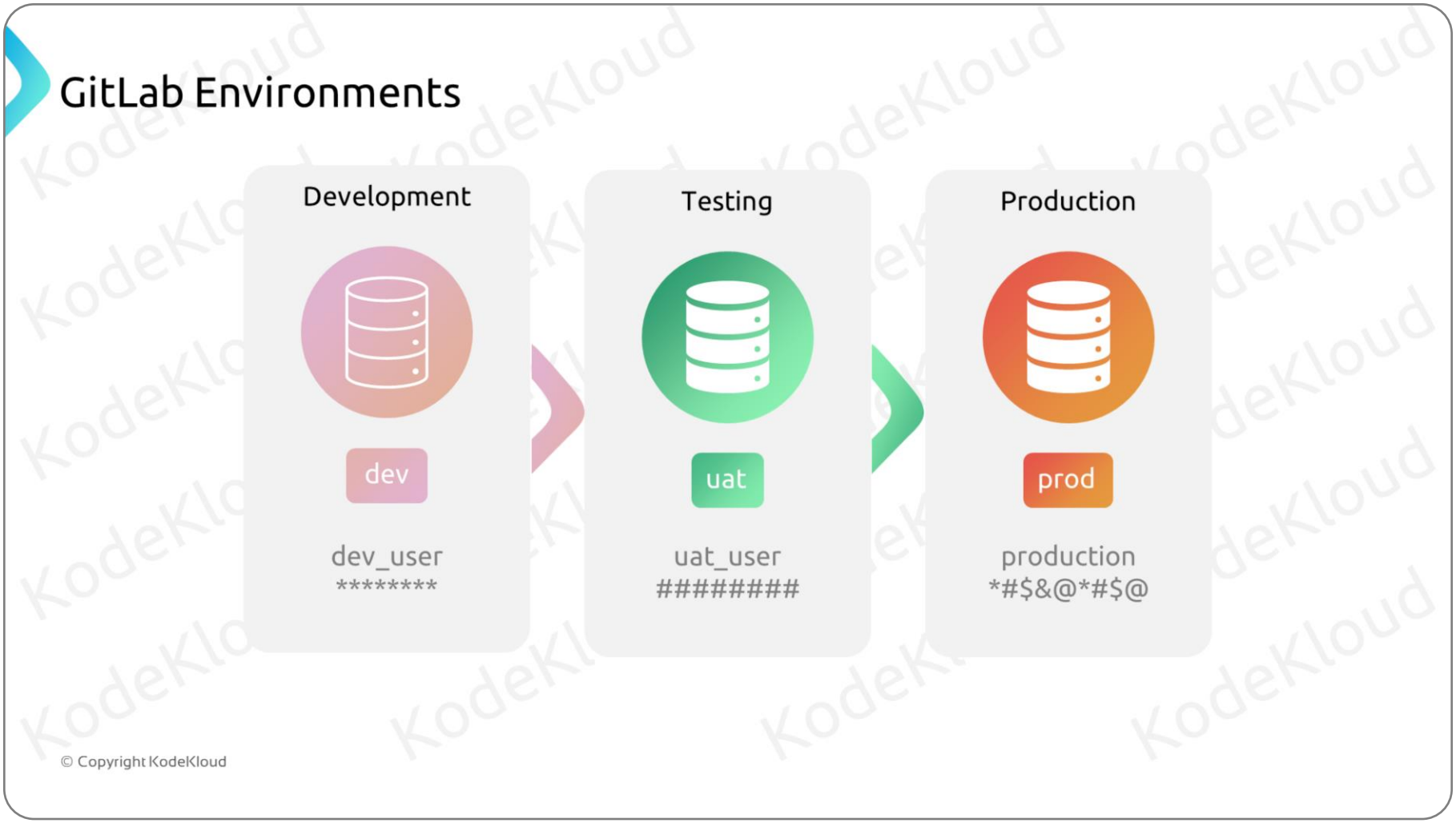
It's suitable for exposing multiple services under a single domain or for setting up more complex routing rules. You can route traffic based on paths, domain names, and other HTTP request attributes.

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.

Understanding GitLab Environments

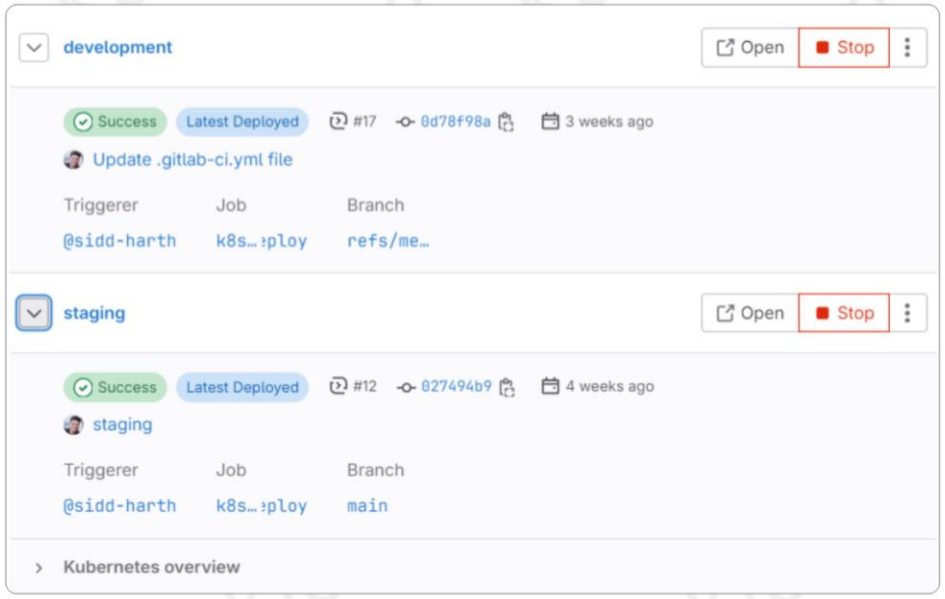


In software, environments are used to isolate different stages of the development process, such as development, testing, and production. This allows developers to work on new features without affecting existing users, and it allows testers to verify that new features work correctly before they are deployed to production.

Each environment has it's now services such as Databases, vaults, rest apis etcs. Most of these services are secured using username, password credentials or some use apikeys and these varies for each environment.

In GitLab CI/CD, environments are a powerful feature that can help you to organize, protect, and visualize your deployments.

GitLab Environments



What are deployments?
Each time GitLab CI/CD deploys a version of code to an environment, a deployment is created.

GitLab Provides a full history of deployments to each environment.
You can track the history of deployments to each environment, making it easy to understand what's currently deployed, re-deploy, and roll back if needed.

Rollback is a feature which can revert to a previous deployment in case of issues, preserving stability and minimizing downtime.

Think of environments as labels: They represent stages where your code gets deployed, such as development, staging, or production.
They don't hold the actual code: They manage the deployment process.
They act as reference points: CI/CD pipelines target specific environments for deployment, and GitLab tracks the history of these deployments.

GitLab Environments

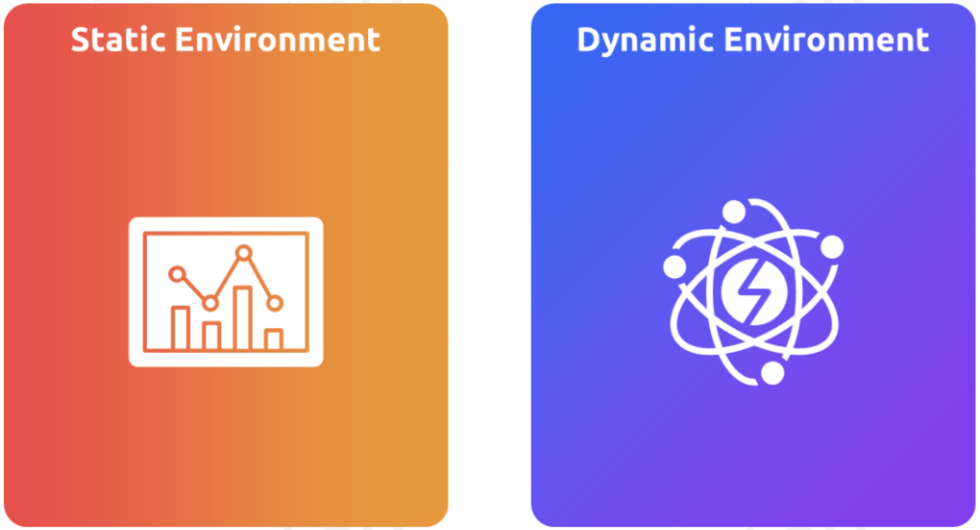
What are deployments?
Each time GitLab CI/CD deploys a version of code to an environment, a deployment is created.

GitLab Provides a full history of deployments to each environment.
You can track the history of deployments to each environment, making it easy to understand what's currently deployed, re-deploy, and roll back if needed.

Rollback is a feature which can revert to a previous deployment in case of issues, preserving stability and minimizing downtime.

Think of environments as labels: They represent stages where your code gets deployed, such as development, staging, or production.
They don't hold the actual code: They manage the deployment process.
They act as reference points: CI/CD pipelines target specific environments for deployment, and GitLab tracks the history of these deployments.

Types of Environment



There are two main types of GitLab environments:
Static environments: These are pre-defined and configured manually in the GitLab UI or .gitlab-ci.yml file. They are ideal for simple deployments with fixed configurations.
Dynamic environments: These are created automatically for each pipeline run with their own unique URLs and names based on CI/CD variables. They are useful for situations with numerous deployments or where environment configurations vary significantly.

Types of Environment

Additional environment types:
Review apps: These are temporary environments created from merge requests for reviewers to test changes before merging.
Protected environments: These environments require additional approvals before deployments, adding an extra layer of security for critical deployments.

Environment Scope for CI/CD Variables

CI/CD Variables </> 8 Reveal Values Add Variable

↑ Key	Value	Environments	Actions
KUBE_CONTEXT	*****	All (default)	
KUBE_INGRESS_BASE_DOMAIN	*****	All (default)	
KUBE_NAMESPACE	*****	Staging	
KUBE_NAMESPACE	*****	Production	
MONGO_PASSWORD	*****	Production	
MONGO_URI	*****	All (default)	

By default, every job in a pipeline has access to all CI/CD variables. If a test job within a pipeline runs a compromised tool, it has the potential to expose all the CI/CD variables utilized by a deployment job. This situation is often encountered in supply chain attacks. GitLab addresses this concern by restricting the environment scope of a variable.

One of the powerful features of GitLab CI/CD environments is the ability to control the scope of CI/CD variables. This adds a layer of security and flexibility to your deployments by ensuring sensitive information is only accessible where

needed. Here's a detailed breakdown of Environment Scope:

What is it?

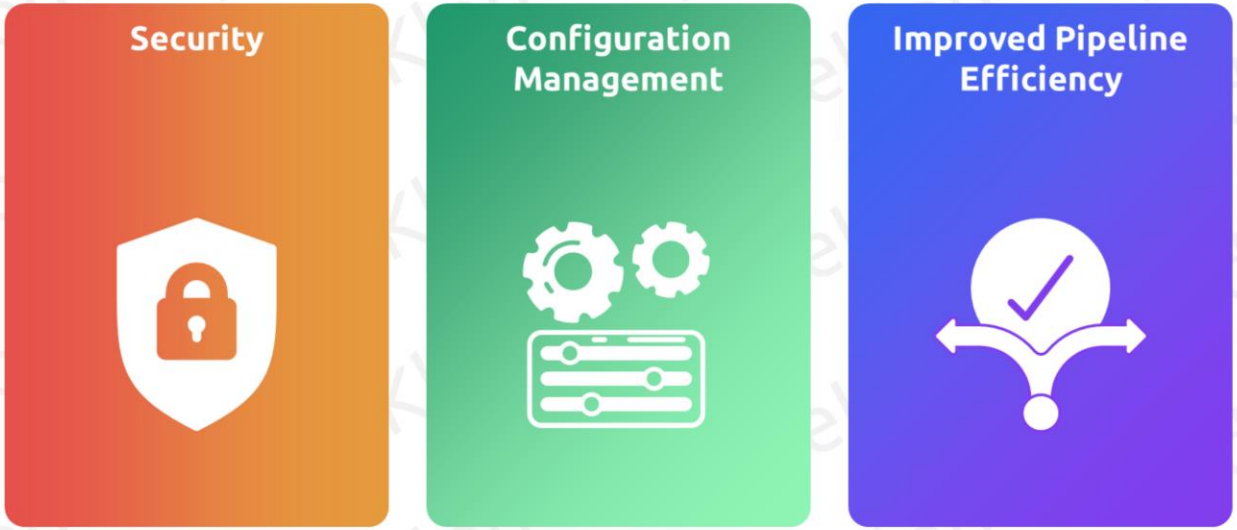
Environment scope allows you to restrict specific CI/CD variables to one or a limited set of environments. By default, all variables defined in your project or group level are accessible to all jobs in all environments. With environment scope, you can explicitly specify which environments can access each variable.

Why use it?

There are several compelling reasons to use environment scope:
Security: Limit access to sensitive information like database credentials, API keys, or production passwords to only the environments where they are needed. This minimizes the risk of accidental exposure if other environments are compromised.
Configuration management: Different environments might require different configurations or settings. By scoping variables to specific environments, you can ensure only the relevant configurations are applied and avoid conflicts or errors.
Improved pipeline efficiency: Scoping variables reduces the overall footprint of your pipelines, making them lighter and faster to run. It also prevents unnecessary exposure of sensitive information in logs or outputs of jobs that wouldn't need them.

In the provided example,
We are 2 variables with the same key name as Kube_namespace, but its value differs based on the environment scope.
Similarly, the MONGO_PASSWORD key/value can only be accessed by jobs which are configured to deploy to production environment.

Environment Scope for CI/CD Variables



Why use it?

There are several compelling reasons to use environment scope:
Security: Limit access to sensitive information like database credentials, API keys, or production passwords to only the environments where they are needed. This minimizes the risk of accidental exposure if other environments are compromised.

Configuration management: Different environments might require different configurations or settings. By scoping variables to specific environments, you can ensure only the relevant configurations are applied and avoid conflicts or errors.
Improved pipeline efficiency: Scoping variables reduces the overall footprint of your pipelines, making them lighter and faster to run. It also prevents unnecessary exposure of sensitive information in logs or outputs of jobs that wouldn't need them.

In the provided example,
We are 2 variables with the same key name as Kube_namespace, but its value differs based on the environment scope
Similarly the MONGO_PASSWORD key/value can only be accessed by jobs which are configured to deploy to production environment.

Environment Scope for CI/CD Variables

CI/CD Variables </> 8 — Reveal Values | Add Variable

↑ Key	Value	Environments	Actions
KUBE_CONTEXT	*****	All (default)	
KUBE_INGRESS_BASE_DOMAIN	*****	All (default)	
KUBE_NAMESPACE	*****	Staging	
KUBE_NAMESPACE	*****	Production	
MONGO_PASSWORD	*****	Production	
MONGO_URI	*****	All (default)	

In the provided example,
We are 2 variables with the same key name as Kube_namespace, but its value differs based on the environment scope. Similarly, the MONGO_PASSWORD key/value can only be accessed by jobs which are configured to deploy to production environment.

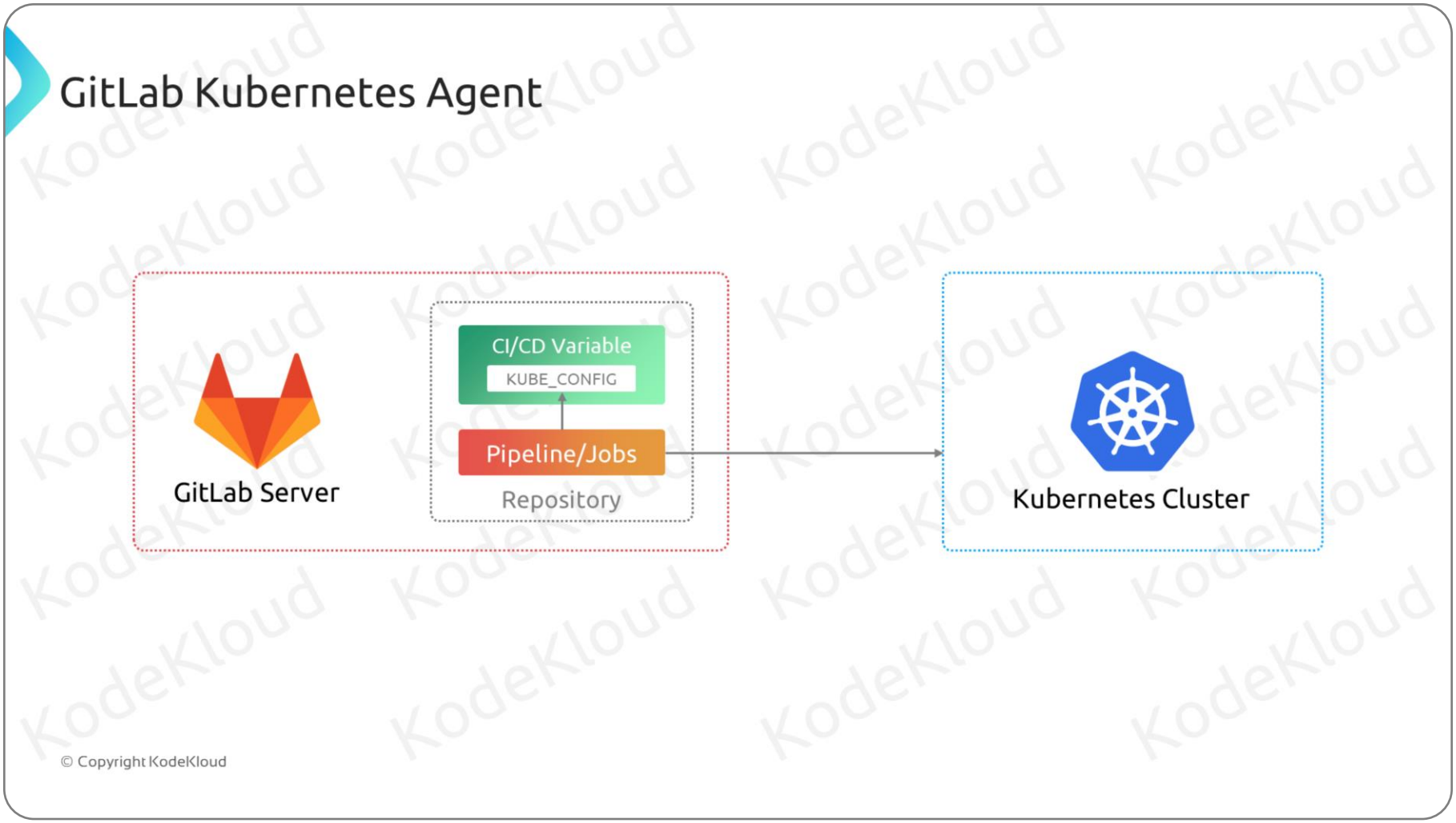
https://gitlab.com/gitlab-org/cluster-integration/gitlab-agent/-/blob/master/doc/architecture.md

With the rise of DevOps and SRE approaches, infrastructure management becomes codified and automatable, and software development best practices gain their place around infrastructure management too.

GitLab offers various features to speed up and simplify your infrastructure management practices.

Issues the agent is trying to address

The GitLab integration with Kubernetes helps you to install, configure, manage, deploy, and troubleshoot cluster applications.
With the GitLab agent, you can connect clusters behind a firewall or NAT, have real-time access to API endpoints, perform pull-based or push-based deployments for production and non-production environments, and much more.



In our previous demo, we deployed an application to Kubernetes by establishing a connection to the cluster through the KUBECONFIG file stored as a variable.

Your kubeconfig file contains sensitive information like the URL of your Kubernetes API server, authentication credentials, and the default namespace you use. If this information falls into the wrong hands, they could gain unauthorized access to your Kubernetes cluster and perform malicious actions, such as deploying malware, deleting

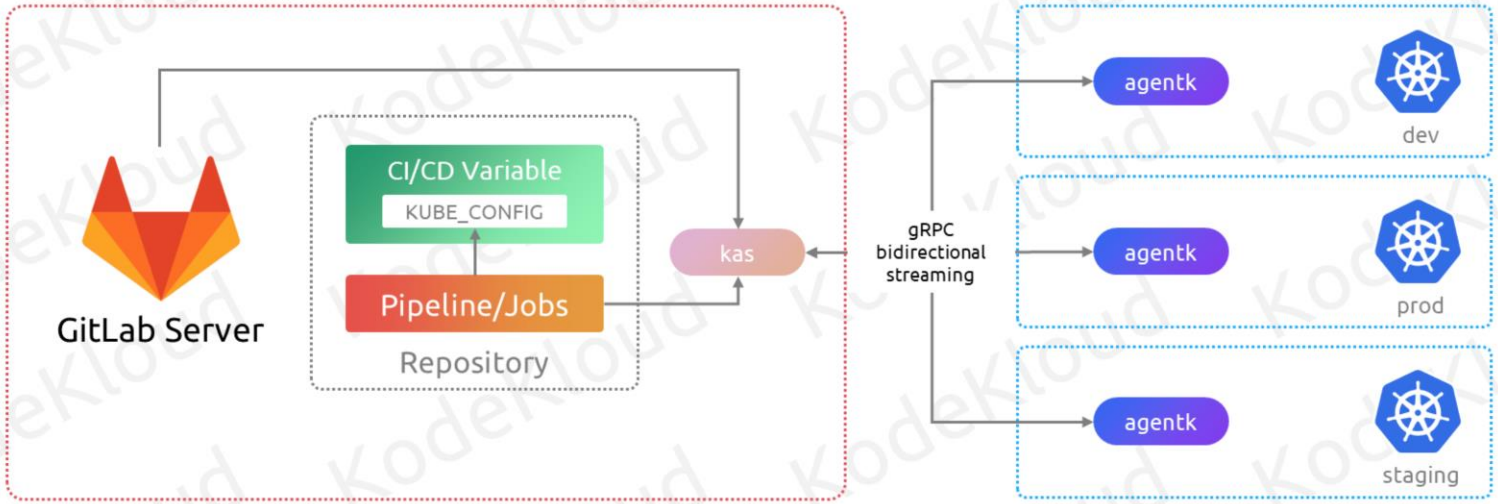
resources, or stealing data.

Despite storing the KUBECONFIG file as a masked CI/CD variable, it still poses a security concern.

Beyond security considerations, there is a potential for increased complexity when establishing a connection to the Kubernetes cluster, especially if the cluster is situated behind a firewall or Network Address Translation (NAT)

To address these issues, let's explore the GitLab Agent for Kubernetes.

GitLab Kubernetes Agent



The GitLab Kubernetes Agent is a key element within GitLab's suite of tools, designed to enable various functionalities like GitOps deployments, CI/CD integration, and remote development.

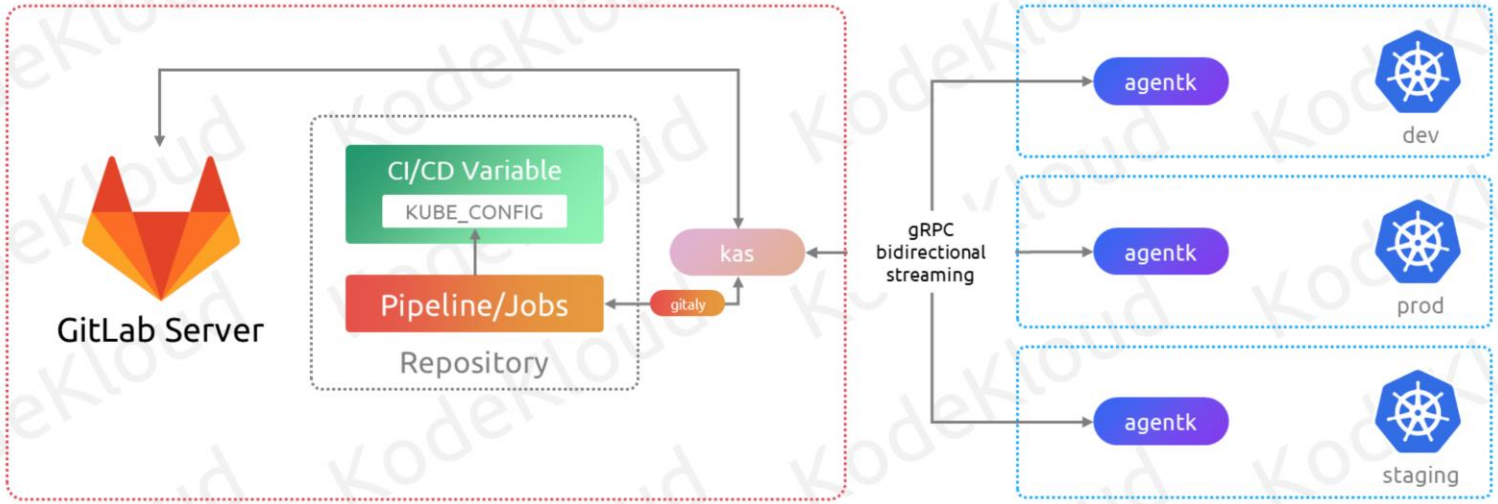
Indeed, as mentioned earlier, GitLab facilitates GitOps through its integration with Fluxcd. Note that this course will not explore GitLab's GitOps capabilities, but with GitOps, you can manage containerized clusters and applications from a GitLab repository that acts as the single source of truth of your system.

While GitLab Kubernetes Agent’s primary role involves GitOps implementation, its functionality extends far beyond this particular use case. It provides a means of secure communication between GitLab and Kubernetes clusters, without the need for exposing the Kubernetes API, and storing credentials in third-party systems.

The Agent is implemented in a client-server configuration; the Kubernetes cluster-side component is called agentk and the GitLab Server-side component is called KAS (GitLab Agent Server). Communication between these agent components is initiated by agentk because the connection must be initiated from inside the Kubernetes cluster to bypass any firewall or NAT that the cluster may be located behind. As the channel is bidirectional in nature, in the context of GitOps, this enables either push- or pull-based operation.

GitLab's SaaS version takes care of managing kas, eliminating the need for users to upgrade agentk in their clusters whenever there are updates or additional features. GitLab prefers implementing modifications and logic directly to kas instead of agentk. Given the lightweight nature of agentk, GitLab suggests having at least one instance of agentk per cluster.

GitLab Kubernetes Agent



The GitLab Kubernetes Agent is a key element within GitLab's suite of tools, designed to facilitate GitOps-style deployments.

Indeed, as mentioned earlier, GitLab facilitates GitOps through its integration with Fluxcd. Note that this course will not explore GitLab's GitOps capabilities, but with GitOps, you can manage containerized clusters and applications from a GitLab repository that acts as the single source of truth of your system.

While GitLab Kubernetes Agent's primary role involves GitOps implementation, its functionality extends far beyond this particular use case. It provides a means of secure communication between GitLab and Kubernetes clusters, without the need for exposing the Kubernetes API, and storing credentials in third-party systems.

The Agent is implemented in a client-server configuration; the Kubernetes cluster-side component is called agentk and the GitLab Server-side component is called KAS (GitLab Agent Server). Communication between these agent components is initiated by agentk because the connection must be initiated from inside the Kubernetes cluster to bypass any firewall or NAT that the cluster may be located behind. As the channel is bidirectional in nature, in the context of GitOps, this enables either push- or pull-based operation.

GitLab is the main application which talk to kas.
agentk is the GitLab Agent. It keeps a connection established to kas instance, waiting for requests to process. It may also actively send information about things happening in the cluster.
kas is responsible for:

- Accepting requests from agentk.
- Authentication of requests from agentk by querying GitLab.
- Fetching agent's configuration from a corresponding Git repository by querying Gitaly.
    - What is Gitaly?
        - Gitaly provides high-level RPC access to Git repositories. used by GitLab to read and write Git data.

This is the method through which agentk and kas establish communication, forwarding requests, responses, sending notifications, and handling other interactions between them.

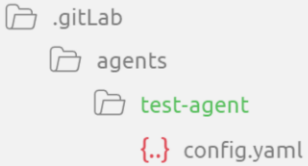
GitLab's SaaS version takes care of managing kas, eliminating the need for users to upgrade agentk in their clusters whenever there are updates or additional features. GitLab prefers implementing modifications and logic directly to kas instead of agentk. Given the lightweight nature of agentk, GitLab suggests having at least

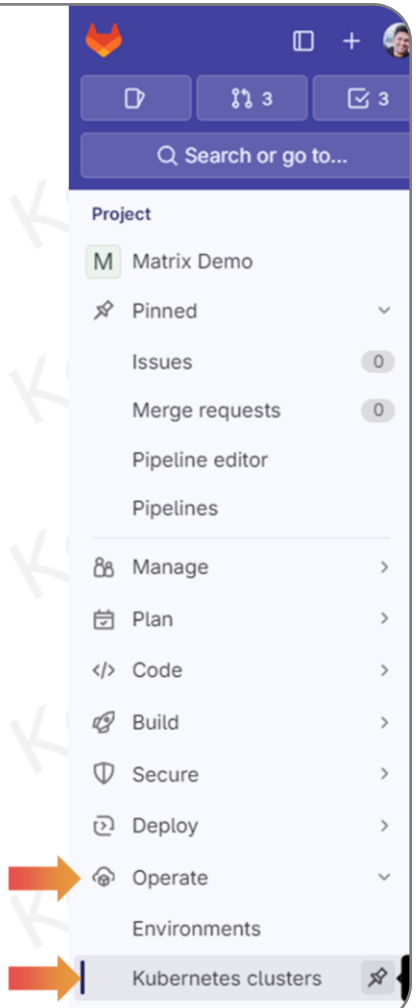
one instance of agentk per cluster.

GitLab Agent Registration

```
helm repo add gitlab https://charts.gitlab.io
helm repo update
helm upgrade --install test-agent gitlab/gitlab-agent \
    --namespace gitlab-agent-test-agent \
    --create-namespace \
    --set image.tag=v16.9.0-rc2 \
    --set config.token=glagent-token \
    --set config.kasAddress=wss://kas.gitlab.com
```

.gitlab/agents/<name>/config.yaml





Registering a GitLab Kubernetes agent connects your Kubernetes cluster to GitLab.

For Prerequisites:
We need to have a
GitLab Account with sufficient permissions to create and manage projects and Kubernetes clusters.
GitLab Project where you want to register the agent. This project will house the agent configuration file and manage

deployments.
Kubernetes Cluster: which meets minimum version requirements for GitLab Agent compatibility.

To register, navigate to the GitLab project, click on "Operate," and then choose "Kubernetes clusters." Create a new agent by specifying a unique name. GitLab automatically generates a unique and secure token for the agent, along with the Helm installation commands. The helm chart uses the generated token and the KAS address to enable the registration and communication process. Run the Helm chart command on your Kubernetes cluster to complete the installation.

After successful installation, the agent connects to GitLab and becomes visible in the Kubernetes clusters section of your project.

We can further manage the GitLab Agent in your project using a config file located at .gitlab/agents/<name>/config.yaml in your project repository.

GitLab Agent Registration

gitLab

```yaml
# GitOps configuration
gitops:
  # Manifest projects to monitor and deploy
  manifest_projects:
    - id: gitlab-org/my-project
      default_namespace: my-namespace # Default namespace

# CI/CD access control
ci_access:
  projects:
    # Project allowed to use this agent for deployments
    - id: gitlab-org/my-project

# Remote development configuration (GitLab Enterprise Edition)
workspaces:
  # Enable workspaces
  enabled: true
  # ... other workspace-specific settings
```


We can further manage the GitLab Agent in your project using a config file located at .gitlab/agents/<name>/config.yaml in your project repository

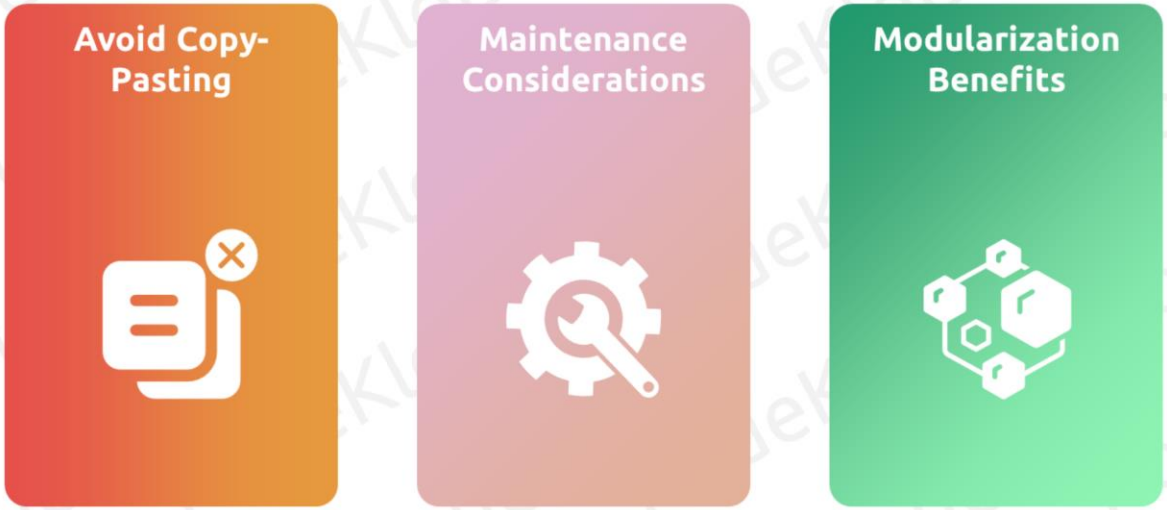
This file allows you to:

- Specify GitOps settings which Defines how the agent monitors changes and triggers deployments for automated workflows.
- Set CI/CD access control to Limit which projects and users can use the agent for deployments.

Configure remote development (GitLab Enterprise Edition): Define settings for creating and managing workspaces within your cluster.

Optimize CI/CD Configuration

Optimizing CI Pipeline Development: Modularization Strategies

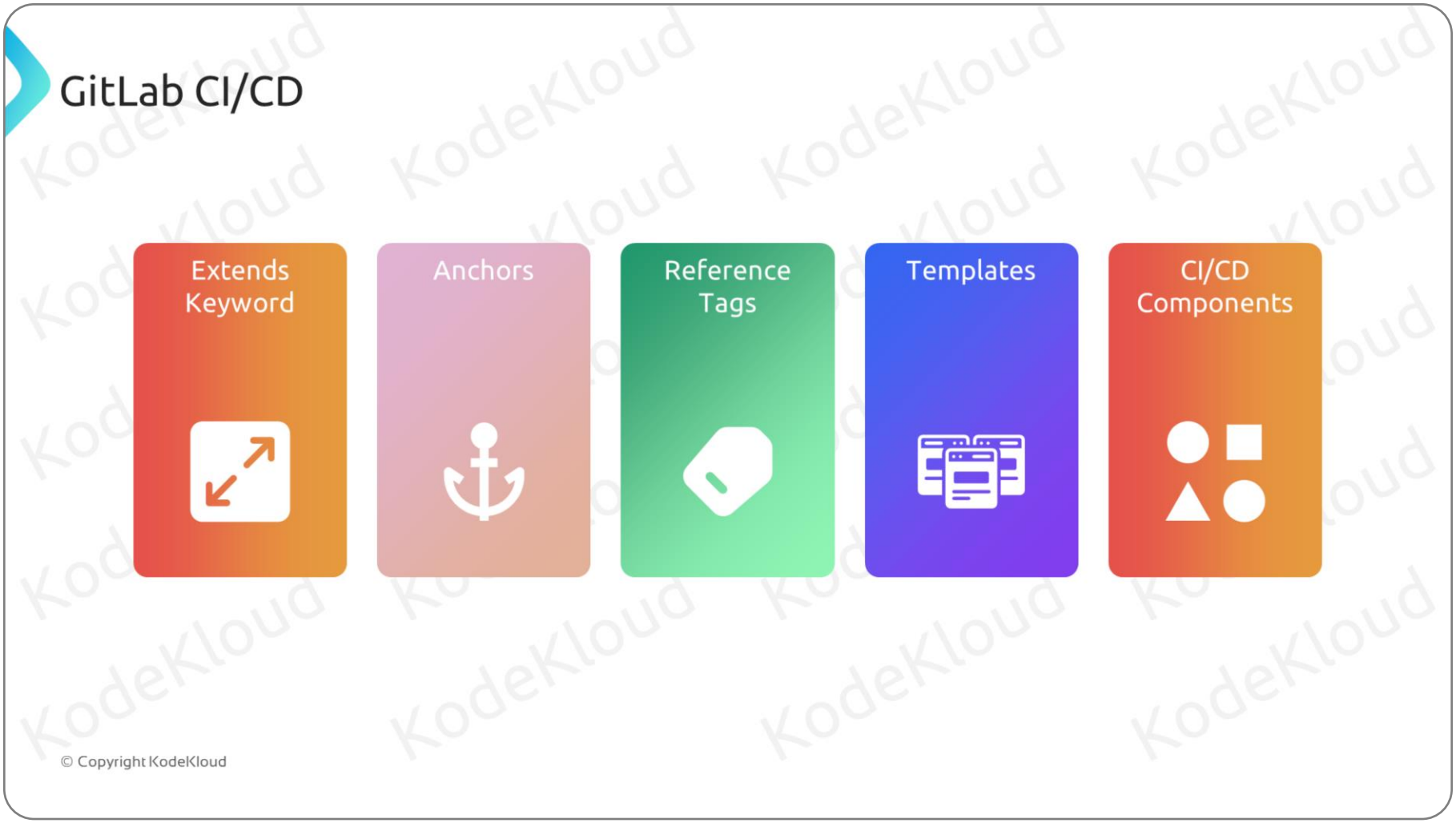


When creating an application in any programming language or setting up an automated pipeline through a CI tool, you've likely encountered situations where you end up copying and pasting portions of code or pipelines from previous work. While copying CI pipelines can accelerate initial project setup, relying heavily on duplication can hinder effective maintenance and scalability.

We should always consider modularizing your CI pipelines to avoid copy-pasting and facilitate easier updates and adjustments as the pipeline evolves.

GitLab CI/CD

GitLab CI's got your back – say goodbye to duplicated code! GitLab CI has multiple features to standardize workflows, boost development speed, and make implementing pipelines a breeze.



To optimize your CI/CD pipelines by making them more modular, reusable, Gitlab provides

- Extends Keyword
- Anchors
- Reference Tags
- Templates and
- CI/CD Components.

In this video, we will talk about the first 3 topics, but before that you need to know about hidden jobs.

Hidden Jobs

GitLab

```
.hidden_job
  cache:
    policy: pull-push
    key:
      files:[package.json]
    paths: [node_modules]
  before_script:
    - npm install
```

GitLab CI supports hidden jobs.

Jobs starting with a dot (".") are not executed by the pipeline.

Hidden jobs can be used to disable or for reusable configuration without deletion.

Remember, hidden jobs are invisible to GitLab CI/CD for direct execution. They act as reusable templates and configuration blocks, improving your pipeline's modularity and maintainability.

Extends Keyword

GitLab

```yaml
unit_testing:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script:
    - npm test

code_coverage:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script: |
    npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

GitLab

```yaml
# This is a hidden job
.base_nodejs_job:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
```


Let's take an example and understand how the extends keyword can be used.

Imagine a pipeline with two Node.js jobs for unit testing and code coverage. They share a lot of setup, like using the same container image, service container, caching strategy, and a before_script for dependency installation.

Although this example involves only two jobs, in other scenarios, there might be many more jobs using a common set of

configuration. How do we modularize them?

The extends keyword in GitLab CI/CD offers a solution: create a hidden job with the shared configuration, and let other jobs inherit it. Here's how it works:

First step is to Create a hidden job named .base_nodejs_job containing the common configuration (image, services, cache, etc.) within the same .gitlab-ci.yml file.
Second step is to Inherit the hidden job: In the unit_testing and code_coverage jobs, use the extends keyword to specify .base_nodejs_job as their parent.
Once it is done, the GitLab CI/CD merges the parent's configuration with the child job's, overriding any duplicates with values from the parent.

For your reference, line numbers 3 to 12 will be merged in both the child jobs.

Extends Keyword

```gitlab
unit_testing:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script:
    - npm test

code_coverage:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script: |
    npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

Reduced duplication

Maintain-ability

Modular

```gitlab
# This is a hidden job
.base_nodejs_job:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install

unit_testing:
  # Using a hidden job
  extends: .base_nodejs_job
  script:
    - npm test

code_coverage:
  # Using a hidden job
  extends: .base_nodejs_job
  script: |
    npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```


This approach offers several benefits:
Reduced code duplication: Eliminate repetitive setup code, making pipelines easier to maintain.
Improved readability: Separate configuration concerns for better clarity.
Enhanced maintainability: Changes to the shared configuration apply to all inheriting jobs automatically.
Modular pipelines: Build reusable templates for common patterns, streamlining pipeline creation.

Extends vs Anchors

Extends keywords

```yaml
# This is a hidden job
.base_nodejs_job:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install

unit_testing:
  # Using a hidden job
  extends: .base_nodejs_job
  script:
    - npm test

code_coverage:
  # Using a hidden job
  extends: .base_nodejs_job
  script: |
    npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

YAML Anchor

```yaml
.base_nodejs_job: &node_config_anchor
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
```

& sets up the name of the anchor

<< merge the given hash into the current one

* includes the named anchor

If you're looking for ways to further optimize your pipeline setup, don't miss out on anchors - they're like extends' mini-me, helping you reuse smaller configurations for maximum efficiency.

YAML specification has a feature called ‘anchors’ that you can use to duplicate content across your document.

Since GitLab CI/CD is written in YAML format, we can use anchors with hidden jobs to share configuration between jobs.

To use a hidden job as an anchor, you can define an anchor using the "&" symbol, followed by a name (in this example, we named it node_config_anchor)
To use the anchor and merge the config into the current job, you need to define “<<“ (a couple of left arrow) symbol and call it by its name with the "*" symbol. Both of these can be seen on lines 14 and 19

That’s it! After these changes, the hidden job will be used by both the jobs.

When comparing the "extends" keyword to anchors, it’s important to note that YAML anchors cannot be used across multiple files; Anchors are only applicable within the file where they are defined. For the purpose of reusing configuration from different YAML files, the recommended approaches are to utilize the "extends" keyword or use "!reference" tags.
While we have discussed the "extends" keyword earlier, let’s dive into the concept of "!reference" tags.

!reference Tags

Normal Pipeline Config

```yaml
unit_testing:
  image: node:17-alpine3.14
  cache:
    paths:
    - node_modules
  before_script:
  - npm install
  script:
  - npm test

code_coverage:
  image: node:17-alpine3.14
  before_script:
  - npm install
  script:
  - npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

!reference tag with hidden & regular job

```yaml
.base_nodejs_config:
  cache:
    paths: [node_modules]
  test_script: [npm test]
  code_coverage_script: [npm run coverage]
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

expanded config

```yaml
.base_nodejs_config:
  cache:
    paths:[node_modules]
  test_script: [npm test]
  code_coverage_script: [npm run coverage]
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual

unit_testing:
  image: node:17-alpine3.14
  cache:
    paths:
    - node_modules
  before_script:
  - npm install
  script:
  - npm test

code_coverage:
  image: node:17-alpine3.14
  before_script:
  - npm install
  script:
  - npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```


Use the !reference custom YAML tag to select keyword configuration from other job sections and reuse it in the current section.
Unlike YAML anchors, you can use !reference tags to reuse configuration from different YAML files.

Imagine a pipeline with two Node.js jobs for unit testing and code coverage. They share a couple of common keywords like image and before_script.

To achieve optimization, we can also create an hidden job with commonly used configuration.

Create a hidden job named .base_nodejs_config containing the shared configuration like cache strategy, rules, and script commands for tests and code coverage.

Now, let's explore how we can reference this configuration from the hidden job within the "unit_testing" job. For the "image" and "before_script" keywords, we directly specify the commands. However, for "cache" and "script" keywords, we utilize the "reference" keyword along with the hidden job name and tag.

Moving on to the "code_coverage" job, the "image" and "before_script" keywords can be directly referenced from the "unit_testing" job.

This highlights the versatility of reference tags, enabling the referencing of keywords from both hidden and regular jobs.

For the remaining "script" and "rules" keywords, the configuration is referenced from the ".base_nodejs_config" hidden job.

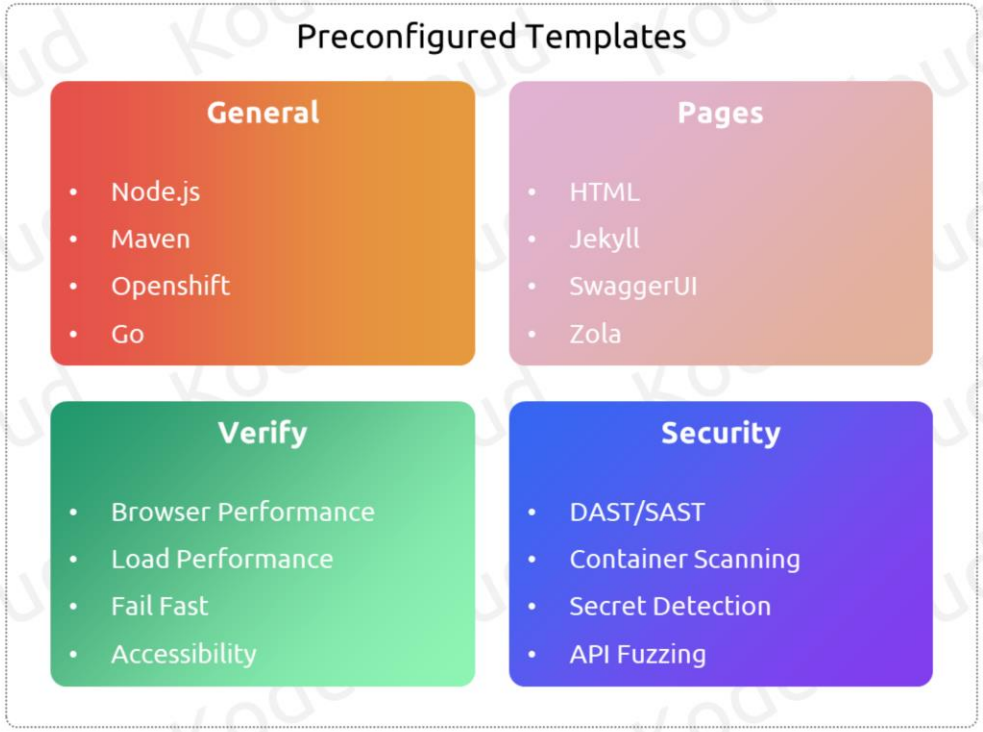
Upon implementing these steps, the final expanded configuration of the pipeline takes the following form.

Templates and Types of Includes

Templates

GitLab

```yaml
unit_testing:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script:
    - npm test

code_coverage:
  image: node:17-alpine3.14
  services:
    - name: siddharth67/mongo-db:non-prod
  cache:
    policy: pull-push
    key:
      files: [package.json]
    paths: [node_modules]
  before_script:
    - npm install
  script: |
    npm run coverage
  rules:
  - if: $CI_COMMIT_BRANCH == "main"
    when: manual
```

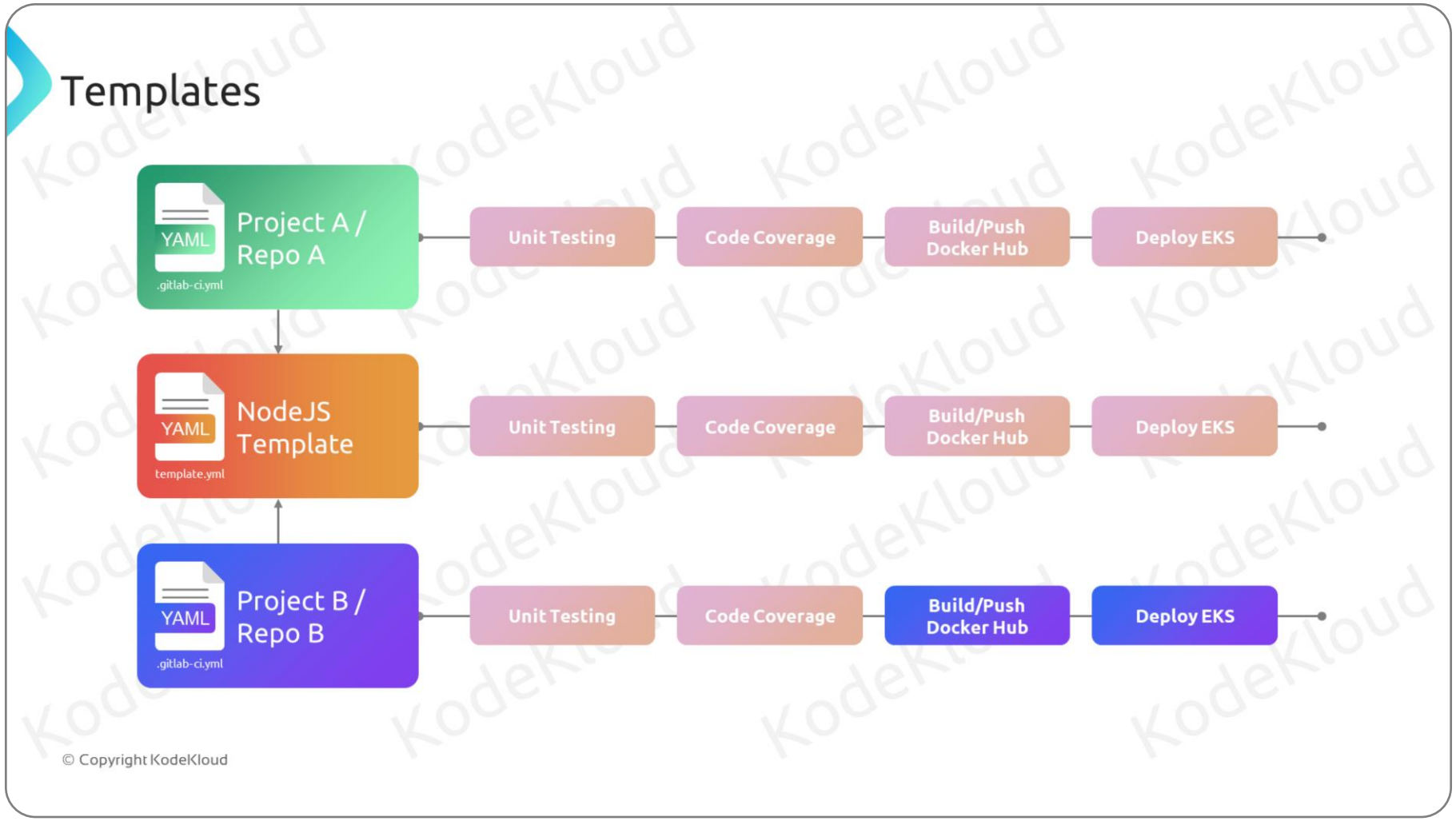


While GitLab CI provides powerful tools for defining your CI/CD pipelines, manually writing YAML code for every project can be tedious and prone to errors. This is where templates come in, offering several compelling advantages that enhance your DevOps workflow.

GitLab CI templates are pre-configured YAML files that you can include in your .gitlab-ci.yml file to define your pipeline stages and jobs.

GitLab already has numerous pre-made templates for different programming languages, security scanning, and others.
You can save a lot of time and effort by not writing the pipeline YAML code for your pipeline from scratch.
You can easily reuse templates across different projects.

Templates come in two main types:
Pipeline templates: These templates define an entire CI/CD pipeline for a specific project type or language, like a Node.js application or a Ruby on Rails project.
Job templates: These templates define individual pipeline jobs, such as a unit test job, security job or a Docker image build job.



Consider there is a pre-built template tailored for Node.js projects that streamlines testing, building, and deployment. It includes stages for:
Unit Testing to ensure code quality.
Code Coverage to measure how thoroughly tests cover the code.
Building and Pushing Docker Images to Docker Hub for easy sharing and deployment.
Deploying to AWS Elastic Kubernetes Service (EKS) for scalable container orchestration.

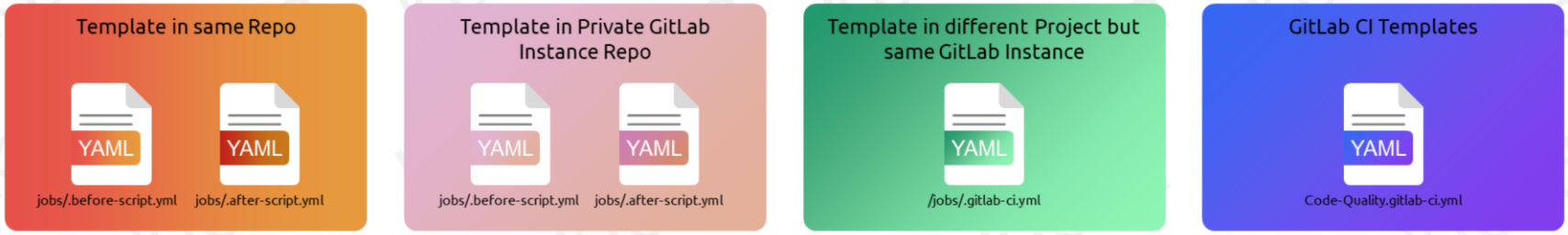
Here's how teams can benefit from this template:

Project A:
Adopted the template: Integrated it into their GitLab CI configuration.
Adjusted test configurations and added credentials for Docker Hub and AWS EKS.
Automated their workflow: Used the pipeline to efficiently test, build, and deploy their application to AWS EKS.

Project B:
Reused the template's core components: Leveraged the Unit Testing and Code Coverage stages.
Modified other stages: Tailored them to push images to Google Container Registry (GCR) and deploy to Google Kubernetes Engine (GKE).

So, what are the Key takeaways:
Reusable templates promote efficiency: They save time and effort by providing a pre-built structure for common tasks.
Customization allows flexibility: Teams can adapt templates to meet their specific needs and infrastructure choices.
Modular design enables selective reuse: Teams can pick and choose the stages they need, making the template adaptable to various project requirements.

Types of Includes



includes

```yaml
include:
  - local: '/jobs/.after-script.yml'
  - '/jobs/.before-script.yml'
  - remote: 'https://private.gitlab.com/awesome-project/raw/main/jobs/.before-script.yml'
  - 'https://private.gitlab.com/awesome-project/raw/main/jobs/.after-script.yml'
  - project: 'my-group/avengers-project'
    ref: main
    file: '/jobs/.gitlab-ci-.yml'
  - template: Code-Quality.gitlab-ci.yml
    rules:
      - if: $CI_COMMIT_BRANCH == "feature/*"
```


Templates may be stored in various locations, including local repositories, private repositories, GitLab-provided templates, and templates from other projects. So, how can we reference and include these templates into our pipeline?

To include external YAML files in your CI/CD configuration, utilize the include keyword.

Assume we have a couple of job templates located within the same project directory as the .gitlab-ci.yml file.

Use include:local to include a file that is in the same repository and branch.
You could also use a shorter syntax to define the path without the local keyword as well.

If you have job templates within the same project directory as the .gitlab-ci.yml file, use include:local to include a file from the same repository and branch. You can also use a shorter syntax to specify the path without the local keyword.

Assume we have a couple of job templates located within a private/another Gitlab instance project.

Use include:remote to include YAML files from another GitLab instance.
Similar to local, You could also use a shorter syntax to define the remote URL without the remote keyword as well.
Be careful when including another project's CI/CD configuration file. From a security perspective, this is similar to pulling a third-party dependency.

For job templates in a private GitLab instance project, use include:remote to include YAML files from another GitLab instance. Similar to local, a shorter syntax without the remote keyword can also be used. However, be careful when including another project's CI/CD configuration file. From a security perspective, this is similar to pulling a third-party dependency.

Assume we have a templates located within another private project on the same GitLab instance

Use include:project keyword to include a file from another project. Possible inputs for this type are include:project: containing The full GitLab project path.
include:file A full file path, or array of file paths, relative to the root directory (/). The YAML files must have the .yml or .yaml extension.

include:ref: Optional. The ref to retrieve the file from. Defaults to the HEAD of the project when not specified.

If the templates are in another private project on the same GitLab instance, use the include:project keyword. Possible inputs for this type are:
include:project:, containing the full GitLab project path.
include:file, a full file path or an array of file paths relative to the root directory (/). The YAML files must have the .yml or .yaml extension.
include:ref (optional), specifying the ref to retrieve the file from. Defaults to the HEAD of the project when not specified.

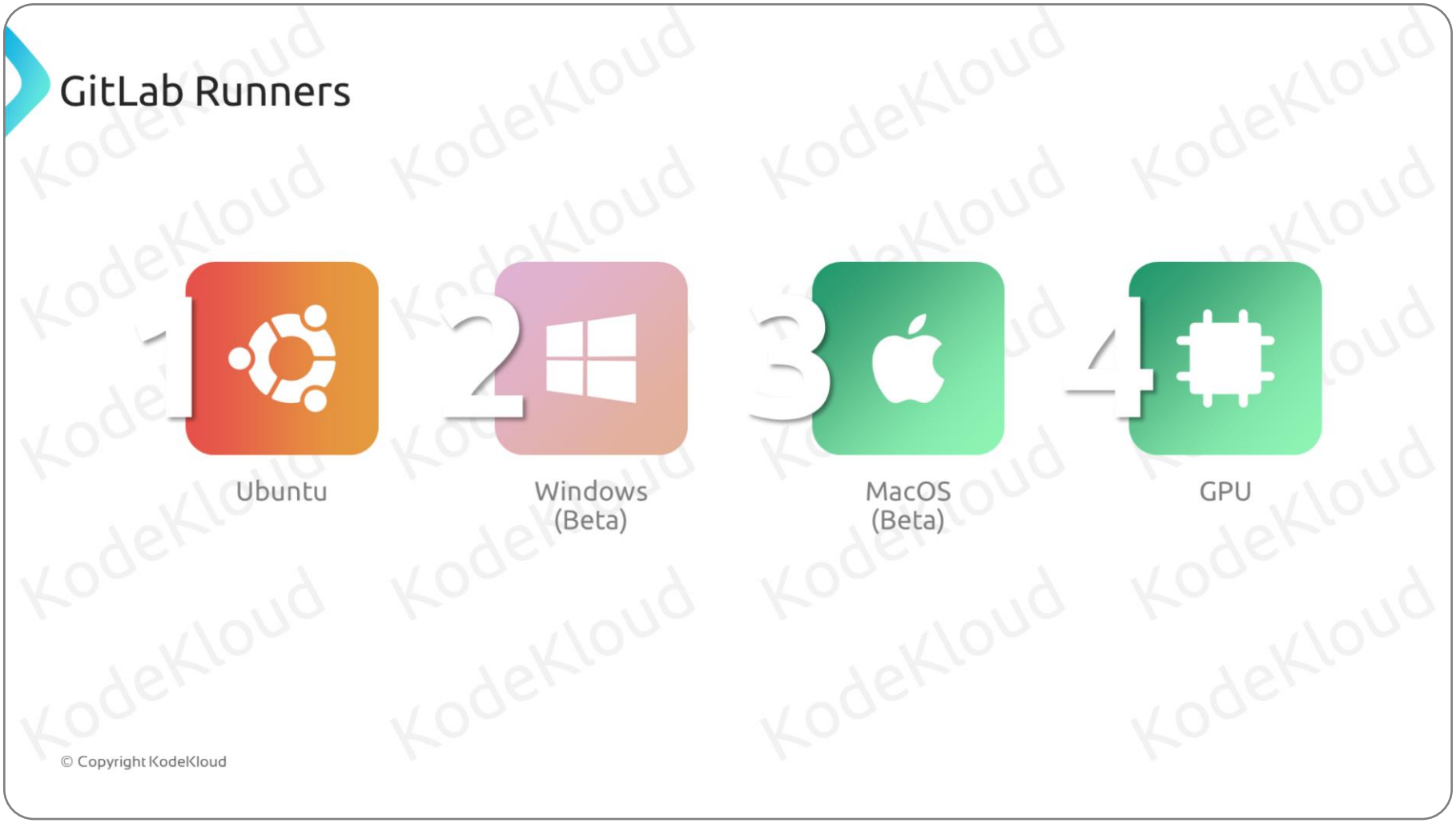
Assume we want to use Gitlab’s pre-built Code-Quality template stored in lib/gitlab/ci/templates directory. Use include:template keyword to include .gitlab-ci.yml templates.

If you wish to use GitLab's pre-built Code-Quality template stored in lib/gitlab/ci/templates directory, utilize the include:template keyword to include .gitlab-ci.yml templates.

Additionally you could use rules with include to conditionally include/exclude configuration files.
In this example the Code-Quality template will only be included and used if the commit branch name starts with feature

Additionally, you can even control when a template is included using rules. For example, include the Code-Quality template only when the commit branch name starts with "feature".

Self-Managed Runners



In our earlier sessions, we discussed about Runners which are essentially virtual machines responsible for performing various tasks within a Gitlab CI/CD pipelines. For example, a runner can handle tasks such as cloning your repository, installing necessary software, and executing specified commands.

We have also discussed about SaaS runners which are hosted and managed by GitLab. These runners are fully integrated with GitLab.com and are enabled by default for all projects, with no configuration required.

Each GitLab-hosted runner is essentially a fresh virtual machine that GitLab hosts. You have the option to choose from different operating systems, including

Linux runners - Supports a wide range of languages and tools
Windows runners (Beta) - Ideal for projects that use Windows-specific tools or require testing on Windows systems
macOS runners (Beta) - Good choice for projects that require macOS-specific tools
GPU-enabled SaaS runners - Accelerate heavy compute workloads for high-performance computing workloads such as the training or deployment of Large Language Models (LLMs) as part of ModelOps workloads

Self-managed runners are machines that you deploy and manage yourself. This gives you more control over the hardware, operating system, and software tools that your workflows run on. Self-managed runners can be run on any operating system, including custom operating systems.

Here are some reasons why you might choose to use self-hosted runners:

Customizable environment: You can set up your runners with the specific software and resources needed for your projects, giving you complete control over the execution environment. This can be useful for running jobs that require specific tools or libraries, or that cannot be run on shared runners due to security restrictions.

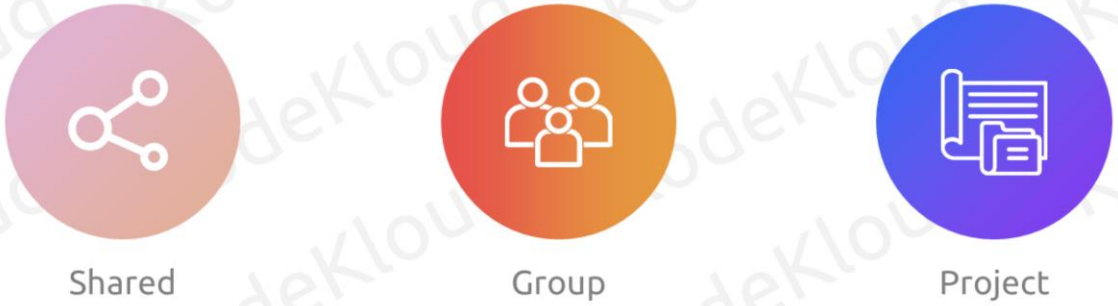
In some organizations, strict security and compliance policies may prevent the use of Gitlab-hosted runners for certain workflows. Self-managed runner can be set up in a controlled environment preventing conflicts and potential security vulnerabilities that could arise from sharing resources with other projects.

Depending on the demand for Gitlab-hosted runners in your organization, there might be wait times for available runners. Self-managed runners eliminate this wait time, as they are dedicated to your projects.

You can scale your self-managed runner pool based on your project's needs. This can be particularly useful if you have many concurrent workflows or if you need to run multiple workflows in parallel.

Self-managed runners can be placed in specific geographic locations to reduce latency for your CI/CD workflows, especially if you have a global user base or need to comply with data residency regulations.

Self-Managed Runners

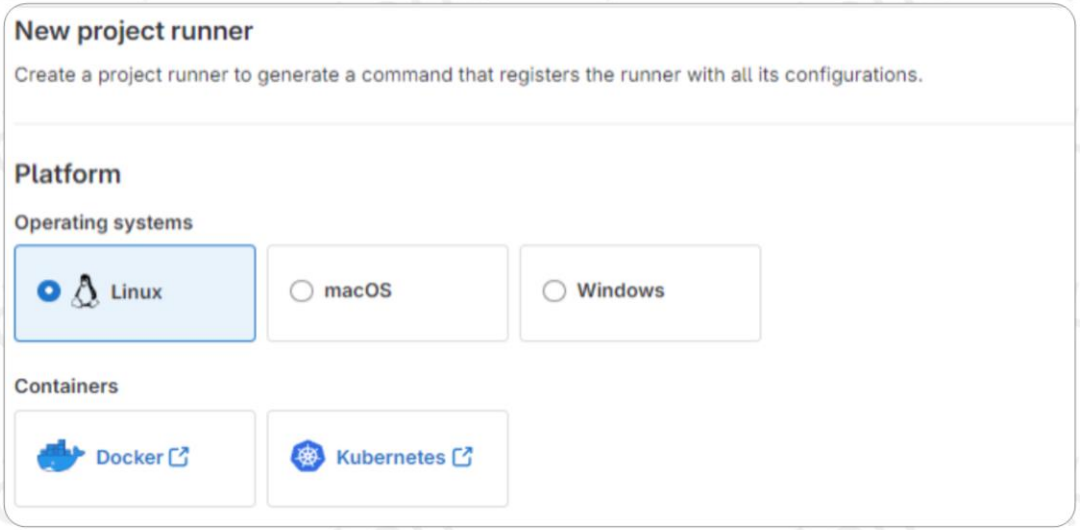


In summary, while Gitlab-hosted runners are convenient and suitable for many use cases, self-managed runners provide more control, flexibility, and customization options.

The next step is to understand how to configure runners in the context of GitLab CI/CD.

GitLab Runners are of three types based on their access scope:
Shared runners are available to all groups and projects in a GitLab instance.
Group runners are available to all projects and subgroups in a group.
Project runners are associated with specific projects. Typically, project runners are used by one project at a time.

Self-Managed Runners

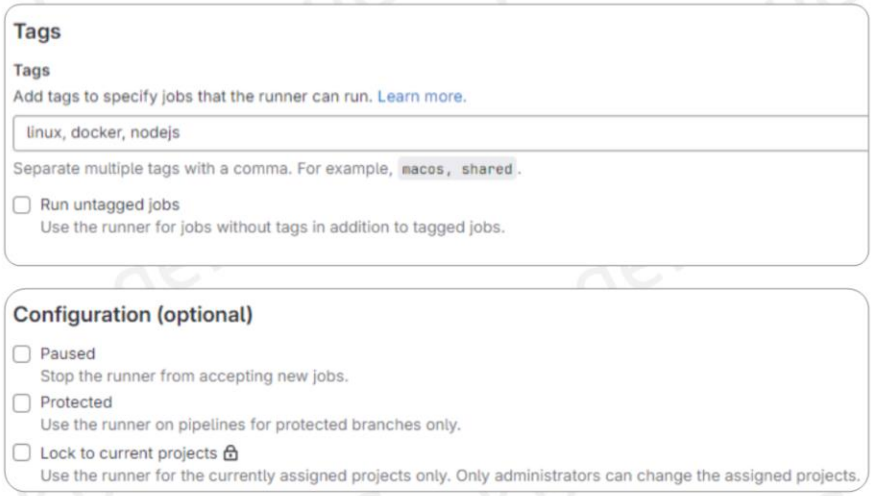


To set up a self-managed runner as a Project runners, you can navigate to the desired projects/repository's CI/CD settings. There, you'll find an option to disable shared runner for this project and also to create a new project-level runner under the "Runners" section.

During this setup, you'll have the opportunity to Select the operating system of the machine where you'll install the runner, add tags and configuration additional settings.

You can use tags to control the jobs a runner can run.
You can also Check the Run untagged jobs box if you don't want to restrict this runner to specific jobs based on tags.

Self-Managed Runners

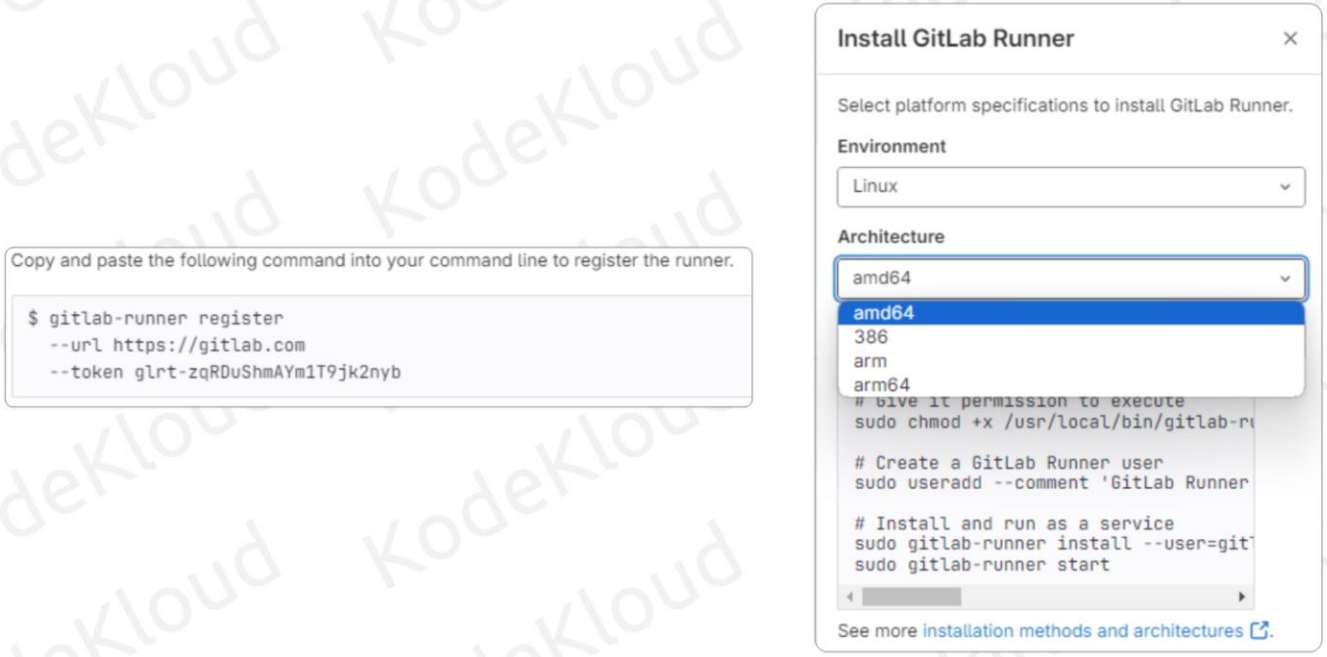


To set up a self-managed runner as a Project runners, you can navigate to the desired projects/repository's CI/CD settings. There, you'll find an option to disable shared runner for this project and also to create a new project level runner under the "Runners" section.

During this setup, you'll have the opportunity to Select the operating system of the machine where you'll install the runner, add tags and configuration additional settings.

You can use tags to control the jobs a runner can run.
You can also Check the Run untagged jobs box if you don't want to restrict this runner to specific jobs based on tags.

Self-Managed Runners



You can click Create runner to generate a registration token to register the runner with Gitlab.

Along with the token the Gitlab UI also provides steps to install the Gitlab runner application for the selected environment and architecture.

Self-Managed Runners

Download and Install GitLab Runner Application

```
# Download the binary for your system
$ curl /usr/local/bin/gitlab-runner https://gitlab-runner-down.s3.com/latest/gitlab-runner-linux-amd64

# Give it permission to execute
$ chmod +x /usr/local/bin/gitlab-runner

# Create a GitLab Runner user
$ useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash

# Install and run as a service
$ gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
$ gitlab-runner start
```


When setting up a runner for the first time on an operating system, it is essential to install the GitLab Runner application prior to registering the runner.

Walkthrough the steps as shown in slide

Now that the GitLab Runner Application is installed, the next step is to register the runner.

Self-Managed Runners

Registering Runner

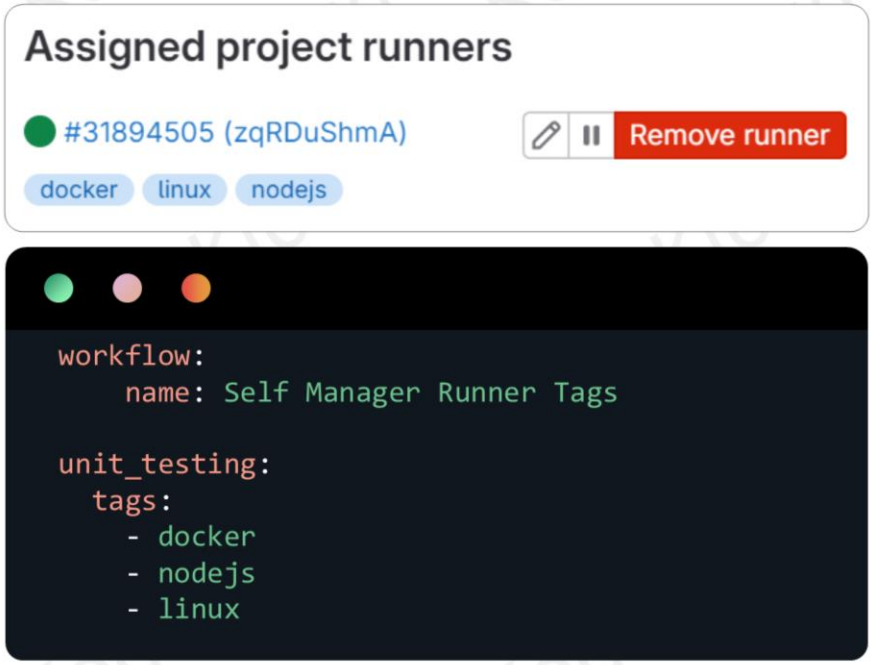
```
$ gitlab-runner register  --url https://gitlab.com  --token glrt-zqRDuShmAYm1T9jk2nyb
Enter the GitLab instance URL (for example, https://gitlab.com/):
[https://gitlab.com]: https://gitlab.com

Verifying runner... is valid                        runner=zqRDuShmA

Enter a name for the runner. This is stored only in the local config.toml file:
[ubuntu]: nodejs-docker-ubuntu-runner

Enter an executor: instance, custom, shell, ssh, docker+machine, kubernetes, docker-autoscaler, parallels,
virtualbox, docker, docker-windows:
Shell

Runner registered successfully.

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
```


During the registration process, you'll be prompted to provide some details:
Your GitLab instance's URL: This ensures the runner connects to the correct GitLab environment.
A name for the runner: This helps you easily identify it within your CI/CD infrastructure.
The preferred executor type: This determines how jobs will be executed, such as using a shell or Docker.
Upon successful registration, all these configuration settings are automatically saved into a file named config.toml. This file serves as a central reference for managing the runner's behavior and settings.

To further customize your runner, edit the config.toml file located in the GitLab Runner installation directory on the machine.
This file allows you to define:

- Runner name and tags for identification.
- Executor like Docker or shell to determine how jobs run.
- Resources like CPU, memory, and disk limits.
- Environment variables for job execution.
- Caching paths and others.

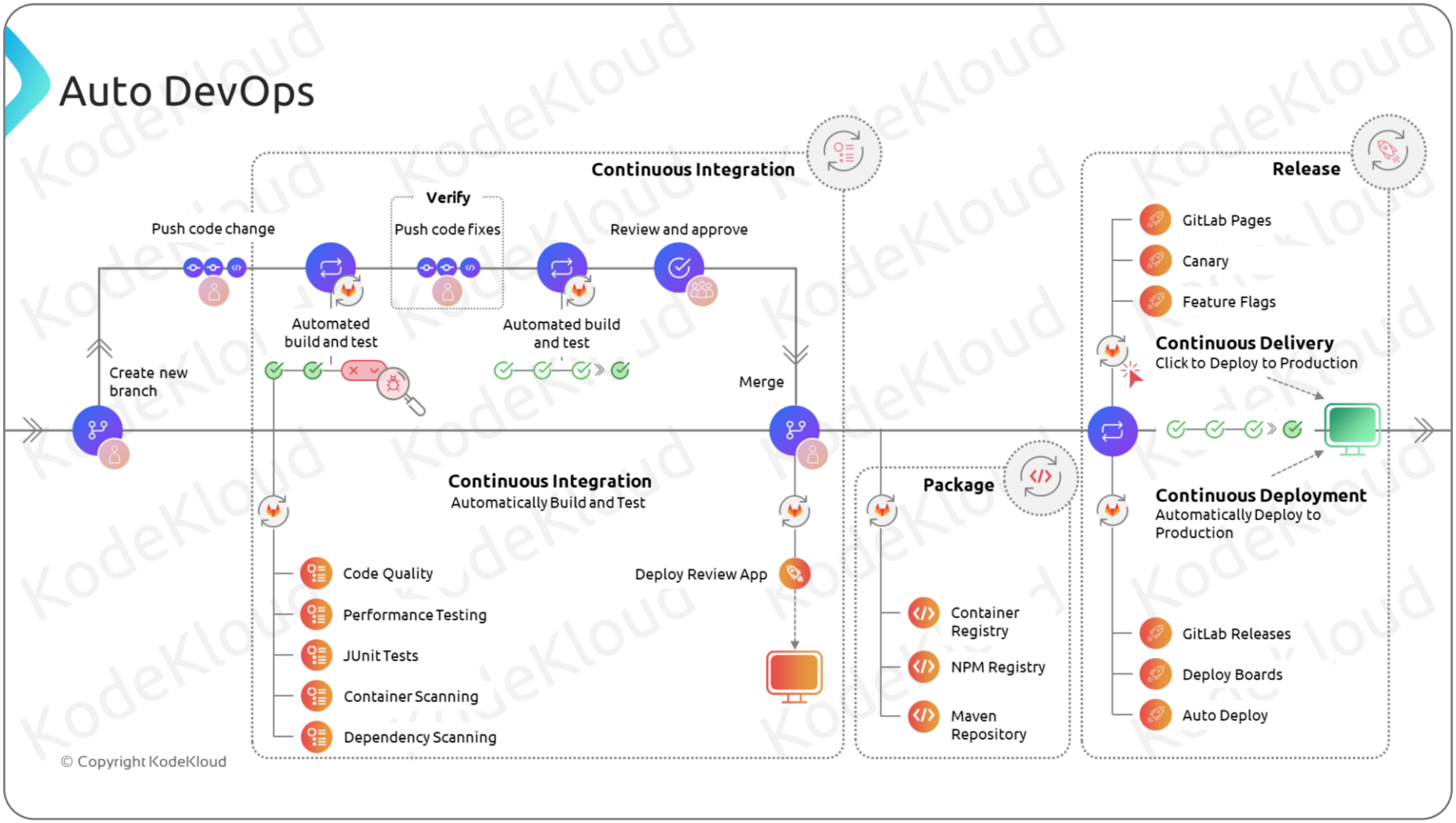
Self-Managed Runners



To check the status of this setup, you can go to the GitLab CI/CD settings and check the runner status.

Use tags at a job level to select a specific runner from the list of all runners that are available for the project.

GitLab Auto DevOps



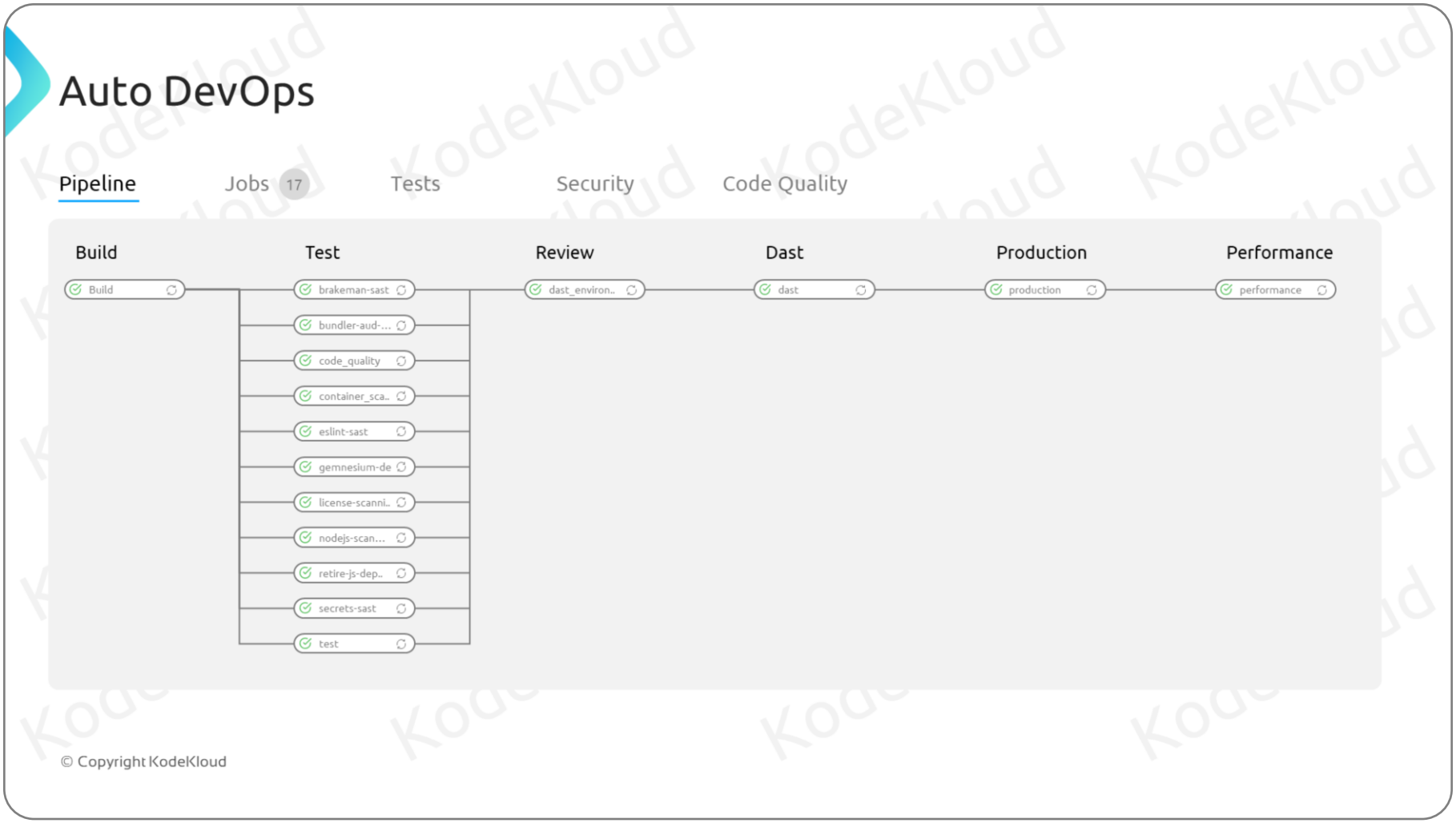
Auto DevOps is a feature within GitLab that takes CI/CD automation to the next level. It essentially configures and runs a complete CI/CD pipeline for you, eliminating the need for manual setup.

What can we expect from AutoDevOps?
Analyze your project's programming language and framework to determine the appropriate tools and settings.
Set up stages for building, testing, and deploying your application, using industry best practices.

Integrate vulnerability scanning, code quality checks, container scanning, dependency scanning and other security measures into the pipeline.
Create review apps for testing the application before merging to the source branch.
Post merging, we can deploy the app using either Continuous delivery or continuous deployment.

Auto DevOps is a powerful tool, but it's not a one-size-fits-all solution. Depending on your project complexity and preferences, you may need to further customize the pipeline

Let's see how it works at the project level.



Once Auto DevOps is enabled at the CI/CD Settings of the project, it starts by detecting your code language and use CI/CD templates to build and run default pipelines.

It starts with the BUILD job, Auto DevOps leverages a pre-existing Dockerfile, if present, or applies Heroku buildpacks to construct a Docker image by identifying the coding language. Regardless of the path taken, a Docker image will be generated by the end of the Build job.

Next is the testing stage. During the "test" stage of GitLab Auto DevOps, your code gets put through its paces using both your project's existing tests and the power of Heroku buildpacks. These handy tools simplify application testing for various languages, taking some of the heavy lifting off your shoulders.

However, it's important to note that not all programming languages are currently supported in this stage. Few of the supported languages are ruby, nodejs, maven, gradel, python, java, go etcs.
This stage also includes Code Quality Analysis, Static Application Security Testing, Auto Secret Detection, Container Scanning and others.
Metrics and reports of all these jobs will be available for you to analyse at the end of the pipeline.

Auto DevOps can also be configured to deploy application to Kubernetes cluster. We can configure our Gitlab project with Kubernetes clusters, it currently supports

Kubernetes.
Amazon Elastic Container Service (ECS).
Amazon Elastic Kubernetes Service (EKS).
Amazon EC2.
Google Kubernetes Engine.
Bare metal.

After the testing stage if deployment is configured, the next stage is review apps.

The review stage is available during merge requests, here GitLab DevOps automatically deploys that change to a different environment to review how the application works before deploying to production. It uses an HELM auto-deploy-app chart to deploy review applications. This helm chart can be modified as per requirements.

After you've deployed your code changes into a Review App, GitLab Auto DevOps takes things a step further. It's not just about seeing your application in action, it's about ensuring it's secure and ready for the real world. That's where Dynamic Application Security Testing (DAST) comes in!

Think of DAST as a tester for your Review App. It uses the open-source OWASP ZAProxy tool to scan your running application, searching high and low for any potential security vulnerabilities or suspicious

behavior.

No need to worry about manual setup: Auto DevOps integrates DAST into the Review App workflow. Once your app is deployed, ZAProxy gets to work, mimicking real-world attacks and probing for weaknesses.

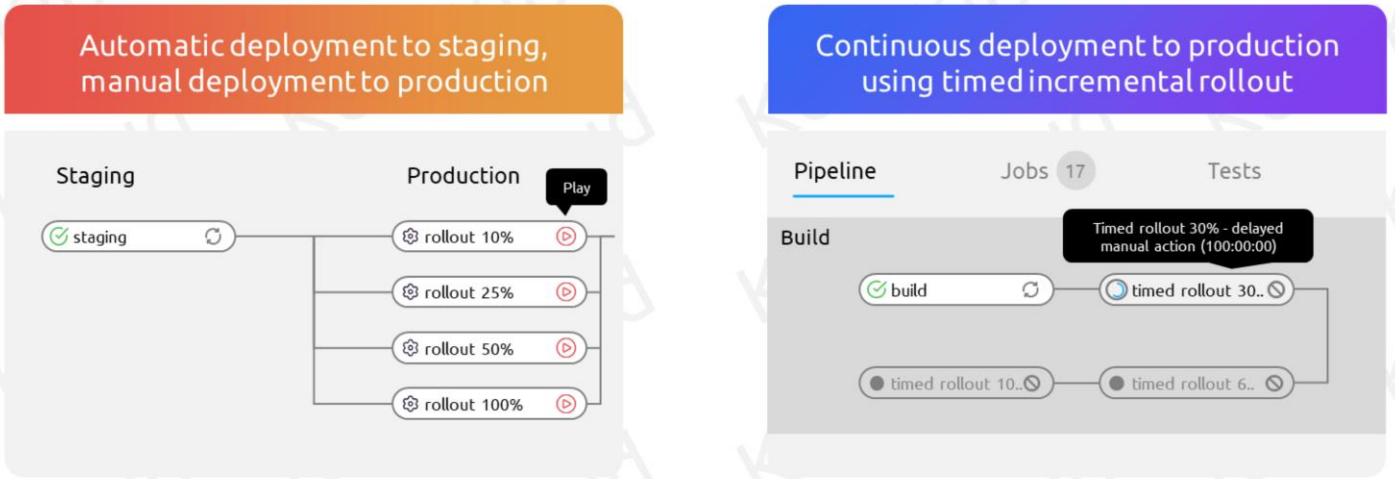
If ZAProxy uncovers any security vulnerabilities, you'll get a detailed report. This report pinpoints the vulnerabilities, explains their severity, and even offers suggestions for fixing them.

Depending on your AutoDevOps deployment configuration, after merging the feature branch to the source branch, Auto DevOps automatically deploys that to the staging or production environment

When your application reaches production or staging, Auto DevOps doesn't stop. It runs browser performance testing to see how quickly your pages load. These tests compare your new version to older ones, making sure it doesn't get sluggish over time.

Auto DevOps Deployment Strategy

Continuous deployment to production



When using Auto DevOps to deploy your applications, you could choose one of the 3 continuous deployment strategies that works best for your needs.

Within Continuous deployment to production strategy – It Enables Auto Deploy job to continuously deployed to the production environment.

In Automatic deployment to staging, manual deployment to production strategy, the default branch is continuously deployed to staging and continuously delivered to production manually. It is possible to configure GitLab to do incremental rollouts manually. Incremental rollout depend on the number of pods that are defined for the deployment, which are configured when the Kubernetes cluster is created.

For example, if your application has 10 pods and a 10% rollout job runs, the new instance of the application is deployed to a single pod while the rest of the pods show the previous instance of the application.

When the jobs are built, a play button appears next to the job’s name. Select play to release each stage of pods.

In Continuous deployment to production using timed incremental rollout strategy it uses incremental rollouts. Timed rollouts behave in the same way as manual rollouts, except that each job is defined with a delay in minutes before it deploys. Selecting the job reveals the countdown.

This strategy Continuously deploy to production with a 5-minute delay between rollouts. The time delay is customizable.

=======

When employing Auto DevOps for application deployment, you have the flexibility to choose from three continuous deployment strategies based on your requirements.
In the "Continuous Deployment to Production" strategy, the Auto Deploy job is configured to deploy continuously to the production environment.
For the "Automatic Deployment to Staging, Manual Deployment to Production" strategy, the default branch undergoes continuous deployment to staging and manual delivery to production. GitLab can be configured to perform manual incremental rollouts. The incremental rollout depends on the number of pods defined for deployment during the creation of the Kubernetes cluster. As an example, if the application has 10 pods and a

10% rollout job is executed, the new application instance is deployed to a single pod while the remaining pods retain the previous application instance. A "play" button appears next to the job's name when jobs are built, allowing the release of each stage of pods.
In the "Continuous Deployment to Production Using Timed Incremental Rollout" strategy, incremental rollouts are employed with timed delays. Timed rollouts function similarly to manual rollouts, but each job is defined with a specified delay in minutes before deployment. The countdown is visible when selecting the job. This strategy involves continuous deployment to production with a customizable 5-minute delay between rollouts.

======

Auto DevOps offers three continuous deployment options for tailoring your application updates:

Continuous Deployment to Production:
Automatic updates directly to production upon code changes.

Automatic Staging, Manual Production:
Updates automatically flow to staging, waiting for manual approval before hitting production.
Optional GitLab feature allows staged rollouts in controlled increments (percentage-based).
Example: For 10 pods and a 10% rollout, one pod updates with the new version, while others remain unchanged.
Manual release of each rollout stage triggered by clicking a "play" button

Timed Incremental Rollout:
Similar to manual staging, but each rollout stage automatically deploys after a set delay (in minutes).
Customizable delay times offer added control over update progression.
Example: Continuous deployment with 5-minute intervals between rollout stages.

By choosing the approach that best suits your needs, you can ensure seamless and controlled updates for your applications!

KodeKloud

Visit www.kodekloud.com to discover more!