



KodeKloud

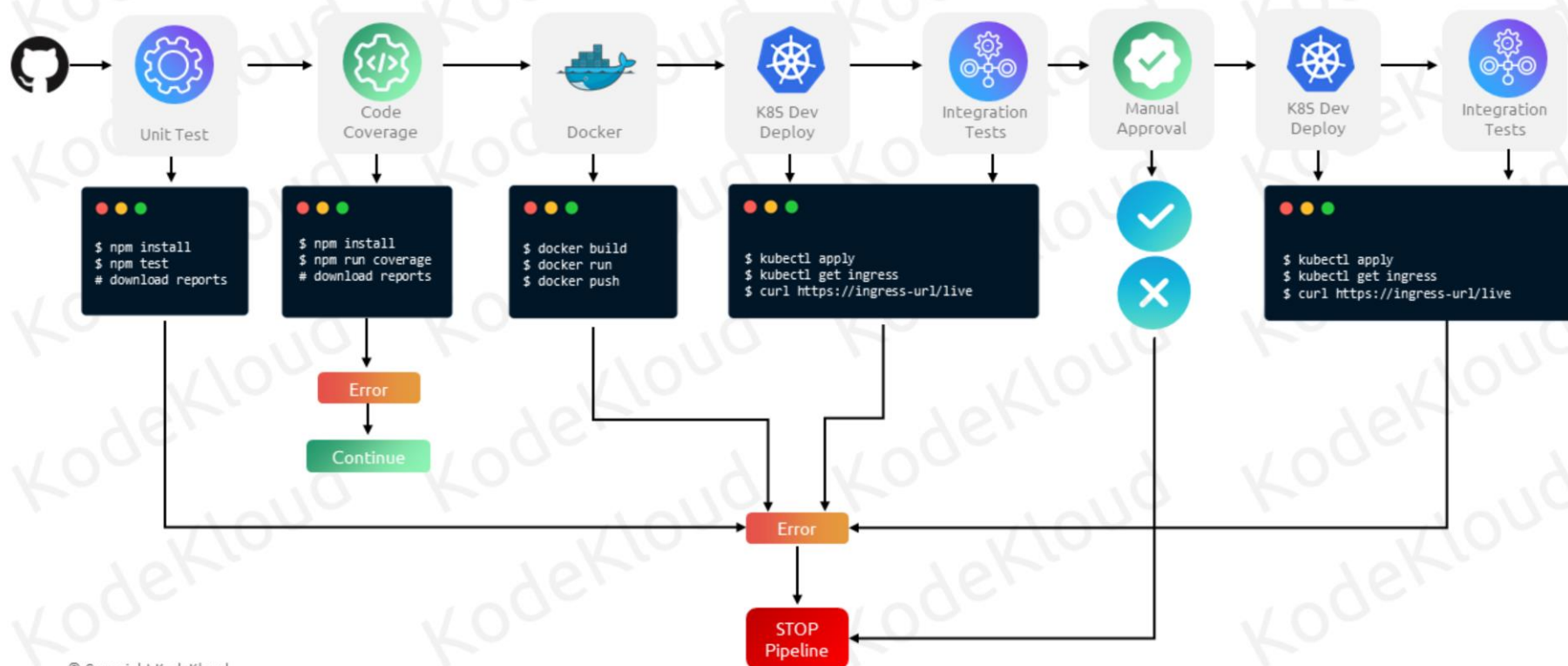
© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.



Understanding Deployment Use Cases

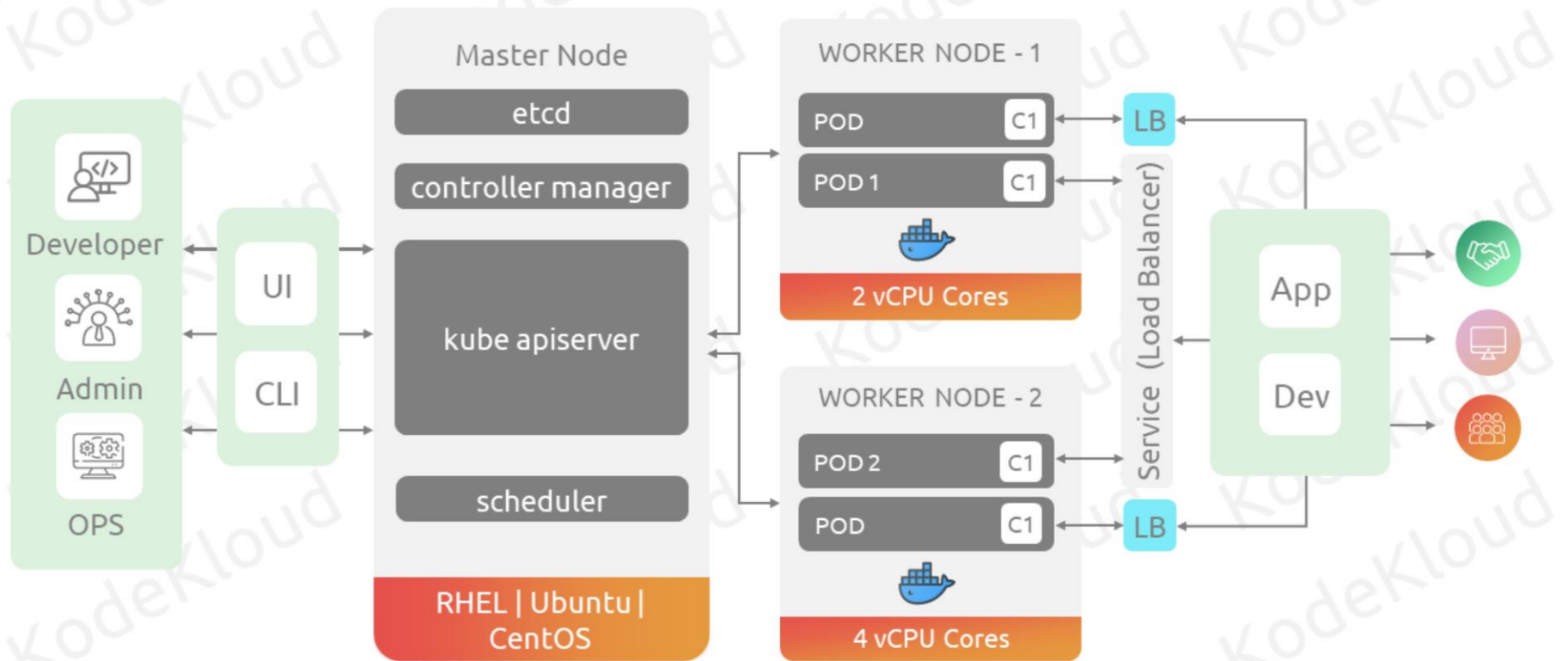
Understanding XYZ Team DevOps Pipeline





Kubernetes – A Brief Overview

Kubernetes Basics



© Copyright KodeKloud

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

1. 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 includes 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.

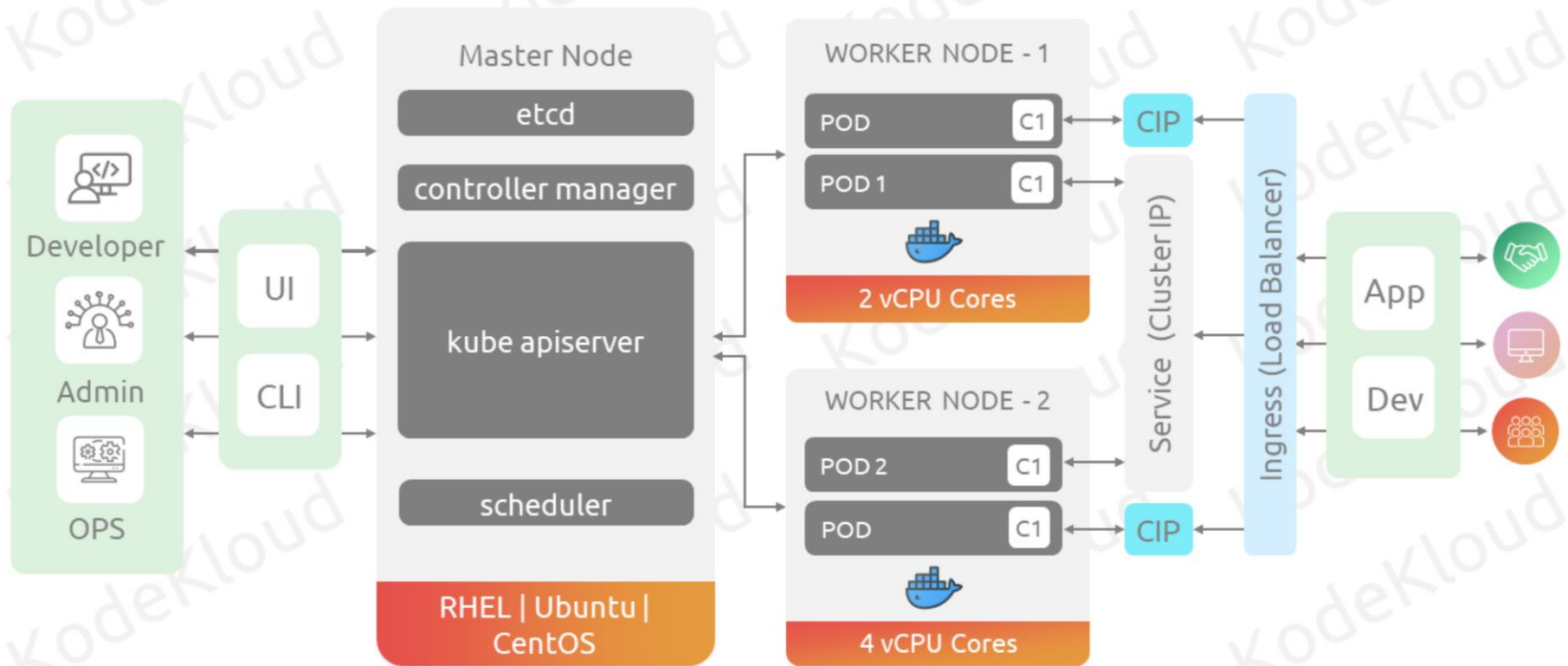
Kubernetes offers several types of services to enable network communication between pods. One of these types is the "LoadBalancer" service which is used to expose your service to the external world, typically in a cloud environment. It creates an external load balancer (e.g., an AWS ELB or GCP Load Balancer)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



© Copyright KodeKloud

1. 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.
2. 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.

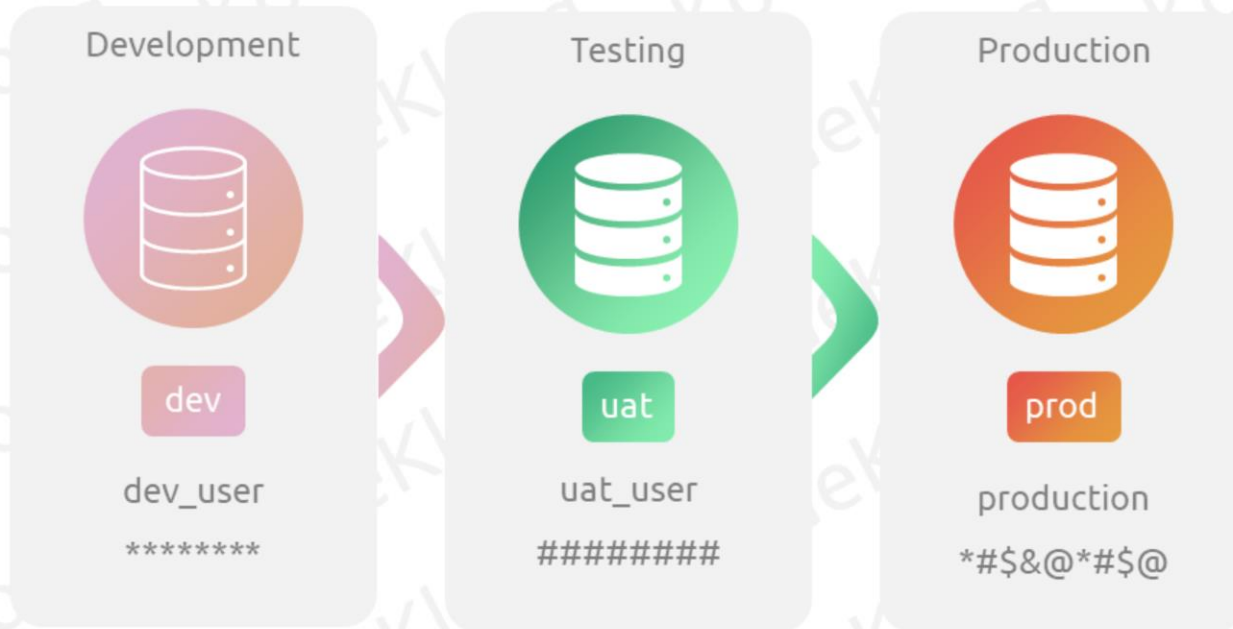
One major reason is ClusterIP services provide internal access to pods within the cluster but are not directly exposed externally. This allows you to have fine-grained control over which services are accessible from within the cluster, ensuring that not every service is exposed by default.

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 GitHub Environments

Environments



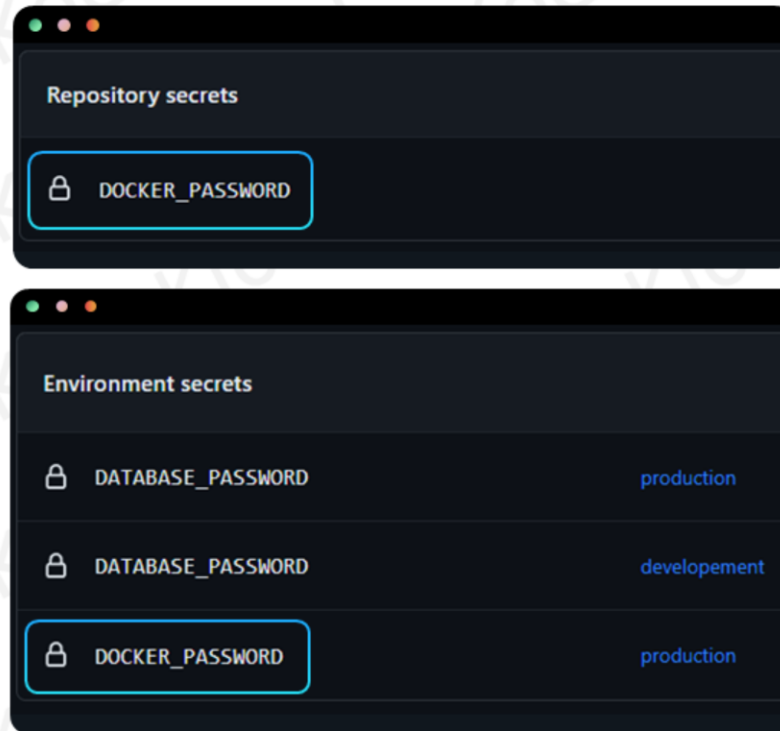
© Copyright KodeKloud

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 its own services such as Databases, vaults, rest apis etc. Most of these services are secured using username, password credentials or some use apikeys and these vary for each environment. So how do we securely store and use them on Github?

In GitHub Actions, environments are a powerful feature that can help you to organize, protect, and visualize your deployments.

One major feature of Environments is to store secrets and variables that are specific to a particular environment. This allows you to keep secrets out of your workflow files and make them easier to manage.

Environments



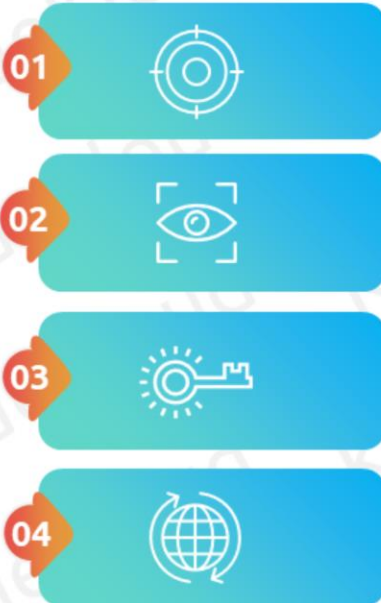
© Copyright KodeKloud

In one of our earlier session, we made use of Repository secrets to secure store Docker credentials.

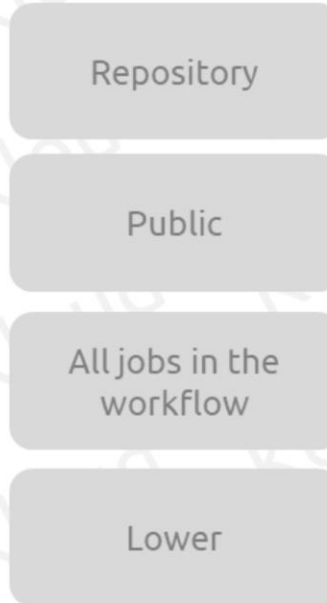
GitHub Actions repository secrets and environment secrets are both ways to store sensitive information that can be used in GitHub Actions workflows. However, there are some key differences between the two:

Environments

Feature



Repository Secrets



Environment Secrets



© Copyright KodeKloud

1. **Scope:** Repository secrets are specific to a single repository, while environment secrets are specific to a particular environment. This means that environment secrets can be used in workflows that run in different repositories, as long as they are referencing the same environment.
2. **Visibility:** Repository secrets are visible to all users who have access to the repository. Environment secrets can be made private, so that they are only visible to users who have access to the environment.

3. **Accessibility:** Repository secrets are accessible to all jobs that run in the workflow. Environment secrets are only accessible to jobs that are running in the environment.
4. **Precedence:** If a secret exists at both the repository and environment level, the environment secret takes precedence.

Deployment Protection Rules



Manual Deployment



Delay Deployment



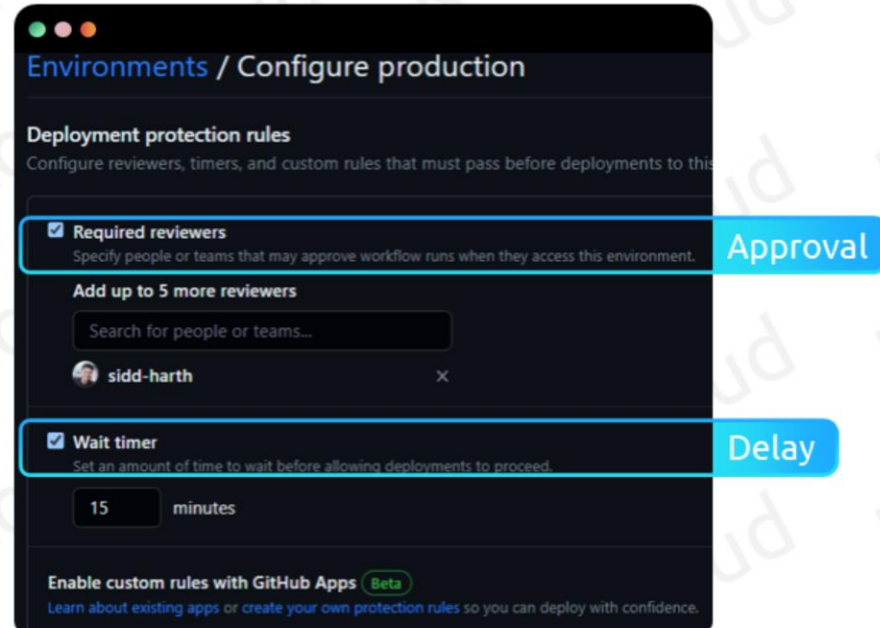
Restrict Deployment

© Copyright KodeKloud

In addition to secret storage, environments also offer Deployment protection rules which provides a way to control who can deploy changes to an environment. They can be used to require manual approval, delay a deployment, or restrict deployments to certain branches or users.

Custom protection rules can also be established through GitHub Apps, utilizing external applications for deployment control.

Deployment Protection Rules



The screenshot shows the 'Environments / Configure production' page in GitHub. Under the 'Deployment protection rules' section, there are two rules configured:

- Required reviewers**: A rule that requires approval. It includes a search bar for adding reviewers, with 'sidd-harth' currently listed. A blue callout box labeled 'Approval' points to this rule.
- Wait timer**: A rule that imposes a delay before deployment. It is set to 15 minutes. A blue callout box labeled 'Delay' points to this rule.

At the bottom, there is a section for 'Enable custom rules with GitHub Apps' with a 'Beta' badge and links to learn more or create custom rules.

© Copyright KodeKloud

So, how do we set up these rules? Environments can be created by accessing the Settings tab within a GitHub repository.

Once the environment is created, it offers the deployment protection rules.

The first one is,

Required reviewers, we can enter up to 6 people or teams. Only one of the required reviewers needs to approve the job for it to proceed. This rule would help to prevent unauthorized changes from being deployed to production or any other environment.

The wait-timer rule introduces a delay to the deployment process for a defined duration. This delay provides an opportunity to review the modifications before they go live in the production environment.

Deployment Protection Rules

Deployment branches

Limit which branches can deploy to this environment based on rules or naming patterns.

Selected branches ▾

Restrict

1 branch allowed

main

Currently applies to 1 branch

Edit



Remove

⊕ Add deployment branch rule

Deployment protection rules

Reviewers, timers, and other rules protecting deployments in this run

Prerequisite

Event	Environments	Comment
<div> sidd-harth approved now</div>	production	looks good!
<div> Wait timer completed 3 weeks ago</div>	development	1 minute wait timer

Lastly, there are the "Deployment Branches Rules," which allow you to restrict deployments to a particular branch. For instance, in the provided example, this rule would prevent changes from being deployed to production unless they are first merged into the main branch.

When a workflow references an environment, the job will not start until all of the environment's protection rules pass. The job will also have access to the environment's secrets. This allows you to use the secrets in your workflow steps.

Ultimately, the best way to decide which deployment protection rules to use is to consider the specific needs of your project. If you are unsure, you can always start with a few simple rules and then add more rules as needed.



KodeKloud

© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.