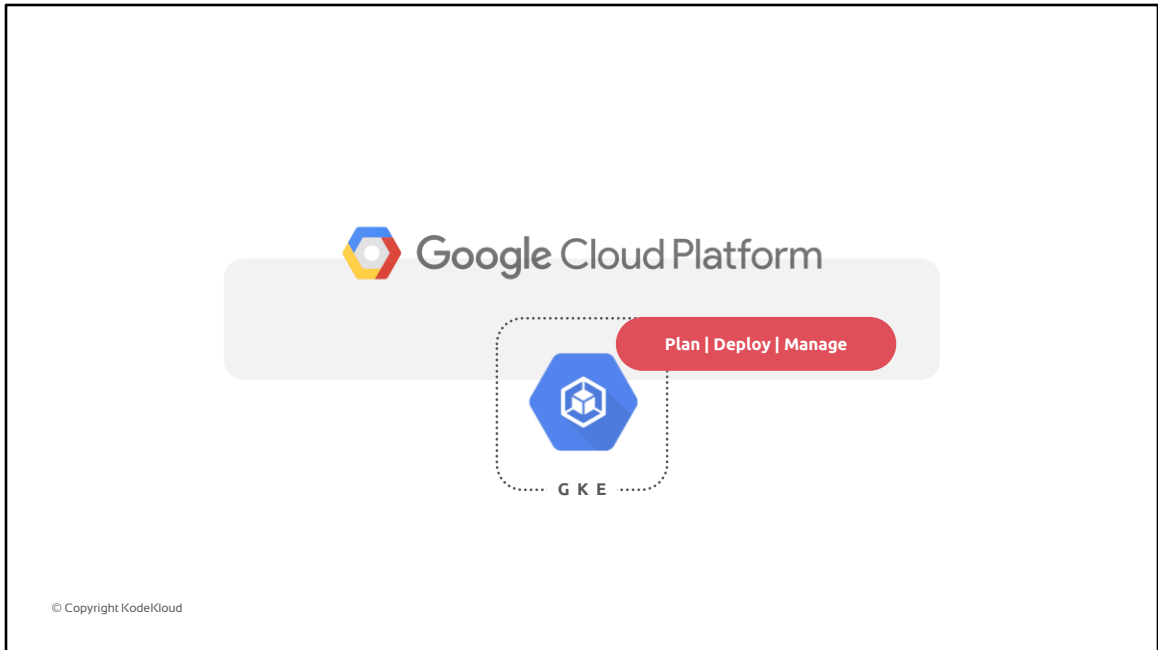
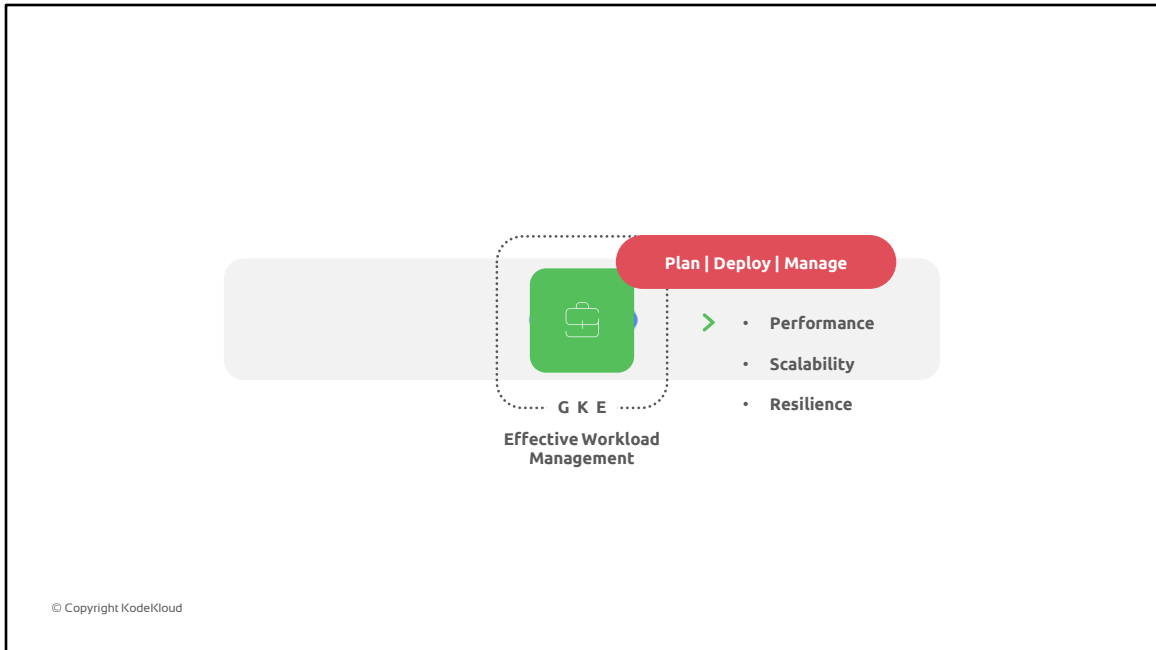


## Section Introduction – Plan, Deploy And Manage Workloads On GKE

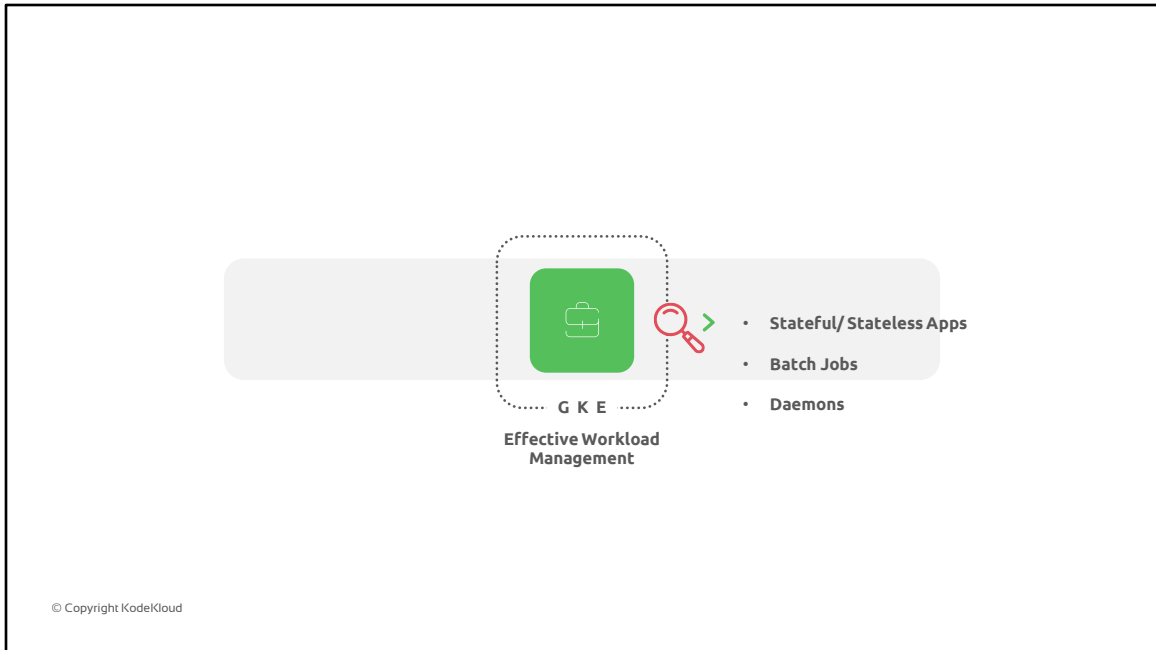
© Copyright KodeKloud



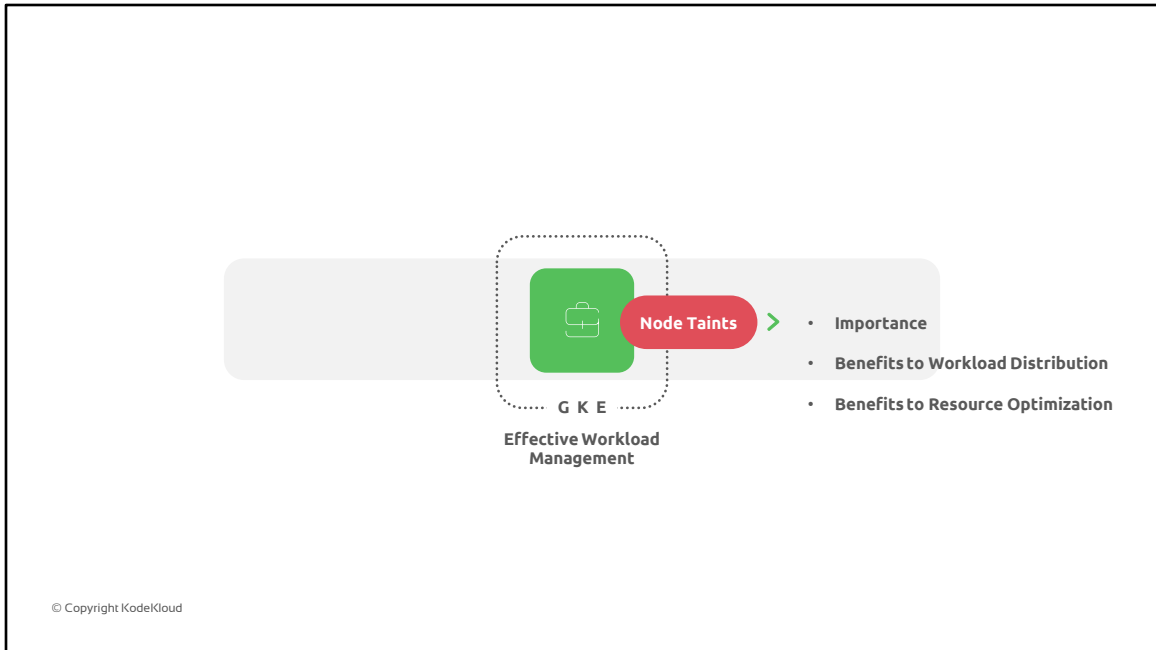
Hello fellow Google Cloud Enthusiasts, welcome to this section where we'll look at different aspects of planning, deploying and managing workloads on GKE.



Effective workload management on Google Kubernetes Engine (GKE) is crucial for seamless performance, scalability, and resilience. In this section we'll look at how to strategically plan, deploy, and manage diverse workloads on GKE.



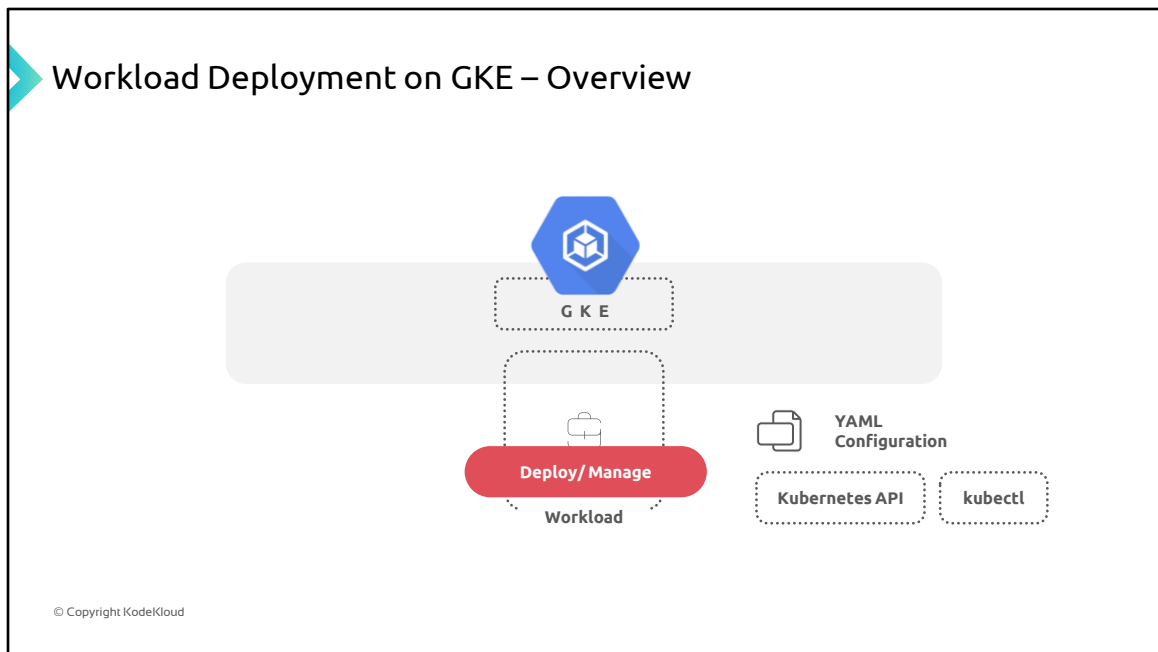
We'll embark on our journey by delving into the concepts of workload planning and deployment strategies. Gain a comprehensive overview of workload deployment options, including stateful and stateless applications, batch jobs, and daemons.



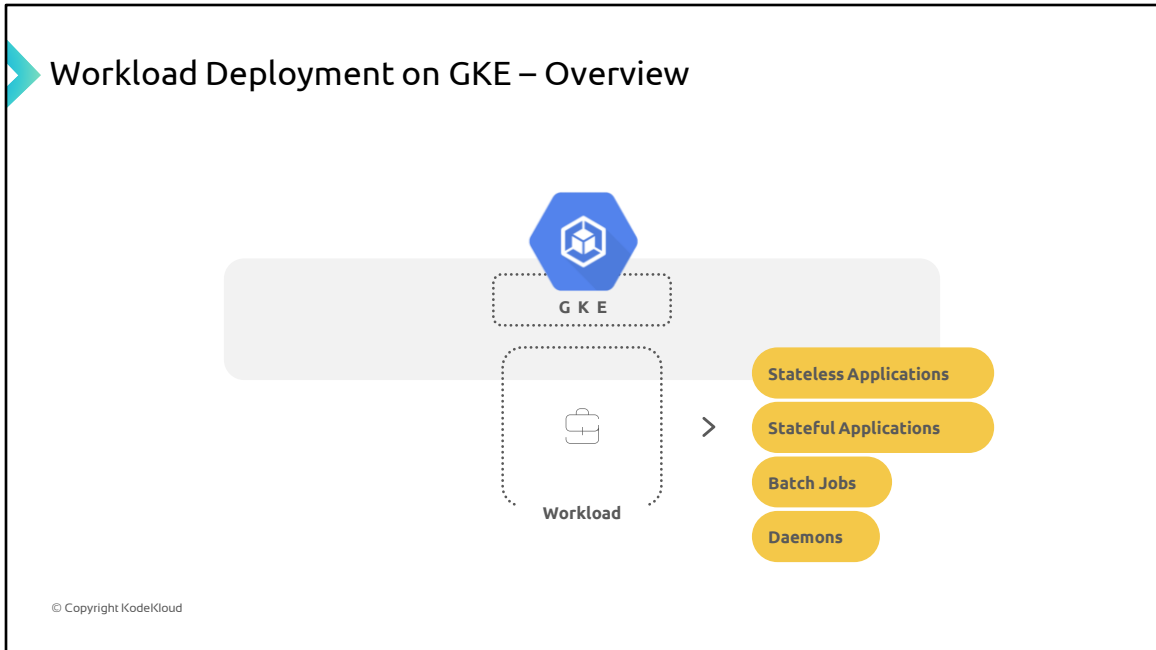
We'll then learn about the concept of **node taints**, their necessity, and the manifold benefits they bring to workload distribution and resource optimization followed by exploring the significance of rolling updates in GKE.

# Plan Workload Deployment on GKE

© Copyright KodeKloud



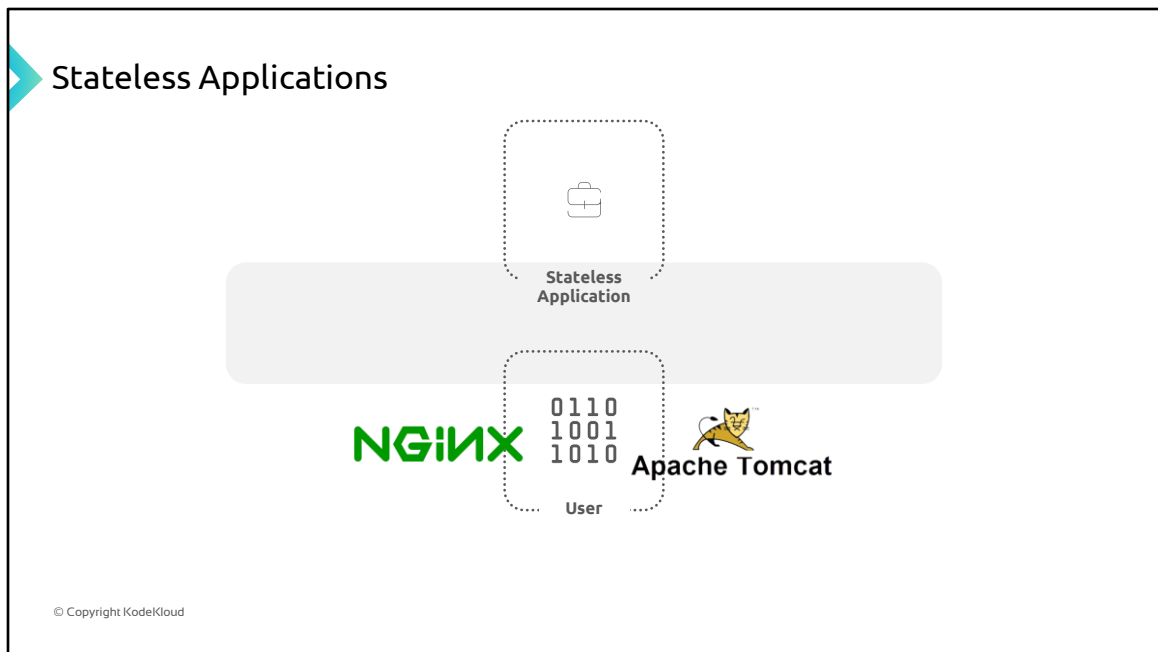
During this lecture, we will explore the broad outline of planning workload deployment and management on GKE. To deploy and manage containerized applications and other workloads on a GKE cluster, the Kubernetes system utilises controller objects. These objects are generated through the Kubernetes API or the kubectl command-line interface. Usually, you would construct a **YAML configuration** file to represent the desired Kubernetes controller object and then employ the file with the Kubernetes API or kubectl to facilitate deployment and management.



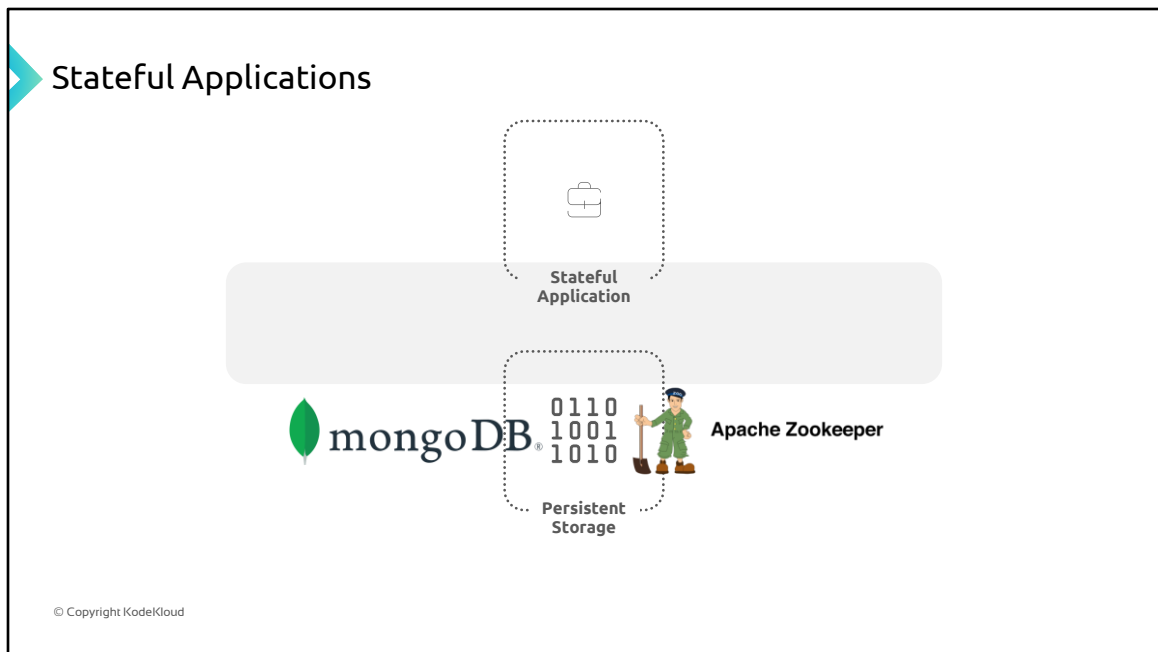
GKE supports a variety of workload types, for example:

- Stateless applications
- Stateful applications
- Batch jobs
- Daemons

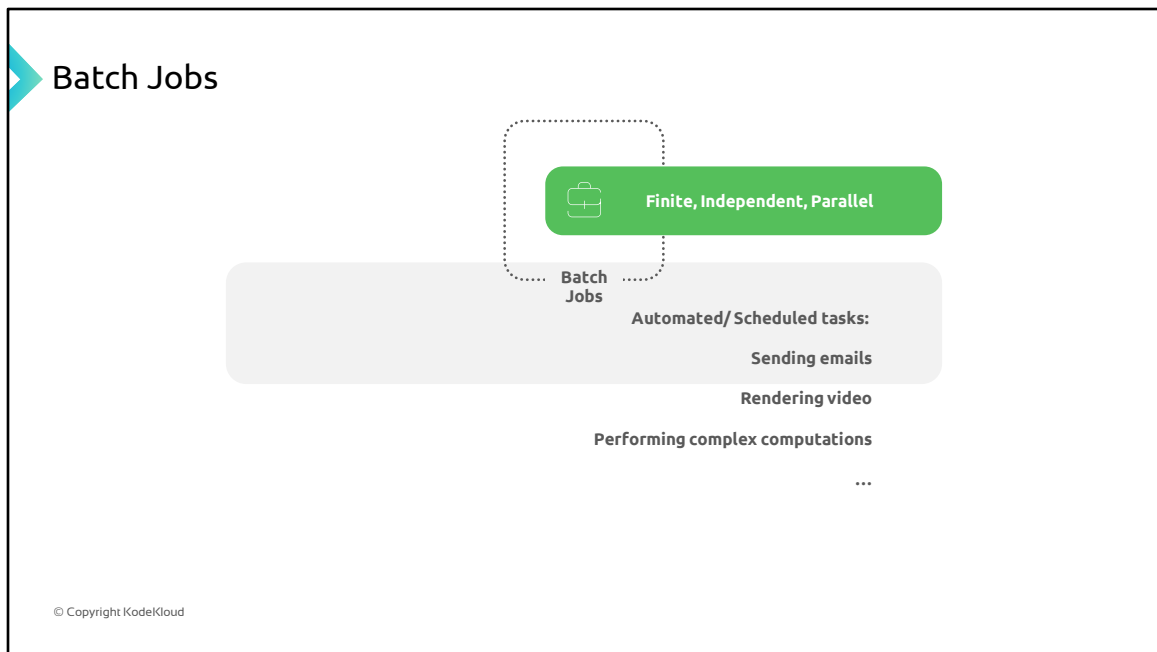




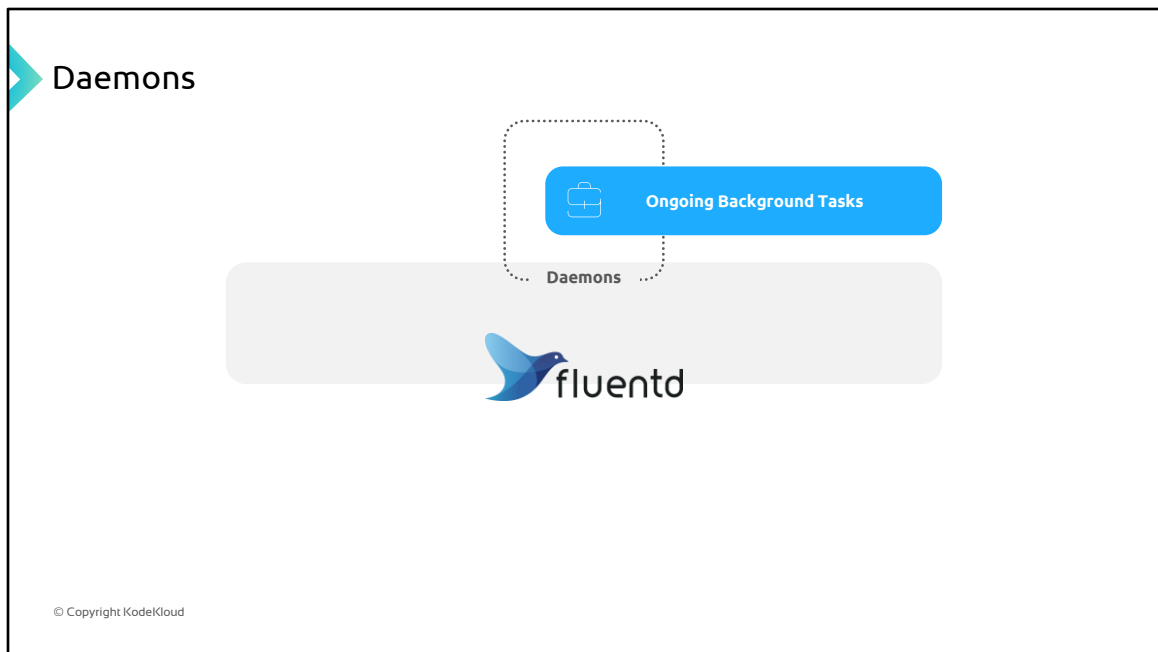
Stateless applications are designed in a way that they do not retain their state or store any data in a persistent storage. Instead, all user and session data remains with the client. Examples of stateless applications include web frontends like **Nginx**, web servers like **Apache Tomcat**, and various other web applications. To deploy a stateless application on a GKE cluster, typically a Kubernetes Deployment is created. Pods created by Deployments are not unique and do not preserve their state, which makes it easier to scale and update stateless applications.



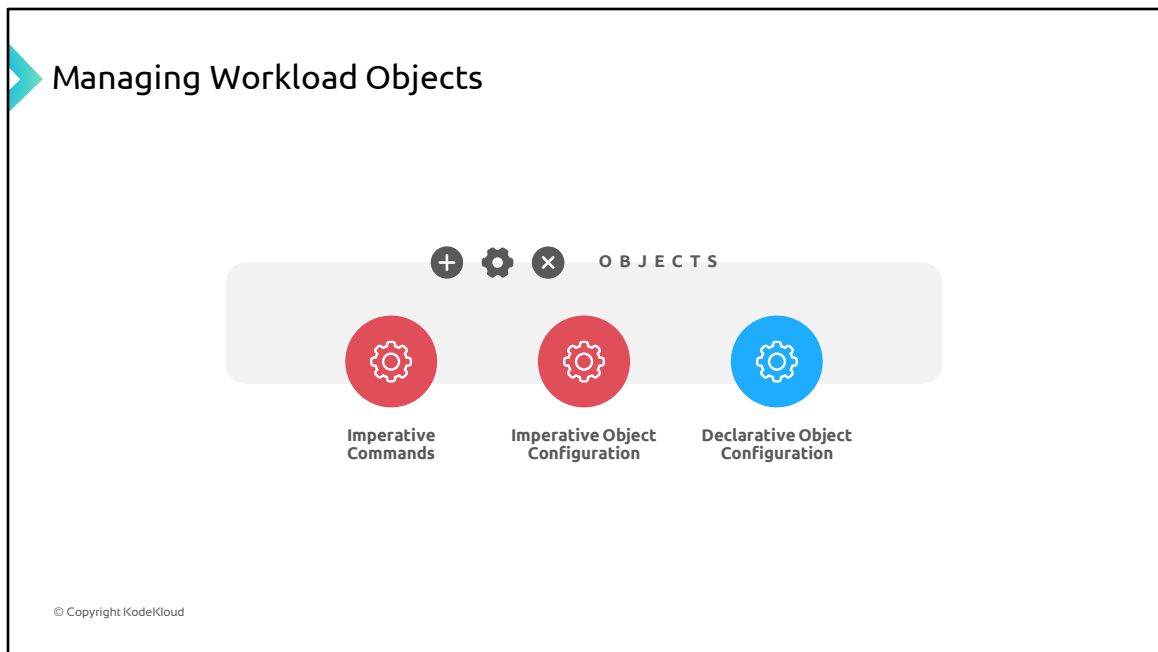
In contrast to stateless applications, stateful applications require their state to be persistent. These applications utilise persistent storage, such as persistent volumes, to store data that is used by the server or other users. Examples of stateful applications include databases like **MongoDB** and message queues like **Apache ZooKeeper**. To deploy a stateful application, a Kubernetes StatefulSet is created. Pods created by StatefulSets have unique identifiers and can be updated in a controlled and safe manner.



Batch jobs refer to tasks that are finite, independent, and often run in parallel until completion. Examples of batch jobs include automated or scheduled tasks like sending emails, rendering video, or performing complex computations. To execute and manage batch tasks on a GKE cluster, a Kubernetes Job is created. With a Job, you can specify the number of Pods that need to complete their tasks before considering the Job as complete, as well as the maximum number of Pods that should run simultaneously.



Daemons are responsible for performing ongoing background tasks on specific nodes without requiring user intervention. Examples of daemons include log collectors like Fluentd and monitoring services. To deploy a daemon on your cluster, you can use a Kubernetes DaemonSet. DaemonSets create one Pod per node, and you have the flexibility to choose a specific node to which the DaemonSet should be deployed.



Just like in a Kubernetes environment, objects in GKE can be created, managed, and deleted using imperative or declarative methods.

## Imperative commands

Imperative commands allow you to quickly create, view, update, and delete objects with `kubectl` command. These commands are useful for one-off tasks or for making changes to active objects in a cluster. Imperative commands are commonly used to operate on live, deployed objects on your cluster.

## Imperative object configuration

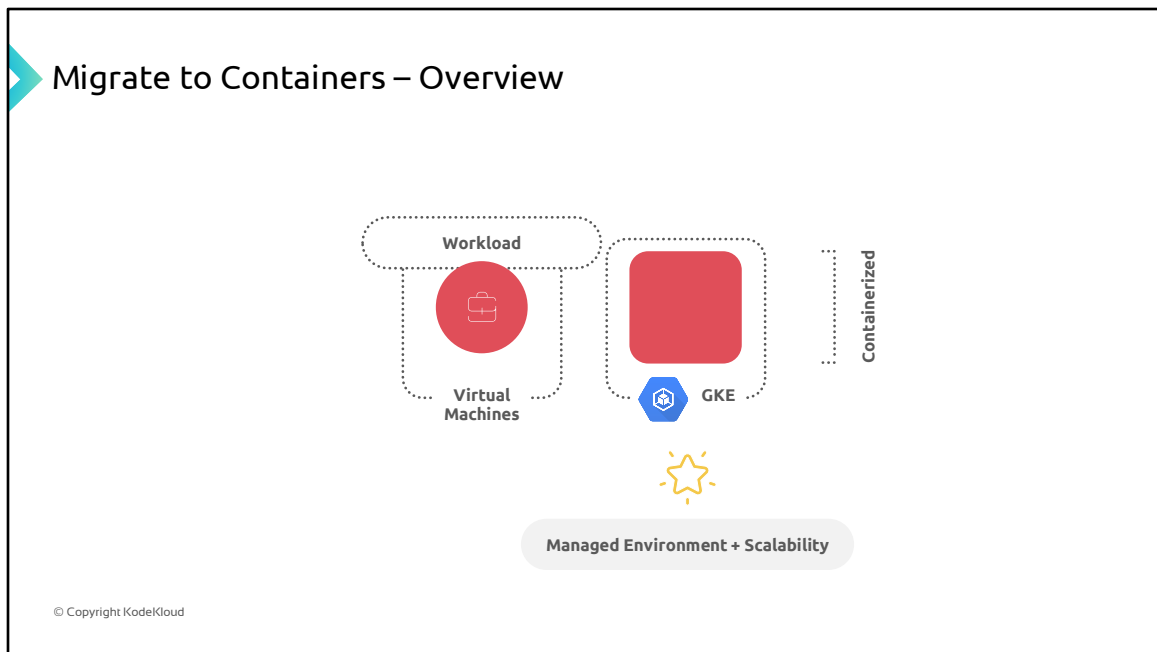
Imperative object configuration creates, updates, and deletes objects using configuration files containing fully defined object definitions. You can store object configuration files in source control systems and audit changes more easily than with imperative commands.

## **Declarative object configuration**

Declarative object configuration operates on locally stored configuration files but does not require explicit definition of the operations to be executed. Instead, operations are automatically detected per object by kubectl. This is useful if you are working with a directory of configuration files with many different operations. Declarative object management requires a strong understanding of object schemas and configuration files.

# Migrate Workloads to GKE

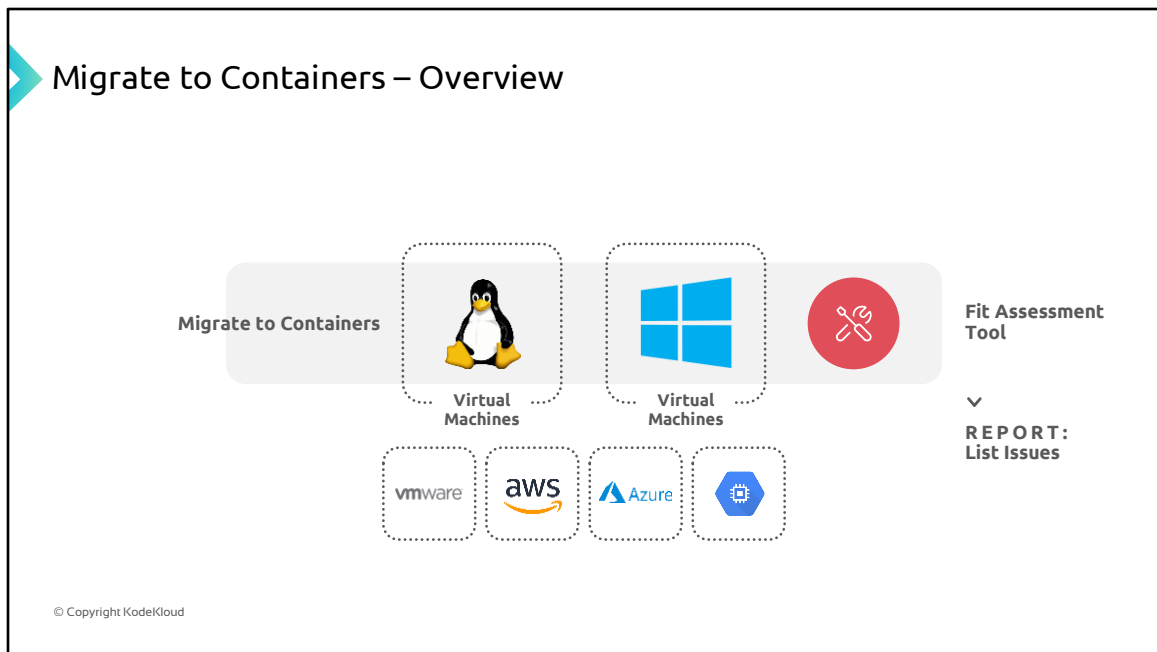
© Copyright KodeKloud



In this lecture, we take a quick look at the “migrate to containers” tool that’s available for migrating workloads that are currently running on virtual machines (VMs) to Google Kubernetes Engine (GKE) by containerizing the workload.

It helps in containerizing the VM-based applications and runs them on GKE, benefiting from the managed environment and scalability provided by Google Cloud infrastructure.

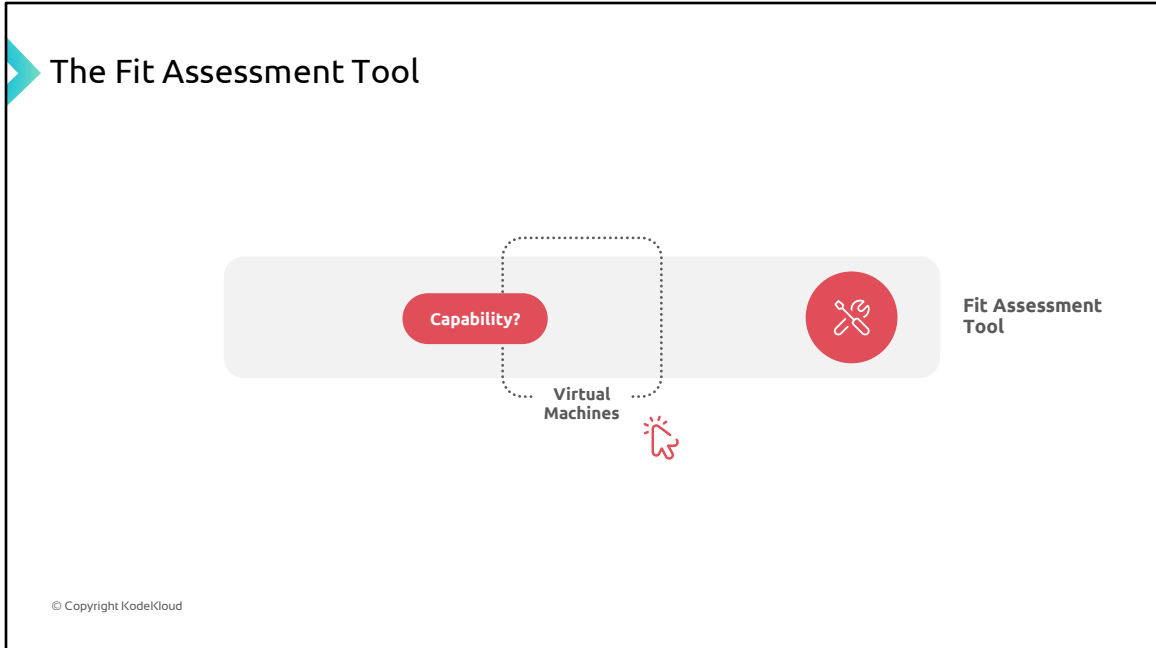




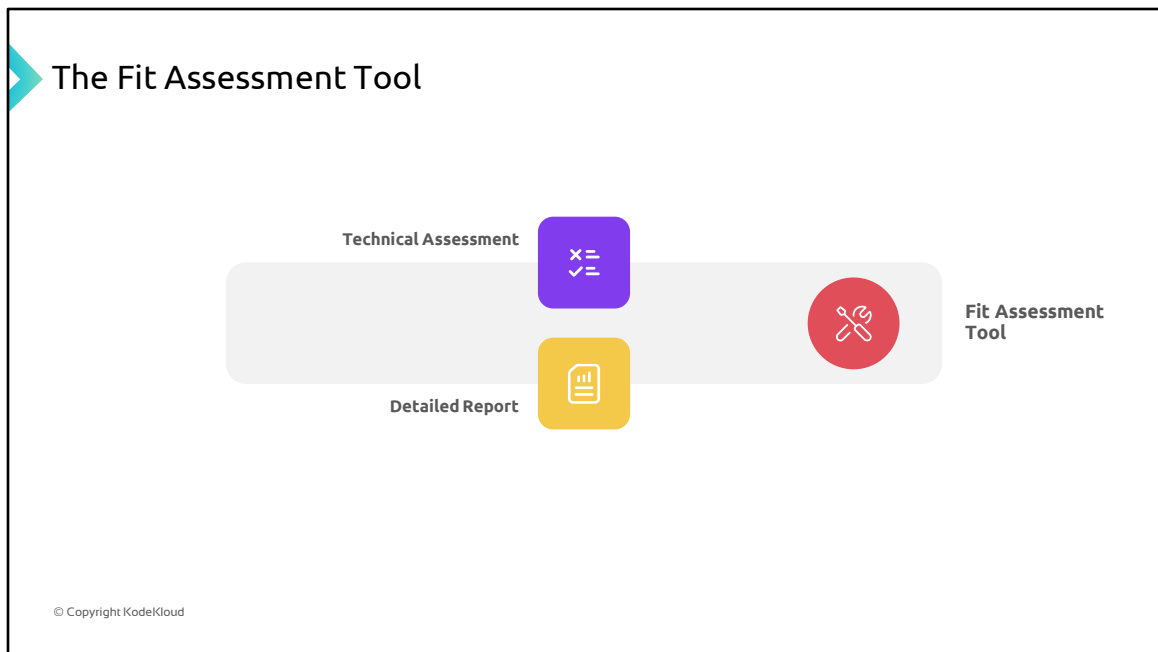
Migrate to containers supports containerizing both Linux and Windows VMs that are running on various platforms such as VMware, AWS, Azure, or Google Cloud Compute Engine. It also offers a tool known as “the fit assessment tool” that helps assess the compatibility of workloads for migration to containers.

To use Migrate to containers, simply run the fit assessment tool against the existing VMs. The tool will then analyse source VMs and generate a report that describes the fit of your applications for migration to a container. The report will also include

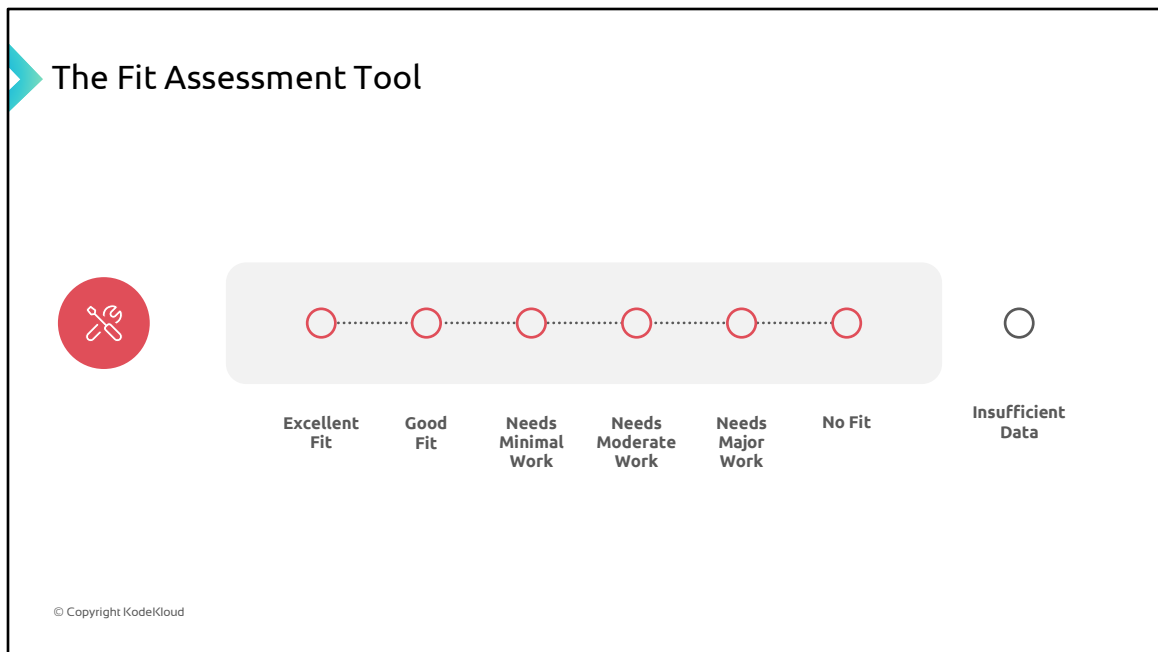
a list of any issues that need to be resolved before migration.



The fit assessment tool in Migrate to Containers helps determine the compatibility of a virtual machine (VM) workload for modernization to a container or migration to Compute Engine.



This tool provides **technical assessments** of workloads, allowing us to evaluate the feasibility of modernization and identify any technical obstacles that may need to be addressed. The fit assessment generates a **detailed report** that highlights potential issues and provides an overall fit assessment score.

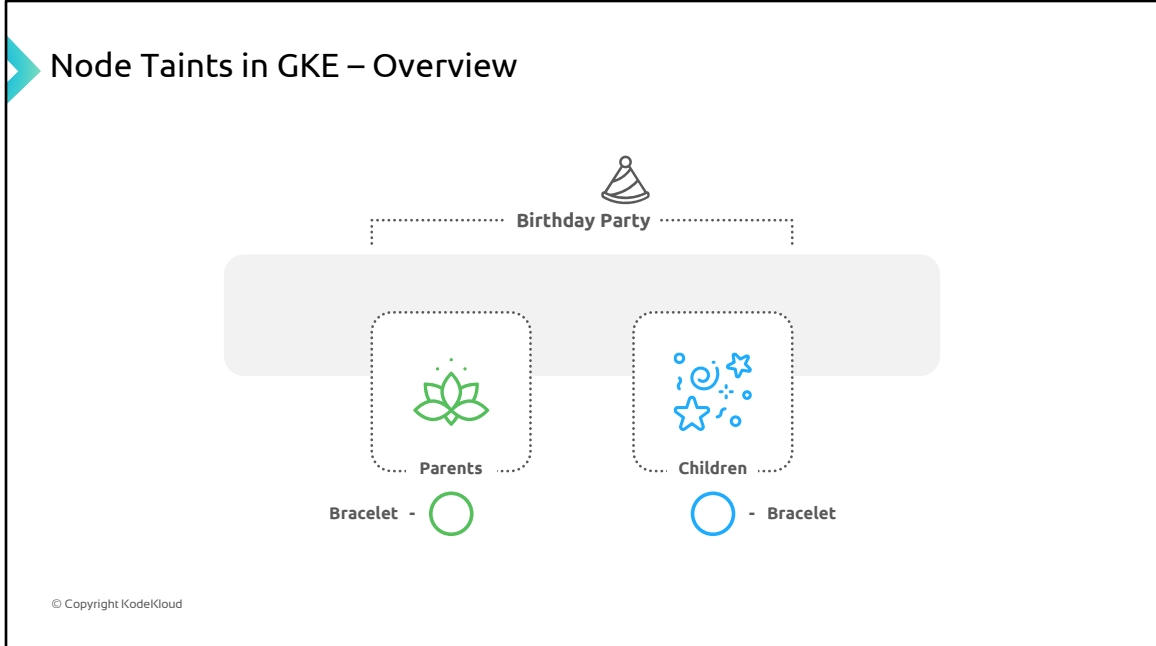


It also provides an overall fit assessment using one of the following scores:

- Excellent fit
- Good fit, with some findings that might require attention
- Needs minimal work before migrating
- Needs moderate work before migrating
- Needs major work before migrating
- No fit
- Insufficient data

# Control Workload Deployments Using Node Taints

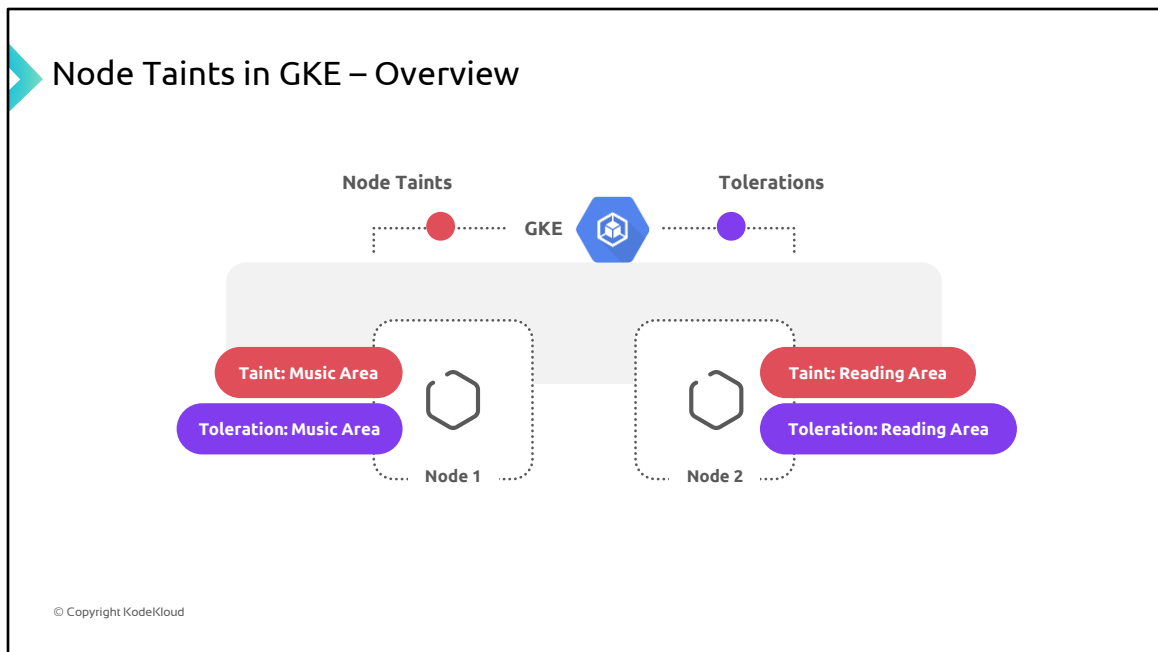
© Copyright KodeKloud



Let's imagine you're hosting a birthday party for your child, and you want to create separate spaces for different groups of guests: one area for parents and another for kids. The kids are eager to have a lively time at the party, engaging in games and dancing to music. Meanwhile, the parents prefer a calm atmosphere where they can enjoy some peaceful music or perhaps read books. Your goal is to ensure that the kids gather in an energetic space while the parents have a peaceful area. To accomplish this, you introduce a system of colored bracelets: blue for the kids and green for the parents.





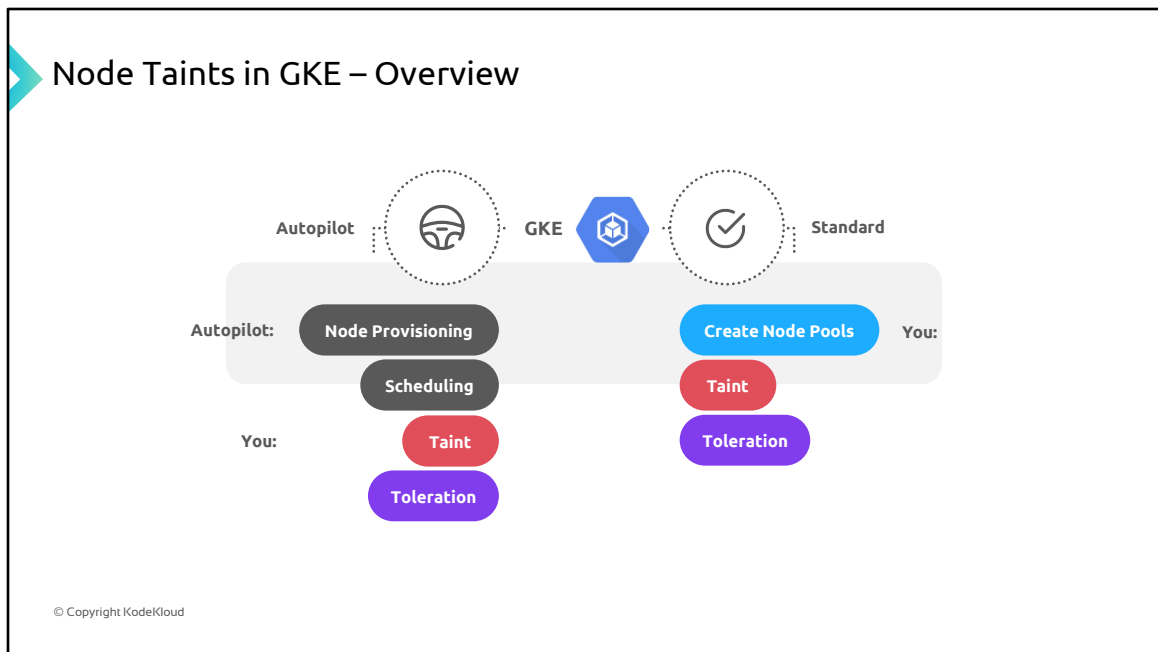


In the context of Google Kubernetes Engine (GKE), workload separation is like organizing your party. GKE allows you to control where your application's components, called Pods, are scheduled on the cluster's nodes. It uses a similar concept called "node taints" and "tolerations."

In our party analogy, the nodes represent different areas in your backyard, and each node has a specific characteristic or "taint." For example, one node may have a taint that represents the music area, while another node may have a taint for the

reading area.

To ensure that the Pods are scheduled in the desired areas, you specify a "toleration" for each Pod. The toleration corresponds to the colored bracelets worn by your guests. The Pods with a toleration for the music area taint will exclusively be scheduled on nodes with that taint, just as the kids gather in their designated area. Similarly, Pods with a toleration for the reading area taint will be scheduled on nodes with that taint, creating a quiet space for the parents.



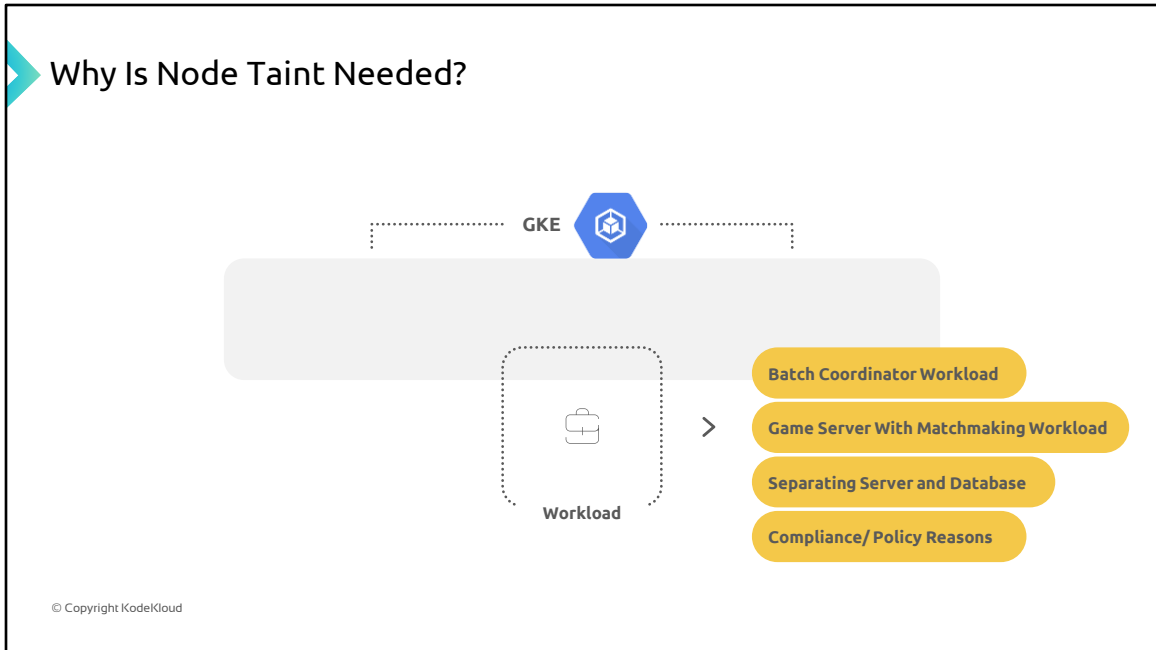
Depending on your GKE cluster configuration, there are different ways to configure workload separation.

For example:

- In an "Autopilot" setup, GKE automatically handles the node provisioning and scheduling for you. You add a toleration and specify the taint using a nodeSelector. It's like telling GKE to create the appropriate nodes and assign the Pods accordingly.
- In a "Standard" setup without auto-provisioning, you manually create node pools with specific taints

and labels. You then add tolerations to the Pods to specify their preferences. It's like manually setting up the different areas in your backyard and assigning guests based on their preferences.

By using workload separation with node taints and tolerations, you can control where your Pods are scheduled, ensuring that different workloads are kept separate and placed in specific locations based on their requirements or characteristics.



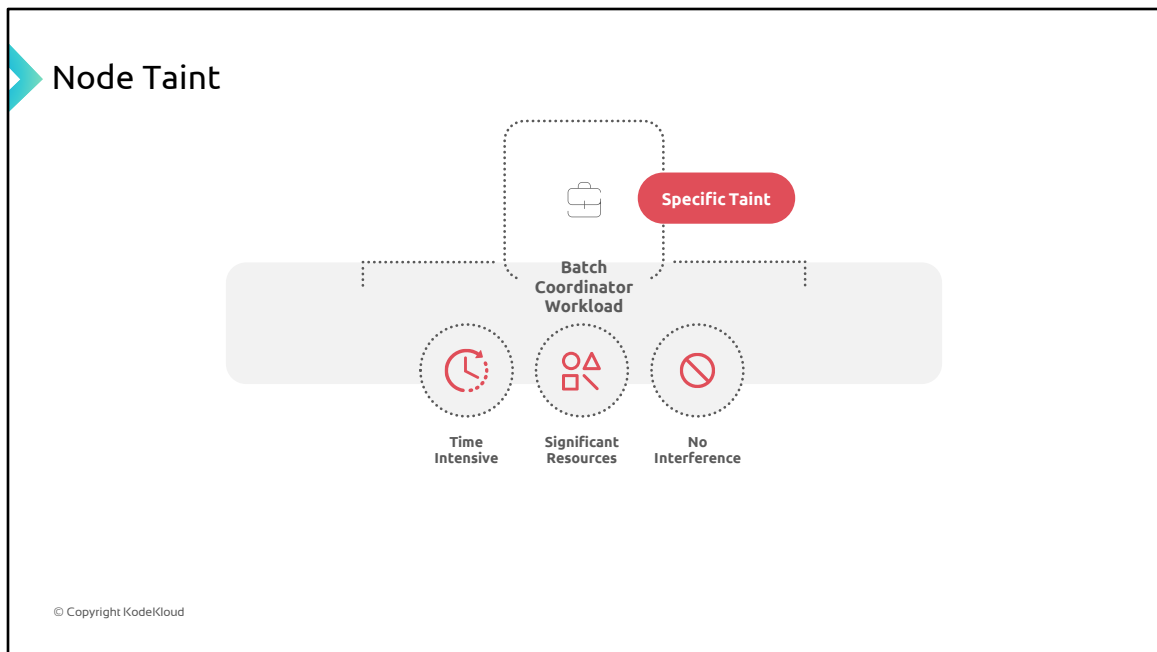
By applying a node taint, you can ensure that certain workloads or Pods are scheduled only on nodes that meet specific criteria or have specific characteristics. This is important because different workloads may have different requirements, dependencies, or sensitivities. For example:

**Batch coordinator workload**

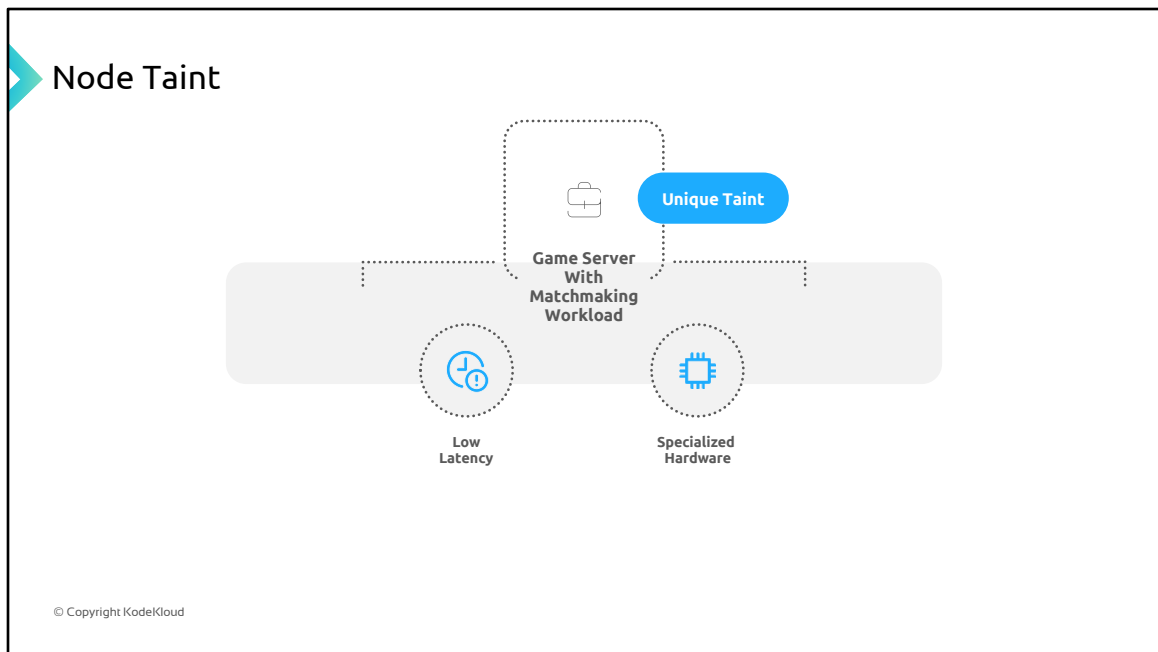
**Game server with matchmaking workload**

**Separating server and database**

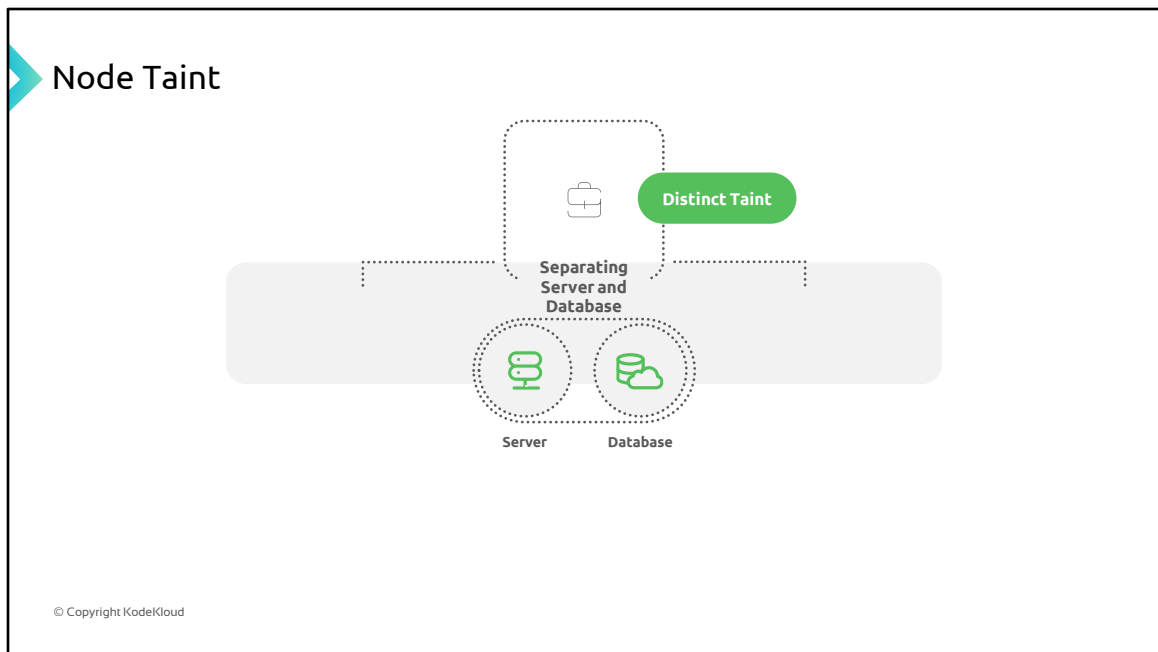
**Compliance or policy reasons**



1. **Batch coordinator workload:** Consider the scenario where you have a batch coordinator workload **responsible for running time-intensive jobs**. These jobs require **significant resources** and **should not interfere** with other workloads. By assigning a specific taint to nodes that can handle such jobs efficiently, you ensure that the batch coordinator workload is scheduled only on those nodes, separate from other workloads.

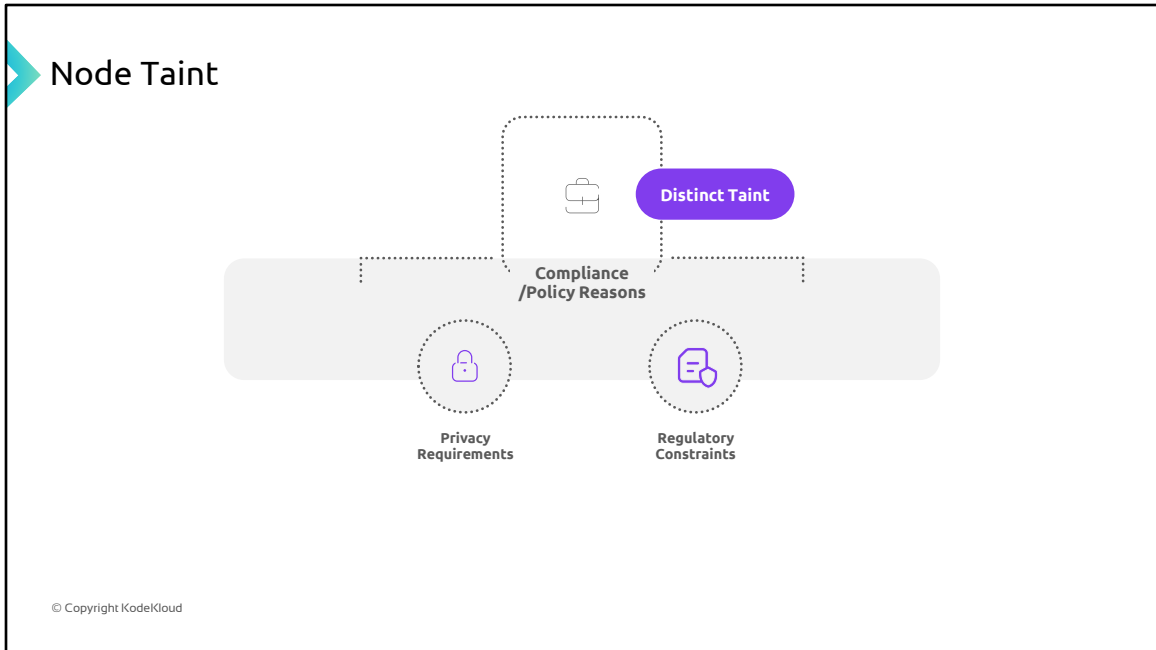


**Game server with matchmaking workload:** In the context of a game server, you may have a matchmaking workload that requires **low latency** and specialized **hardware**. To optimize its performance, you can assign a unique taint to nodes with the necessary hardware capabilities. This ensures that the matchmaking workload is isolated and runs only on nodes that meet its requirements, separate from other session Pods.

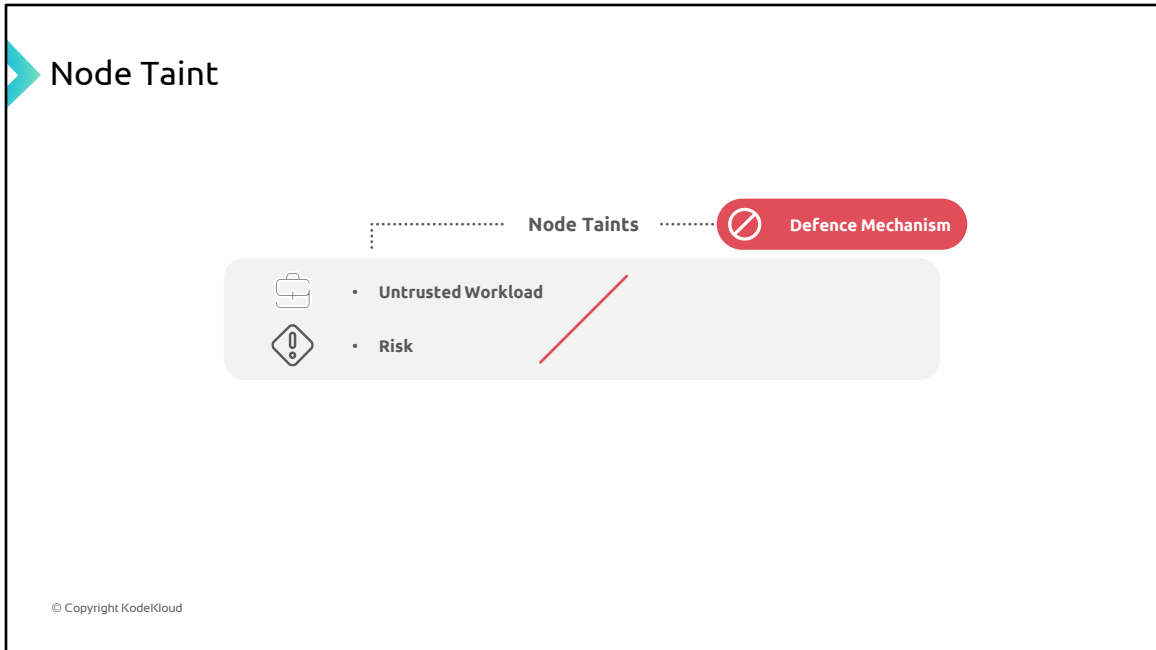


**1. Separating server and database:** In a typical application stack, you might have a **server component** that interacts with a **database**. To improve performance, security, or scalability, you may want to separate the server and database workloads onto different nodes. By assigning distinct taints to nodes for server and database workloads, you ensure they are scheduled separately and don't interfere with each other's resources or dependencies.

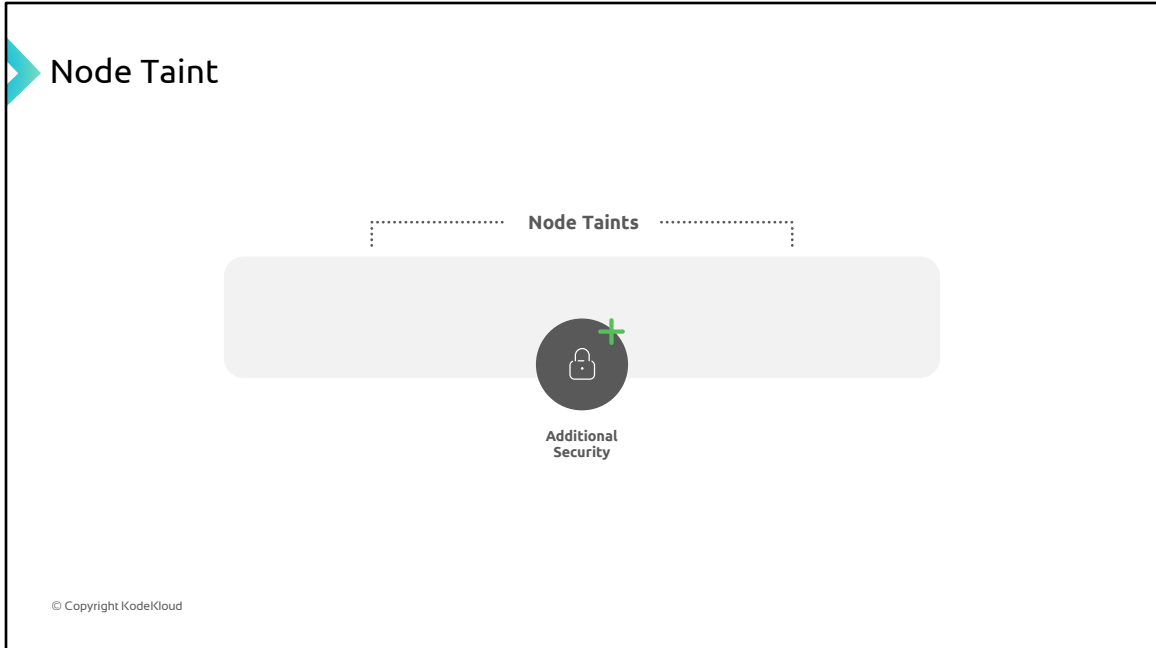




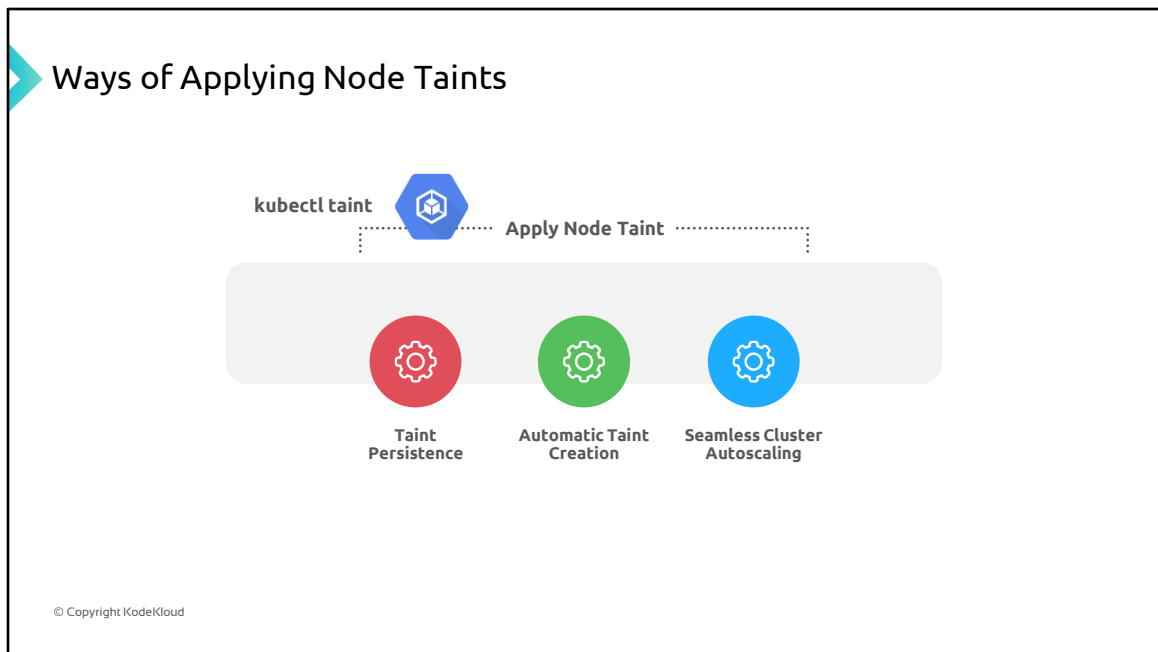
**1.Compliance or policy reasons:** Workload separation can also be enforced for compliance or policy reasons. For example, certain workloads might have specific data **privacy requirements** or **regulatory constraints** that necessitate their isolation from other workloads. By applying unique taints to nodes that meet the compliance criteria, you can ensure that the respective workloads are scheduled separately and adhere to the necessary policies.



It's important to note that workload separation using node taints should not be considered a **primary security boundary** or used as a defense mechanism against untrusted workloads. It is not intended to isolate untrusted workloads or mitigate all possible security risks.



For security purposes, additional measures should be taken. Workload separation with node taints primarily serves to organize and optimize workload placement based on their requirements and characteristics within the cluster.



Node taints can either be applied to clusters and nodes in GKE or by using the [kubectl taint](#) command to a specific node. When it comes to setting node taints in Google Kubernetes Engine (GKE), choosing to specify them within GKE itself offers several advantages over using the "kubectl taint" command, such as:

**1. Taint Persistence:** Node taints specified in GKE are preserved even when a node is restarted or replaced. This means that the taints remain intact and continue to influence the scheduling of Pods on

those nodes, ensuring consistent behavior over time.

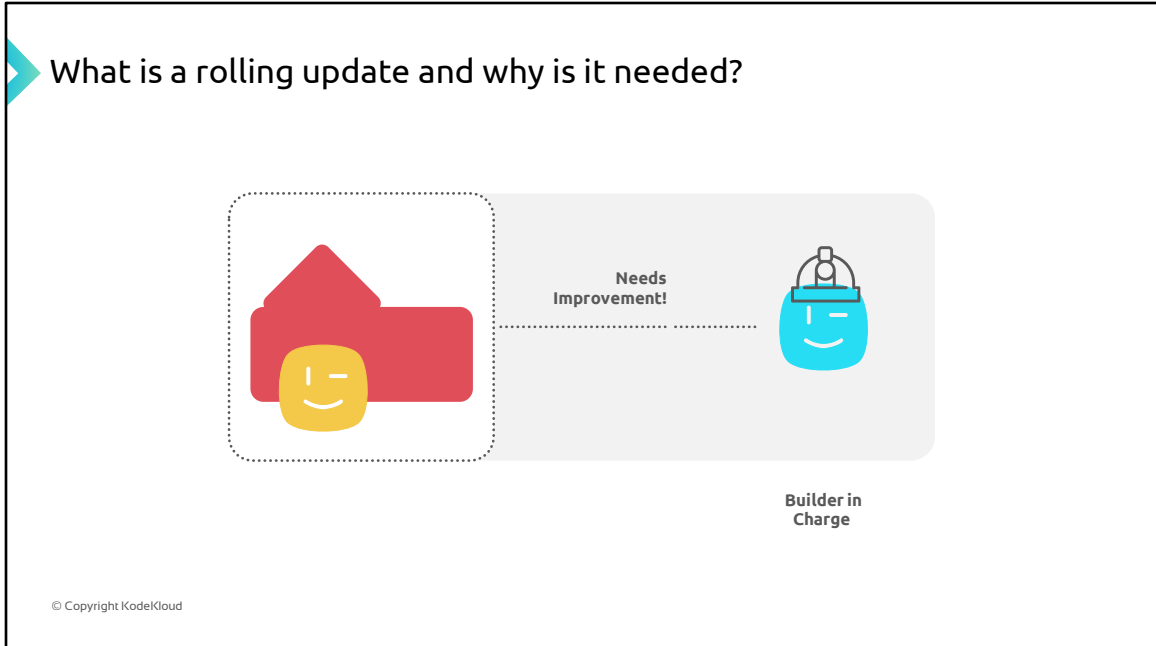
**2. Automatic Taint Creation:** When nodes are added to a node pool or cluster in GKE, taints are automatically created for them. This eliminates the need for manual intervention and ensures that the specified taints are immediately applied to the newly added nodes.

**3. Seamless Cluster Autoscaling:** GKE seamlessly handles the creation of taints during cluster autoscaling events. As the cluster scales up or down based on workload demands, the corresponding taints are automatically created or removed on the newly added or removed nodes, maintaining the desired taint configuration.

It's worth noting that if you manually remove a taint from a node within a node pool using the "kubectl" command, GKE does not retain or preserve the taint after the node undergoes a restart. Therefore, to ensure consistent behavior, it's recommended to manage taints through GKE's native mechanisms rather than making manual modifications with "kubectl" directly.

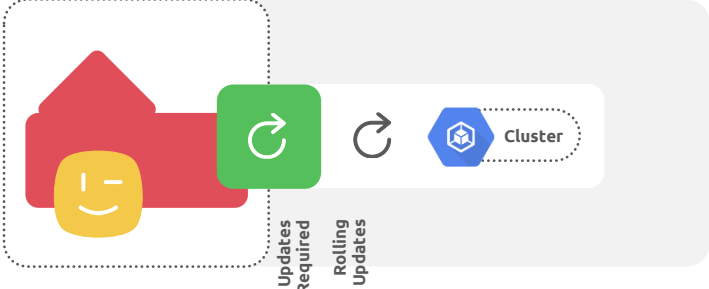
# Performing Rolling Updates on a GKE Cluster

© Copyright KodeKloud



Let's consider our builder analogy again to understand rolling updates and why it might be necessary. Imagine you have been living in the house for a few years now and want to do some improvements in the house. You have appointed a builder who is in charge of renovating the house.

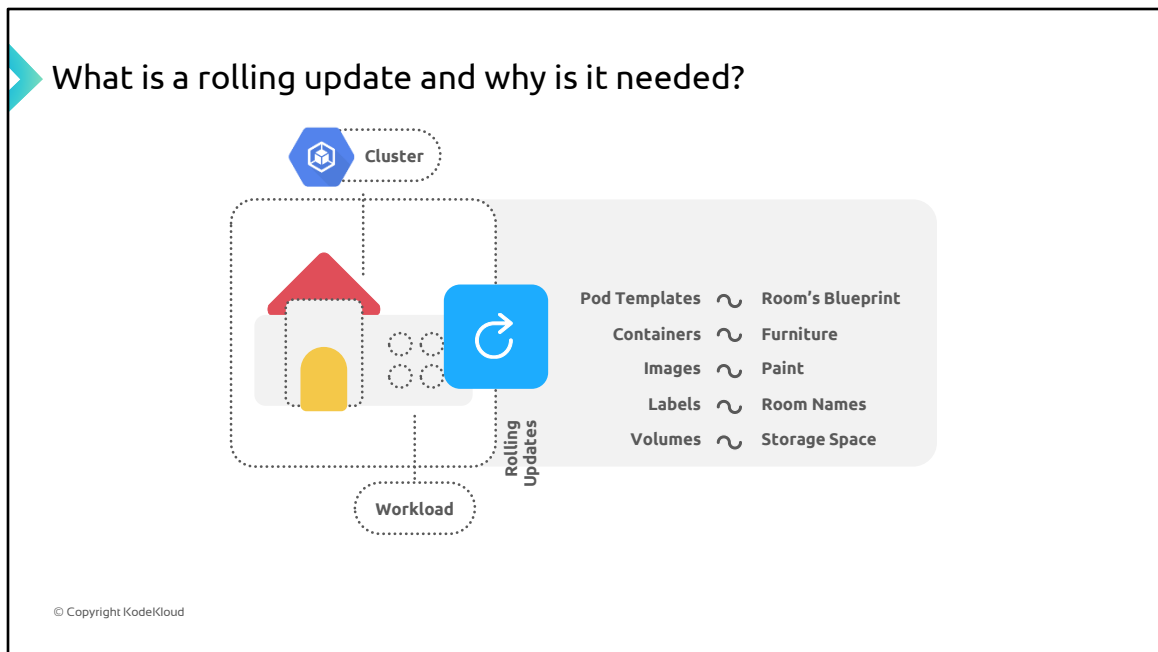
▶ What is a rolling update and why is it needed?



© Copyright KodeKloud

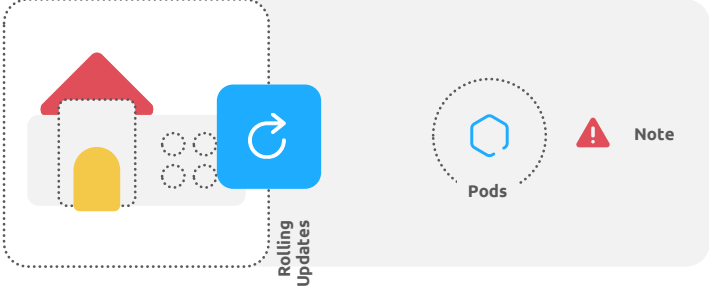
You ask the builder to update various aspects of the house, such as the paint color, furniture, and appliances, without causing any disruptions to the people living there. In the same way, performing a rolling update in a Google Kubernetes Engine (GKE) cluster allows you to update different elements of your applications without causing downtime.





In this analogy, the house represents your cluster, and the various components inside the house represent the workloads in your cluster, such as DaemonSets, Deployments, and StatefulSets. To perform a rolling update, you focus on updating the "**Pod template**" of these workloads, which is like modifying the blueprint of each room in the house. The Pod template includes specifications for **containers** (like furniture), **images** (like paint color), **labels** (like room names), and **volumes** (like storage spaces) in different rooms.

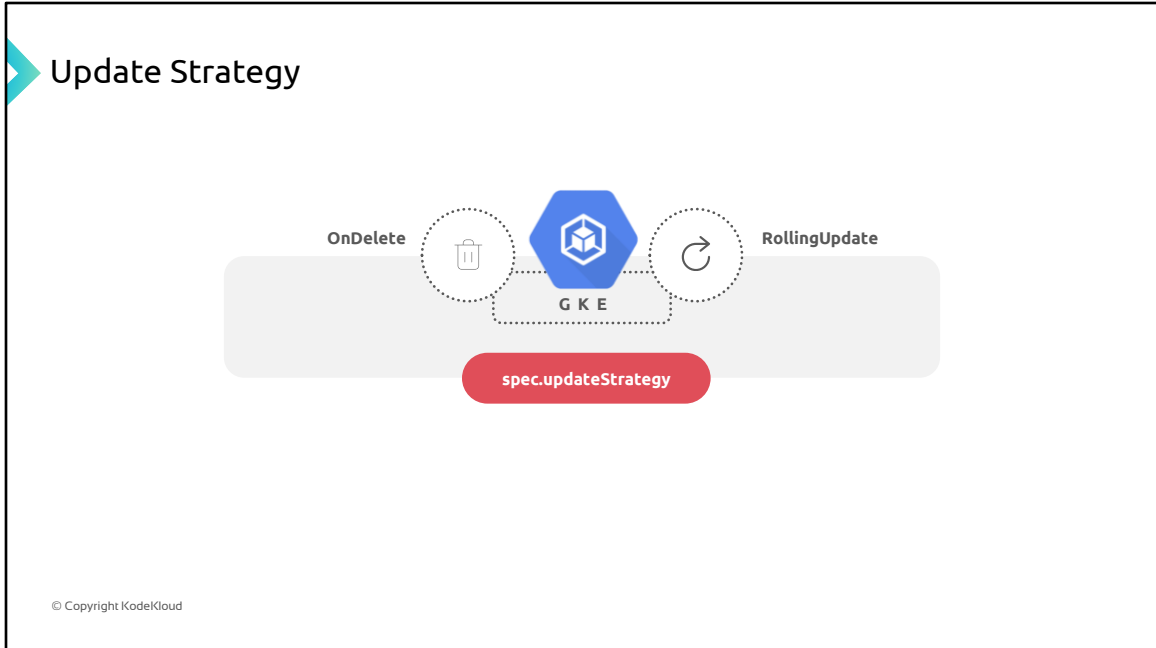
▶ What is a rolling update and why is it needed?



© Copyright KodeKloud

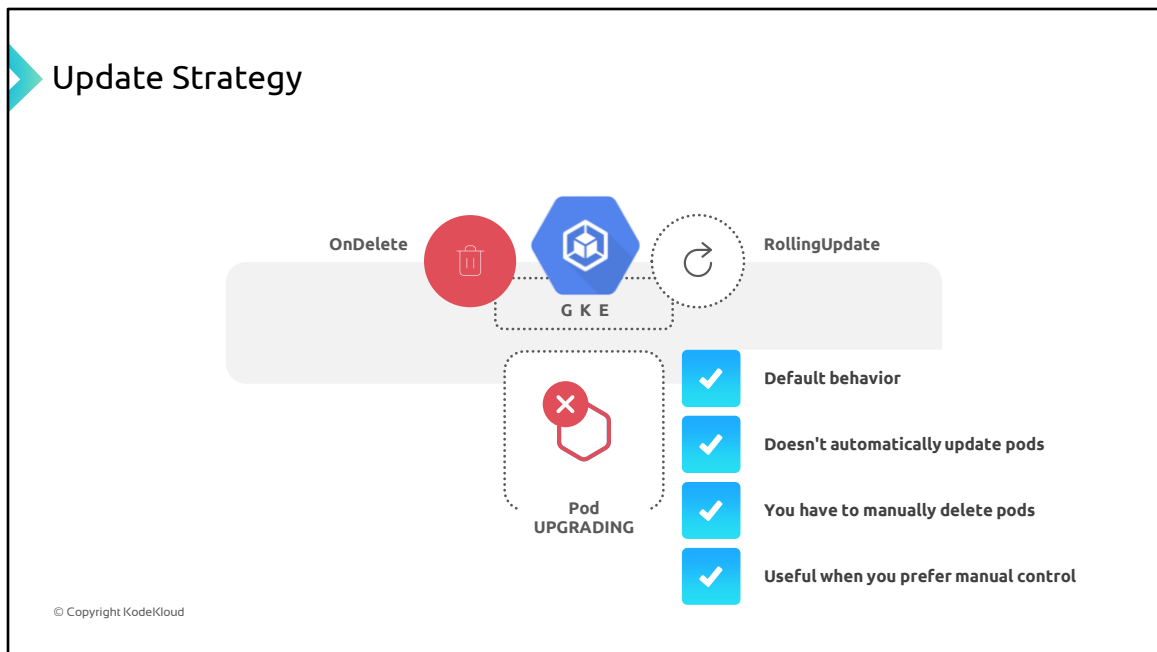
By updating the Pod template, you initiate a rolling update process that gradually replaces the existing Pods with new ones. Just like you would update one room at a time in the house, GKE incrementally replaces Pods in a controlled manner, ensuring that there are always enough available resources on the nodes.

It's important to note that scaling a resource or updating fields outside of the Pod template does not trigger a rolling update.



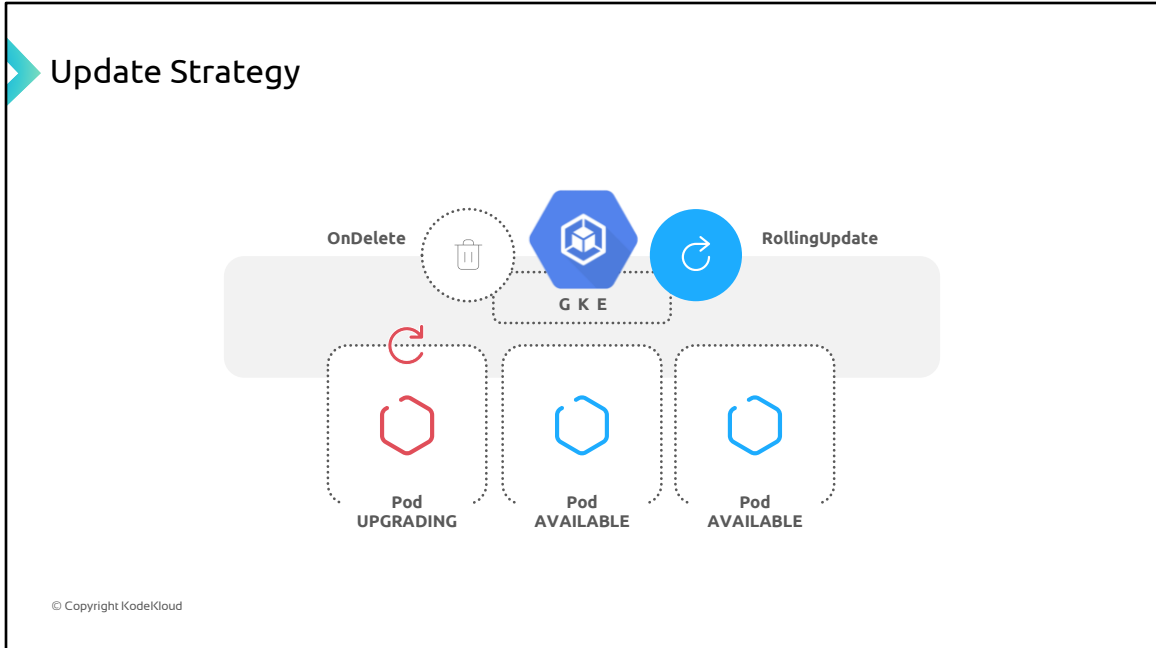
Similar to the Kubernetes platform, Google Kubernetes Engine (GKE) Update strategy is a concept that allows you to configure how your Pods are updated. You can use the update strategy to control whether your Pods are updated automatically or manually, and how many Pods are updated at a time. It is specified using the `spec.updateStrategy` field.

There are two types of update strategies available: OnDelete and RollingUpdate.

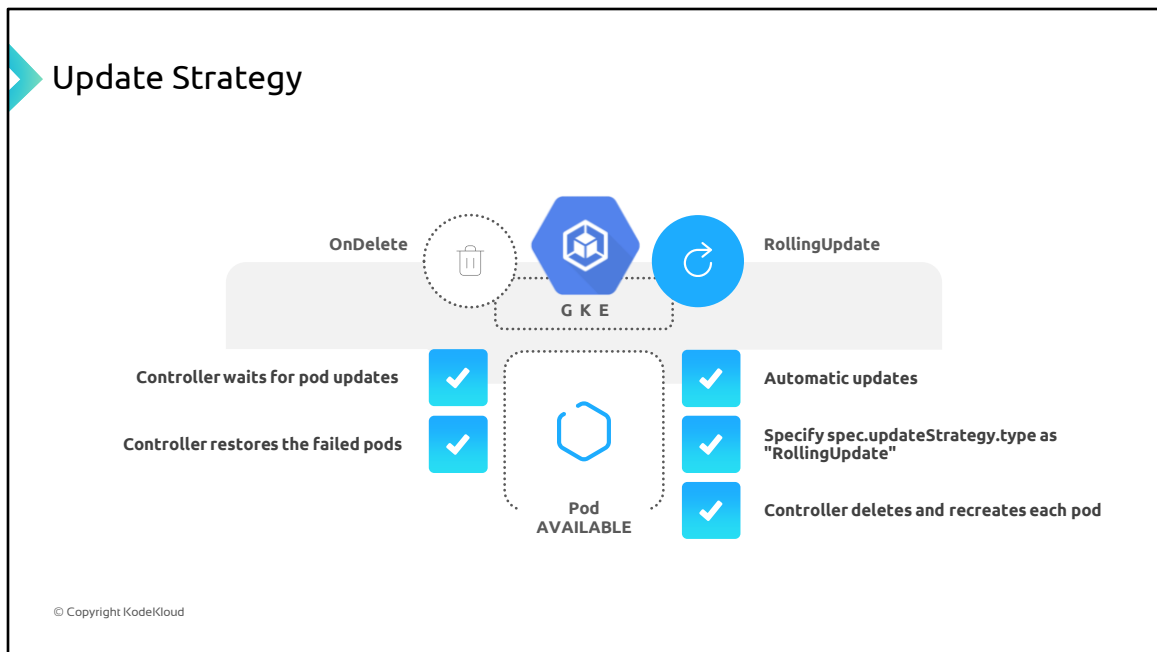


**1. OnDelete Strategy:** OnDelete is a type of update strategy that allows you to update your Pods manually. This means that you will need to delete your Pods one at a time and then create new Pods with the updated configuration.

- This is the default behaviour if `spec.updateStrategy.type` is not specified.
- With this strategy, the controller does not automatically update the Pods.
- If you want to apply changes and update the Pods, you need to manually delete the existing Pods. Once deleted, the controller will create new Pods that reflect the changes you made.
- This strategy is useful when you prefer to have manual control over updating the Pods.



1. RollingUpdate Strategy: RollingUpdate is a type of update strategy that allows you to update your Pods one at a time. This means that your Pods will continue to be available while the update is being performed.



- This strategy enables automated rolling updates for Pods in StatefulSets or DaemonSets.
- To apply the RollingUpdate strategy, you need to specify `spec.updateStrategy.type` as "RollingUpdate".
- With RollingUpdate, the controller deletes and recreates each Pod, one at a time, in a controlled and sequential manner.
- The controller waits for each updated Pod to be running and ready before updating its predecessor.
- If a Pod fails during the update process, the controller restores the failed Pod to its current version. Pods that have already been updated are restored to the updated version, while Pods that have not received the update yet are restored to the previous version.