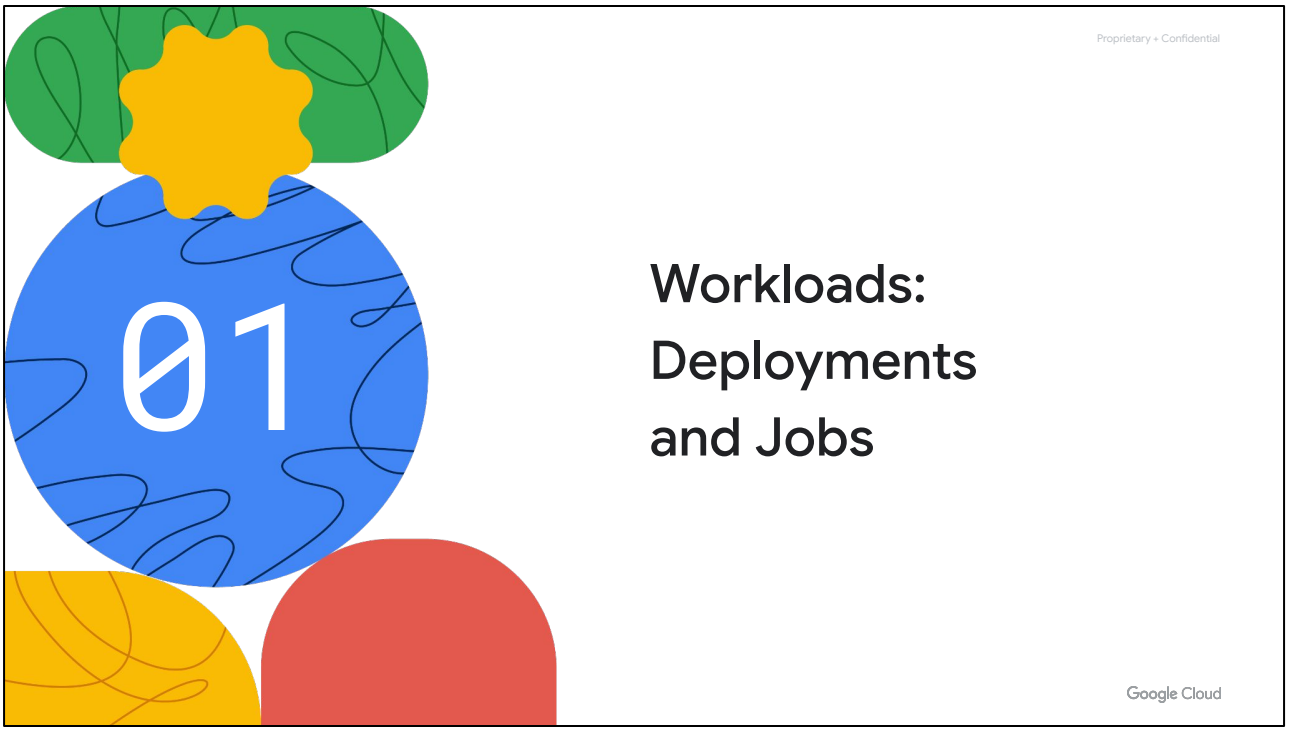


Google Kubernetes Engine, or GKE, leverages containerized applications–which are packaged units with all dependencies included–for efficient deployment and management.

In both GKE and Kubernetes, containers are called **workloads**. The concept of workloads in GKE streamlines the management of both applications and batch jobs. This becomes even more efficient with the help of Kubernetes' declarative approach.

Because Kubernetes is declarative, you define your cluster's desired state, and GKE's orchestration engine will then automatically manage the cluster to match that state.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

So, in the first section of this course titled, “Workloads: Deployments and Jobs,” you’ll learn to:

- Configure, manage, and update Deployment.
- Define Jobs and CronJobs in GKE and explore relevant use cases.
- Scale clusters manually and automatically.
- Control Pod placement with labels and affinity rules.
- Control Pod placement with taints and tolerations.
- And get software into a cluster.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

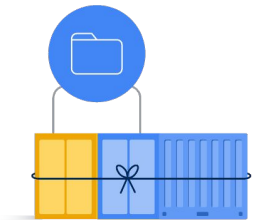
04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Let’s begin with Deployments.

Pods, Deployments, and controllers



Pods

Tightly coupled containers that share resources.

Deployment

A Kubernetes resource that describes the desired state of Pods.

Controller

A process that ensures that the desired state of objects match the observed state.

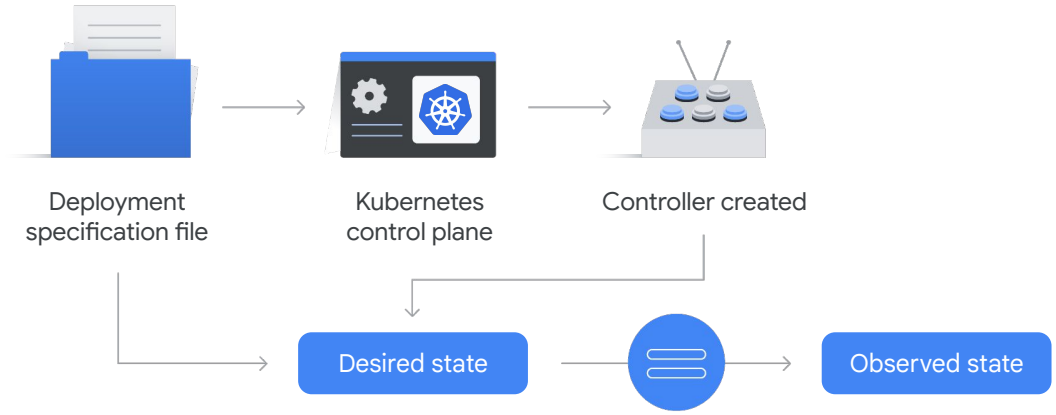
But before defining what a deployment is, it's important to understand what Pods are.

Pods are the smallest deployable units in Kubernetes. They consist of one or more containers that are tightly coupled and share resources like storage and networking. Pods are ephemeral, which means that they can be created and destroyed as needed.

A **Deployment** is a Kubernetes resource that describes the desired state of Pods. Deployments are managed by a built-in controller.

A **controller** is a continuously running process that monitors the state of an object–or set of objects–running on the cluster, and takes action to ensure that the desired state matches the observed state.

The deployment process

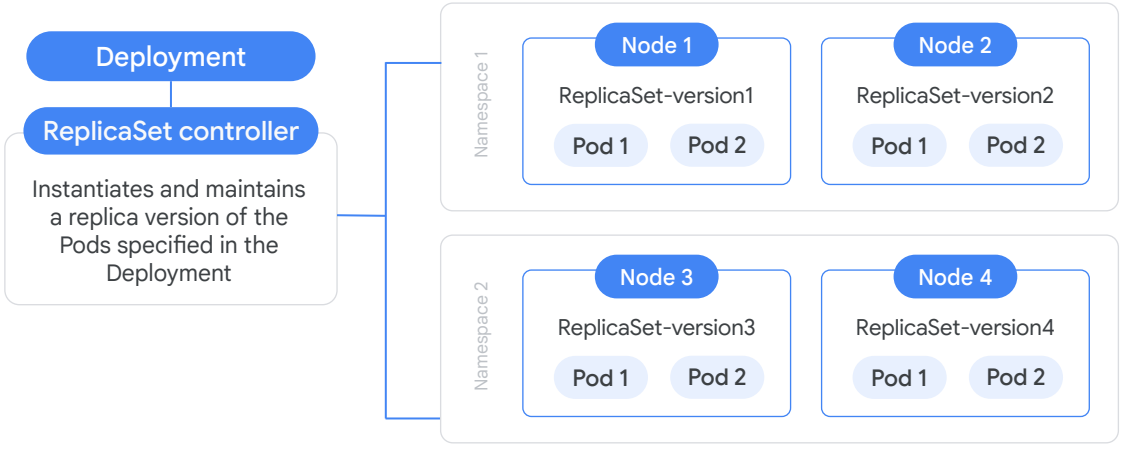


Now that we have defined some crucial terminology, let's describe the deployment process, at a high level.

The desired state of a Pod, including characteristics and details about how they should run and handle lifecycle events, is described in the **Deployment specification** file.

After this file is submitted to the Kubernetes control plane, a Deployment controller is created. The controller is responsible for converting the desired state into the current state, and maintaining that desired state over time.

The Deployment creates and configures a ReplicaSet



During this process, the Deployment also creates and configures a ReplicaSet. The ReplicaSet controller instantiates and maintains a replica version of the Pods specified in the Deployment.

Deployment object file details

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: my-app
spec:
   replicas: 3
   template:
      metadata:
        labels:
          app: my-app
      spec:
         containers:
         - name: my-app
           image: gcr.io/demo/my-app:1.0
           ports:
           - containerPort: 8080
```

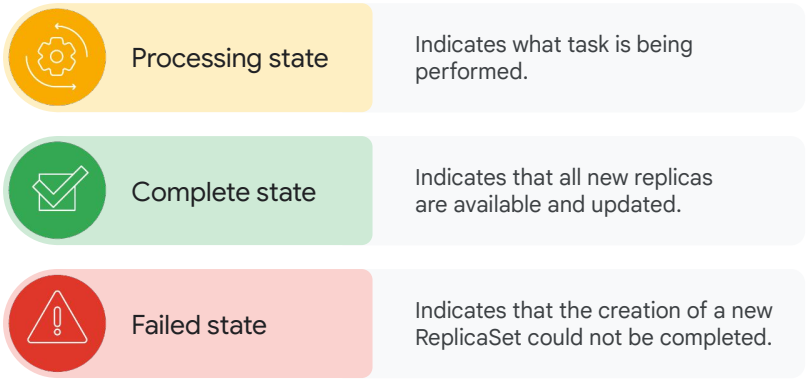
Written in YAML and typically identifies:

- The API version
- The kind, which in this case is a Deployment
- The name of the Deployment
- The number of Pod replicas
- A Pod template, which defines the metadata and specifications of each of the Pods in the ReplicaSet
- A container image
- A specific port to expose and accept traffic for the container

So, what details are included within a Deployment? A Deployment object file is written in a YAML format and typically identifies:

- The API version
- The kind, which in this case is a Deployment.
- The name of the Deployment
- The number of Pod replicas
- A Pod template, which defines the metadata and specifications of each of the Pods in the ReplicaSet
- A container image
- And a specific port to expose and accept traffic for the container

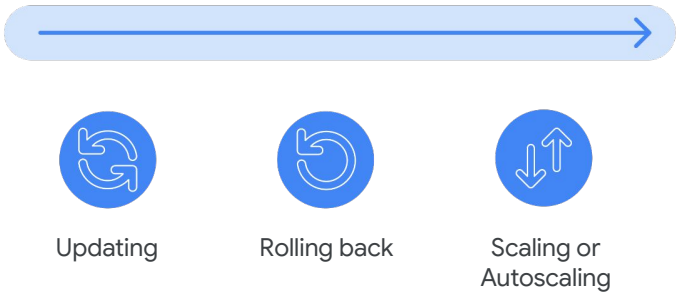
Deployment lifecycle states



Deployments exist in three distinct lifecycle states: Processing, complete, and failed.

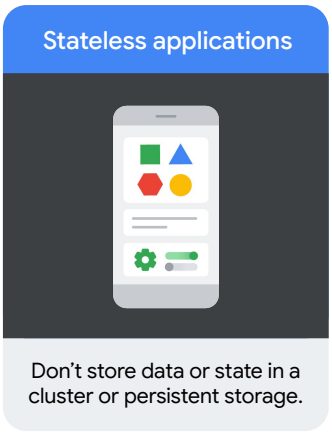
- The **processing** state indicates what task is being performed. For example the Deployment may be creating a new ReplicaSet, or scaling a ReplicaSet up or down.
- The **complete** state indicates that all new replicas are available and updated to the most current version. It also confirms that any old replicas are not running.
- And finally, the **failed** state occurs when the creation of a new ReplicaSet could not be completed. For example, if Kubernetes was unable to surface images for the new Pods, or the user who launched the operation lacks permissions, the failed state will occur.

Deployments can be used for Pod management



Because Deployments declare the state of Pods, they can be used for Pod management tasks such as updating, rolling back to previous revisions, and scaling or autoscaling.

Deployments are designed for stateless applications

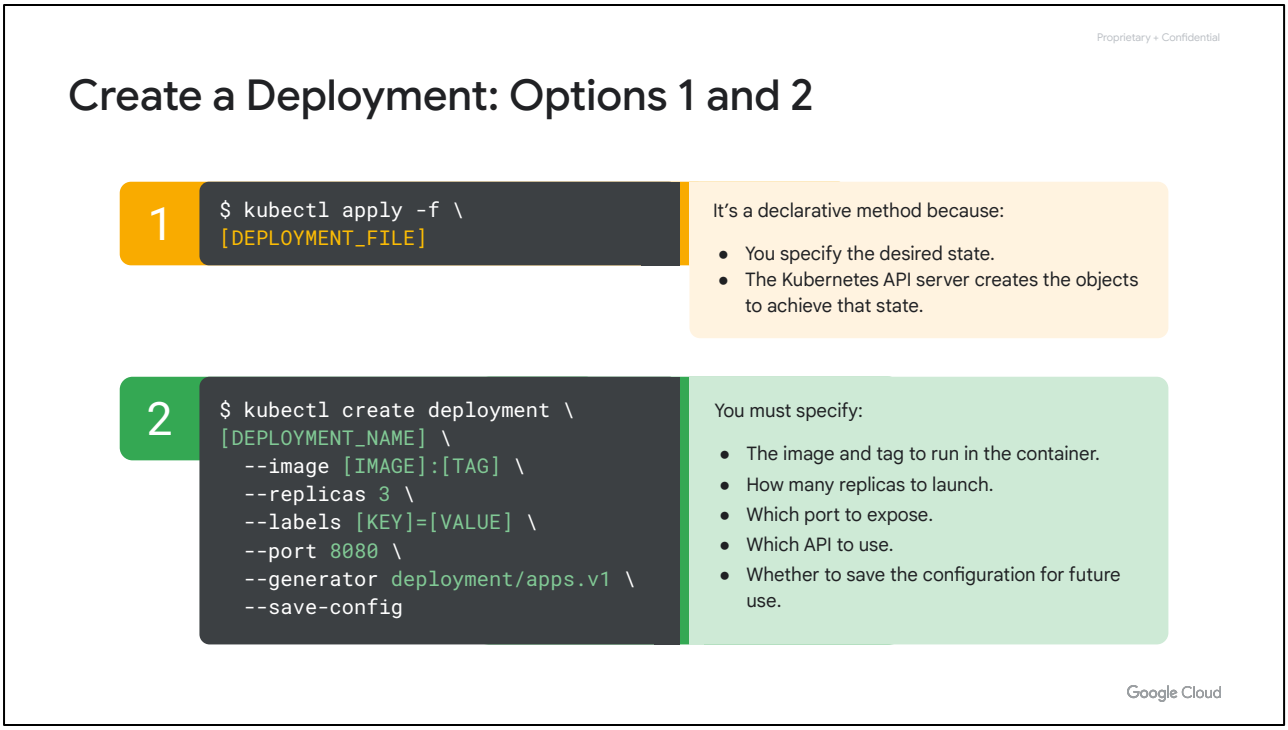


- Can be scaled up or down as needed without impacting the application's functionality.
- Examples include:
  - API servers that provide access to data.
  - Websites that don't contain dynamic content.

Deployments are designed for stateless applications.

A stateless application does not store data or state in a cluster or persistent storage, and can be scaled up or down as needed without impacting the application's functionality.

Examples of stateless applications include API servers that provide access to data, and websites that don't contain dynamic content.



So, how do you create a Deployment?

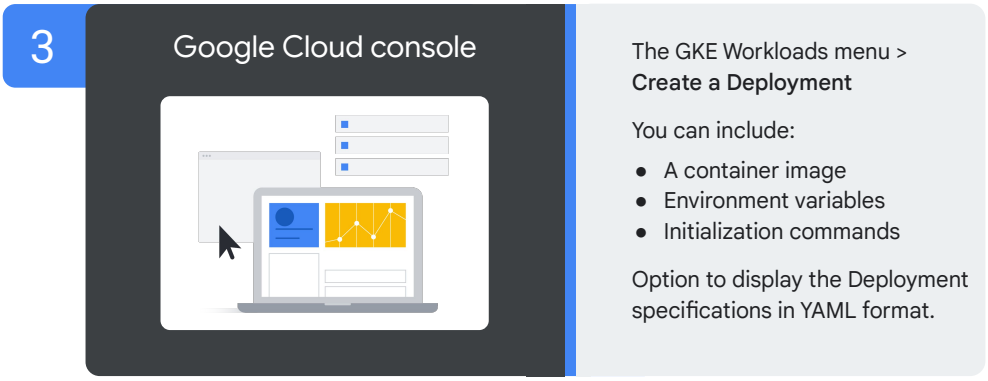
The **first option** is to create a YAML file and use the `kubectl apply` command to describe the desired state of the Deployment object. This is a *declarative* method, because *you* specify the desired state, and then the Kubernetes API server creates the necessary objects to achieve that state.

The **second option** is to use the `kubectl create deployment` command, which specifies the parameters inline.
This is an *imperative* method because you explicitly create the objects that you want to exist.

If you use the `kubectl create deployment` command, you need to specify:

- The image and tag to run in the container.
- How many replicas to launch.
- Which port to expose,
- Which API to use,
- And whether to save this configuration for future use.

Create a Deployment: Option 3



Finally, you can create a Deployment through the Google Cloud console.

The GKE Workloads menu includes an option to create a Deployment.

With this method you can include details such as a container image, environment variables, and initialization commands.

The Google Cloud console also gives the option to display the Deployment specifications in YAML format.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Let’s explore how to inspect the state of a Deployment.

Two commands to inspect the state of a Deployment

```
$ kubectl get deployment [DEPLOYMENT NAME]
```

```
$ kubectl describe deployment [DEPLOYMENT NAME]
```


Two useful commands to inspect the state of a Deployment are the kubectl get and describe commands.

kubectl get deployment command

```
$ kubectl get deployment [DEPLOYMENT NAME]
```

```
$ kubectl describe deployment [DEPLOYMENT NAME]
```

```
master $ kubectl get deployment nginx-deployment
NAME               READY  UP-TO-DATE  AVAILABLE  AGE
Nginx-deployment   3/3    3           3          3m
```

- **READY**: Displays the number of replicas in the Deployment specification and how many replicas are currently running.
- **UP-TO-DATE**: Displays the number of replicas that are fully up to date.
- **AVAILABLE**: Displays the number of replicas available to the users.
- **AGE**: Displays the time the replicas have been available to the users.

The `kubectl get deployment` command, displays the ready, up-to-date, and available status of all the ReplicaSets within a Deployment, along with their ages.

- The `Ready` field displays the desired number of replicas in the Deployment specification and how many replicas are currently running.
- The `Up-to-date` field displays the number of replicas that are fully up to date as per the current Deployment specification.
- The `available` field displays the number of replicas available to the users.
- And the `age` field displays the time the replicas have been available to the users.

kubectl get deployment command

```
$ kubectl get deployment [DEPLOYMENT NAME]
```

```
$ kubectl describe deployment [DEPLOYMENT NAME]
```

```
master $ kubectl get deployment nginx-deployment
NAME               READY  UP-TO-DATE  AVAILABLE  AGE
Nginx-deployment   3/3    3           3          3
```

```
$ kubectl get deployment [DEPLOYMENT NAME] -o yaml > this.yaml
```

Useful for making a Deployment a permanent, managed part of your infrastructure.

The `kubectl get deployment` command can also be used to output the Deployment configuration in a YAML format.

This could be useful if you originally created a Deployment by using the **`kubectl run`** command, but then decided to make it a permanent, managed part of your infrastructure. By editing that YAML file to remove the unique details of the Deployment you created it from, you can add it to your repository of YAML files for future Deployments.

kubectl describe command

```
$ kubectl get deployment [DEPLOYMENT NAME]
```

```
$ kubectl describe deployment [DEPLOYMENT NAME]
```

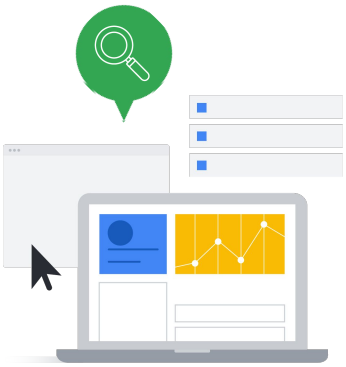
Includes the Deployment's:

- Current state
- Desired state
- Replicas
- Selector

```
master $ kubectl get deployment nginx-deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Thur, 4 Apr 2024 15:23:46 +0000
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
   Labels: app=nginx
   Containers
     nginx:
      Image:       nginx:1.15.4
      Port:        80/TCP
      Host Port:   0/TCP
```


Another option to inspect the state of a Deployment is through the **`kubectl describe`** command. Information includes the Deployment's current state, its desired state, its replicas, its selector, and its events.

Inspect through the Google Cloud console



Google Cloud console

Displayed is the:

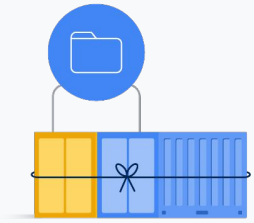
- Deployment name
- Date the Deployment was created
- Desired and current state
- Number of replicas
- Selector being used to match Pods
- List of events that have occurred

And finally, a third option to inspect a Deployment is through the Google Cloud console. Displayed in the Google Cloud console is the Deployment name, the date the Deployment was created, the desired state, the current state, the number of replicas the Deployment is trying to maintain, the selector being used to match Pods, and a list of events that have occurred for the Deployment. The details here can be exported to a YAML file for future use.

Deployment update

Example

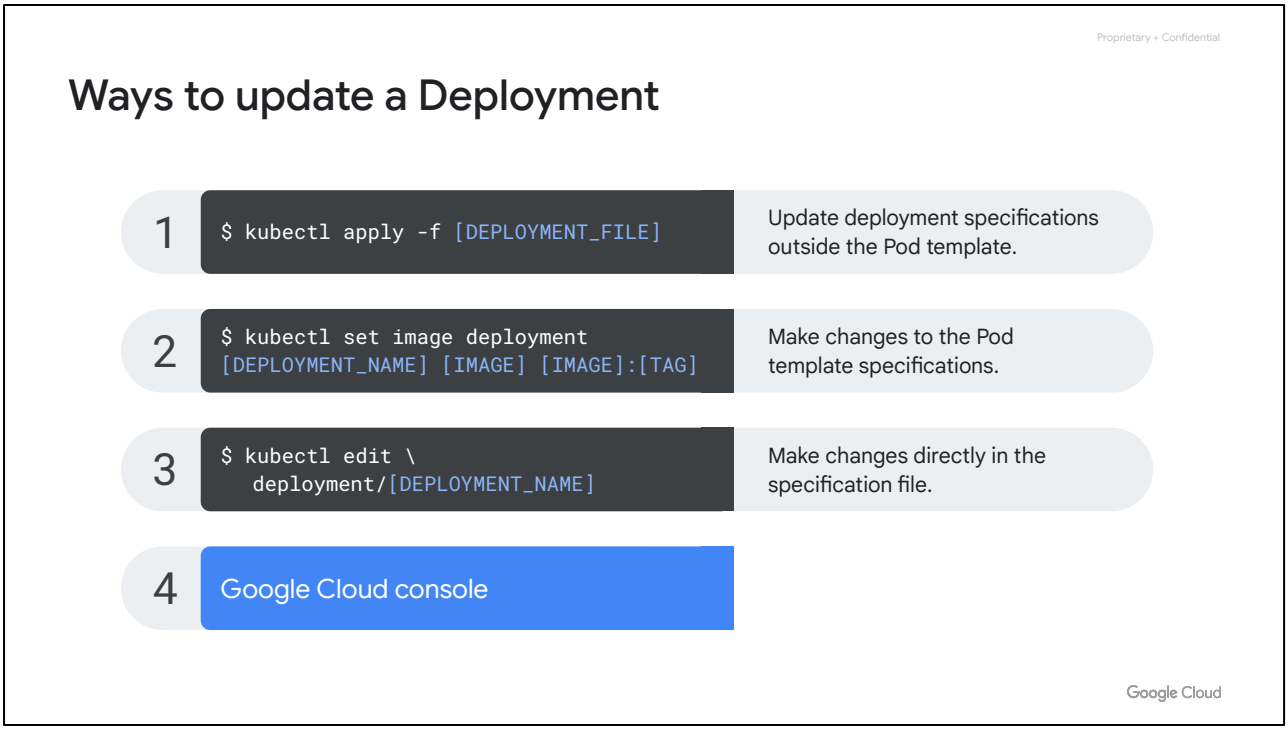
You updated a container image with new features



You need to update your Pod with the new version

Let's say you recently updated a container image with new and exciting features. You want your Pods to benefit from the improvements, so you need to update your Pod's specification with the new version of the image. This is an example of a deployment update.

Deployment updates are common, and they do not affect the availability of other applications and services in the cluster.



There are a few different ways to update a Deployment.

The first option is to use the `kubectl apply` command with an updated Deployment specification YAML file. This command lets you update Deployment specifications, such as the number of replicas, outside the Pod template.

The next option is the `kubectl set` command, which lets you make changes to Pod template specifications, like the image, resources, or selector values.

Next is the `kubectl edit` command, which lets you make changes directly in the specification file. Vim, which is an open-source, screen-based text editor, can be used to open the file.

After you save your changes, `kubectl` will automatically apply the updates. And finally, you can also update Deployments by using the Google Cloud console.

Rolling updates, aka a “ramped strategy”

App update

New set of Pods are launched in a ReplicaSet

Old Pods are gracefully retired

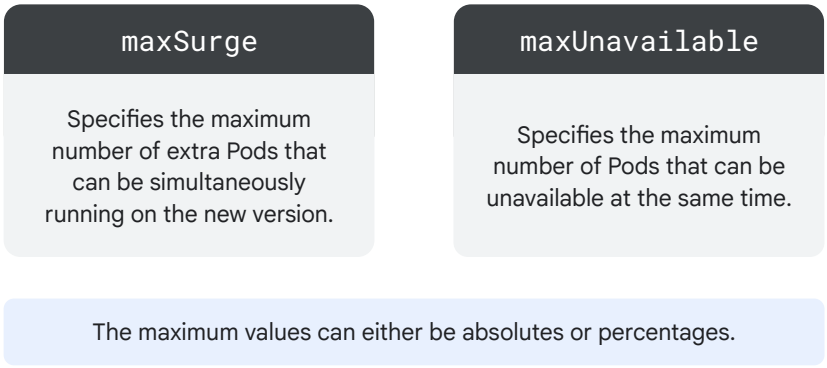
GKE updates Pods in a Deployment one at a time to avoid downtime or disruptions

Now, imagine updating your app without anyone noticing. That’s the magic of a **rolling update**, also known as a “ramped strategy.”

When a Deployment is updated, a new set of Pods are launched in a new ReplicaSet. Then, after the new Pods start running smoothly, the old Pods in the outdated ReplicaSet are gracefully retired.

GKE updates the Pods in a Deployment one at a time, meaning there’s always at least one Pod running the old version of your application and there won’t be any downtime or disruptions.

Configuring a rolling update



So, how is a rolling update configured?

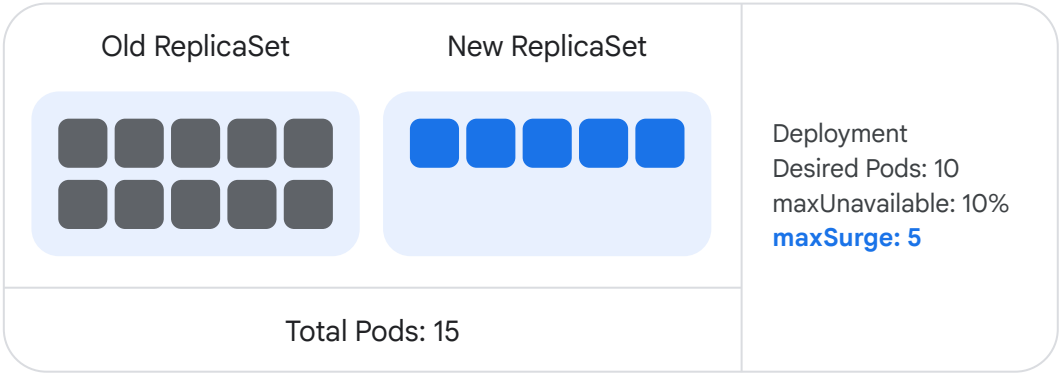
There are two primary parameters used to control the speed of rolling updates: `maxSurge` and `maxUnavailable`.

`maxSurge` specifies the maximum number of extra Pods that can be simultaneously running on the new version.

`maxUnavailable`, in contrast, specifies the maximum number of Pods that can be unavailable at the same time.

The maximum values can either be absolutes or percentages. For example, if you set `maxSurge` to 1 and `maxUnavailable` to 0, Kubernetes will update one Pod at a time, without any downtime.

New Pods are created in a new ReplicaSet



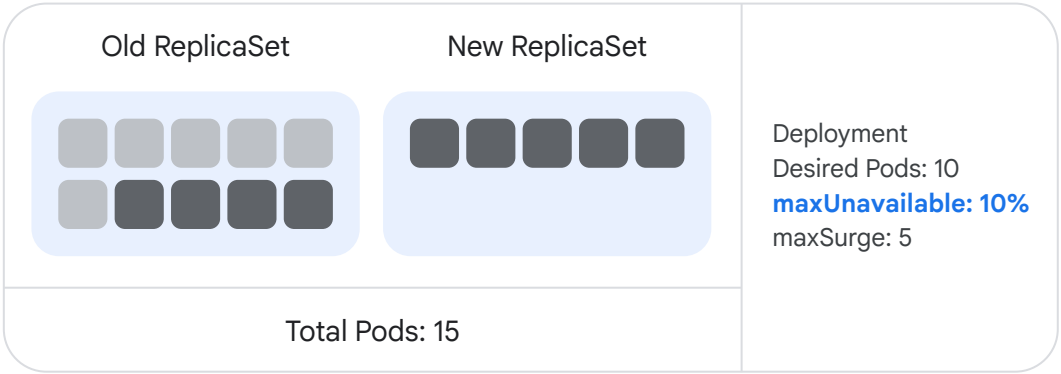
To explain this further, let's work through an example of a rolling update.

A Deployment has a desired number of Pods set to 10, with the `maxUnavailable` parameter set to 10% and the `maxSurge` parameter set to 5.

The old ReplicaSet has 10 Pods.

The Deployment will begin by creating 5 new Pods in a new ReplicaSet, based on the `maxSurge` parameter. When those new Pods are ready, the total number of Pods will change to 15.

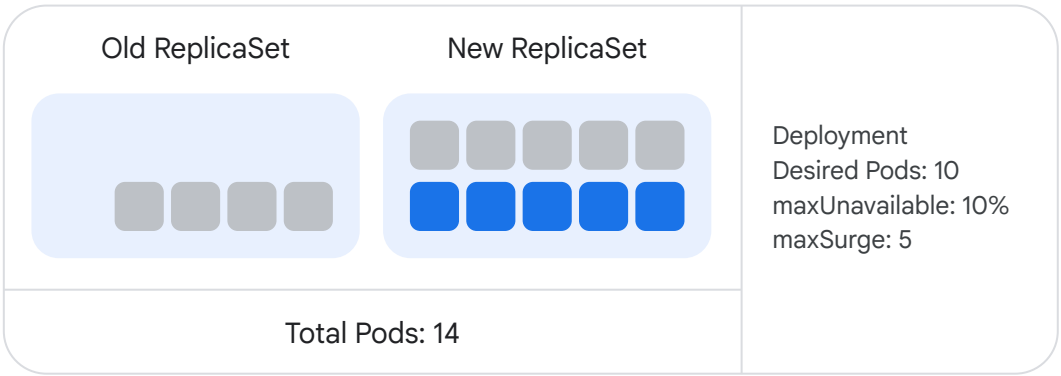
maxUnavailable



Since the `maxUnavailable` parameter is set to 10%, the minimum number of Pods that can run, regardless of whether they are from the old or new ReplicaSet, is 10 minus 10%. This equals at least a minimum of 9 Pods.

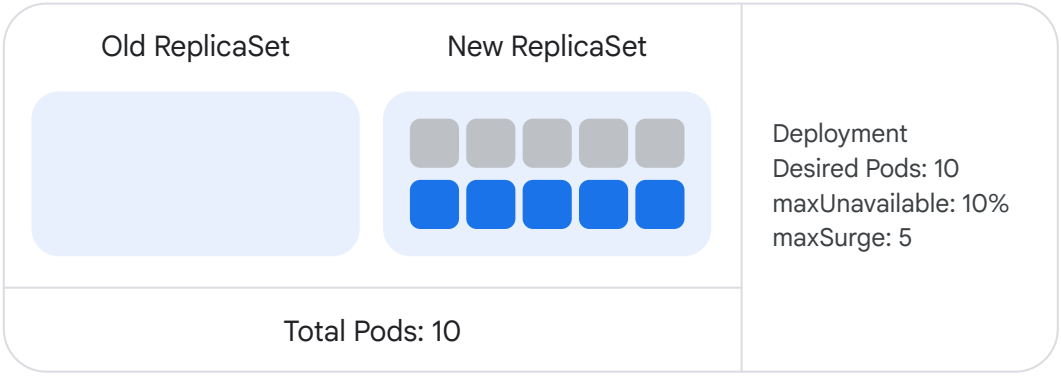
Therefore 6 of the 15 Pods can be removed from the old ReplicaSet, leaving a minimum of 9: 5 in the new ReplicaSet and 4 in the old ReplicaSet..

Additional Pods are launched in the new ReplicaSet



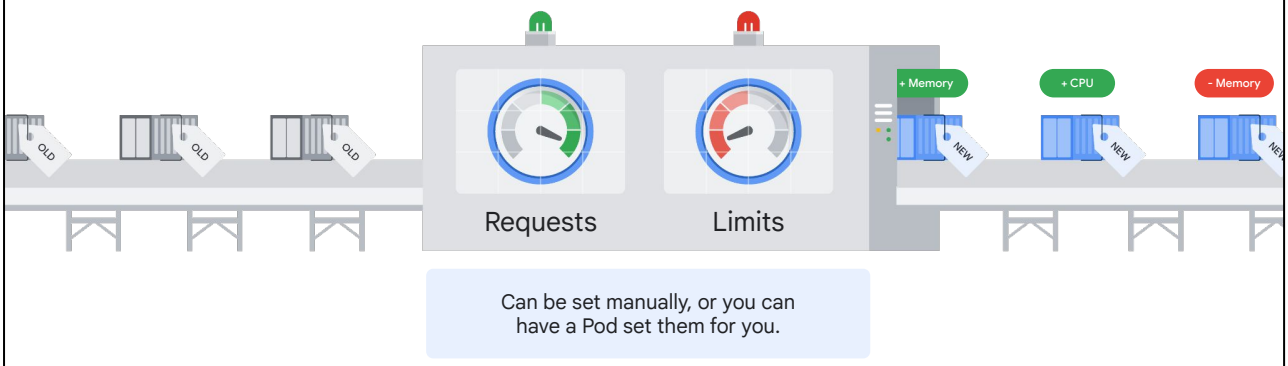
Next, an additional 5 Pods will be launched in the new ReplicaSet, totalling 10 Pods in the new ReplicaSet and 14 in all ReplicaSets.

Remaining old ReplicaSet Pods are deleted



Finally, the remaining 4 Pods in the old ReplicaSet will be deleted. The old ReplicaSet will be retained for rollback even though it's empty, and this will leave 10 Pods in the new ReplicaSet.

Control cluster resources with requests and limits



Cluster resources, like CPU and memory, will change as rolling updates occur. To control these resources, you can use `requests` and `limits`. Requests and limits can be set manually, or you can have a Pod set them for you.

Requests

Requests

Minimum amount of CPU and memory that a container will be allocated on a node.

- The scheduler will only assign a container to a node if that node has enough available resources to meet the container's requests.
- If you specify a CPU or memory value that is larger than the available resources on your nodes, your pod will never be scheduled.

Requests determine the minimum amount of CPU and memory that a container will be allocated on a node. The scheduler will only assign a container to a node if that node has enough available resources to meet the container's requests.

If you specify a CPU or memory value that is larger than the available resources on your nodes, your pod will never be scheduled.

Limits

Limits

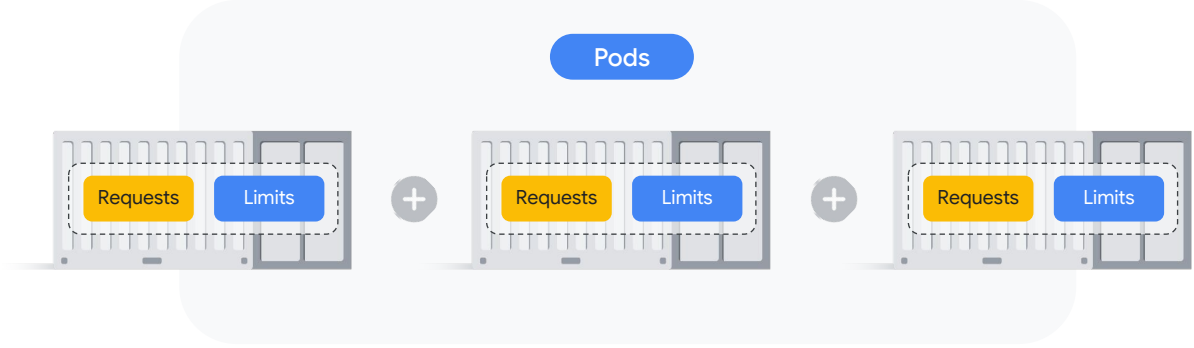
How much CPU and memory a container can use on a node.

- Prevents one container from consuming excessive resources and affecting other containers or critical node processes.
- The limit cannot be lower than the request.

Limits can set the upper boundary of how much CPU and memory a container can use on a node. This prevents one container from consuming excessive resources and affecting other containers or critical node processes.

Please note that the limit cannot be lower than the request.

Pods are scheduled as a group

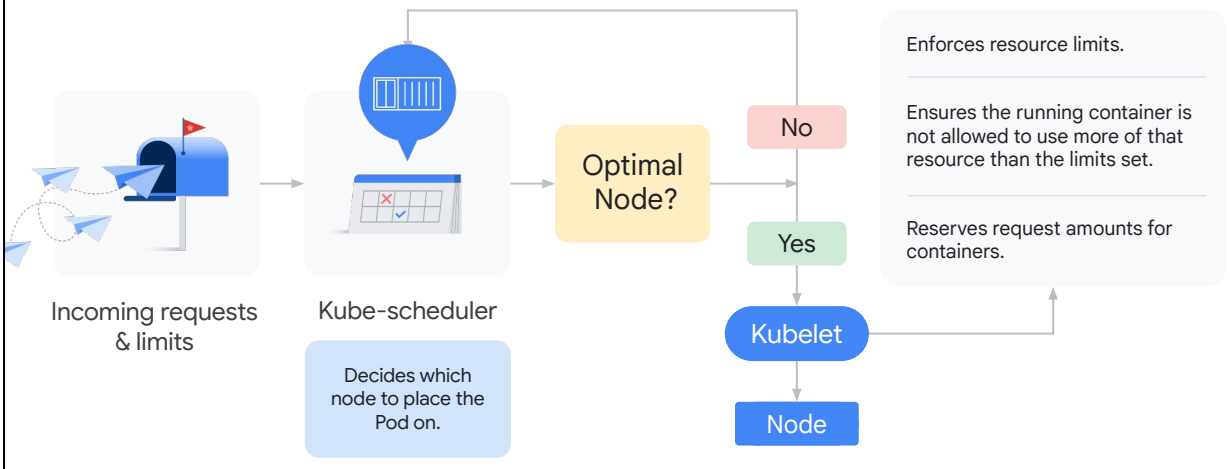


Calculate the total resource requests and limits by adding up the values from each individual container within the Pod.

Requests and limits must be for each individual container in the Pod.

Pods, however, are scheduled as a group, so you still need to calculate the total resource requests and limits by adding up the values from each individual container within the Pod.

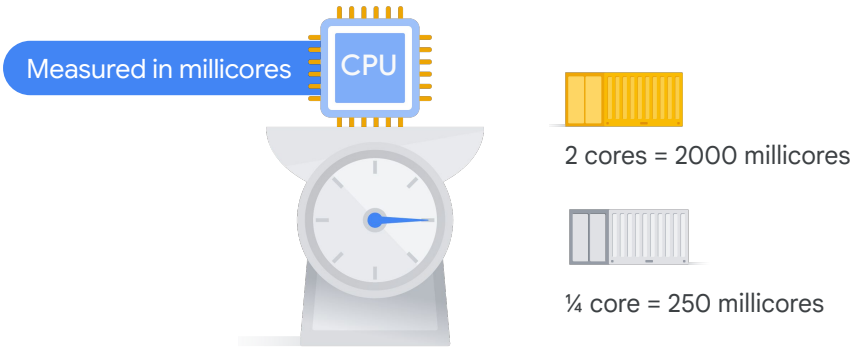
Manage requests and limits



To manage the requests and limits as they come into a cluster, Kubernetes uses the **kube-scheduler** to decide which node to place the Pod on. If the optimal node cannot be found for the request, it is sent back to the **kube-scheduler** to try again.

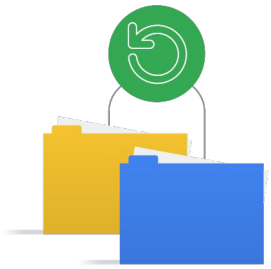
Once the optimal node is found, the **kubelet** is used to enforce resource limits and ensure that the running container is not allowed to use more of that resource than the limits set. The **kubelet** is also responsible for reserving request amounts for containers.

Central processing unit (CPU) resources

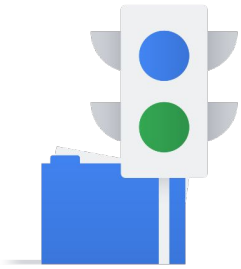


CPU resources are measured in millicores. For example a container that needs two full cores to run would be measured as 2000m (millicores). A container that needs one quarter of a core, would be measured as 250m.

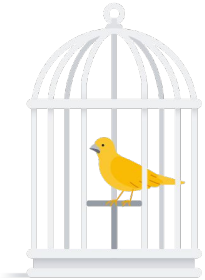
Other GKE Deployment strategies



Recreate deployment



Blue/green deployment

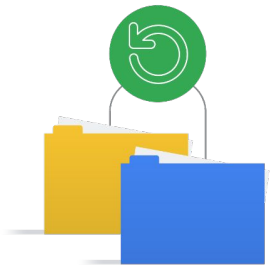


Canary test

GKE supports multiple deployment strategies to give you the flexibility and control over how you introduce changes and updates to your applications.

We've already seen a rolling update and how it functions, so now let's briefly explore three other deployment options: **recreate deployment**, **blue/green deployment**, and **canary test.**

The recreate strategy



- **Strategy**: Delete old Pods before creating new ones.
- **Advantage**: All users will gain access to the updated deployment at the same time.
- **Disadvantage**: Users will experience disruptions while new Pods are being created.

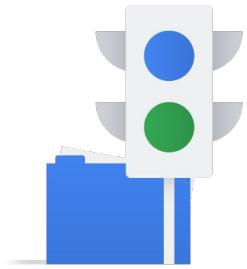
First, you can use the **recreate** strategy.

Unlike a rolling strategy, where both old and new Pods are running simultaneously, the recreate strategy is to delete the old Pods before creating new ones.

The advantage of this strategy is that once the new Pods are available, all users will gain access to the updated deployment at the same time.

The disadvantage is that because all Pods won't be instantly available, users will experience disruptions while new Pods are being created.

The blue/green deployment strategy



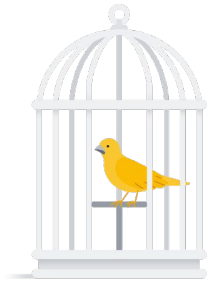
- **Strategy**: Create a completely new Deployment with a newer version of the application. Blue refers to the old version, and green refers to the new version.
- **Advantage**: Rollouts are instantaneous, and the newer versions can be tested internally before releasing.
- **Disadvantage**: Resource usage is doubled during the Deployment process.

Another option is to use the **blue/green deployment** strategy. This strategy is to create a completely new Deployment with a newer version of the application. Blue refers to the old version, green to the new version.

When the Pods are ready, the traffic can be switched from the old blue version to the new green version. Then, Pods in the blue deployment version will be deleted.

The advantage of this strategy is that the rollouts are instantaneous, and the newer versions can be tested internally before releasing them to all users. The disadvantage is that resource usage is doubled during the Deployment process, resulting in cost increases, capacity constraints, and inefficiencies.

The canary strategy



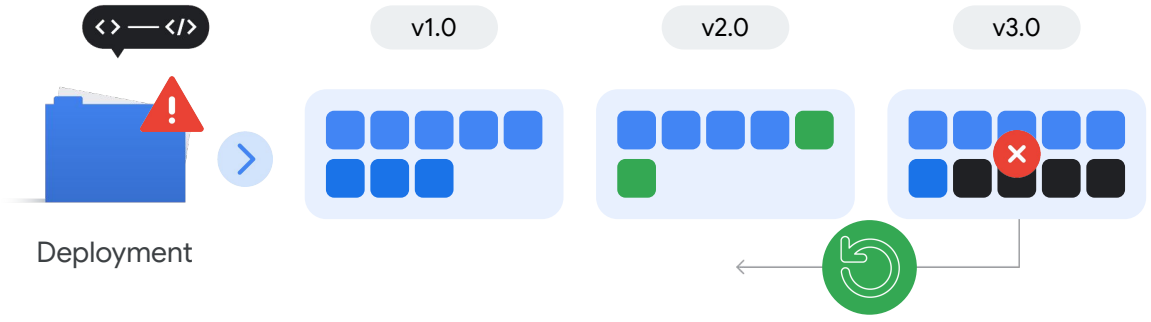
- **Strategy**: Gradually move traffic to the new version of your application.
- **Advantage**: It minimizes excess resource usage during the update and helps identify issues before rollout.
- **Disadvantages**: It's slower and may require additional tooling.

Finally, the **canary** update strategy is to gradually move traffic to the new version of your application.

The main advantage of this strategy is that it minimizes excess resource usage during the update. Also, because the rollout is gradual, issues can be identified before they affect all instances of the application. The disadvantages are that canary deployments are often slower and may require tools like Anthos Service Mesh to accurately move the traffic to the new version.

Rolling back updates

```
$ kubectl rollout undo deployment [DEPLOYMENT_NAME]
```



Now, what if there's an issue with a deployment? You'll likely need to undo, or "roll back" the update by using the kubectl rollout undo command.

The rollout undo command will revert the Deployment to the previous version or to a different version that you specify.

Inspect the rollout history

```
$ kubectl rollout history deployment [DEPLOYMENT_NAME] --revision=2
```

Cloud Shell

Google Cloud console

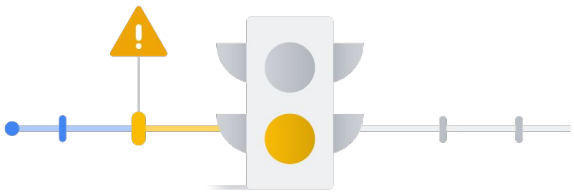
By default, you can inspect the rollout history from the previous 10 ReplicaSets.

Using the `kubectl rollout history` command, you can inspect the rollout history.

By default, the previous 10 ReplicaSets are retained. You can change the default by specifying a revision history limit in the Deployment specification.
The kubectl rollout undo and kubectl rollout history commands can also be run from Cloud Shell in the Google Cloud console. You can view a revision list before making any changes.

Pause rollouts to troubleshoot issues

```
$ kubectl rollout pause deployment [DEPLOYMENT_NAME]
```

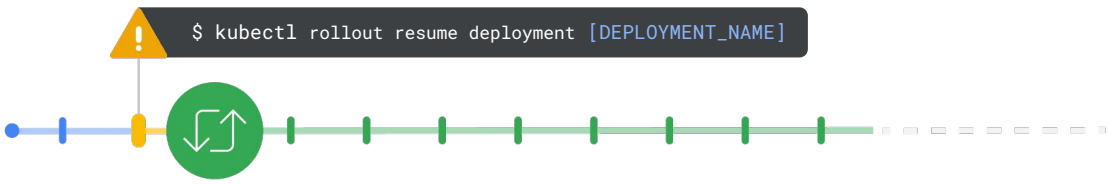


- An automatic rollout is triggered when a deployment is edited.
- Frequent updates lead to a large amount of rollouts.
- The kubectl rollout pause command will halt a rollout so that you can troubleshoot.

Typically, an automatic rollout is triggered when a deployment is edited, so frequent updates with small fixes will lead to a large amount of rollouts.

In these situations, it can be difficult to pinpoint which specific rollout is causing a problem. You can use the `kubectl rollout pause` command to halt the rollout so that you can troubleshoot.

Resume rollouts

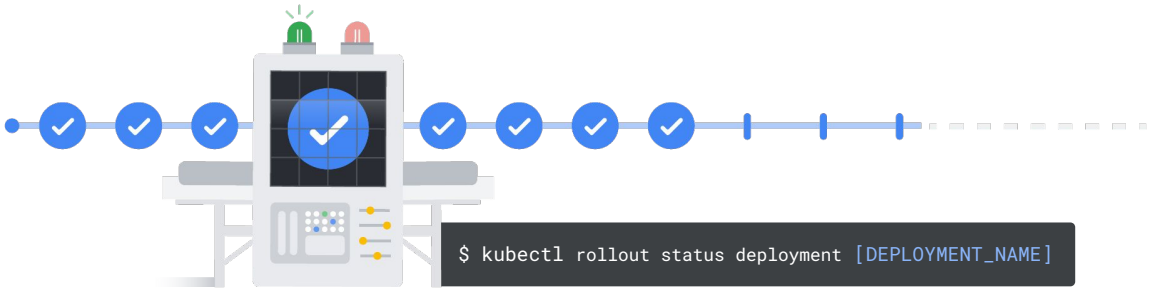


New changes to the deployment will only be implemented after the rollout is resumed.

All new changes will be released as one new version.

Please note that any new changes to the deployment will only be implemented after the rollout is resumed by using the `kubectl rollout resume` command. Then all new changes will be released as one new version.

Monitor the status of a rollout

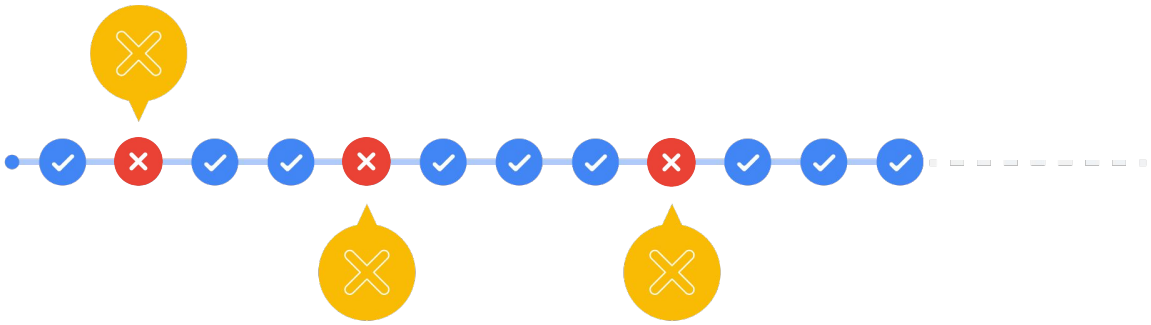


To monitor the status of a rollout, you can use the `kubectl rollout status` command.

Delete a rollout

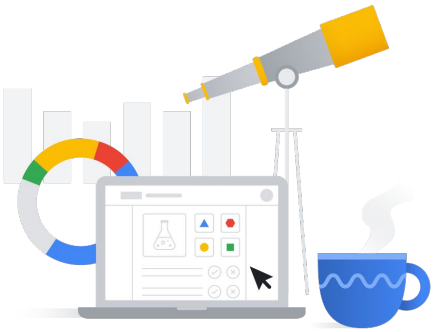
```
$ kubectl delete deployment [DEPLOYMENT_NAME]
```

Google Cloud console



Finally, you can use the `kubectl delete` command to delete a rollout, or you can delete it by using the Google Cloud console. Kubernetes will delete all resources managed by the Deployment, including any running Pods.

Lab: Creating Google Kubernetes Engine Deployments



01 Create deployment manifests, deploy to cluster, and verify Pod rescheduling as nodes are disabled.

02 Trigger manual scaling up and down of Pods in deployments.

03 Trigger deployment rollout and rollbacks.

04 Perform a Canary deployment.

It's time for some hands-on practice with GKE.

In the lab titled "Creating Google Kubernetes Engine Deployments," you'll:

- Create deployment manifests, deploy to cluster, and verify Pod rescheduling as nodes are disabled.
- Trigger manual scaling up and down of Pods in deployments.
- Trigger deployment rollout (which is a rolling update to new version) and rollbacks.
- And perform a Canary deployment to complete the lab.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

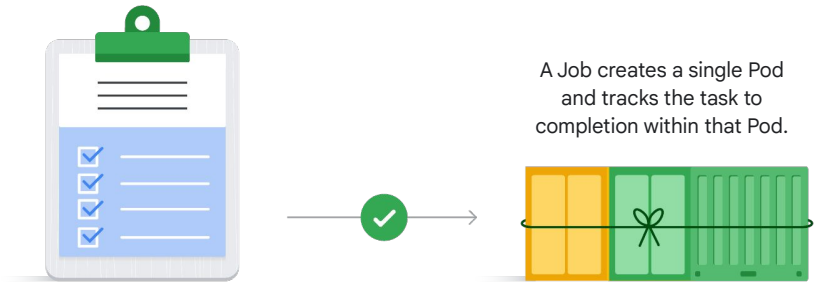
03 Scale clusters

04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Jobs create Pods to execute tasks



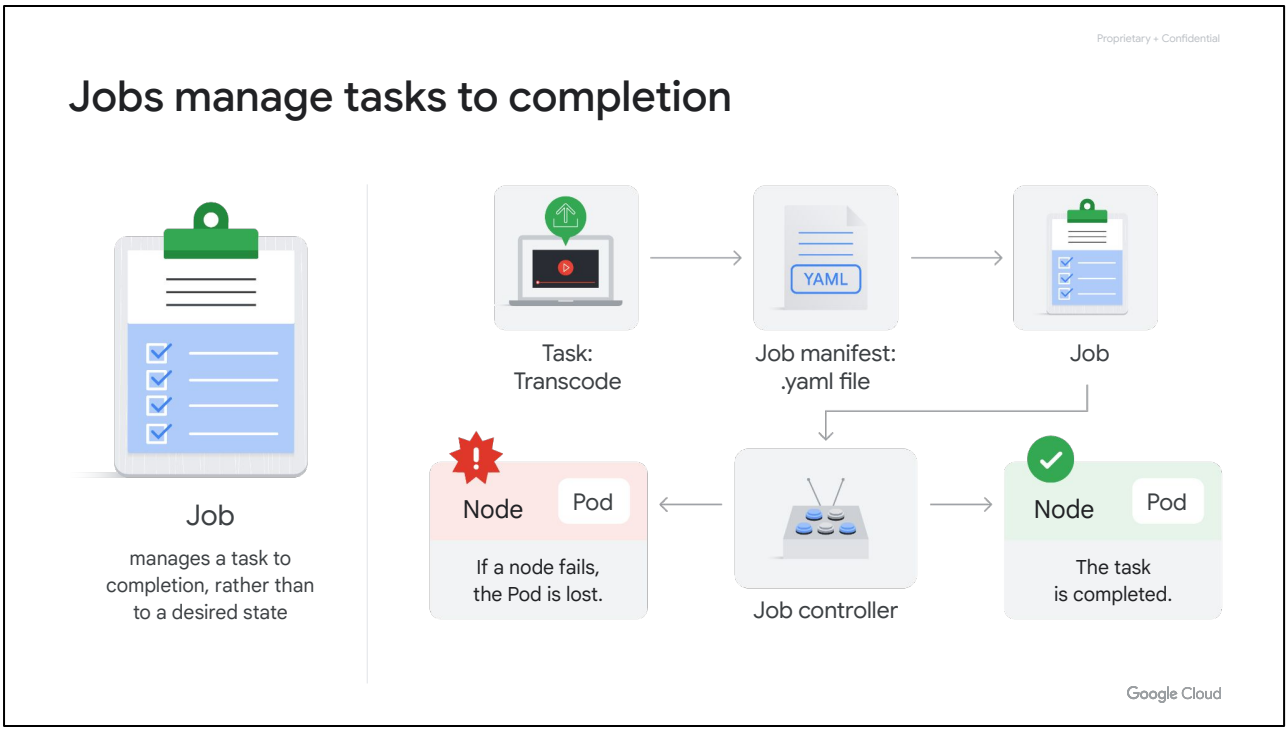
Job

A Kubernetes resource.

Task

Like a Deployment, a **Job** is a Kubernetes resource. Jobs create one or more Pods to execute a specific task, and then automatically terminate the pods once the task is finished.

At its simplest, a Job creates a single Pod, and tracks the task to completion within that Pod.



Jobs are also useful if a Pod fails.

Unlike Kubernetes controllers, a Job manages a task to completion, rather than to a desired state.

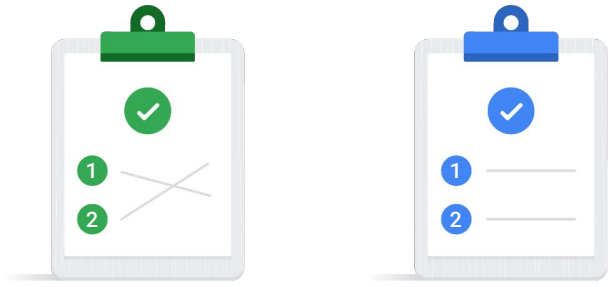
Let's explore this concept with an example–like when a user uploads a video file to a web server for transcoding. In this instance, transcoding would be the task. To begin, the web server would create a Job manifest in a YAML file for the task, and then a Job would be created on the cluster.

From there, the Job controller would schedule a Pod on a node to complete the task, and then monitor the Pod. If a node fails, the Pod is lost.

The Job controller, which monitors the Pod, would be aware that the task has not been completed.

As a result, the Job controller would then reschedule the Pod to run on a different node, and continue to monitor the Pod until the task is completed.

There are two types of Jobs

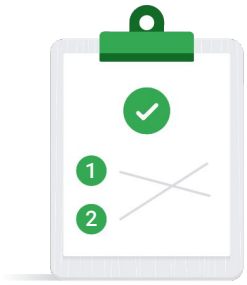


Non-parallel

Parallel

There are two main types of Jobs: non-parallel and parallel.

Non-parallel Jobs



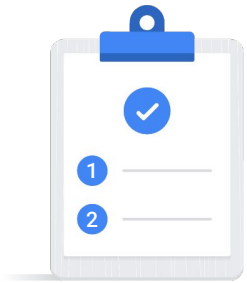
Non-parallel

- Used to run a single task one time and ensure its completion.
  - Image processing
  - Data migration
- One Pod executes the task.
- The Job is finished when the Pod exits successfully, or when the required number of completions is reached.
- If the Pod fails, it will be recreated.

Non-parallel Jobs are the simplest type, used to run a single task one time and ensure its completion. Examples include image processing and data migration.

Kubernetes creates one Pod to execute the task. And the Job is considered finished when the Pod exits successfully or when the required number of completions is reached. If the Pod fails, it will be recreated.

Parallel Jobs



Parallel

- Schedule multiple Pods to work on a Job concurrently, but with a predefined limit on the number of completions.
- Each Pod works independently on its assigned task.
- The Job finishes when the specified number of tasks are completed successfully.
- Useful when tasks need to be completed more than once:
  - Bulk image resizing
  - Parallel scientific computations

Alternatively, parallel Jobs schedule multiple Pods to work on the Job concurrently, but with a predefined limit on the number of completions. This means that each Pod works independently on its assigned task and the Job finishes when the specified number of tasks are completed successfully.

Parallel Jobs are useful for situations where tasks need to be completed more than once, like bulk image resizing or parallel scientific computations.

Other types of parallel Jobs exist such as work queues and indexed completion jobs, but these are beyond the scope of this course.

The Kind property

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

```
$ kubectl apply -f [JOB_FILE]
```

```
$ kubectl run pi --image perl:5.34 --restart Never -- perl
-Mbignum=bpi -wle 'print bpi(2000)'
```

Acts as a label that tells Kubernetes:

- What type of batch process the Job represents.
- How it should be handled.

Let's explore another example, this time how a parallel Job can be used to calculate pi to 2000 decimal places.

In a manifest file, the specific type of a Job object is identified through the value of its "Kind" property. This property acts as a label that tells the Kubernetes system what type of batch process the Job represents and how it should be handled.

The Job spec and Pod template

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

Job Spec: How the Job should perform its tasks.

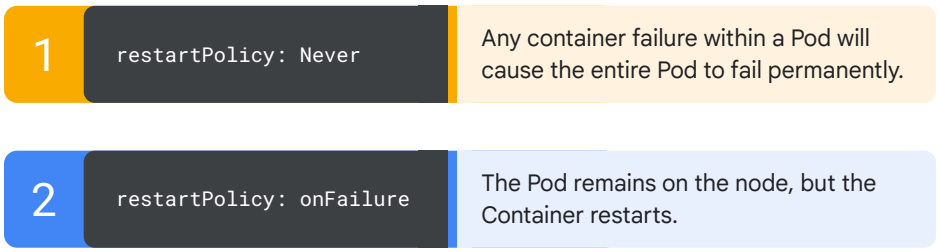
Pod template: A blueprint for the Pods that the Job will create.

```
$ kubectl apply -f [JOB_FILE]
```

```
$ kubectl run pi --image perl:5.34 --restart Never -- perl
-Mbignum=bpi -wle 'print bpi(2000)'
```


Details about how the job should perform its tasks are specified in the Job spec. Nested within the job spec is a Pod template, which acts as a blueprint for the Pods that the Job will create. Also, this is where its restartPolicy is defined.

Configuring the restartPolicy



The restartPolicy can either be set to *Never* or *onFailure*.

If the restartPolicy is set to *Never*, it means that any container failure within a Pod will cause the entire Pod to fail permanently. In response, the Job controller will automatically create a new Pod to continue the task.

If set to *OnFailure*, the Pod remains on the node, but the Container restarts.

backOffLimit field controls restarts

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

Controls how many times the Job controller should attempt to restart failed Pods.

```
$ kubectl apply -f [JOB_FILE]
```

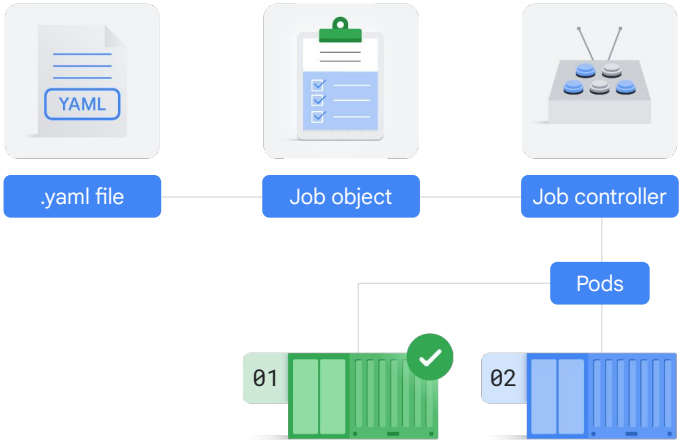
```
$ kubectl run pi --image perl:5.34 --restart Never -- perl
-Mbignum=bpi -wle 'print bpi(2000)'
```


And then there is a backOffLimit field that controls how many times the Job controller should attempt to restart failed Pods before considering the Job itself as failed.

Identify completions and parallelism

Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[…]
```



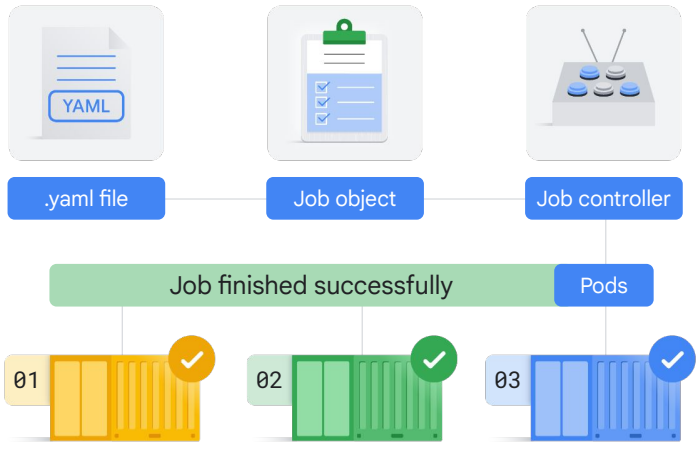
Setting spec.parallelism greater than 1 signals to the Job controller that this is a parallel job and that it should create and run multiple Pods concurrently to execute the Job's tasks.

To run multiple Pods concurrently for a Job, you need to identify both the completions and parallelism values. The Job controller will initially create as many Pods as specified by parallelism.

New Pods launch until completion

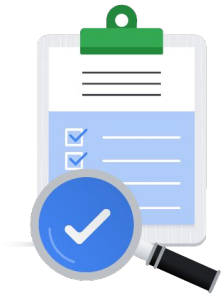
Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[…]
```



It will then keep launching new Pods to replace any that finish, up until the total successful completions reach the completions count.

Inspecting Jobs and filtering pods



```
$ kubectl describe
job [JOB_NAME]
```

Used to inspect Jobs.

```
$ kubectl get pod -l
[job-name=my-app-job]
```

Used to filter Pods.

Like other Kubernetes objects, Jobs can be inspected by using the **`kubectl describe`** command.

You can use the **`kubectl get`** command and label selector to filter Pods.

Job details can also be viewed from the Google Cloud console.

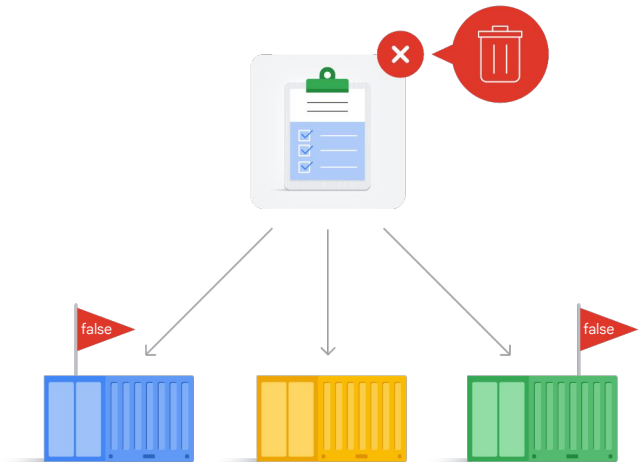
Delete a Job

```
$ kubectl delete -f [JOB_FILE]
```

```
$ kubectl delete job [JOB_NAME]
```

```
$ kubectl delete job [JOB_NAME]
   --cascade false
```

Delete jobs directly in the Google Cloud console



You can delete a Job by using a kubectl delete command.

When you delete a Job, the Job's Pods will also be deleted, but you can retain the Pods by setting the cascade flag to false.

Alternatively, you can delete jobs directly in the Google Cloud console.

Use CronJobs for repeated execution

CronJobs

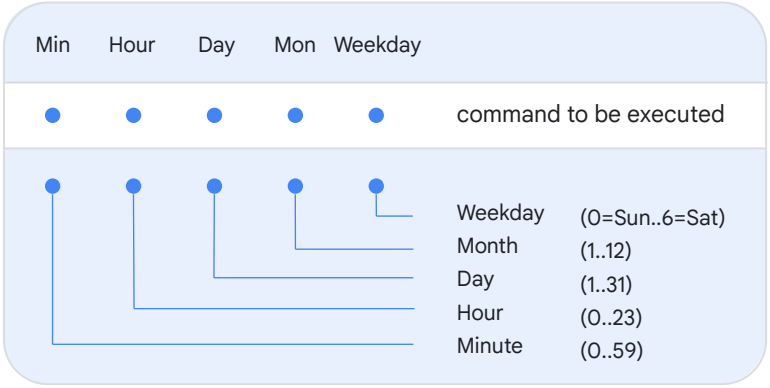
Specify precise dates and times for repeated execution

Let's finish this lesson by exploring Cronjobs.

For scheduling Jobs, the Cron format offers a powerful and flexible way to specify precise dates and times for repeated execution.

Cron uses an easy-to-read text format

- Different parts separated by spaces.
- Each part controls a different aspect of when and what to execute.
- Use an * asterisk to represent an entire range of possible values.



Cron uses a straightforward text format, which is like a mini-language with specific rules. You write a string with different parts separated by spaces, and each part controls a different aspect of when and what to execute.

Specify individual times or ranges

Examples

`0 * * * *`	Every hour
`*/15 * * * *`	Every 15 minutes
`0 */6 * * *`	Every 6 hours
`0 20 * * 1-5`	Every week Mon-Fri at 8 pm
`0 0 1 */2 *`	At 00:00 on day 1 of every other month

Schedule Jobs by specifying individual times or ranges.

You can use an asterisk to represent an entire range of possible values, for example, each minute or each hour.

Cron lets you schedule your jobs precisely by specifying individual times or ranges. You can do this by listing specific values separated by commas (like 1, 3, 7) or using a hyphen to define a range (like 1-5).

Repeat values at regular intervals

Examples

`0 * * * *`	Every hour
`*/15 * * * *`	Every 15 minutes
`0 */6 * * *`	Every 6 hours
`0 20 * * 1-5`	Every week Mon-Fri at 8 pm
`0 0 1 */2 *`	At 00:00 on day 1 of every other month

To make a value repeat at regular intervals, add a slash followed by the interval number.

To make a value repeat at regular intervals, add a slash followed by the interval number. This works for both asterisks (to repeat every interval) and ranges (to repeat the entire range at the specified interval).

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

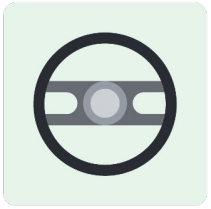
05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Use GKE Standard mode for manual cluster scaling

Standard mode

Custom-managed cluster scaling



Autopilot mode

Automatic cluster scaling

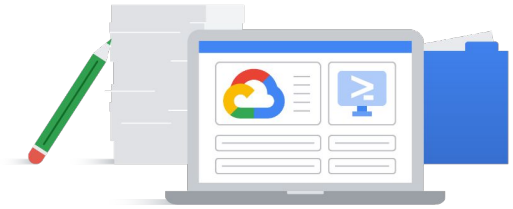
Let's explore cluster scaling in Google Kubernetes Engine.

GKE can either be used in Standard mode or Autopilot mode, but cluster scaling is only available in Standard mode.

Unlike Autopilot mode, which automatically scales your cluster based on demand, Standard mode hands you the reins for manual scaling.

Application resources fluctuate

Easily scale a cluster up or down to match those changing demands.

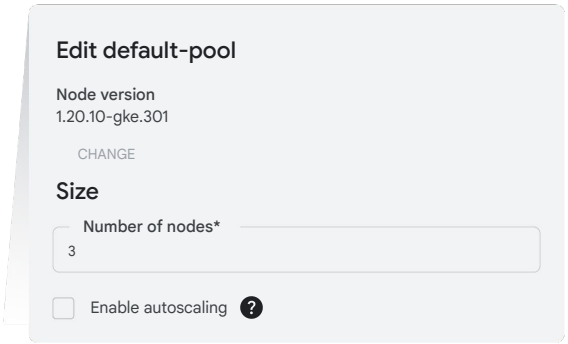


Google Cloud console

Cloud Shell

As your applications' needs fluctuate, the resources they require will change too. The good news is that you can easily scale your cluster up or down to match those changing demands, right from the Google Cloud console or Cloud Shell.

Node pools



1. A node pool **groups nodes** that have the same configuration type within a cluster.
2. NodeConfig specification
3. Container cluster

In GKE, a cluster contains one or more node pools. A **node pool** groups nodes that have the same configuration type within a cluster.

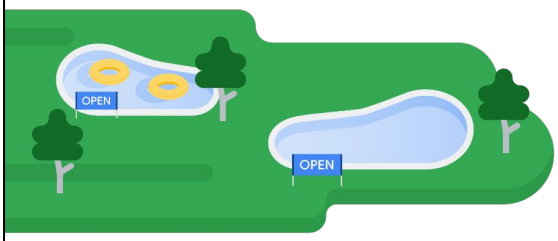
The size of a node pool is set by specifying a minimum and maximum number of nodes, the maximum being 1000.

Node pools use a NodeConfig specification. Each node in the pool has a Kubernetes node label that has the node pool's name as its value.

When a container cluster is created, the number and type of nodes specified becomes the default node pool.

Then additional custom node pools of different sizes and types can be added to the cluster.

A cluster can not be entirely shut down



- A cluster must have at least one node to run system Pods.
- A cluster can be scaled down to 0, but the cluster itself can not be entirely shut down.

And while individual node pools within a cluster can be scaled down to 0, the cluster itself can not be entirely shut down. This is because a cluster must have at least one node to run system Pods.

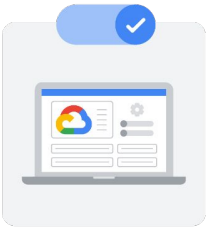
When a cluster is scaled down, nodes are treated the same regardless of whether or not they are running Pods.

The resize gcloud command for manual scaling

Standard mode

```
gcloud container clusters resize projectdemo
--node-pool default-pool \
  --size 6
```

- Manually resizes a cluster.
- Removes instances at random.
- Terminates running Pods gracefully.

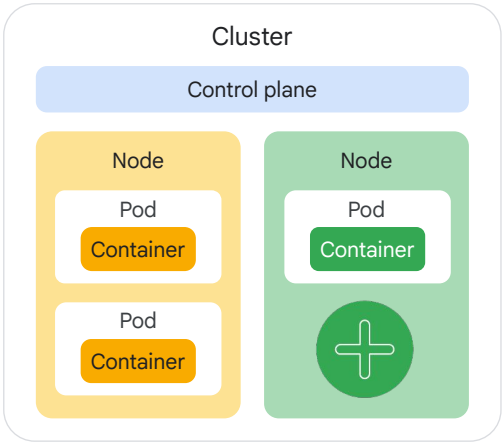


Google Cloud console

As stated previously, a cluster can be resized manually or automatically. You can resize a cluster in Standard mode by manually entering the `resize` gcloud command, or by using the Google Cloud console.

The `resize` command will remove instances at random, and any running Pods will terminate gracefully.

GKE cluster autoscaler



- Available on Standard cluster node pools.
- Automatically resizes a cluster based on the workloads' resource demands.
- Disabled by default.
- Automatically adds new nodes to a cluster.
- Deletes underutilized nodes.

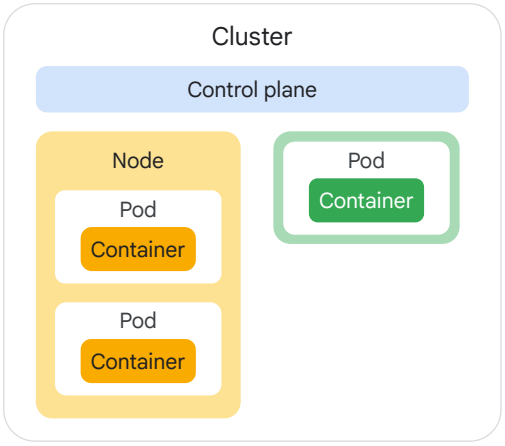
And what if you want to automatically scale clusters?

GKE's cluster autoscaler, which is a feature available on your Standard cluster node pools, automatically resizes a cluster based on the resource demands of your workload.

The cluster autoscaler is disabled by default. When enabled, GKE will automatically add new nodes to a cluster whenever it detects that your Pods lack sufficient resources to operate as intended.

GKE will also delete underutilized nodes if their Pods can run on other nodes.

Nodes with low resource capacity

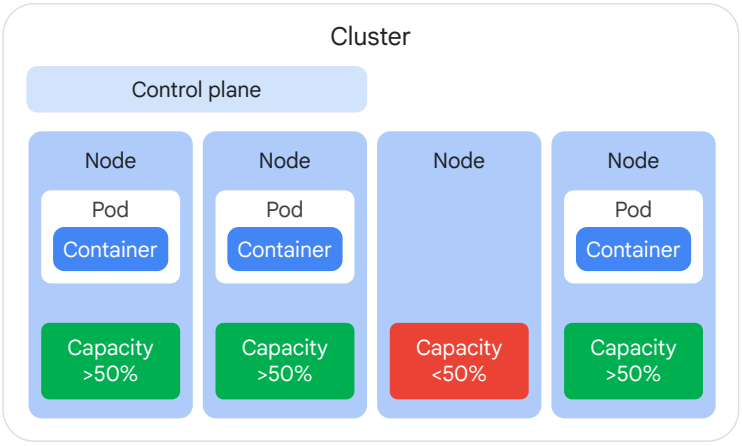


- Scenario: all of your node pools are low on resource capacity.
- Solution: terminate Pods or add more nodes.
- Process:
  - Pod enters a holding pattern.
  - The scheduler sets the schedulable Pod condition to false and marks it as `unschedulable`.

Now let's say there is a scenario where all of your node pools are low on resource capacity. To fix this, one or more Pods will need to be terminated or additional nodes need to be added.

The Pod will enter a holding pattern as it awaits additional resource capacity. During this period, the scheduler, which has the job of filtering any nodes that don't meet a Pod's specific scheduling needs, sets the schedulable Pod condition to false and marks it as `Unschedulable`.

Cluster autoscaler checks for disposable nodes



A node is considered disposable if all are true:

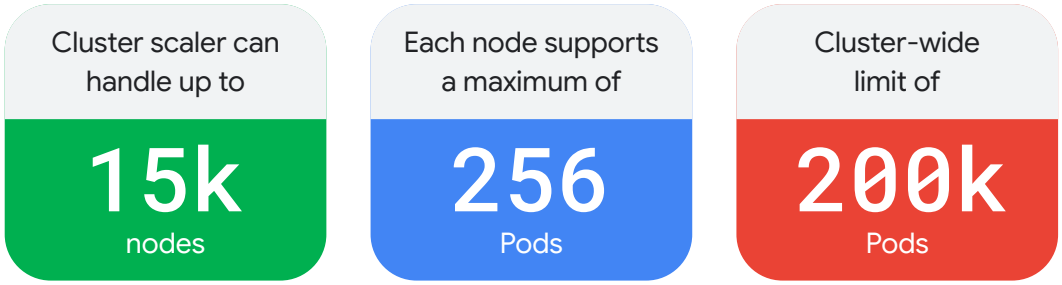
- Total CPU and memory is less than 50% of a node's allocatable capacity.
- All pods running on the node can be moved to other nodes.
- The cluster does not have scale-down disabled.

If a cluster doesn't need to scale up, the cluster autoscaler will check for disposable nodes every 10 seconds. A node is considered disposable if all of the following conditions are true:

- Total CPU and memory is less than 50% of a node's allocatable capacity.
- All pods running on the node can be moved to other nodes.
- The cluster does not have scale-down disabled.

The cluster autoscaler will then continue to monitor. And if a node is unneeded for more than 10 minutes, it will be terminated.

Cluster autoscaler limits



The cluster autoscaler can handle up to 15,000 nodes, with each node supporting a maximum of 256 pods.

However, there's a cluster-wide limit of 200,000 pods, regardless of your node setup.

Be aware that standard Google Cloud limits for Compute Engine instances still apply. So if you haven't increased your default quota, new VMs won't start and disruptions may occur.

gcloud commands for autoscaling

Create a cluster with autoscaling enabled

```
gcloud container clusters create
[CLUSTER_NAME] --enable-autoscaling
--min-nodes 15 --max-nodes 50 [--zone
COMPUTE_ZONE]
```

Enable autoscaling for an existing node pool

```
gcloud container clusters update
[CLUSTER_NAME] --enable-autoscaling \
--min-nodes 1 --max-nodes 10 --zone
[COMPUTE_ZONE] --node-pool [POOL_NAME]
```

Add a node pool with autoscaling enabled

```
gcloud container node-pools create
[POOL_NAME] --cluster [CLUSTER_NAME]
--enable-autoscaling --min-nodes 15
--max-nodes 50 [--zone COMPUTE_ZONE]
```

Disable autoscaling for an existing node pool

```
gcloud container clusters update
[CLUSTER_NAME] --no-enable-autoscaling \
--node-pool [POOL_NAME] [--zone
[COMPUTE_ZONE] --project [PROJECT_ID]]
```


Let's wrap up by exploring some common gcloud commands used for autoscaling.

You can add the **`--enable-autoscaling`** flag to a **`clusters create`** command to create a new cluster with autoscaling enabled.

You can also add **`--enable-autoscaling`** flag to a **`node-pools create`** command to create a new node pool with autoscaling enabled.

You can enable or disable autoscaling on existing node pools by adding a **`--enable-autoscaling`**, or **`--no-enable-autoscaling`** `flag` to a **`cluster update`** command.

These actions can be executed from the Google Cloud console too.

With zonal clusters, all resources (nodes and the control plane) are created in the same zone by default. If secondary zones are enabled, all node pools will be duplicated in the secondary zone, similar to how pools are duplicated for regional clusters.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

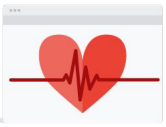
05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Pod placement in GKE

The process of controlling where your application's Pods are deployed across the cluster's nodes.

Optimize performance



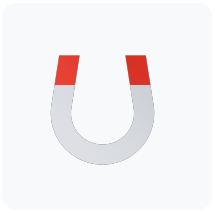
Ensure availability

Manage resource allocation

Pod placement in GKE refers to the process of controlling where your application's Pods are deployed across the cluster's nodes. This process plays a crucial role in optimizing performance, ensures availability, and manages resource allocation within your GKE environment.

Control and manage Pod placement

Labels



Affinity rules

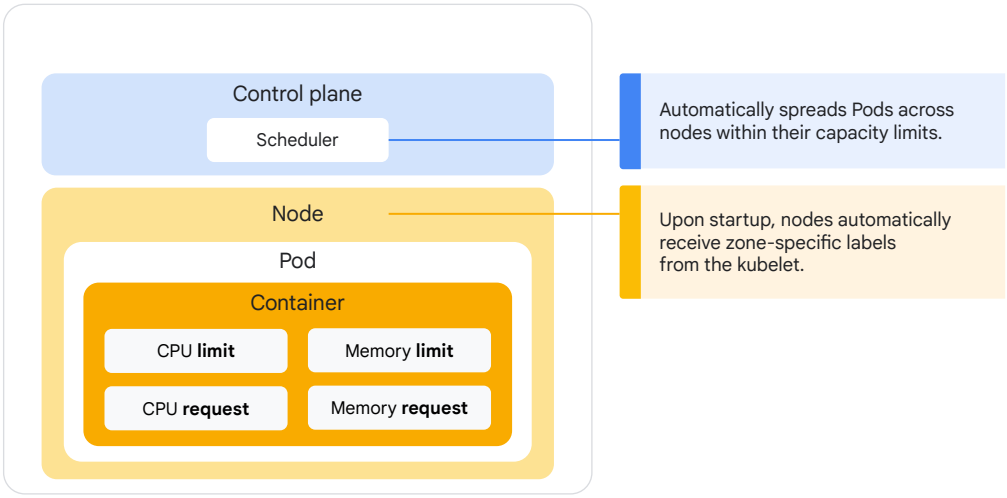
Taints

Tolerations

In GKE Standard mode, you can control and manage Pod placement by using labels, affinity rules, taints, and tolerations.

Let's begin by exploring labels and affinity rules.

Controlled scheduling

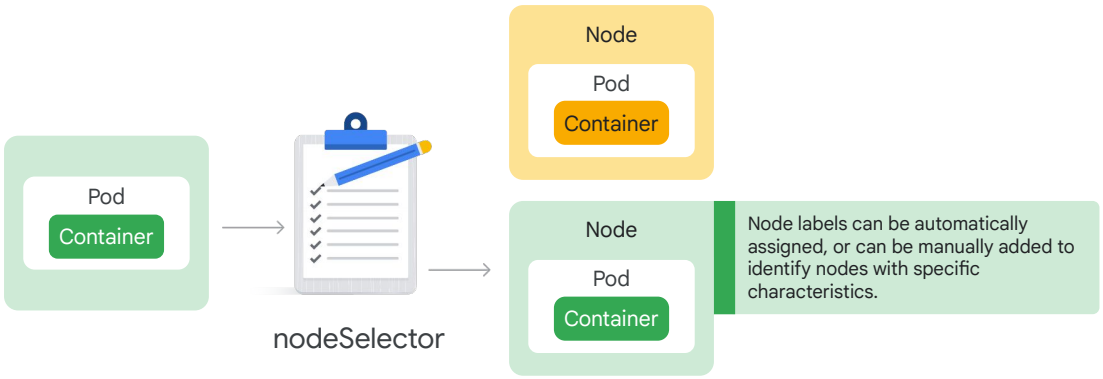


To identify the appropriate location to place a Pod, GKE uses a Pod's specifications like resource requests and limits.

Then, the scheduler automatically spreads Pods across nodes within their capacity limits.

Upon startup, nodes automatically receive zone-specific labels from the kubelet. This ensures accurate tracking even when nodes span compute zones.

nodeSelector specifies preferred node labels



But what if you want certain application types to run on a specific node?

In GKE Standard mode, you can use the `nodeSelector` field within a Pod's configuration to specify its preferred node labels. Think of it as a checklist the Pod uses to find a compatible home.

Node labels can be automatically assigned, or can be manually added to identify nodes with specific characteristics. This creates a system of matching between Pod preferences and node capabilities.

Match Autopilot compute class to Pod preferences

```
apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: hello-app
  spec:
   replicas: 3
    selector:
      matchLabels:
        app: hello-app
      spec:
        nodeSelector:
          cloud.google.com/compute-class: Balanced
   […]
```

Use the nodeSelector field to request the "Balanced" compute class and ensure Pods are placed on nodes with the resources they need.

Enable Autopilot to this by adding the cloud.google.com/compute-class label.

If your Pods need more CPU or memory than the general-purpose class allows, or if they require a specific CPU type, you can use the **nodeSelector** field in a Pod's specification to request the "Balanced" compute class. This will ensure Pods are placed on nodes with the resources they need.

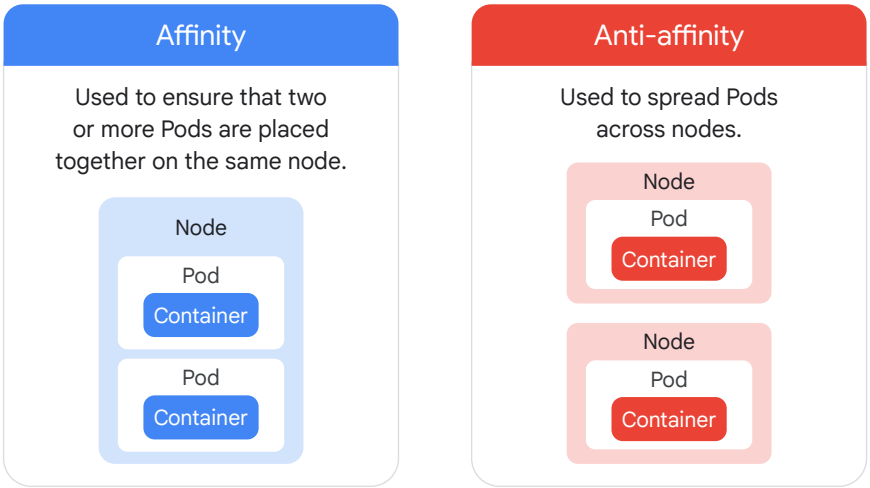
Autopilot can do this for you, just be sure to add the cloud.google.com/compute-class label in the **nodeSelector** rule for the Pods to be placed on a specific compute class. This can also be accomplished with node affinity rules.

A node's capabilities must match a Pod's preferences

If a node's capabilities match the Pod's preferences, then the node is considered a suitable host for that Pod.

As an example, let's say a Pod needs a node from the "`ssd`" node pool to run. In this case, if a node doesn't have an "`ssd`" label, the Pod will stay in waiting mode. However, please note that future label changes won't impact Pods that are already running.

Affinity rules influence Pod placement

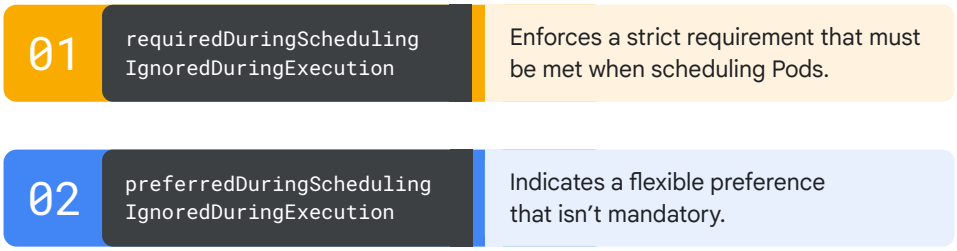


Like NodeSelectors, affinity rules can also be used to influence Pod placement. Node affinity and anti-affinity are label-based Pod placement approaches, but with more flexible and expressive rules.

Affinity rules can be used to ensure that two or more pods are placed together on the same node, and anti-affinity rules can be used to spread Pods across different nodes.

Unlike nodeSlectors, which only schedule Pods if they meet requirements, node affinity lets you define rules as preferences instead of requirements.

Affinity and anti-affinity rules keywords

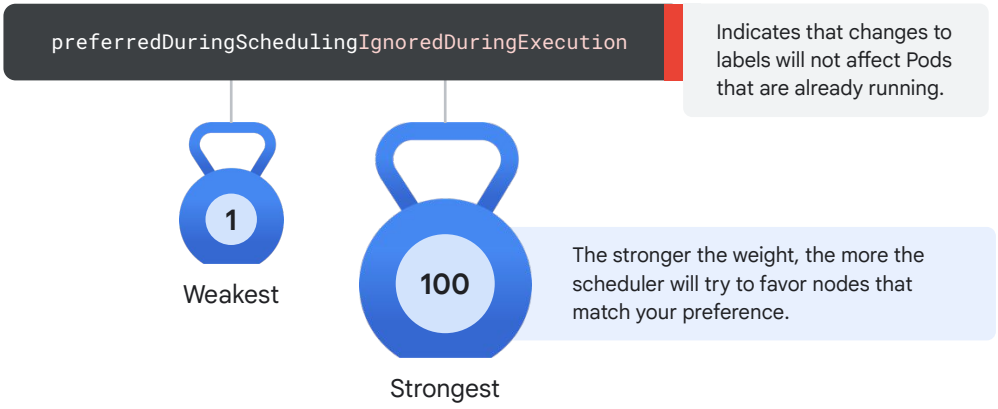


Affinity and anti-affinity rules can be expressed by using two specific keywords:

- 'requiredDuringSchedulingIgnoredDuringExecution'
- And 'preferredDuringSchedulingIgnoredDuringExecution'

'RequiredDuringScheduling' enforces a strict requirement that must be met when scheduling Pods, whereas 'PreferredDuringScheduling' indicates a flexible preference that isn't mandatory.

Preference weights



You can set a weight for this preference, where 1 is the weakest and 100 is the strongest. The stronger the weight, the more the scheduler will try to favor nodes that match your preference.

'IgnoredDuringExecution' indicates that changes to labels will not affect Pods that are already running.

Scenario 1

The NodeSelectorTerms field is used and the node must meet all matchExpression requirements.

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchExpressions:
      - key: accelerator-type
        operator: In
        values:
        - gpu
        - tpu
```

Example of affinity: a single `matchExpression` `–accelerator-type–` is specified.

`In` operator indicates only one of the listed values must match the preferences.

Now let's say there is a scenario where the NodeSelectorTerms field is used and the node must meet all matchExpression requirements.

This is an example of affinity, where a single matchExpression `–accelerator-type` – is specified. Logically, the matchExpression values are joined using the boolean `AND`.

From there, the `In` operator would indicate that only one of the listed values must match the preferences.

Other operators, like 'NotIn', can be used to configure a node anti-affinity rule.

Scenario 2

Defining the intensity of preference.

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 90
    preference:
    - matchExpressions:
      - key: cloud.google.com/gke-nodepool
        operator: In
        values:
        - n1-highmem-4
        - n1-highmem-8
```

A weight of 90 signals a strong preference.

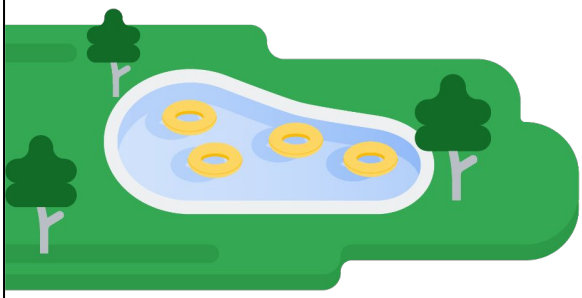
When the scheduler assigns the Pod to the node, it takes this preference and weight into account.

Let's explore another example, this time one that indicates a flexible preference.

You might recall that you can set a weight for a preference. So if the weight has been set to 90, it signals a strong preference. After the scheduler evaluates each available node, it will assign the Pod to the node that earns the highest overall score, and it will take this preference and its weight into account.

The node affinity rules in this example are set to express a strong preference for nodes that are in the n1-highmem-4 or n1highnem-8 node pools.

Node pool names



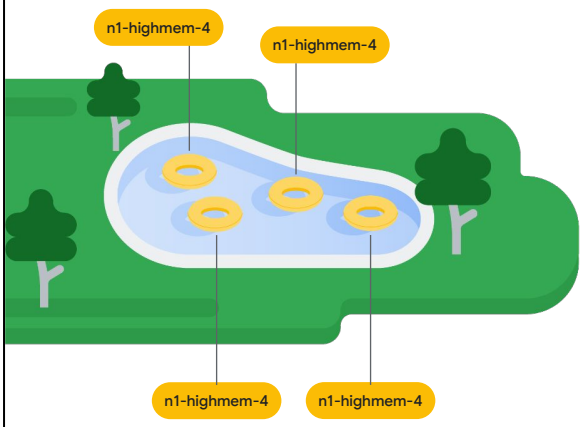
`n1-highmem-4`

Node pool names should indicate the type of compute instances that will be used to create the nodes.

- Descriptive
- Reflects hardware specs

It's recommended that node pool names indicate the type of compute instances that will be used to create the nodes. All nodes within a node pool share identical configurations, so a descriptive name accurately reflects their hardware specifications.

GKE assigns labels based on node pool names



n1-highmem-4

Easily create preferences for specific hardware types using those labels during Pod scheduling.

Also, GKE automatically assigns labels to nodes based on their node pool names. This empowers you to easily create preferences for specific hardware types using those labels during Pod scheduling.

Affinity and anti-affinity rules applications

```
[...]
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
    podAffinityTerm:
    labelSelector:
    matchExpressions:
      - key: app
        operator: In
        values:
        - webserver
    topologyKey: topology.kubernetes.io/zone
```

Affinity and anti-affinity rules:

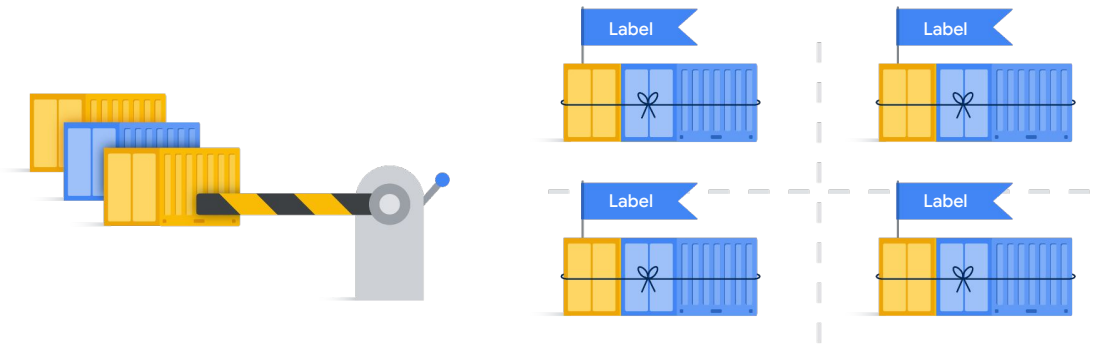
- Aren't limited to individual nodes.
- Can be applied at broad levels of infrastructure organization.

`topologyKeys` let you define preferences based on topology domains (e.g. zones and regions).

Affinity and anti-affinity rules aren't limited to individual nodes—they can be applied at broader levels of infrastructure organization. You can achieve this by using topologyKeys, which allow you to define preferences based on broader topology domains like zones and regions.

For example, you can use a topologyKey to indicate that you prefer not to schedule a Pod to a webserver zone if other Pods are already running in that zone.

Granular control with inter-pod affinity and anti-affinity



And if you need more granular control, inter-pod affinity and anti-affinity features can be used, because they consider the labels of other Pods already running on nodes.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Taints and tolerations

Taint

A special label that acts as a restriction indicating node suitability for a Pod.

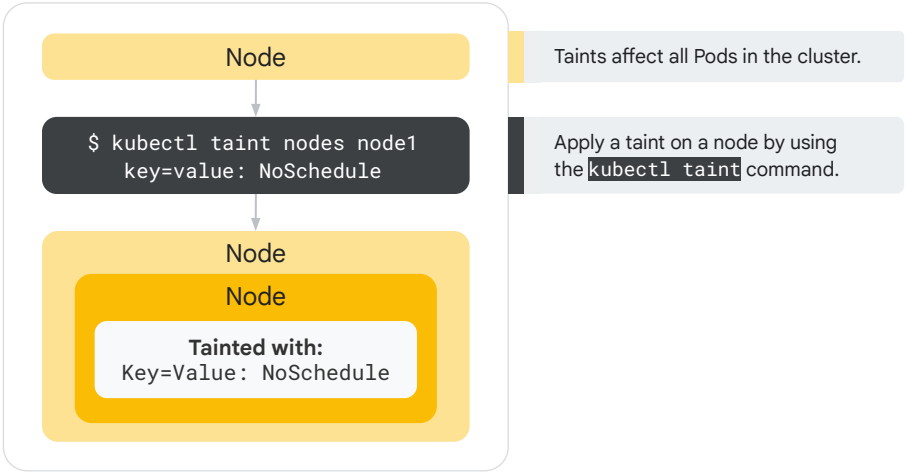
Toleration

A specification that allows the Pod to be scheduled onto a node that has a matching taint.

A taint is a special label applied to a node that acts as a restriction. It indicates that the node is not suitable for running certain Pods unless those Pods specifically tolerate the taint.

A toleration, therefore, is a specification within a Pod's configuration that allows it to be scheduled onto a node that has a matching taint.

Taints are configured on nodes

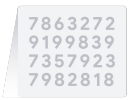


Unlike NodeSelector, affinity, and anti-affinity rules, taints are configured on nodes instead of Pods, and they affect all Pods in the cluster. You can apply a taint on a node by using the kubectl taint command.

A taint consists of three elements

Key

Gives a descriptive name that represents the taint's purpose.



Value

Provides optional information that further clarifies the taint.

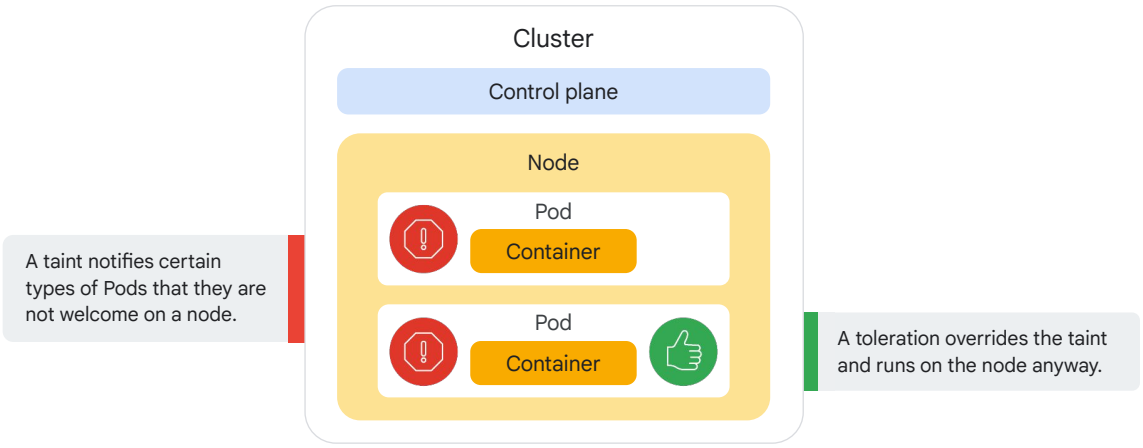
Effect

Prevents or discourages Pods from being scheduled on tainted nodes.

A taint consists of three elements:

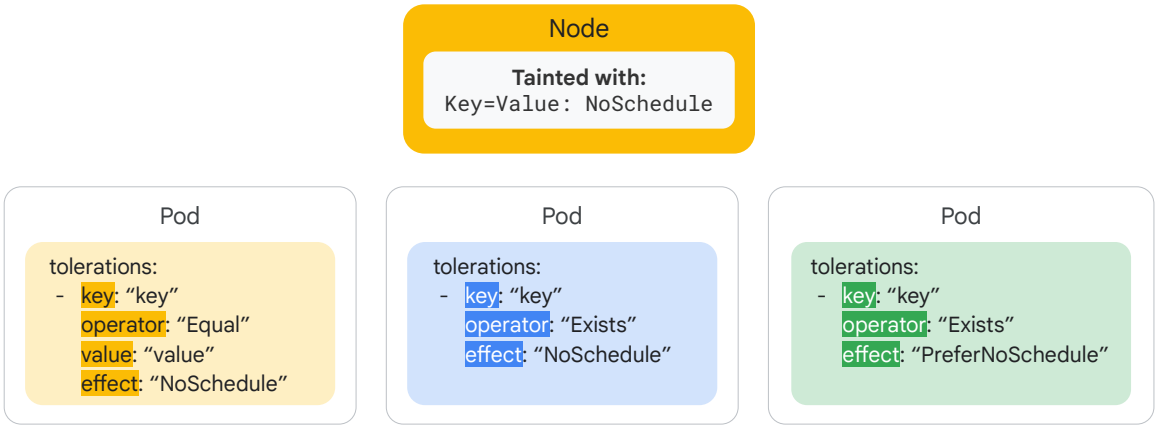
- A key, which gives a descriptive name that represents the taint's purpose.
- A value, which provides optional information that further clarifies the taint.
- And an effect to prevent or discourage Pods from being scheduled on tainted nodes.

A toleration acts like an exception



When a node has a taint, it notifies certain types of Pods that they are not welcome on a node. A **toleration** on a Pod acts like an exception, and it allows it to override the taint and run on the node anyway.

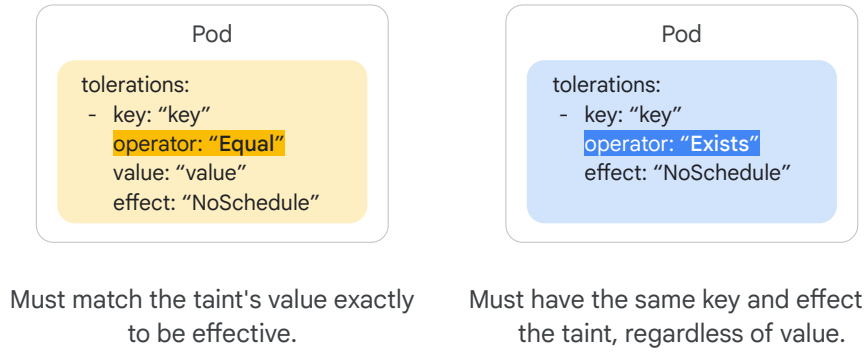
A toleration consists of four components



A toleration field consists of four components: a key, an operator, a value, and an effect, and operator.

A Pod can only override a taint if its toleration has the same key, effect, and passes the value check using its operator.

A toleration's operator



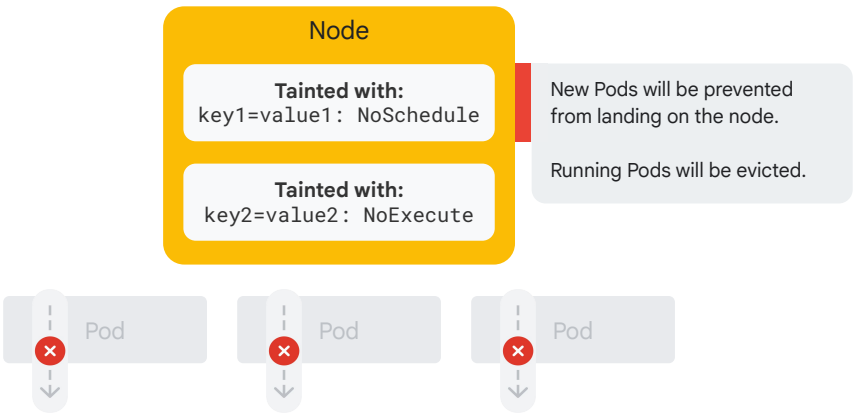
Must match the taint's value exactly to be effective.

Must have the same key and effect as the taint, regardless of value.

If a toleration's operator is "Equal", its value must match the taint's value exactly to be effective.

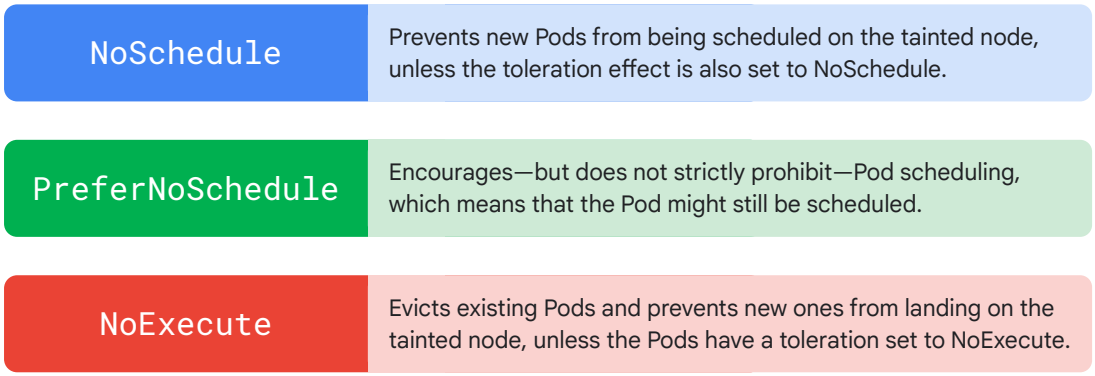
If a toleration's operator is "Exists", it only needs to have the same key and effect as the taint, regardless of value.

What happens when multiple taints are applied to a node



And if multiple taints are applied to a node, new Pods will be prevented from landing on the node, and running Pods will be evicted.

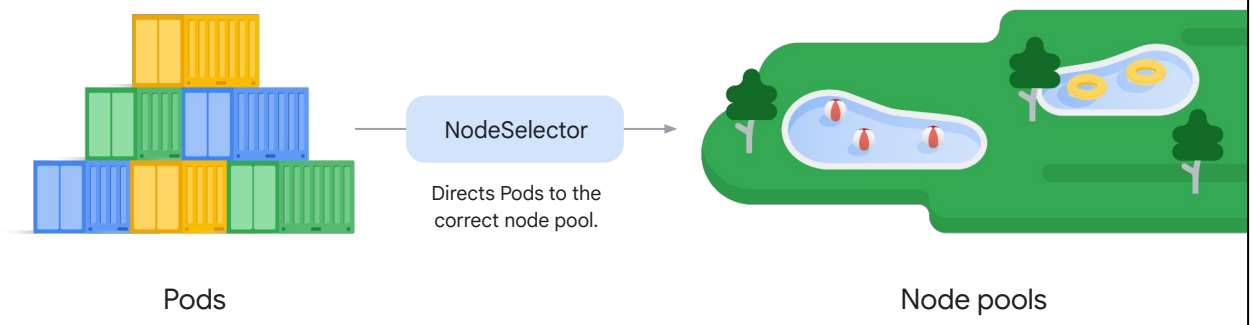
Three effect settings for taints and tolerations



There are three effect settings that can be applied to taints and tolerations:

- **NoSchedule** prevents new Pods from being scheduled on the tainted node, unless the toleration effect is also set to NoSchedule.
- **PreferNoSchedule** encourages–but does not strictly prohibit–Pod scheduling, which means that the Pod might still be scheduled.
- **NoExecute** evicts existing Pods and prevents new ones from landing on the tainted node, unless the Pods have a toleration set to NoExecute.

GKE uses node pools to place Pods on the correct hardware

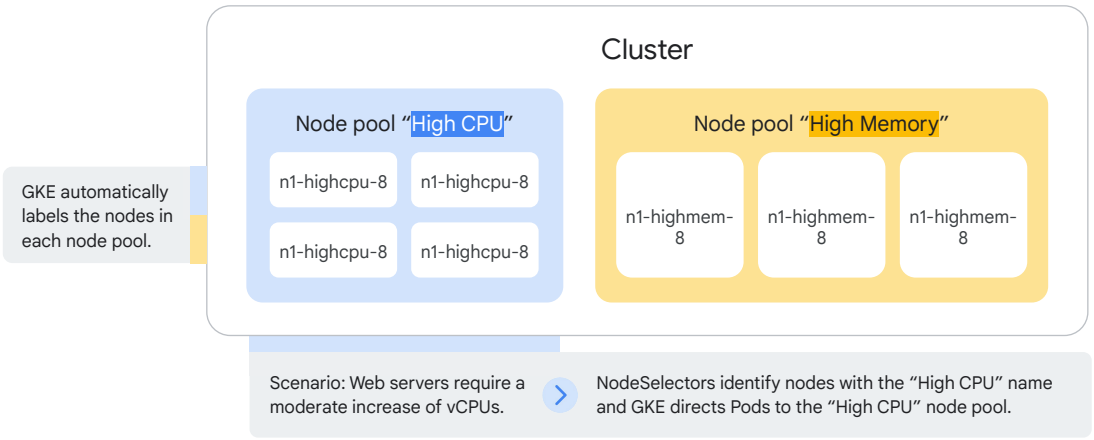


Constraining Pods to particular nodes adds additional complexity to cluster management.

GKE helps abstract some of that complexity by using node pools to place Pods on the correct hardware.

This is because node pools have the same hardware, and NodeSelectors can be used to direct Pods to the correct node pool.

NodeSelectors can direct Pods to nodes



Let's demonstrate this concept through an example of a GKE cluster that has two node pools, one named "High CPU" and the other named "High Memory."

GKE automatically labels the nodes in each node pool with the specified name, so that NodeSelectors can direct Pods to the appropriate nodes.

Imagine that a fleet of frontend web servers requires a moderate increase of vCPUs relative to memory. In this case, the nodeSelectors identify nodes with the "High CPU" name and GKE will direct those Pods to the "High CPU" node pool.

Workloads: Deployments and Jobs

01 Configure, manage, and update Deployments

02 Jobs and CronJobs

03 Scale clusters

04 Control Pod placement with labels and affinity rules

05 Control Pod placement with taints and tolerations

06 Get software into a cluster

Tools to help get software into a cluster

Cloud Build

Serverless tool to build, test, and deploy software across various environments and programming languages.

Artifact Registry

Provides a single location to store, manage, and secure build artifacts.

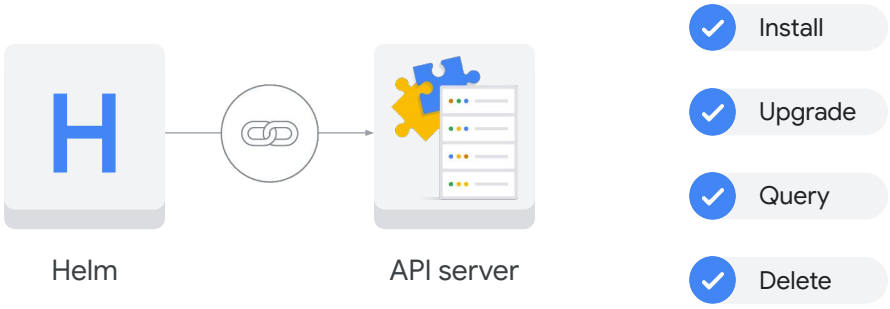
Helm

An open-sourced package manager that simplifies application management.

Google Cloud tools like Cloud Build, Artifact Registry, and Helm help get software into your cluster. But *you* are responsible for defining deployment patterns and services for reliable and efficient operation.

- **Cloud Build** is a serverless tool that helps you build, test, and deploy software across various environments and programming languages.
- **Artifact Registry** provides a single location to store, manage, and secure build artifacts, like container images. GKE can fetch these images from the registry and then run them in Pods.
- **Helm** is an open-sourced, package manager that simplifies application management.
  - Helm provides traditional software installation and management functionalities similar to what apt-get and yum provide for Linux.
  - Developers can use Helm to organize Kubernetes objects into packages called “charts.” Charts manage the deployment of complex applications, and can be versioned, shared, and published. And they also manage the installation of required dependencies.

Helm and the API server



Helm and the API server work together to install, upgrade, query, and remove Kubernetes resources.

Helm must be managed

Google Cloud Marketplace

- Development stacks
- Solutions
- Services

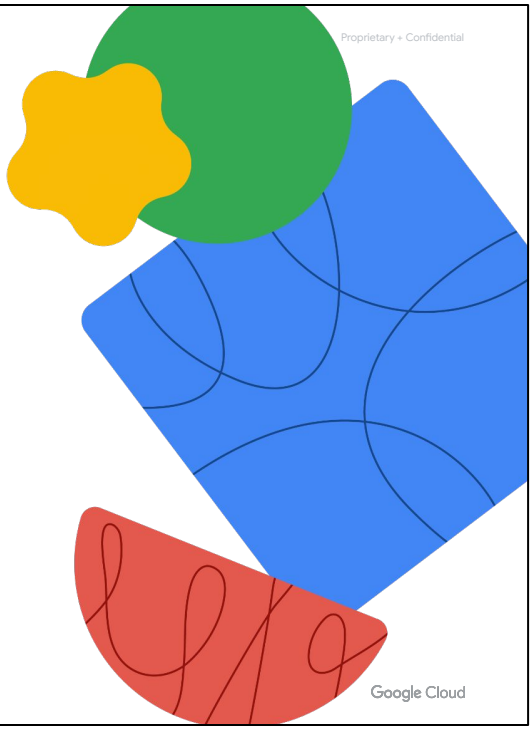
Helm makes open-source software deployment easier and reduces the risk of error, but the tool still needs to be managed.

Google Cloud Marketplace offers development stacks, solutions, and services to help manage and accelerate development.

And thanks to kublectl commands and Helm charts, installation is automated.

Quiz questions

Let's pause for a quick check in.



Quiz | Question 1

Question

After a Deployment has been created and its component Pods are running, which component is responsible for ensuring that a replacement Pod is launched whenever a Pod fails or is evicted?

A. ReplicaSet

B. Deployment

C. StatefulSet

D. DaemonSet

Quiz | Question 1

Answer

After a Deployment has been created and its component Pods are running, which component is responsible for ensuring that a replacement Pod is launched whenever a Pod fails or is evicted?

A. ReplicaSet

B. Deployment

C. StatefulSet

D. DaemonSet

After a Deployment has been created and its component Pods are running, which component is responsible for ensuring that a replacement Pod is launched whenever a Pod fails or is evicted?

A. **ReplicaSet**: This is the **correct answer** because a **ReplicaSet** ensures a specified number of identical Pods are running at any given time. .
B. Deployment: This is the **incorrect answer** because a **deployment** manages the creation, scaling, and updating of pods..
C. StatefulSet: This is the **incorrect answer** because a **StatefulSet** helps you manage stateful applications, which are applications that require persistent storage and a stable network identity.
D. DaemonSet: This is the **incorrect answer** because a **DaemonSet** ensures a copy of a Pod runs on all (or some) Nodes within your cluster. It's particularly useful for deploying background processes.

Quiz | Question 2

Question

Which of the following commands will display the desired, current, up-to-date, and available status of all the ReplicaSets within a Deployment?

A. kubectl describe

B. kubectl logs

C. kubectl get

D. Google Cloud Console

Quiz | Question 2

Answer

Which of the following commands will display only the desired, current, up-to-date, and available status of all the ReplicaSets within a Deployment?

A. kubectl describe
B. kubectl logs
C. kubectl get 
D. Google Cloud Console

Which of the following commands will display the desired, current, up-to-date, and available status of all the ReplicaSets within a Deployment?

A. kubectl describe: This is the **incorrect answer** because **kubectl describe** shows the details of a specific resource or resource type.
B. kubectl logs: This is the **incorrect answer** because **kubectl logs** allows you to view the logs of a specific Pod or container within a Pod.
C. **kubectl get**: This is the **correct answer** because **kubectl get** is used to retrieve information about resources within your Kubernetes cluster..
D. Google Cloud Console: This is the **incorrect answer** because the **Google Cloud Console** is a way to view the current structure of your project and not an actual command..

Quiz | Question 3

Question

You are resolving a range of issues with a Deployment and need to make a large number of changes. Which command can you execute to group these changes into a single rollout, thus avoiding pushing out a large number of rollouts?

A. kubectl rollout resume deployment

B. kubectl delete deployment

C. kubectl stop deployment

D. kubectl rollout pause deployment

Quiz | Question 3

Answer

You are resolving a range of issues with a Deployment and need to make a large number of changes. Which command can you execute to group these changes into a single rollout, thus avoiding pushing out a large number of rollouts?

A. kubectl rollout resume deployment
B. kubectl delete deployment
C. kubectl stop deployment
D. kubectl rollout pause deployment 

You are resolving a range of issues with a Deployment and need to make a large number of changes. Which command can you execute to group these changes into a single rollout, thus avoiding pushing out a large number of rollouts?

A. kubectl rollout resume deployment: This is the **incorrect answer** because **kubectl rollout resume deployment** resumes a paused Deployment.
B. kubectl delete deployment: This is the **incorrect answer** because **kubectl delete deployment** deletes a deployment.
C. kubectl stop deployment: This is the **incorrect answer** because **kubectl stop deployment** is not actually a valid command. However, you can achieve a similar effect by scaling the deployment down to zero replicas.
D. **kubectl rollout pause deployment**: This is the **correct answer** because **kubectl rollout pause deployment** will pause a Deployment being rolled out.

.

Quiz | Question 4

Question

You are configuring a Job to convert a sample of a large number of video files to a different format. Which parameter should you configure to stop the process once a sufficient quantity is reached?

A. parallelism=4

B. completions=4

C. backofflimit=4

D. replicas=4

Quiz | Question 4

Answer

You are configuring a Job to convert a sample of a large number of video files to a different format. Which parameter should you configure to stop the process once a sufficient quantity is reached?

A. parallelism=4
B. completions=4 
C. backofflimit=4
D. replicas=4

You are configuring a Job to convert a sample of a large number of video files to a different format. Which parameter should you configure to stop the process once a sufficient quantity is reached?

A. parallelism=4: This is the **incorrect answer** because the **parallelism** value specifies the maximum number of Pods that can run simultaneously to execute the Job's tasks.
B. **completions=4**: This is the **correct answer** because the **completions** value specifies the number of successfully finished Pods that need to be reached before the Job is considered complete.
C. backofflimit=4: This is the **incorrect answer** because the **backofflimit** value controls how many times the Job controller should attempt to restart failed Pods before considering the Job itself as failed.
D. replicas=4: This is the **incorrect answer** because the **replicas** value is not a valid field for a GKE job.

Quiz | Question 5

Question

How do you configure a Kubernetes Job so that Pods are retained after completion?

A. Configure the cascade flag for the Job with a value of false.
B. Configure the backofflimit parameter with a non-zero value.
C. Set an activeDeadlineSeconds value high enough to allow you to access the logs.
D. Set a startingDeadlineSeconds value high enough to allow you to access the logs.

Quiz | Question 5

Answer

How do you configure a Kubernetes Job so that Pods are retained after completion?

A. Configure the cascade flag for the Job with a value of false. 

B. Configure the backofflimit parameter with a non-zero value.

C. Set an activeDeadlineSeconds value high enough to allow you to access the logs.

D. Set a startingDeadlineSeconds value high enough to allow you to access the logs.

How do you configure a Kubernetes Job so that Pods are retained after completion?

A. **Configure the cascade flag for the Job with a value of false.**: This is the **correct answer** because in GKE, when you complete a Job, the Job's Pods are also deleted by default. This is referred to as cascading deletion. You can prevent this behavior by setting the flag to false. This will delete the Job but leave its Pods running.
B. Configure the backofflimit parameter with a non-zero value.: This is the **incorrect answer** because because the **backofflimit** value controls how many times the Job controller should attempt to restart failed Pods before considering the Job itself as failed..
C. Set an activeDeadlineSeconds value high enough to allow you to access the logs.: This is the **incorrect answer** because **activeDeadlineSeconds** sets a maximum time limit for the job to actively run. If the job exceeds this limit, its pods are terminated and the job is marked as failed..
D. Set a startingDeadlineSeconds value high enough to allow you to access the logs.: This is the **incorrect answer** because **startingDeadlineSeconds** is not used in the GKE jobs.

Quiz | Question 6

Question

You have autoscaling enabled on your cluster. What conditions are required for the autoscaler to decide to delete a node?

A. If a node is underutilized and there are no Pods currently running on the Node.

B. If the overall cluster is underutilized, the least busy node is deleted.

C. If the overall cluster is underutilized, a randomly selected node is deleted.

D. If a node is underutilized and running Pods can be run on other Nodes.

Quiz | Question 6

Answer

You have autoscaling enabled on your cluster. What conditions are required for the autoscaler to decide to delete a node?

A. If a node is underutilized and there are no Pods currently running on the Node.
B. If the overall cluster is underutilized, the least busy node is deleted.
C. If the overall cluster is underutilized, a randomly selected node is deleted.
D. If a node is underutilized and running Pods can be run on other Nodes.

You have autoscaling enabled on your cluster. What conditions are required for the autoscaler to decide to delete a node?

A. If a node is underutilized and there are no Pods currently running on the Node.: This is the **incorrect answer** because even though the node is underutilized., unless all pods running on the node can be moved to other nodes it will not be deleted.
B. If the overall cluster is underutilized, the least busy node is deleted.: This is the **incorrect answer** because this Node may not be able to be run its Pod on another Node.
C. If the overall cluster is underutilized, a randomly selected node is deleted.: This is the **incorrect answer** because this would risk deleting a node currently executing work.
D. **If a node is underutilized and running Pods can be run on other Nodes.**: This is the **correct answer** because the Pods can be run on other Nodes..

Quiz | Question 7

Question

Inter-pod affinity rules are specified at the zone level, not at the individual Node level. To apply this override, which additional parameter must be configured in the Pod manifest YAML?

A. zone: topology.kubernetes.io/zone

B. topologyKey: topology.kubernetes.io/zone

C. labels: topology.kubernetes.io/zone

D. matchLabels: topology.kubernetes.io/zone

Quiz | Question 7

Answer

Inter-pod affinity rules are specified at the zone level, not at the individual Node level. To apply this override, which additional parameter must be configured in the Pod manifest YAML?

A. zone: topology.kubernetes.io/zone

B. topologyKey: topology.kubernetes.io/zone 

C. labels: topology.kubernetes.io/zone

D. matchLabels: topology.kubernetes.io/zone

Inter-pod affinity rules are specified at the zone level, not at the individual Node level. To apply this override, which additional parameter must be configured in the Pod manifest YAML?

A. zone: topology.kubernetes.io/zone: This is the **incorrect answer** because the **zone** flag is not used in affinity specification
B. **topologyKey: topology.kubernetes.io/zone**: This is the **correct answer** because the **topologyKey** flag is the correct way to define Zonal topology.
C. labels: topology.kubernetes.io/zone: This is the **incorrect answer** because the **labels** flag is used to specify identifying attributes of objects.
D. matchLabels: topology.kubernetes.io/zone: This is the **incorrect answer** because the **matchLabels** flag acts as a filter, allowing you to precisely control where Pods are placed based on their labels and the labels on Nodes.