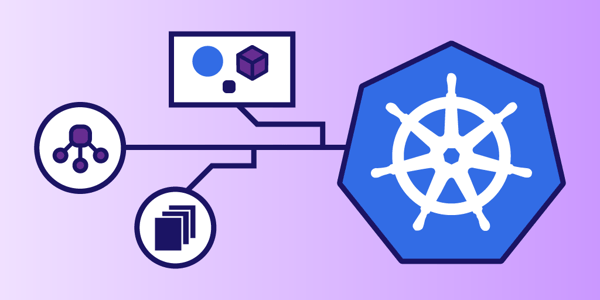
**A deep dive into Kubernetes Deployment strategies**

Nov 19, 2020 - 11 min read

Ryan Thelin



With the use of containerized applications and microservices on the rise, there’s never been a better time to [become a Kubernetes expert](https://www.educative.io/catalog/kubernetes).

Kubernetes is an open-source **container orchestration system** designed to automate application scaling and management. While Kubernetes is incredibly powerful, it’s also notoriously difficult to learn.

Today, we’ll get you one step closer to Kubernetes expertise with a walkthrough of the **Deployment object**, Kubernetes’ vehicle for ReplicaSet management, and live system updates.

**Here’s what we’ll cover today:**

* [What is a Deployment in Kubernetes?](https://www.educative.io/blog/kubernetes-deployments-strategies#what)
* [Deployment update strategies](https://www.educative.io/blog/kubernetes-deployments-strategies#strategies)
* [Rolling update strategy](https://www.educative.io/blog/kubernetes-deployments-strategies#rolling)
* [Recreate update strategy](https://www.educative.io/blog/kubernetes-deployments-strategies#recreate)
* [Canary update strategy](https://www.educative.io/blog/kubernetes-deployments-strategies#canary)
* [What to learn next](https://www.educative.io/blog/kubernetes-deployments-strategies#next)

**Brush up on top Kubernetes concepts**

**Become an in-demand Kubernetes expert with hands-on lessons and practice build right into your browser.**

[**Learn Kubernetes: A Deep Dive**](https://www.educative.io/courses/the-kubernetes-course)

**What is a Deployment in Kubernetes?**

A Deployment is a **resource object** in Kubernetes that defines the **desired state** for your program.

Deployments are declarative, meaning that you don’t dictate how to achieve the state. Instead, you declare your **endpoint** (called a “desired state”) and allow the Deployment-controller to automatically reach that end goal in the most efficient way.

Once running, the Deployment compares the current program state to the desired state. If they do not match, the Deployment-controller automatically alters the state to match.

This automatic state maintenance is what gives Kubernetes it’s beloved self-healing properties.

Desired states can include how many Pods are running, what type of Pods those are, what container images are available to the program, and the desired workload for each Pod.

If any aspect of the desired state is missing, the Deployment-controller will alter the program until they are met.

**ReplicaSets**

Relationship between Deployments, ReplicaSets, and Pods

The main way Deployments maintain a program’s desired state is through the use of **ReplicaSets**.

A ReplicaSet is a set of **identical backup Pods** maintained on the backend side to ensure a Pod is always available. If a user-facing Pod fails or becomes overworked, the Deployment allocates work to a Pod from the ReplicaSet to maintain responsiveness.

If a Pod from the ReplicaSet fails, it automatically creates an additional Pod from the template.

The Pods in a ReplicaSet are designed using a **single Pod template** provided by the Deployment that defines the specifications shared by the Pod cluster. The template specifications are properties like:

* What applications should run in the Pods?
* What labels should the Pods have?
* Under what conditions will the Pod restart?

It’s best practice to not manage ReplicaSets directly. You should perform all actions against the Deployment object and leave the Deployment to manage ReplicaSets.

Each Deployment can only manage a single Pod template but can manage multiple Replica Pods from the same template.

You’ll need to create multiple Deployments to maintain multiple different ReplicaSets.

**Updating with Deployments**

The main advantage of Deployments is for **automatically updating** your Kubernetes program.

Without Deployments, you’d have to manually end all old Pods, start new Pod versions and run a check to see if there were any problems during Pod creation.

Deployments automate the whole updating process as you can simply update the Pod template or desired state. The Deployment will alter the program state in the background with actions like creating new Pods, allocating more resources, and so on, until the updated desired state is met.

You can even **rollback updates** to a previous version. Old ReplicaSets still exist with full Pod configurations but simply don’t manage any Pods once a new ReplicaSet is made.

If you want to rollback to a previous version, you simply need to change the desired state to favor the old ReplicaSet and the Deployment will automatically revert.

**Update Deployment Strategies**

Kubernetes offers Deployment strategies that allow you to update in a variety of ways depending on the needs of the system. The three most common are:

* **Rolling update strategy**: Minimizes downtime at the cost of update speed.
* **Recreation Strategy**: Causes downtime but updates quickly.
* **Canary Strategy**: Quickly updates for a select few users with a full rollout later.

Let’s take a deeper look at each of these three strategies!

**Rolling update strategy**

The rolling update strategy is a gradual process that allows you to update your Kubernetes system with only a minor effect on performance and no downtime.

Rolling update strategy flowchart

In this strategy, the Deployment selects a Pod with the old programming, deactivates it, and creates an updated Pod to replace it. The Deployment repeats this process until no outdated Pods remain.

The advantage of the rolling update strategy is that the update is applied **Pod-by-Pod** so the greater system can remain active.

There is a minor performance reduction during this update process because the system is consistently one active Pod short of the desired number of Pods. This is often much preferred to a full system deactivation.

The rolling update strategy is used as the **default update strategy** but isn’t suited for all situations. Some considerations when deciding to use a rolling update strategy are:

* How will my system react to momentarily duplicated Pods?
* Is the update substantial enough to malfunction with some Pods still running old specifications?
* Will a minor performance reduction greatly affect the usability of my system? How finely time-sensitive is my system?

For example, imagine we wanted to change the specifications for our Pods. We’d first change the Pod template to new specifications, which is passed from the Deployment to the ReplicaSet.

The Deployment would then recognize that the current program state (Pods with old specifications) is different from the desired state (Pods with new specifications).

The Deployment would create Pods and a ReplicaSet with the updated specifications and transfer workload one-by-one from the old Pods to the new Pods.

By the end, we’ll have an entirely new set of Pods and ReplicaSet without any service downtime.

**Rolling Update Implementation**

We’ll use a YAML file declaration named deploy.yaml to create our Deployment.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

apiVersion: apps/v1  *#Older versions of k8s use apps/v1beta1*

kind: Deployment

metadata:

  name: hello-deploy

spec:

  replicas: 10

  selector:

    matchLabels:

      app: hello-world

  minReadySeconds: 10

  strategy:

    type: RollingUpdate

    rollingUpdate:

      maxUnavailable: 1

      maxSurge: 1

  template:

    metadata:

      labels:

        app: hello-world

    spec:

      containers:

      - name: hello-Pod

        image: educative/k8sbook:latest

        ports:

        - containerPort: 8080





Right at the very top, you specify the Kubernetes API version to use. Assuming that you’re using an up-to-date version of Kubernetes, Deployment objects are in the apps/v1 API group.

Next, the .kind field tells Kubernetes you’re defining a Deployment object.

The .metadata section is where we give the Deployment a name and labels.

The .spec section is where most of the action happens. Anything directly below .spec relates to the Pod. Anything nested below .spec.template relates to the Pod template that the Deployment will manage. In this example, the Pod template defines a single container.

* .spec.replicas tells Kubernetes how many Pod replicas to deploy.
* spec.selector is a list of labels that Pods must have in order for the Deployment to manage them.
* .spec.strategy tells Kubernetes how to perform updates to the Pods managed by the Deployment, in this case RollingUpdate.

Finally, we’ll apply this Deployment to our Kubernetes cluster with the command:

$ kubectl apply -f deploy.yml

**Keep learning Kubernetes.**

Don’t stop with just Deployments. Standout from your peers by mastering Kubernetes concepts from beginner to expert. Educative’s text-based courses are skimmable and loaded with hands-on practice.

[**Learn Kubernetes: A Deep Dive**](https://www.educative.io/courses/the-kubernetes-course)

**Recreate update strategy**

The recreate update strategy is an all-or-nothing process that allows you to update all aspects of the system at once with a brief downtime period.

Recreate update strategy flowchart

In this strategy, the Deployment selects all **outdated Pods** and deactivates them at once.

Once all old Pods are deactivated, the Deployment creates updated Pods for the entire system. The system is inoperable starting at the old Pod’s deactivation and ending once the final updated Pod is created.

The recreate strategy is used for systems that cannot function in a partially updated state or if you would rather have downtime than provide users a lesser experience. The bigger the update, the more likely a rolling update will cause an error.

Therefore, recreate strategy is better for large updates and overhauls.

When you’re considering the recreate strategy, ask yourself:

* Would my users have a better experience with downtime or temporarily reduced performance?
* Could my system function during a rolling update?
* Is there a time I could update the system without affecting a significant number of users?

**Recreate Update Implementation**

This implementation is very similar to the rolling update strategy.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

apiVersion: apps/v1  *#Older versions of k8s use apps/v1beta1*

kind: Deployment

metadata:

  name: hello-deploy

spec:

  replicas: 10

  selector:

    matchLabels:

      app: hello-world

  minReadySeconds: 10

  strategy:

    type: Recreate

    rollingUpdate:

      maxUnavailable: 1

      maxSurge: 1

  template:

    metadata:

      labels:

        app: hello-world

    spec:

      containers:

      - name: hello-Pod

        image: educative/k8sbook:latest

        ports:

        - containerPort: 8080





As you can see, the only difference between the implementation of rolling update and recreate is on line 12 where we’ve replaced strategy.type: RollingUpdate to strategy.type: Recreate.

Just as last time, we’ll deploy the Deployment via the command-line using:

$ kubectl apply -f deploy.yml

**Canary update strategy**

The canary update strategy is a partial update process that allows you to test your new program version on a real userbase without a commitment to a full rollout.

Both steps of the canary update strategy

In this strategy, the Deployment creates a **few new Pods** while keeping most Pods on the previous version, usually at a **1:4 ratio**.

Most users still use the previous version, but a small subset unknowingly use the new version to act as testers.

If we don’t detect any bugs from this subset, we can scale up the updated ReplicaSet to produce a full rollout.

If we do find a bug, we can easily rollback the few updated Pods until we’ve fixed the bug.

The advantage of the canary update strategy is it allows you to test a new version without the risk of a full system failure.

In the worst-case scenario, all users from the test subset experience critical errors while 75% or more of the user base continues without interruption.

The rollback process is also much quicker than the rolling update strategy because you only have to rollback a portion of the Pods rather than the entire system.

The downside is that the updated Pods will require a **separate Deployment**, which can be hard to manage at scale. Also, the canary strategy results in a slower rollout due to the waiting period after rollout to our initial subset and the completion of a full rollout.

When considering the canary strategy, ask yourself:

* What’s the worst-case scenario if this update fails?
* How soon do I need to finish the full rollout?
* How much internal testing have I done?

**Canary Update Implementation**

For this implementation, we’ll need to create two Deployments in two YAML files.

**Version 1 Deployment**

Our first file, k8s-deployment.yaml, will be our outdated version that most of our Pods will run.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

apiVersion: apps/v1

kind: Deployment

metadata:

  name: helloworld

spec:

  replicas: 3

  strategy:

    rollingUpdate:

    maxSurge: 1

    maxUnavailable: 1

  minReadySeconds: 5

  template:

    metadata:

      labels:

        app: helloworld

        track: stable

    spec:

      containers:

      - name: helloworld

        image: educative/helloworld:1.0

        ports:

        - containerPort: 80

        resources:

          requests:

            cpu: 50m

          limits:

            cpu: 100m





This will create 3 Pods of v1 with the app:helloworld label that our Kubernetes service is looking for. Our image for these Pods is educative/helloworld:1.0, meaning that these Pods will be created off the old Pod specifications.

This Deployment will evenly divide any workload among available Pods.

You will deploy this by entering this line into the command-line:

kubectl apply -f k8s-deployment.yaml

**Notice:** Unlike the previous implementations, canary is not listed under strategy as its implementation is more complicated.

Instead, both versions are listed as rollingUpdates because updates within each version will be rollout while the overall system’s strategy is canary.

**Version 2 Deployment**

Now, we’ll create our second yaml file, k8s-deployment-canary.yaml, which is our new canary v2 version.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

apiVersion: apps/v1

kind: Deployment

metadata:

  name: helloworld-canary

spec:

  replicas: 1

  strategy:

    rollingUpdate:

      maxSurge: 1

      maxUnavailable: 1

  minReadySeconds: 5

  template:

    metadata:

      labels:

        app: helloworld

        track: canary

    spec:

      containers:

      - name: helloworld

        image: educative/helloworld:2.0

        ports:

        - containerPort: 80

        resources:

          requests:

            cpu: 50m

          limits:

            cpu: 100m





For this Deployment, we only create a single Pod (line 6) to ensure most of our userbase still interacts with v1.

Both Deployments will balance the workload among all Pods which ensures only 25% of our workload will be on the updated Pod.

You can deploy this Deployment via the command-line using:

kubectl apply -f k8s-deployment-canary.yaml

Once you’re satisfied that v2 works, simply replace the image in our first Deployment YAML file, k8s-deployment.yaml, to be educative/helloworld:2.0 rather than educative/helloworld:1.0.

Then remove the canary Deployment with:

kubectl delete -f k8s-deployment-canary.yaml

Our desired state will then be to have all Pods with v2 and workload will be balanced among the remaining 3 v2 Pods.

Canary update achieved!

**Introduction to Kubernetes Volume**

Kubernetes volume is a directory or a container’s file system, which has a customizable information store. It is a function as a single point from which kubernetes can allocate resources and services throughout the network distribution. This customizable information store also gives administration access to managing its objects and their attributes at a single point. The Kubernetes lifecycle depends as long as the container is alive. This file system is available until the Container terminates and restarts. Kubernetes volumes are useful for storing temporary data as per the existence of pods. If necessary, the information store can be distributed among many pods differently. As the kubernetes volume is a file system that acts as an administrator, it can easily search for information from various pods throughout the network.

**Types of Kubernetes Volume**

**1. Local-Node directory memory**: The Local-node memory is attached to the pod, it is either stored in RAM or persistent storage. Due to their pod dependency, as the pod is running their content is available to use. The data is lost if it goes down.

**For example,** The emptyDir and hostPath are attached to the pod, stored either in storage or RAM.

**2. Cloudstorage volumes**: This type of volume placed outside of the pod which is independent of the application. it is used to preserve data. it has a lot of disadvantages over advantages. Please find the list below:-

1. It is difficult to handle because of their communication
2. The user must have sufficient knowledge about the storage before connecting to the Pod.

For example: – gcePersistentDisk, awsElasticBlockStore.

**3. File-sharing volumes**: In this volume type, the NFS is used to connect with volume through the YAML file. Initially, the content of the volume is unmounted to use but it’s available. Unlike the emptydir in NFS, the volume exists even the pod s not available.

**4. Distributed-file systems**: This type of volume has to be mounted into your pods. This volume can be pre-populated with data, and that data can be handed to respective Pods. For Example: glusterfs

Please find below yaml used for glusterfs:-

apiVersion: v1

kind: Pod

metadata:

name: glusterfs

spec:

containers:

- name: glusterfs

image: nginx

volumeMounts:

- mountPath: "/mnt/glusterfs"

name: glusterfsvol

volumes:

- name: glusterfsvol

glusterfs:

endpoints: glusterfs-cluster

path: kube\_vol

readOnly: true

**[WINDOWS POWERSHELL Course Bundle - 7 Courses in 1](https://www.educba.com/software-development/courses/powershell-training/?btnz=edu-in-between-para-banner" \t "_blank)**[26+ Hours of HD Videos | 7 Courses | Verifiable Certificate of Completion | Lifetime Access  
4.5](https://www.educba.com/software-development/courses/powershell-training/?btnz=edu-in-between-para-banner" \t "_blank)

Please find the output below:-

**5. Special volume types**: Persistent volume and PersistentVolumeClaim is special kind of volume which mainly used in kubernetes clusters.

Persistent volume (PV) is the piece of storage that has physical existence in the host machine to store your persisted data. The main feature of a PV is that it has an independent life cycle, and it continues to live when the pods accessing it have shut down. Each PV contains a spec and status, which is the specification and status of the volume. Please find the example below:-

apiVersion: v1

kind: PersistentVolume

metadata:

name: cba

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

nfs:

path: /temp

server: 127.0.0.1

Please find the output below:-

PV will have a specific storage capacity. This is set using the PV’s capacity attribute.

**Persistent volume (PV) Access Mode**

Each volume is supported by the specific mode of access modes given by PV. Please find the access modes details below:-

The different available access modes are:

**[SALESFORCE Course Bundle - 4 Courses in 1](https://www.educba.com/data-science/courses/salesforce-training-course/?btnz=edu-in-between-para-banner" \t "_blank)**[34+ Hours of HD Videos | 4 Courses | Verifiable Certificate of Completion | Lifetime Access  
4.5](https://www.educba.com/data-science/courses/salesforce-training-course/?btnz=edu-in-between-para-banner" \t "_blank)

**ReadWriteOnce –** If there is a single node, the volume can be mounted as read-write by the only node

**ReadOnlyMany –** If there is many nodes, the volume can be mounted read-only by many other nodes

**ReadWriteMany –** if there is any node the volume can be mounted as read-write by many other nodes.

**Phases of volume**

**Available –** If the free resources of the volume is not ready to claim, such a state of the volume is in the available state.

**Bound –** When the volume is bound to a claim by PersistentVolumeClaim. Such a state of volume is considered a bound state.

**Released –** When the volume resources are not available in the cluster, although the existing claim has been deleted. In this case, the volume is considered as in the Released state.

**Failed –** A volume is considered as in the failed state if the volume has failed to claim by its automatic process.

A persistent volume claim is a kind of request for the platform to create a Persistent volume in the applications. The communication between Persistent volume (PV) and Persistent volume claim (PVC) used to happen by their lifecycle process. Kubernetes looks for the condition defined in PVC and if there is one, it matches the claim to PV. This is called binding. We can provision the Persistent volume (PV) in two ways:-

**Static –** the cluster administration use to create the volume assign statically in the yaml file. The administration carries information about the storage which is available for cluster users.

**[MINITAB Course Bundle - 9 Courses in 1 | 2 Mock Tests](https://www.educba.com/data-science/courses/minitab-training/?btnz=edu-in-between-para-banner" \t "_blank)**[49+ Hours of HD Videos | 9 Courses | 2 Mock Tests & Quizzes | Verifiable Certificate of Completion | Lifetime Access  
4.5](https://www.educba.com/data-science/courses/minitab-training/?btnz=edu-in-between-para-banner" \t "_blank)

**Dynamic –** Clusters used to try dynamically provision the volume once the administrator created to match a user’s PersistentVolumeClaim. The pods can access storage by claiming volume. The pod’s namespace and their claim use to find by the cluster so that PersistentVolume can claim easily. The volume is then available to the host as well as the pod.

Please find the Configurations for Persistent volume claim (PVC)

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: claimVol

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 8Gis

Find the output below:-

Each PVC has a specification and status of the claim. Whenever PVC requested storage with specific access modes, Claims uses the same conventions as volumes.

**The Lifecycle of a Volume and Claim**

**Provisioning:** A cluster administrator will create several PVs. They carry the details of the storage which is available for use.

**Binding:** A user creates a PersistentVolumeClaim with a specific amount of storage requested by using any of its access modes.

**Using:** Pods usually do claim for their volumes. The cluster inspects the claim to find the bound volume and mounts the same for a pod.

**Releasing:** When a user is done with their volume, they can delete the PVC objects from the API which allows reclamation of the resource.

**[SAS PROGRAMMING Course Bundle - 18 Courses in 1 | 8 Mock Tests](https://www.educba.com/data-science/courses/sas-training-course/?btnz=edu-in-between-para-banner" \t "_blank)**[161+ Hours of HD Videos | 18 Courses | 8 Mock Tests & Quizzes | Verifiable Certificate of Completion | Lifetime Access  
4.5](https://www.educba.com/data-science/courses/sas-training-course/?btnz=edu-in-between-para-banner" \t "_blank)

**Reclaiming:** The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released

* [VMware Glossary](https://www.vmware.com/in/topics/glossary.html)
* [Content](https://www.vmware.com/in/topics/glossary/content.html)
* [Kubernetes Namespace](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html)
* [What is a Kubernetes Namespace?](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-462641936)
* [What is the “default” namespace in Kubernetes?](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-758266208)
* [Why use Kubernetes namespaces?](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-141395449)
* [When should one use multiple Kubernetes namespaces?](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-207085470)
* [What are the basic kubectl commands relating to namespace?](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-2101604685)
* [Recommended for You](https://www.vmware.com/in/topics/glossary/content/kubernetes-namespace.html#cat-2028033179)

What is a Kubernetes Namespace?

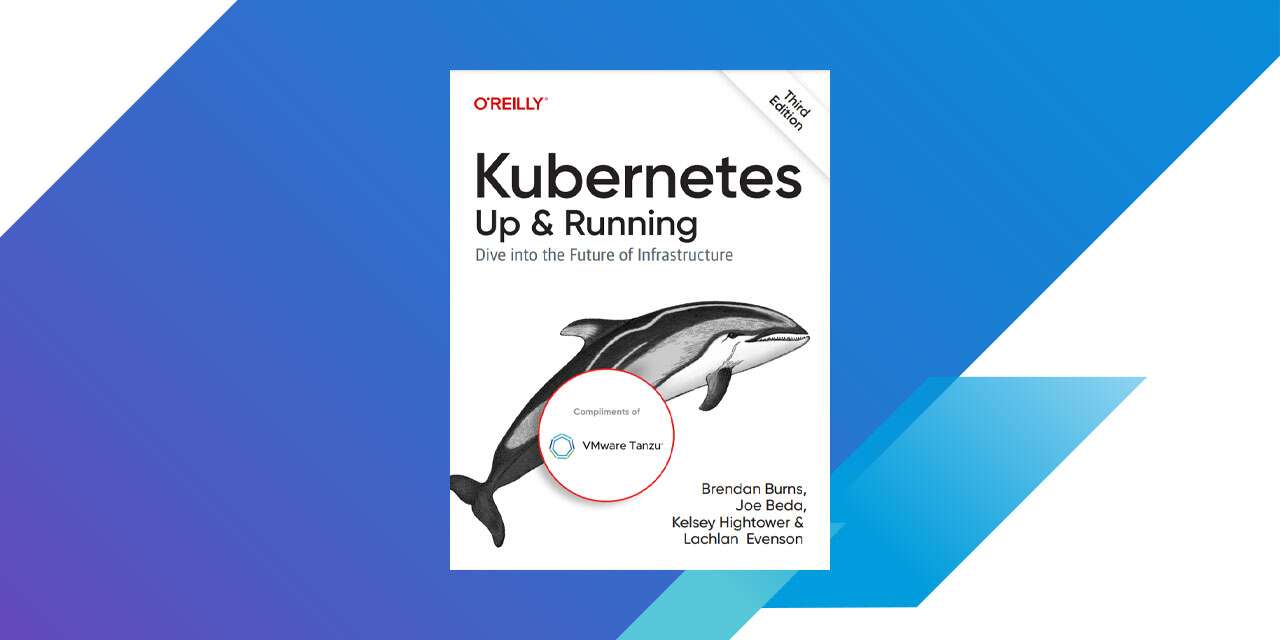
Namespaces are a way to organize clusters into virtual sub-clusters — they can be helpful when different teams or projects share a Kubernetes cluster. Any number of namespaces are supported within a cluster, each logically separated from others but with the ability to communicate with each other. Namespaces cannot be nested within each other.

Any resource that exists within [Kubernetes](https://tanzu.vmware.com/kubernetes-vs-docker) exists either in the default namespace or a namespace that is created by the cluster operator. Only nodes and persistent storage volumes exist outside of the namespace; these low-level resources are always visible to every namespace in the cluster.



State of Kubernetes

[DOWNLOAD NOW](https://hello-tanzu.vmware.com/state-of-kubernetes-2022/" \o "DOWNLOAD NOW" \t "_blank)



Kubernetes Up and Running

[DOWNLOAD NOW](https://hello-tanzu.vmware.com/kubernetes-up-and-running-3/)

What is the “default” namespace in Kubernetes?

Kubernetes comes with three namespaces out-of-the-box. They are:

1. **default:** As its name implies, this is the namespace that is referenced by default for every Kubernetes command, and where every Kubernetes resource is located by default. Until new namespaces are created, the entire cluster resides in ‘default’.
2. **kube-system:** Used for [Kubernetes components](https://www.vmware.com/in/topics/glossary/content/components-kubernetes.html) and should be avoided.
3. **kube-public:** Used for public resources. Not recommended for use by users.

Why use Kubernetes namespaces?

There are many use cases for Kubernetes namespaces, including:

* Allowing teams or projects to exist in their own virtual clusters without fear of impacting each other’s work.
* Enhancing role-based access controls (RBAC) by limiting users and processes to certain namespaces.
* Enabling the dividing of a cluster’s resources between multiple teams and users via resource quotas.
* Providing an easy method of separating development, testing, and deployment of containerized applications enabling the entire lifecycle to take place on the same cluster.

When should one use multiple Kubernetes namespaces?

Small teams or smaller organizations may be perfectly content using the default namespace. This is particularly relevant if there is no need to isolate developers or users from each other. However, there are many useful benefits to having multiple namespaces, including:

* **Isolation.** Large or growing teams can use namespaces to isolate their projects and microservices from each other. Teams can re-use the same resource names in different workspaces without a problem. Also, taking an action on items in one workspace never affects other workspaces.
* **Organization.** Organizations that use a single cluster for development, testing, and production can use namespaces to sandbox dev and test environments. This ensures production code is not affected by changes that developers or testers make in their own namespaces throughout the application lifecycle.
* **Permissions.** Namespaces enable the use of Kubernetes RBAC, so teams can define roles that group lists of permissions or abilities under a single name. This can ensure that only authorized users have access to resources in a given namespace.
* **Resource Control.** Policy-driven resource limits can be set on namespaces by defining resource quotas for CPU or memory utilization. This can ensure that every project or namespace has the resources it needs to run, and that no one namespace is hogging all available resources.
* **Performance.** Using namespaces can help improve performance of a given cluster. If a cluster is separated into multiple namespaces for different projects, the Kubernetes API will have fewer items to search when performing operations. This can reduce latency and speed overall application performance for each application running on the cluster.

How can pods communicate across Kubernetes namespaces?

Although namespaces are separate from each other, they can easily communicate with each other. Kubernetes DNS service directory can easily locate any service by its name by using the expanded form of DNS addressing:

..svc.cluster.local

Simply adding the namespace name to the service name provides access to services in any namespace on the cluster. For example, to access the payroll service in the development namespace you would use the address  
payroll.development  
To access the payroll service in the production namespace you would use:  
payroll.production

Note that network policies can optionally be utilized to control access between namespaces. For example, a network policy can allow or deny all traffic from other namespaces. Network polices apply only to connections and are not a substitute for firewalls that perform packet inspection.

What are the basic kubectl commands relating to namespace?

What is the command to find current Kubernetes namespaces?

All namespaces in the cluster can be displayed with the command:  
kubectl get namespace  
This will return a list of all namespaces in the cluster, including the default namespaces, along with their status and age.

What is the command to create a new Kubernetes namespace?

Namespaces are created simply with the command:  
kubectl create namespace  
As with any other Kubernetes resource, a YAML file can also be created and applied to create a namespace:

**newspace.yaml:**kind: Namespace  
apiVersion: v1  
metadata:  
name: newspace  
labels:  
name: newspacekubectl apply -f newspace.yaml

How do you switch between Kubernetes namespaces?

To address namespaces once they are created, actions must include the –namepsace= option in the command. Since this can get cumbersome, the default namespace can be changed by using the kubectl config command to set the namespace in the cluster context.  
For example, to change from the default namespace to one named ‘testing’ you would enter:  
kubectl config set-context --current --namespace=testing  
This will set the default namespace to ‘testing’ for all future kubectl commands.

How do you rename a Kubernetes namespace?

It is not standard practice to rename a Kubernetes namespace, so choose namespaces (outside of the default) carefully.

How do you delete a Kubernetes namespace?

Namespaces are deleted with the command:  
kubectl delete namespaces  
Since the deletion is an asynchronous activity, the namespace will show as ‘terminating’ until the namespace is deleted.

A warning about deleting Kubernetes namespaces

Deleting a namespace is a final act. Everything in the namespace including all services, running pods, and artifacts will be deleted. Garbage collection will run on anything that had existed in that namespace. Be certain everything in the namespace should be deleted before taking this action.

Initial namespaces

Kubernetes starts with four initial namespaces:

**default**

Kubernetes includes this namespace so that you can start using your new cluster without first creating a namespace.

**kube-node-lease**

This namespace holds [Lease](https://kubernetes.io/docs/concepts/architecture/leases/) objects associated with each node. Node leases allow the kubelet to send [heartbeats](https://kubernetes.io/docs/concepts/architecture/nodes/#heartbeats) so that the control plane can detect node failure.

**kube-public**

This namespace is readable by *all* clients (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

**kube-system**

The namespace for objects created by the Kubernetes system