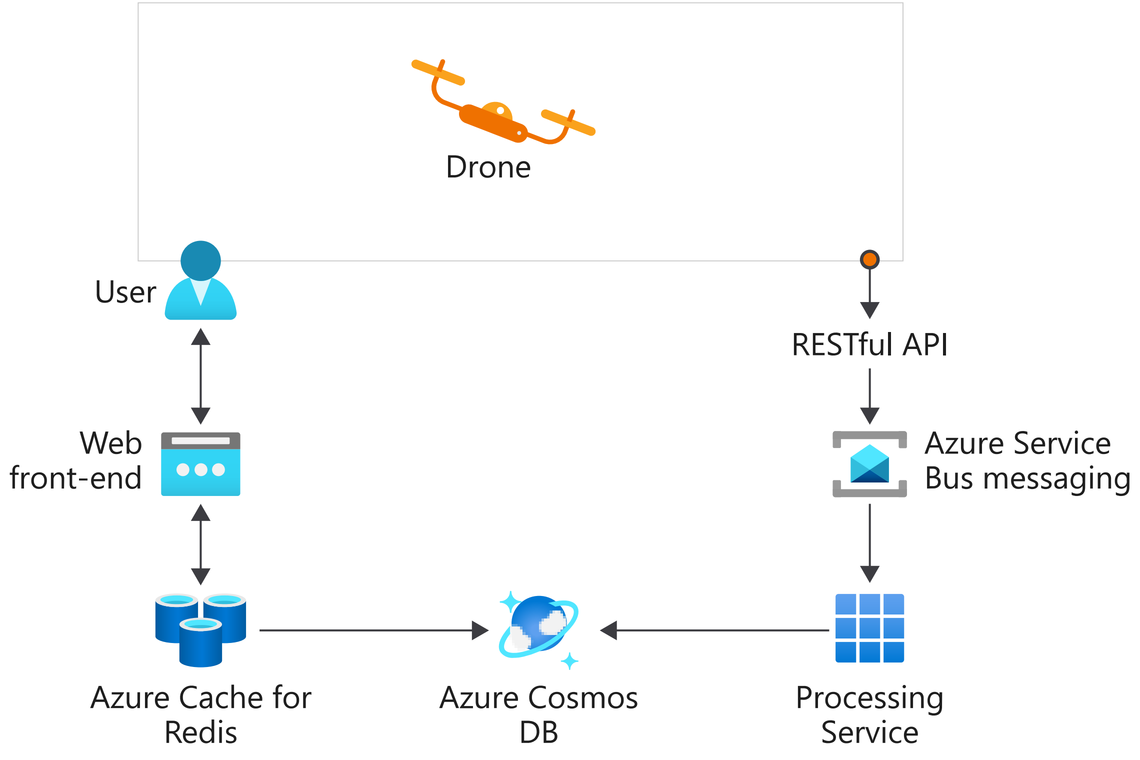
✓ 100 XP

How Kubernetes deployments work

10 minutes

The drone-tracking app has several components that are deployed separately from each other. It's your job to configure deployments for these components on the cluster. Here, you'll look at some of the deployment options available to you to deploy these components.



Pod deployment options

There are several options to manage the deployment of pods in a Kubernetes cluster when you're using `kubectl`. The options are:

- Pod templates
- Replication controllers
- Replica sets
- Deployments

You can use any of these four Kubernetes object-type definitions to deploy a pod or pods. These files make use of YAML to describe the intended state of the pod or pods to be deployed.

What is a pod template?

A pod template enables you to define the configuration of the pod you want to deploy. The template contains information such as the name of container image and which container registry to use to fetch the images. The template may also include runtime configuration information, such as ports to use. Templates are defined by using YAML in the same way as when you create Docker files.

You can use templates to deploy pods manually. However, a manually deployed pod isn't relaunched after it fails, is deleted, or is terminated. To manage the lifecycle of a pod, you need to create a higher-level Kubernetes object.

What is a replication controller?

A replication controller uses pod templates and defines a specified number of pods that must run. The controller helps you run multiple instances of the same pod, and ensures pods are always running on one or more nodes in the cluster. The controller replaces running pods in this way with new pods if they fail, are deleted, or are terminated.

For example, assume you deploy the drone tracking front-end website, and users start accessing the website. If all the pods fail for any reason, the website is unavailable to your users unless you launch new pods. A replication controller helps you make sure your website is always available.

What is a replica set?

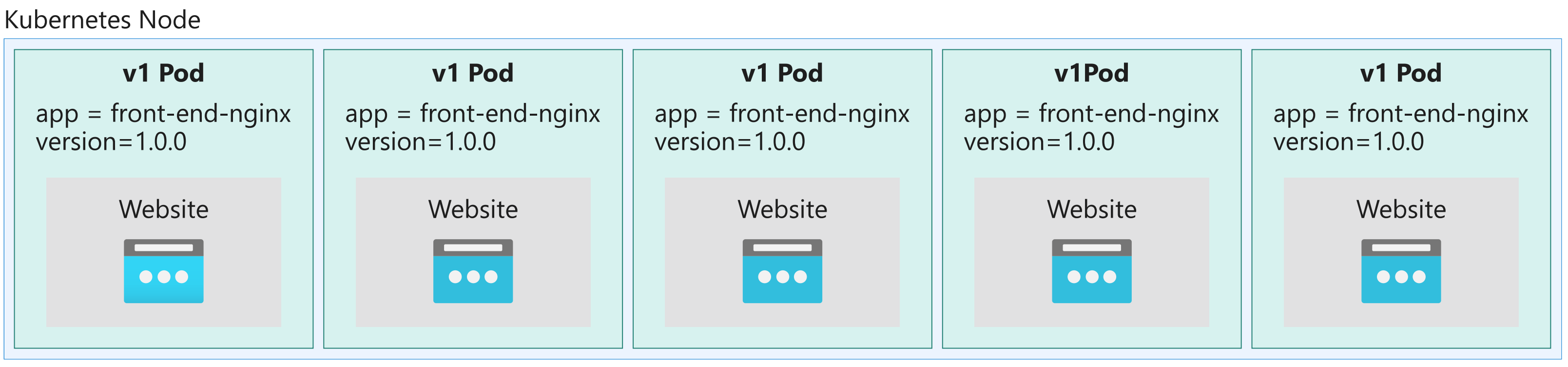
A replica set replaces the replication controller as the preferred way to deploy replicas. A replica set includes the same functionality as a replication controller, but it has an extra configuration option to include a selector value.

A selector enables the replica set to identify all the pods running underneath it. Using this feature, you can manage pods labeled with the same value as the selector value, but not created with the replicated set.

What is a deployment?

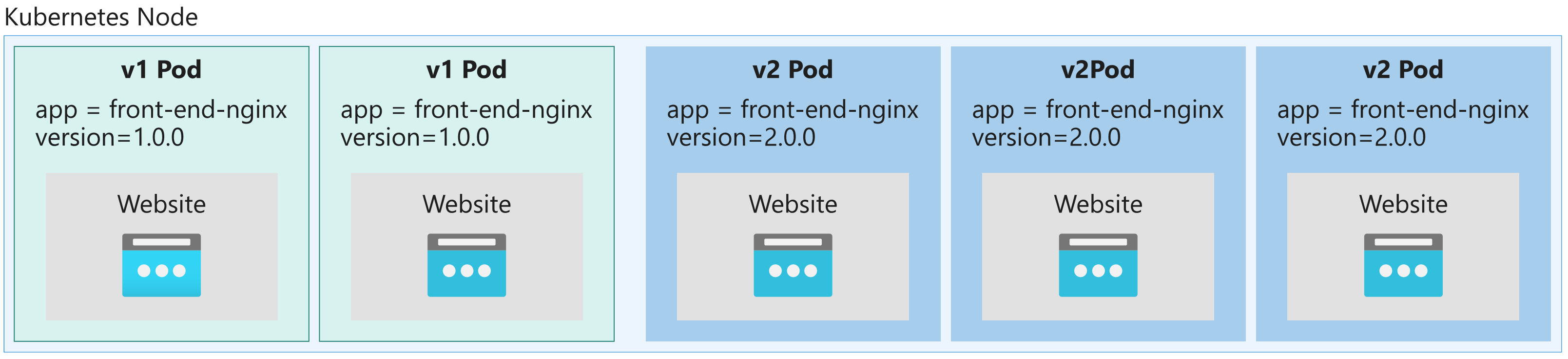
A deployment creates a management object one level higher than a replica set, and allows you to deploy and manage updates for pods in a cluster.

Assume that you have five instances of your app deployed in your cluster. There are five pods running version 1.0.0 of your app.



If you decide to update your app manually, you can remove all pods, then launch new pods running version 2.0.0 of your app. With this strategy, your app will experience downtime.

Instead, you'll want to execute a rolling update where you launch pods with the new version of your app before you remove the older app versioned pods. Rolling updates will launch one pod at a time instead of taking down all the older pods at once. Deployments honor the number of replicas configured in the section that describes information about replica sets. It will maintain the number of pods specified in the replica set as it replaces old pods with new pods.



Deployments, by default, provide a rolling update strategy for updating pods. You can also use a re-create strategy. This strategy will terminate pods before launching new pods.

Deployments also provide you with a rollback strategy, which you can execute by using `kubectl`.

Deployments make use of YAML-based definition files and make it easy to manage deployments. Keep in mind that deployments allow you to apply any changes to your cluster. For example, you can deploy new versions of an app, update labels, and run other replicas of your pods.

`kubectl` has convenient syntax to create a deployment automatically when you're using the `kubectl run` command to deploy a pod. This command creates a deployment with the required replica set and pods. However, the command doesn't create a definition file. It's a best practice to manage all deployments with deployment definition files, and track changes by using a version-control system.

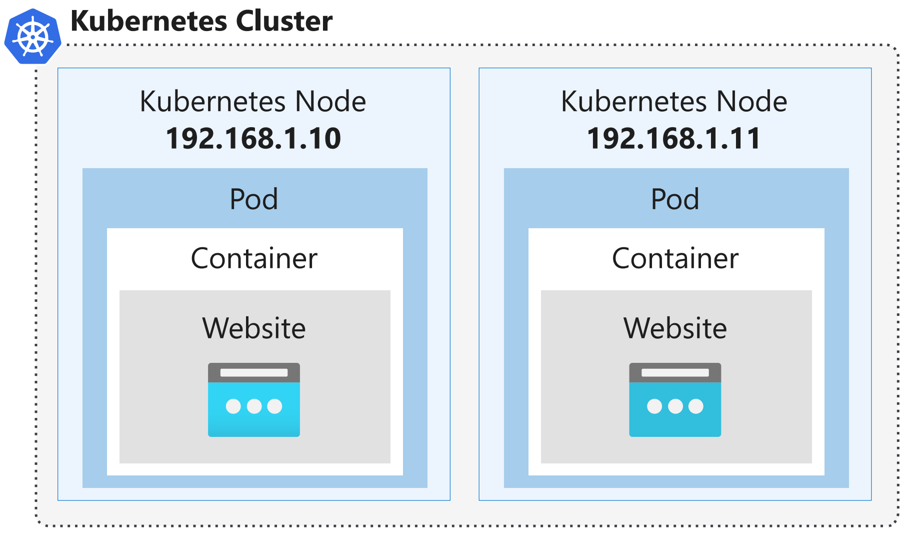
Deployment considerations

Kubernetes has specific requirements about how you configure networking and storage for a cluster. How you configure these two aspects affects your decisions about how to expose your apps on the cluster network and store data.

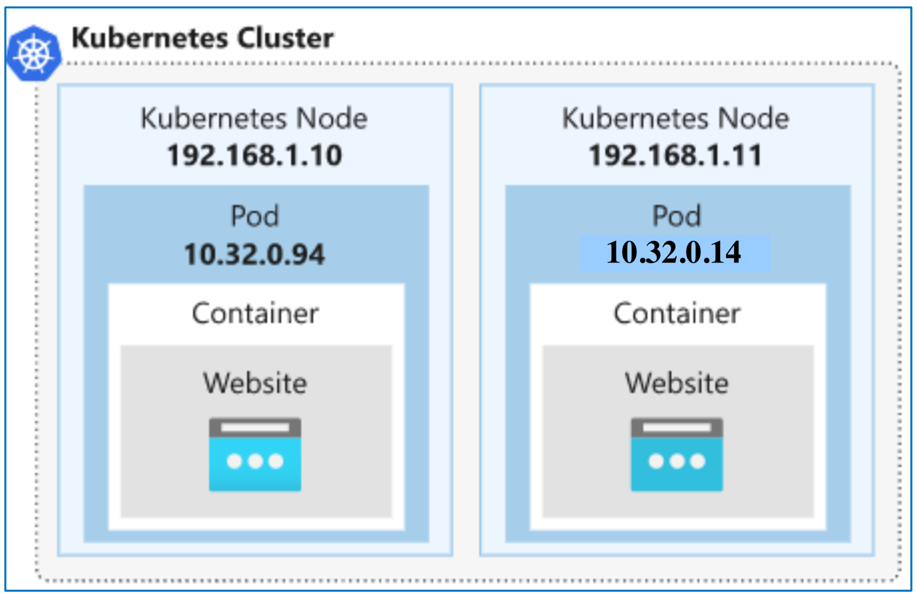
For example, each of the services in the drone-tracking app has specific requirements for user access, inter-process network access, and data storage. Now, take a look at these aspects of a Kubernetes cluster and how they affect the deployment of apps.

Kubernetes networking

Assume you have a cluster with one control plane and two nodes. When you add nodes to Kubernetes, an IP address is automatically assigned to each node from an internal private network range. For example, assume that your local network range is 192.168.1.0/24.



Each pod that you deploy gets assigned an IP from a pool of IP addresses. For example, assume that your configuration uses the 10.32.0.0/12 network range, as the following image shows.



By default, the pods and nodes can't communicate with each other by using different IP address ranges.

To further complicate matters, recall that pods are transient. The pod's IP address is temporary, and can't be used to reconnect to a newly created pod. This configuration affects how your app communicates with its internal components and how you and services interact with it externally.

To simplify communication, Kubernetes expects you to configure networking in such a way that:

- Pods can communicate with one another across nodes without Network Address Translation (NAT).
- Nodes can communicate with all pods, and vice versa, without NAT.
- Agents on a node can communicate with all nodes and pods.

Kubernetes offers several networking options that you can install to configure networking. Examples include Antrea, Cisco Application Centric Infrastructure (ACI), Cilium, Flannel, Kubenet, VMware NSX-T, and Weave Net.

Cloud providers also provide their own networking solutions. For example, Azure Kubernetes Service (AKS) supports the Azure Virtual Network container network interface (CNI), Kubenet, Flannel, Cilium, and Antrea.

Kubernetes services

A Kubernetes service is a Kubernetes object that provides stable networking for pods. A Kubernetes service enables communication between nodes, pods, and users of your app, both internal and external, to the cluster.

Kubernetes assigns a service an IP address on creation, just like a node or pod. These addresses get assigned from a service cluster's IP rang; for example, 10.96.0.0/12. A service is also assigned a DNS name based on the service name, and an IP port.

In the drone-tracking app, network communication is as follows:

- The website and RESTful API are accessible to users outside the cluster.
- The in-memory cache and message queue services are accessible to the front end and the RESTful API, respectively, but not to external users.
- The message queue needs access to the data processing service, but not to external users.
- The NoSQL database is accessible to the in-memory cache and data processing service, but not to external users.

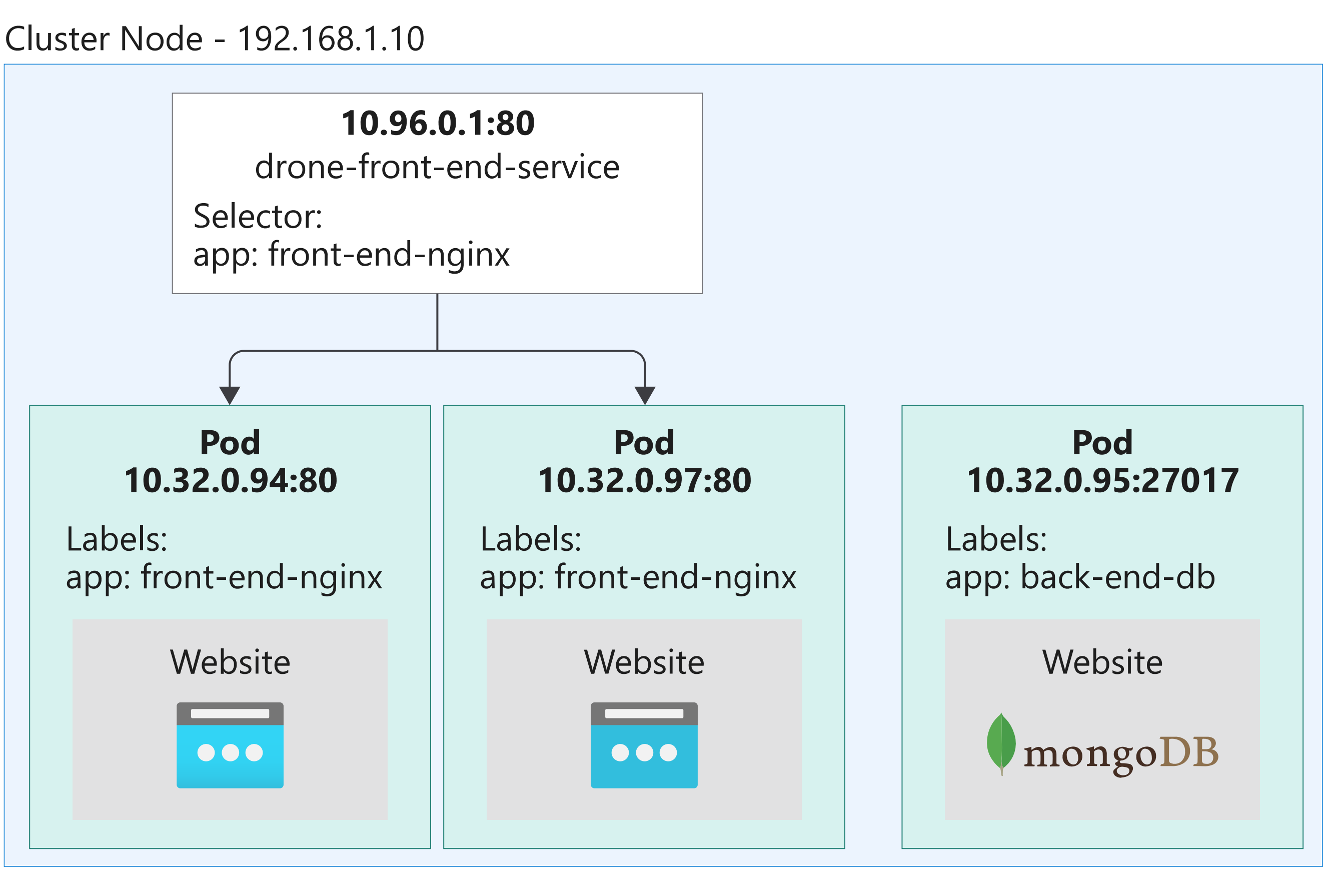
To support these scenarios, you can configure three types of services to expose your app's components.

Service	Description
ClusterIP	The address assigned to a service that makes the service available to a set of services inside the cluster. For example, communication between the front-end and back-end components of your app.
NodePort	The node port between 30000 and 32767 that the Kubernetes control plane assigns to the service; for example, 192.169.1.11 on clusters01. You then configure the service with a target port on the pod that you want to expose. For example, configure port 80 on the pod running one of the front ends. You can now access the front end through a node IP and port address.
LoadBalancer	The load balancer that allows for the distribution of load between nodes running your app, and exposing the pod to public network access. You typically configure load balancers when you use cloud providers. In this case, traffic from the external load balancer is directed to the pods running your app.

In the drone-tracking app, you might decide to expose the tracking website and the RESTful API by using a LoadBalancer and the data processing service by using a ClusterIP.

How to group pods

Managing pods by IP address isn't practical. Pod IP addresses change as controllers re-create them, and you might have any number of pods running.

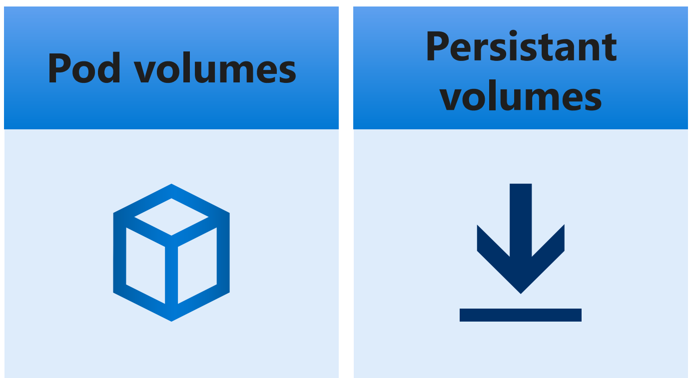


A service object allows you to target and manage specific pods in your cluster by using selector labels. You set the selector label in a service definition to match the pod label defined in the pod's definition file.

For example, assume that you have many running pods. Only a few of these pods are on the front end, and you want to set a LoadBalancer service that targets only the front-end pods. You can apply your service to expose these pods by referencing the pod label as a selector value in the service's definition file. The service will now group only the pods that match the label. If a pod is removed and re-created, the new pod is automatically added to the service group through its matching label.

Kubernetes storage

Kubernetes uses the same storage volume concept that you find when using Docker. Docker volumes are less managed than the Kubernetes volumes, because Docker volume lifetimes aren't managed. The Kubernetes volume's lifetime is an explicit lifetime that matches the pod's lifetime. This lifetime match means a volume outlives the containers that run in the pod. However, if the pod is removed, so is the volume.



Kubernetes provides options to provision persistent storage with the use of *PersistentVolumes*. You can also request specific storage for pods by using *PersistentVolumeClaims*.

Keep both of these options in mind when you're deploying app components that require persisted storage, like message queues and databases.

Cloud integration considerations

Kubernetes doesn't dictate the technology stack you use in your cloud-native app. In a cloud environment such as Azure, you can use several services outside the Kubernetes cluster.

Recall from earlier that Kubernetes doesn't provide any of the following services:

- Middleware
- Data-processing frameworks
- Databases
- Caches
- Cluster storage systems

In this drone-tracking solution, there are three services that provide middleware functionality: a NoSQL database, an in-memory cache service, and a message queue. You might select MongoDB Atlas for the NoSQL solution, Redis to manage in-memory cache and, RabbitMQ or Kafka depending on your message-queue needs.

When you're using a cloud environment such as Azure, it's a best practice to use services outside the Kubernetes cluster. This decision can simplify the cluster's configuration and management. For example, you can use *Azure Cache for Redis* for the in-memory caching services, *Azure Service Bus messaging* for the message queue, and *Azure Cosmos DB* for the NoSQL database.

Next unit: Exercise - Explore the functionality of a Kubernetes cluster

Continue >