



Understanding Kubernetes



UPDATED FOR 2022

A Guide to Modernizing Your Cloud Infrastructure

Table of Contents

State of Kubernetes	03
Why Kubernetes?	05
Interacting with Kubernetes	07
Worker Nodes and the Control Plane	09
Objects	12
Controllers	17
Networking	21
Our Take	22
Next Steps	23
About	24

Understand fundamental concepts of Kubernetes, from the components of a Kubernetes cluster to network model implementation. Along with a working knowledge of containers, after reading this guide, you will be able to jump right in and deploy your first Kubernetes cluster.

Kubernetes is a highly-active project with a growing feature set and expanding user base. In this 2022 update, you can see how Kubernetes became the standard for container orchestration and learn about new key features that support developers building with Kubernetes since we first released this ebook in 2020.

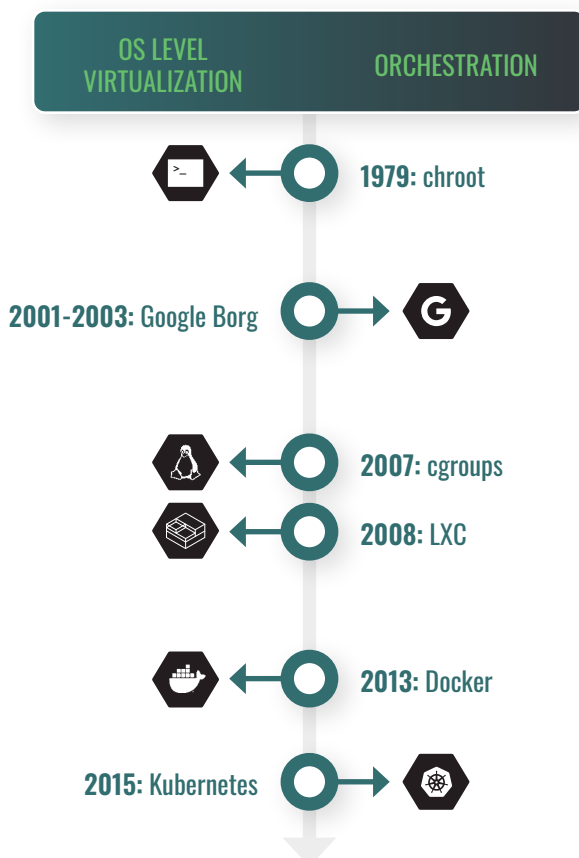
Linode Kubernetes Engine has enabled native Kubernetes features including horizontal cluster autoscaling, high-availability control planes, dashboards, and others.

State of Kubernetes

Since the release of version 1.0 in 2015, Kubernetes has quickly become mainstream in the world of DevOps. [A 2021 report from the Cloud Native Computing Foundation \(CNCF\)](#) and SlashData showed that 31% of developers worldwide are using Kubernetes, a 67% increase from 2020. This wasn't an isolated jump but a continuation of the preceding years. [An earlier report](#) from the CNCF showed the use of Kubernetes in production had increased 300% from their first survey in 2016. Users range from individuals and small businesses to large enterprises, featuring some names you're certainly familiar with like [Spotify](#), [Squarespace](#), and [Pinterest](#). What's driving the increase? To understand the adoption of Kubernetes, you have to understand the benefits of containers.

A Brief History of Containers

Containers are similar to virtual machines. They are lightweight isolated runtimes that share the physical resources of the host operating system kernel without having to run a full operating system themselves. Containers are light on resources compared to a full OS but contain a complete set of information needed to execute their containerized application images, such as files, environment variables, and libraries. Running your application in a container gives you the power of portability, reduces environmental variables, and can remove the need to separately manage dependencies. Using containers can greatly cut down on development time and overhead.



The concept of containers and OS virtualization—a prerequisite for containers—has existed since the 1970s. In 1979, computing hit a major milestone with the release of UNIX V7. V7 introduced process isolation with **chroot**, which **changes** the **root** directory for a group of processes to a separate location in the file system. This concept serves as a baseline for what would fundamentally lead to resource isolation and application containerization.

Jumping ahead to 2007, development arrived at the release of *Process Containers*, designed to isolate system resources for a collection of processes. Eventually renamed **cgroups** and merged into the Linux kernel, this formed the foundation of *Linux Containers* or **LXC**, the first complete container manager for the Linux operating system. Development of **Docker** began in 2010 and was originally built around LXC until its own library was built in 2013 (*libcontainer*).

It's no coincidence that these advances came around the same time that commercial multithreading processors became mainstream in the early/mid 2000s. Simultaneous process isolation is not feasible with single core processors due to the lack of parallelisation resulting in major bottlenecks. With this obstacle removed, development could take a great step forward.

The conceptual benefits of containers had been known for decades, but they finally started to become technically practical from a hardware and software standpoint. Now, we need a way to manage them at scale. Welcome to **Kubernetes**, a container orchestrator.

Tracing its roots back to **Borg**, an internal tool developed by Google to manage container clusters, Kubernetes was announced as an open source project in 2014. Originally developed by Google as well, Kubernetes hit 1.0 in 2015 and was given to the newly formed offshoot of the Linux Foundation, the *Cloud Native Computing Foundation (CNCF)*, which maintains the project to this day.

Adoption came quickly after 1.0 reached GA. Only one year later in 2016, *Pokémon Go!* was released for Android and iOS, running on a massive Kubernetes deployment. This freemium game immediately exploded in popularity and would need to scale for millions of active users around the globe. In 2017, [GitHub announced](#) that all web and API requests are served by containers running on Kubernetes. Citing the need for more flexibility as a driving factor for the change, Kubernetes was chosen over other platforms because of its open source community, first run experience, and wealth of available information.

As more companies began to follow suit, the project has continued to expand exponentially. As of May 2022, the Kubernetes project would reach version 1.24 with over 3,000 individual contributors. Enterprise users would also become its largest contributors, with Red Hat, Microsoft, and VMware, adding significantly to the project alongside independent users. The [CNCF](#) in May of 2022 now consists of almost 800 members with over 20 trillion dollars in market cap.



Why Kubernetes?

Kubernetes quickly became the new industry-standard tool for running cloud-native applications. As more developers and organizations move away from on-premises infrastructure to take advantage of the cloud, advanced management is needed to ensure high availability and scalability for containerized applications. To build and maintain applications, you need to coordinate resources across different machine types, networks, and environments.

Kubernetes allows developers of containerized applications to develop more reliable infrastructure, a critical need for applications and platforms that need to respond to events like rapid changes in traffic or the need to restart failed services. With Kubernetes, you can now delegate events that would otherwise require the manual intervention of an on-call developer.

Kubernetes also optimizes container orchestration deployment and management for cloud infrastructure by creating containers and groups of containers, called Pods, that can scale without writing additional lines of code, allowing them to respond to the needs of the application as they arise. Key benefits of moving to container-centric infrastructure with Kubernetes are knowing that infrastructure will self-heal and that there will be environmental consistency from development through to production.



What is Kubernetes?

Kubernetes is a container orchestration system used to scale containerized applications in the cloud. Kubernetes can manage the lifecycle of containers, creating and destroying them depending on the needs of the application, as well as providing a host of other features. Kubernetes has become one of the most discussed concepts in cloud-based application development, and the rise of Kubernetes signals a shift in the way that applications are developed and deployed.

In general, Kubernetes is formed by a *cluster* of servers, called Nodes, each running Kubernetes agent processes and communicating with one another. Any node or nodes in your cluster can run a collection of processes that make up the *Control Plane*, which enacts and maintains the desired state of the Kubernetes cluster. *Worker Nodes* are responsible for running the containers that form your applications and services.

Containers in Kubernetes

Kubernetes is a container orchestration tool and, therefore, needs a container runtime installed to work. There is no default runtime for Kubernetes, but commonly used runtimes include Docker Engine, containerd, CRI-O, and Mirantis. Examples throughout this guide will use Docker Engine as the container runtime.

For additional information, refer to Kubernetes documentation on [Container Runtimes](#).

- **Containerization** is a virtualization method to run distributed applications in containers using microservices. Containerizing an application requires a base image to create an instance of a container. Once an application's image exists, you can push it to a centralized container registry that Kubernetes can use to deploy container instances in a cluster's *Pods*, which you can learn more about in [Beginner's Guide to Kubernetes: Objects](#).
- **Orchestration** is the automated configuration, coordination, and management of computer systems, software, middleware, and services. It takes advantage of automated tasks to execute processes. For Kubernetes, container orchestration automates all the provisioning, deployment, and availability of containers as well as load balancing, resource allocation between containers, and health monitoring of the cluster.

Interacting with Kubernetes

Kubernetes API

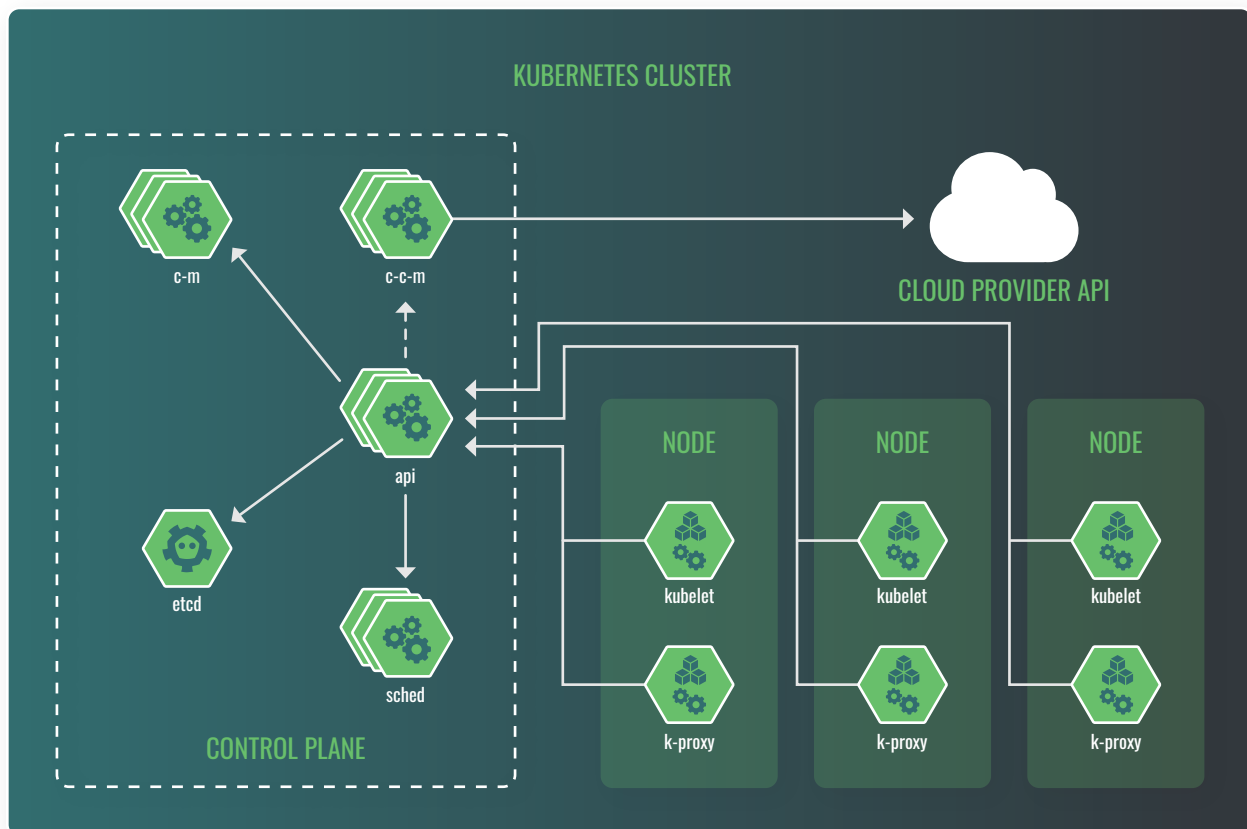
Kubernetes is built around a robust RESTful API. Every action taken in Kubernetes—be it inter-component communication or user command—interacts in some fashion with the Kubernetes API. The goal of the API is to help facilitate the desired state of the Kubernetes cluster. To create this desired state, you create *objects*, which are normally represented by YAML files called *manifests*, and apply them through the command line with the **kubectl** tool.

kubectl

kubectl is a command line tool used to interact with the Kubernetes cluster. It offers a host of features, including the ability to create, stop, and delete resources; describe active resources; and auto scale resources.

For more information on the types of commands and resources, you can use with kubectl, consult the [Kubernetes kubectl documentation](#).

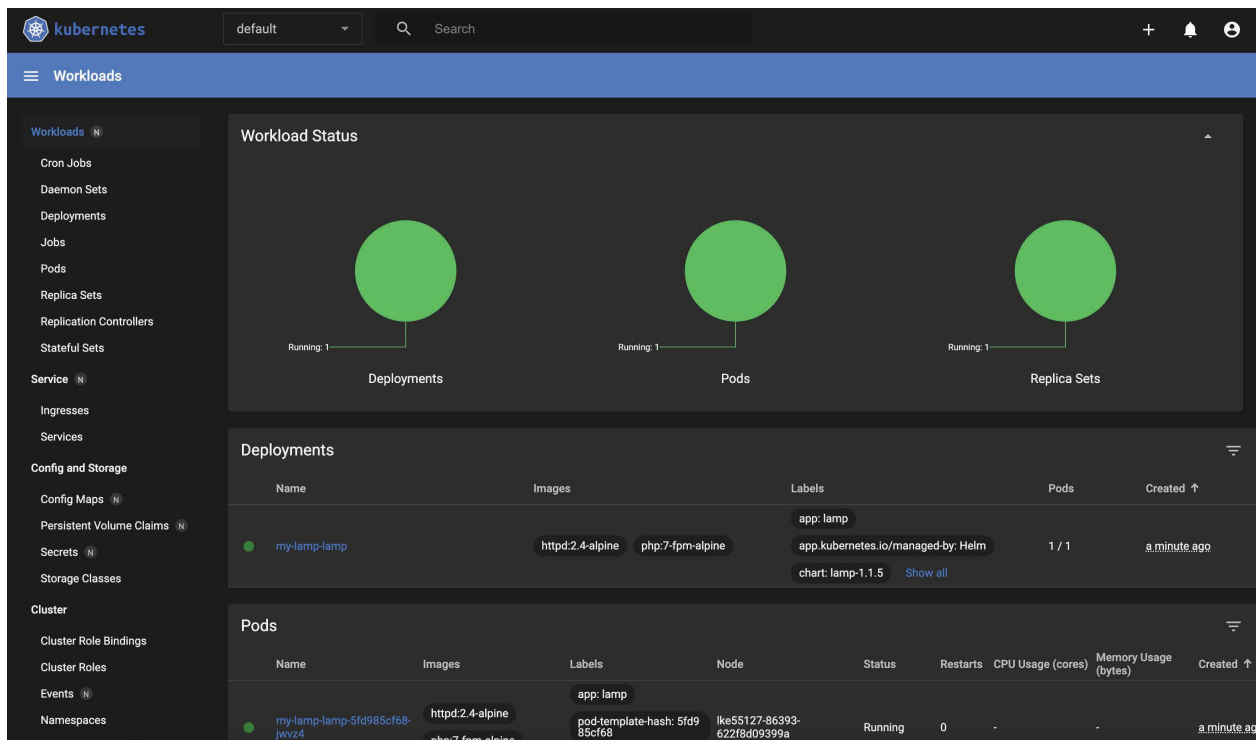
At the highest level of Kubernetes, there are *Nodes*. These can be Linodes, VMs, or physical servers. Together, these servers form a *cluster* controlled by the services that make up the *Control Plane*.



Dashboard

In addition to `kubectl`, there are many GUI options to consider for interacting with the orchestrator. There is a dashboard implementation from the Kubernetes project as well as many third-party options that add additional functionality and streamlined actions.

Focusing on the standard [Kubernetes Dashboard](#) from the Kubernetes dev team, you have an industry-standard, open source, web-based UI designed to be a visual hub for interacting with a Kubernetes cluster. It allows for a more intuitive interface for observing cluster objects, components, containers, applications, and more. While providing an easy way to monitor a Kubernetes cluster and perform health checks, the Kubernetes dashboard additionally provides a means to perform administration tasks, similarly to how one would perform administration through the command line tool `kubectl`.



Many managed Kubernetes services [such as Linode Kubernetes Engine \(LKE\)](#) have incorporated or expanded upon the official dashboard, integrating it with their existing interface.

Worker Nodes and the Control Plane

For your Kubernetes cluster to maintain homeostasis for your application, it requires a central source of communications and commands. Your Kubernetes Worker Nodes and Control Plane are the essential components that run and maintain your cluster.

Worker Nodes

Kubernetes Nodes are worker servers that run your application(s). The user creates and determines the number of Nodes. In addition to running your application, each Node runs two processes:

1. **kubelet** receives descriptions of the desired state of a Pod from the API server, and ensures the Pod is healthy, and running on the Node.
2. **kube-proxy** is a networking proxy that proxies the UDP, TCP, and SCTP networking of each Node, and provides load balancing. kube-proxy is only used to connect to Services.



The Control Plane

Together, kube-apiserver, kube-controller-manager, kube-scheduler, and etcd form what is known as the *control plane*. The control plane components can run on one or multiple nodes in your cluster and are responsible for making decisions about the cluster and pushing it towards the desired state. kube-apiserver, kube-controller-manager, and kube-scheduler are processes, and etcd is a database.

- **kube-apiserver** is the front end for the Kubernetes API server.
- **kube-controller-manager** is a daemon that manages the Kubernetes control loop. For more on Controllers, see the [Beginner's Guide to Kubernetes: Controllers](#).
- **kube-scheduler** is a function that looks for newly created Pods that have no Nodes, and assigns them a Node based on a host of requirements. For more information on kube-scheduler, consult the [Kubernetes kube-scheduler documentation](#).
- **etcd** is a highly available key-value store that provides the backend database for Kubernetes. It stores and replicates the entirety of the Kubernetes cluster state. It's written in Go and uses the [Raft protocol](#), which means it maintains identical logs of state-changing commands across nodes and coordinates the order in which these state changes occur.

There are a number of objects that are abstractions of your Kubernetes system's desired state. These objects represent your application, its networking, and disk resources—all of which together form your application.

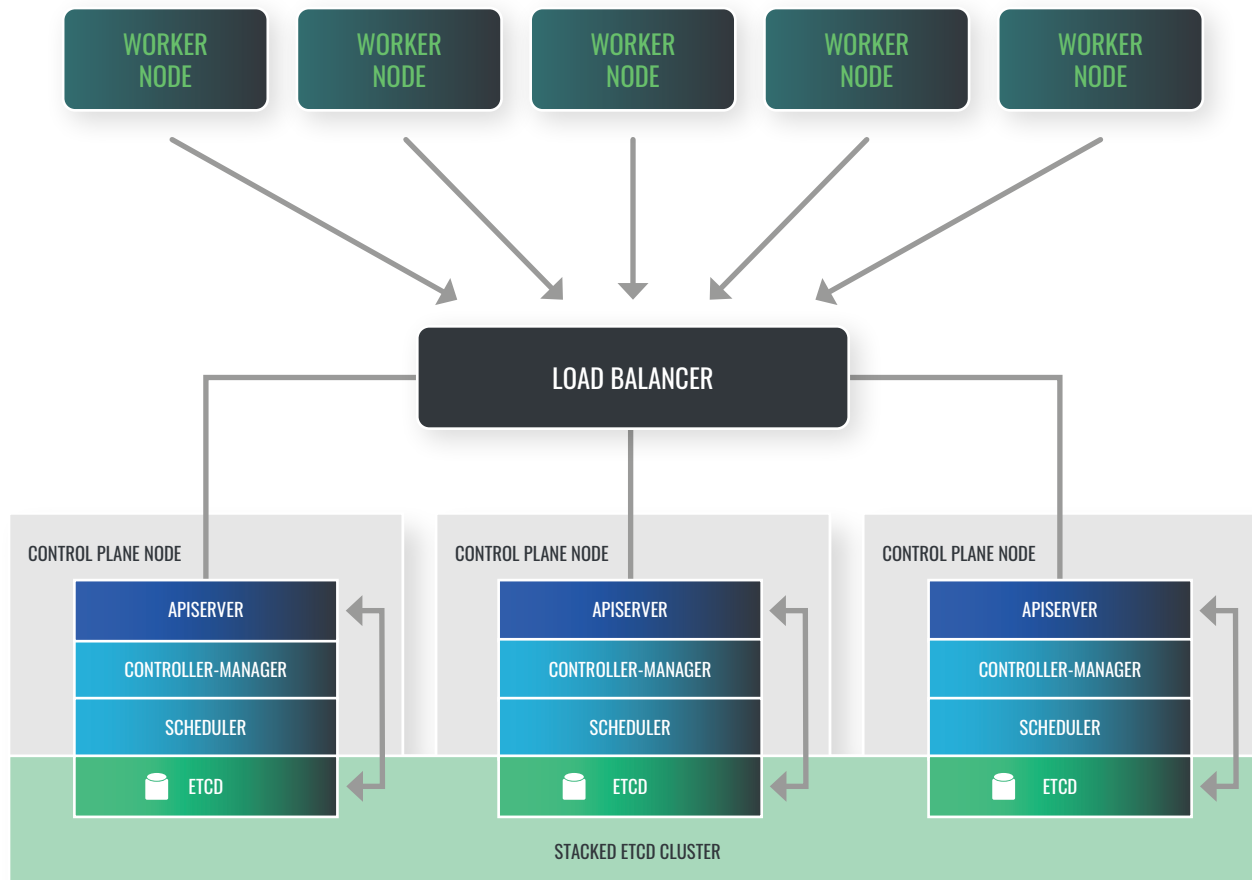
Exclusively found when interaction with cloud environments is needed, an additional control plane component, the **cloud-controller-manager** will be used. This component is responsible for interacting with a cloud provider's API to assist with unique tasks like managing DNS or hardware autoscaling. The cloud-controller-manager will, in some ways, be unique to the cloud environment Kubernetes is using. When hosting on-premises or in other unique configurations, this component will generally not be found or needed.

High-Availability Control Plane

A High-Availability (HA) Control Plane provides increased reliability for production applications and workloads by creating additional replicas of your control plane. While a highly-available control plane is not necessary for all Kubernetes deployments, it is generally recommended for production workloads.

HA is achieved by redundancy in control plane components, with a good configuration reflecting the following:

- etcd and kube-api-server increases from one to three replicas.
- All other components, the cloud-controller-manager, kube-scheduler, and kube-controller-manager, increase from one to two replicas, with leader election put in place.



Each control plane has access to individual worker nodes to ensure no single point of failure in your cluster. Placing multiple replicas on separate infrastructure is recommended to remove another single point of failure at the hardware level.

Objects

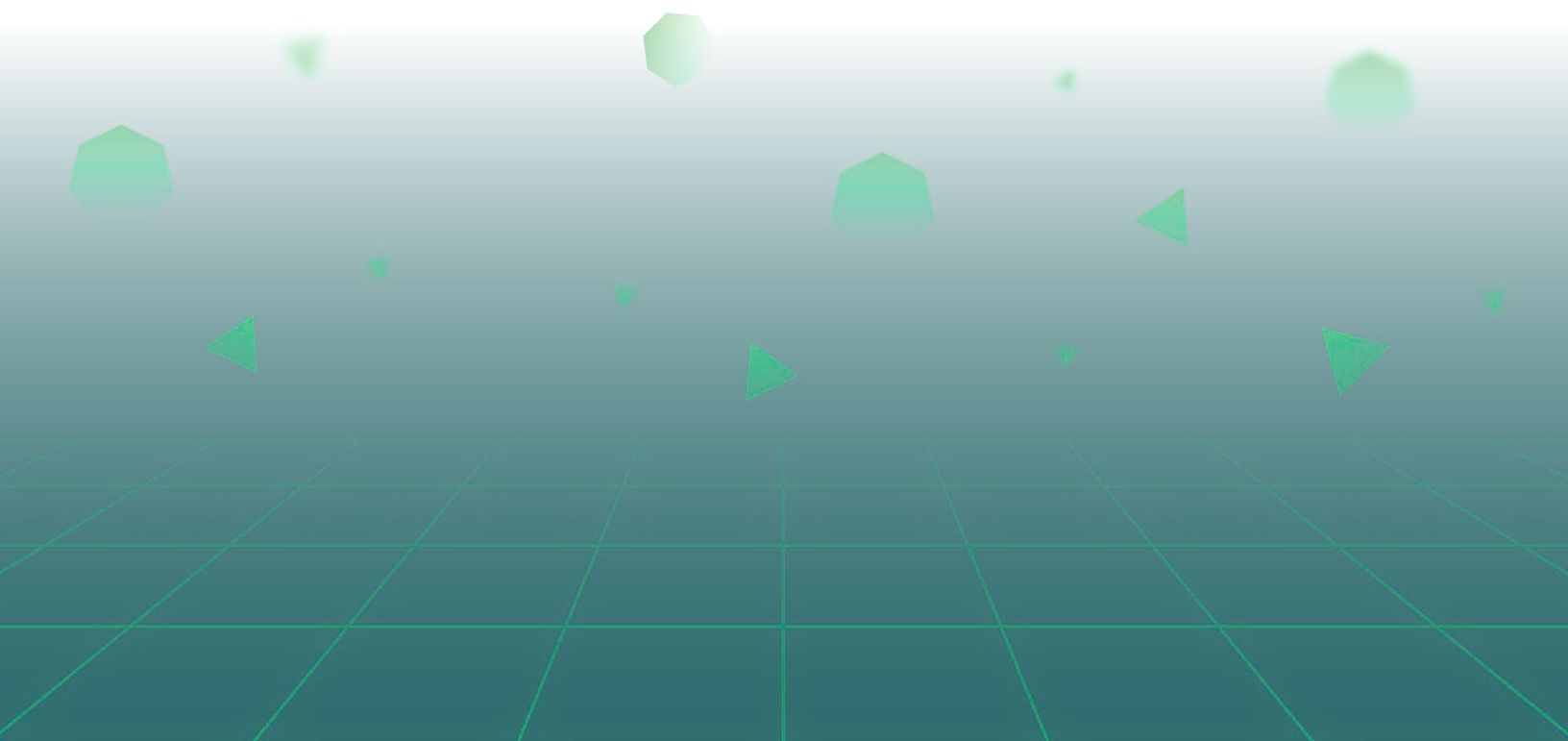
In the Kubernetes API, four basic Kubernetes objects: Pods; Services; Volumes; and Namespaces represent the abstractions that communicate what your cluster is doing. These objects describe what containerized applications are running, the nodes they are running on, available resources, and more.

Pods

All containers exist within *Pods*. Pods are the smallest unit of the Kubernetes architecture. You can view them as a kind of wrapper for your containers. Each Pod gets its own IP address that interacts with other Pods within the cluster.

Usually, a Pod contains only one container, but a Pod can contain multiple containers if those containers need to share resources. If there is more than one container in a Pod, these containers can communicate with one another via localhost.

Pods in Kubernetes are “mortal,” which means they are created and destroyed depending on the needs of the application. For instance, you might have a web app backend that sees a spike in CPU usage. This situation might cause the cluster to scale up the number of backend Pods from two to ten, in which case eight new Pods would be created. Once the traffic subsides, the Pods might scale back to two, in which case eight Pods would be destroyed.



It is important to note that Pods get destroyed without respect to which Pod was created first. And, while each Pod has its own IP address, this IP address will only be available for the lifecycle of the Pod. While Kubernetes has mechanisms in place to keep track of IP address changes, a user making manual configuration changes does not.

Here is an example of a Pod manifest:

```

my-apache-pod.yaml
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: apache-pod
5    labels:
6      app: web
7  spec:
8    containers:
9    - name: apache-container
10     image: httpd

```

Each manifest has four necessary parts:

1. The version of the API in use
2. The kind of resource you'd like to define
3. Metadata about the resource
4. Though not required by all objects, a spec, which describes the desired behavior of the resource, is necessary for most objects and controllers.

In the case of this example, the API in use is v1 and the kind is a Pod. The metadata field is used for applying a name, labels, annotations, and other identifiers that help to uniquely identify an object. Names differentiate resources, while labels, which will come into play more when defining [Services](#) and [Deployments](#), group like resources. Annotations are for attaching arbitrary data to the resource.

The spec is where the desired state of the resource is defined. In this case, a Pod with a single Apache container is desired, so the container's field is supplied with a name, 'apache-container', and an image, the latest version of Apache. The image is pulled from [Docker Hub](#), as that is the container registry being used for this Kubernetes configuration.

For more information on the type of fields you can supply in a Pod manifest, refer to the Pod Section of the [Kubernetes API Reference Docs for v1.19](#).

Services

Services group identical Pods together to provide a consistent means of accessing them. For instance, you might have three Pods that are all serving a website, and all of those Pods need to be accessible on port 80. A Service can ensure that all of the Pods are accessible at that port and can distribute traffic between those Pods through load balancers.

Additionally, a Service can allow your application to be accessible from the internet. Each Service gets an IP address and a corresponding local DNS entry. Additionally, Services exist across Nodes. If you have two replica Pods on one Node and an additional replica Pod on another Node, the Service can include all three Pods. Here are the different types of Services:

- **ClusterIP:** Exposes the Service internally to the cluster. This is the default setting for a Service.
- **NodePort:** Exposes the Service to the internet from the IP address of the Node at the specified port number. You can only use ports in the 30000-32767 range.
- **Load Balancer:** This will create a load balancer assigned to a fixed IP address in the cloud, so long as the cloud provider supports it. In the case of Linode, this is the responsibility of the [Linode Cloud Controller Manager](#), which will create a NodeBalancer for the cluster. This is the best way to expose your cluster to the internet.
- **ExternalName:** Maps the Service to a DNS name by returning a CNAME record redirect. ExternalName is good for directing traffic to outside resources, such as a database that is hosted on another cloud.

Here is an example of a Service manifest that uses the v1 API:

my-apache-service.yaml

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: apache-service
5    labels:
6      app: web
7  spec:
8    type: NodePort
9    ports:
10   - port: 80
11     targetPort: 80
12     nodePort: 30020
13   selector:
14     app: web

```

Like the Pod example in the previous section, this manifest has a name and a label. Unlike the Pod example, this spec uses the ports field to define the exposed port on the container (port), and the target port on the Pod (targetPort). The type NodePort unlocks the use of nodePort field, which allows traffic on the host Node at that port. Lastly, the selector field targets only the Pods assigned the app: web label.

Volumes

A *volume* is a way to handle data storage on Kubernetes. Kubernetes offers many different types of these volumes using different storage methods. Generally, Kubernetes volumes are best classified under Ephemeral and Persistent types.

Ephemeral Kubernetes volumes differ from Docker volumes because they exist inside the Pod rather than inside the container. When a container gets restarted, the volume persists. Note, however, that ephemeral volumes on Kubernetes are still tied to the lifecycle of the Pod, so if the Pod gets destroyed, the volume gets destroyed with it.

Here is an example of how to create and use a basic ephemeral volume by creating a Pod manifest using the simple *emptyDir* volume type:

my-apache-pod-with-volume.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: apache-with-volume
5  spec:
6    volumes:
7    - name: apache-storage-volume
8      emptyDir: {}
9
10   containers:
11   - name: apache-container
12     image: httpd
13     volumeMounts:
14     - name: apache-storage-volume
15       mountPath: /data/apache-data
```

A volume has two unique aspects to its definition. In this example, the first aspect is the volumes block that defines the type of volume you want to create, which in this case is a simple empty directory (*emptyDir*). The second aspect is the volumeMounts field within the container's spec. This field is given the name of the volume you are creating and a mount path within the container.

Persistent volumes, as opposed to ephemeral storage, remain beyond the life of a Pod and are typically used for stateful applications (most commonly, databases). These are PersistentVolume objects that represent a piece of existing storage on your cluster generally located on cloud hosted storage devices. Linode, for example, offers a [Container Storage Interface \(CSI\)](#) driver that allows the cluster to persist data on a Block Storage volume.

Namespaces

Namespaces are a means of isolating resources and groups of resources that exist within the Kubernetes cluster, helping to organize objects. Every cluster has at least three namespaces: default, kube-system, and kube-public. When interacting with the cluster it is important to know which Namespace the object you are looking for is in as many commands will default to only showing you what exists in the default namespace. Resources created without an explicit namespace will be added to the default namespace.

Namespaces consist of alphanumeric characters, dashes (-), and periods (.). Here is an example of how to define a Namespace with a manifest:

my-namespace.yaml

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: my-app
```

Here is an example of a Pod with a Namespace:

my-apache-pod-with-namespace.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: apache-pod
5    labels:
6      app: web
7    namespace: my-app
8  spec:
9    containers:
10     - name: apache-containter
11       image: httpd
```

Controllers

A Controller is a control loop that continuously watches the Kubernetes API and tries to manage the desired state of certain aspects of the cluster. Here are short references of the most popular controllers.

Deployments

Deployments can keep a defined number of replica Pods up and running. A Deployment can also update those Pods to resemble the desired state by means of rolling updates. For example, if you want to update a container image to a newer version, you would create a Deployment. The controller would update the container images one by one until the desired state is achieved, ensuring that there is no downtime when updating or altering your Pods.

Here is an example of a Deployment:

my-apache-deployment.yaml

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: apache-deployment
5    labels:
6      app: web
7  spec:
8    replicas: 5
9    selector:
10     matchLabels:
11       app: web
12   template:
13     metadata:
14       labels:
15         app: web
16     spec:
17       containers:
18       - name: apache-container
19         image: httpd: 2.4.35

```

In this example, the number of replica Pods is set to five, meaning the Deployment will attempt to maintain five of a certain Pod at any given time. A Deployment chooses which Pods to include by use of the selector field. In this example, the selector mode is matchLabels, which instructs the Deployment to look for Pods defined with the app:web label.

As you will see, the only noticeable difference between a Deployment's manifest and that of a [ReplicaSet](#) is the kind.

ReplicaSets

Note: Kubernetes now [recommends](#) the use of Deployments instead of ReplicaSets. Deployments provide declarative updates to Pods, among other features, that allow you to define your application in the spec section. In this way, ReplicaSets have essentially become deprecated.

Kubernetes allows an application to scale horizontally. A *ReplicaSet* is one of the controllers responsible for keeping a given number of replica Pods running. If one Pod goes down in a ReplicaSet, another gets created to replace it. In this way, Kubernetes is *self-healing*. However, for most use cases, it is recommended to use a [Deployment](#) instead of a ReplicaSet.

my-apache-replicaset.yaml

```

1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    name: apache-replicaset
5    labels:
6      app: web
7  spec:
8    replicas: 5
9    selector:
10     matchLabels:
11       app: web
12   template:
13     metadata:
14       labels:
15         app: web
16     spec:
17       containers:
18       - name: apache-container
19         image: httpd

```

The `apiVersion` (`apps/v1`) differs from the previous examples, which were `apiVersion: v`, because ReplicaSets do not exist in the `v1` core. They instead reside in the `apps` group of `v1`. Also, note the `replicas` field and the `selector` field. The `replicas` field defines how many replica Pods you want to be running at any given time. The `selector` field defines which Pods, matched by their label, will be controlled by the ReplicaSet.

Jobs

Jobs manage a Pod created for a single or set of tasks. This is handy if you need to create a Pod that performs a single function or calculation of a value that can be terminated once the task is completed. The deletion of the Job will delete the Pod.

Here is an example of a Job that simply prints “Hello World!” and ends:

my-job.yaml

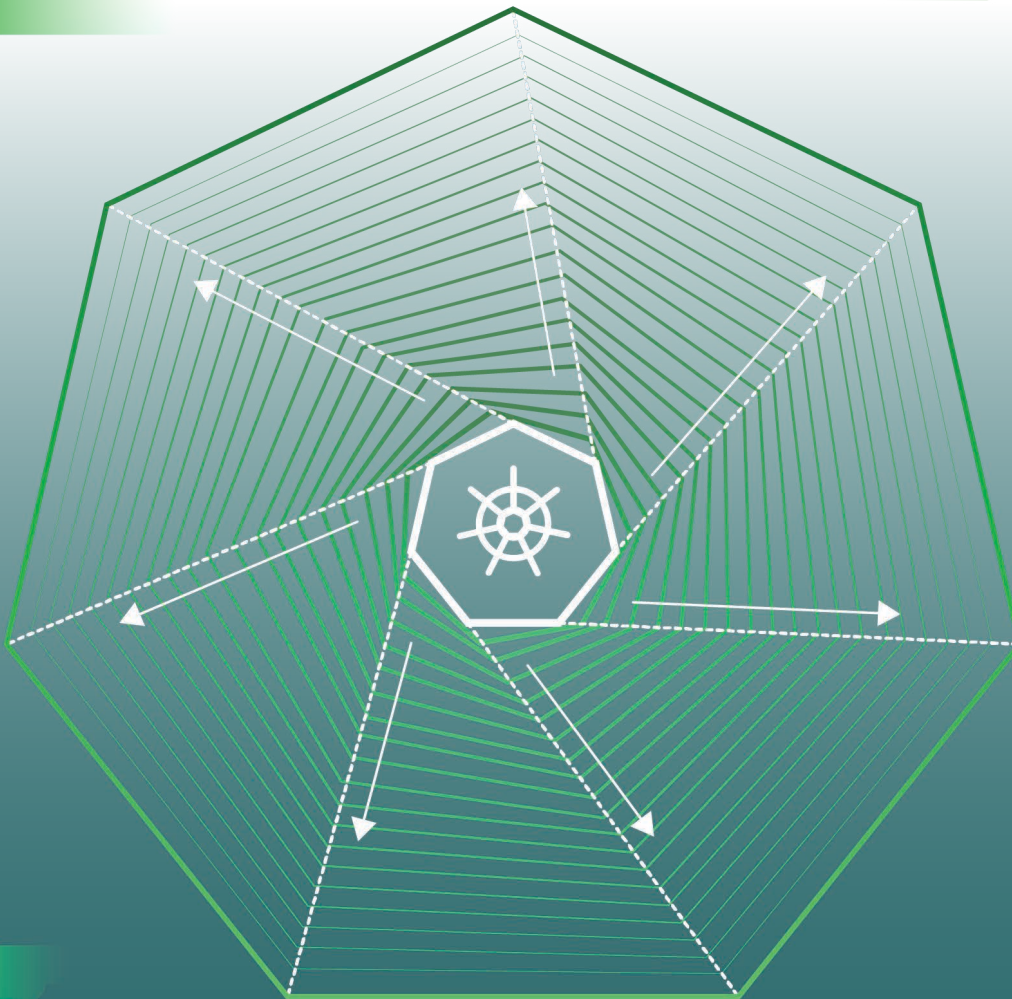
```
1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: hello-world
5  spec:
6    template:
7      metadata:
8        name: hello-world
9      spec:
10     containers:
11     - name: output
12       image: debian
13       command:
14         - "bin/bash"
15         - "-c"
16         - "echo 'Hello World!'"
17     restartPolicy: Never
```

Autoscaling

You can autoscale both Pods and clusters for Kubernetes. Horizontal Pod Autoscaling is native to Kubernetes and enables Pods to increase or decrease the number of nodes based on the amount of demand and a pre-set threshold.

[Horizontal Cluster Autoscaling](#) lets you automatically scale the number of nodes available in each cluster up or down based on the number of Pods available in your cluster by setting thresholds of min and max values for nodes in the node pool.

Autoscaling makes managing node pools more efficient, resulting in highly available and stable applications.



Networking

Networking in Kubernetes makes it simple to port existing apps from VMs to containers, and subsequently, Pods.

The basic requirements of the Kubernetes networking model are:

- Pods can communicate with each other across Nodes without the use of NAT.
- Agents on a Node, like kubelet, can communicate with all of a Node's Pods.
- In the case of Linux, Pods in a Node's host network can communicate to all other Pods without NAT.

Though the rules of the Kubernetes networking model are simple, the implementation of those rules is an advanced topic. Because Kubernetes does not come with its own implementation, it is up to the user to provide a networking model.

The most popular options are [Flannel](#) and [Calico](#).

- **Flannel** is a networking overlay that meets the functionality of the Kubernetes networking model by supplying a layer 3 network fabric and is relatively easy to set up.
- **Calico** enables networking and networking policy through the NetworkPolicy API to provide simple virtual networking.

Our Take

Business requirements are pressing developers to design distributed applications to run bigger, better, and faster. To keep up with demand and the constant need for highly-available and resource-efficient applications, developers are increasingly turning to containers and the necessity of Kubernetes to reduce management complexity. Kubernetes, while already an industry standard, is still a rapidly evolving and expanding platform that will remain a cornerstone of cloud native development.

The true impact of Kubernetes as an open source project increases as managed Kubernetes services become more affordable and widely available, and as more third-party integrations give developers the ability to customize their Kubernetes experiences. As the ecosystem continues to advance, developers are expanding their use cases for production workloads. It's critical to find a sustainable and affordable managed Kubernetes service designed to work for developers who are ready to use Kubernetes for production workloads or are simply exploring and testing Kubernetes for potential adoption.



Free Course: Linode Kubernetes Engine

Learn at your own pace in our free course
in collaboration with KodeKloud!

START NOW

Next Steps

A complex management tool, early in its development, Kubernetes often made workloads more complicated to deploy and manage instead of removing developers' burdens. Kubernetes solutions were known to be clunky and inefficient. However, as the open source community worked to build a more stable and reliable tool, the same occurred for managed services, including Linode Kubernetes Engine.

On Linode, developers gain access to powerful infrastructure without a premium price tag and minimize time spent on deployment by setting up a Kubernetes cluster and downloading its kubeconfig file in less than ten clicks.

Setting up your first cluster is just the beginning. Take a look at the following resources to master the basics and advance your Kubernetes knowledge.

- Ready to deploy your first cluster on Linode? Follow our guide to get started on [Linode Kubernetes Engine](#).
- Find a variety of Kubernetes courses on [edX.org](#) courtesy of the Linux Foundation.
- Become a [Certified Kubernetes Administrator](#) or [Certified Kubernetes Application Developer](#) through the Cloud Native Computing Foundation's online exams.

As Kubernetes continues to grow, so does the variety and availability of add-ons to incorporate new cluster administration features and customize networking, container visualization, and more. Check out the full list of [Addons on Kubernetes.io](#).

About Linode

Akamai's Linode cloud is one of the easiest-to-use and most trusted infrastructure-as-a-service platforms. Built by developers for developers, Linode accelerates innovation by making cloud computing simple, affordable, and accessible to all. Learn more at linode.com or follow Linode on [Twitter](#) and [LinkedIn](#).



CLOUD COMPUTING SERVICES FROM



Cloud Computing Developers Trust

linode.com | Support: 855-4-LINODE | Sales: 844-869-6072
249 Arch St., Philadelphia, PA 19106 Philadelphia, PA 19106