linode

# Understanding Kubernetes

A Guide to **Modernizing** Your **Cloud** Infrastructure

linode

EBOOK KUBERNETES

## Table of Contents

*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.*

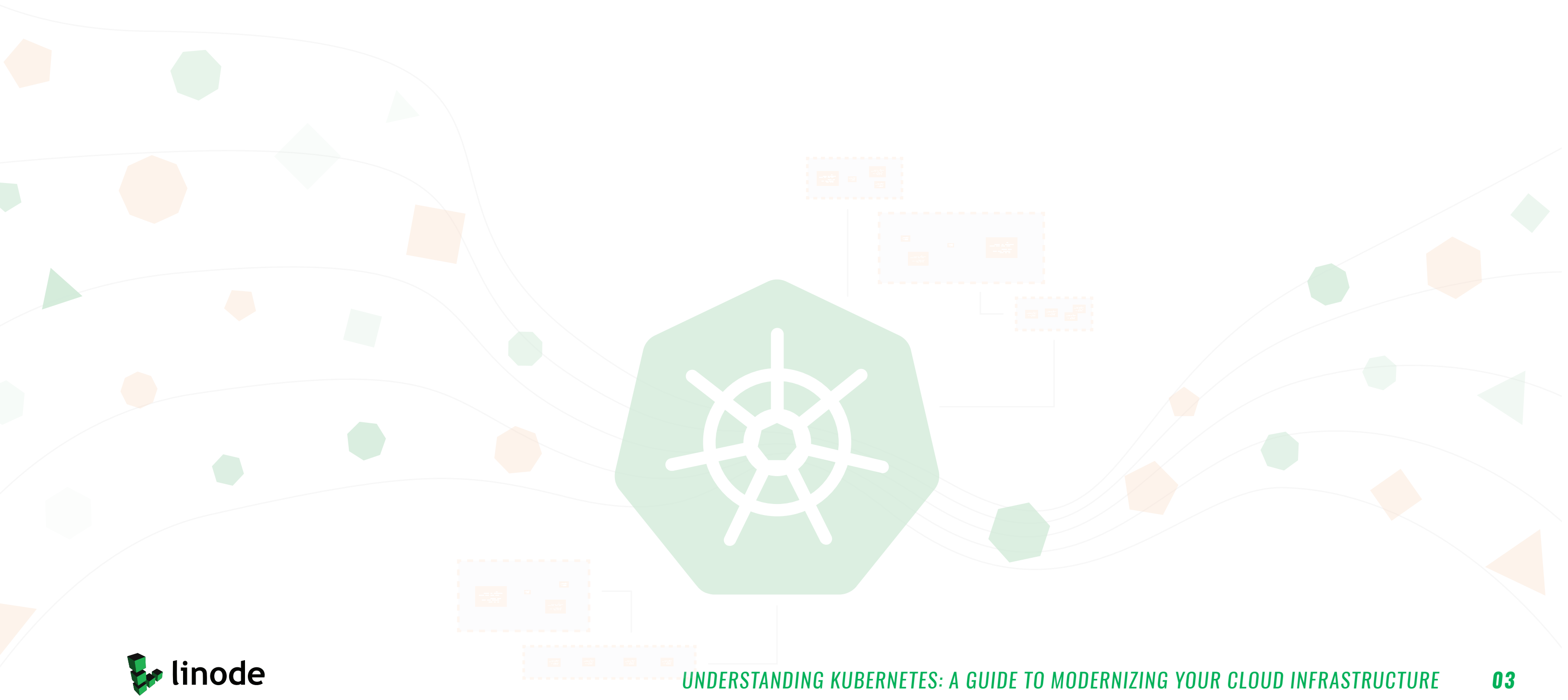
## Why Kubernetes?

**Kubernetes** is quickly becoming the new industry-standard tool for running cloud-native applications.

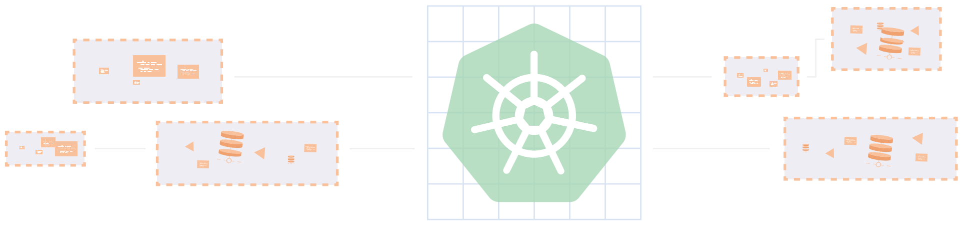
As more developers and organizations move away from on-premises infrastructure to take advantage of the cloud, advanced management is needed to ensure high availability and scalability for containerized applications. To build and maintain applications, you need to coordinate resources across different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

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

Kubernetes (K8s) optimizes container orchestration deployment and management for cloud infrastructure by creating groups, or Pods, of containers that scale without writing additional lines of code and responding to the needs of the application. Key benefits of moving to container-centric infrastructure with Kubernetes is 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 that was initially designed by Google to help 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. The **Master Node** contains a collection of processes called the control plane that helps enact and maintain the desired state of the Kubernetes cluster, while **Worker Nodes** are responsible for running the containers that form your applications and services.

## CONTAINERS

Kubernetes is a **container** orchestration tool and, therefore, needs a container runtime installed to work.

In practice, the default container runtime for Kubernetes is Docker, though other runtimes like rkt, and LXD will also work. With the advent of the Container Runtime Interface (CRI), which hopes to standardize the way Kubernetes interacts with containers, other options like containerd, cri-o, and Frakti are also available. Examples throughout this guide will use Docker as the container runtime.

- **Containers** are similar to virtual machines. They are light-weight isolated runtimes that share resources of the operating system without having to run a full operating system themselves. Containers consume few resources but contain a complete set of information needed to execute their contained application images such as files, environment variables, and libraries.
- **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; load balancing; resource allocation between containers; and health monitoring of the cluster.

## 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.

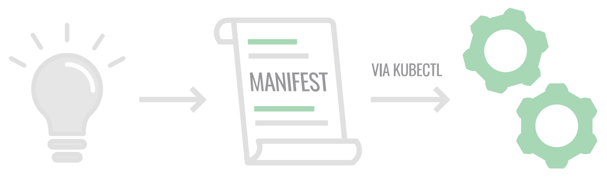
The Kubernetes API is a "declarative model," meaning that it focuses on the what, not the how. You tell it what you want to accomplish, and it does it. This might involve creating or destroying resources, but you don't have to worry about those details. To create this desired state, you create **objects**, which are normally represented by YAML files called **manifests**, and apply them through the command line with the **kubectl** tool.

## KUBECTL

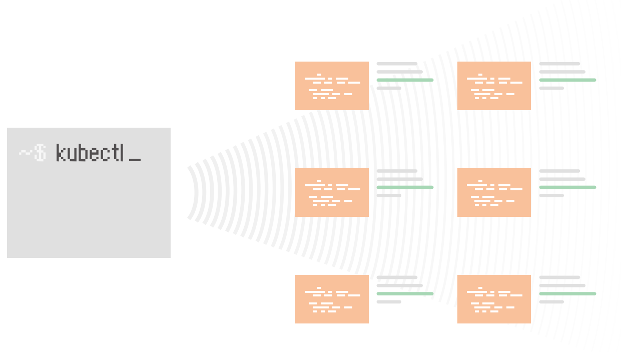
**kubectl** is a command line tool used to interact with the Kubernetes cluster. It offers a host of features, including the ability to create, stop, and delete resources; describe active resources; and auto scale resources.

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

Kubernetes API



kubectl



## Master, Nodes, and the Control Plane

At the highest level of Kubernetes, there exist two kinds of servers, a **Master** and a **Node**. These servers can be Linodes, VMs, or physical servers. Together, these servers form a cluster controlled by the services that make up the **Control Plane**.

For your Kuberentes cluster to maintain homeostasis for your application, it requires a central source of communications and commands. Your **Kubernetes Master**, **Nodes**, and **Control Plane** are the essential components that run and maintain your cluster. The Control Plane refers to the functions that make decisions about cluster maintenance, whereas the Master is what you interact with on the command-line interface to assess your cluster's state.

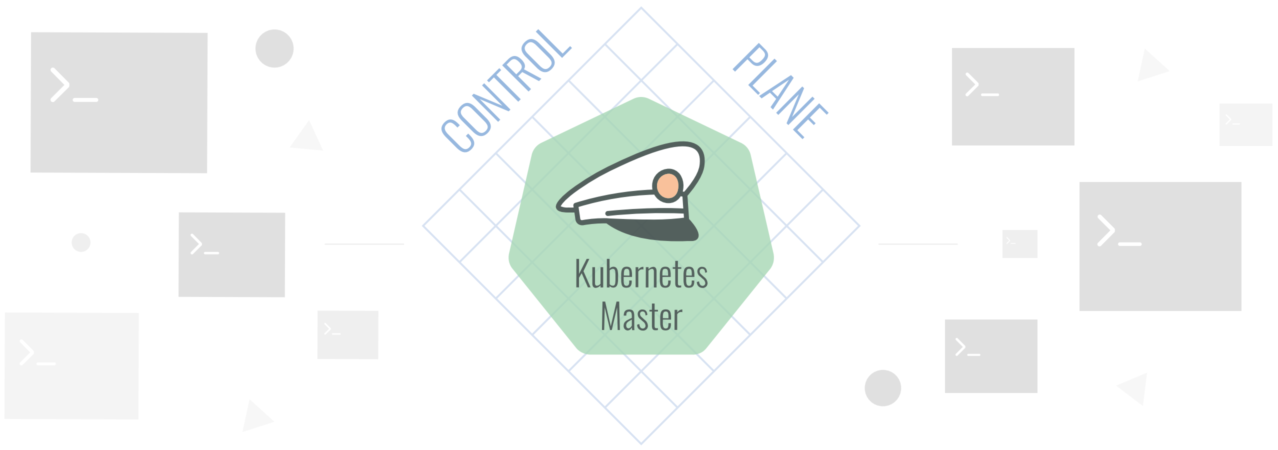
### KUBERNETES MASTER

The **Kubernetes Master** is normally a separate server responsible for maintaining the desired state of the cluster. It does this by telling the Nodes how many instances of your application it should run and where.

### 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:

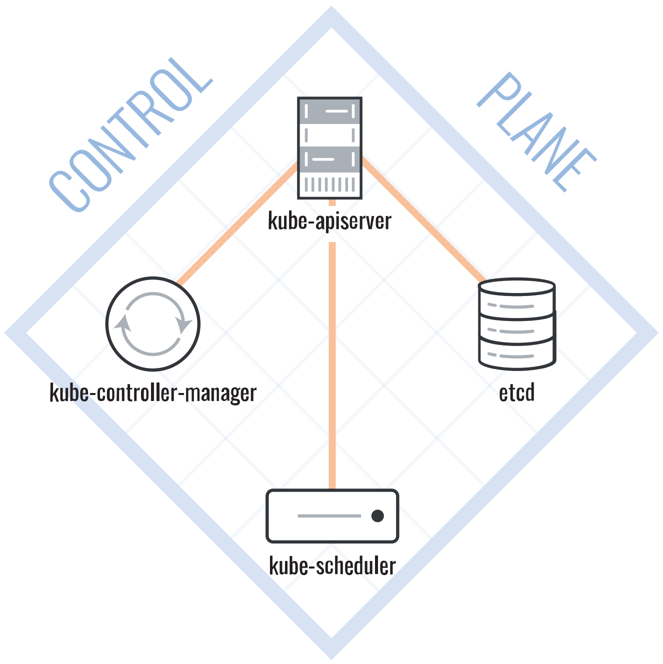
1. **kubelet** receives descriptions of the desired state of a Pod from the API server, and ensures the Pod is healthy, and running on the Node.
2. **kube-proxy** is a networking proxy that proxies the UDP, TCP, and SCTP networking of each Node, and provides load balancing. kube-proxy is only used to 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 is responsible for making decisions about the cluster and pushing it toward the desired state. kube-apiserver, kube-controller-manager, and kube-scheduler are processes, and etcd is a database; the Kubernetes Master runs all four.

- **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.



# Objects

In Kubernetes, 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.

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

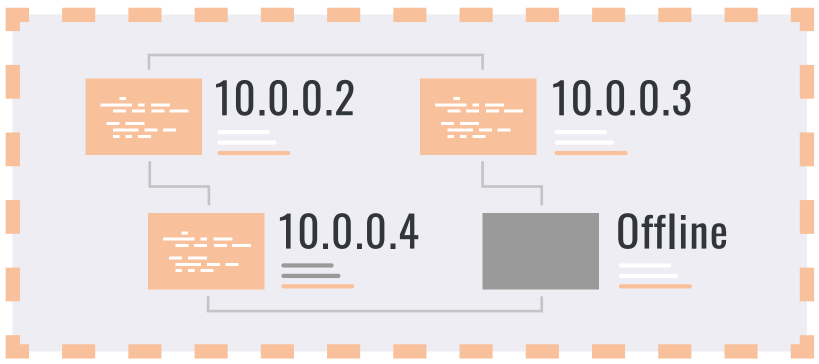
In Kubernetes, all containers exist within **Pods**. Pods are the smallest unit of the Kubernetes architecture. You can view them as a kind of wrapper for your container. Each Pod gets its own IP address with which it can interact with other Pods within the cluster.

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.

Pods



Here is an example of a Pod manifest:

my-apache-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: apache-pod
 labels:
   app: web
spec:
 containers:
 - name: apache-container
   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, and annotations. 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 containers 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 default container registry for Kubernetes.

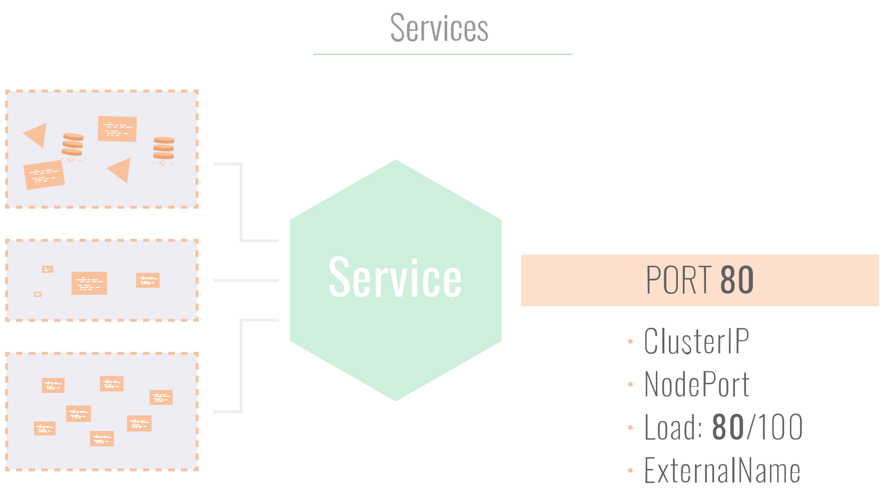
For more information on the type of fields you can supply in a Pod manifest, refer to the Kubernetes Pod API documentation.

## 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 load balance traffic between those Pods.

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. There are four 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.
- **LoadBalancer:** 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
```

```yaml
apiVersion: v1
kind: Service
metadata:
 name: apache-service
 labels:
   app: web
spec:
 type: NodePort
 ports:
  - ports: 80
    targetPort: 80
    nodePort: 30020
 selector:
    app: web
```

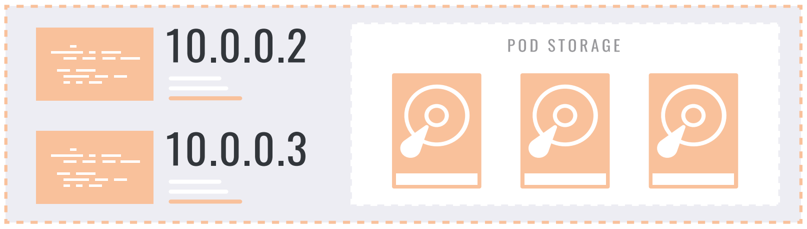
Like the Pod example in the previous section, this manifest has a name and a label. Unlike the Pod example, this spec uses the ports field to define the exposed port on the container (port), and the target port on the Pod (targetPort). The type NodePort unlocks the use of nodePort field, which allows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

## VOLUMES

A **Volume** in Kubernetes is a way to share file storage between containers in a Pod. 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 these Volumes are still tied to the lifecycle of the Pod, so if the Pod gets destroyed, the Volume gets destroyed with it.

Linode also offers a Container Storage Interface (CSI) driver that allows the cluster to persist data on a Block Storage volume.

Volumes



Here is an example of how to create and use a Volume by creating a Pod manifest:

```yaml
# my-apache-pod-with-volume.yaml
apiVersion: v1
kind: Pod
metadata:
 name: apache-with-volume
spec:
 volumes:
 - name: apache-storage-volume
   emptyDir: {}

   containers:
   - name: apache-container
     image: httpd
     volumeMounts:
     - name: apache-storage-volume
       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.

There are a number of different Volume types you could create in addition to emptyDir depending on your cloud host·

## NAMESPACES

Namespaces are virtual clusters that exist within the Kubernetes cluster that help to group and 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.

```yaml
# my-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
 name: my-app
```

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

A Deployment 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-pod-with-namespace.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: apache-pod
  labels:
    app: web
  namespace: my-app
spec:
  containers:
  - name: apache-container
    image: httpd
```

Namespaces consist of alphanumeric characters, dashes (-), and periods (.).

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.

my-apache-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: apache-deployment
 labels:
   app: web
spec:
 replicas: 5
 selector:
   matchLabels:
     app: web
 template:
   metadata:
     labels:
       app: web
   spec:
     containers:
     - name: apache-container
       image: httpd:2.4.35
```

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

# Controllers

## 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.

There are three important considerations regarding this ReplicaSet. First, 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.

my-apache-replicaset.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: apache-replicaset
 labels:
   app: web
spec:
 replicas: 5
 selector:
   matchLabels:
     app: web
 template:
   metadata:
     labels:
       app: web
   spec:
     containers:
     - name: apache-container
       image: httpd
```

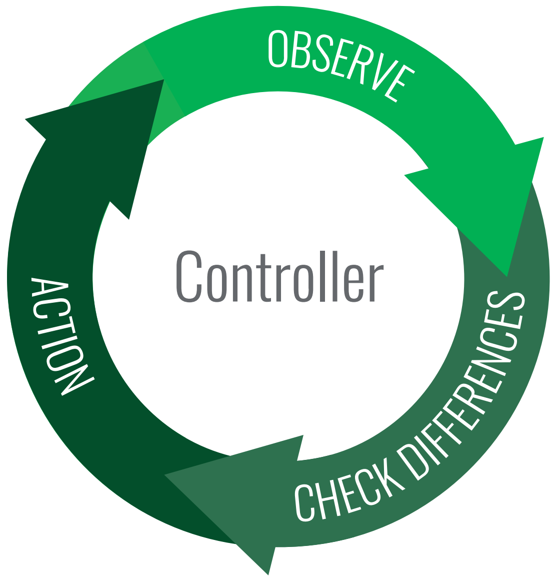
## JOBS

A Job is a controller that manages 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 calculates a value. 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

```
apiVersion: batch/v1
kind: Job
metadata:
 name: hello-world
spec:
 template:
  metadata:
   name: hello-world
  spec:
   containers:
   - name: output
     image: debian
     command:
     - "bin/bash"
     - "-c"
     - "echo 'Hello World!'"
   restartPolicy: Never
```



## Networking

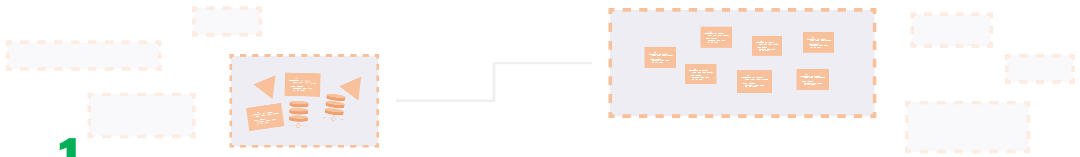
Networking in Kubernetes makes it simple to port existing apps from VMs to containers, and subsequently, Pods. The basic requirements of the Kubernetes networking model are:

- 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.

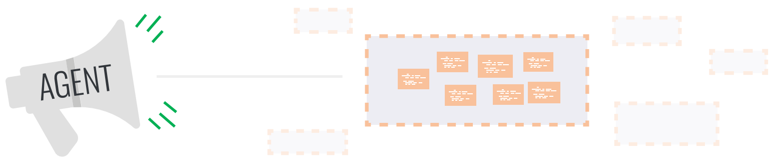
Though the rules of the Kubernetes networking model are simple, the implementation of those rules is an advanced topic. Because Kubernetes does not come with its own implementation, it is up to the user to provide a networking model.

Two of the most popular options are Flannel and Calico.

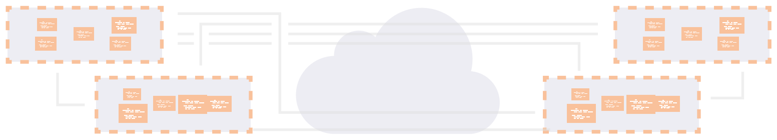
- **Flannel** is a networking overlay that meets the functionality of the Kubernetes networking model by supplying a layer 3 network fabric and is relatively easy to set up.
- **Calico** enables networking and networking policy through the NetworkPolicy API to provide simple virtual networking.



1. Pods can communicate with each other across Nodes without NAT



2. Agents on a Node can communicate with all of a Node's Pods



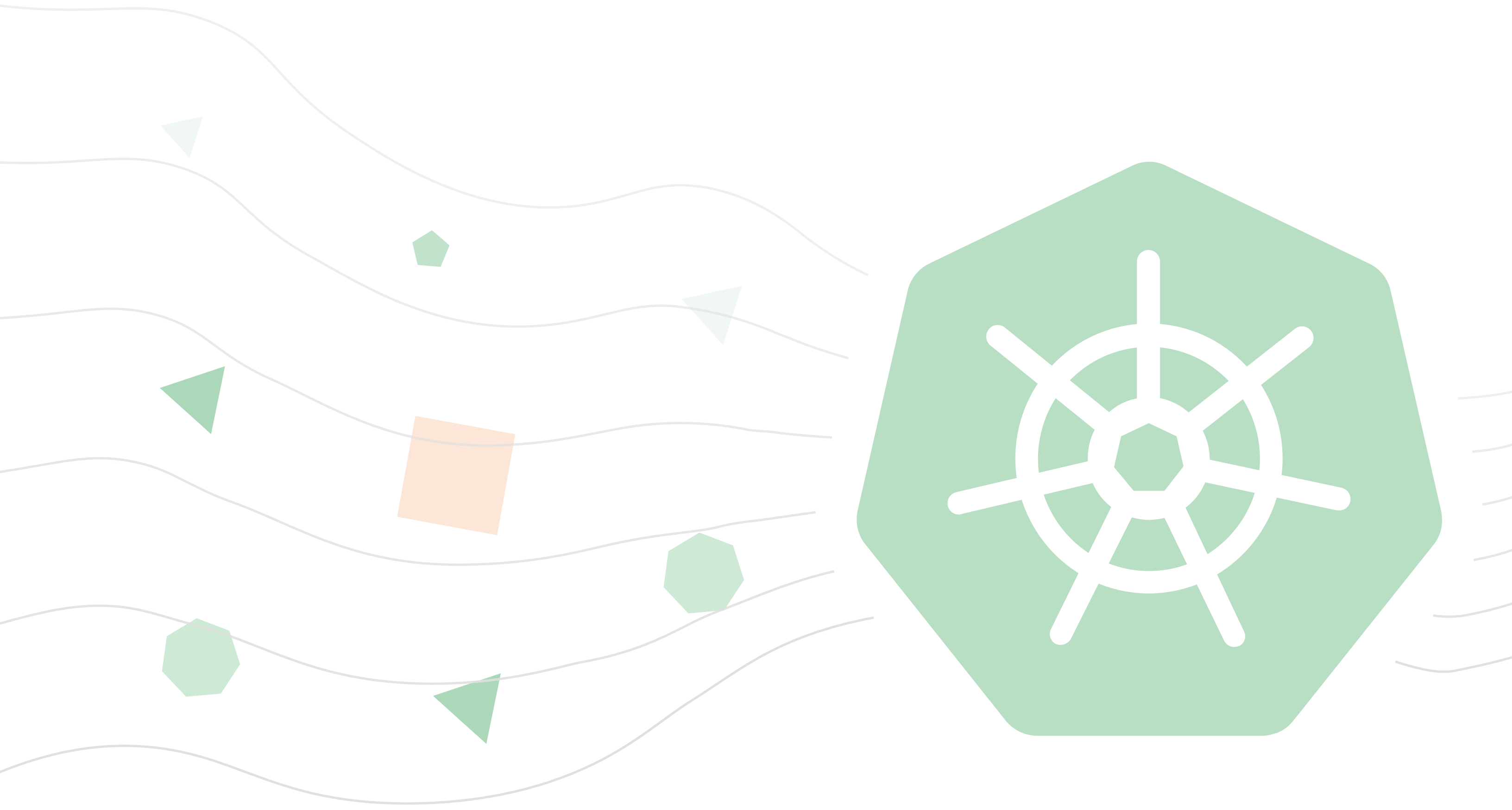
3. Pods in a Node's host network can communicate to all other Pods without NAT

## Our Take

Over the past decade, developers have used containers to both deploy and maintain applications more efficiently. As the use of container engines like Docker continued to grow, it was only a matter of time before developers would need to simplify this process even further. Kubernetes is the result of several years of innovation and advancing best practices to build an orchestrator that streamlines the complexity of containers.

Kubernetes is rapidly evolving. The true impact of Kubernetes as an open source project increases as managed Kubernetes services become more affordable, widely available, and as more third-party integrations give developers the ability to customize their Kubernetes experiences. As the ecosystem continues to advance, developers will expand their use cases for production workloads. Kubernetes is here to stay, and Kubernetes skills will become even more in-demand.

With the emergence of Kubernetes as a widely used tool in cloud computing, it's critical for developers to find sustainable and affordable managed Kubernetes services. Linode Kubernetes Engine (LKE) is designed to work for developers who are ready to use Kubernetes for production workloads with efficient and affordable resources, as well as developers who are simply exploring how Kubernetes will work for them.



## 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 K8s, including LKE.

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 with Linode Kubernetes Engine.
- Find a variety of Kubernetes courses on Linux Academy with topics ranging from basic cluster deployment to advanced configuration and security.
- 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

Our mission is to accelerate innovation by making cloud computing simple, affordable, and accessible to all.

Linode accelerates innovation by making cloud computing simple, accessible, and affordable to all. Founded in 2003, Linode helped pioneer the cloud computing industry and is today the largest independent open cloud provider in the world. Headquartered in Philadelphia's Old City, the company empowers more than a million developers, startups, and businesses across its global network of 11 data centers.

linode

The World's Largest
Independent Open Cloud