**Introduction**

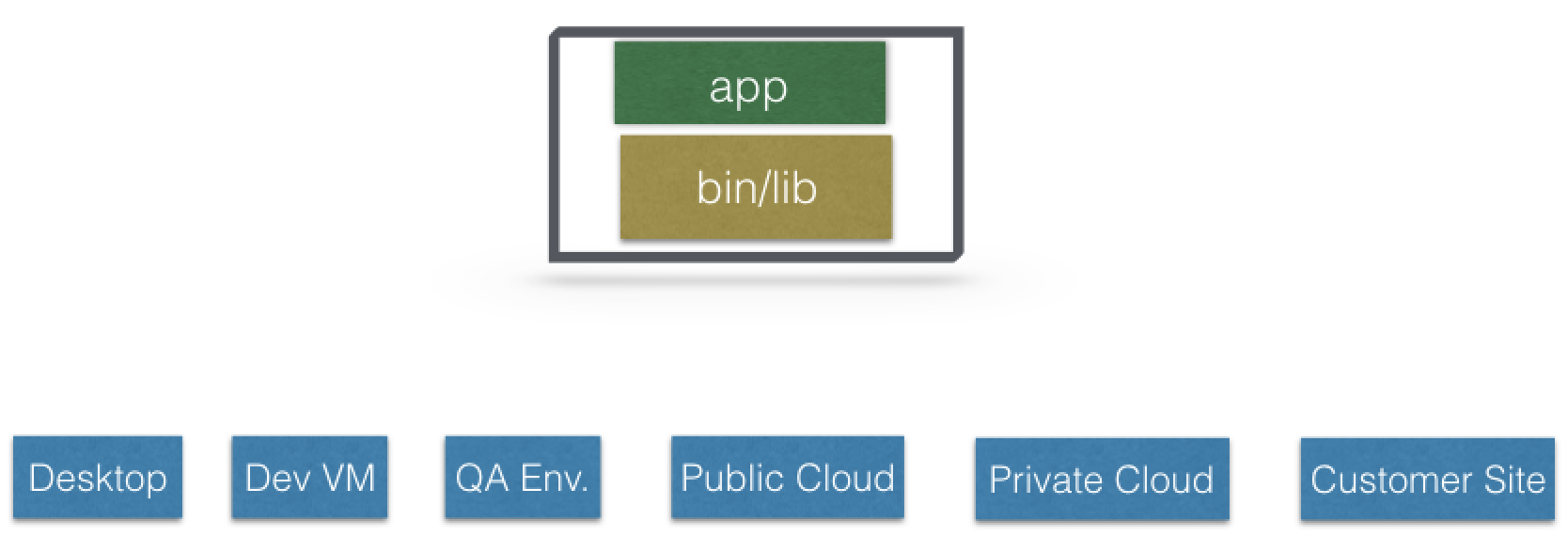
With container images, we confine the application code, its runtime, and all of its dependencies in a pre-defined format. And, with container runtimes like **runC**, **containerd**, or **rkt** we can use those pre-packaged images, to create one or more containers. All of these runtimes are good at running containers on a single host. But, in practice, we would like to have a fault-tolerant and scalable solution, which can be achieved by creating a single **controller/management unit,** after connecting multiple nodes together. This controller/management unitis generally referred to as a **container orchestrator.**

In this chapter, we will explore why we should use container orchestrators, different implementations of container orchestrators, and where to deploy them.

**What Are Containers?**

Before we dive into container orchestration, let's review first what containers are.

**Containers** are an application-centric way to deliver high-performing, scalable applications on the infrastructure of your choice.



**Containers**

With a **container image**, we bundle the application along with its runtime and dependencies. We use that image to create an isolated executable environment, also known as container. We can deploy containers from a given image on the platform of our choice, such as desktops, VMs, cloud, etc.

**What Is Container Orchestration?**

In the**quality assurance (QA)** **environments**, we can get away with running containers on a single host to develop and test applications. However, when we go to production, we do not have the same liberty, as we need to ensure that our applications:

* + - Are fault-tolerant
    - Can scale, and do this on-demand
    - Use resources optimally
    - Can discover other applications automatically, and communicate with each other
    - Are accessible from the external world
    - Can update/rollback without any downtime.

**Container orchestrators** are the tools which group hosts together to form a cluster, and help us fulfill the requirements mentioned above.

**Container Orchestrators**

Nowadays, there are many container orchestrators available, such as:

* + - **Docker Swarm**  
      [Docker Swarm](https://docs.docker.com/engine/swarm/) is a container orchestrator provided by [Docker, Inc](https://www.docker.com/). It is part of [Docker Engine](https://docs.docker.com/engine/).
    - **Kubernetes**  
      [Kubernetes](https://kubernetes.io/) was started by Google, but now, it is a part of the [Cloud Native Computing Foundation](https://www.cncf.io/) project.
    - **Mesos Marathon**  
      [Marathon](https://github.com/mesosphere/marathon) is one of the frameworks to run containers at scale on [Apache Mesos](http://mesos.apache.org/).
    - **Amazon ECS**  
      [Amazon EC2 Container Service](https://aws.amazon.com/ecs/) (ECS) is a hosted service provided by AWS to run Docker containers at scale on its infrastructrue.
    - **Hashicorp Nomad**  
      [Nomad](https://www.nomadproject.io/) is the container orchestrator provided by [HashiCorp](https://www.hashicorp.com/).

## Why Use Container Orchestrators?

Though we can argue that containers at scale can be maintained manually, or with the help of some scripts, container orchestrators can make things easy for operators.

Container orchestrators can:

* + - Bring multiple hosts together and make them part of a cluster
    - Schedule containers to run on different hosts
    - Help containers running on one host reach out to containers running on other hosts in the cluster
    - Bind containers and storage
    - Bind containers of similar type to a higher-level construct, like services, so we don't have to deal with individual containers
    - Keep resource usage in-check, and optimize it when necessary
    - Allow secure access to applications running inside containers.

With all these built-in benefits, it makes sense to use container orchestrators to manage containers. In this course, we will explore **Kubernetes**.

**Where to Deploy Container Orchestrators?**

Most container orchestrators can be deployed on the infrastructure of our choice. We can deploy them on bare metal, VMs, on-premise, or on a cloud of our choice. For example, Kubernetes can be deployed on our laptop/workstation, inside a company's datacenter, on AWS, on OpenStack, etc. There are even one-click installers available to set up Kubernetes on the cloud, like Google Kubernetes Engine on Google Cloud, or Azure Container Service on Microsoft Azure. Similar solutions are available for other container orchestrators, as well.

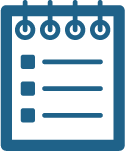
There are companies that offer managed Container Orchestration as a Service. We will explore them for Kubernetes in one of the later chapters.

**Introduction**

In this chapter, we will explain what **Kubernetes** is, its features, and the reasons why one should use it. We will explore the evolution of Kubernetes from **Borg**, which is a cluster manager created by Google.

We will also talk about the **Cloud Native Computing Foundation** **(CNCF)**, which currently hosts the Kubernetes project, along with other cloud-native projects, like Prometheus, Fluentd, rkt, containerd, etc.

**Learning Objectives**

By the end of this chapter, you should be able to:

* + - Define Kubernetes.
    - Explain the reasons for using Kubernetes.
    - Discuss the features of Kubernetes.
    - Discuss the evolution of Kubernetes from Borg.
    - Explain what the Cloud Native Computing Foundation does.

**What Is Kubernetes?**

According to the [Kubernetes website](https://kubernetes.io/),

*"Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."*

**Kubernetes** comes from the Greek word **κυβερνήτης:**, which means *helmsman* or *ship pilot*. With this analogy in mind, we can think of Kubernetes as the manager for shipping containers.

Kubernetes is also referred to as **k8s**, as there are 8 characters between *k* and *s*.

Kubernetes is highly inspired by the Google Borg system, which we will explore in this chapter. It is an open source project written in the Go language, and licensed under the [Apache License Version 2.0](https://www.apache.org/licenses/LICENSE-2.0).

Kubernetes was started by Google and, with its v1.0 release in July 2015, Google donated it to the [Cloud Native Computing Foundation](https://www.cncf.io/) (CNCF). We will talk more about CNCF later in this chapter.

Generally, Kubernetes has new releases every three months. The current stable version is 1.9 (as of February 2018).

**From Borg to Kubernetes**

According to the abstract of Google's [Borg paper](https://research.google.com/pubs/pub43438.html), published in 2015,

*"Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines."*

For more than a decade, Borg was Google's secret to run containerized workloads in production. Whatever services we use from Google, like Gmail, Drive, etc., they are all serviced using Borg.

Some of the initial authors of Kubernetes were Google employees who have used Borg and developed it in the past. They poured in their valuable knowledge and experience while designing Kubernetes. Some of the features/objects of Kubernetes that can be traced back to Borg, or to lessons learnt from it, are:

* + - API servers
    - Pods
    - IP-per-Pod
    - Services
    - Labels.

We will explore all of them, and more, in this course.

**Kubernetes Features I**

Kubernetes offers a very rich set of features for container orchestration. Some of its fully supported features are:

* + - **Automatic binpacking**  
      Kubernetes automatically schedules the containers based on resource usage and constraints, without sacrificing the availability.
    - **Self-healing**  
      Kubernetes automatically replaces and reschedules the containers from failed nodes. It also kills and restarts the containers which do not respond to health checks, based on existing rules/policy.
    - **Horizontal scaling**  
      Kubernetes can automatically scale applications based on resource usage like CPU and memory. In some cases, it also supports dynamic scaling based on customer metrics.
    - **Service discovery and Load balancing**  
      Kubernetes groups sets of containers and refers to them via a Domain Name System (DNS). This DNS is also called a Kubernetes **service**. Kubernetes can discover these services automatically, and load-balance requests between containers of a given service.

## Kubernetes Features II

Some other fully supported Kubernetes features are:

* + - **Automated rollouts and rollbacks**  
      Kubernetes can roll out and roll back new versions/configurations of an application, without introducing any downtime.
    - **Secrets and configuration management**  
      Kubernetes can manage secrets and configuration details for an application without re-building the respective images. With secrets, we can share confidential information to our application without exposing it to the stack configuration, like on GitHub.
    - **Storage orchestration**  
      With Kubernetes and its plugins, we can automatically mount local, external, and storage solutions to the containers in a seamless manner, based on software-defined storage (SDS).
    - **Batch execution**Besides long running jobs, Kubernetes also supports batch execution.

There are many other features besides the ones we just mentioned, and they are currently in alpha/beta phase. They will add great value to any Kubernetes deployment once they become stable features. For example, support for role-based access control (RBAC) is  stable as of the Kubernetes 1.8 release.

**Why Use Kubernetes?**

We just looked at some of the fully-supported Kubernetes features. We should also mention that Kubernetes is very portable and extensible. Kubernetes can be deployed on the environment of our choice, be it VMs, bare metal, or public/private/hybrid/multi-cloud setups. Also, Kubernetes has a very modular and pluggable architecture. We can write custom APIs or plugins to extend its functionalities.

For a successful open source project, the community is as important as having great code. Kubernetes has a very thriving community across the world. It has more than 1600 contributors, who, over time, have done over 62,000 commits. There are meet-up groups in different cities which meet regularly to discuss Kubernetes and its ecosystem. There are *Special Interest Groups* (SIGs), which focus on special interests, such as scaling, bare metal, networking, etc. We will talk more about them in our last chapter, *Kubernetes Communities*.

**Kubernetes Users**

With just a few years since its debut, many companies are running workloads using Kubernetes. We can find numerous [user case studies](https://kubernetes.io/case-studies/) on the Kubernetes website:

* + - [Pearson](https://kubernetes.io/case-studies/pearson/)
    - [Box](https://blog.box.com/blog/kubernetes-box-microservices-maximum-velocity/)
    - [eBay](https://www.nextplatform.com/2015/11/12/inside-ebays-shift-to-kubernetes-and-containers-atop-openstack/)
    - [Wikimedia](https://www.youtube.com/watch?v=6XGUTu3WhBw)
    - [Huawei](https://kubernetes.io/case-studies/huawei/)
    - [Haufe Group](https://kubernetes.io/case-studies/haufegroup/)
    - [BlackRock](https://kubernetes.io/case-studies/blackrock/)
    - [BlaBlaCar](https://kubernetes.io/case-studies/blablacar/)
    - And many more.

## Cloud Native Computing Foundation (CNCF)

The [Cloud Native Computing Foundation](https://www.cncf.io/) (CNCF) is one of the projects hosted by [The Linux Foundation](https://www.linuxfoundation.org/). CNCF aims to accelerate the adoption of containers, microservices, and cloud-native applications.



CNCF hosts a set of projects, with more to be added in the future. CNCF provides resources to each of the projects, but, at the same time, each project continues to operate independently under its pre-existing governance structure and with its existing maintainers. At the time this course was created, the following projects were part of CNCF:

* + - [containerd](http://containerd.io/) for container runtime
    - [rkt](https://github.com/rkt/rkt) for container runtime
    - [Kubernetes](https://kubernetes.io/) for container orchestration
    - [Linkerd](https://linkerd.io/) for service mesh
    - [Envoy](https://github.com/envoyproxy/envoy) for service mesh
    - [gRPC](http://www.grpc.io/) for remote procedure call (RPC)
    - [Container Network Interface](https://github.com/containernetworking/cni) (CNI) for networking API
    - [CoreDNS](https://coredns.io/) for service discovery
    - [Rook](https://github.com/rook/rook) for cloud-native storage
    - [Notary](https://github.com/theupdateframework/notary) for security
    - [The Update Framework](https://github.com/theupdateframework/specification) (TUF) for software updates
    - [Prometheus](https://prometheus.io/) for monitoring
    - [OpenTracing](http://opentracing.io/) for tracing
    - [Jaeger](https://github.com/jaegertracing/jaeger) for distributed tracing
    - [Fluentd](http://www.fluentd.org/) for logging
    - [Vitess](http://vitess.io/) for storage.

As we can see, this set of CNCF projects can cover the entire lifecycle of an application, from its execution using container runtimes, to its monitoring and logging. This is very important to meet the CNCF goal.

**CNCF and Kubernetes**

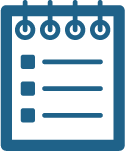
For Kubernetes, the Cloud Native Computing Foundation:

* + - Provides a neutral home for the Kubernetes trademark and enforces proper usage
    - Provides license scanning of core and vendored code
    - Offers legal guidance on patent and copyright issues
    - Creates open source [curriculum](https://github.com/cncf/curriculum), [training](https://training.linuxfoundation.org/linux-courses/system-administration-training/kubernetes-fundamentals), and [certification](https://www.cncf.io/certification/expert/)
    - Manages a software conformance [working group](https://ponymail.cncf.io/thread.html/Zaw9xi4cg7fx9v6)
    - Actively markets Kubernetes
    - Hosts and funds developer marketing activities like [K8Sport](http://k8sport.org/)
    - Supports ad hoc activities
    - Funds conferences and meetup events.

**Introduction**

In this chapter, we will explore the **Kubernetes** architecture, the different components of the **master** and **worker** **nodes**, the cluster state management with **etcd**and the network setup requirements. We will also talk about the network specification called **Container Network Interface (CNI)**, which is used by Kubernetes.

**Learning Objectives**

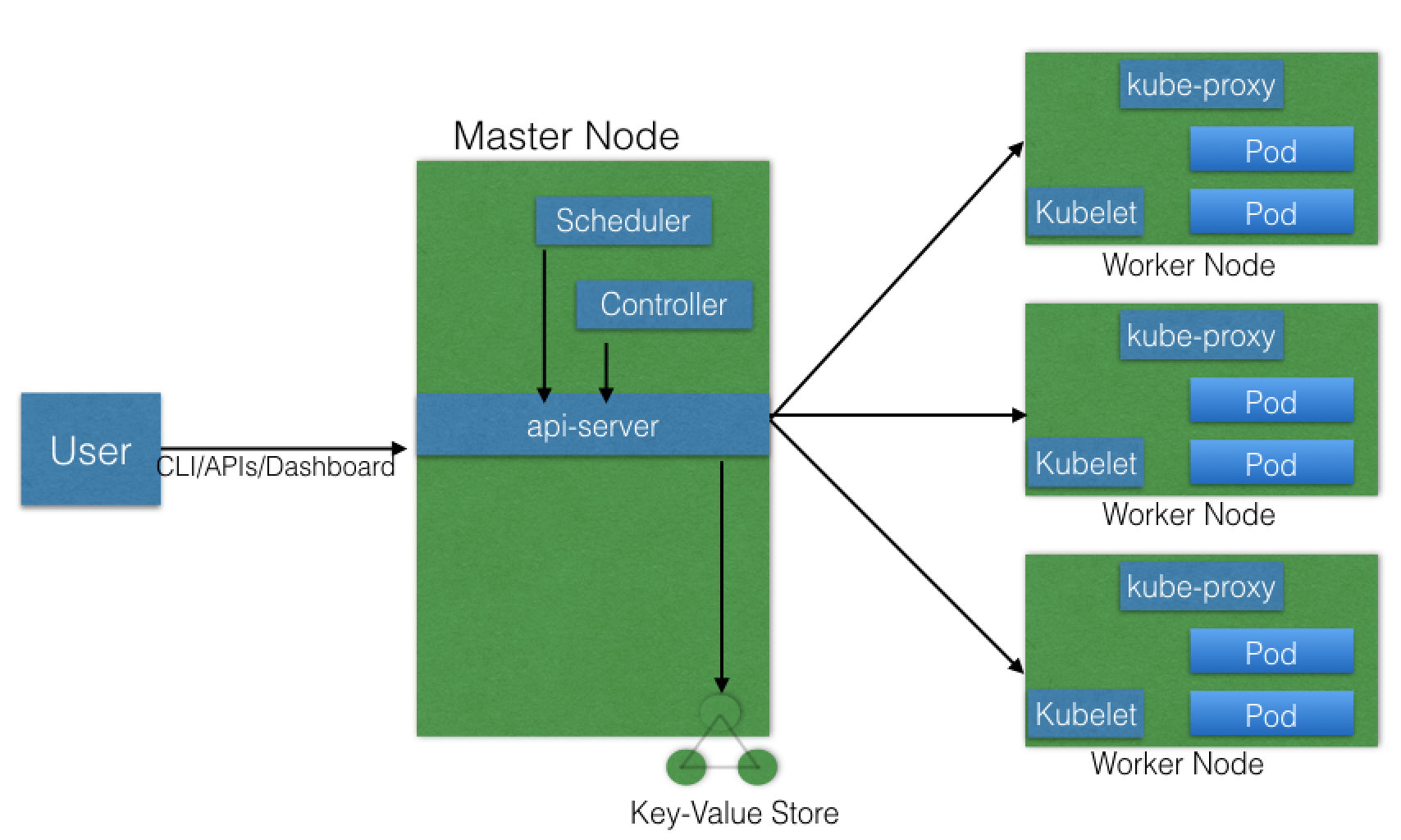
By the end of this chapter, you should be able to:

* + - Discuss the Kubernetes architecture.
    - Explain the different components for master and worker nodes.
    - Discuss about cluster state management with etcd.
    - Review the Kubernetes network setup requirements.

**Kubernetes Architecture**

At a very high level, Kubernetes has the following main components:

* + One or more **master nodes**
  + One or more **worker nodes**
  + Distributed key-value store, like **etcd**.

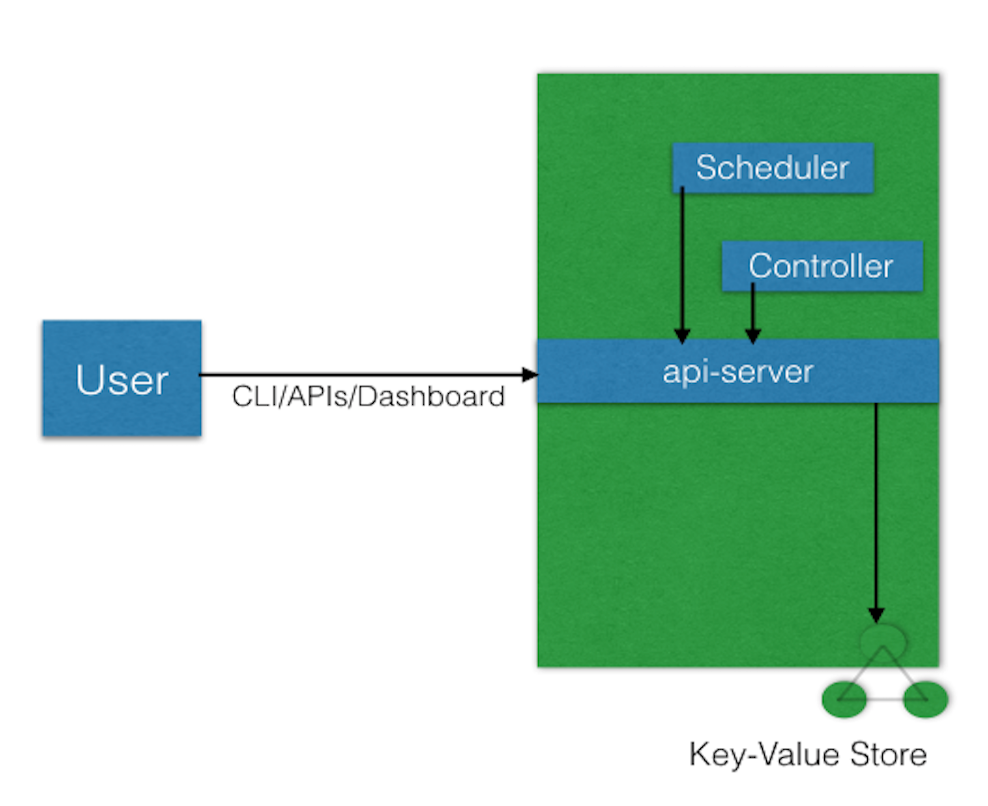


**Kubernetes Architecture**

Next, we will explore the Kubernetes architecture in more detail.

**Master Node**

The **master node** is responsible for managing the Kubernetes cluster, and it is the entry point for all administrative tasks. We can communicate to the master node via the CLI, the GUI (Dashboard), or via APIs.



**Kubernetes Master Node**

For fault tolerance purposes, there can be more than one master node in the cluster. If we have more than one master node, they would be in a HA (High Availability) mode, and only one of them will be the leader, performing all the operations. The rest of the master nodes would be followers.

To manage the cluster state, Kubernetes uses [etcd](https://coreos.com/etcd/), and all master nodes connect to it. **etcd** is a distributed key-value store, which we will discuss in a little bit. The key-value store can be part of the master node. It can also be configured externally, in which case, the master nodes would connect to it.

**Master Node Components**

A master node has the following components:

* + - API server
    - Scheduler
    - Controller manager
    - etcd.

## Master Node Components: API Server

All the administrative tasks are performed via the **API server** within the master node. A user/operator sends REST commands to the API server, which then validates and processes the requests. After executing the requests, the resulting state of the cluster is stored in the distributed key-value store.

**Master Node Components: Scheduler**

As the name suggests, the **scheduler** schedules the work to different worker nodes. The scheduler has the resource usage information for each worker node. It also knows about the constraints that users/operators may have set, such as scheduling work on a node that has the label **disk==ssd** set. Before scheduling the work, the scheduler also takes into account the quality of the service requirements, data locality, affinity, anti-affinity, etc. The scheduler schedules the work in terms of Pods and Services.

## Master Node Components: Controller Manager

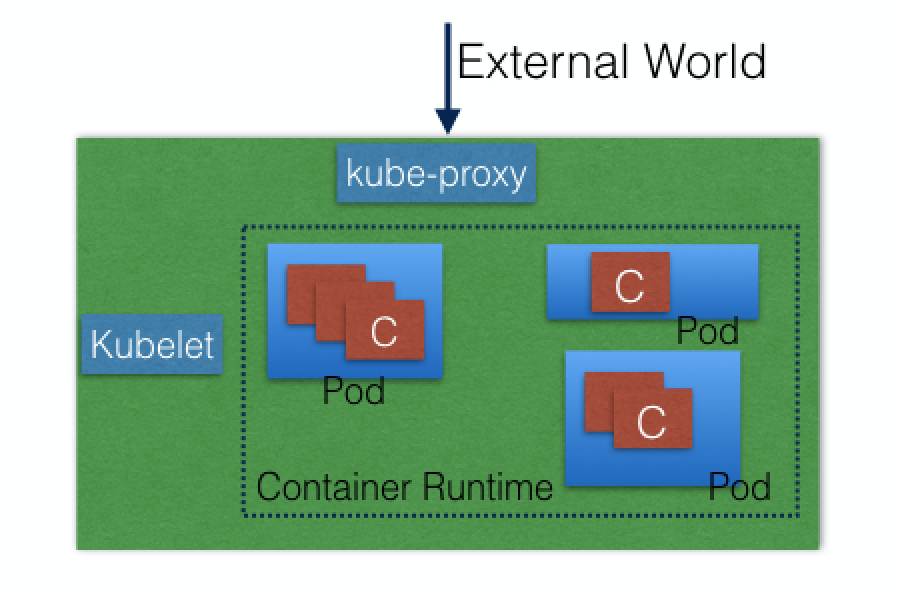
The **controller manager** manages different non-terminating control loops, which regulate the state of the Kubernetes cluster. Each one of these control loops knows about the desired state of the objects it manages, and watches their current state through the API server. In a control loop, if the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the current state is the same as the desired state.

## Master Node Components: etcd

As discussed earlier, **etcd** is a distributed key-value store which is used to store the cluster state. It can be part of the Kubernetes Master, or, it can be configured externally, in which case, master nodes would connect to it.

## Worker Node

A **worker node** is a machine (VM, physical server, etc.) which runs the applications using Pods and is controlled by the master node. Pods are scheduled on the worker nodes, which have the necessary tools to run and connect them. A Pod is the scheduling unit in Kubernetes. It is a logical collection of one or more containers which are always scheduled together. We will explore them further in later chapters.



**Kubernetes Worker Node**

Also, to access the applications from the external world, we connect to worker nodes and not to the master node/s. We will dive deeper into this in future chapters.

## Worker Node Components

A worker node has the following components:

* + - Container runtime
    - kubelet
    - kube-proxy.

In the next few sections, we will discuss them in more detail.

## Worker Node Components: Container Runtime

To run and manage a container's lifecycle, we need a **container runtime** on the worker node. Some examples of container runtimes are:

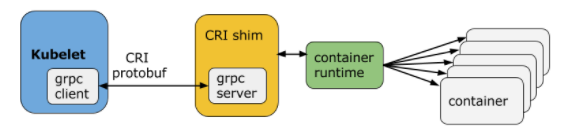
* + - [containerd](https://containerd.io/)
    - [rkt](https://coreos.com/rkt/)
    - [lxd](https://linuxcontainers.org/lxd/).

Sometimes, [Docker](https://www.docker.com/) is also referred to as a container runtime, but to be precise, Docker is a platform which uses **containerd** as a container runtime.

**Worker Node Components: kubelet**

The **kubelet** is an agent which runs on each worker node and communicates with the master node. It receives the Pod definition via various means (primarily, through the API server), and runs the containers associated with the Pod. It also makes sure that the containers which are part of the Pods are healthy at all times.

The kubelet connects to the container runtime using **Container Runtime Interface (CRI)**. The [Container Runtime Interface](https://github.com/kubernetes/community/blob/master/contributors/devel/container-runtime-interface.md) consists of protocol buffers, gRPC API, and libraries.



**Container Runtime Interface**(Retrieved from [blog.kubernetes.io](http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html))

As shown above, the kubelet (grpc client) connects to the CRI shim (grpc server) to perform container and image operations. CRI implements two services: **ImageService** and **RuntimeService**. The **ImageService** is responsible for all the image-related operations, while the **RuntimeService** is responsible for all the Pod and container-related operations.

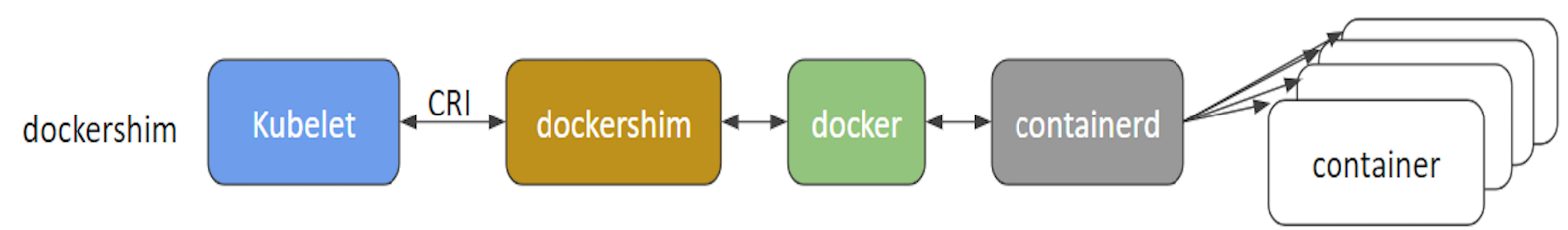
Container runtimes used to be hard-coded in Kubernetes, but with the development of CRI, Kubernetes can now use different container runtimes without the need to recompile. Any container runtime that implements CRI can be used by Kubernetes to manage Pods, containers, and container images.

In the next section, we will discuss some of the CRI shims.

**Worker Node Components: kubelet: CRI shims**

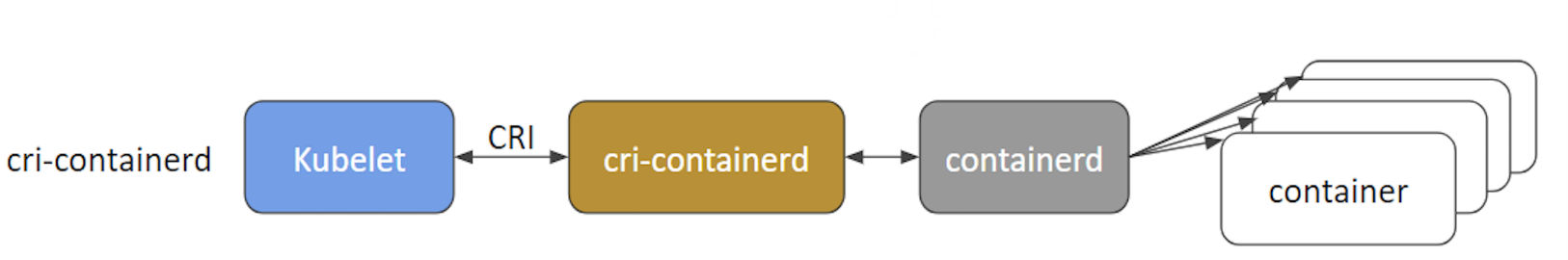
Below you will find some examples of CRI shims:

* + - **dockershim**  
      With dockershim, containers are created using Docker installed on the worker nodes. Internally, Docker uses containerd to create and manage containers.



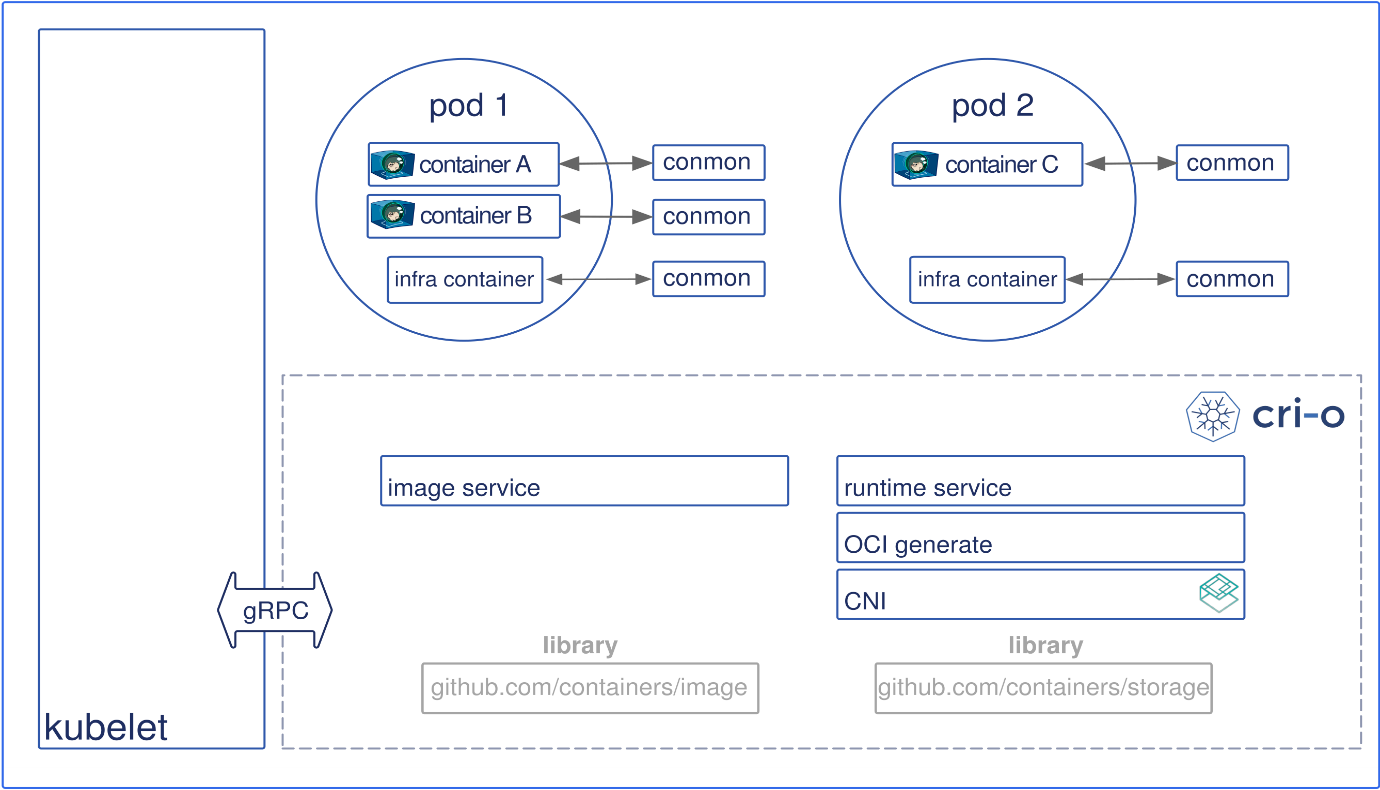
**dockershim**(Retrieved from [blog.kubernetes.io](http://blog.kubernetes.io/2017/11/containerd-container-runtime-options-kubernetes.html))

* + - **cri-containerd**  
      With cri-containerd, we can directly use Docker's smaller offspring containerd to create and manage containers.



**cri-containerd**(Retrieved from [blog.kubernetes.io](http://blog.kubernetes.io/2017/11/containerd-container-runtime-options-kubernetes.html))

* + - **CRI-O**  
      CRI-O enables using any Open Container Initiative (OCI) compatible runtimes with Kubernetes. At the time this course was created, CRI-O supported runC and Clear Containers as container runtimes. However, in principle, any OCI-compliant runtime can be plugged-in.



**CRI-O**

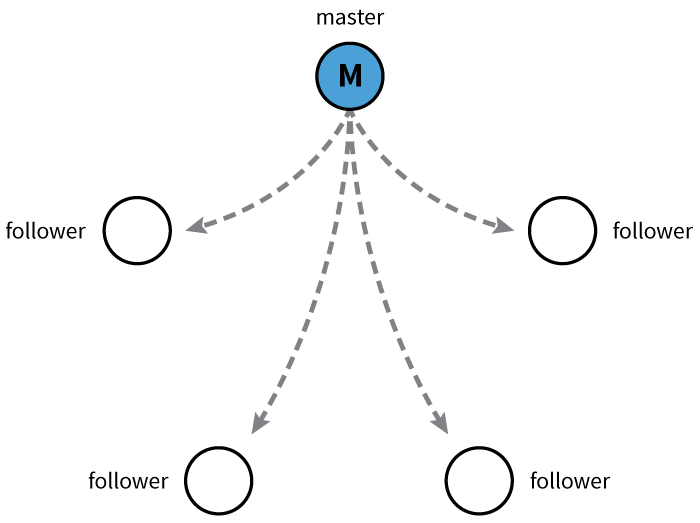
**Worker Node Components: kube-proxy**

Instead of connecting directly to Pods to access the applications, we use a logical construct called a **Service** as a connection endpoint. A Service groups related Pods and, when accessed, load balances to them. We will talk more about Services in later chapters.

**kube-proxy** is the network proxy which runs on each worker node and listens to the API server for each Service endpoint creation/deletion. For each Service endpoint, kube-proxy sets up the routes so that it can reach to it. We will also explore this in more detail in later chapters.

## State Management with etcd

As we mentioned earlier, Kubernetes uses **etcd** to store the cluster state. etcd is a distributed key-value store based on the [Raft Consensus Algorithm](https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14). Raft allows a collection of machines to work as a coherent group that can survive the failures of some of its members. At any given time, one of the nodes in the group will be the master, and the rest of them will be the followers. Any node can be treated as a master.



**Master and Followers**

etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.

**Network Setup Challenges**

To have a fully functional Kubernetes cluster, we need to make sure of the following:

* + - A unique IP is assigned to each Pod
    - Containers in a Pod can communicate to each other
    - The Pod is able to communicate with other Pods in the cluster
    - If configured, the application deployed inside a Pod is accessible from the external world.

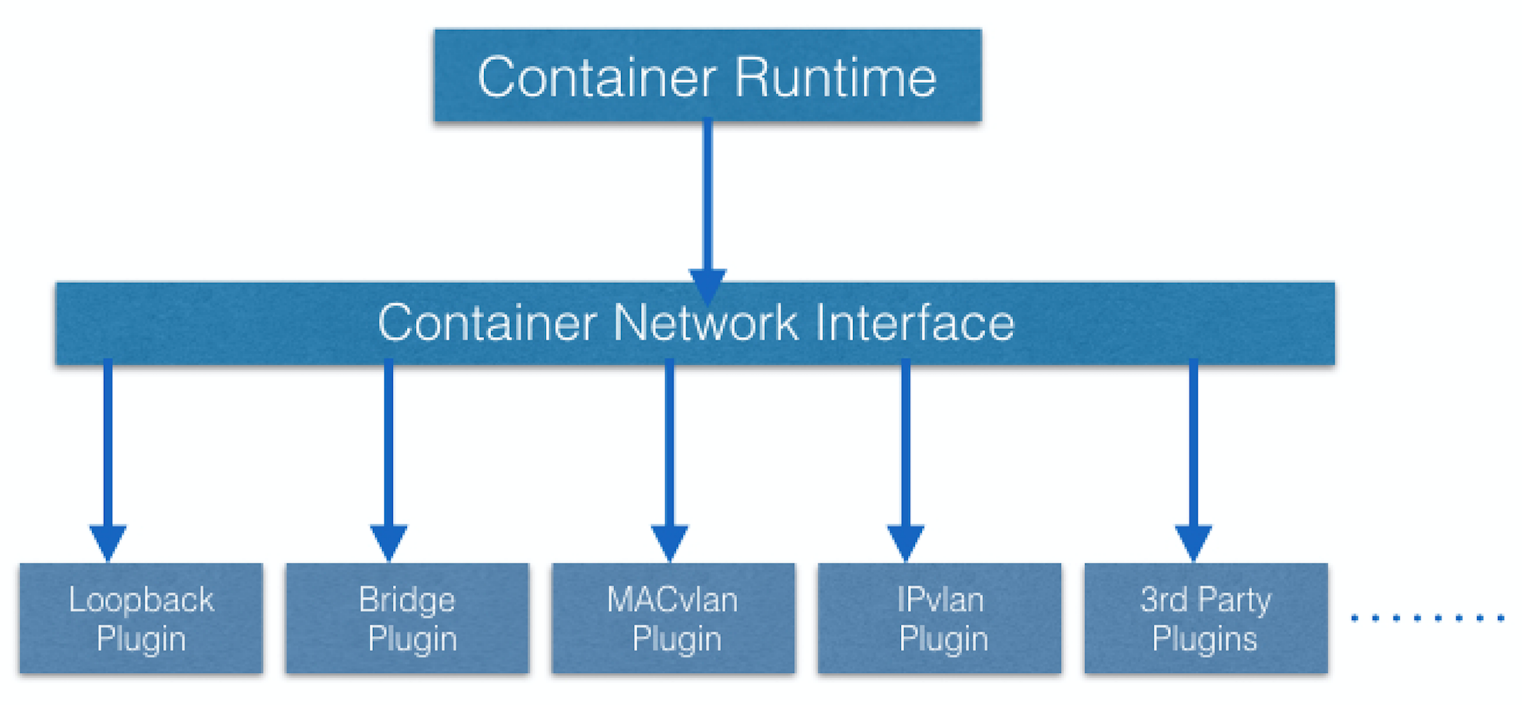
All of the above are networking challenges which must be addressed before deploying the Kubernetes cluster. Next, we will see how we can solve these challenges.

**Assigning a Unique IP Address to Each Pod**

In Kubernetes, each Pod gets a unique IP address. For container networking, there are two primary specifications:

* + - **Container Network Model (CNM)**, proposed by Docker
    - **Container Network Interface (CNI)**, proposed by CoreOS.

Kubernetes uses CNI to assign the IP address to each Pod.



**Container Network Interface (CNI)**

The container runtime offloads the IP assignment to CNI, which connects to the underlying configured plugin, like Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested container runtime.

**Container-to-Container Communication Inside a Pod**

With the help of the underlying host operating system, all of the container runtimes generally create an isolated network entity for each container that it starts. On Linux, that entity is referred to as a **network namespace**. These network namespaces can be shared across containers, or with the host operating system.

Inside a Pod, containers share the network namespaces, so that they can reach to each other via localhost.

**Pod-to-Pod Communication Across Nodes**

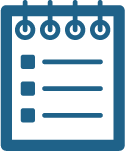
In a clustered environment, the Pods can be scheduled on any node. We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able to reach any Pod. Kubernetes also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-to-Pod communication across hosts. We can achieve this via:

* + - Routable Pods and nodes, using the underlying physical infrastructure, like Google Kubernetes Engine
    - Using Software Defined Networking, like [Flannel](https://coreos.com/flannel/docs/latest/), [Weave](https://www.weave.works/oss/net/), [Calico](https://www.projectcalico.org/), etc.

## Communication Between the External World and Pods

By exposing our services to the external world with **kube-proxy**, we can access our applications from outside the cluster. We will have a complete chapter dedicated to this, so we will dive into this later.

**Learning Objectives**

By the end of this chapter, you should be able to:

* + - Discuss about the different Kubernetes configuration options.
    - Discuss infrastructure considerations before installing Kubernetes.
    - Discuss infrastructure choices for a Kubernetes deployment.
    - Review Kubernetes installation tools and resources.

## Kubernetes Configuration

Kubernetes can be installed using different configurations. The four major installation types are briefly presented below:

* + - **All-in-One Single-Node Installation**  
      With all-in-one, all the master and worker components are installed on a single node. This is very useful for learning, development, and testing. This type should not be used in production. Minikube is one such example, and we are going to explore it in future chapters.
    - **Single-Node etcd, Single-Master, and Multi-Worker Installation**  
      In this setup, we have a single master node, which also runs a single-node etcd instance. Multiple worker nodes are connected to the master node.
    - **Single-Node etcd, Multi-Master, and Multi-Worker Installation**  
      In this setup, we have multiple master nodes, which work in an HA mode, but we have a single-node etcd instance. Multiple worker nodes are connected to the master nodes.
    - **Multi-Node etcd, Multi-Master, and Multi-Worker Installation**  
      In this mode, etcd is configured in a clustered mode, outside the Kubernetes cluster, and the nodes connect to it. The master nodes are all configured in an HA mode, connecting to multiple worker nodes. This is the most advanced and recommended production setup.

## Infrastructure for Kubernetes Installation

Once we decide on the installation type, we also need to make some infrastructure-related decisions, such as:

* + - Should we set up Kubernetes on bare metal, public cloud, or private cloud?
    - Which underlying system should we use? Should we choose RHEL, CoreOS, CentOS, or something else?
    - Which networking solution should we use?
    - And so on.

The Kubernetes documentation has details in regards to [choosing the right solution](https://kubernetes.io/docs/setup/pick-right-solution/). Next, we will take a closer look at these solutions.

**Localhost Installation**

There are a few localhost installation options available to deploy single- or multi-node Kubernetes clusters on our workstation/laptop:

* + - [Minikube](https://kubernetes.io/docs/getting-started-guides/minikube/)
    - [Ubuntu on LXD](https://kubernetes.io/docs/getting-started-guides/ubuntu/local/).

Minikube is the preferred and recommended way to create an all-in-one Kubernetes setup. We will be using it extensively in this course.

**On-Premise Installation**

Kubernetes can be installed on-premise on VMs and bare metal.

* + - **On-Premise VMs**  
      Kubernetes can be installed on VMs created via Vagrant, VMware vSphere, KVM, etc. There are different tools available to automate the installation, like [Ansible](https://www.ansible.com/" \t "_blank) or [kubeadm](https://github.com/kubernetes/kubeadm).
    - **On-Premise Bare Metal**Kubernetes can be installed on on-premise bare metal, on top of different operating systems, like RHEL, CoreOS, CentOS, Fedora, Ubuntu, etc. Most of the tools used to install VMs can be used with bare metal as well.

**Cloud Installation**

Kubernetes can be installed and managed on almost any cloud environment:

* + - **Hosted Solutions**With hosted solutions, any given software is completely managed by the provider. The user will just need to pay hosting and management charges. Some examples of vendors providing hosted solutions for Kubernetes are listed below:
      * + [Google Kubernetes Engine (GKE)](https://cloud.google.com/container-engine/)
        + [Azure Container Service (AKS)](https://azure.microsoft.com/en-us/services/container-service/)
        + [Amazon Elastic Container Service for Kubernetes (EKS) - Currently in Tech Preview](https://aws.amazon.com/eks/)
        + [OpenShift Dedicated](https://www.openshift.com/dedicated/)
        + [Platform9](https://platform9.com/support/kubernetes-at-the-command-line-up-and-running-with-kubectl/)
        + [IBM Cloud Container Service](https://console.ng.bluemix.net/docs/containers/container_index.html).
  + **Turnkey Cloud Solutions**For Kubernetes, we have some Turnkey Cloud Solutions, with which Kubernetes can be installed with just a few commands on an underlying IaaS platform, such as:
    - * + [Google Compute Engine](https://kubernetes.io/docs/getting-started-guides/gce/)
        + [Amazon AWS](https://kubernetes.io/docs/getting-started-guides/aws/)
        + [Microsoft Azure](https://kubernetes.io/docs/getting-started-guides/azure/)
        + [Tectonic by CoreOS](https://coreos.com/tectonic).
  + **Bare Metal**Kubernetes can be installed on bare metal provided by different cloud providers.

## Kubernetes Installation Tools/Resources

While discussing installation configuration and the underlying infrastructure, let's take a look at some useful tools/resources available:

* + - **kubeadm**  
      [kubeadm](https://github.com/kubernetes/kubeadm) is a first-class citizen on the Kubernetes ecosystem. It is a secure and recommended way to bootstrap the Kubernetes cluster. It has a set of building blocks to setup the cluster, but it is easily extendable to add more functionality. Please note that kubeadm does not support the provisioning of machines.
    - **KubeSpray**  
      With [KubeSpray](https://github.com/kubernetes-incubator/kubespray) (formerly known as Kargo), we can install Highly Available Kubernetes clusters on AWS, GCE, Azure, OpenStack, or bare metal. KubeSpray is based on Ansible, and is available on most Linux distributions. It is a [Kubernetes Incubator](https://github.com/kubernetes-incubator/kargo) project.
    - **Kops**  
      With [Kops](https://github.com/kubernetes/kops), we can create, destroy, upgrade, and maintain production-grade, highly-available Kubernetes clusters from the command line. It can provision the machines as well. Currently, AWS is officially supported. Support for GCE and VMware vSphere are in alpha stage, and other platforms are planned for the future.

If the existing solutions and tools do not fit your requirements, then [you can always install Kubernetes from scratch](https://kubernetes.io/docs/getting-started-guides/scratch/).

It is worth checking out the [*Kubernetes The Hard Way*](https://github.com/kelseyhightower/kubernetes-the-hard-way) GitHub project by [Kelsey Hightower](https://twitter.com/kelseyhightower), which shares the manual steps involved in bootstrapping a Kubernetes cluster.

## Introduction

As we mentioned in the previous chapter, [Minikube](https://github.com/kubernetes/minikube) is the easiest and most recommended way to run an all-in-one Kubernetes cluster locally. In this chapter, we will check out the requirements to install Minikube on our workstation, as well as the installation instructions to set it up on Linux, Mac, and Windows.



**Requirements for Running Minikube**

In most of the cases, Minikube runs inside a VM on Linux, Mac, or Windows. Therefore, we need to make sure that we have the supported hardware and the hypervisor to create VMs. Next, we outline the requirements to run Minikube on our workstation/laptop:

* + - [kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/)  
      **kubectl** is a binaryused  to access any Kubernetes cluster. Generally, it is installed before starting Minikube, but we can install it later, as well. If **kubectl** is not found while installing Minikube, we will get a warning message, which can be safely ignored (just remember that we will have to install **kubectl** later). We will explore **kubectl** in future chapters.
    - On Linux  
      [VirtualBox](https://www.virtualbox.org/wiki/Downloads) or [KVM](https://github.com/kubernetes/minikube/blob/master/docs/drivers.md#kvm-driver) hypervisors  
        
      **NOTE:** Minikube also supports a **--vm-driver=none** option that runs the Kubernetes components on the host and not in a VM. Docker is required to use this driver, but no hypervisor. If you use **--vm-driver=none**, be sure to specify a [bridge network](https://docs.docker.com/network/bridge/#configure-the-default-bridge-network) for Docker. Otherwise, it might change between network restarts, causing loss of connectivity to your cluster.
    - On macOS  
      [Hyperkit drive](https://github.com/kubernetes/minikube/blob/master/docs/drivers.md#hyperkit-driver)r, [xhyve driver](https://github.com/kubernetes/minikube/blob/master/docs/drivers.md" \l "xhyve-driver" \t "_blank), [VirtualBox](https://www.virtualbox.org/wiki/Downloads) or [VMware Fusion](http://www.vmware.com/products/fusion.html) hypervisors
    - On Windows  
      [VirtualBox](https://www.virtualbox.org/wiki/Downloads) or [Hyper-V](https://github.com/kubernetes/minikube/blob/master/docs/drivers.md#hyperV-driver) hypervisors
    - VT-x/AMD-v virtualization must be enabled in BIOS
    - Internet connection on first run.

In this chapter, we will use VirtualBox as hypervisor on all three operating systems - Linux, macOS, and Windows, to create the Minikube VM.