**Introduction**

In this chapter, we will explain what **Kubernetes** is, its features, and the reasons why you should use it. We will explore the evolution of Kubernetes from **Borg**, Google's very own distributed workload manager.

We will also learn about the **Cloud Native Computing Foundation** **(CNCF)**, which currently hosts the Kubernetes project, along with other popular cloud-native projects, such as Prometheus, Fluentd, cri-o, containerd, Helm, Envoy, and Contour, just to name a few.

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 the role of the Cloud Native Computing Foundation.

## What Is Kubernetes?

* According to the [Kubernetes](https://kubernetes.io/) website,
* *"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 pilot on a ship of containers.
* Kubernetes is also referred to as **k8s** (pronounced Kate's), as there are 8 characters between *k* and *s*.
* Kubernetes is highly inspired by the Google Borg system, a container and workload orchestrator for its global operations for more than a decade. 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).
* New Kubernetes versions are released in 3 months cycles. The current stable version is 1.19 (as of August 2020).

## 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 has been Google's secret, running its worldwide containerized workloads in production. Services we use from Google, such as Gmail, Drive, Maps, Docs, 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 learned 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 bin packing**  
      Kubernetes automatically schedules containers based on resource needs and constraints, to maximize utilization without sacrificing availability.
    - **Self-healing**  
      Kubernetes automatically replaces and reschedules containers from failed nodes. It kills and restarts containers unresponsive to health checks, based on existing rules/policy. It also prevents traffic from being routed to unresponsive containers.
    - **Horizontal scaling**  
      With Kubernetes applications are scaled manually or automatically based on CPU or custom metrics utilization.
    - **Service discovery and Load balancing**  
      Containers receive their own IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set.

## Kubernetes Features II

Some other fully supported Kubernetes features are:

* + - **Automated rollouts and rollbacks**  
      Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, constantly monitoring the application's health to prevent any downtime.
    - **Secret and configuration management**  
      Kubernetes manages sensitive data and configuration details for an application separately from the container image, in order to avoid a re-build of the respective image. Secrets consist of sensitive/confidential information passed to the application without revealing the sensitive content to the stack configuration, like on GitHub.
    - **Storage orchestration**  
      Kubernetes automatically mounts software-defined storage (SDS) solutions to containers from local storage, external cloud providers, distributed storage, or network storage systems.
    - **Batch execution**  
      Kubernetes supports batch execution, long-running jobs, and replaces failed containers.

There are many additional features currently in alpha or 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 only as of the Kubernetes 1.8 release.

## Why Use Kubernetes?

In addition to its fully-supported features, Kubernetes is also portable and extensible. It can be deployed in many environments such as local or remote Virtual Machines, bare metal, or in public/private/hybrid/multi-cloud setups. It supports and it is supported by many 3rd party open source tools which enhance Kubernetes' capabilities and provide a feature-rich experience to its users.

Kubernetes' architecture is modular and pluggable. Not only that it orchestrates modular, decoupled microservices type applications, but also its architecture follows decoupled microservices patterns. Kubernetes' functionality can be extended by writing custom resources, operators, custom APIs, scheduling rules or plugins.

For a successful open source project, the community is as important as having great code. Kubernetes is supported by a thriving community across the world. It has more than 2,800 contributors, who, over time, have pushed over 94,000 commits. There are meet-up groups in different cities and countries which meet regularly to discuss Kubernetes and its ecosystem. There are Special Interest Groups (SIGs), which focus on special topics, such as scaling, bare metal, networking, etc. We will learn more about them in our last chapter, Kubernetes Communities.

## Kubernetes Users

With just a few years since its debut, many enterprises of various sizes run their workloads using Kubernetes. It is a solution for workload management in banking, education, finance and investments, gaming, information technology, media and streaming, online retail, ridesharing, telecommunications, and many other industries. There are numerous user [case studies](https://kubernetes.io/case-studies/) and success stories on the Kubernetes website:

* + - [BlaBlaCar](https://kubernetes.io/case-studies/blablacar/)
    - [BlackRock](https://kubernetes.io/case-studies/blackrock/)
    - [Box](https://kubernetes.io/case-studies/box/)
    - [eBay](https://www.nextplatform.com/2015/11/12/inside-ebays-shift-to-kubernetes-and-containers-atop-openstack/)
    - [Haufe Group](https://kubernetes.io/case-studies/haufegroup/)
    - [Huawei](https://kubernetes.io/case-studies/huawei/)
    - [IBM](https://kubernetes.io/case-studies/ibm/)
    - [ING](https://kubernetes.io/case-studies/ing/)
    - [Nokia](https://kubernetes.io/case-studies/nokia/)
    - [Pearson](https://kubernetes.io/case-studies/pearson/)
    - [Wikimedia](https://kubernetes.io/case-studies/wikimedia/)
    - 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 multitude 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. Projects within CNCF are categorized based on achieved status: Sandbox, Incubating, and Graduated. At the time of this writing, a dozen projects had reached Graduated status with many more Incubating and in the Sandbox.

Graduated projects:

* + - [Kubernetes](https://kubernetes.io/) for container orchestration
    - [Prometheus](https://prometheus.io/) for monitoring
    - [Envoy](https://github.com/envoyproxy/envoy) for service mesh
    - [CoreDNS](https://coredns.io/) for service discovery
    - [containerd](http://containerd.io/) for container runtime
    - [Fluentd](http://www.fluentd.org/) for logging
    - [Harbor](https://goharbor.io/) for registry
    - [Helm](https://www.helm.sh/) for package management
    - [Vitess](http://vitess.io/) for cloud-native storage
    - [Jaeger](https://github.com/jaegertracing/jaeger) for distributed tracing
    - [TUF](https://github.com/theupdateframework/specification) for software updates
    - [TiKV](https://tikv.org/) for key/value store

Incubating projects:

* + - [CRI-O](https://cri-o.io/) for container runtime
    - [Linkerd](https://linkerd.io/) for service mesh
    - [Contour](https://projectcontour.io/) for ingress
    - [etcd](https://github.com/etcd-io) for key/value store
    - [gRPC](http://www.grpc.io/) for remote procedure call (RPC)
    - [CNI](https://github.com/containernetworking/cni) for networking API
    - [Rook](https://github.com/rook/rook) for cloud-native storage
    - [Notary](https://github.com/theupdateframework/notary) for security
    - [NATS](https://nats.io/) for messaging
    - [OpenTracing](http://opentracing.io/) for distributed tracing
    - [Open Policy Agent](https://www.openpolicyagent.org/) for policy
    - And many [more](https://www.cncf.io/projects/).

There are many projects in the CNCF [Sandbox](https://www.cncf.io/sandbox-projects/) geared towards metrics, monitoring, identity, scripting, serverless, nodeless, edge, expecting to achieve Incubating and possibly Graduated status. While many active projects are preparing for takeoff, others are being [archived](https://www.cncf.io/archived-projects/) once they become less active. The first archived project is the [rkt](https://github.com/rkt/rkt" \t "_blank) container runtime.

The projects under CNCF cover the entire lifecycle of a cloud-native application, from its execution using container runtimes, to its monitoring and logging. This is very important to meet the goals of CNCF.

## 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 vendor code
    - Offers legal guidance on patent and copyright issues
    - Creates open source learning [curriculum](https://github.com/cncf/curriculum), [training](https://www.cncf.io/certification/training/), and certification for both Kubernetes [administrators](https://www.cncf.io/certification/CKA/) (CKA) and [application developers](https://www.cncf.io/certification/ckad/) (CKAD)
    - Manages a software conformance [working group](https://lists.cncf.io/g/cncf-k8s-conformance)
    - Actively markets Kubernetes
    - Supports ad hoc activities
    - Sponsors conferences and meetup events.

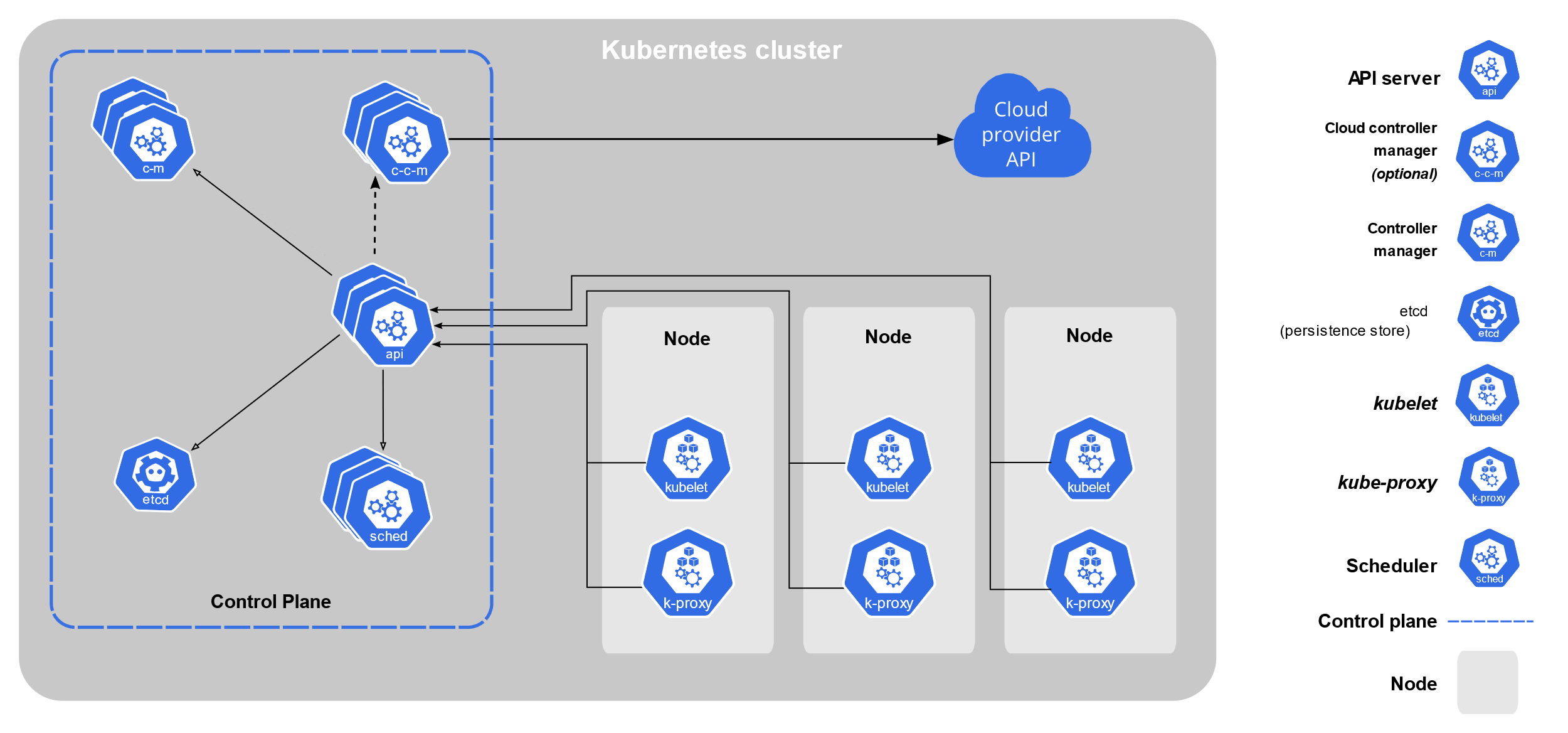
## Introduction

In this chapter, we will explore the **Kubernetes architecture**, the components of its **control plane**, the **master** and **worker** **nodes**, the cluster state management with **etcd**and the network setup requirements. We will also learn about the **Container Network Interface (CNI)**, as Kubernetes' network specification.

## Kubernetes Architecture

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

* + One or more **master nodes**, part of the **control plane**
  + One or more **worker nodes**.



**Components of Kubernetes Architecture**

## Master Node Overview

The **master node** provides a running environment for the **control plane** responsible for managing the state of a Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management. In order to communicate with the Kubernetes cluster, users send requests to the control plane via a Command Line Interface (CLI) tool, a Web User-Interface (Web UI) Dashboard, or Application Programming Interface (API).

It is important to keep the control plane running at all costs. Losing the control plane may introduce downtime, causing service disruption to clients, with possible loss of business. To ensure the control plane's fault tolerance, master node replicas can be added to the cluster, configured in High-Availability (HA) mode. While only one of the master nodes is dedicated to actively manage the cluster, the control plane components stay in sync across the master node replicas. This type of configuration adds resiliency to the cluster's control plane, should the active master node fail.

To persist the Kubernetes cluster's state, all cluster configuration data is saved to [etcd](https://etcd.io/" \t "_blank). **etcd** is a distributed key-value store which only holds cluster state related data, no client workload data. etcd may be configured on the master node ([stacked](https://kubernetes.io/docs/setup/independent/ha-topology/#stacked-etcd-topology) topology), or on its dedicated host ([external](https://kubernetes.io/docs/setup/independent/ha-topology/#external-etcd-topology) topology) to help reduce the chances of data store loss by decoupling it from the other control plane agents.

With stacked etcd topology, HA master node replicas ensure the etcd data store's resiliency as well. However, that is not the case with external etcd topology, where the etcd hosts have to be separately replicated for HA, a configuration that introduces the need for additional hardware.

**Master Node Components**

A master node runs the following control plane components:

* + - API Server
    - Scheduler
    - Controller Managers
    - Data Store.

In addition, the master node runs:

* + - Container Runtime
    - Node Agent
    - Proxy.

## Master Node Components: API Server

All the administrative tasks are coordinated by the **kube-apiserver**, a central control plane component running on the master node. The API Server intercepts RESTful calls from users, operators and external agents, then validates and processes them. During processing the API Server reads the Kubernetes cluster's current state from the etcd data store, and after a call's execution, the resulting state of the Kubernetes cluster is saved in the distributed key-value data store for persistence. The API Server is the only master plane component to talk to the etcd data store, both to read from and to save Kubernetes cluster state information - acting as a middle interface for any other control plane agent inquiring about the cluster's state.

The API Server is highly configurable and customizable. It can scale horizontally, but it also supports the addition of custom secondary API Servers, a configuration that transforms the primary API Server into a proxy to all secondary, custom API Servers and routes all incoming RESTful calls to them based on custom defined rules.

## Master Node Components: Scheduler

The role of the **kube-scheduler** is to assign new workload objects, such as pods, to nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new object's requirements. The scheduler obtains from the etcd data store, via the API Server, resource usage data for each worker node in the cluster. The scheduler also receives from the API Server the new object's requirements which are part of its configuration data. Requirements may include constraints that users and operators set, such as scheduling work on a node labeled with **disk==ssd** key/value pair. The scheduler also takes into account Quality of Service (QoS) requirements, data locality, affinity, anti-affinity, taints, toleration, cluster topology, etc. Once all the cluster data is available, the scheduling algorithm filters the nodes with predicates to isolate the possible node candidates which then are scored with priorities in order to select the one node that satisfies all the requirements for hosting the new workload. The outcome of the decision process is communicated back to the API Server, which then delegates the workload deployment with other control plane agents.

The scheduler is highly configurable and customizable through scheduling policies, plugins, and profiles. Additional custom schedulers are also supported, then the object's configuration data should include the name of the custom scheduler expected to make the scheduling decision for that particular object; if no such data is included, the default scheduler is selected instead.

A scheduler is extremely important and complex in a multi-node Kubernetes cluster. In a single-node Kubernetes cluster, such as the one explored later in this course, the scheduler's job is quite simple.

## Master Node Components: Controller Managers

The **controller managers** are control plane components on the master node running controllers to regulate the state of the Kubernetes cluster. Controllers are watch-loops continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from etcd data store via the API server). In case of a mismatch corrective action is taken in the cluster until its current state matches the desired state.

The **kube-controller-manager** runs controllers responsible to act when nodes become unavailable, to ensure pod counts are as expected, to create endpoints, service accounts, and API access tokens.

The **cloud-controller-manager** runs controllers responsible to interact with the underlying infrastructure of a cloud provider when nodes become unavailable, to manage storage volumes when provided by a cloud service, and to manage load balancing and routing.

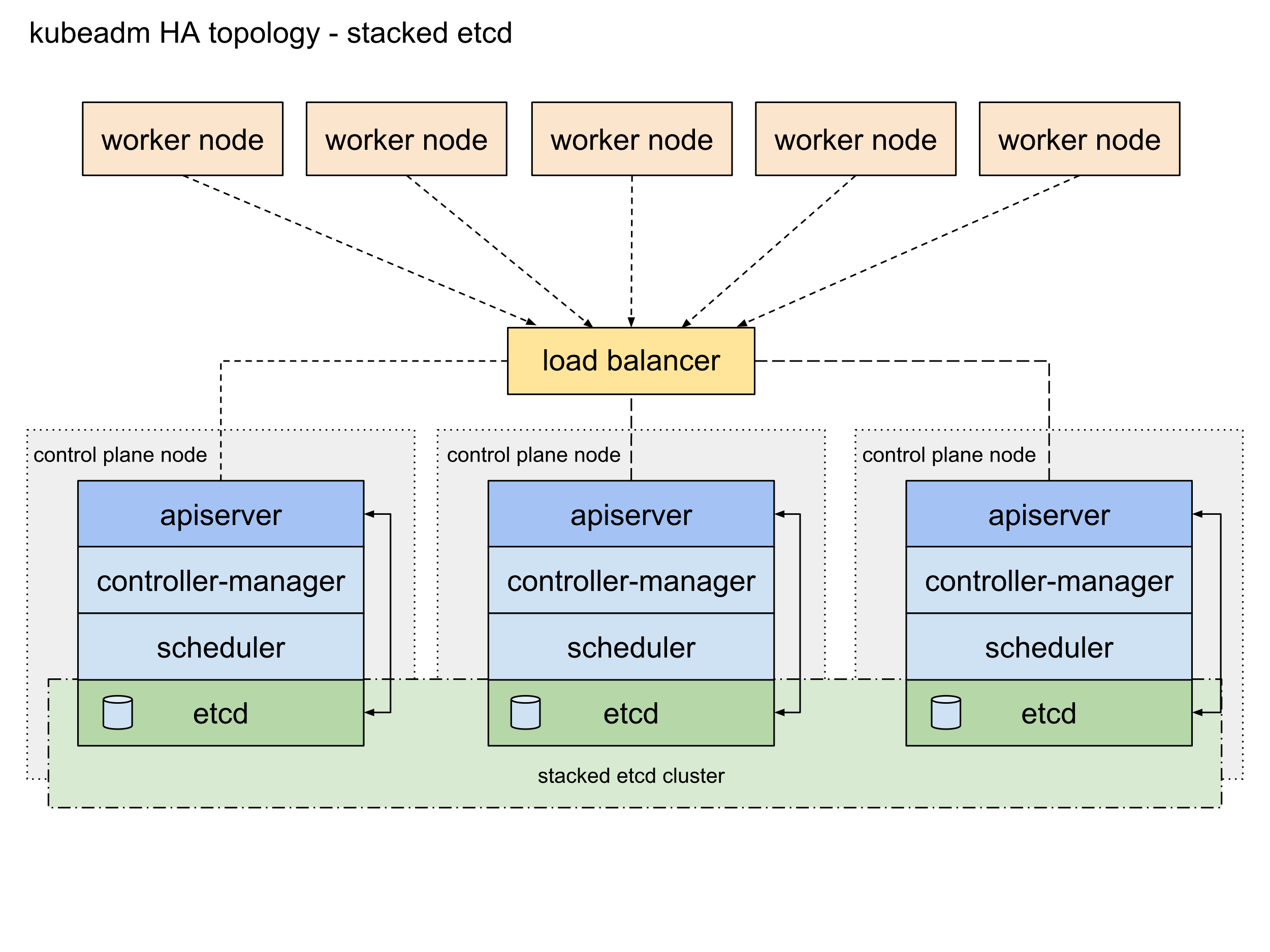
## Master Node Components: Data Store

**etcd** is a strongly consistent, distributed key-value **data store** used to persist a Kubernetes cluster's state. New data is written to the data store only by appending to it, data is never replaced in the data store. Obsolete data is compacted periodically to minimize the size of the data store.

Out of all the control plane components, only the API Server is able to communicate with the etcd data store.

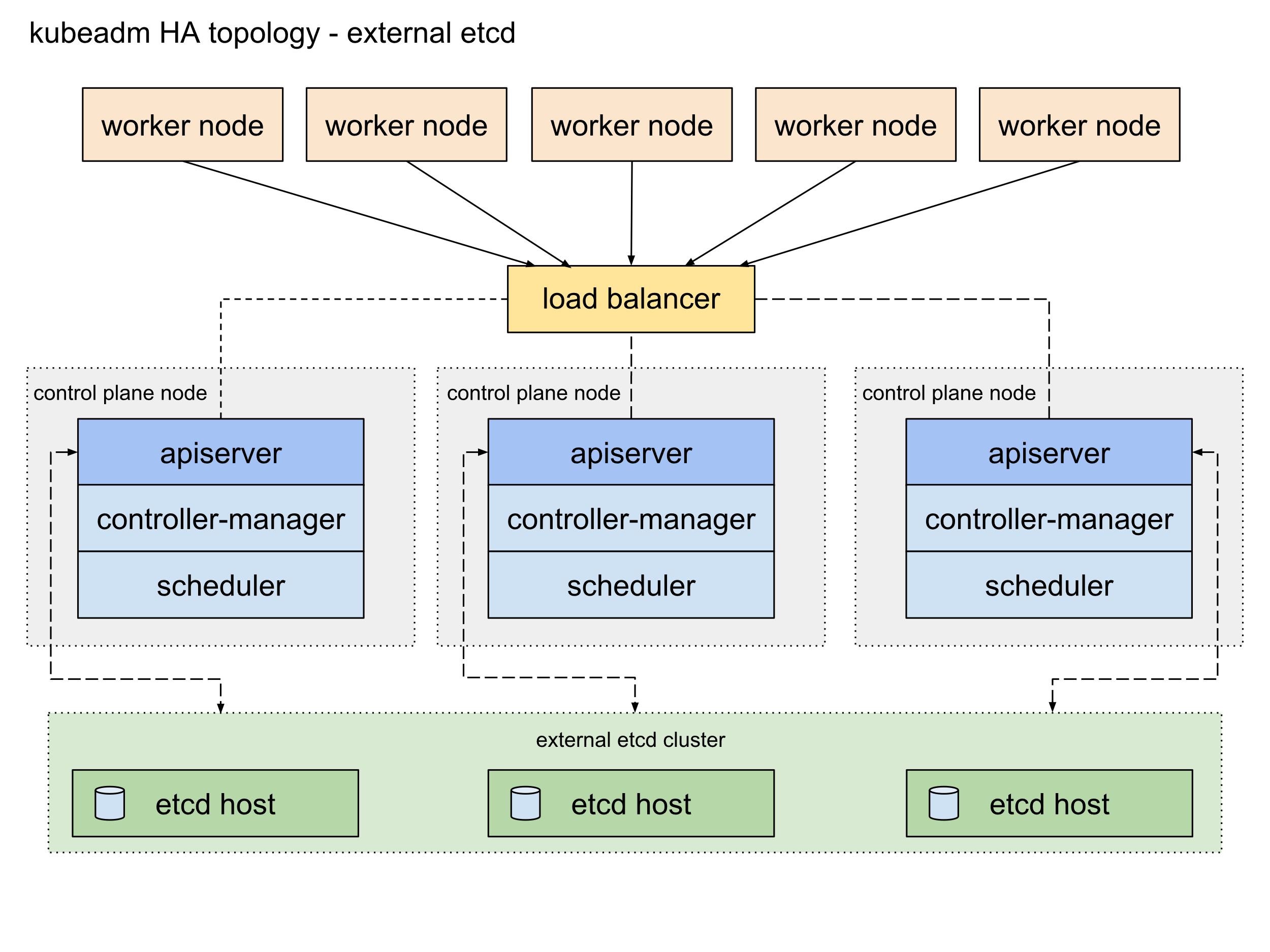
etcd's CLI management tool - **etcdctl**, provides backup, snapshot, and restore capabilities which come in handy especially for a single etcd instance Kubernetes cluster - common in Development and learning environments. However, in Stage and Production environments, it is extremely important to replicate the data stores in HA mode, for cluster configuration data resiliency.

Some Kubernetes cluster bootstrapping tools, such as **kubeadm**, by default, provision stacked etcd master nodes, where the data store runs alongside and shares resources with the other control plane components on the same master node.



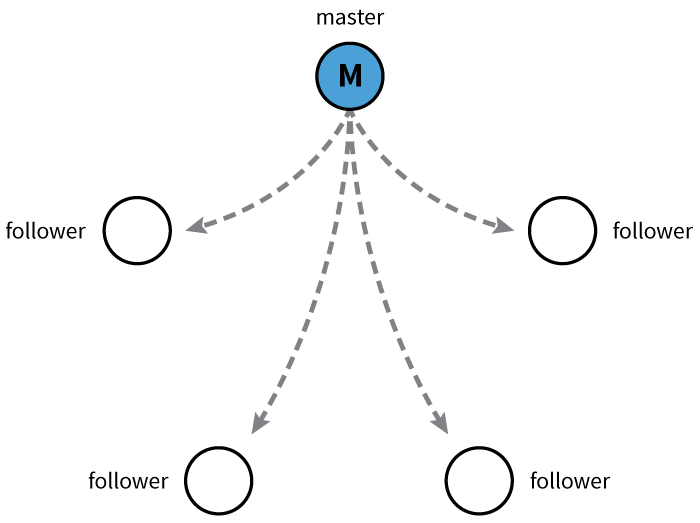
**Stacked etcd topology**

For data store isolation from the control plane components, the bootstrapping process can be configured for an external etcd topology, where the data store is provisioned on a dedicated separate host, thus reducing the chances of an etcd failure.



**External etcd topology**

Both stacked and external etcd configurations support HA configurations. etcd is based on the [Raft Consensus Algorithm](https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14) which 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. etcd gracefully handles master elections and can tolerate node failure, including master node failures. 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.

## Worker Node Overview

A **worker node** provides a running environment for client applications. Though containerized microservices, these applications are encapsulated in Pods, controlled by the cluster control plane agents running on the master node. Pods are scheduled on worker nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world. A Pod is the smallest scheduling unit in Kubernetes. It is a logical collection of one or more containers scheduled together, and the collection can be started, stopped, or rescheduled as a single unit of work.

Also, in a multi-worker Kubernetes cluster, the network traffic between client users and the containerized applications deployed in Pods is handled directly by the worker nodes, and is not routed through the master node.

## Worker Node Components

A worker node has the following components:

* + - Container Runtime
    - Node Agent - kubelet
    - Proxy - kube-proxy
    - Addons for DNS, Dashboard user interface, cluster-level monitoring and logging.

## Worker Node Components: Container Runtime

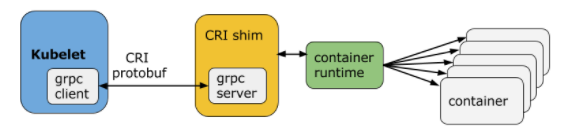
Although Kubernetes is described as a "container orchestration engine", it does not have the capability to directly handle containers. In order to manage a container's lifecycle, Kubernetes requires a **container runtime** on the node where a Pod and its containers are to be scheduled. Kubernetes supports many container runtimes:

* + - [Docker](https://www.docker.com/) - although a container platform which uses **containerd** as a container runtime, it is the most popular container runtime used with Kubernetes
    - [CRI-O](https://cri-o.io/) - a lightweight container runtime for Kubernetes, it also supports Docker image registries
    - [containerd](https://containerd.io/)- a simple and portable container runtime providing robustness
    - [frakti](https://github.com/kubernetes/frakti#frakti) - a hypervisor-based container runtime for Kubernetes

## Worker Node Components: Node Agent - kubelet

The **kubelet** is an agent running on each node and communicates with the control plane components from the master node. It receives Pod definitions, primarily from the API Server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health and resources of Pods running containers.

The kubelet connects to container runtimes though a plugin based interface - the [Container Runtime Interface](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md) (CRI). The CRI consists of protocol buffers, gRPC API, libraries, and additional specifications and tools that are currently under development. In order to connect to interchangeable container runtimes, kubelet uses a **shim** application which provides a clear abstraction layer between kubelet and the container runtime.



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

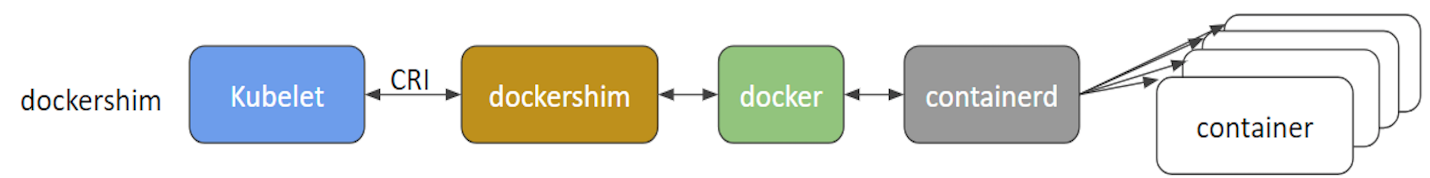
As shown above, the kubelet acting as grpc client connects to the CRI shim acting as grpc server to perform container and image operations. The CRI implements two services: **ImageService** and **RuntimeService**. The **ImageService** is responsible for all the image-related operations, while the **RuntimeService**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

Container runtimes used to be hard-coded into kubelet, but since the CRI was introduced, Kubernetes has become more flexible to use different container runtimes without the need to recompile. Any container runtime that implements the CRI can be used by Kubernetes to manage Pods, containers, and container images.

## Worker Node Components: kubelet - CRI shims

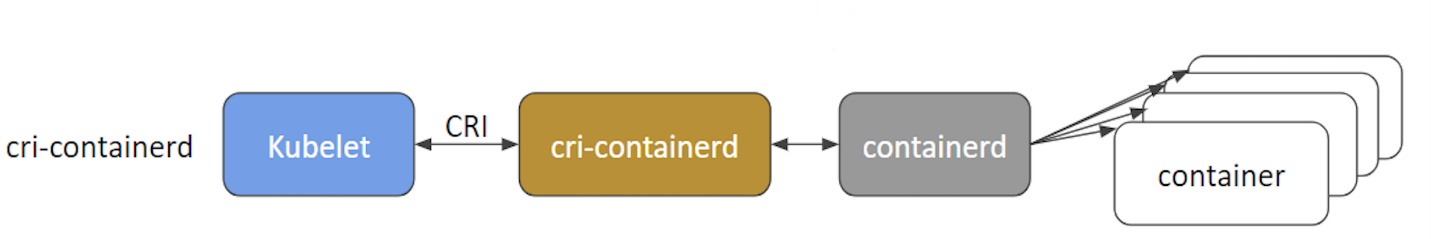
Shims are CRI implementations, or interfaces, specific to each container runtime supported by Kubernetes. Below we present some examples of CRI shims:

* + - **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 containerd to create and manage containers:



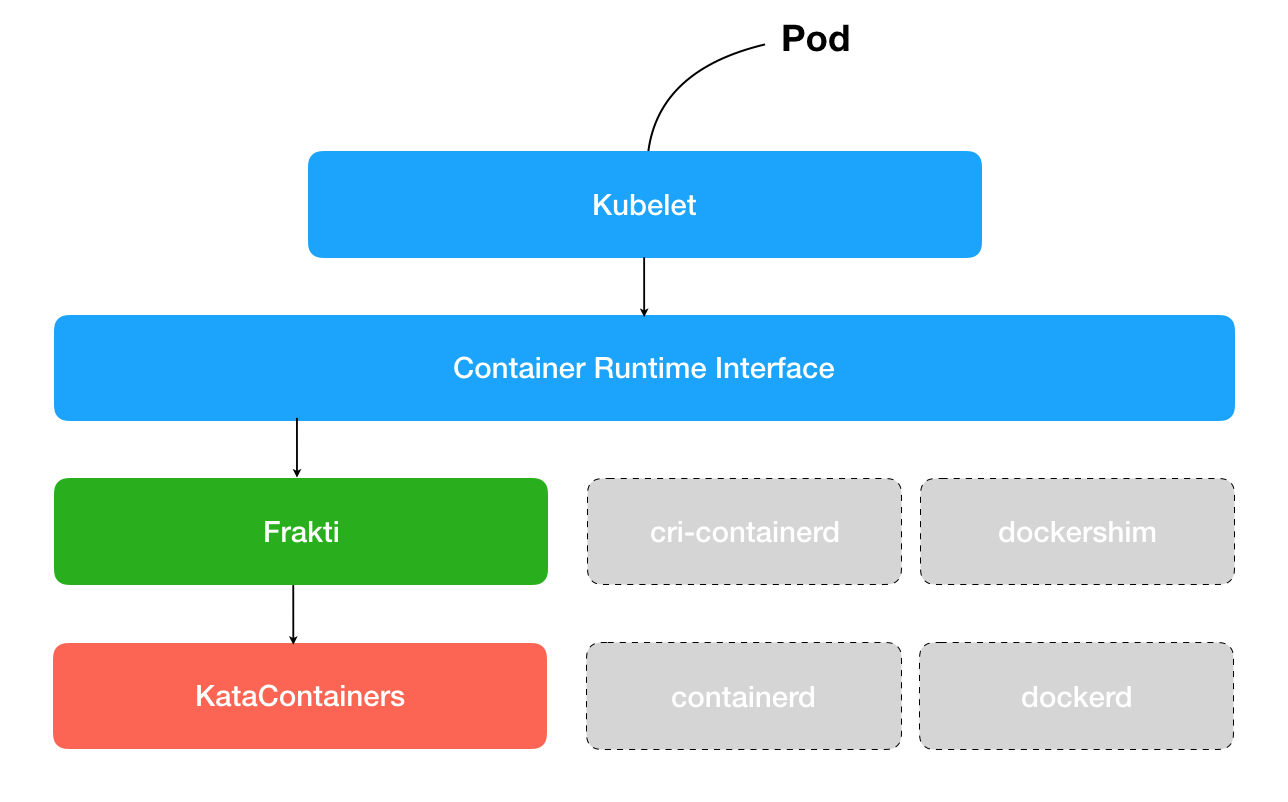
**cri-containerd**(Retrieved from [blog.kubernetes.io](http://blog.kubernetes.io/2017/11/containerd-container-runtime-options-kubernetes.html))

* + - **CRI-O**  
      CRI-O enables the use of any Open Container Initiative (OCI) compatible runtime 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**(Retrieved from [cri-o.io](http://cri-o.io/))

* + - **frakti**   
      frakti enables CRI implementation through hardware virtualization, aimed to achieve a higher level of security and isolation than the traditional Linux OS level containers based on cgroups and namespaces. The frakti CRI shim is aimed at enabling kubelet to interact with [Kata Containers](https://katacontainers.io/):



**frakti**   
(Retrieved from [github.com/kubernetes/frakti](https://github.com/kubernetes/frakti#frakti))

## Worker Node Components: Proxy - kube-proxy

The **kube-proxy** is the network agent which runs on each node responsible for dynamic updates and maintenance of all networking rules on the node. It abstracts the details of Pods networking and forwards connection requests to Pods.

The kube-proxy is responsible for TCP, UDP, and SCTP stream forwarding or round-robin forwarding across a set of Pod backends, and it implements forwarding rules defined by users through Service API objects.

## Worker Node Components: Addons

**Addons** are cluster features and functionality not yet available in Kubernetes, therefore implemented through 3rd-party pods and services.

* + - **DNS** - cluster DNS is a DNS server required to assign DNS records to Kubernetes objects and resources
    - **Dashboard** - a general purposed web-based user interface for cluster management
    - **Monitoring** - collects cluster-level container metrics and saves them to a central data store
    - **Logging** - collects cluster-level container logs and saves them to a central log store for analysis.

## Networking Challenges

Decoupled microservices based applications rely heavily on networking in order to mimic the tight-coupling once available in the monolithic era. Networking, in general, is not the easiest to understand and implement. Kubernetes is no exception - as a containerized microservices orchestrator it needs to address a few distinct networking challenges:

* + - Container-to-container communication inside Pods
    - Pod-to-Pod communication on the same node and across cluster nodes
    - Pod-to-Service communication within the same namespace and across cluster namespaces
    - External-to-Service communication for clients to access applications in a cluster.

All these networking challenges must be addressed before deploying a Kubernetes cluster.

## Container-to-Container Communication Inside Pods

Making use of the underlying host operating system's kernel virtualization features, a container runtime creates an isolated network space for each container it starts. On Linux, this isolated network space is referred to as a **network namespace**. A network namespace can be shared across containers, or with the host operating system.

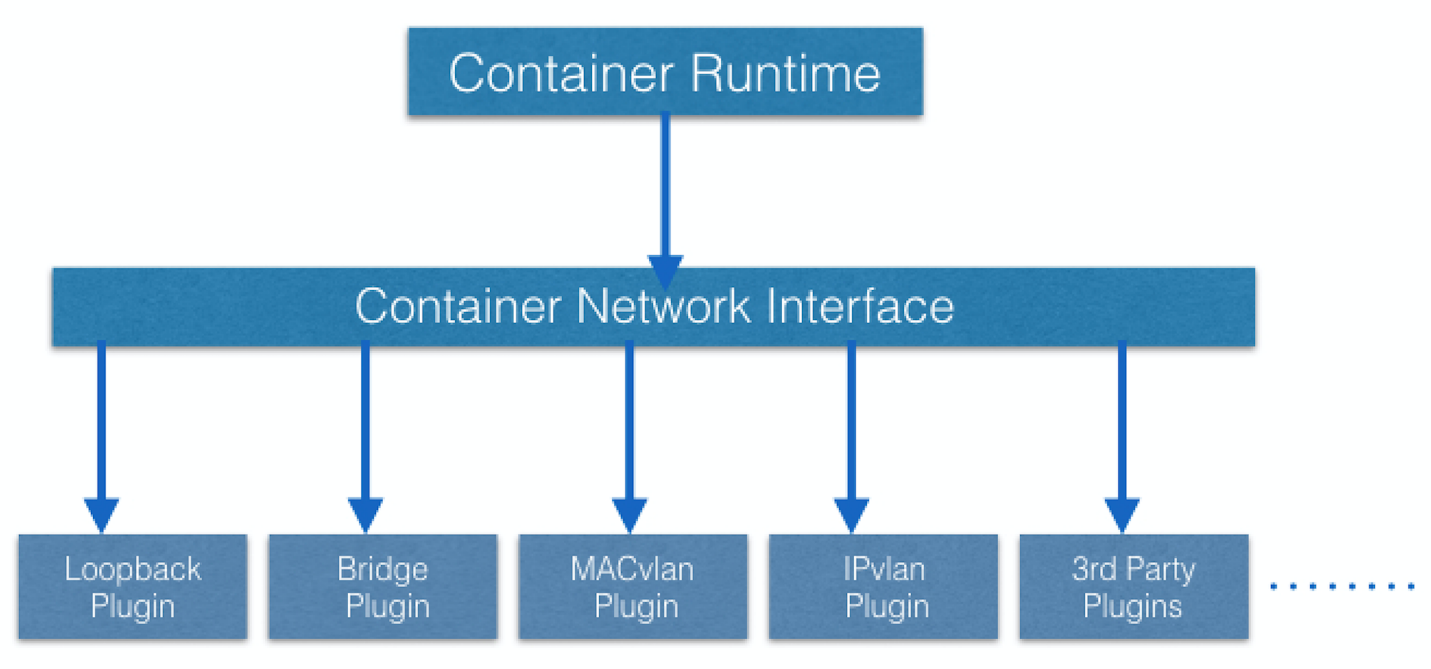
When a Pod is started, a special **Pause** container is initialized by the Container Runtime for the sole purpose to create a network namespace for the Pod. All additional containers, created through user requests, running inside the Pod will share the Pause container's network namespace so that they can all talk to each other via localhost.

## Pod-to-Pod Communication Across Nodes

In a Kubernetes cluster Pods are scheduled on nodes in a nearly unpredictable fashion. Regardless of their host node, Pods are expected to be able to communicate with all other Pods in the cluster, all this without the implementation of Network Address Translation (NAT). This is a fundamental requirement of any networking implementation in Kubernetes.

The Kubernetes network model aims to reduce complexity, and it treats Pods as VMs on a network, where each VM is equipped with a network interface - thus each Pod receiving a unique IP address. This model is called "**IP-per-Pod**" and ensures Pod-to-Pod communication, just as VMs are able to communicate with each other on the same network.

Let's not forget about containers though. They share the Pod's network namespace and must coordinate ports assignment inside the Pod just as applications would on a VM, all while being able to communicate with each other on **localhost** - inside the Pod. However, containers are integrated with the overall Kubernetes networking model through the use of the [Container Network Interface](https://github.com/containernetworking/cni) (CNI) supported by [CNI plugins](https://github.com/containernetworking/cni" \l "3rd-party-plugins" \t "_blank). CNI is a set of a specification and libraries which allow plugins to configure the networking for containers. While there are a few [core plugins](https://github.com/containernetworking/plugins#plugins), most CNI plugins are 3rd-party Software Defined Networking (SDN) solutions implementing the Kubernetes networking model. In addition to addressing the fundamental requirement of the networking model, some networking solutions offer support for Network Policies. [Flannel](https://github.com/coreos/flannel/), [Weave](https://www.weave.works/oss/net/), [Calico](https://www.projectcalico.org/) are only a few of the SDN solutions available for Kubernetes clusters.



**Container Network Interface (CNI) Core Plugins**

The container runtime offloads the IP assignment to CNI, which connects to the underlying configured plugin, such as 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.

For more details, you can explore the [Kubernetes documentation](https://kubernetes.io/docs/concepts/cluster-administration/networking/).

## Pod-to-External World Communication

A successfully deployed containerized application running in Pods inside a Kubernetes cluster may require accessibility from the outside world. Kubernetes enables external accessibility through **Services**, complex encapsulations of network routing rule definitions stored in **iptables** on cluster nodes and implemented by **kube-proxy** agents. By exposing services to the external world with the aid of **kube-proxy**, applications become accessible from outside the cluster over a virtual IP address.

## Introduction

In this chapter we will explore Kubernetes cluster deployment considerations. First, we will learn about Kubernetes cluster configuration options, followed by infrastructure requirements and installation tools specific to various cluster deployment models.

## Kubernetes Configuration

Kubernetes can be installed using different cluster configurations. The major installation types are described below:

* + - **All-in-One Single-Node Installation**  
      In this setup, all the master and worker components are installed and running on a single-node. While it is useful for learning, development, and testing, it should not be used in production. Minikube is an installation tool originally aimed at single-node cluster installations, and we are going to explore it in future chapters.
    - **Single-Master and Multi-Worker Installation**  
      In this setup, we have a single-master node running a stacked etcd instance. Multiple worker nodes can be managed by the master node.
    - **Single-Master with Single-Node etcd, and Multi-Worker Installation**  
      In this setup, we have a single-master node with an external etcd instance. Multiple worker nodes can be managed by the master node.
    - **Multi-Master and Multi-Worker Installation**  
      In this setup, we have multiple master nodes configured for High-Availability (HA), with each master node running a stacked etcd instance. The etcd instances are also configured in an HA etcd cluster and, multiple worker nodes can be managed by the HA masters.
    - **Multi-Master with Multi-Node etcd, and Multi-Worker Installation**  
      In this setup, we have multiple master nodes configured in HA mode, with each master node paired with an external etcd instance. The external etcd instances are also configured in an HA etcd cluster, and multiple worker nodes can be managed by the HA masters. This is the most advanced cluster configuration recommended for production environments.

As the Kubernetes cluster's complexity grows, so does its hardware and resources requirements. While we can deploy Kubernetes on a single host for learning, development, and possibly testing purposes, the community recommends multi-host environments that support High-Availability control plane setups and multiple worker nodes for client workload.

## Infrastructure for Kubernetes Installation

Once we decide on the installation type, we need to decide on the infrastructure. Infrastructure related decisions are typically guided by the desired environment type, either learning or production environment. For infrastructure, we need to decide on the following:

* + - Should we set up Kubernetes on bare metal, public cloud, private, or hybrid cloud?
    - Which underlying OS should we use? Should we choose a Linux distribution such as RHEL, CentOS, CoreOS, or Windows?
    - Which networking solution should we use?

Explore the [Kubernetes documentation](https://kubernetes.io/docs/setup/) for details on choosing the right solution.

## Localhost Installation

There are a variety of installation tools allowing us to deploy single- or multi-node Kubernetes clusters on our workstations. While not an exhaustive list, below we enumerate a few popular ones:

* + - [Minikube](https://minikube.sigs.k8s.io/docs/) - single-node local Kubernetes cluster, recommended for a learning environment deployed on a single host.
    - [Kind](https://kind.sigs.k8s.io/docs/) - multi-node Kubernetes cluster deployed in Docker containers acting as Kubernetes nodes, recommended for a learning environment.
    - [Docker Desktop](https://www.docker.com/products/docker-desktop) - including a local Kubernetes cluster for Docker users.
    - [MicroK8s](https://microk8s.io/) - local and cloud Kubernetes cluster, from Canonical.
    - [K3S](https://k3s.io/) - lightweight Kubernetes cluster for local, cloud, edge, IoT deployments, from Rancher.

Minikube is the easiest and most preferred method to create an all-in-one Kubernetes setup locally. We will be using it extensively in this course to manage a single-node cluster. A newer experimental Minikube feature is the multi-node cluster support, which will not be explored at this time.

## 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, or another Configuration Management (CM) tool in conjunction with a hypervisor software. There are different tools available to automate the installation, such as [Ansible](https://kubernetes.io/blog/2019/03/15/kubernetes-setup-using-ansible-and-vagrant/) or [kubeadm](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/" \t "_blank).
    - **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 Kubernetes on VMs can be used with bare metal installations as well.

## Cloud Installation

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

* + - **Hosted Solutions**With Hosted Solutions, any given software is completely managed by the provider, while the user pays hosting and management charges. Popular vendors providing hosted solutions for Kubernetes are (listed in alphabetical order):
      * + [Alibaba Cloud Container Service for Kubernetes](https://www.alibabacloud.com/product/kubernetes) (ACK)
        + [Amazon Elastic Kubernetes Service](https://aws.amazon.com/eks/) (EKS)
        + [Azure Kubernetes Service](https://azure.microsoft.com/en-us/services/kubernetes-service/) (AKS)
        + [DigitalOcean Kubernetes](https://www.digitalocean.com/products/kubernetes/)
        + [Google Kubernetes Engine](https://cloud.google.com/kubernetes-engine/) (GKE)
        + [IBM Cloud Kubernetes Service](https://www.ibm.com/cloud/container-service/)
        + [Oracle Cloud Container Engine for Kubernetes](https://www.oracle.com/cloud/compute/container-engine-kubernetes.html) (OKE)
        + [Platform9 Managed Kubernetes](https://platform9.com/managed-kubernetes/) (PMK)
        + [Red Hat OpenShift](https://www.redhat.com/en/technologies/cloud-computing/openshift)
        + [VMware Tanzu Kubernetes](https://tanzu.vmware.com/kubernetes-grid).
    - **Turnkey Cloud Solutions**Below are only a few of the [Turnkey Cloud Solutions](https://kubernetes.io/docs/setup/production-environment/turnkey-solutions/) (listed in alphabetical order), to install Kubernetes on an underlying IaaS platform, such as:
      * + Alibaba Cloud
        + Amazon AWS (AWS EC2)
        + Google Compute Engine (GCE)
        + IBM Cloud Private
        + Microsoft Azure (AKS).
    - **Turnkey On-Premise Solutions**The On-Premise Solutions install Kubernetes on secure internal private clouds:
      * + [GKE On-Prem](https://cloud.google.com/anthos/gke/docs/on-prem) part of Google Cloud [Anthos](https://cloud.google.com/anthos)
        + [IBM Private Cloud](https://www.ibm.com/cloud/learn/introduction-to-private-cloud)
        + [OpenShift Container Platform](https://www.openshift.com/products/container-platform) by Red Hat.

## Kubernetes Installation Tools/Resources

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

* + - **kubeadm**  
      [kubeadm](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/) is a first-class citizen on the Kubernetes ecosystem. It is a secure and recommended method to bootstrap a multi-node production ready Highly Available Kubernetes cluster, on-prem or in the cloud. Kubeadm can also bootstrap a single-node cluster for learning. It has a set of building blocks to setup the cluster, but it is easily extendable to add more features. Please note that [kubeadm](https://kubernetes.io/docs/reference/setup-tools/kubeadm/" \t "_blank) does not support the provisioning of hosts.



**kubeadm**

* + - **kubespray**  
      With [kubespray](https://kubernetes.io/docs/setup/production-environment/tools/kubespray/" \t "_blank) (formerly known as kargo), we can install Highly Available production ready Kubernetes clusters on AWS, GCE, Azure, OpenStack, vSphere, or bare metal. Kubespray is based on Ansible, and is available on most Linux distributions. It is a [Kubernetes Incubator](https://github.com/kubernetes-sigs/kubespray) project.
    - **kops**  
      With [kops](https://kubernetes.io/docs/setup/production-environment/tools/kops/), we can create, 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 OpenStack is in beta, VMware vSphere is in alpha support, and other platforms are planned for the future. Explore the [kops project](https://github.com/kubernetes/kops) for more details.



**kops**

* + - **kube-aws**  
      With [kube-aws](https://github.com/kubernetes-incubator/kube-aws" \t "_blank) we can create, upgrade and destroy Kubernetes clusters on AWS from the command line. Kube-aws is also a Kubernetes Incubator project.

In addition, for a manual installation approach, the [*Kubernetes The Hard Way*](https://github.com/kelseyhightower/kubernetes-the-hard-way) GitHub project by [Kelsey Hightower](https://twitter.com/kelseyhightower) is an extremely helpful installation guide and resource. The project aims to teach all the detailed steps involved in the bootstrapping of a Kubernetes cluster.

## Kubernetes on Windows

The Windows operating system plays a key role in running and managing enterprise applications and services. With that in mind, the Kubernetes community worked very hard to bring Windows support to Kubernetes.

With the release of Kubernetes v1.14, Windows was successfully introduced as a [supported](https://kubernetes.io/docs/setup/production-environment/windows/intro-windows-in-kubernetes/) production ready operating system only for worker nodes of a Kubernetes cluster. This enabled Kubernetes to support the deployment of Windows containers in the cluster, either as a dedicated Windows cluster, or a shared cluster with Windows nodes running alongside Linux nodes. Keep in mind, however, that the control plane nodes are limited to running on Linux only, with no plans to extend the support to Windows master nodes.

With Windows Server 2019 being the only Windows OS supported by Kubernetes, the same container workload orchestration tool can [schedule](https://kubernetes.io/docs/setup/production-environment/windows/user-guide-windows-containers/) and deploy both Linux and Windows containers in the same cluster. The user is responsible to configure the workload scheduling according to the expected OS, that is to schedule Linux and Windows containers on nodes with their respective operating systems when nodes of each OS are found in the same Kubernetes cluster.

## Introduction

As mentioned earlier, [Minikube](https://kubernetes.io/docs/setup/learning-environment/minikube/" \t "_blank) is the easiest and most popular method to run an all-in-one Kubernetes cluster in a virtual machine (VM) locally on our workstations. Minikube is the tool responsible for the installation of Kubernetes components, cluster bootstrapping, and cluster tear-down when no longer needed. It includes a few additional tools aimed to ease the user interaction with the Kubernetes cluster, but nonetheless, it initializes for us an all-in-one single-node Kubernetes cluster. While the latest Minikube release also supports multi-node Kubernetes clusters, this feature is still experimental.

In this chapter, we will explore the requirements to install Minikube locally on our workstation, together with the installation instructions to set it up on local Linux, macOS, and Windows operating systems.



**Minikube**

## Requirements for Running Minikube

Minikube is installed and runs directly on a local Linux, macOS, or Windows workstation. However, in order to fully take advantage of all the features Minikube has to offer, a [Type-2 Hypervisor](https://en.wikipedia.org/wiki/Hypervisor) or a Container Runtime should be installed on the local workstation, to run in conjunction with Minikube. This does not mean that we need to create any VMs with guest operating systems with this Hypervisor or Containers. Minikube builds all its infrastructure as long as the Type-2 Hypervisor, or a Container Runtime, is installed on our workstation.

Minikube uses [libmachine](https://github.com/docker/machine/tree/master/libmachine" \t "_blank) to invoke the Hypervisor that provisions the VM which hosts a single-node Kubernetes cluster, or the Container Runtime to run the Container that hosts the cluster. Minikube then uses [kubeadm](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/" \t "_blank) to provision the Kubernetes cluster. Although VirtualBox is Minikube's original hypervisor driver, several other drivers are supported by Minikube when provisioning the environment in which the Kubernetes cluster is to be bootstrapped. Thus we need to make sure that we have the necessary hardware and software required by Minikube to build our environment.

Below we outline the requirements to run Minikube on our local workstation:

* + - VT-x/AMD-v virtualization must be enabled on the local workstation in BIOS, and/or [verify](https://kubernetes.io/docs/tasks/tools/install-minikube/#before-you-begin) if virtualization is supported by your workstation's OS.
    - [kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/)  
      **kubectl** is a binary used to access and manage any Kubernetes cluster. It is installed through Minikube and accessed through the **minikube** command, or it can be installed separately. We will explore **kubectl** in more detail in future chapters.
    - Type-2 Hypervisor or Container Runtime
      * + On Linux [VirtualBox](https://www.virtualbox.org/wiki/Downloads), [KVM2](https://www.linux-kvm.org/page/Main_Page) or Docker and Podman runtimes
        + On macOS [VirtualBox](https://www.virtualbox.org/wiki/Downloads), [HyperKit](https://github.com/moby/hyperkit" \t "_blank), [VMware Fusion](http://www.vmware.com/products/fusion.html), [Parallels](https://minikube.sigs.k8s.io/docs/drivers/parallels/) or Docker runtime
        + On Windows [VirtualBox](https://www.virtualbox.org/wiki/Downloads), [Hyper-V](https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v), or Docker runtime.  
            
          **NOTE:** Minikube supports a [**--driver=none**](https://minikube.sigs.k8s.io/docs/drivers/none/) (on Linux) option that runs the Kubernetes components directly on the host OS and not inside a VM. With this option a Docker installation is required and a Linux OS on the local workstation, but no hypervisor installation. If you use **--driver=none** in Debian or its derivatives, be sure to download [**.deb** Docker packages](https://www.docker.com/products/docker-desktop) instead of the snap package which does not work with Minikube. In addition to hypervisors, Minikube also supports a limited number of container runtimes, with the [**--driver=docker**](https://minikube.sigs.k8s.io/docs/drivers/docker/) (on Linux, macOS, and Windows) and [**--driver=podman**](https://minikube.sigs.k8s.io/docs/drivers/podman/) (on Linux) options, to install and run the Kubernetes cluster on top of a container runtime.
    - Internet connection at least on first Minikube run - to download packages, dependencies, updates and pull images needed to initialize the Minikube Kubernetes cluster. Subsequent runs will require an internet connection only when new Docker images need to be pulled from a container registry or when deployed containerized applications need it. Once an image has been pulled it can be reused without an internet connection.

In this chapter, we use VirtualBox as hypervisor on all three operating systems - Linux, macOS, and Windows, to allow Minikube to provision the VM which hosts the single-node Kubernetes cluster.

Read more about Minikube from the official [Minikube documentation](https://minikube.sigs.k8s.io/docs/" \t "_blank), the official [Kubernetes documentation](https://kubernetes.io/docs/setup/learning-environment/minikube/), or [GitHub](https://github.com/kubernetes/minikube).

## Installing Minikube on Linux

Let's learn how to install the latest Minikube release on Ubuntu Linux 18.04 LTS with VirtualBox v6.1 specifically.

**NOTE: For other Linux OS distributions or releases, VirtualBox and Minikube versions the installation steps may vary! Check the [Minikube installation](https://kubernetes.io/docs/tasks/tools/install-minikube/" \t "_blank)!**

**Verify the virtualization support on your Linux OS (a non-empty output indicates supported virtualization):**

**$ grep -E --color 'vmx|svm' /proc/cpuinfo**

**Install the**[**VirtualBox**](https://www.virtualbox.org/wiki/Linux_Downloads)**hypervisor**  
Add the source repository for the **bionic** distribution (Ubuntu 18.04), download and register the public key, update and install:

**$ sudo bash -c 'echo "deb [arch=amd64] https://download.virtualbox.org/virtualbox/debian bionic contrib" >> /etc/apt/sources.list'**

**$ wget -q https://www.virtualbox.org/download/oracle\_vbox\_2016.asc -O- | sudo apt-key add -**

**$ sudo apt update**

**$ sudo apt install -y virtualbox-6.1**

**Install Minikube**  
We can download the latest release or a specific release from the [Minikube release page](https://github.com/kubernetes/minikube/releases" \t "_blank). Once downloaded, we need to make it executable and add it to our **PATH**:

**$ curl -Lo minikube**[**https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64**](https://storage.googleapis.com/minikube/releases/v1.0.1/minikube-linux-amd64)**&& chmod +x minikube && sudo mv minikube /usr/local/bin/**

**NOTE:** Replacing **/latest/** with a particular version, such as **/v1.13.0/** will download that specified version.

**Start Minikube**  
We can start Minikube with the **minikube start** command, that bootstraps a single-node cluster with the latest stable Kubernetes version release. For a specific Kubernetes version the **--kubernetes-version** option can be used as such **minikube start --kubernetes-version v1.19.0** (where **latest** is default and acceptable version value, and **stable** is also acceptable):

**$ minikube start**

**😄  minikube v1.13.1 on Ubuntu 18.04**  
**✨  Automatically selected the virtualbox driver**  
**💿  Downloading VM boot image ...**  
**> minikube-v1.13.1.iso.sha256: 65 B / 65 B [-------------] 100.00% ? p/s 0s**  
**> minikube-v1.13.1.iso: 173.91 MiB / 173.91 MiB [] 100.00% 26.59 MiB p/s 6s**  
**👍  Starting control plane node minikube in cluster minikube**  
**💾  Downloading Kubernetes v1.19.2 preload ...**  
**> preloaded-images-k8s-v6-v1.19.2-docker-overlay2-amd64.tar.lz4: 486.36 MiB**  
**🔥  Creating virtualbox VM (CPUs=2, Memory=3900MB, Disk=20000MB) ...**  
**🐳  Preparing Kubernetes v1.19.2 on Docker 19.03.12 ...**  
**🔎  Verifying Kubernetes components...**  
**🌟  Enabled addons: default-storageclass, storage-provisioner**  
**💡  kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'**  
**🏄  Done! kubectl is now configured to use "minikube" by default**

**Check the status**

With the **minikube status** command, we display the status of Minikube:

**$ minikube status**

**minikube**  
**type: Control Plane**  
**host: Running**  
**kubelet: Running**  
**apiserver: Running**  
**kubeconfig: Configured**

**Stop Minikube**  
With the **minikube stop**command, we can stop Minikube:  
  
**$ minikube stop**

**Stopping node "minikube"  ...**  
**1 nodes stopped.**

## Installing Minikube on macOS

Let's learn how to install the latest Minikube release on Mac OS X with VirtualBox v6.1 specifically.

**NOTE: For other VirtualBox and Minikube versions the installation steps may vary! Check the [Minikube installation](https://kubernetes.io/docs/tasks/tools/install-minikube/" \t "_blank)!**

**Verify the virtualization support on your macOS (VMX in the output indicates enabled virtualization):**

**$ sysctl -a | grep -E --color 'machdep.cpu.features|VMX'**

Although VirtualBox is the default hypervisor for Minikube, on Mac OS X we can configure Minikube at startup to use another hypervisor (downloaded separately), with the **--driver=parallels** or **--driver=hyperkit** start option.

**Install the**[**VirtualBox**](https://www.virtualbox.org/wiki/Downloads)**hypervisor for 'OS X hosts'**  
Download and install the **.dmg** package.

**Install Minikube**  
We can download the latest release or a specific release from the [Minikube release page](https://github.com/kubernetes/minikube/releases" \t "_blank). Once downloaded, we need to make it executable and add it to our **PATH**:

**$ curl -Lo minikube**[**https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64**](https://storage.googleapis.com/minikube/releases/v1.0.1/minikube-darwin-amd64)**&& chmod +x minikube && sudo mv minikube /usr/local/bin/**

**NOTE:** Replacing **/latest/** with a particular version, such as **/v1.13.0/** will download that specified version.

**Start Minikube**  
We can start Minikube with the **minikube start** command, that bootstraps a single-node cluster with the latest stable Kubernetes version release. For a specific Kubernetes version the **--kubernetes-version** option can be used as such **minikube start --kubernetes-version v1.19.0** (where **latest** is default and acceptable version value, and **stable** is also acceptable):

**$ minikube start**

**😄  minikube v1.13.1 on Darwin 10.15.6**  
**✨  Automatically selected the virtualbox driver**  
**💿  Downloading VM boot image ...**  
**👍  Starting control plane node minikube in cluster minikube**  
**💾  Downloading Kubernetes v1.19.2 preload ...**  
**🔥  Creating virtualbox VM (CPUs=2, Memory=3900MB, Disk=20000MB) ...**  
**🐳  Preparing Kubernetes v1.19.2 on Docker 19.03.12 ...**  
**🔎  Verifying Kubernetes components...**  
**🌟  Enabled addons: default-storageclass, storage-provisioner**  
**💡  kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'**  
**🏄  Done! kubectl is now configured to use "minikube" by default**

**Check the status**  
With the **minikube status** command, we display the status of Minikube:

**$ minikube status**

**minikube**  
**type: Control Plane**  
**host: Running**  
**kubelet: Running**  
**apiserver: Running**  
**kubeconfig: Configured**

**Stop Minikube**  
With the **minikube stop**command, we can stop Minikube:  
  
**$ minikube stop**

**Stopping node "minikube"  ...**  
**1 nodes stopped.**

## Installing Minikube on Windows

Let's learn how to install the latest Minikube release on Windows 10 with VirtualBox v6.1 specifically.

**NOTE: For other OS, VirtualBox, and Minikube versions, the installation steps may vary! Check the [Minikube installation](https://kubernetes.io/docs/tasks/tools/install-minikube/" \t "_blank)!**

**Verify the virtualization support on your Windows system (multiple output lines ending with 'Yes' indicate supported virtualization):**

**PS C:\WINDOWS\system32> systeminfo**

**Install the**[**VirtualBox**](https://www.virtualbox.org/wiki/Downloads)**hypervisor for 'Windows hosts'**  
Download and install the **.exe** package.

**NOTE:** You **may** need to disable Hyper-V on your Windows host (if previously installed and used) while running VirtualBox.

**Install Minikube**  
We can download the latest release or a specific release from the [Minikube release page](https://github.com/kubernetes/minikube/releases" \t "_blank). Once downloaded, we need to make sure it is added to our **PATH**.

There are two **.exe** packages available to download for Windows found under a Minikube release:

* + - **minikube-windows-amd64.exe** which requires to be added to the **PATH:** manually
    - **minikube-installer.exe** which automatically adds the executable to the **PATH**.

Let's download and install the **latest** [**minikube-installer.exe**](https://github.com/kubernetes/minikube/releases/latest/download/minikube-installer.exe) package.

**Start Minikube**  
We can start Minikube using the **minikube start** command, that bootstraps a single-node cluster with the latest stable Kubernetes version release. For a specific Kubernetes version the **--kubernetes-version** option can be used as such **minikube start --kubernetes-version v1.19.0** (where **latest** is default and acceptable version value, and **stable** is also acceptable). Open the PowerShell using the *Run as Administrator* option and execute the following command:

**PS C:\WINDOWS\system32> minikube start**

**😄  minikube v1.13.1 on Windows 10**  
**✨  Automatically selected the virtualbox driver**  
**💿  Downloading VM boot image ...**  
**👍  Starting control plane node minikube in cluster minikube**  
**💾  Downloading Kubernetes v1.19.2 preload ...**  
**🔥  Creating virtualbox VM (CPUs=2, Memory=3900MB, Disk=20000MB) ...**  
**🐳  Preparing Kubernetes v1.19.2 on Docker 19.03.12 ...**  
**🔎  Verifying Kubernetes components...**  
**🌟  Enabled addons: default-storageclass, storage-provisioner**  
**💡  kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'**  
**🏄  Done! kubectl is now configured to use "minikube" by default**

**Check the status**  
We can see the status of Minikube using the **minikube status** command. Open the PowerShell using the *Run as Administrator* option and execute the following command:

**PS C:\WINDOWS\system32> minikube status**

**minikube**  
**type: Control Plane**  
**host: Running**  
**kubelet: Running**  
**apiserver: Running**  
**kubeconfig: Configured**

**Stop Minikube**  
We can stop Minikube using the **minikube stop** command. Open the PowerShell using the *Run as Administrator* option and execute the following command:

**PS C:\WINDOWS\system32> minikube stop**

**Stopping node "minikube"  ...**  
**1 nodes stopped.**

## Minikube CRI-O

According to the [CRI-O website](http://cri-o.io/),

*"CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes."*

**Start Minikube with CRI-O as container runtime, instead of Docker, with the following command:**

**NOTE:**While **docker** is the default runtime, minikube Kubernetes also supports **cri-o** and **containerd**.

**$ minikube start --container-runtime cri-o**

**😄  minikube v1.13.1 on Ubuntu 18.04**  
**✨  Automatically selected the virtualbox driver**  
**💿  Downloading VM boot image ...**  
**👍  Starting control plane node minikube in cluster minikube**  
**💾  Downloading Kubernetes v1.19.2 preload ...**  
**> preloaded-images-k8s-v6-v1.19.2-cri-o-overlay-amd64.tar.lz4: 551.15 MiB /**  
**🔥  Creating virtualbox VM (CPUs=2, Memory=3900MB, Disk=20000MB) ...**  
**🐳  Preparing Kubernetes v1.19.2 on CRI-O 1.17.3 ...**  
**🔗  Configuring bridge CNI (Container Network Interface) ...**  
**🔎  Verifying Kubernetes components...**  
**🌟  Enabled addons: default-storageclass, storage-provisioner**  
**🏄  Done! kubectl is now configured to use "minikube" by default**

**By describing a running Kubernetes pod, we can extract the Container ID field of the pod that includes the name of the runtime:**

**$ kubectl -n kube-system describe pod kube-scheduler-minikube | grep "Container ID"**

**Container ID:  cri-o://1090869caeea44cb179d31b70ba5b6de...**

**Let's login via ssh into the Minikube's VM:**

**$ minikube ssh**

\_ \_

**\_ \_ ( ) ( )**

**\_\_\_ \_\_\_ (\_) \_\_\_ (\_)| |/') \_ \_ | |\_ \_\_**

**/' \_ ` \_ `\| |/' \_ `\| || , < ( ) ( )| '\_`\ /'\_\_`\**

**| ( ) ( ) || || ( ) || || |\`\ | (\_) || |\_) )( \_\_\_/**

**(\_) (\_) (\_)(\_)(\_) (\_)(\_)(\_) (\_)`\\_\_\_/'(\_,\_\_/'`\\_\_\_\_)**

**$ \_**

**NOTE: If you try to list containers using the docker command, it will not produce any results, because Docker is not running our containers at this time:**

**$ sudo docker container ls**

**Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?**

**List the containers created by the CRI-O container runtime and extract the container manager from the config file:**

**$ sudo runc list**

**ID                                                                 PID         STATUS       BUNDLE                                                                                 CREATED                          OWNER**  
**1090869caeea44cb179d31b70ba5b6de96f10a8a5f4286536af5dac1c4312030   3661        running     /run/containers/storage/overlay-containers/1090869caeea44cb179d31b70ba5b6de96f10a8a5f4286536af5dac1c4312030/userdata   2020-10-11T02:26:03.675763329Z   root**  
**1e9f8dce6d535b67822e744204098060ff92e574780a1809adbda48ad8605d06   3614        running     /run/containers/storage/overlay-containers/1e9f8dce6d535b67822e744204098060ff92e574780a1809adbda48ad8605d06/userdata   2020-10-11T02:25:21.650715545Z   root**  
**1edcfc78bca52be153cc9f525d9fc64be75ccea478897004a5032f37c6c4c9dc   3812        running     /run/containers/storage/overlay-containers/1edcfc78bca52be153cc9f525d9fc64be75ccea478897004a5032f37c6c4c9dc/userdata   2020-10-11T02:25:33.468528462Z   root**  
**...**

**$ sudo cat /run/containers/storage/overlay-containers/1090869caeea44cb179d31b70ba5b6de96f10a8a5f4286536af5dac1c4312030/userdata/config.json | grep manager**

**"io.container.manager": "cri-o",**

## Installing Minikube (Demo)

[Video link](8c97de85-f52f-481b-8c4e-95ec27a8d38f-mp4_720p.mp4)  
[Text](Ch%206%20-%20Installing%20Minikube-en.txt)

## Introduction

In this chapter, we will learn about different methods of accessing a Kubernetes cluster.

We can use a variety of external clients or custom scripts to access our cluster for administration purposes. We will explore **kubectl** as a CLI tool to access the Minikube Kubernetes cluster, the **Kubernetes Dashboard** as a web-based user interface to interact with the cluster, and the **curl** command with proper credentials to access the cluster via APIs.

## Accessing Minikube

Any healthy running Kubernetes cluster can be accessed via any one of the following methods:

* + - **Command Line Interface (CLI) tools and scripts**
    - **Web-based User Interface (Web UI) from a web browser**
    - **APIs from CLI or programmatically**

These methods are applicable to all Kubernetes clusters.

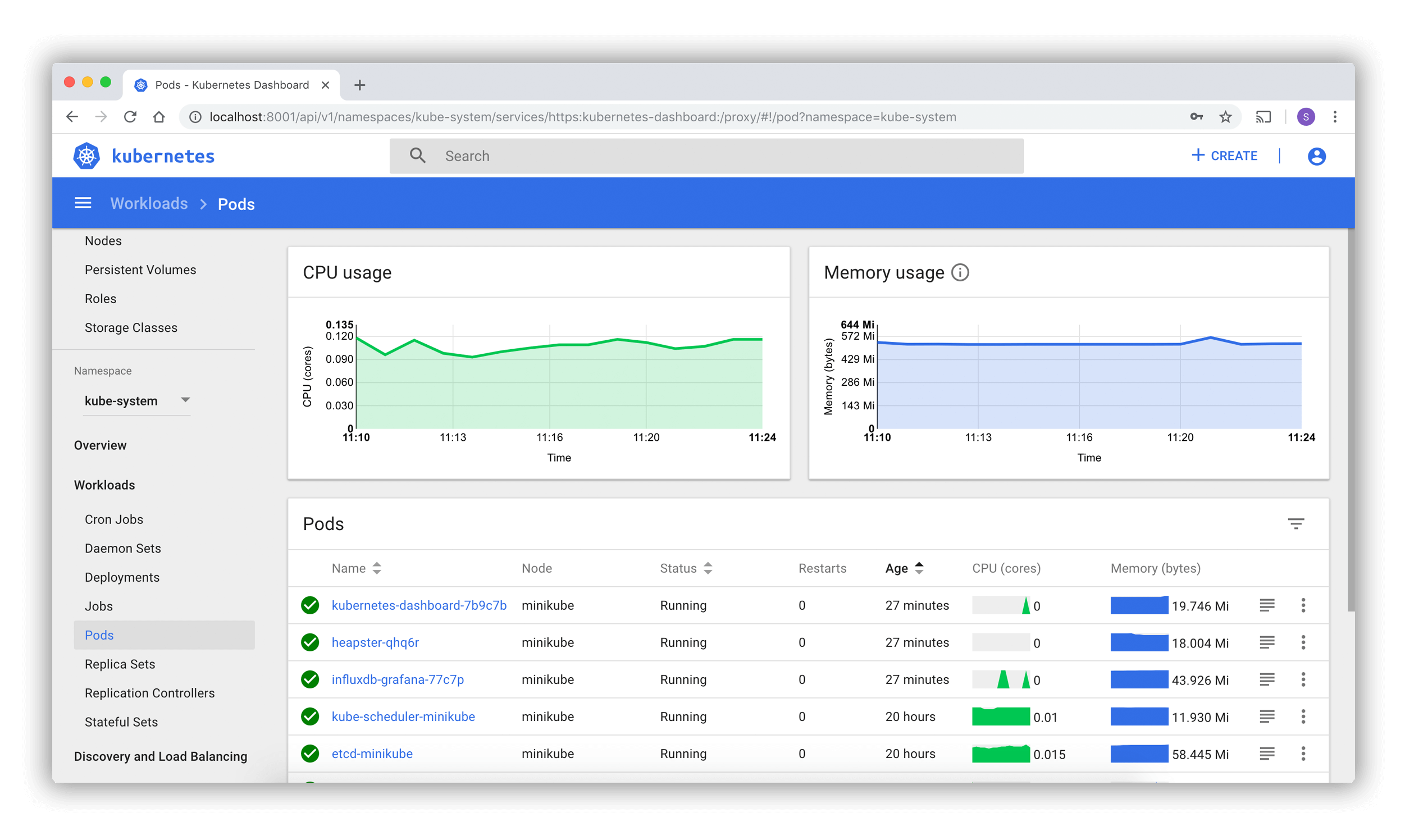
## Accessing Minikube: Command Line Interface (CLI)

[**kubectl**](https://kubernetes.io/docs/reference/kubectl/overview/) is the **Kubernetes Command Line Interface (CLI) client** to manage cluster resources and applications. It is very flexible and easy to integrate with other systems, therefore it can be used standalone, or part of scripts and automation tools. Once all required credentials and cluster access points have been configured for **kubectl**, it can be used remotely from anywhere to access a cluster.

In later chapters, we will be using **kubectl** to deploy applications, manage and configure Kubernetes resources.

## Accessing Minikube: Web-based User Interface (Web UI)

The [**Kubernetes Dashboard**](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) provides a **Web-Based User Interface (Web UI)** to interact with a Kubernetes cluster to manage resources and containerized applications. In one of the later chapters, we will be using it to deploy a containerized application.

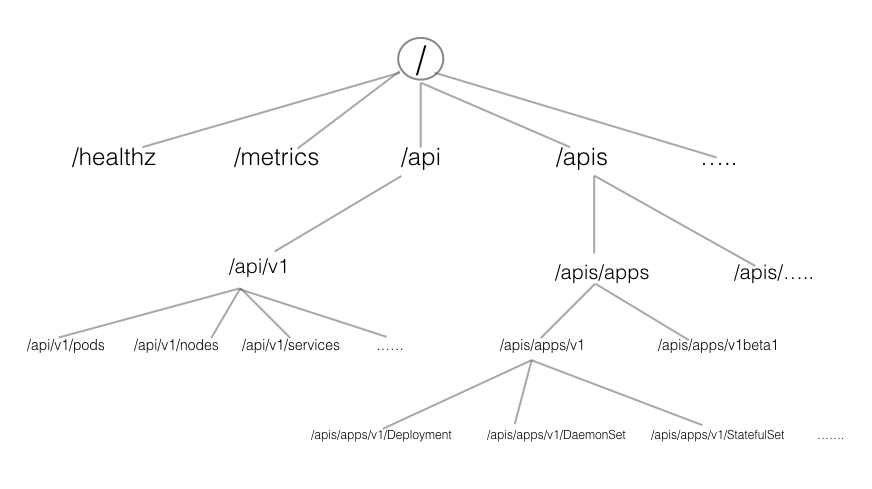


**Kubernetes Dashboard User Interface**

## Accessing Minikube: APIs

The main component of the Kubernetes control plane is the **API server**, responsible for exposing the Kubernetes APIs. The APIs allow operators and users to directly interact with the cluster. Using both CLI tools and the Dashboard UI, we can access the API server running on the master node to perform various operations to modify the cluster's state. The API server is accessible through its endpoints by agents and users possessing the required credentials.

Below, we can see the representation of the HTTP API directory tree of Kubernetes:



**HTTP API directory tree of Kubernetes**

HTTP API directory tree of Kubernetes can be divided into three independent group types:

* + - **Core Group (/api/v1)**This group includes objects such as Pods, Services, Nodes, Namespaces, ConfigMaps, Secrets, etc.
    - **Named Group**This group includes objects in **/apis/$NAME/$VERSION** format. These different API versions imply different levels of stability and support:  
      *Alpha level* - it may be dropped at any point in time, without notice. For example, **/apis/batch/v2alpha1**.  
      *Beta level* - it is well-tested, but the semantics of objects may change in incompatible ways in a subsequent beta or stable release. For example, **/apis/certificates.k8s.io/v1beta1**.  
      *Stable level* - appears in released software for many subsequent versions. For example, **/apis/networking.k8s.io/v1**.
    - **System-wide**This group consists of system-wideAPI endpoints, like **/healthz**, **/logs**, **/metrics**, **/ui**, etc.

We can access an API server either directly by calling the respective API endpoints, using the CLI tools, or the Dashboard UI.

Next, we will see how we can access the Minikube Kubernetes cluster we set up in the previous chapter.

## kubectl

**kubectl** allows us to manage local Kubernetes clusters like the Minikube cluster, or remote clusters deployed in the cloud. It is generally installed before installing and starting Minikube, but it can also be installed after the cluster bootstrapping step. Once installed, **kubectl** receives its configuration automatically for Minikube Kubernetes cluster access. However, in different Kubernetes cluster setups, we may need to manually configure the cluster access points and certificates required by **kubectl** to securely access the cluster.

There are different methods that can be used to install **kubectl** listed in the [Kubernetes documentation](https://kubernetes.io/docs/tasks/tools/install-kubectl/). For best results, it is recommended to keep **kubectl** at the same version with the Kubernetes run by Minikube. Next, we will look at a few steps to install it on Linux, macOS, and Windows systems.

Details about the **kubectl** command line client can be found in the [kubectl book](https://kubectl.docs.kubernetes.io/" \t "_blank), the [Kubernetes official documentation](https://kubernetes.io/search/?q=kubectl), or its [github repository](https://github.com/kubernetes/kubectl" \t "_blank).

## Installing kubectl on Linux

To install **kubectl** on Linux, follow the instruction below:

**Download the latest stable kubectl binary, make it executable and move it to the PATH:**  
  
**$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl" && chmod +x kubectl && sudo mv kubectl /usr/local/bin/**

Where [**https://storage.googleapis.com/kubernetes-release/release/stable.txt**](https://storage.googleapis.com/kubernetes-release/release/stable.txt) aims to display the latest Kubernetes stable release version.

**NOTE:** To download and setup a specific version of **kubectl** (such as v1.19.1), issue the following command:

**$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.19.1/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/**

**A typical helpful post-installation configuration is to enable**[**shell autocompletion**](https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion)**for kubectl. It can be achieved by running the following four commands:**

**$ sudo apt install -y bash-completion**

**$ source /usr/share/bash-completion/bash-completion**

**$ source <(kubectl completion bash)**

**$ echo 'source <(kubectl completion bash)' >>~/.bashrc**

## Installing kubectl on macOS

There are two methods to install **kubectl** on macOS: manually and using the Homebrew package manager. Next, we present both installation methods.

**To manually install kubectl, download the latest stable kubectl binary, make it executable and move it to the PATH with the following command:**

**$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl" && chmod +x kubectl && sudo mv kubectl /usr/local/bin/**

Where [**https://storage.googleapis.com/kubernetes-release/release/stable.txt**](https://storage.googleapis.com/kubernetes-release/release/stable.txt) aims to display the latest Kubernetes stable release version.

**NOTE:** To download and setup a specific version of **kubectl** (such as v1.19.1), issue the following command:

**$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.19.1/bin/darwin/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/**

**To install kubectl with**[**Homebrew package manager**](https://brew.sh/)**, issue the following command:**

**$ brew install kubernetes**

or

**$ brew install kubernetes-cli**

**A typical helpful post-installation configuration is to enable**[**shell autocompletion**](https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion)**for kubectl.**

## Installing kubectl on Windows

To install **kubectl**, we can download the binary directly or use **curl** from the CLI. Once downloaded the binary needs to be added to the **PATH**.

**Direct download link for v1.19.2 binary (just click below):**

[**https://storage.googleapis.com/kubernetes-release/release/v1.19.2/bin/windows/amd64/kubectl.exe**](https://storage.googleapis.com/kubernetes-release/release/v1.19.2/bin/windows/amd64/kubectl.exe)

**NOTE:**Obtain the latest **kubectl** stable release version number from the link below, and if needed, edit the download link for the desired binary version from above:

[**https://storage.googleapis.com/kubernetes-release/release/stable.txt**](https://storage.googleapis.com/kubernetes-release/release/stable.txt)

**Use the curl command (if installed) from the CLI:**

**curl -LO**[**https://storage.googleapis.com/kubernetes-release/release/v1.19.2/bin/windows/amd64/kubectl.exe**](https://storage.googleapis.com/kubernetes-release/release/v1.19.2/bin/windows/amd64/kubectl.exe)

**Once downloaded, move the kubectl binary to the PATH.**

**NOTE: Docker Desktop** for Windows adds its own version of **kubectl** to **PATH**. If you have installed **Docker Desktop** before, you may need to place your **PATH** entry before the one added by the **Docker Desktop** installer or remove the **Docker Desktop**'s **kubectl**.

## kubectl Configuration File

To access the Kubernetes cluster, the **kubectl** client needs the master node endpoint and appropriate credentials to be able to securely interact with the API server running on the master node. While starting Minikube, the startup process creates, by default, a configuration file, **config**, inside the**.kube** directory (often referred to as the **[kubeconfig](https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/" \t "_blank)**), which resides in the user's **home** directory. The configuration file has all the connection details required by **kubectl**. By default, the **kubectl** binary parses this file to find the master node's connection endpoint, along with credentials. Multiple **kubeconfig** files can be configured with a single **kubectl** client. To look at the connection details, we can either display the content of the **~/.kube/config** file (on Linux) or run the following command:

**$ kubectl config view**

**apiVersion: v1**  
**clusters:**  
**- cluster:**  
**certificate-authority: /home/student/.minikube/ca.crt**  
**server: https://192.168.99.100:8443**  
**name: minikube**  
**contexts:**  
**- context:**  
**cluster: minikube**  
**user: minikube**  
**name: minikube**  
**current-context: minikube**  
**kind: Config**  
**preferences: {}**  
**users:**  
**- name: minikube**  
**user:**  
**client-certificate: /home/student/.minikube/profiles/minikube/client.crt**  
**client-key: /home/student/.minikube/profiles/minikube/client.key**

The kubeconfig includes the API server's endpoint **server: https://192.168.99.100:8443** and the **minikube** user's client authentication **key** and **certificate** data.

Once **kubectl** is installed, we can display information about the Minikube Kubernetes cluster with the **kubectl cluster-info** command:

**$ kubectl cluster-info**

**Kubernetes master is running at https://192.168.99.100:8443**  
**KubeDNS is running at https://192.168.99.100:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy**

**To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.**

You can find more details about the **kubectl** command line options [here](https://kubernetes.io/docs/reference/kubectl/overview/).

Although for the Kubernetes cluster installed by Minikube the **~/.kube/config** file gets created automatically, this is not the case for Kubernetes clusters installed by other tools. In other cases, the config file has to be created manually and sometimes re-configured to suit various networking and client/server setups.

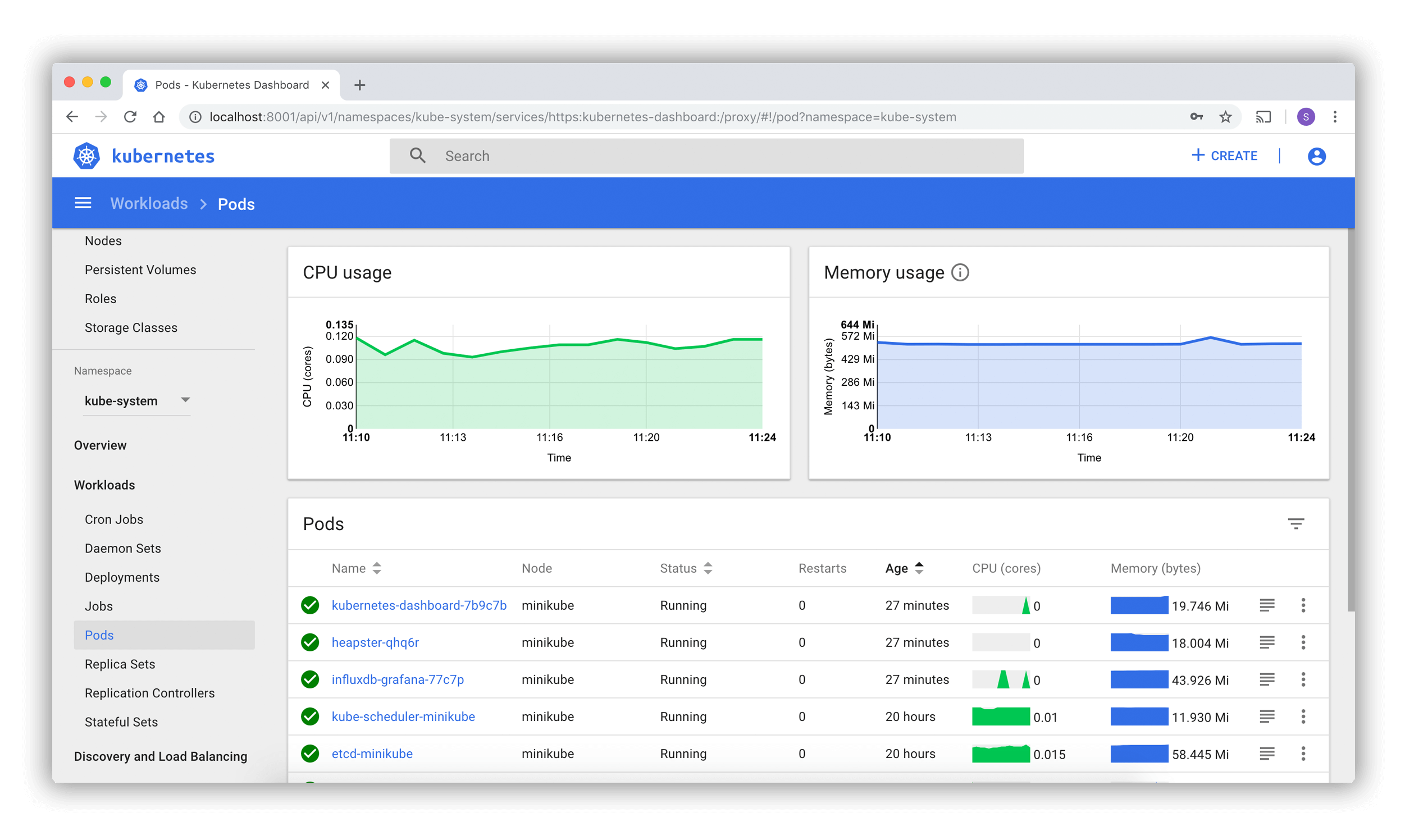
## Installing kubectl CLI Client (Demo)

[video](1fe4774e-7cf7-46af-88ec-a39d56ab659f-mp4_720p.mp4)  
[Text](Ch%207%20-%20Installing%20kubectl%20CLI%20Client-en.txt)

## Kubernetes Dashboard

As mentioned earlier, the [**Kubernetes Dashboard**](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) provides a web-based user interface for Kubernetes cluster management. To access the dashboard from Minikube, we can use the **minikube dashboard** command, which opens a new tab in our web browser, displaying the Kubernetes Dashboard:

**$ minikube dashboard**

****

**Kubernetes Dashboard User Interface**

**NOTE:** In case the browser is not opening another tab and does not display the Dashboard as expected, verify the output in your terminal as it may display a link for the Dashboard (together with some Error messages). Copy and paste that link in a new tab of your browser. Depending on your terminal's features you may be able to just click or right-click the link to open directly in the browser. The link may look similar to:

[**http://localhost:37751/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/**](http://localhost:37751/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/)

Chances are that the only difference is the PORT number 37751 above, while your port number may be different.

After a logout/login or a reboot of your workstation the normal behavior should be expected (where the **minikube dashboard** command directly opens a new tab in your browser displaying the Dashboard).

## Kubernetes Dashboard with 'kubectl proxy'

Issuing the **kubectl proxy** command, **kubectl** authenticates with the API server on the master node and makes the Dashboard available on a slightly different URL than the one earlier, this time through the default proxy port **8001**.

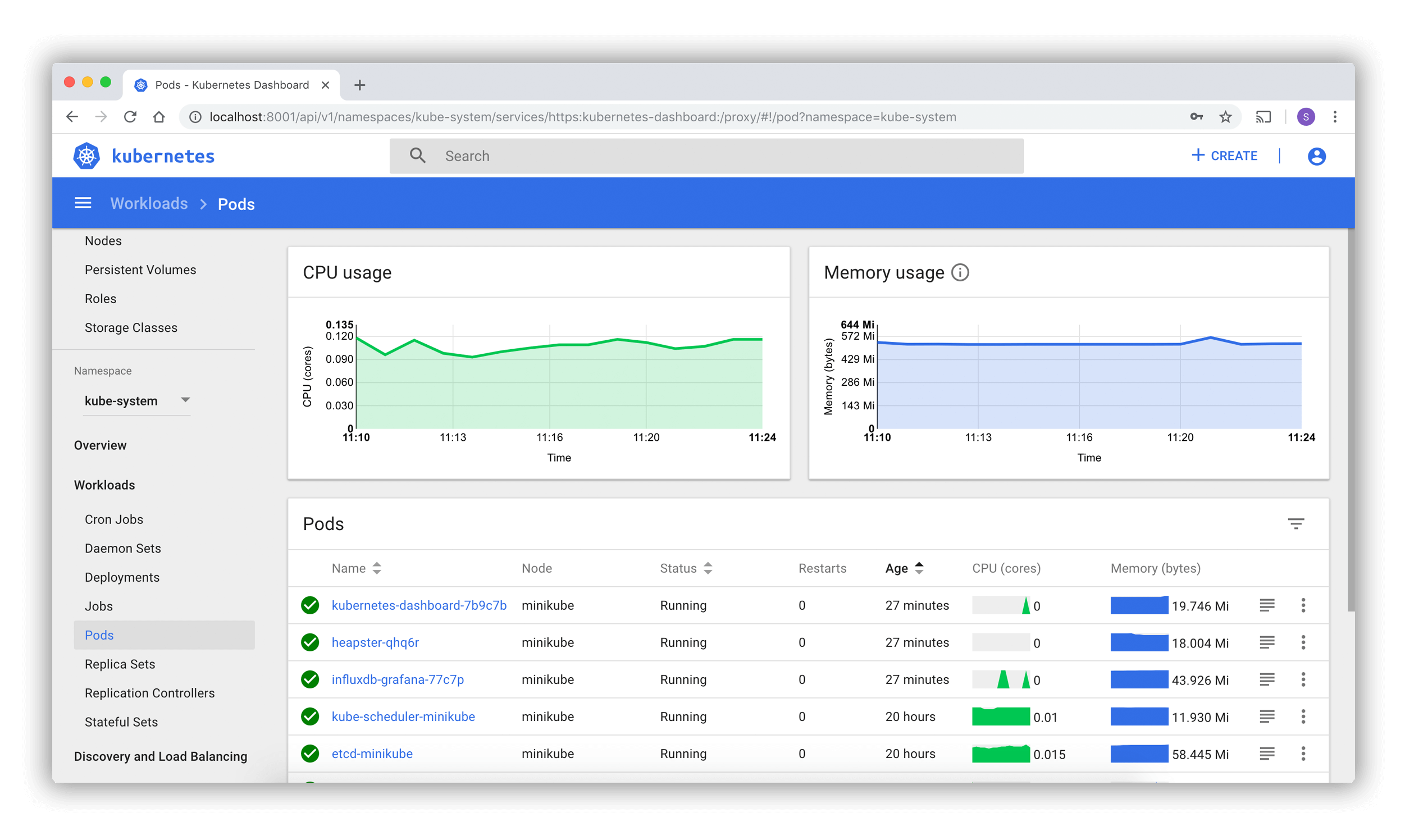
First, we issue the **kubectl proxy** command:

**$ kubectl proxy**

**Starting to serve on 127.0.0.1:8001**

It locks the terminal for as long as the proxy is running, unless we run it in the background (with **kubectl proxy &**). With the **proxy** running we can access the **Dashboard** over the new URL (just click on it below - it should work on your workstation). Once we stop the proxy (with CTRL + C) the Dashboard URL is no longer accessible.

[**http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/**](http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/)



**Kubernetes Dashboard over the proxy**

## Enabling the Dashboard and the metrics-server (Demo)

[video](d1594a1d-8c01-4e2b-b696-f6366e48bdec-mp4_720p.mp4)  
[Text](Ch%207%20-%20Enabling%20the%20Dashboard%20and%20the%20metrics-serv-en.txt)

## APIs with 'kubectl proxy'

When **kubectl proxy** is running, we can send requests to the API over the **localhost** on the default proxy port **8001** (from another terminal, since the proxy locks the first terminal when running in foreground):

**$ curl**[**http://localhost:8001/**](http://localhost:8001/)

**{**  
**"paths": [**  
**"/api",**  
**"/api/v1",**  
**"/apis",**  
**"/apis/apps",**  
**......**  
**......**  
**"/logs",**  
**"/metrics",**  
**"/openapi/v2",**  
**"/version"**  
**]  
}**

With the above **curl** request, we requested all the API endpoints from the API server. Clicking on the link above (in the **curl** command), it will open the same listing output in a browser tab.

We can explore several path combinations with **curl** or in a browser as well, such as:

[**http://localhost:8001/api/v1**](http://localhost:8001/api/v1)

[**http://localhost:8001/apis/apps/v1**](http://localhost:8001/apis/apps/v1)

[**http://localhost:8001/healthz**](http://localhost:8001/healthz)

[**http://localhost:8001/metrics**](http://localhost:8001/metrics)

## APIs with Authentication

When not using the **kubectl proxy**, we need to authenticate to the API server when sending API requests. We can authenticate by providing a **Bearer Token** when issuing a **curl**, or by providing a set of **keys** and **certificates**.

A **Bearer Token** is an **access token** which is generated by the authentication server (the API server on the master node) and given back to the client. Using that token, the client can connect back to the Kubernetes API server without providing further authentication details, and then, access resources.

**Get the token:**

**$ TOKEN=$(kubectl describe secret -n kube-system $(kubectl get secrets -n kube-system | grep default | cut -f1 -d ' ') | grep -E '^token' | cut -f2 -d':' | tr -d '\t' | tr -d " ")**

**Get the API server endpoint:**

**$ APISERVER=$(kubectl config view | grep https | cut -f 2- -d ":" | tr -d " ")**

**Confirm that the APISERVER stored the same IP as the Kubernetes master IP by issuing the following 2 commands and comparing their outputs:**  
  
**$ echo $APISERVER**

**https://192.168.99.100:8443**

**$ kubectl cluster-info**

**Kubernetes master is running at https://192.168.99.100:8443 ...**

**Access the API server using the curl command, as shown below:**  
  
**$ curl $APISERVER --header "Authorization: Bearer $TOKEN" --insecure**

**{**  
**"paths": [**  
**"/api",**  
**"/api/v1",**  
**"/apis",**  
**"/apis/apps",**  
**......**  
**......**  
**"/logs",**  
**"/metrics",**  
**"/openapi/v2",**  
**"/version"**  
**]**  
**}**

Instead of the **access token,** we can extract the client certificate, client key, and certificate authority data from the **.kube/config** file. Once extracted, they can be encoded and then passed with a **curl** command for authentication. The new **curl** command would look similar to the example below. Keep in mind, however, that the below example command would only work with the encoded client certificae, key and certificate authority data.

**$ curl $APISERVER --cert encoded-cert --key encoded-key --cacert encoded-ca**

## Accessing the Kubernetes API with kubectl (Demo)

[video](f781f2ec-8e70-4123-a2b8-c3dd7adfc219-mp4_720p.mp4)  
[Text](Ch%207%20-%20Accessing%20the%20Kubernetes%20API%20with%20kubectl-en.txt)

## Introduction

In this chapter, we will explore the **Kubernetes object model**and describe some of its fundamental building blocks, such as **Pods**, **ReplicaSets**, **Deployments**,**Namespaces**, etc. We will also discuss the essential role of **Labels**and **Selectors** in a microservices driven architecture as they logically group decoupled objects together.

## Kubernetes Object Model

Kubernetes has a very rich object model, representing different persistent entities in the Kubernetes cluster. Those entities describe:

* + - What containerized applications we are running
    - The nodes where the containerized applications are deployed
    - Application resource consumption
    - Policies attached to applications, like restart/upgrade policies, fault tolerance, etc.

With each object, we declare our intent, or the desired state of the object, in the **spec** section. The Kubernetes system manages the **status** section for objects, where it records the actual state of the object. At any given point in time, the Kubernetes Control Plane tries to match the object's actual state to the object's desired state.

Examples of Kubernetes objects are Pods, ReplicaSets, Deployments, Namespaces, etc. We will explore them next.

When creating an object, the object's configuration data section from below the **spec** field has to be submitted to the Kubernetes API server. The **spec** section describes the desired state, along with some basic information, such as the object's name. The API request to create an object must have the **spec** section, as well as other details. Although the API server accepts object definition files in a JSON format, most often we provide such files in a YAML format which is converted by **kubectl** in a JSON payload and sent to the API server.

Below is an example of a [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) object's configuration manifest in YAML format:

**apiVersion: apps/v1**  
**kind: Deployment**  
**metadata:**  
**name: nginx-deployment**  
**labels:**  
**app: nginx**  
**spec:**  
**replicas: 3**  
**selector:**  
**matchLabels:**  
**app: nginx**  
**template:**  
**metadata:**  
**labels:**  
**app: nginx**  
**spec:**  
**containers:**  
**- name: nginx**  
**image: nginx:1.15.11**  
**ports:**  
**- containerPort: 80**

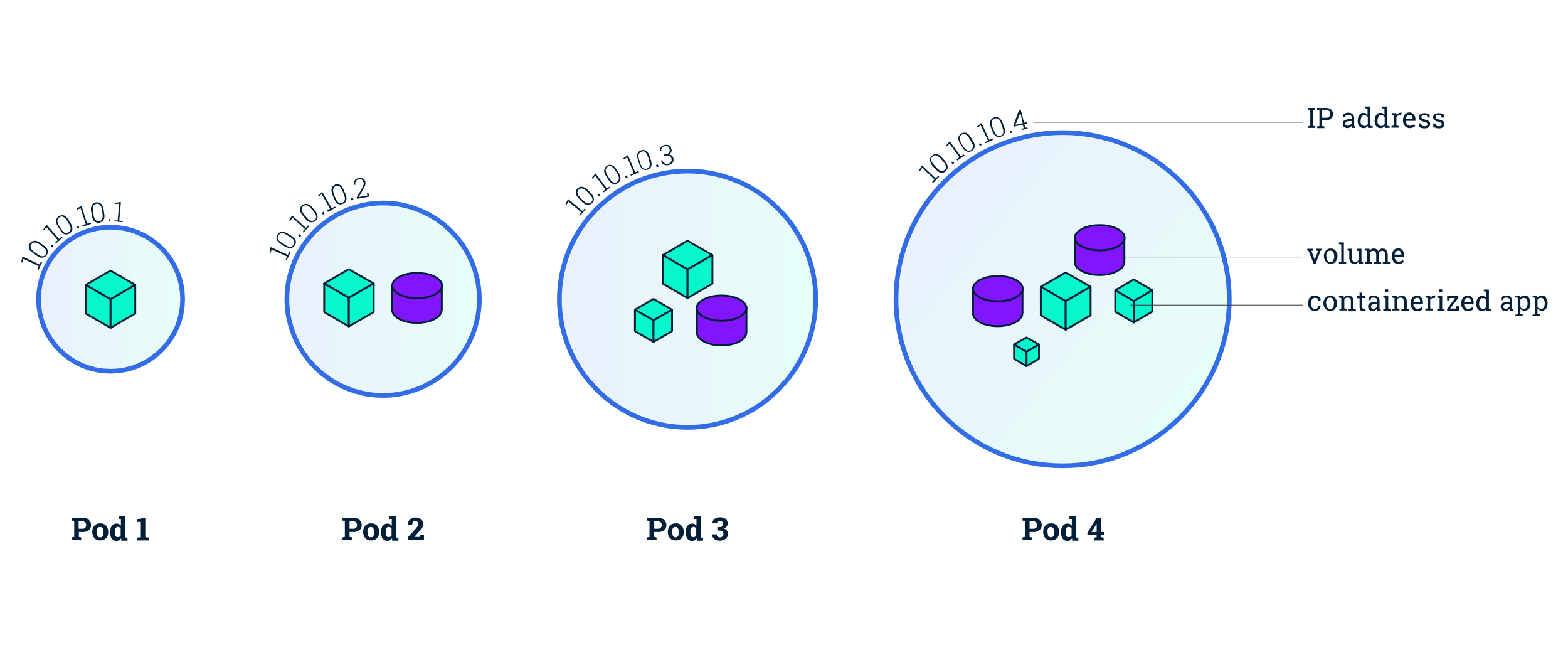
The **apiVersion**field is the first required field, and it specifies the API endpoint on the API server which we want to connect to; it must match an existing version for the object type defined. The second required field is **kind**, specifying the object type - in our case it is **Deployment**, but it can be Pod, Replicaset, Namespace, Service, etc. The third required field **metadata**, holds the object's basic information, such as name, labels, namespace, etc. Our example shows two **spec**fields (**spec** and **spec.template.spec**). The fourth required field **spec** marks the beginning of the block defining the desired state of the Deployment object. In our example, we are requesting that 3 replicas, or 3 instances of the Pod, are running at any given time. The Pods are created using the Pod Template defined in **spec.template**. A nested object, such as the Pod being part of a Deployment, retains its **metadata** and **spec** and loses the **apiVersion** and **kind** - both being replaced by **template**. In **spec.template.spec**, we define the desired state of the Pod. Our Pod creates a single container running the **nginx:1.15.11** image from [Docker Hub](https://hub.docker.com/_/nginx).

Once the Deployment object is created, the Kubernetes system attaches the **status** field to the object and populates it with all necessary status fields.

## Pods

A [**Pod**](https://kubernetes.io/docs/concepts/workloads/pods/) is the smallest and simplest Kubernetes object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, which:

* + Are scheduled together on the same host with the Pod
  + Share the same network namespace, meaning that they share a single IP address originally assigned to the Pod
  + Have access to mount the same external storage (volumes).



**Single- and Multi-Container Pods**

Pods are ephemeral in nature, and they do not have the capability to self-heal themselves. That is the reason they are used with controllers which handle Pods' replication, fault tolerance, self-healing, etc. Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc. We attach a nested Pod's specification to a controller object using the Pod Template, as we have seen in the previous section.

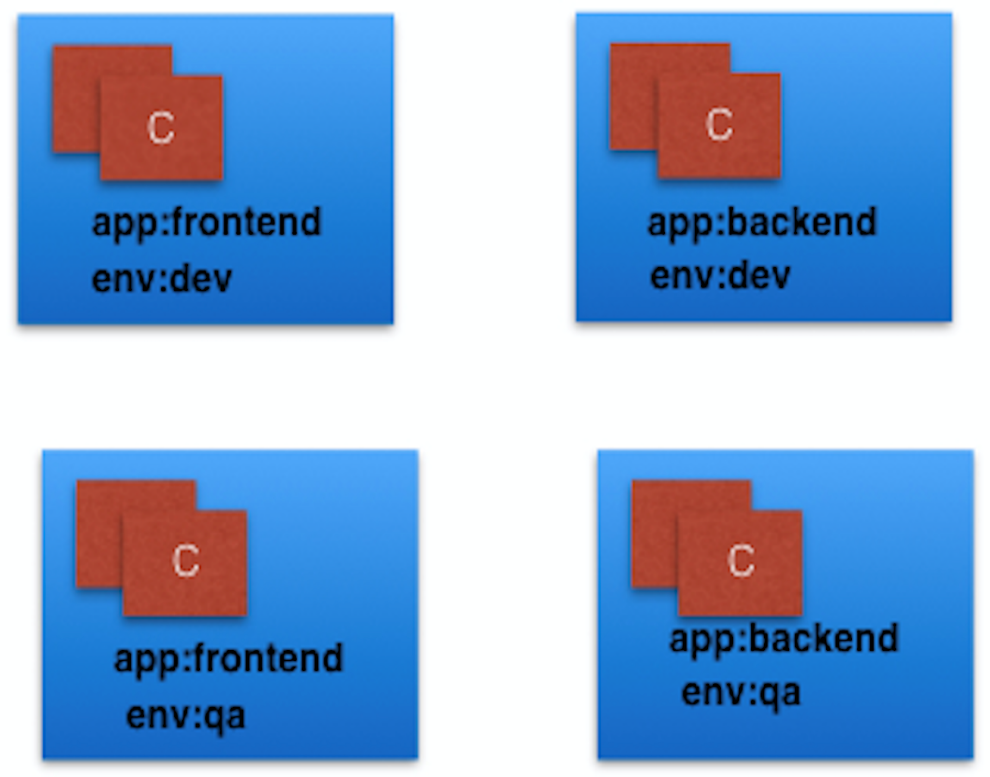
Below is an example of a Pod object's configuration manifest in **YAML** format:

**apiVersion: v1**  
**kind: Pod**  
**metadata:**  
**name: nginx-pod**  
**labels:**  
**app: nginx**  
**spec:**  
**containers:**  
**- name: nginx**  
**image: nginx:1.15.11**  
**ports:**  
**- containerPort: 80**

The **apiVersion**field must specify **v1** for the **Pod** object definition. The second required field is **kind** specifying the **Pod** object type. The third required field **metadata**, holds the object's name and label. The fourth required field **spec** marks the beginning of the block defining the desired state of the Pod object - also named the **PodSpec**. Our Pod creates a single container running the **nginx:1.15.11** image from [Docker Hub](https://hub.docker.com/_/nginx). The **containerPort** field specifies the container port to be exposed by Kubernetes resources for inter-application access or external client access.

## Labels

[**Labels**](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) are **key-value pairs** attached to Kubernetes objects (e.g. Pods, ReplicaSets, Nodes, Namespaces, Persistent Volumes). Labels are used to organize and select a subset of objects, based on the requirements in place. Many objects can have the same Label(s). Labels do not provide uniqueness to objects. Controllers use Labels to logically group together decoupled objects, rather than using objects' names or IDs.



**Labels**

In the image above, we have used two Label keys: **app** and **env**. Based on our requirements, we have given different values to our four Pods. The Label **env=dev** logically selects and groups the top two Pods, while the Label **app=frontend** logically selects and groups the left two Pods. We can select one of the four Pods - bottom left, by selecting two Labels: **app=frontend** **AND** **env=qa**.

## Label Selectors

Controllers use [**Label Selectors**](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) to select a subset of objects. Kubernetes supports two types of Selectors:

* + **Equality-Based Selectors**  
    Equality-Based Selectors allow filtering of objects based on Label keys and values. Matching is achieved using the **=**, **==** (equals, used interchangeably), or **!=** (not equals) operators. For example, with **env==dev** or **env=dev** we are selecting the objects where the **env** Label key is set to value **dev**.
  + **Set-Based Selectors**  
    Set-Based Selectors allow filtering of objects based on a set of values. We can use **in**, **notin** operators for Label values, and **exist/does not exist** operators for Label keys. For example, with **env in (dev,qa)** we are selecting objects where the **env** Label is set to either **dev** or **qa**; with **!app** we select objects with no Label key **app**.



**Selectors**

## ReplicationControllers

Although no longer a recommended controller, a **[ReplicationController](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/" \t "_blank)** ensures a specified number of replicas of a Pod is running at any given time, by constantly comparing the actual state with the desired state of the managed application. If there are more Pods than the desired count, the replication controller randomly terminates the number of Pods exceeding the desired count, and, if there are fewer Pods than the desired count, then the replication controller requests additional Pods to be created until the actual count matches the desired count. Generally, we do not deploy a Pod independently, as it would not be able to re-start itself if terminated in error because a Pod misses the much desired self-healing feature that Kubernetes otherwise promises. The recommended method is to use some type of a controller to run and manage Pods.

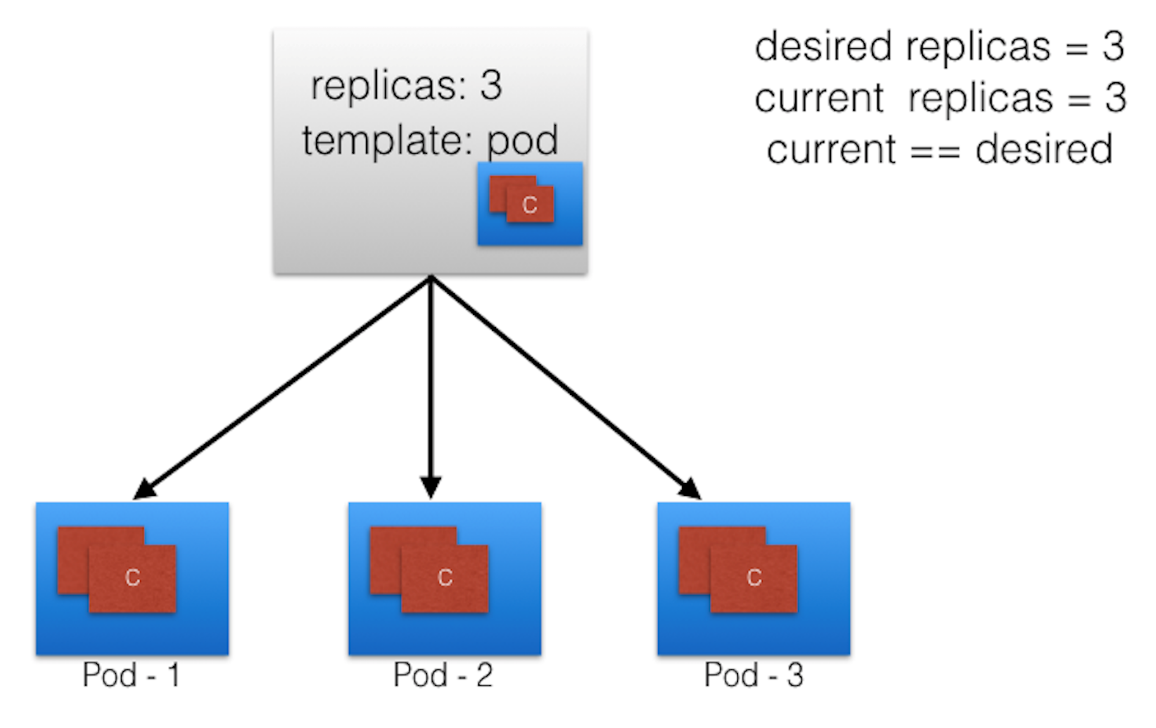
The default recommended controller is the [**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) which configures a **[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/" \t "_blank)** controller to manage Pods' lifecycle.

## ReplicaSets I

A **[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/" \t "_blank)** is, in part, the next-generation ReplicationController, as it implements the replication and self-healing aspects of the ReplicationController. ReplicaSets support both equality- and set-based Selectors, whereas ReplicationControllers only support equality-based Selectors.

With the help of a ReplicaSet, we can scale the number of Pods running a specific application container image. Scaling can be accomplished manually or through the use of an [autoscaler](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/" \t "_blank).

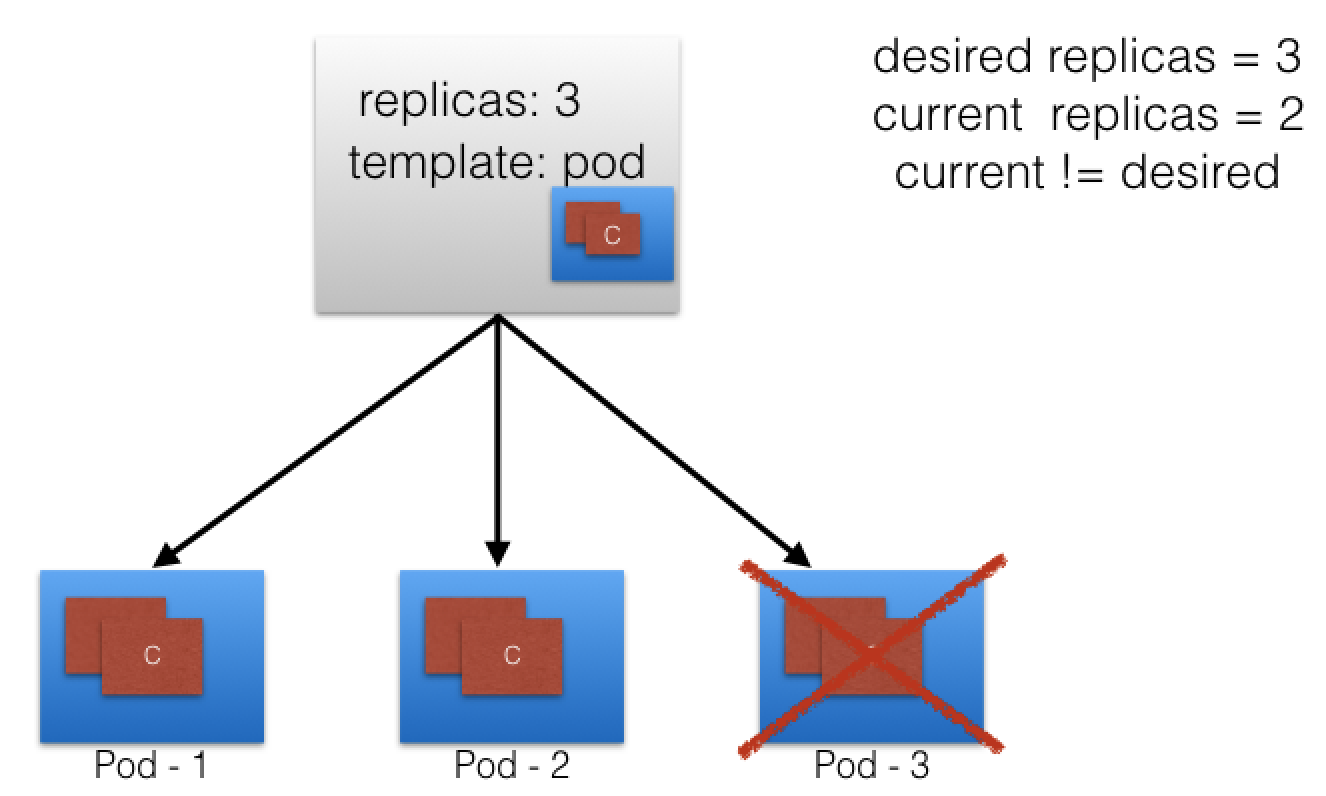
Below we graphically represent a ReplicaSet, with the replica count set to 3 for a specific Pod template. Pod-1, Pod-2, and Pod-3 are identical, running the same application container image, being cloned from the same Pod template. For now, the current state matches the desired state.



**ReplicaSet (Current State and Desired State Are the Same)**

## ReplicaSets II

Let's continue with the same ReplicaSet example and assume that one of the Pods is forced to unexpectedly terminate (due to insufficient resources, timeout, etc.), causing the current state to no longer match the desired state.



**ReplicaSet (Current State and Desired State Are Different)**

## ReplicaSets III

The ReplicaSet detects that the current state is no longer matching the desired state and triggers a request for an additional Pod to be created, thus ensuring that the current state matches the desired state.



**ReplicaSet (Creating a Pod to Match Current State with Desired State)**

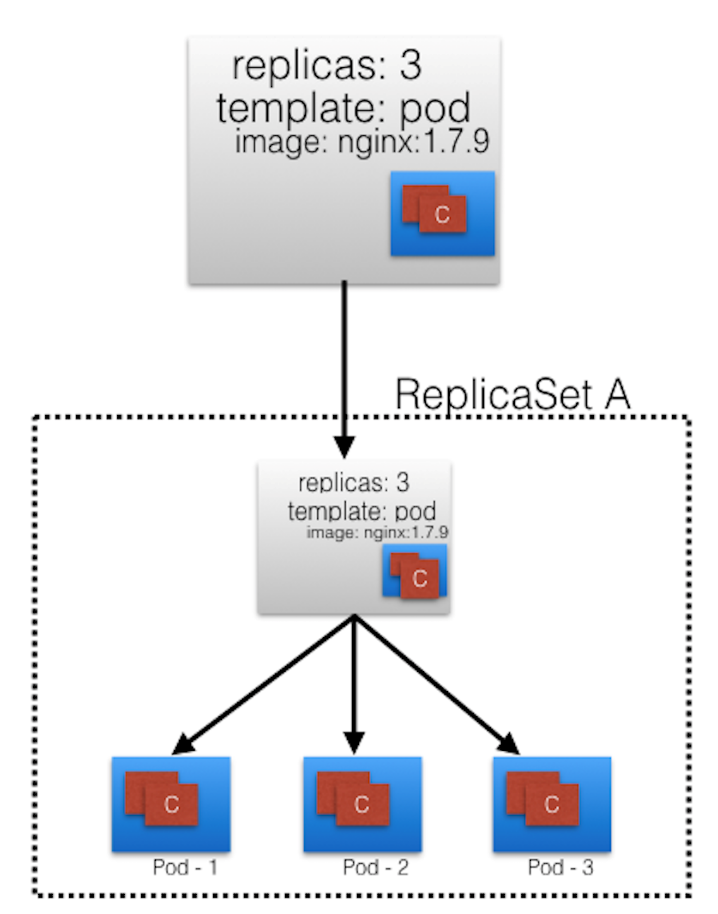
ReplicaSets can be used independently as Pod controllers but they only offer a limited set of features. A set of complementary features are provided by Deployments, the recommended controllers for the orchestration of Pods. Deployments manage the creation, deletion, and updates of Pods. A Deployment automatically creates a ReplicaSet, which then creates a Pod. There is no need to manage ReplicaSets and Pods separately, the Deployment will manage them on our behalf.

We will take a closer look at Deployments next.

## Deployments I

[**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the master node's controller manager, and as a controller it also ensures that the current state always matches the desired state. It allows for seamless application updates and rollbacks through **rollouts** and **rollbacks**, and it directly manages its ReplicaSets for application scaling.

In the following example, a new **Deployment** creates **ReplicaSet** **A** which then creates **3 Pods**, with each Pod Template configured to run one **nginx:1.7.9** container image. In this case, the **ReplicaSet A** is associated with **nginx:1.7.9** representing a state of the **Deployment**. This particular state is recorded as **Revision 1**.



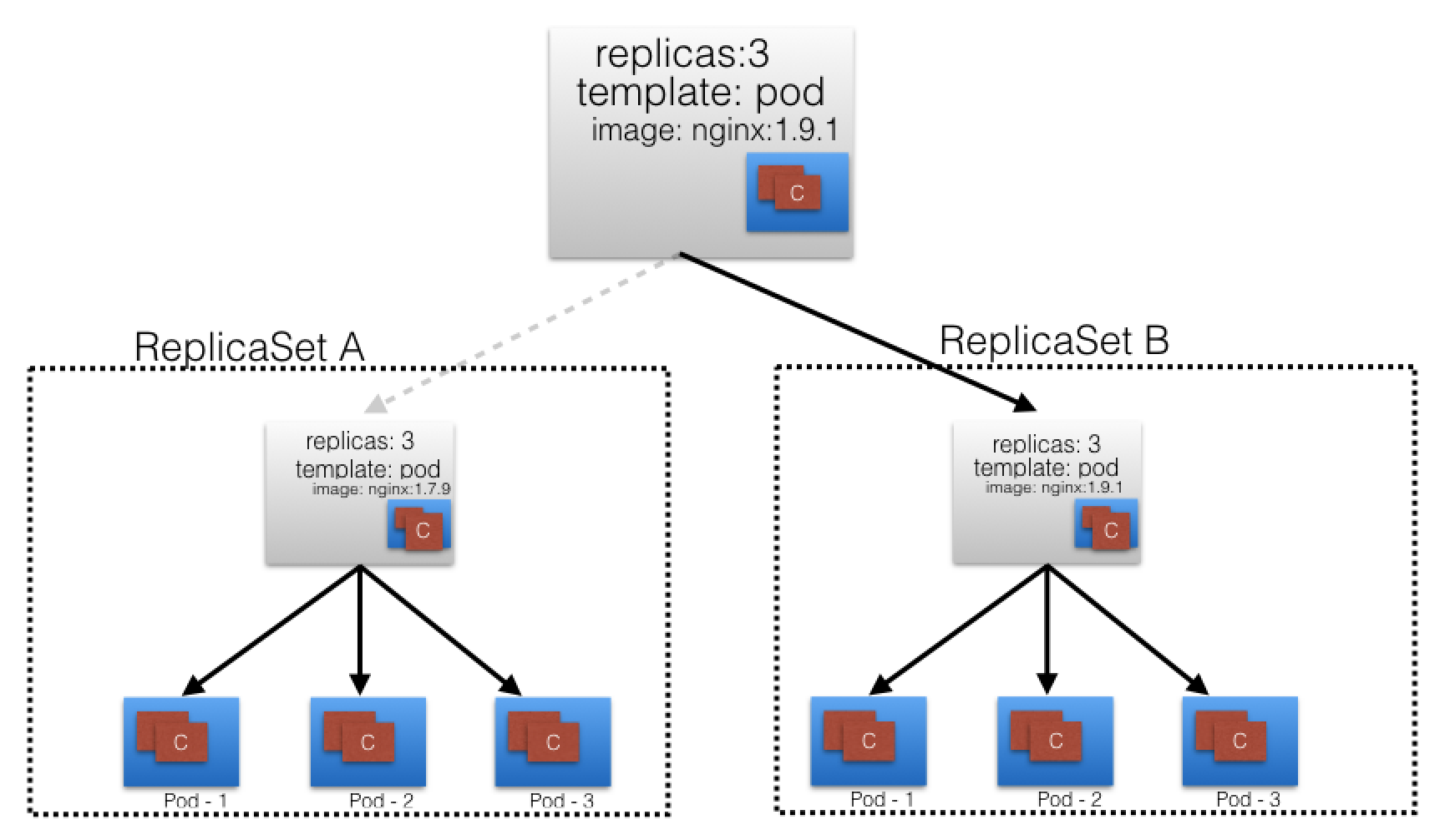
**Deployment (ReplicaSet A Created)**

## Deployments II

In time, we need to push updates to the application managed by the Deployment object. Let's change the Pods' Template and update the container image from **nginx:1.7.9**to **nginx:1.9.1**. The **Deployment** triggers a new **ReplicaSet B** for the new container image versioned **1.9.1** and this association represents a new recorded state of the **Deployment**, **Revision 2**. The seamless transition between the two ReplicaSets, from **ReplicaSet A** with 3 Pods versioned **1.7.9** to the new **ReplicaSet B** with 3 new Pods versioned **1.9.1**, or from **Revision 1** to **Revision 2**, is a Deployment **rolling update**.

A **rolling update** is triggered when we update specific properties of the Pod Template for a deployment. While updating the container image, container port, volumes, and mounts would trigger a new Revision, other operations like scaling or labeling the deployment do not trigger a rolling update, thus do not change the Revision number.

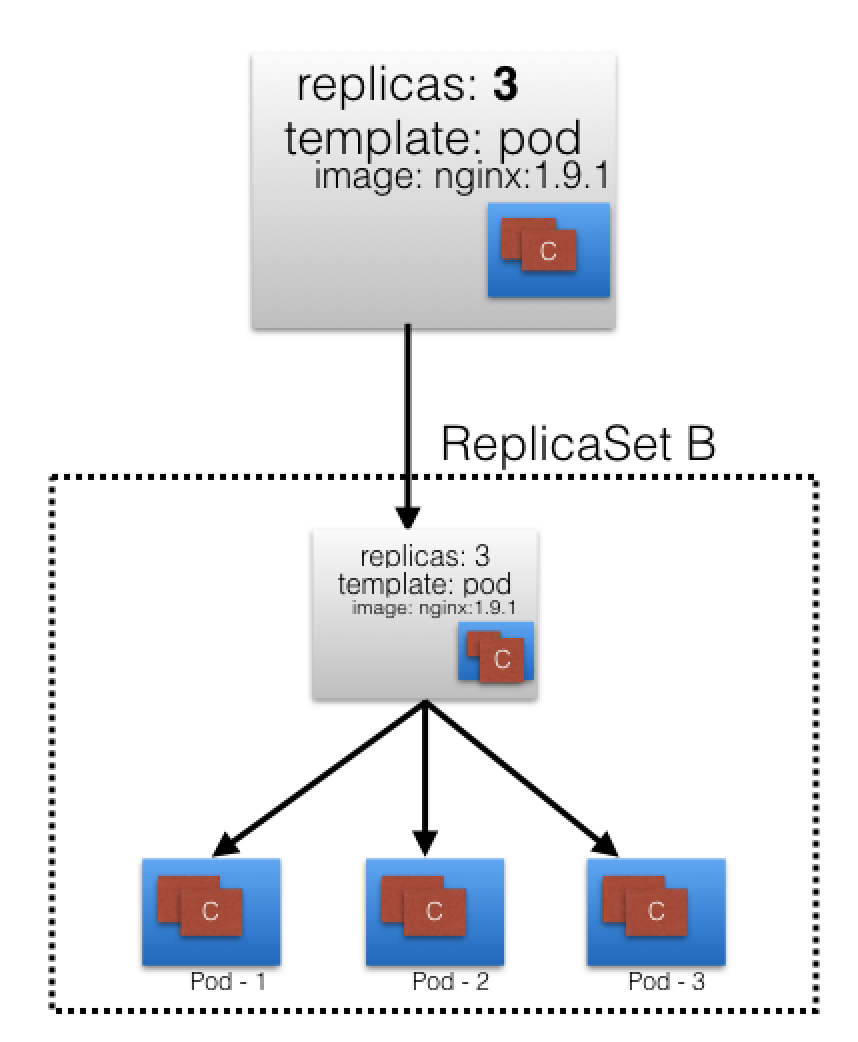
Once the rolling update has completed, the **Deployment** will show both **ReplicaSets A** and **B**, where **A** is scaled to 0 (zero) Pods, and **B** is scaled to 3 Pods. This is how the Deployment records its prior state configuration settings, as **Revisions**.



**Deployment (ReplicaSet B Created)**

## Deployments III

Once **ReplicaSet B** and its **3 Pods** versioned **1.9.1** are ready, the **Deployment** starts actively managing them. However, the Deployment keeps its prior configuration states saved as Revisions which play a key factor in the **rollback** capability of the Deployment - returning to a prior known configuration state. In our example, if the performance of the new **nginx:1.9.1** is not satisfactory, the Deployment can be rolled back to a prior Revision, in this case from **Revision 2** back to **Revision 1** running **nginx:1.7.9** once again.



**Deployment Points to ReplicaSet B**

## Namespaces

If multiple users and teams use the same Kubernetes cluster we can partition the cluster into virtual sub-clusters using [**Namespaces**](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/). The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.

To list all the Namespaces, we can run the following command:

**$ kubectl get namespaces**  
**NAME              STATUS       AGE**  
**default           Active       11h**  
**kube-node-lease   Active       11h**  
**kube-public       Active       11h**  
**kube-system       Active       11h**

Generally, Kubernetes creates four Namespaces out of the box: **kube-system**,**kube-public**,**kube-node-lease**, and **default.** The **kube-system**Namespace contains the objects created by the Kubernetes system, mostly the control plane agents. The **default**Namespace contains the objects and resources created by administrators and developers, and objects are assigned to it by default unless another Namespace name is provided by the user. **kube-public**is a special Namespace, which is unsecured and readable by anyone, used for special purposes such as exposing public (non-sensitive) information about the cluster. The newest Namespace is **kube-node-lease** which holds node lease objects used for node heartbeat data. Good practice, however, is to create additional Namespaces, as desired, to virtualize the cluster and isolate users, developer teams, applications, or tiers.

Namespaces are one of the most desired features of Kubernetes, securing its lead against competitors, as it provides a solution to the multi-tenancy requirement of today's enterprise development teams.

[**Resource Quotas**](https://kubernetes.io/docs/concepts/policy/resource-quotas/)help users limit the overall resources consumed within Namespaces, while **[LimitRanges](https://kubernetes.io/docs/concepts/policy/limit-range/" \t "_blank)** help limit the resources consumed by Pods or Containers in a Namespace. We will briefly cover quota management in a later chapter.

## Deployment Rolling Update and Rollback (Demo)

[Video](LinuxFoundationXLFS158x-V000800_DTH.mp4)  
[TExt](Ch%208-v01%20-%20Deployment%20Rolling%20Update%20and%20Rollback-en.txt)