Kubernetes Glossary and Terminology

The Primary responsibility of Kubernetes is container orchestration. This means making sure that all the containers that execute various workloads are scheduled to run on physical or virtual machines. The containers must be packed efficiently and follow the constraints of the deployment environment and the cluster configuration. In addition, Kubernetes must keep an eye on all running containers and dead, unresponsive, or otherwise unhealthy containers. Kubernetes provides many more capabilities that you will learn about in the following chapters. In this section, the focus is on containers their orchestration.

Physical machines, virtual machines, and Containers

It all starts and ends with hardware. In order to run your workloads, you need some real hardware provisioned. That includes actual physical machines, with certain compute capabilities (CPUs or cores), memory, and some local persistent storage (spinning disks or SSDs). In addition, you will need some shared persistent storage and networking to hook up all these machines so they can find and talk to each other. At this point, you can run multiple virtual machines on the physical machines or stay at the bare-metal lever (no virtual machines). Kubernetes can be deployed on a bare-metal cluster (real hardware) or on a cluster of virtual machines. Kubernetes in the run can orchestrate the containers it manages directly on bare-metal or virtual machines. In theory, a Kubernetes Cluster can be composed of a mix of bare-metal and virtual machines, but this is not very common.

The benefits of containers
Containers represent a true paradigm shift in the development and operation of large, complicated software systems. Here are some of the benefits compared to more traditional models:

- Agile application creation and deployment
- Continuous development, integration, and deployment
- Dev and ops separation of concerns
- Environmental consistency across development, testing, and production
- Cloud-and OS-distribution portability
- Application-centric management
- Loosely Coupled, distributed, elastic, liberated microservices
- Resource isolation
- Resource utilization

Containers in the cloud

Containers are ideal to package microservices because, while providing isolation to the microservices, they are very lightweight, and you don’t incur a lot of overhead when deploying many microservices as you do with virtual machines. That makes containers ideal for cloud deployment, where allocating a whole virtual machine for each microservice would be cost-prohibitive.

All major cloud providers, such as Amazon AWS, Google’s GCE Microsoft’s Azure, and even Alibaba Cloud, provide container-hosting services these days. Google’s GKE has always been based on Kubernetes. AWS ECS is based on their own orchestration solution, Microsoft Azure’s container service was based on Apache Mesos. Kubernetes can be deployed on all cloud platforms, but it wasn't deeply integrated with other services until today. But at the end of 2017, all cloud providers announced direct support for Kubernetes. Microsoft launched AKS, AWS released EKS, and Alibaba Cloud started working on a Kubernetes controller manager to integrate Kubernetes seamlessly.

Cattle versus pets

In the olden days, when Systems were small, each server had a name. Developers' users knew exactly what software was running on each machine. I remember that, in many of the companies I worked for, we had multi-day discussions to decide on a naming theme for our servers, For example, Composers and Greek mythology characters were popular choices. Everything was very cozy. You treated your servers like beloved pets. When a server died, it was a major crisis. Everybody scrambled to figure out where to get another server, what was even running on the dead server, and how to get it working on the new server. If the server stored some important data, then hopefully you had an up-to-date backup and maybe you’d even be able to recover it.

Obviously, that approach doesn’t scale. When you have a few tens or hundreds of servers, you must start treating them like cattle. You think about the collective and not individuals. You may still have some pets, your web servers are just cattle.

Kubernetes takes the cattle approach to the extreme and tasks full responsibility for allocating containers to specific machines. You don’t need to interact with individual machines (nodes) most of the time. This works best for the stateless workload. For stateful applications, the situation is a little different, but Kubernetes provides a solution called Statefulset, which we’ll discuss soon.
In this section, we covered the idea of container orchestration and discussed the relationships between hosts (physical or virtual) and containers, as well as the benefits of running containers in the cloud, and finished with a discussion about cattle versus pets. In the following section, we will get to know the world of Kubernetes and learn its concepts and terminology.

Kubernetes concepts

In this section, I’ll briefly introduce many important Kubernetes concepts and give you some context as to why they are needed and how they interact with other concepts. The goal is to get familiar with these terms and concepts. Later, we will see how these concepts are woven together and organized into API group and resource categories to achieve awesomeness. You can consider many of these concepts as a building blocks. Some of the concepts, such as nodes and masters, are implemented as a set of Kubernetes components. These components are at a different abstraction level, and I discuss them in detail in a dedicated section, Kubernetes components.

Here is the famous Kubernetes architecture diagram:

Cluster
A cluster is a collection of computing, storage, and networking resource that Kubernetes uses to run the various workload that comprises your system. Note that your entire system may consist of multiple clusters. We will discuss this advanced use case of the federation in detail later.

Node
A node is a single host. It may be a physical or virtual machine. Its job is to run pods, which we will look at in a moment. Each Kubernetes node runs several Kubernetes components, such as a kubelet and Kube Proxy. Nodes are managed by a Kubernetes master. The Nodes are the worker bees of Kubernetes and shoulder all the heavy lifting. In the past, they were called minions. If you have read some old documentation or articles, don't get confused. Minions are nodes.

Master
The master is the control plane of Kubernetes. It consists of several components, such as an API server, a scheduler, and a controller manager. The master is responsible for the global, cluster-lever scheduling of pods and the handing of events. Usually, all the master components are set up on a single host. When considering high-availability scenarios or very large clusters, you will want to have master redundancy. I will discuss highly available clusters in detail in chapter 4, High Availability Reliability.

Pod
A pod is the unit of work in Kubernetes. Each pod contains one or more containers. Pods are always scheduled together (that is, they always run on the same machine). All the containers in pod have the same IP address and port space; they can communicate using localhost or standard interprocess communication. In addition, all the containers in a pod can have access to shared local storage on the node hosting the pod. The shared storage can be mounted on each container. Pods are an important feature of Kubernetes. It is possible to run multiple applications inside a single Docker container by having something such as supervisors as the main Docker application that runs multiple processes, but this practice is often frowned upon for the following reasons:

- Transparency: Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of convenient functionalities for users.
- Decoupling software dependencies: The individual containers may be versioned, rebuilt, and redeployed independently. Kubernetes may even support live updates of individual containers someday.
- Ease of use: Users don't need to run their own Process managers, worry about signal and exit-code propagation, and so on.
- Efficiency: Because the infrastructure takes on more responsibility, containers can be more lightweight.

Pods provide a great solution for managing groups of closely related containers that depend on each other and need to cooperate on the same host to accomplish their purpose. It's important to remember that pods are considered ephemeral, throwaway entities that can be discarded and replaced at will. Any pod storage is destroyed with its pod. Each pod gets a unique ID (UID), so you can still distinguish between them if necessary.

Label

Labels are key-value pairs that are used to group together sets of objects, very often pods. This is important for several other concepts, such as replication controllers, replica sets, and services that operate on a dynamic group of objects and need to identify the members of the group. There is an NxN relationship between object and label. Each object may have multiple labels, and each label may be applied to different objects. There are certain restrictions on labels by design. Each label on an object must have a unique key. The label key must adhere to a strict syntax. It has two parts: prefix and name. The prefix is optional. If it exists, then it is separated from the name by a forward slash (/) and it must be a valid DNS subdomain. The prefix must be 253 characters long at most. The name is mandatory and must be 63 characters long at most. Names must start and end with an alphanumeric character (a-z, A-Z,0-9) and contain only alphanumeric characters, dots, dashes, and underscores. Values follow the same restrictions as names. Note that labels are dedicated to identifying objects and to attaching arbitrary metadata to objects. This is what annotations are for (see the following section).

Annotations

Annotation lets you associate arbitrarily with Kubernetes objects. Kubernetes just stores the annotation and makes their metadata available. Unlike Labels, they don't have strict restrictions about allowed characters and size limits.

In my experience, you always need such metadata for complicated systems, and it is nice that Kubernetes recognizes this need and provides it out of the box so you don't have to come up with your own separate metadata store and map objects to their metadata.

We've covered most, if not all, of Kubernetes's concepts; there are a few more I mentioned briefly. In the next section, we will continue our journey into Kubernetes's Architecture by looking into its design motivations, the internals, and its implementation, and even pick at the source code.

Label selectors

Label selectors are used to select objects based on their labels. Equality-based selectors specify a key name and a value. There are two operators, = (or ==) and !=, to denote equality or inequality based on the value. For example:
role = webserver
This will select all objects that have that label key and value.
Label selectors can have multiple requirements separated by a comma. For example:
role = webserver, application != foo
Set-based selectors extend the capabilities and allow selection based on multiple values: role in (webserver, backend)

Replication controllers and replica sets

Replication controllers and replica sets both manage a group of pods identified by a label selector and ensure that a certain number is always up and running. The main difference between them is that replication controller tests for membership by name equality and replica sets can use set-based selection. Replica sets are the way to go, as they are a superset of replication controllers. I expect replication controllers to be deprecated at some point.

Kubernetes guarantees that you will always have the same number of pods running that you specified in a replication controller or a replica set. Whenever the number drops because of a problem with the hosting node or the pod itself, Kubernetes will fire up new instances. Note that if you manually start pods and exceed the specified number, the replication controller will kill extra pods.
Replication controller used to be central to many workflows, such as rolling updates and running one-off jobs. As Kubernetes evolved, it introduced direct support for many of these workflows, with dedicated objects such as Deployment, Job, and DaemonSet. We will meet them all later.

Services
Services are used to expose a certain functionality to users or other services. They usually encompass a group of pods, usually identified by–you guessed it–a label. You can have services that provide access to external resources, or to pods, you control directly at the virtual IP level. Native Kubernetes services are exposed through convenient endpoints. Note that the service operates at layer 3 (TCP/UDP). Kubernetes 1.2 added the Ingress Object, which provides access to HTTP object–more on that later. Services are published or discovered through one of two mechanisms: DNS or environment variables. Services can be load-balanced by Kubernetes, but developers can choose to manage load balancing themselves in the case of services that use external or require special treatment.

There are many gory details associated with IP addresses virtual IP addresses, and port spaces. We will discuss them in-depth in a future chapter.

Volume
Local storage on the pod is ephemeral and goes away with the pod. Sometimes that's all you need if the goal is just to exchange data between containers of the node, but sometimes it's important for the data to outlive the pod, or it's necessary to share data between pods. The volume concept supports that need. Note that, while Docker has volume concepts too, it is quite limited (although it is getting more powerful). Kubernetes uses its own separate volumes. Kubernetes also supports additional container types, such as rkt, so it can't rely on Docker volumes, even in principle.

There are many volume types. Kubernetes currently directly supports many volume types, but the modern approach for extending Kubernetes with more volume types is through the Container Storage Interface (CSI), which I'll discuss in detail later. The emptyDir volume type mounts a volume on each container that is backed by default by whatever is available on the hosting machine. You can request a memory medium if you want. This storage is deleted when the pod is terminated for any reason. There are many volume types for specific cloud environments, various networked filesystems, and even Git repositories. An interesting volume type is the persistentDiskClaim, which abstracts the details a little bit and uses the default persistent storage in your environment (typically in a cloud provider).

Statefulset
Pods come and go, and you care about their data, then you can use persistent storage. That's all good. But sometimes you might want Kubernetes to manage a distributed data store, such as Kubernetes or MySQL Galera. These clustered stores keep the data distributed across uniquely identified nodes. You can't model that with regular pods and services. Enter Statefulset. If you remember, earlier I discussed treating servers as pets or cattle and how cattle is the way to go. Well, Statefulset sits somewhere in the middle. Statefulset ensures (similar to a replication set) that a given number of pet with unique identities are running at any given time. The pets have the following properties:

- A stable hostname, available in DNS
- An ordinal index
- Stable storage linked to the ordinal and hostname

Statefulset can help with peer discovery, as well as adding or removing pets.

Secrets

Secrets are small objects that contain sensitive information, such as credentials and tokens. They are stored in etcd, are accessible by the Kubernetes API server, and can be mounted as files into pods (using dedicated secret volume that piggyback on regular data volume) that need access to them. The same secret can be mounted into multiple pods. Kubernetes itself creates secrets for its components, and you can create your own secret. Another approach is environment variables. Note that secrets in a pod is always stored in memory (tmpfs, in the case of mounted secrets) for better security.

Names

Each object in Kubernetes is identified by a UID and a name. The name is used to refer to the object in API calls. Names should be up to 253 characters long and lowercase alphanumeric characters, dashes (-), and dots (.). If you delete an object, you can create another object with the same name as the deleted object, but the UIDs must be unique across the life cycle of the cluster. The UIDs are generated by Kubernetes, so you don't have to worry about that.

Namespace

A namespace is a virtual cluster. You can have a single physical cluster that contains multiple virtual clusters segregated by namespace. Each virtual cluster is totally isolated from the other virtual clusters, and they can only communicate through public interfaces. Note that node objects and persistent volumes don't live in a namespace. Kubernetes may schedule pods from different namespaces to run on the same node. likewise, pods from different namespaces can use the same persistent storage.

When using namespace, you have to consider network policies and resource quotas to ensure proper access and distribution of the physical cluster resources.

Diving into Kubernetes architecture in-depth

Kubernetes has very ambitious goals. It aims to manage and simplify the orchestration, deployment, and management of distributed systems across a wide range of environments and cloud provides. It provides many capabilities and services that should work across all that diversity while evolving and remaining simple enough for more mortals to use. This is a tall order. Kubernetes architecture this by following a crystal-clear, high-level design and using a well-thought-out architecture that promotes extensibility and pluggability. Many parts of Kubernetes are still hardcoded or environmentally aware, but the trend is to refactor them into plugins and keep the core generic and abstract. In this section, we will peel Kubernetes like an onion, starting with the various distributed systems design patterns and how Kubernetes supports them, then go over the mechanics of Kubernetes, including its set of APIs, and then take a look at actual components that comprise Kubernetes. Finally, we will take a quick tour of the source-code tree to gain even better insight into the structure of Kubernetes itself.
At the end of this section, you will have a solid understanding of the Kubernetes architecture and implementation, and why certain design decisions were made.
Distributed systems design patterns

All happy (working) distributed systems are alike, to paraphrase Tolstoy in Anna Karenina. This means that to function properly, all well-designed distributed systems must follow some best practices and principles. Kubernetes doesn't want to be just a management system. It wants to support and enable best practices and provide high-level service to developers and administers. Let's look at some of these design patterns.

Sidecar pattern

The sidecar pattern is about co-locating another container in a pod in addition to the main application container. The application container is unaware of the sidecar container and just goes about its business. A great example is a central logging agent. Your main container can just log to stdout, but the sidecar container will send all along to a central logging service where they will be aggregated with the logs from the entire system. The benefits of using a sidecar container versus adding central logging to the main application container are enormous. First, applications are no longer burdened with central logging, which could be a nuisance. If you want to upgrade or change your container and deploy or switch to a totally new provider, you just need to update the sidecar container and deploy it. None of your application containers change, so you can't break them by accident.

Ambassador pattern

The ambassador pattern is about is represents a remote service as if it were local and possibly enforcing a policy. A good example of the ambassador pattern is if you have a Redis cluster with one master for writes and many replicas for reads. A local ambassador container can serve as a proxy and expose Redis to the main application container on the localhost. The main application container simply connects to Redis on localhost: 6379 (Redis's default port), but it connects to the ambassador running in the same pod, which filters the requests, sends write requests to the real Redis master, and read requests randomly to one of the read replicas. Just as we saw with the sidecar pattern, the main application has no idea what's going on. They can help a lot when testing against a real local Redis. Also, If the Redis cluster configuration changes, only the ambassador need to be modified; the main application remains blissfully unaware.

Adapter pattern

The adapter pattern is about standardizing output from the main application container. Consider the case of a service that is being rolled out incrementally: It may generate reports in a format that doesn't conform to the previous version. Other services and applications that consume that output haven't been upgraded yet. As adapter container can be deployed in the same pod with the new application container and can alter its output to match the old version until all consumers have been upgraded. The adapter container shares the filesystem with the main application container, so it can watch the local filesystem, and whenever the new application writes something, it immediately adapts it.

Multinode patters

The single-node patterns are all supported directly by Kubernetes through pods. Multinode patterns, such as leader election, work queues, and scatter-gather, are not supported directly, but composing pods with standard interfaces to accomplish them is a viable approach with Kubernetes.

The Kubernetes APIs

If you want to understand the capabilities of a system and what it provides, you must pay a lot of attention to its APIs. These APIs provide a comprehensive view of what you can do with the system as a user. Kubernetes exposes several sets of REST APIs for different purposes and audiences through API groups. Some of the APIs are used primarily by tools and some can be used directly by developments. An important fact regarding the APIs is that they are under constant development. The Kubernetes developers keep it manageable by trying to extend it (by adding new objects and new fields to existing objects) and avoid renaming or dropping existing objects and fields. In addition, all API endpoints are versioned and often have an alpha or beta notation too. For example:
/api/v1
/api/v2a1pha1
You can access the API through the kubectl CLI, client libraries, or directly through REST API calls. There is an authentication and authorization mechanism that we will explore in a later chapter. If you have the right permissions, you can list, view create, update, and delete various Kubernetes objects. At This point, let's glimpse the surface area of the APIs. The best way to explore these APIs is through API groups. Some API groups are enabled by default. Another group can be enabled/disabled via flags. For example, to disable the batch VI groups and enable the batch V2 alpha group, you can set the–runtime-config flag when running the API server as follows:
–runtime-config=batch/v1=false,batch/v2a1pha=true

The following resources are enabled by default, in addition to the core resources:

- DaemonSets
- Deployments
- HorizontalpodAutoscalers
- Ingress

- Jobs
- ReplicaSets

Resources categories

In addition to API groups, another useful classification of the available APIs is functionality. The Kubernetes API is huge, and breaking it down into categories helps a lot when you're trying to find your way around. Kubernetes defines the following resources categories:

- Workloads: The objects you use to manage and run containers on the cluster.
- Discovery and load balancing: The objects you use to expose your workloads to the world as externally accessible, load-balanced services.
- Config and Storage: The objects you use to initialize and configure your applications, and to persist data that is outside the container.
- Cluster: The objects that define how itself is configured; these are typically used only by cluster operators.
- Metadata: The objects you use to configure the behavior of other resources within the cluster, such as HorizontapodAutoscaler for scaling workload.

In the following subsections, I'll list the resources that belong to each group, along with the API group they belong to. I will not specify the version here because APIs move rapidly from alpha to beta to general availability (GA), and then from V1 to V2, and so on.

Workload API

The workloads API contains the following resources:

- Container: Core
- cronJob: Batch
- DaemonSet: Apps
- Deployment: Apps
- Job: Batch
- Pod: Core
- ReplicaSet: Apps
- ReplicationController: Core
- StatefulSet: Apps

Containers are created by the controller using pods. Pods Run containers and provide environmental dependencies, such as shared or persistent storage volumes, and configuration or secret data injected into the container.

Here is a detailed description of one of the most common operations, which gets a list of all the pods as a REST API:
GET /API/v1pods

It accepts various query parameters (all optional):

- Pretty: If true, the output is pretty printed
- label selector: A selector expression to limit the result
- watch: If true, this watches for changes and returns a stream of events
- resourcesVerion: Returns only events that occurred after that version
- timeout seconds: Timeout for the list or watch operation

Discovery and load balancing

By default, workloads are only accessible within the cluster, and they must be exposed externally using either a LoadBalancer or NodePort service. During development, the internally accessible workload can be accessed via a proxy through the API master using the kubectl proxy command:

- Endpoints: Core
- Ingress: Extension
- Service: Core

Config and storage

Dynamic configuration without redeployment is a cornerstone of Kubernetes and running complex distributed applications on your Kubernetes cluster:

- ConfigMap: Core
- Secret: Core
- PersistentVolumeClaim: Core
- StorageClass: Storage
- VolumeAttachment: Storage

Metadata

The metadata resource typically are embedded as subresources of the resources they configure. For example, a limit range will be part be of a pod configuration. You will not interact with these object directly most of the time. There are many metadata resources. You can find the complete list at https://kubernetes.io/docs/reference/ generated/Kubernetes-api/v1.10/#-storage-metadata-storage-,

Cluster

The resources in the cluster category are designed for by cluster operators as opposed to developers. There are many resources in this category as well. Here are some of the most important resources:

- Namespace: Core
- Node: Core
- PersistentVolume: Core
- ResourcesQuota: Core
- ClusterRole: RBAC
- NetworkPolicy: Networking

Kubernetes components

A Kubernetes cluster has several master components that are used to control the cluster, as well as node components that run on each cluster node. Let’s get to know all these components and they work together.

Master Components

The master components typically run on nodes, but in a highly available or very large cluster, they may be spread across multiple nodes.

Ver exposes the Kubernetes REST API. It easily scales horizontally as it stores all the data all data in the etcd cluster. The API server is the embodiment of the control plane.

Y reliable, distributed data store. Kubernetes uses it to store the entire cluster of all transient clusters, a single instance of etcd can on some node as all ster components. But for more substantial clusters, it is typical to have a three- five-node etcd cluster for redundancy and high availability.

Controller manager

A controller manager is a collection of various managers rolled up into one binary. The replication controller, the pod controller, the services controller, the controller, and others. All these managers watch over the state of the cluster the API and their job is to steer the cluster into the desired state.

D controller manager

Running in the cloud, Kubernetes allows cloud provides to integrate their platform purpose of managing nodes, routes, services, and volumes. The cloud provider code cts with the Kubernetes code. It replaces some of the functionality of the Kube Controller manager. When running Kubernetes with a cloud controller manager, you must Kube controller manager flag—cloud-provider to external. This will disable the loops that that cloud controller manager is taking over. The cloud controller manager was introduced in Kubernetes 1.6 and it is being used by multiple clouds provides already

Kube-scheduler

Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it requires considering multiple interacting factors, such as the following:

- Resources requirements
- Service requirements
- Hardware/software policy constraints
- Node affinity and anti-affinity specifications
- Pod attinity and antiffinity specifications
- Taints and toleration
- Data locality
- Deadlines

If you need some special scheduling logic not covered by the default Kube Scheduler, you can replace it with your own custom scheduler. You can also run your custom scheduler side by side with the default scheduler and have your custom schedule scheduler only a subset of the pods.

DNS

Since Kubernetes 1.3, a DNS service has been part of the Standard Kubernetes cluster. It is scheduled as a regular pod. Every service (except headless service) receivers a DNS name Pods can receive a DNS name too. This is very useful for automatic discovery.
Node components
Nodes in the cluster need a couple of components to interact with the cluster master components and to receive workloads to execute and update the cluster on their status.

Proxy

The Kube proxy dose low-level, network housekeeping each on each node. It reflects the Kubernetes service locally and can do TCP and UDP forwarding. It finds cluster IPs through environment variables or DNS.

Kubelet

The Kubelet is the Kubernetes representative on the node. If oversees communicating with the master components and manages the running pods. This includes the following actions:

- Downloading pod secrets from the API server
- Mounting volumes
- Running the pod's container (through the CRI or rkt)
- Reporting the status of the node each pod
- Running container liveness probes

In this section, we dug into the guts of Kubernetes, explored its architecture (from a very high-level perspective), and supported design patterns, through its APIs and the components used to control and manage the cluster. In the section, we will take a quick look at various runtimes that Kubernetes supports.

Kubernetes runtimes

Kubernetes originally only supported Docker as a container runtime engine. But that is no longer the case. Kubernetes now supports several different runtimes:

- Docker (through a CRI a shim)
- Rkt (direct integration to be replaced with rktlet)
- Cri-o
- Frakti (Kubernetes on the hypervisor, previously Hypernetes)
- rktlet (CRI implementation for rkt)
- cri-contained

A major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The Container Runtime Interface (CRI) enables this.

In this section, you'll get a closer look at the CRI and get to know the individual runtime engines. At the end of this section, you'll be able to make a well-informed decision about which runtime engine is appropriate for your use case and under what circumstances you may switch or even combine multiple in the same system.

Docker

Docker is, of course, the 800-pound gorilla of containers. Kubernetes was originally designed to manage only Docker containers. The multi-runtime capability was first introduced in Kubernetes 1.3 and the CRI in Kubernetes 1.5 Until then, Kubernetes Could only manage Docker containers.
If you are reading this book, I assume you're very familiar with Docker and what it brings to the table. Docker is enjoying tremendous popularity and growth, but there is also a lot of criticism directed toward it. Critics often mention the following concerns:

- security
- Difficulty setting up multi-container applications (in particular, networking)
- Development, monitoring, and logging
- Limitation of Docker container running one command
- Releasing half-baked features too fast

Docker is aware of the criticisms and has addressed some of these concerns. It particular Docker has invested in its Docker Swarm Product. Docker swarm is a Docker-native Orchestration Solution that Competes with Kubernetes. It is simpler to use than Kubernetes but it's not as powerful or mature.

Since Docker 1.12, swarm mode has been included in the Docker daemon natively, which upset some people because of its bloat and scope creep. The in turn made more people turn to CoreOS rkt as an alternative solution.

Since Docker 1.11, released in April 2016, has changed the way it runs containers. The runtime now uses containerd and runC To rum Open Container Initiative (OCI) images in containers:

Rkt

Rkt is a container manager from CoreOs (the developers of the CoreOS Linux distro, etcd, flannel, and more). The rkt runtime prides itself on its simplicity and a strong emphasis on security and isolation. It doesn't have a daemon like the Docker engine and relies on the OS init system, such as the system, to launch the rkt executable. Rkt can download images (both app container (appc) images and OCI images), verify them, and run them in containers. Its architecture is much simpler.

App Container

CoreOS started a standardization effort in December 2014 called apps. This included the standard image format (ACI), runtime, signing, and discovery. A few months later, Docker started its own standardization effort with OCI. At this point, it seems these efforts will converge. this is great as tools, images, and runtime will be able to interoperate freely. We're not there yet.

Cri-O
Cri-o is a Kubernetes incubator project. It is designed to provide an integration path between Kubernetes and OCI-compliant Container runtimes, such as Docker. this idea is that Cri-o will provide the following capabilities:

- Support multiple image formats, including the existing Docker image format
- Support multiple means of downloading images, including trust and image verification
- Container process life cycle management(managing image layers, overlaying filesystems, and so on)
- Container process life cycle management
- The monitoring and logging required to satisfy the CRI
- Resources isolation as required by the CRI

Then any OCI-compliant container can be plugged in and will be integrated with Kubernetes.
Rktnetes
Rktnetes is Kubernetes plus rkt as a runtime engine. Kubernetes is still in the process of abstracting away the runtime engine. Rktnetes is not really a separate product. From the outside, all it takes is running the Kubelet on each node with a couple of command-line switches.

Is rkt ready for use in production?

I don't have a lot of hands-on experience with rkt. However, it is used by Tectonic–the commercial CoreOS-based Kubernetes distribution. If you run a different type of cluster, I would suggest that you wait until rkt is integrated with Kubernetes through the CRI/rktle. There are some known issues you need to be aware of when using rkt as opposed to Docker with Kubernetes¬¬¬-for example, missing volumes created automatically, Kubectl's attach and get logs don't init container is not supported, among other issues.

Hyper Containers
Hyper containers are another option. A Hyper container has a lightweight VM (its own guest kernel) and it runs on bare metal. Instead of relying on Linux Cgroups for isolation, it relies on a hypervisor. This approach presents an interesting mix compared to the standard, bare-metal clusters that are difficult to set up public clouds where containers are deployed on heavyweight VMs.

Stackable
Stackable (previously called Hypernetes) is a multitenant distribution that uses Hyper container as well as some OpenStack components for authentication, persistent storage, and networking. Since containers don't share the host kernel, it is safe to run containers of different tenants on the same physical host. Stackable uses Frakti as its container runtime, of course.
In this section, we've covered the various runtime engines that Kubernetes supports, as well as the trend toward standardization and convergence. In the next section, we'll take a step back and look at the big picture, as well as how Kubernetes fits into CI/CD pipeline.

Continuous integration and deployment

Kubernetes is a great platform for running your microservice-based applications. But, at the end of the day, it is an implementation detail. Users, and often most developers, may not be aware that the system is deployed on Kubernetes. But Kubernetes can change the game and make things that were too difficult before possible.
In this section, we'll explore the CI/CD pipeline and what Kubernetes brings to the table. At the end of this section, You'll be able to design that takes advantage of Kubernetes properties, such as easy-scaling and development-production parity, to improve the productivity and robustness of your day-to-day development and deployment.