**DOCKER AND KUBERNETES**

**VIRTUAL MACHINE**

Defined as an emulation of the computersystems in computing. It is based on computer architectures. It gives the functionality of physical computers. The implementation of VM may consider specialize software, hardware, or a combination of both. A virtual machine (VM) is **a digital version of a physical computer**. Virtual machine software can run programs and operating systems, store data, connect to networks, and do other computing functions, and requires maintenance such as updates and system monitoring.

There are two types of Virtual Machine,

* Process virtual machines-These virtual machines are sometimes called application virtual machines or Managed Runtime Environments (MREs).
* System virtual machines-These kinds of VMs are completely virtualized to replace a real machine.

Virtual machines (VMs) **allow a business to run an operating system that behaves like a completely separate computer in an app window on a desktop.** It can run in a window as a separate computing environment, often to run a different operating system—or even to function as the user's entire computer experience—as is common on many people's work computers.

**VMs have several advantages:**

* Lower hardware costs. Many organizations don't fully utilize their hardware resources. ...
* Quicker Desktop Provisioning and Deployment. Deploying a new physical server often takes numerous time-consuming steps. ...
* Smaller Footprint. ...
* Enhanced Data Security. ...
* Portability. ...
* Improved IT Efficiency.

**DOCKER**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker is **a software platform that allows you to build, test, and deploy applications quickly.** Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime.

It is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called [containers](https://aws.amazon.com/containers/) that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.

**Docker Features**

* Easy and Faster Configuration
* Increase productivity
* Application Isolation
* Swarm
* Routing Mesh
* Services
* Security Management

Easy and Faster Configuration

This is a key feature of docker that helps us to configure the system easily and faster. As Docker can be used in a wide variety of environments, the requirements of the infrastructure are no longer linked with the environment of the application.

Increase Productivity

By easing technical configuration and rapid deployment of application.  It has increase productivity. Docker not only helps to execute the application in isolated environment but also it has reduced the resources.

Application Isolation

It provides containers that are used to run applications in isolation environment. Each container is independent to another and allows us to execute any kind of application.

Swarm

It is a clustering and scheduling tool for Docker containers. Swarm uses the Docker API as its front end, which helps us to use various tools to control it.

Routing Mesh

It routes the incoming requests for published ports on available nodes to an active container. This feature enables the connection even if there is no task is running on the node.

Services

Services is a list of tasks that lets us specify the state of the container inside a cluster. Each task represents one instance of a container that should be running and Swarm schedules them across nodes.

Security Management

It allows us to save secrets into the swarm itself and then choose to give services access to certain secrets.

Diagram

Description automatically generated**Docker Architecture**

Docker includes  three main components that are **Docker Client, Docker Host,** and **Docker Registry**

Docker Client

Docker client uses **commands** and **REST APIs** to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

Docker Client uses Command Line Interface (CLI) to run the following commands –

docker build

docker pull

docker run

Docker Host

Docker Host is used to provide an environment to execute and run applications. It contains the docker daemon, images, containers, networks, and storage

Docker Registry

Docker Registry manages and stores the Docker images.

There are two types of registries in the Docker -

**Public Registry -** Public Registry is also called as **Docker hub**

**Private Registry -** It is used to share images within the enterprise.

**Dockerization process**

**Identify the application components**: Before Dockerizing an application, it's important to identify its components, including the application code, dependencies, libraries, and any other assets required to run the application.

**Choose a base image**: Next, you need to choose a base image for your Docker container. The base image provides the foundation for your container and typically includes a specific operating system, such as Ubuntu, CentOS, or Alpine, along with some pre-installed software packages.

**Create a Dockerfile**: The Dockerfile is a script that specifies the steps required to build the Docker image. It typically includes instructions for installing the application dependencies, copying the application code into the container,and configuring the container environment.

**Build the Docker image**: With the Dockerfile in place, you can use the "docker build" command to create the Docker image. This command reads the Dockerfile and builds the image according to the specified instructions.

**Test the Docker container**: Once the Docker image is built, you can use the "docker run" command to start a container based on that image. This will launch the application inside the container, allowing you to test it and make sure that it's working correctly.

**Optimize the Docker image**: After testing the Docker container, you can optimize the Docker image to reduce its size and improve its performance. This may involve removing unnecessary files, cleaning up temporary files, and minimizing the number of layers in the image.

**Publish the Docker image**: Finally, you can publish the Docker image to a container registry, such as Docker Hub, so that it can be easily shared and used by others. This step involves using the "docker push" command to upload the image to the registry.

Advantages

* It runs the container in seconds instead of minutes.
* It uses less memory.
* It provides lightweight virtualization.
* It does not a require full operating system to run applications.
* It uses application dependencies to reduce the risk.
* Allows you to use a remote repository to share your container with others.
* It provides continuous deployment and testing environment.

**Docker Dockerfile**

A Dockerfile is a text document that contains commands that are used to assemble an image. We can use any command that call on the command line. Docker builds images automatically by reading the instructions from the Dockerfile.

**Docker Container and Image**

Docker container is a running instance of an image. You can use Command Line Interface (CLI) commands to run, start, stop, move, or delete a container. You can also provide configuration for the network and environment variables. Docker container is an isolated and secure application platform, but it can share and access to resources running in a different host or container.

An image is a read-only template with instructions for creating a Docker container. A docker image is described in text file called a **Dockerfile**, which has a simple, well-defined syntax. An image does not have states and never changes. Docker Engine provides the core Docker technology that enables images and containers.

**Container Orchestration**

**Kubernetes**

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community. Designed on the same principles that allow Google to run billions of containers a week, Kubernetes can scale without increasing your operations team. Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is. Kubernetes is open source giving you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

* Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
* Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
* Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the [Open Service Broker](https://openservicebrokerapi.org/).
* Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
* Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
* Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
* Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

**Nodes**

* Kubernetes runs your [workload](https://kubernetes.io/docs/concepts/workloads/) by placing containers into Pods to run on *Nodes*. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) and contains the services necessary to run [Pods](https://kubernetes.io/docs/concepts/workloads/pods/).
* Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.
* The [components](https://kubernetes.io/docs/concepts/overview/components/#node-components) on a node include the [kubelet](https://kubernetes.io/docs/reference/generated/kubelet" \o "" \t "_blank), a [container runtime](https://kubernetes.io/docs/setup/production-environment/container-runtimes), and the [kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/" \t "_blank).

# Components

* A Kubernetes cluster consists of a set of worker machines, called [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/), that run containerized applications. Every cluster has at least one worker node.
* The worker node(s) host the [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) that are the components of the application workload. The [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Graphical user interface, diagram, application

Description automatically generated

The components of a Kubernetes cluster

## **Control Plane Components**

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new [pod](https://kubernetes.io/docs/concepts/workloads/pods/) when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See [Creating Highly Available clusters with kubeadm](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/) for an example control plane setup that runs across multiple machines.

### **kube-apiserver**

The API server is a component of the Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is [kube-apiserver](https://kubernetes.io/docs/reference/generated/kube-apiserver/). kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

### **etcd**

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a [back up](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/" \l "backing-up-an-etcd-cluster) plan for the data.

You can find in-depth information about etcd in the official [documentation](https://etcd.io/docs/).

### **kube-scheduler**

Control plane component that watches for newly created [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) with no assigned [node](https://kubernetes.io/docs/concepts/architecture/nodes/), and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

### **kube-controller-manager**

Control plane component that runs [controller](https://kubernetes.io/docs/concepts/architecture/controller/) processes.

Logically, each [controller](https://kubernetes.io/docs/concepts/architecture/controller/) is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

* Node controller: Responsible for noticing and responding when nodes go down.
* Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
* ServiceAccount controller: Create default ServiceAccounts for new namespaces.

### **cloud-controller-manager**

A Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

* Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
* Route controller: For setting up routes in the underlying cloud infrastructure
* Service controller: For creating, updating and deleting cloud provider load balancers

## **Node Components**

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

### **kubelet**

An agent that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in the cluster. It makes sure that [containers](https://kubernetes.io/docs/concepts/containers/) are running in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/).

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

### **kube-proxy**

kube-proxy is a network proxy that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster, implementing part of the Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) concept.

[kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

### **Container runtime**

The container runtime is the software that is responsible for running containers.

Kubernetes supports container runtimes such as [containerd](https://containerd.io/docs/" \o "" \t "_blank), [CRI-O](https://cri-o.io/#what-is-cri-o), and any other implementation of the [Kubernetes CRI (Container Runtime Interface)](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md).

## **Addons**

Addons use Kubernetes resources ([DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset" \o "" \t "_blank), [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

Selected addons are described below; for an extended list of available addons, please see [Addons](https://kubernetes.io/docs/concepts/cluster-administration/addons/).

### **DNS**

While the other addons are not strictly required, all Kubernetes clusters should have [cluster DNS](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/), as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

### **Web UI (Dashboard)**

[Dashboard](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

### **Container Resource Monitoring**

[Container Resource Monitoring](https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/) records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

### **Cluster-level Logging**

A [cluster-level logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/) mechanism is responsible for saving container logs to a central log store with search/browsing interface.

# Containers

Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it.

Containers decouple applications from underlying host infrastructure. This makes deployment easier in different cloud or OS environments.

Each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in a Kubernetes cluster runs the containers that form the [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node.

## **Container images**

A [container image](https://kubernetes.io/docs/concepts/containers/images/) is a ready-to-run software package, containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.

Containers are intended to be stateless and [immutable](https://glossary.cncf.io/immutable-infrastructure/): you should not change the code of a container that is already running. If you have a containerized application and want to make changes, the correct process is to build a new image that includes the change, then recreate the container to start from the updated image.

## **Container runtimes**

The container runtime is the software that is responsible for running containers.

Kubernetes supports container runtimes such as [containerd](https://containerd.io/docs/" \o "" \t "_blank), [CRI-O](https://cri-o.io/#what-is-cri-o), and any other implementation of the [Kubernetes CRI (Container Runtime Interface)](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md).

Usually, you can allow your cluster to pick the default container runtime for a Pod. If you need to use more than one container runtime in your cluster, you can specify the [RuntimeClass](https://kubernetes.io/docs/concepts/containers/runtime-class/) for a Pod to make sure that Kubernetes runs those containers using a particular container runtime.

You can also use RuntimeClass to run different Pods with the same container runtime but with different settings.

## **Container environment**

The Kubernetes Container environment provides several important resources to Containers:

* A filesystem, which is a combination of an [image](https://kubernetes.io/docs/concepts/containers/images/) and one or more [volumes](https://kubernetes.io/docs/concepts/storage/volumes/).
* Information about the Container itself.
* Information about other objects in the cluster.

### **Container information**

The hostname of a Container is the name of the Pod in which the Container is running. It is available through the hostname command or the [gethostname](https://man7.org/linux/man-pages/man2/gethostname.2.html) function call in libc.

The Pod name and namespace are available as environment variables through the [downward API](https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/).

User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image.

### **Cluster information**

A list of all services that were running when a Container was created is available to that Container as environment variables. This list is limited to services within the same namespace as the new Container's Pod and Kubernetes control plane services.

## **Container hooks**

There are two hooks that are exposed to Containers:

PostStart

This hook is executed immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPOINT. No parameters are passed to the handler.

PreStop

This hook is called immediately before a container is terminated due to an API request or management event such as a liveness/startup probe failure, preemption, resource contention and others. A call to the PreStop hook fails if the container is already in a terminated or completed state and the hook must complete before the TERM signal to stop the container can be sent. The Pod's termination grace period countdown begins before the PreStop hook is executed, so regardless of the outcome of the handler, the container will eventually terminate within the Pod's termination grace period. No parameters are passed to the handler.

# Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more [containers](https://kubernetes.io/docs/concepts/containers/), with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

Pods in a Kubernetes cluster are used in two main ways:

* **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
* **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate *sidecar* container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

# Services, Load Balancing, and Networking

## **The Kubernetes network model**

Every [Pod](https://kubernetes.io/docs/concepts/workloads/pods/) in a cluster gets its own unique cluster-wide IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.  
This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, [load balancing](https://kubernetes.io/docs/concepts/services-networking/ingress/#load-balancing), application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

* pods can communicate with all other pods on any other [node](https://kubernetes.io/docs/concepts/architecture/nodes/) without NAT
* agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.

Kubernetes networking addresses four concerns:

* Containers within a Pod [use networking to communicate](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/) via loopback.
* Cluster networking provides communication between different Pods.
* The [Service](https://kubernetes.io/docs/concepts/services-networking/service/) API lets you [expose an application running in Pods](https://kubernetes.io/docs/tutorials/services/connect-applications-service/) to be reachable from outside your cluster.
  + [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) provides extra functionality specifically for exposing HTTP applications, websites and APIs.
* You can also use Services to [publish services only for consumption inside your cluster](https://kubernetes.io/docs/concepts/services-networking/service-traffic-policy/).

##### [Service](https://kubernetes.io/docs/concepts/services-networking/service/)

Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

##### [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/)

Make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API.

##### [Ingress Controllers](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/)

In order for an [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) to work in your cluster, there must be an ingress controller running. You need to select at least one ingress controller and make sure it is set up in your cluster. This page lists common ingress controllers that you can deploy.

##### [EndpointSlices](https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/)

The EndpointSlice API is the mechanism that Kubernetes uses to let your Service scale to handle large numbers of backends, and allows the cluster to update its list of healthy backends efficiently.

##### [Network Policies](https://kubernetes.io/docs/concepts/services-networking/network-policies/)

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), NetworkPolicies allow you to specify rules for traffic flow within your cluster, and also between Pods and the outside world. Your cluster must use a network plugin that supports NetworkPolicy enforcement.

##### [DNS for Services and Pods](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)

Your workload can discover Services within your cluster using DNS; this page explains how that works.

##### [IPv4/IPv6 dual-stack](https://kubernetes.io/docs/concepts/services-networking/dual-stack/)

Kubernetes lets you configure single-stack IPv4 networking, single-stack IPv6 networking, or dual stack networking with both network families active. This page explains how.

##### [Topology Aware Routing](https://kubernetes.io/docs/concepts/services-networking/topology-aware-routing/)

Topology Aware Routing provides a mechanism to help keep network traffic within the zone where it originated. Preferring same-zone traffic between Pods in your cluster can help with reliability, performance (network latency and throughput), or cost.

[**Storage**](https://kubernetes.io/docs/concepts/storage/)

# Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state. A second problem occurs when sharing files between containers running together in a Pod. The Kubernetes [volume](https://kubernetes.io/docs/concepts/storage/volumes/) abstraction solves both of these problems.

# Persistent Volumes

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using [Storage Classes](https://kubernetes.io/docs/concepts/storage/storage-classes/). It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

# Projected Volumes

Each projected volume source is listed in the spec under sources. The parameters are nearly the same with two exceptions:

* For secrets, the secretName field has been changed to name to be consistent with ConfigMap naming.
* The defaultMode can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the mode for each individual projection.

# Ephemeral Volumes

Ephemeral volumes are specified inline in the Pod spec, which simplifies application deployment and management.

### **Types :**

Kubernetes supports several different kinds of ephemeral volumes for different purposes:

* [emptyDir](https://kubernetes.io/docs/concepts/storage/volumes/#emptydir): empty at Pod startup, with storage coming locally from the kubelet base directory (usually the root disk) or RAM
* [configMap](https://kubernetes.io/docs/concepts/storage/volumes/#configmap), [downwardAPI](https://kubernetes.io/docs/concepts/storage/volumes/" \l "downwardapi), [secret](https://kubernetes.io/docs/concepts/storage/volumes/#secret): inject different kinds of Kubernetes data into a Pod
* [CSI ephemeral volumes](https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/#csi-ephemeral-volumes): similar to the previous volume kinds, but provided by special [CSI drivers](https://github.com/container-storage-interface/spec/blob/master/spec.md) which specifically [support this feature](https://kubernetes-csi.github.io/docs/drivers.html)
* [generic ephemeral volumes](https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/#generic-ephemeral-volumes), which can be provided by all storage drivers that also support persistent volumes

emptyDir, configMap, downwardAPI, secret are provided as [local ephemeral storage](https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#local-ephemeral-storage). They are managed by kubelet on each node.

Each StorageClass contains the fields provisioner, parameters, and reclaimPolicy, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned.

The name of a StorageClass object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating StorageClass objects, and the objects cannot be updated once they are created.

[**Configuration**](https://kubernetes.io/docs/concepts/configuration/)

A ConfigMap is an API object used to store non-confidential data in key-value pairs. [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a [volume](https://kubernetes.io/docs/concepts/storage/volumes/).

A ConfigMap allows you to decouple environment-specific configuration from your [container images](https://kubernetes.io/docs/reference/glossary/?all=true#term-image), so that your applications are easily portable.

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/) specification or in a [container image](https://kubernetes.io/docs/reference/glossary/?all=true#term-image). Using a Secret means that you don't need to include confidential data in your application code.

Secrets are similar to [ConfigMaps](https://kubernetes.io/docs/concepts/configuration/configmap/" \o "" \t "_blank) but are specifically intended to hold confidential data.

When you specify a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/), you can optionally specify how much of each resource a [container](https://kubernetes.io/docs/concepts/containers/) needs. The most common resources to specify are CPU and memory (RAM); there are others.

Use kubeconfig files to organize information about clusters, users, namespaces, and authentication mechanisms. The kubectl command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.

 Windows uses a [job object](https://docs.microsoft.com/windows/win32/procthread/job-objects) per container with a system namespace filter to contain all processes in a container and provide logical isolation from the host.

Windows does not have an out-of-memory process killer as Linux does. Windows always treats all user-mode memory allocations as virtual, and pagefiles are mandatory.

Windows can limit the amount of CPU time allocated for different processes but cannot guarantee a minimum amount of CPU time.

To account for memory and CPU used by the operating system, the container runtime, and by Kubernetes host processes such as the kubelet, you can (and should) reserve memory and CPU resources with the --kube-reserved and/or --system-reserved kubelet flags. On Windows these values are only used to calculate the node's [allocatable](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable) resources.

# Security

The Pod Security Standards define three different policies to broadly cover the security spectrum. These policies are cumulative and range from highly-permissive to highly-restrictive. This guide outlines the requirements of each policy.

A service account is a type of non-human account that, in Kubernetes, provides a distinct identity in a Kubernetes cluster. Application Pods, system components, and entities inside and outside the cluster can use a specific ServiceAccount's credentials to identify as that ServiceAccount. This identity is useful in various situations, including authenticating to the API server or implementing identity-based security policies.

[**Policies**](https://kubernetes.io/docs/concepts/policy/)

A LimitRange provides constraints that can:

* Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
* Enforce minimum and maximum storage request per [PersistentVolumeClaim](https://kubernetes.io/docs/concepts/storage/persistent-volumes/" \l "persistentvolumeclaims" \o "" \t "_blank) in a namespace.
* Enforce a ratio between request and limit for a resource in a namespace.
* Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

A LimitRange is enforced in a particular namespace when there is a LimitRange object in that namespace.

# Scheduling, Preemption and Eviction

In Kubernetes, scheduling refers to making sure that [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) are matched to [Nodes](https://kubernetes.io/docs/concepts/architecture/nodes/) so that the [kubelet](https://kubernetes.io/docs/reference/generated/kubelet" \o "" \t "_blank) can run them. Preemption is the process of terminating Pods with lower [Priority](https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/#pod-priority) so that Pods with higher Priority can schedule on Nodes. Eviction is the process of terminating one or more Pods on Nodes.

[Pod disruption](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/) is the process by which Pods on Nodes are terminated either voluntarily or involuntarily.

Voluntary disruptions are started intentionally by application owners or cluster administrators. Involuntary disruptions are unintentional and can be triggered by unavoidable issues like Nodes running out of resources, or by accidental deletions.

# Extending Kubernetes

Kubernetes is highly configurable and extensible. As a result, there is rarely a need to fork or submit patches to the Kubernetes project code.  It is aimed at [cluster operators](https://kubernetes.io/docs/reference/glossary/?all=true#term-cluster-operator) who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective [Platform Developers](https://kubernetes.io/docs/reference/glossary/?all=true#term-platform-developer) or Kubernetes Project [Contributors](https://kubernetes.io/docs/reference/glossary/?all=true#term-contributor) will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.