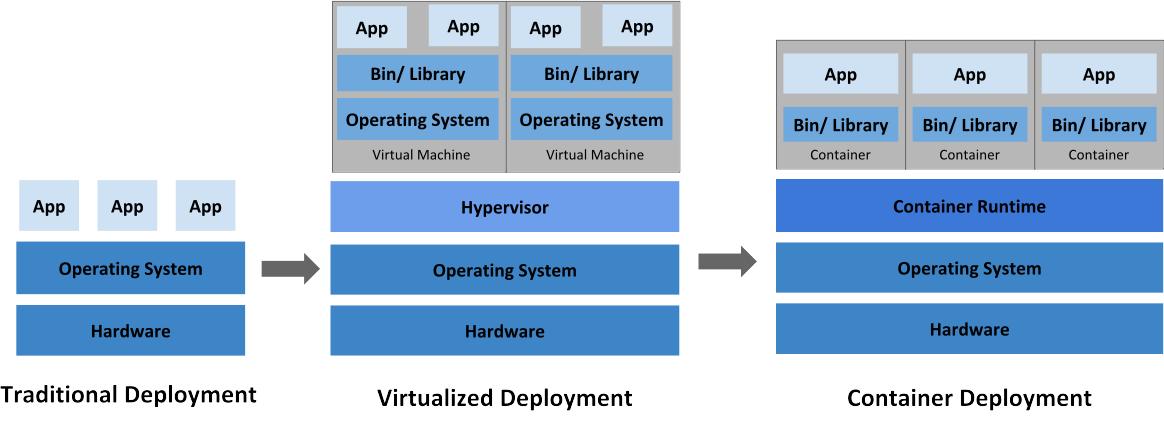
## **Why we go for a Kubernetes**

Let's take a look at why Kubernetes is so useful by going back in time.



**Traditional deployment era:** Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

**Virtualized deployment era:** As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

**Container deployment era:** Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

* Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
* Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
* Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
* Observability not only surfaces OS-level information and metrics, but also application health and other signals.
* Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
* Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
* Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
* Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
* Resource isolation: predictable application performance.
* Resource utilization: high efficiency and density.

## **Why you need Kubernetes and what it can do**

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

* **Service discovery and load balancing**  
  Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
* **Storage orchestration**  
  Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
* **Automated rollouts and rollbacks**  
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
* **Automatic bin packing**  
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
* **Self-healing**  
  Kubernetes restarts containers that fail, replaces containers, kills containers that don’t respond to your user-defined health check, and doesn’t advertise them to clients until they are ready to serve.
* **Secret and configuration management**  
  Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

## **What Kubernetes is not**

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.

# **Kubernetes Architecture**

# 

# **Concepts**

The Concepts section helps you learn about the parts of the Kubernetes system and the abstractions Kubernetes uses to represent your [cluster](https://kubernetes.io/docs/reference/glossary/?all=true" \l "term-cluster), and helps you obtain a deeper understanding of how Kubernetes works.

## **Overview**

To work with Kubernetes, you use Kubernetes API objects to describe your cluster’s desired state: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. You set your desired state by creating objects using the Kubernetes API, typically via the command-line interface, kubectl. You can also use the Kubernetes API directly to interact with the cluster and set or modify your desired state.

Once you’ve set your desired state, the Kubernetes Control Plane makes the cluster’s current state match the desired state via the Pod Lifecycle Event Generator ([PLEG](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/pod-lifecycle-event-generator.md)). To do so, Kubernetes performs a variety of tasks automatically–such as starting or restarting containers, scaling the number of replicas of a given application, and more. The Kubernetes Control Plane consists of a collection of processes running on your cluster:

* The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the master node. Those processes are: [kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/), [kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/) and [kube-scheduler](https://kubernetes.io/docs/admin/kube-scheduler/).
* Each individual non-master node in your cluster runs two processes:
  + [**kubelet**](https://kubernetes.io/docs/admin/kubelet/), which communicates with the Kubernetes Master.
  + [**kube-proxy**](https://kubernetes.io/docs/admin/kube-proxy/), a network proxy which reflects Kubernetes networking services on each node.

# **kube-apiserver**

The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. The API Server services REST operations and provides the frontend to the cluster’s shared state through which all other components interact.

# **kube-controller-manager**

In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. Examples of controllers that ship with Kubernetes today are the replication controller, endpoints controller, namespace controller, and serviceaccounts controller.

# **Kubernetes Scheduler**

In Kubernetes, scheduling refers to making sure that [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) are matched to [Nodes](https://kubernetes.io/docs/concepts/architecture/nodes/) so that [Kubelet](https://kubernetes.io/docs/reference/generated/kubelet) can run them.

* **[Scheduling overview](https://kubernetes.io/docs/concepts/scheduling-eviction/" \l "scheduling)**
* **[kube-scheduler](https://kubernetes.io/docs/concepts/scheduling-eviction/" \l "kube-scheduler)**

## **Scheduling overview**

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on.

[kube-scheduler](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/) is the default scheduler for Kubernetes and runs as part of the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true" \l "term-control-plane).

kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.

For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called feasible nodes.

If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod.

The scheduler then notifies the API server about this decision in a process called binding.

Factors that need 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 so on.

### Node selection in kube-scheduler

kube-scheduler selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The filtering step finds the set of Nodes where it’s feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resource to meet a Pod’s specific resource requests.

In the scoring step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement.

The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

Finally, kube-scheduler assigns the Pod to the Node with the highest ranking.

If there is more than one node with equal scores, kube-scheduler selects one of these at random.

# **kubelet**

The kubelet is the primary “node agent” that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.

The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod..

Podspec is healthy then the kublet is running on it .

The kubelet doesn’t manage containers which were not created by Kubernetes.

Other than from a PodSpec from the apiserver, there are three ways that a container manifest can be provided to the Kubelet.

File: Path passed as a flag on the command line. Files under this path will be monitored periodically for updates. The monitoring period is 20s by default and is configurable via a flag.

HTTP endpoint: HTTP endpoint passed as a parameter on the command line. This endpoint is checked every 20 seconds (also configurable with a flag).

HTTP server: The kubelet can also listen for HTTP and respond to a simple API (underspec’d currently) to submit a new manifest.

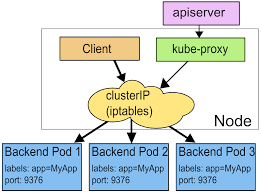
# **kube-proxy**

The Kubernetes network proxy runs on each node. This reflects services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends.

Service cluster IPs and ports are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy.

There is an optional addon that provides cluster DNS for these cluster IPs.

The user must create a service with the apiserver API to configure the proxy.



## **Kubernetes objects**

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing. These abstractions are represented by objects in the Kubernetes API. See [Understanding Kubernetes objects](https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/" \l "kubernetes-objects) for more details.

The basic Kubernetes objects include:

* [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/)
* [Service](https://kubernetes.io/docs/concepts/services-networking/service/)
* [Volume](https://kubernetes.io/docs/concepts/storage/volumes/)
* [Namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)

Kubernetes also contains higher-level abstractions that rely on [controllers](https://kubernetes.io/docs/concepts/architecture/controller/) to build upon the basic objects, and provide additional functionality and convenience features. These include:

* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)
* [Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)

# **Pod Overview**

A Pod is the basic execution unit of a Kubernetes application–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents processes running on your [cluster](https://kubernetes.io/docs/reference/glossary/?all=true" \l "term-cluster).

**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, and Kubernetes manages the Pods rather than the containers directly.

**Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public,

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as replication

#### **Networking**

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports.

#### **Storage**

A Pod can specify a set of shared storage [volumes](https://kubernetes.io/docs/concepts/storage/volumes/). All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted

## **Working with Pods**

When a Pod gets created (directly by you, or indirectly by a [\_controller\_](https://kubernetes.io/docs/concepts/architecture/controller/)), it is scheduled to run on a [Node](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster. The Pod remains on that node until the process is terminated, the pod object is deleted, the Pod is evicted for lack of resources, or the node fails.

### Pods and controllers

You can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node.

**apiVersion**: batch/v1

**kind**: Job

**metadata**:

**name**: hello

**spec**:

**template**:

*# This is the pod template*

**spec**:

**containers**:

- **name**: hello

**image**: busybox

**command**: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']

**restartPolicy**: OnFailure

**apiVersion**: apps/v1 *# for versions before 1.9.0 use apps/v1beta2*

**kind**: Deployment

**metadata**:

**name**: nginx-deployment

**spec**:

**selector**:

**matchLabels**:

**app**: nginx

**replicas**: 2 *# tells deployment to run 2 pods matching the template*

**template**:

**metadata**:

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

For example, a Deployment controller ensures that the running Pods match the current pod template. If the template is updated, the controller has to remove the existing Pods and create new Pods based on the updated template.

# **Service**

An abstract way to expose an application running on a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) as a network service.

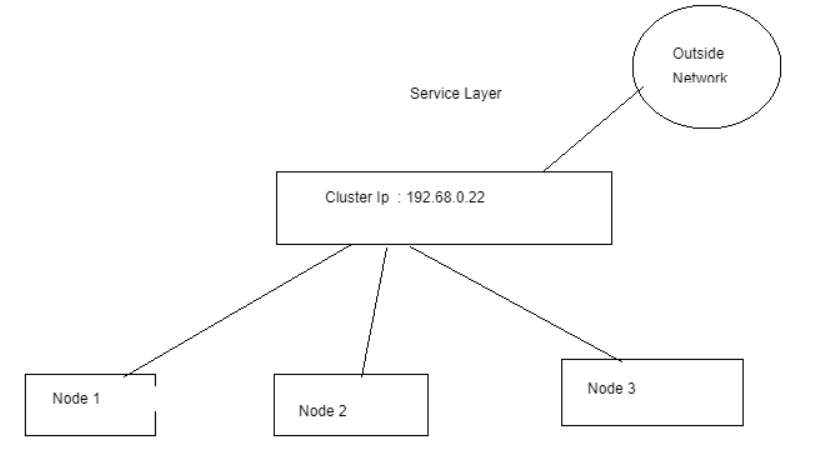
A service is assigned an IP address (“cluster IP”), which the service proxies use.

## **Types of Kubernetes services**

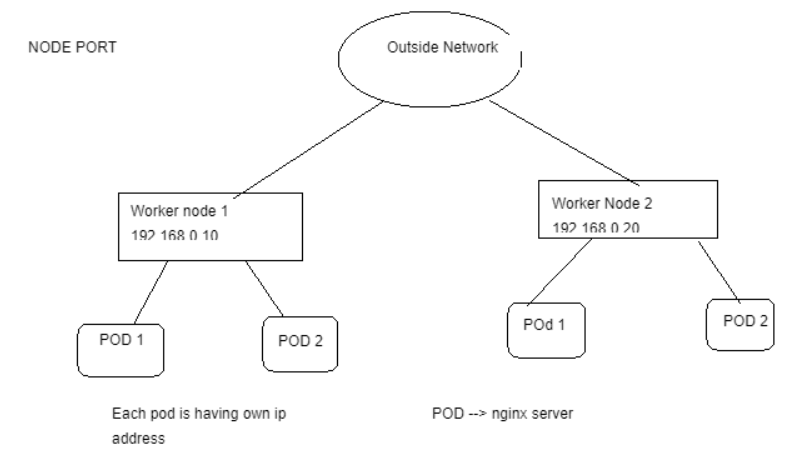
There are four types of Kubernetes services:

1. **ClusterIP.** This default type exposes the service on a cluster-internal IP. You can reach the service only from within the cluster.
2. **NodePort.** This type of service exposes the service on each node’s IP at a static port. A ClusterIP service is created automatically, and the NodePort service will route to it. From outside the cluster, you can contact the NodePort service by using “<NodeIP>:<NodePort>”.
3. **LoadBalancer.** This service type exposes the service externally using the load balancer of your cloud provider. The external load balancer routes to your NodePort and ClusterIP services, which are created automatically.
4. **ExternalName.** This type maps the service to the contents of the externalName field (e.g., foo.bar.example.com). It does this by returning a value for the CNAME record.

CLUSTER IP



NODEPORT



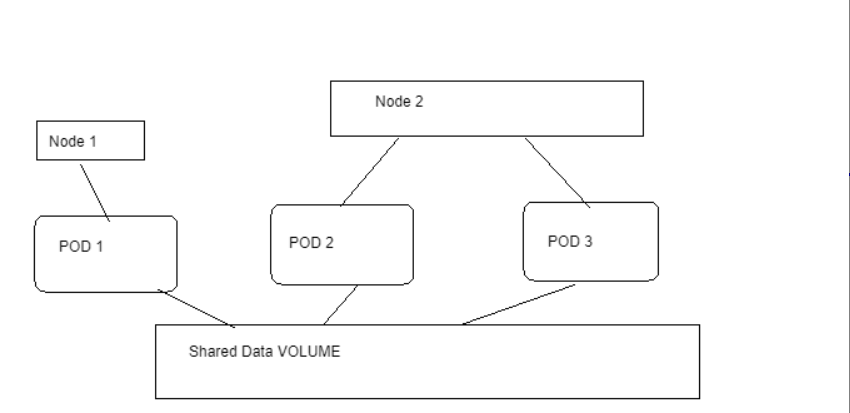
# **Volumes**

First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state.

Second, when running Containers together in a Pod it is often necessary to share files between those Containers.

The Kubernetes Volume abstraction solves both of these problems.

SHARED VOLUME



Volume

===========

step 1 : create a pv

step 2: create a pvc

step 3: pod that use the pv -->pvc

Note: Default reclaim policy is retain

PV reclaim policy

==================================

1.retain --> once PVC is deleted and data can't be lost

2.recycle --> its for dynamic provissioning ,

3.delete --> once pod is delete or pvc,its delete the data also

ACL Permision on PV/PVC

=================

1.Read write only

2.read only

3.read write many

volume types

===============

1.local (hostpath) --> its only support one node for ex: if the pods is running on 1st node ,the data stored in 1st node.

if pod fails data wont be deleted but if the node failed data also failed,not recomend for production

2.nfs hostpath --> default mount path is /tmp

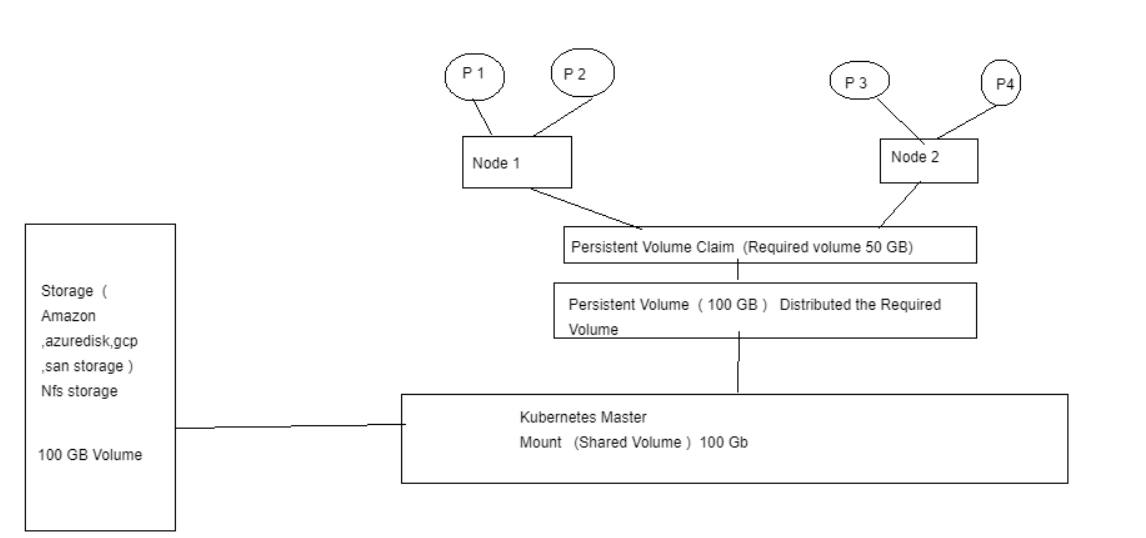
3.iscsi

4.azure disk

5.ebs

6.gke

PVC



YAML FILES

Ex: PV

======

apiversion: v1

kind: PersisitentVolume

metadata:

name: pv-hostpath-local-mount

labels:

type: local

spec:

storageclassname: manual

capacity:

storage: 1G

accessmodes:

- ReadWriteMany

hostpath:

path: /data

Ex: PVC

================

apiversion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-hostpath-local-mount

Spec:

storageclassname: manual

accessmodes:

- ReadWriteMany

Resources:

requests:

storage: 100Mi

Eg: Now How pod is using the pvc

=========================================

apiversion: v1

kind: pod

metadata:

name: pod-volume-pv

spec:

volumes:

- name: pvc-pod-volume

PersistentVolumeClaim

Cliamname: pvc-hostpath-local-mount --> already configure in PVC --> PVC Name

Containers:

- image: busybox

name: busybox

command: /bin/bash

args: "echo redhat"

volumemounts:

- name: pvc-pod-volume

mountpath: /data

Namespace

======= =====

Namespaces are intended for use in environments with many users spread across multiple teams, or projects

#kubectl create namespace namespace dev-project

# Kubectl create myngnix –image nginx -n dev-project

# **Deployments**

A Deployment provides declarative updates for [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) and [ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/).

You describe a desired state in a Deployment, and the Deployment [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) changes the actual state to the desired state at a controlled rate.

You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

#kubectl delete pod nginx

#kubectl delete deploy nginx

**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:1.14.2

**ports**:

- **containerPort**: 80

# **DaemonSet**

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

* running a cluster storage daemon, such as glusterd, ceph, on each node.
* running a logs collection daemon on every node, such as fluentd or filebeat.
* running a node monitoring daemon on every node, such as [Prometheus Node Exporter](https://github.com/prometheus/node_exporter), [Flowmill](https://github.com/Flowmill/flowmill-k8s/), [Sysdig Agent](https://docs.sysdig.com/), collectd, [Dynatrace OneAgent](https://www.dynatrace.com/technologies/kubernetes-monitoring/), [AppDynamics Agent](https://docs.appdynamics.com/display/CLOUD/Container+Visibility+with+Kubernetes), [Datadog agent](https://docs.datadoghq.com/agent/kubernetes/daemonset_setup/), [New Relic agent](https://docs.newrelic.com/docs/integrations/kubernetes-integration/installation/kubernetes-installation-configuration), Ganglia gmond, [Instana Agent](https://www.instana.com/supported-integrations/kubernetes-monitoring/) or [Elastic Metricbeat](https://www.elastic.co/guide/en/beats/metricbeat/current/running-on-kubernetes.html).

**apiVersion**: apps/v1

**kind**: DaemonSet

**metadata**:

**name**: fluentd-elasticsearch

**namespace**: kube-system

**labels**:

**k8s-app**: fluentd-logging

**spec**:

**selector**:

**matchLabels**:

**name**: fluentd-elasticsearch

**template**:

**metadata**:

**labels**:

**name**: fluentd-elasticsearch

**spec**:

**tolerations**:

*# this toleration is to have the daemonset runnable on master nodes*

*# remove it if your masters can't run pods*

- **key**: node-role.kubernetes.io/master

**effect**: NoSchedule

**containers**:

- **name**: fluentd-elasticsearch

**image**: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2

**resources**:

**limits**:

**memory**: 200Mi

**requests**:

**cpu**: 100m

**memory**: 200Mi

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: varlibdockercontainers

**mountPath**: /var/lib/docker/containers

**readOnly**: **true**

**terminationGracePeriodSeconds**: 30

**volumes**:

- **name**: varlog

**hostPath**:

**path**: /var/log

- **name**: varlibdockercontainers

**hostPath**:

**path**: /var/lib/docker/containers

# **StatefulSets**

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/), and provides guarantees about the ordering and uniqueness of these Pods.

Like a [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: nginx

**labels**:

**app**: nginx

**spec**:

**ports**:

- **port**: 80

**name**: web

**clusterIP**: None

**selector**:

**app**: nginx

---

**apiVersion**: apps/v1

**kind**: StatefulSet

**metadata**:

**name**: web

**spec**:

**selector**:

**matchLabels**:

**app**: nginx *# has to match .spec.template.metadata.labels*

**serviceName**: "nginx"

**replicas**: 3 *# by default is 1*

**template**:

**metadata**:

**labels**:

**app**: nginx *# has to match .spec.selector.matchLabels*

**spec**:

**terminationGracePeriodSeconds**: 10

**containers**:

- **name**: nginx

**image**: k8s.gcr.io/nginx-slim:0.8

**ports**:

- **containerPort**: 80

**name**: web

**volumeMounts**:

- **name**: www

**mountPath**: /usr/share/nginx/html

**volumeClaimTemplates**:

- **metadata**:

**name**: www

**spec**:

**accessModes**: [ "ReadWriteOnce" ]

**storageClassName**: "my-storage-class"

**resources**:

**requests**:

**storage**: 1Gi

# **Jobs - Run to Completion**

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

**apiVersion**: batch/v1

**kind**: Job

**metadata**:

**name**: pi

**spec**:

**template**:

**spec**:

**containers**:

- **name**: pi

**image**: perl

**command**: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

**restartPolicy**: Never

**backoffLimit**: 4

Nodes

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node contains the services necessary to run [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/), managed by the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true" \l "term-control-plane).

# kubectl describe node <insert-node-name-here>

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

* HostName: The hostname as reported by the node’s kernel. Can be overridden via the kubelet --hostname-override parameter.
* ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
* InternalIP: Typically the IP address of the node that is routable only within the cluster.

### Conditions

The conditions field describes the status of all Running nodes. Examples of conditions include:

| Node Condition | Description |
| --- | --- |
| Ready | True if the node is healthy and ready to accept pods, False if the node is not healthy and is not  accepting pods, and Unknown if the node controller has not heard from the node in the  last node-monitor-grace-period (default is 40 seconds) |
| DiskPressure | True if pressure exists on the disk size–that is, if the disk capacity is low; otherwise False |
| MemoryPressure | True if pressure exists on the node memory–that is, if the node memory is low; otherwise False |
| PIDPressure | True if pressure exists on the processes—that is, if there are too many processes on the node;  otherwise False |
| NetworkUnavailable | True if the network for the node is not correctly configured, otherwise False |

The node condition is represented as a JSON object. For example, the following structure describes a healthy node:

"conditions": [

{

**"type"**: "Ready",

**"status"**: "True",

**"reason"**: "KubeletReady",

**"message"**: "kubelet is posting ready status",

**"lastHeartbeatTime"**: "2019-06-05T18:38:35Z",

**"lastTransitionTime"**: "2019-06-05T11:41:27Z"

}

]

If the Status of the Ready condition remains Unknown or False for longer than the pod-eviction-timeout (an argument passed to the [kube-controller-manager](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/)), all the Pods on the node are scheduled for deletion by the node controller. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

The node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. You can see the pods that might be running on an unreachable node as being in the Terminating or Unknown state

### Node controller

The node [controller](https://kubernetes.io/docs/concepts/architecture/controller/) is a Kubernetes control plane component that manages various aspects of nodes.

The node controller has multiple roles in a node’s life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller’s internal list of nodes up to date with the cloud provider’s list of available machines. When running in a cloud environment, whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes’ health. The node controller is responsible for updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable (i.e. the node controller stops receiving heartbeats for some reason, for example due to the node being down), and then later evicting all the pods from the node (using graceful termination) if the node continues to be unreachable. (The default timeouts are 40s to start reporting ConditionUnknown and 5m after that to start evicting pods.) The node controller checks the state of each node every --node-monitor-period seconds.

#### **Heartbeats**

Heartbeats, sent by Kubernetes nodes, help determine the availability of a node.

The kubelet is responsible for creating and updating the NodeStatus and a Lease object.

* The kubelet updates the NodeStatus either when there is change in status, or if there has been no update for a configured interval. The default interval for NodeStatus updates is 5 minutes (much longer than the 40 second default timeout for unreachable nodes).
* The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from the NodeStatus updates. If the Lease update fails, the kubelet retries with exponential backoff starting at 200 milliseconds and capped at 7 seconds.

### Node capacity

Node objects track information about the Node’s resource capacity (for example: the amount of memory available, and the number of CPUs). Nodes that [self register](https://kubernetes.io/docs/concepts/architecture/nodes/" \l "self-registration-of-nodes) report their capacity during registration. If you [manually](https://kubernetes.io/docs/concepts/architecture/nodes/" \l "manual-node-administration) add a Node, then you need to set the node’s capacity information when you add it.

The Kubernetes [scheduler](https://kubernetes.io/docs/reference/generated/kube-scheduler/) ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node’s capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet’s control.

## **Control Plane to node**

There are two primary communication paths from the control plane (apiserver) to the nodes. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver’s proxy functionality.

### apiserver to kubelet

The connections from the apiserver to the kubelet are used for:

* Fetching logs for pods.
* Attaching (through kubectl) to running pods.
* Providing the kubelet’s port-forwarding functionality.

These connections terminate at the kubelet’s HTTPS endpoint. By default, the apiserver does not verify the kubelet’s serving certificate, which makes the connection subject to man-in-the-middle attacks, and **unsafe** to run over untrusted and/or public networks.

To verify this connection, use the --kubelet-certificate-authority flag to provide the apiserver with a root certificate bundle to use to verify the kubelet’s serving certificate.

If that is not possible, use [SSH tunneling](https://kubernetes.io/docs/concepts/architecture/master-node-communication/" \l "ssh-tunnels) between the apiserver and kubelet if required to avoid connecting over an untrusted or public network.

Finally, [Kubelet authentication and/or authorization](https://kubernetes.io/docs/admin/kubelet-authentication-authorization/) should be enabled to secure the kubelet API.

### apiserver to nodes, pods, and services

The connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing https: to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials so while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted and/or public networks.

### SSH tunnels

Kubernetes supports SSH tunnels to protect the control plane to nodes communication paths. In this configuration, the apiserver initiates an SSH tunnel to each node in the cluster (connecting to the ssh server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

SSH tunnels are currently deprecated so you shouldn’t opt to use them unless you know what you are doing. The Konnectivity service is a replacement for this communication channel.

### Control via API server

The [Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion) controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/), or perhaps several Pods, to carry out a task and then stop.

(Once [scheduled](https://kubernetes.io/docs/concepts/scheduling-eviction/), Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true" \l "term-control-plane) act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

(This is a bit like how some thermostats turn a light off to indicate that your room is now at the temperature you set).