KPLABS Course

Docker Certified Associate 2020

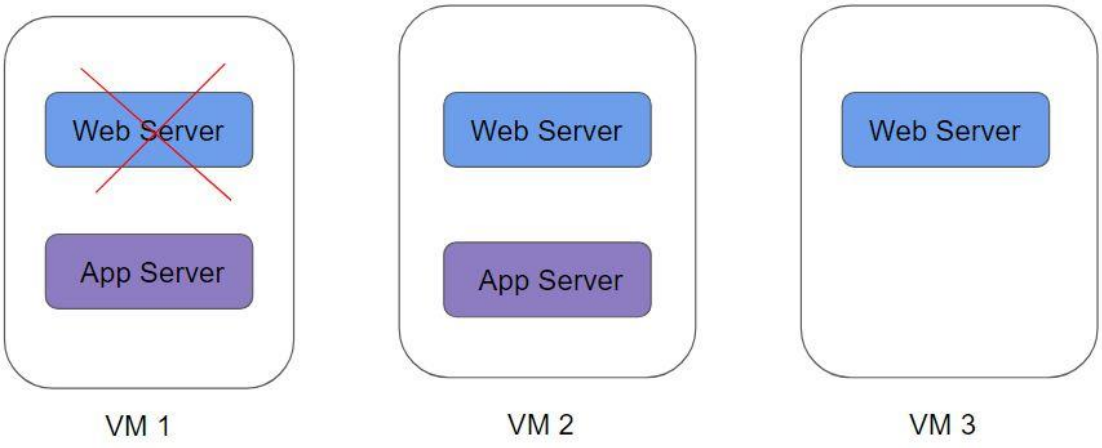
Orchestration

ISSUED BY
Zeal Vora

REPRESENTATIVE
instructors@kplabs.in

Module 1: Overview of Container Orchestration

Container orchestration is all about managing the life cycles of containers, especially in large, dynamic environments.



Container Orchestration can be used to perform a lot of tasks, some of them includes:

- Provisioning and deployment of containers
- Scaling up or removing containers to spread application load evenly
- Movement of containers from one host to another if there is a shortage of resources
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts

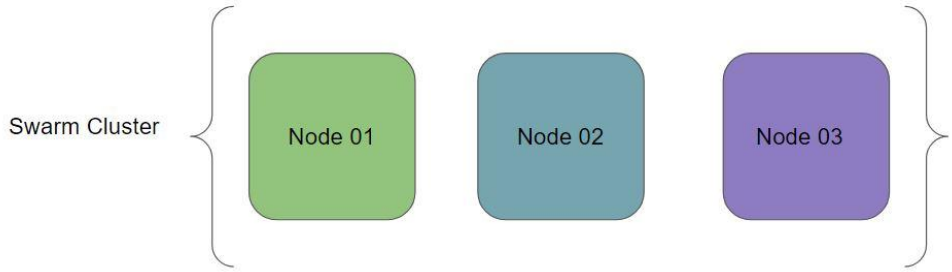
There are many container orchestration solutions which are available, some of the popular ones include:

- Docker Swarm
- Kubernetes
- Apache Mesos
- Elastic Container Service (AWS ECS)

There are also various container orchestration platforms available like EKS.

Module 2: Overview of Docker Swarm

Docker Swarm is a container orchestration tool that is natively supported by Docker.



A node is an instance of the Docker engine participating in the swarm.

To deploy your application to a swarm, you submit a service definition to a manager node.

The manager node dispatches units of work called tasks to worker nodes.

Module 3: Initializing Docker Swarm

Manager Node Command:

```
docker swarm init --advertise-addr <MANAGER-IP>
```

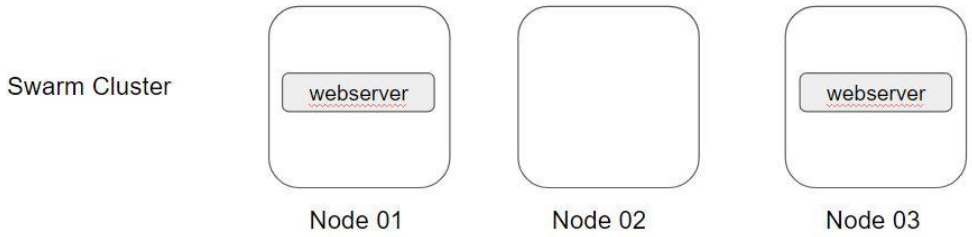
2. Worker Nodes Command:

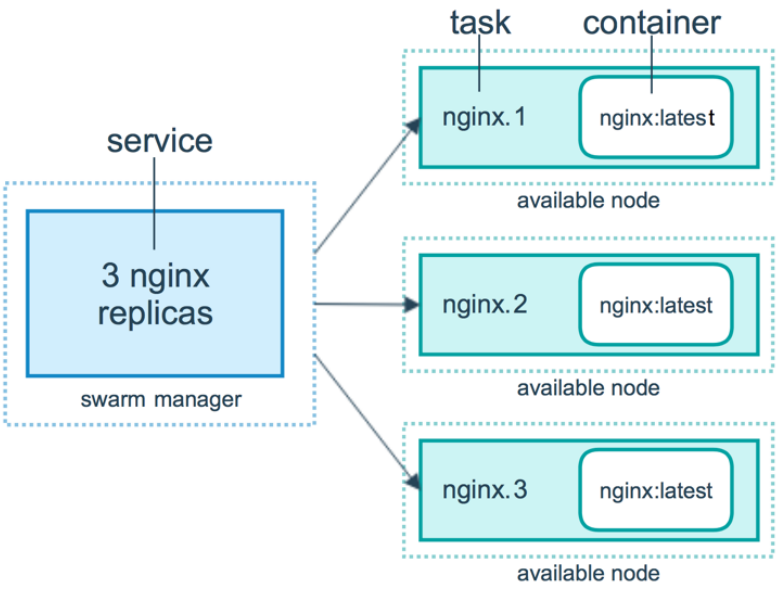
```
docker swarm join-token worker
```

Module 4: Services, Tasks, and Containers

A service is the definition of the tasks to execute on the manager or worker nodes.

docker service create --name webserver --replicas 1 nginx





I

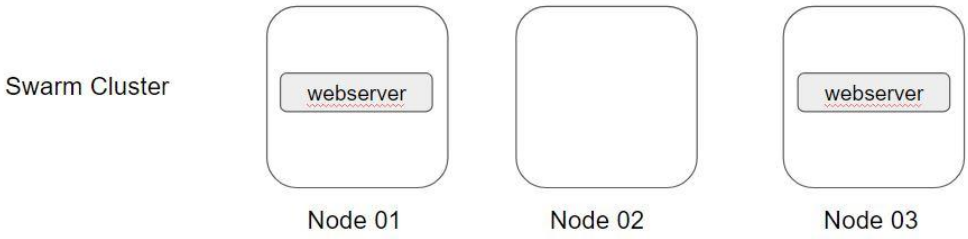
Module 5: Scaling Swarm Service

Once you have deployed a service to a swarm, you are ready to use the Docker CLI to scale the number of containers in the service.

Containers running in service are called “tasks.”

There are two ways in which you can scale service in a swarm:

- docker service scale webserver=5
- docker service update --replicas 5 mywebsever



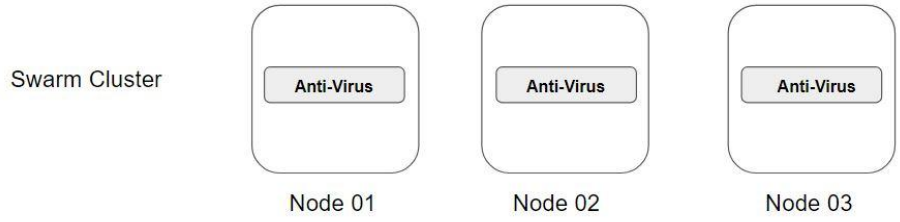
Module 6: Replicated vs Global Service

There are two types of services deployments, replicated and global

For a replicated service, you specify the number of identical tasks you want to run. For example, you decide to deploy an NGINX service with two replicas, each serving the same content.

A global service is a service that runs one task on every node.

Each time you add a node to the swarm, the orchestrator creates a task and the scheduler assigns the task to the new node.



Module 7: Docker Compose

Compose is a tool for defining and running multi-container Docker applications.

With Compose, you use a YAML file to configure your application's services.

We can start all the services with a single command - docker compose up

We can stop all the services with a single command - docker compose down

Module 8: Docker Stack

A specific web-application might have multiple containers that are required as part of the build process.

Whenever we make use of docker service, it is typically for a single container image.

The docker stack can be used to manage a multi-service application.

A stack is a group of interrelated services that share dependencies and can be orchestrated and scaled together.

A stack can compose a YAML file like the one that we define during Docker Compose.

We can define everything within the YAML file that we might define while creating a Docker Service.

Module 9: Locking Swarm Cluster

Swarm Cluster contains a lot of sensitive information, some of which includes:

- TLS key used to encrypt communication among swarm node
- Keys used to encrypt and decrypt the Raft logs on disk

If your Swarm is compromised and if data is stored in plain-text, an attack can get all the sensitive information.

Docker Lock allows us to have control over the keys.

Module 10: Troubleshooting Service Deployment

A service may be configured in such a way that no node currently in the swarm can run its tasks.

In this case, the service remains in state pending

There are multiple reasons why service might go into a pending state

```
[root@swarm01 ~]# docker service ps demotroubleshoot
ID                  NAME                    IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
ulggnljj0ber        demotroubleshoot.1      nginx:latest                            Running             Pending 17 seconds ago   "no suitable node (scheduling …"
g819kx5dqp9b        demotroubleshoot.2      nginx:latest                            Running             Pending 17 seconds ago   "no suitable node (scheduling …"
k83f9wo0u6xc        demotroubleshoot.3      nginx:latest                            Running             Pending 17 seconds ago   "no suitable node (scheduling …"
```

If all nodes are drained, and you create a service, it is pending until a node becomes available.

You can reserve a specific amount of memory for a service. If no node in the swarm has the required amount of memory, the service remains in a pending state until a node is available which can run its tasks.

You have imposed some kind of placement constraints

Module 11: Control Service Placement

Swarm services provide a few different ways for you to control the scale and placement of services on different nodes.

- Replicated and Global Services
- Resource Constraints [requirement of CPU and Memory]
- Placement Constraints [only run on nodes with label pci_compliance = true]
- Placement Preferences

Module 12: Overlay Networks

The overlay network driver creates a distributed network among multiple Docker daemon hosts

Overlay network allows containers connected to it to communicate securely.

To create a custom overlay network, the following command can be used:

docker network create -d overlay my-overlay

Module 13: Securing Overlay Networks

For the overlay networks, the containers can be spread across multiple servers.

If the containers are communicating with each other, it is recommended to secure the communication.

To enable encryption, when you create an overlay network pass the --opt encrypted flag:

docker network create --opt encrypted --driver overlay my-overlay-secure-network

When you enable overlay encryption, Docker creates IPSEC tunnels between all the nodes where tasks are scheduled for services attached to the overlay network.

These tunnels also use the AES algorithm in GCM mode and manager nodes automatically rotate the keys every 12 hours.

Overlay network encryption is not supported on Windows. If a Windows node attempts to connect to an encrypted overlay network, no error is detected but the node will not be able to communicate.

Module 14: Creating Swarm Services using Templates

We can make use of templates while running the service create command in the swarm.

Let us understand this with an example:

docker service create --name demoservice
--hostname="{{.Node.Hostname}}-{{.Service.Name}}" nginx

Here are some of the valid place holders:

Placeholder	Description
Service.ID	Service ID
.Service.Name	Service name
.Service.Labels	Service labels
.Node.ID	Node ID
.Node.Hostname	Node Hostname
.Task.ID	Task ID
.Task.Name	Task name
.Task.Slot	Task slot

There are three supported flags for the placeholder templates:

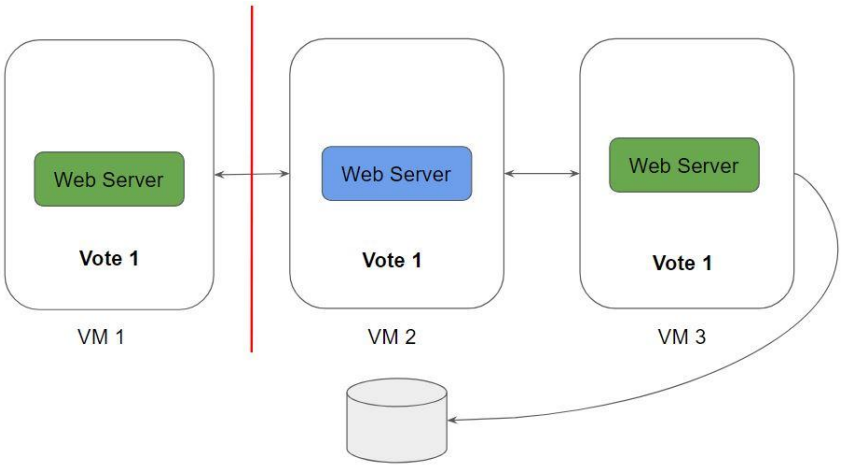
--hostname
--mount
--env

Module 15: Split Brain & Importance of Quorum

Swarm manager nodes use the Raft Consensus Algorithm to manage the swarm state. You only need to understand some general concepts of Raft in order to manage a swarm.

There is no limit on the number of manager nodes. The decision about how many manager nodes to implement is a trade-off between performance and fault-tolerance.

Adding manager nodes to a swarm makes the swarm more fault-tolerant.



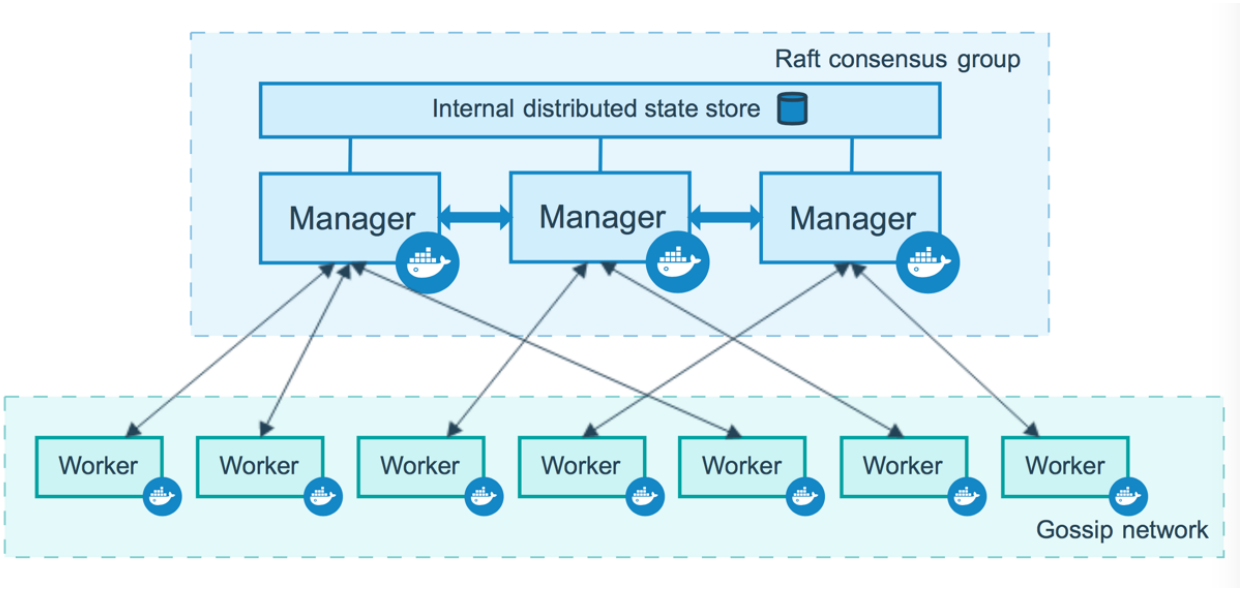
,

We should maintain an odd number of nodes within the cluster.

Cluster Size	Majority	Fault Tolerance
1	0	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

Module 16: Swarm Manager Node HA

Manager Nodes are responsible for handling the cluster management tasks.



Manager Node has many responsibilities within the swarm, these include:

- Maintaining the cluster state
- Scheduling services
- Serving swarm mode HTTP API endpoints

Using a Raft implementation, the managers maintain a consistent internal state of the entire swarm and all the services running on it

Swarm comes with its own fault-tolerance features.

Docker recommends you implement an odd number of nodes according to your organization's high-availability requirements.

Using a Raft implementation, the managers maintain a consistent internal state of the entire swarm and all the services running on it

An N manager cluster tolerates the loss of at most (N-1)/2 managers.

Module 17: Running Manager-Only nodes in Swarm

By default manager nodes also act as worker nodes. This means the scheduler can assign tasks to a manager node.

For small and non-critical swarms assigning tasks to managers is relatively low-risk as long as you schedule services using resource constraints for CPU and memory.

To avoid interference with manager node operation, you can drain manager nodes to make them unavailable as worker nodes:

docker node update --availability drain <NODE>

When you drain a node, the scheduler reassigns any tasks running on the node to other available worker nodes in the swarm. It also prevents the scheduler from assigning tasks to the node.

Module 18: Recover from losing the quorum

Swarm is resilient to failures and the swarm can recover from any number of temporary node failures (machine reboots or crash with restart) or other transient errors. However, a swarm cannot automatically recover if it loses a quorum.

The best way to recover from losing the quorum is to bring the failed nodes back online. If you can't do that, the only way to recover from this state is to use the **--force-new-cluster** action from a manager node.

This removes all managers except the manager the command was run from. The quorum is achieved because there is now only one manager. Promote nodes to be managers until you have the desired number of managers.

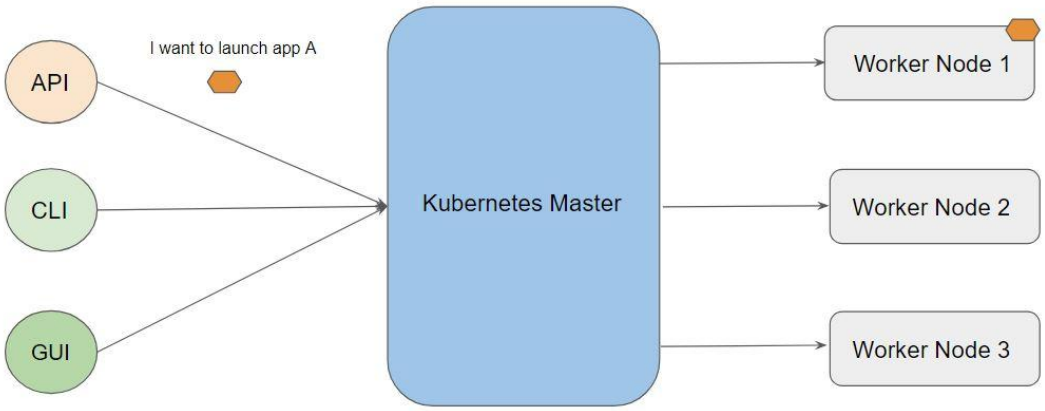
Sample Command:

docker swarm init --force-new-cluster --advertise-addr node01:2377

Module 19: Introduction to Kubernetes

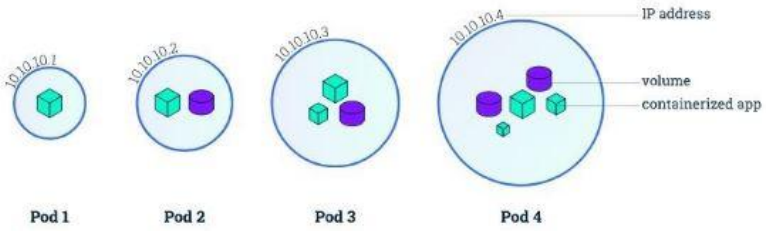
Kubernetes (K8s) is an open-source container orchestration engine developed by Google.

It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation.

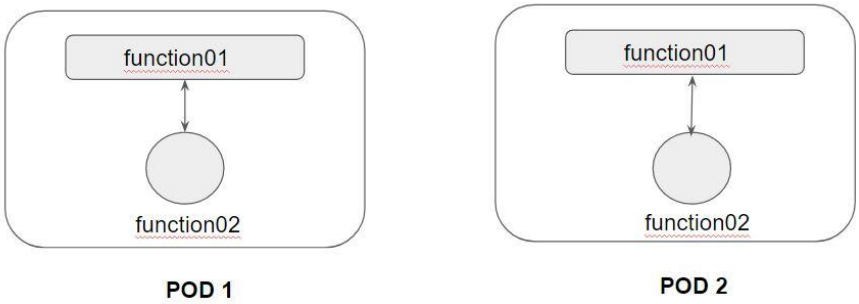


Module 20: PODS

A Pod in Kubernetes represents a group of one or more application containers and some shared resources for those containers.



Containers within a Pod share an IP address and port space and can find each other via the localhost.



A Pod always runs on a Node.

A Node is a worker machine in Kubernetes.

Each Node is managed by the Master.

A Node can have multiple pods.

Module 21: Kubernetes Object

Kubernetes Objects is basically a record of intent that you pass on to the Kubernetes cluster.

Once you create the object, the Kubernetes system will constantly work to ensure that object exists.

There are various ways in which we can configure a Kubernetes Object.

- The first approach is through the kubectl commands.
- The second approach is through a configuration file written in YAML.

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mywebserver
spec:
  containers:
  - name: mywebserver
    image: nginx
```

YAML is a human-readable data-serialization language.

It designed to be human friendly and works perfectly with other programming languages.

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

Module 22: Creating First POD Configuration in YAML

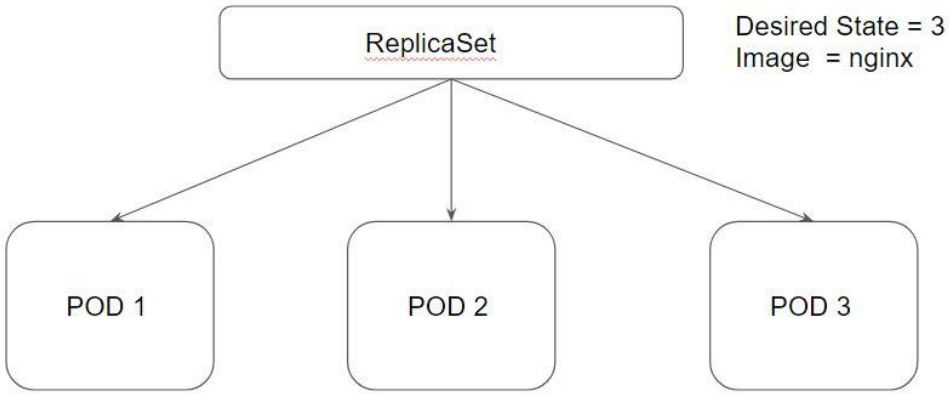
Key	Description
apiVersion	Version of API
kind	Kind of object you want to create.
metadata name	Name of object that uniquely identifies it.
spec	Desired state of the object.

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mywebserver
spec:
  containers:
  - name: mywebserver
    image: nginx
```

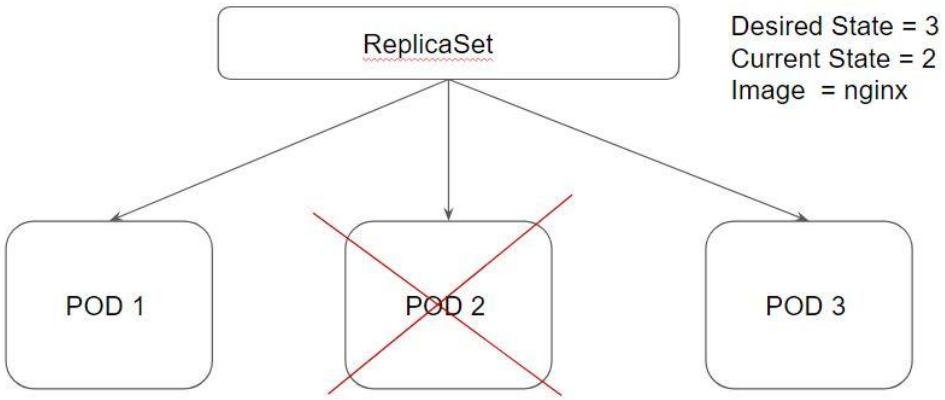
Module 23: ReplicaSets

A ReplicaSet purpose is to maintain a stable set of replica Pods running at any given time.



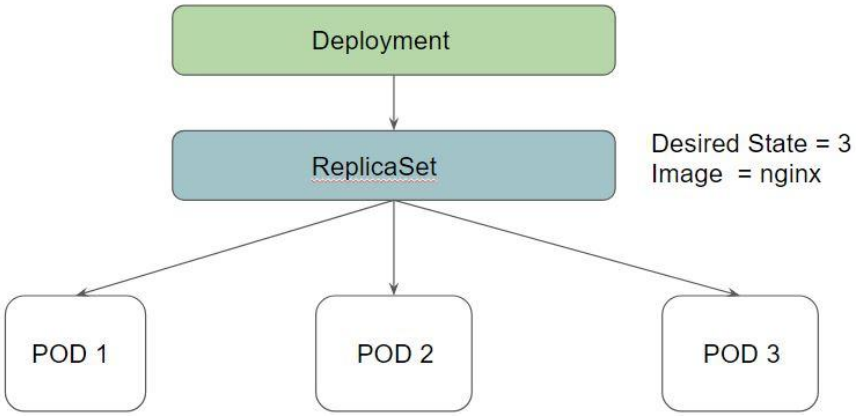
Desired State - The state of pods which is desired.

Current State - The actual state of pods that are running.



Module 24: Deployments

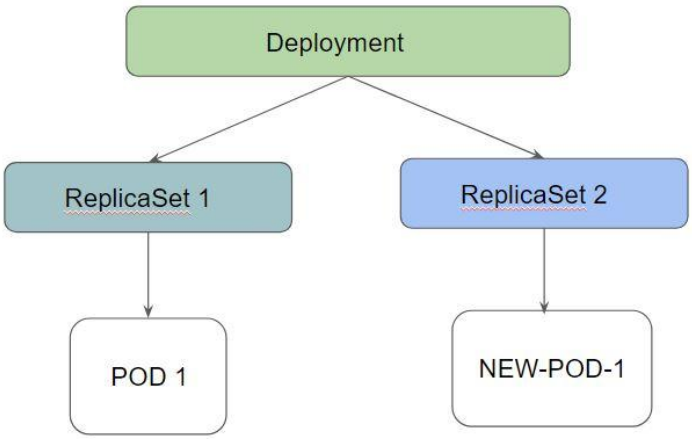
Deployments provide replication functionality with the help of ReplicaSets, along with various additional capability like rolling out of changes, rollback changes if required.



24.1 Benefits of Deployment - Rollout Changes

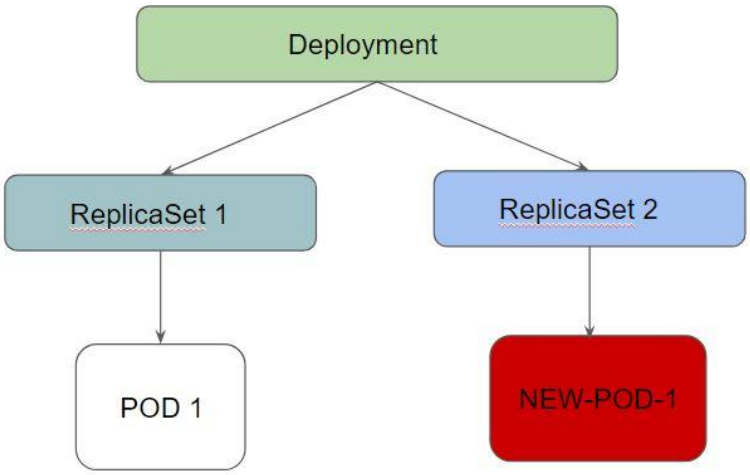
We can easily roll out new updates to our application using deployments.

Deployments will perform an update in a rollout manner to ensure that your app is not down.



24.2 Benefits of Deployment - Rollback Changes

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping



Deployment ensures that only a certain number of Pods are down while they are being updated.

By default, it ensures that at least 25% of the desired number of Pods are up (25% max unavailable).

Deployments keep the history of revision which had been made.

Module 25: Deployment Configuration

While performing a rolling update, there are two important configurations to know.

Configuration Parameter	Description
maxSurge	Maximum Number of PODS that can be scheduled above original number of pods.
maxUnavailable	Maximum number of pods that can be unavailable during the update

maxUnavailable=0 and maxSurge=20% << Full Capacity is maintained.

maxUnavailable=10% and maxSurge=0   << Update with no extra capacity. In-place updates.

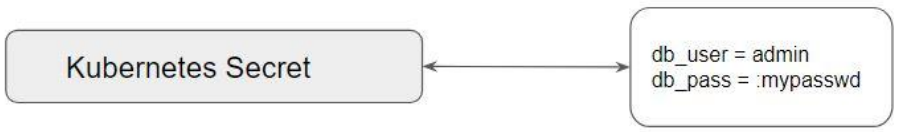
If you want fast rollout, make use of maxSurge.

If there might be a resource quota in place and partial unavailability is acceptable, maxUnavailable can be used.

Module 26: Kubernetes Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.

Allows customers to store secrets centrally to reduce risk of exposure.
Stored in ETCD database.



Following is the syntax for creating a secret via CLI:

kubectl create secret [TYPE] [NAME] [DATA]

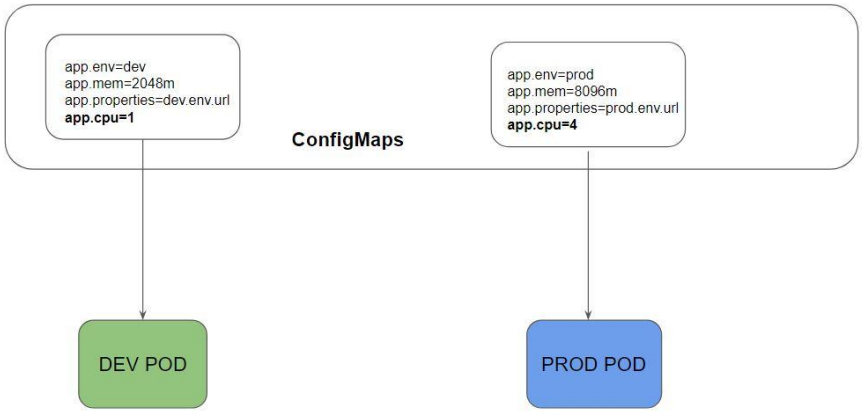
Elaborating Type:

i) Generic:
File (--from-file)
directory
literal value

ii) Docker Registry
iii) TLS

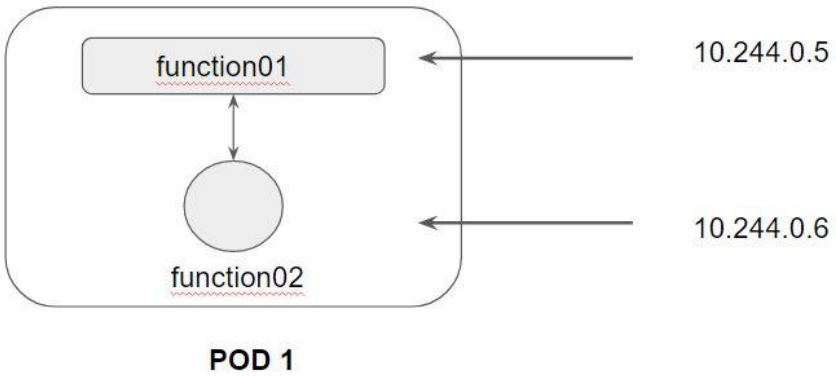
Module 27: ConfigMaps

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

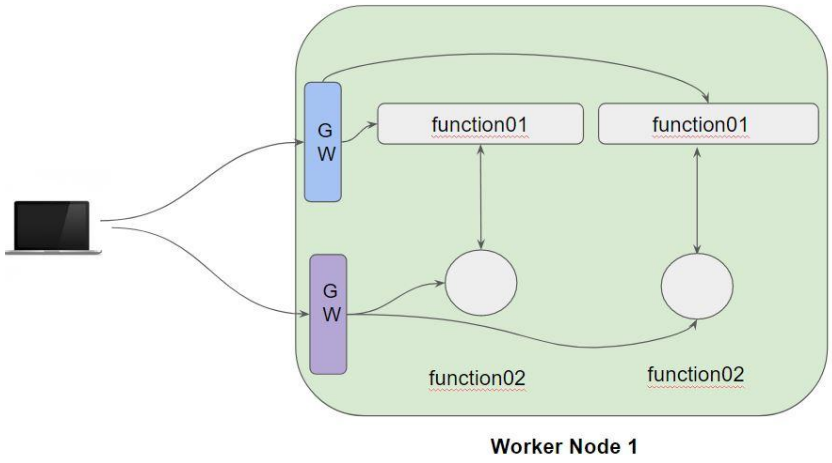


Module 28: Overview of Service

Whenever you create a Pod, the containers created will have Private IP addresses.



Following is a high-level diagram on the functionality of Service:



In a Kubernetes cluster, each Pod has an internal IP address.

Pods are generally ephemeral, they can come and go anytime.

We can make use of service which acts as a gateway and can get us connected with right set of pods.

Service is an abstract way of exposing application running in the pods as a network service.

There are several types of Kubernetes Services which are available:

- NodePort
- ClusterIP
- LoadBalancer
- ExternalName

Module 29: Service Type - ClusterIP

Whenever service type is ClusterIP, an internal cluster IP address is assigned to the service.

Since an internal cluster IP is assigned, it can only be reachable from within the cluster.

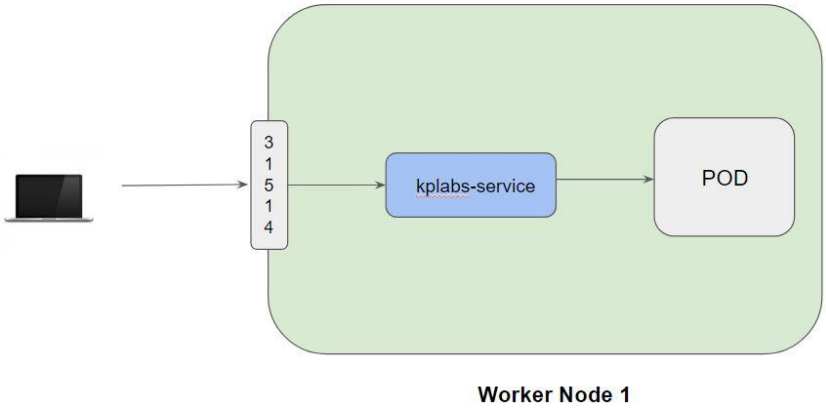
This is a default ServiceType.

Module 30: Service Type - NodePort

From the name, we can identify that it has to do with opening a port on the nodes.

If service type is NodePort, then Kubernetes will allocate a port (default: 30000-32767) on every worker node.

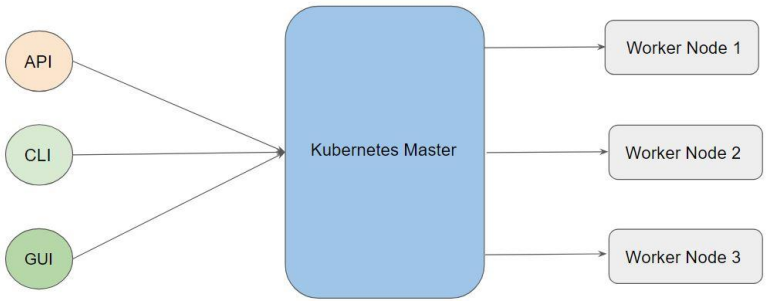
Each node will proxy that port into your service.



Module 31: K8s Networking Model

Kubernetes was built to run on distributed systems where there can be hundreds of worker nodes in which Pods would be running.

This makes networking a very important part component and with the understanding of the Kubernetes networking model, it will allow administrators to properly run, monitor as well as troubleshoot applications in K8s clusters.



Kubernetes imposes the following fundamental requirements on any networking implementation

- pods on a node can communicate with all pods on all nodes without NAT
- all Nodes can communicate with all Pods without NAT.
- the IP that a Pod sees itself as is the same IP that others see it as.

Based on the constraints set, there are four different networking challenges that need to be solved:

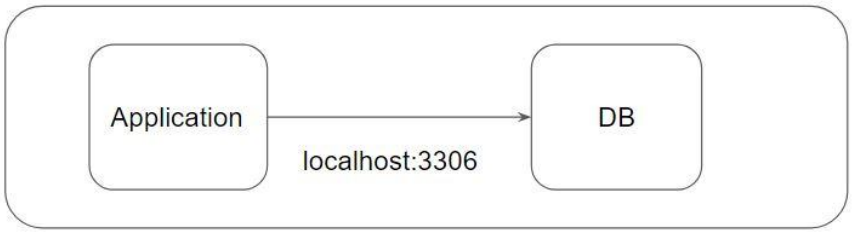
- Container-to-Container Networking
- Pod-to-Pod Networking
- Pod-to-Service Networking
- Internet-to-Service Networking

31.1 - Container-to-Container Networking

Container to Container networking primarily happens inside a pod.

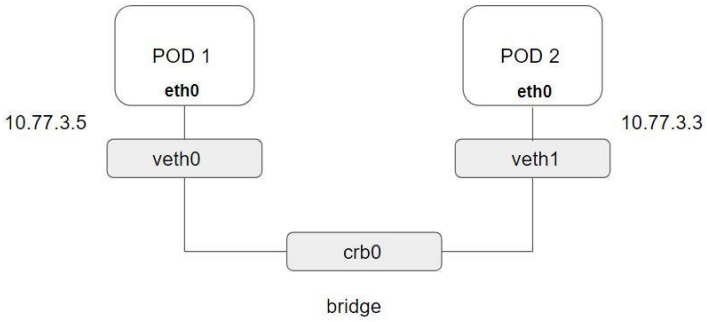
PODs can contain group of containers with the same IP address.

Communication between the containers inside pods happens via localhost



31.2 -Pod to Pod Networking

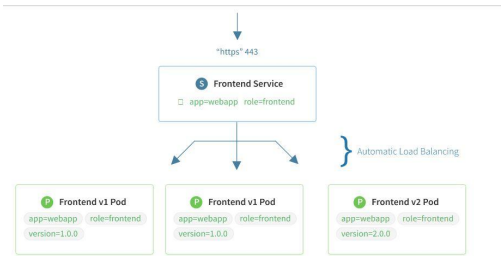
The primary aim is to understand how Pod to Pod communication works.



31.3 -Pod to Service Communication

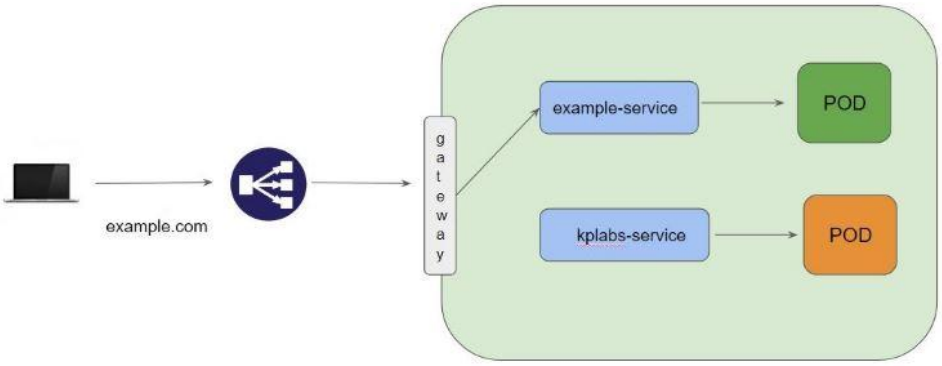
Kubernetes Service can act as an abstraction which can provide a single IP address and DNS through which pods can be accessed.

Endpoints track the IP address of the objects that service can send traffic to.



31.4 -Internet to Service Communication

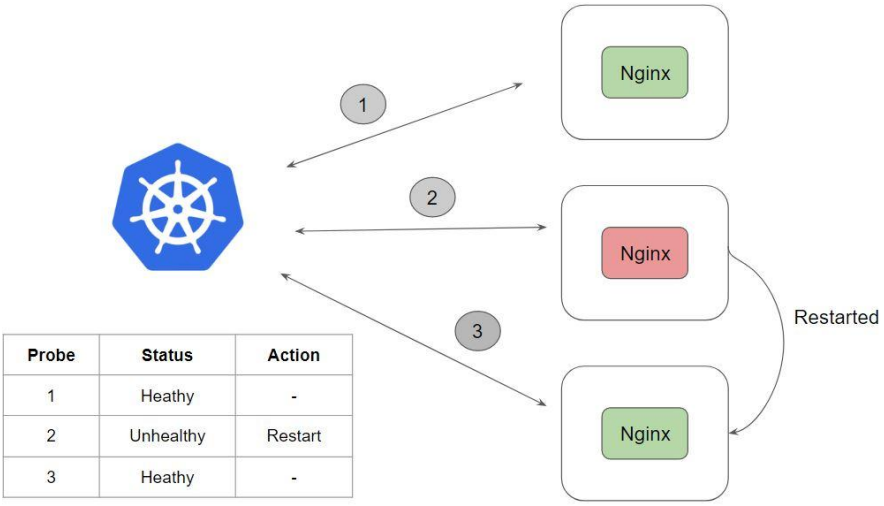
Kubernetes Ingress is a collection of routing rules which governs how external users access the services running within the Kubernetes cluster.



Module 32: Liveness Probe

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted.

Kubernetes provides liveness probes to detect and remedy such situations.



Probe	Status	Action
1	Heathy	-
2	Unhealthy	Restart
3	Heathy	-

There are 3 types of probes which can be used with Liveness

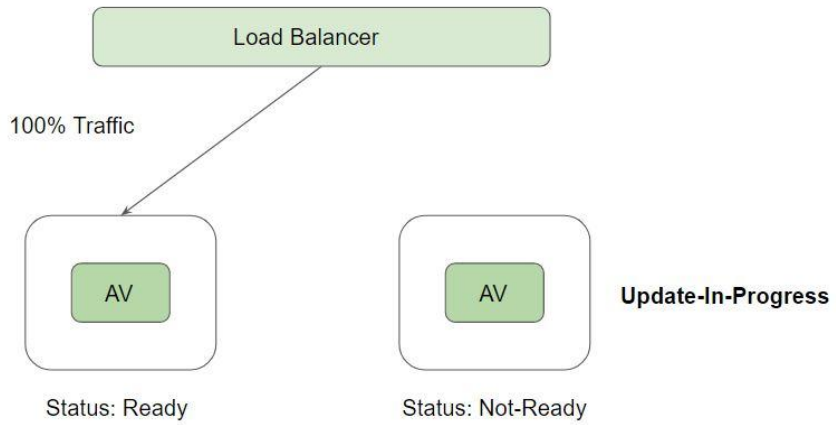
-
- HTTP
- Command
- TCP

Module 33: Readiness Probe

It can happen that an application is running but temporarily unavailable to serve traffic.

For example, the application is running but it is still loading it’s large configuration files from external vendors.

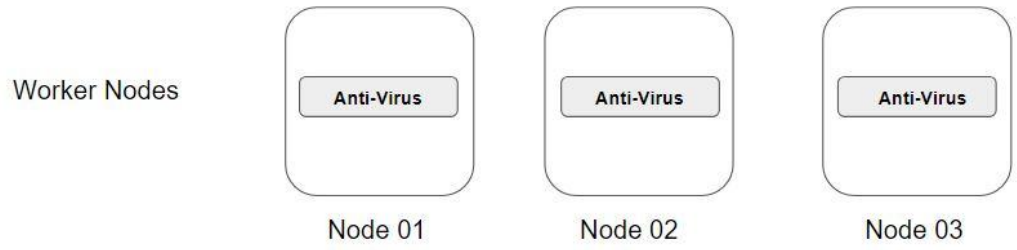
In such a case, we don’t want to kill the container however we also do not want it to serve the traffic.



Module 34: Daemonsets

A DaemonSet can ensure that all Nodes run a copy of a Pod.

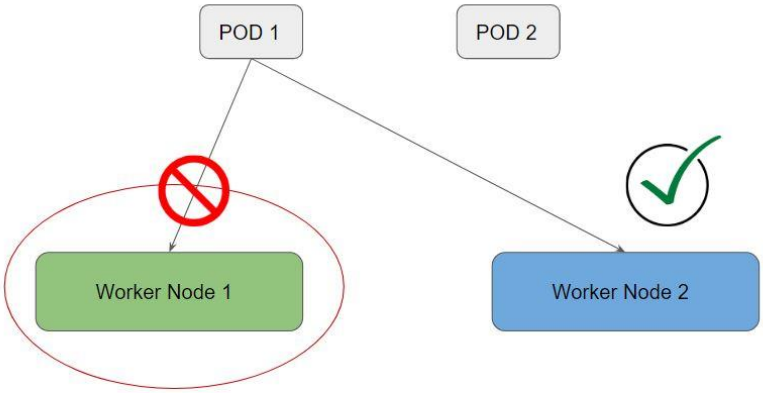
As nodes are added to the cluster, Pods are added to them.



Module 35: Taints & Toleration

33.1 Understanding Taints:

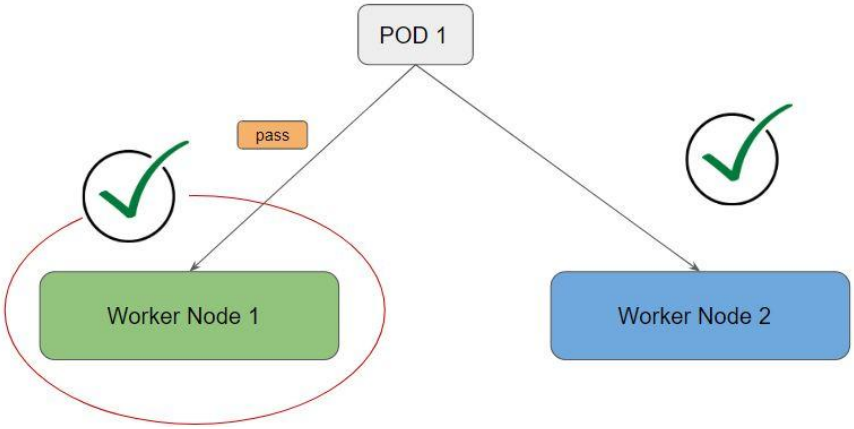
Taints are used to repel the pods from a specific node.



33.2 Understanding Toleration:

In order to enter the taint worker node, you need a special pass.

This pass is called toleration.



Module 36: Labels & Selector

Labels are key/value pairs that are attached to objects, such as pods

Selectors allow us to filter objects based on labels.

Example:

Show me all the objects which have a label where env: prod

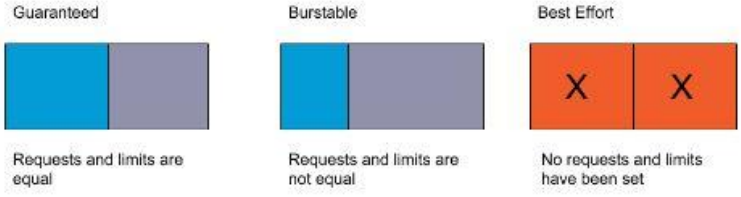
.Module 37: Requests and Limits

Requests and Limits are two ways in which we can control the amount of resource that can be assigned to a pod (resource like CPU and Memory)

Requests: Guaranteed to get.
Limits: Makes sure that the container does not take node resources above a specific value.

Kubernetes Scheduler decides the ideal node to run the pod depending on the requests and limits.

If your POD requires 8GB of RAM, however, there are no nodes within your cluster which has 8GB RAM, then your pod will never get scheduled.


Guaranteed
Burstable
Best Effort
X
X
Requests and limits are equal
Requests and limits are not equal
No requests and limits have been set