KUBERNETES ADMINISTRATOR

Gaurav

RHCSA | RHCE | CKA | SCA | SCE | SCA+ HA | ANSIBLE | DOCKER | OPENSHIFT` | OPEN SOURCE TECHNOLOGIES

IT Learning Solution

TABLE OF CONTENT

- Core Concepts
- Installation of Kubernetes Cluster
- Using Kubernetes Features
- Networking in Kubernetes
- Security in Kubernetes
- Storage
- Logging & Monitoring

WHAT IS CONTAINER?

Containers are a way for developers to easily package and deliver applications, and for operations to easily run them anywhere in seconds, with no installation or setup necessary

WHY CONTAINER?

Containers require less system resources than traditional or hardware virtual machine environments because they don't include operating system images. Increased portability. Applications running in containers can be deployed easily to multiple different operating systems and hardware platforms.

CONTAINER ORCHESTRATION

SCENARIO

SCENARIO OF GUARD

- You are owner of a building and you have 5 spots where people can enter your building and you want 5 security guards guarding the spots. All good till now.

- Now consider one of the guard was out of service for 2 hours due to some personal reasons. Now as a building owner its your responsibility to guard or employ another guard replacing the existing. Do you like to be manually interrupted from your task to look after who is out and whom to replace.

- No, no one likes to be. Now the solution could be, go to a third party vendor who provides 24*7 availability of the guards. Its the responsibility of the vendor to make 24*7 availability based on the requirements.

WHAT PROBLEM DOES KUBERNETES SOLVE?

Now lets replace the characters.

- Building -> Your application
- Owner -> The application owner
- Guards -> Containers
- Vendor -> Kubernetes
- Configuration (repliica)-> That we make a deal with vendor to maintain appropriate conf 24*7.

CONTAINER ORCHESTRATION

- Container orchestration automates the deployment, management, scaling, and networking of containers. Enterprises that need to deploy and manage hundreds or thousands of Linux® containers so in this case we don’t want downtime so container orchestration will give this facility.

What is container orchestration used for?

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Container availability
- Scaling or removing containers based on balancing workloads across your infrastructure
- Load balancing and traffic routing
- Monitoring container health
- Configuring applications based on the container in which they will run
- Keeping interactions between containers secure

TOOLS

- DOCKER SWARM
- KUBERNETES

DIFFERENCE BETWEEN DOCKER SWARM AND KUBERNETES

INSTALLATION CLUSTER CONFIGURATION

DOCKER SWARM

- SETTING UP THE CLUSTER IS SIMPLE
- REQUIRES ONLY 2 COMMANDS

KUBERNETES

- SETTING UP THE CLUSTER IS CHALLENGING AND COMPLICATED

USER INTERFACE

DOCKER SWARM

- THERE IS NOT GUI AVAILABLE

KUBERNETES

- PROVIDES A GUI (KUBERNETES DASHBOARD)

SCALABILITY

DOCKER SWARM

- Scalling up is 5X faster than kubernetes
- Better when 10-20 containers

KUBERNETES

- Scalling up is easy
- Better when 100-1000 containers in pod

AUTO SCALING

DOCKER SWARM

- Scaling up or scaling down has to be done manually

KUBERNETES

- Based on server traffic, containers will be scalled automatically

LOAD BALANCING

DOCKER SWARM

- SWARM DOES AUTO LOAD BALANCING

KUBERNETES

- Manual configuration needed for load balancing traffic

ROLLING UPDATE AND ROLLBACKS

DOCKER SWARM

- Rolling updates progressively updates the containers one after the other while ensuring HA
- No automatic Rollbacks

KUBERNETES

- Rolling updates progressively updates the pods one after the other while ensuring HA
- Automatic Rollbacks in case of failure

DATA VOLUMES

DOCKER SWARM

- Storage volumes can be shared with any other container in the node

KUBERNETES

- Storage volumes can be shared only between containers within the same pod

LOGGING AND MONITORING

DOCKER SWARM

- 3rd party logging and monitoring tools required

KUBERNETES

- In-built logging and monitoring tools in place

WHAT IS KUBERNETES

Kubernetes is a system for managing containerized applications across a cluster of nodes.

Why you need Kubernetes and what can it 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!

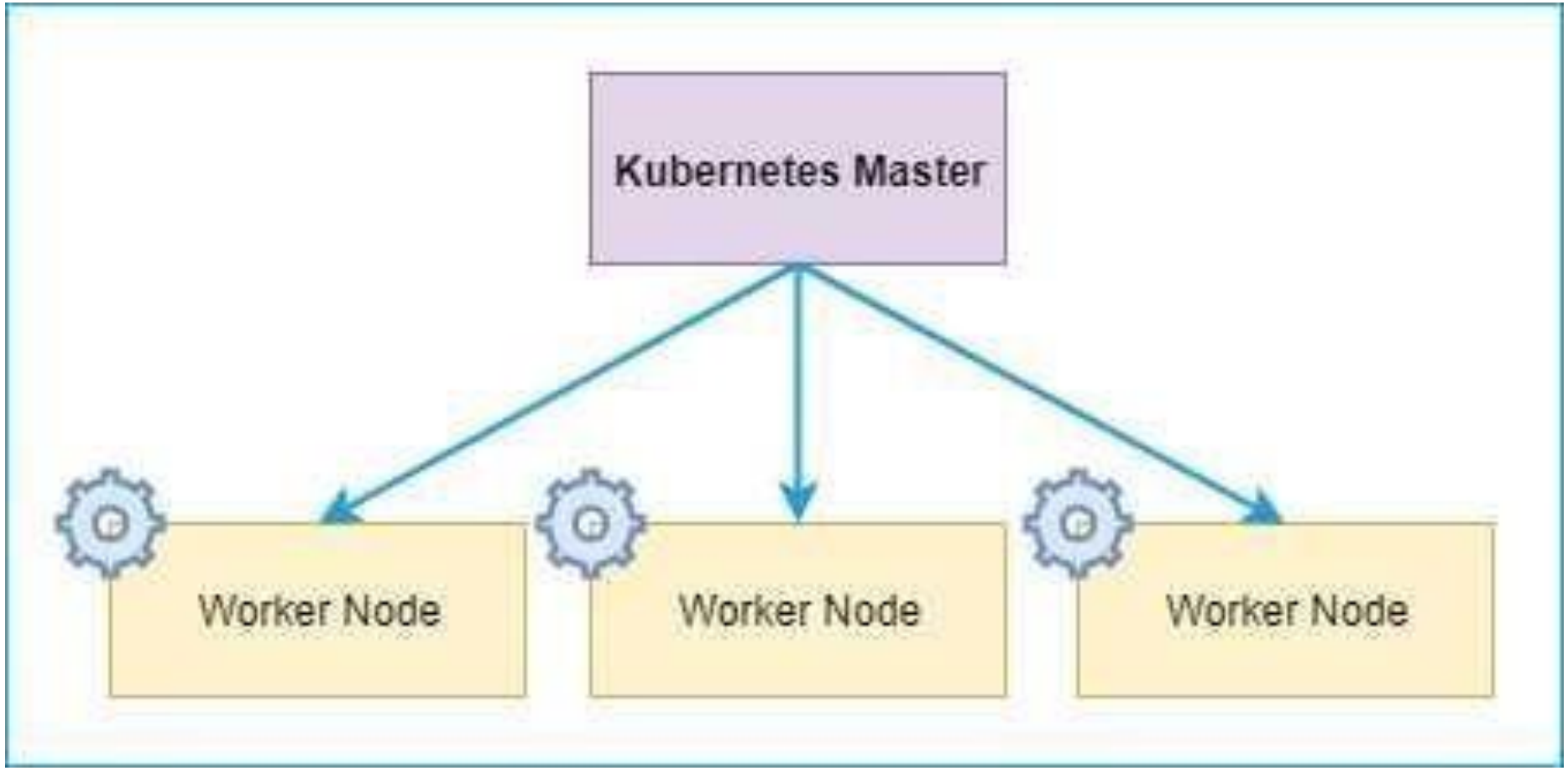
KUBERNETES PROVIDES

- Service Discovery and Load Balancing
- Storage Orchestration
- Automated rollouts and rollback
- Automatic provides nodes as per container requirement
- Self-healing
- Secret and configuration management

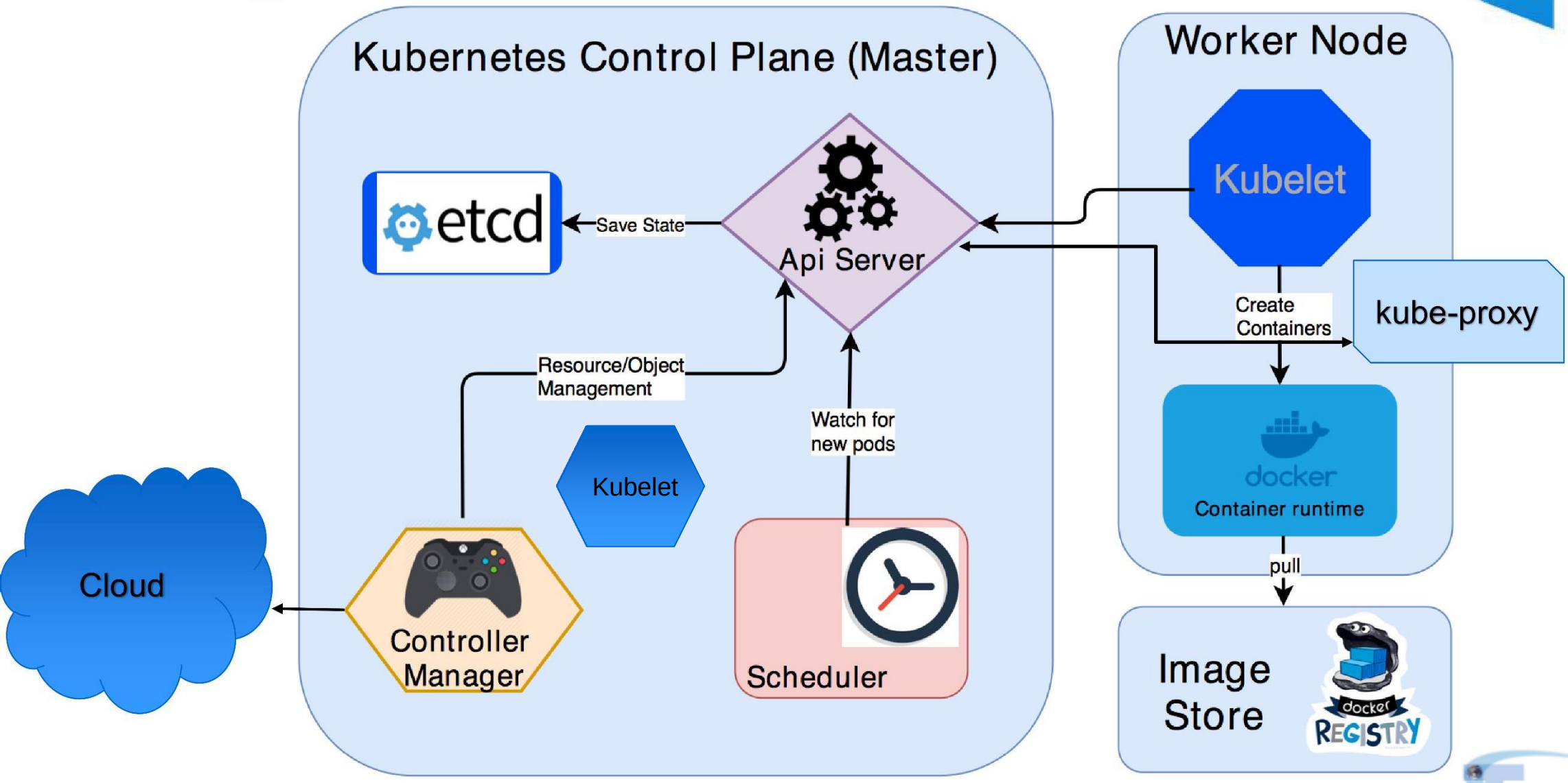
kubernetes

- Self-Healing
- Automated Rollbacks
- Auto Scaling
- Load Balancing

KUBERNETES COMPONENTS



KUBERNETES COMPONENTS / ARCHITECTURE



KUBE-APISERVER

Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

Kube-api Server

1. Authenticate User
2. Validate Request
3. Retrieve data
4. Update ETCD
5. Scheduler
6. Kubelet

ETCD

- ETCD is a distributed reliable key-vaue store that is simple,secure and fast.
- It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes.
- It is storing all the information of master and worker nodes
- It is accessible only by Kubernetes API server as it may have some sensitive information.
- It is a distributed key value Store which is accessible to all.

ETCD STORE information of

1) Nodes
2) POD's
3) Configs
4) Secrets
5) Accounts
6) Roles
7) Binding

KUBE-CONTROLLER-MANAGER

Controllers are the brain behind the orchestration. They are responsible for noticing and responding when nodes, containers or end point goes down. The controllers make decision to make up new containers in such cases. This component is responsible for most of the collectors that regulates the state of cluster and performs a task.

KUBE -CONTROLLER-MANAGER

Some of the controllers that controller-manager include:

- **Node Controller**: Responsible for noticing and responding when nodes go down.
- **Replication Controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- **Endpoints Controller**: Populates the Endpoints object (that is, joins Services & Pods).
- **Service Account & Token Controllers**: Create default accounts and API access tokens for new namespaces.

NODE CONTROLLER

- 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

Controller

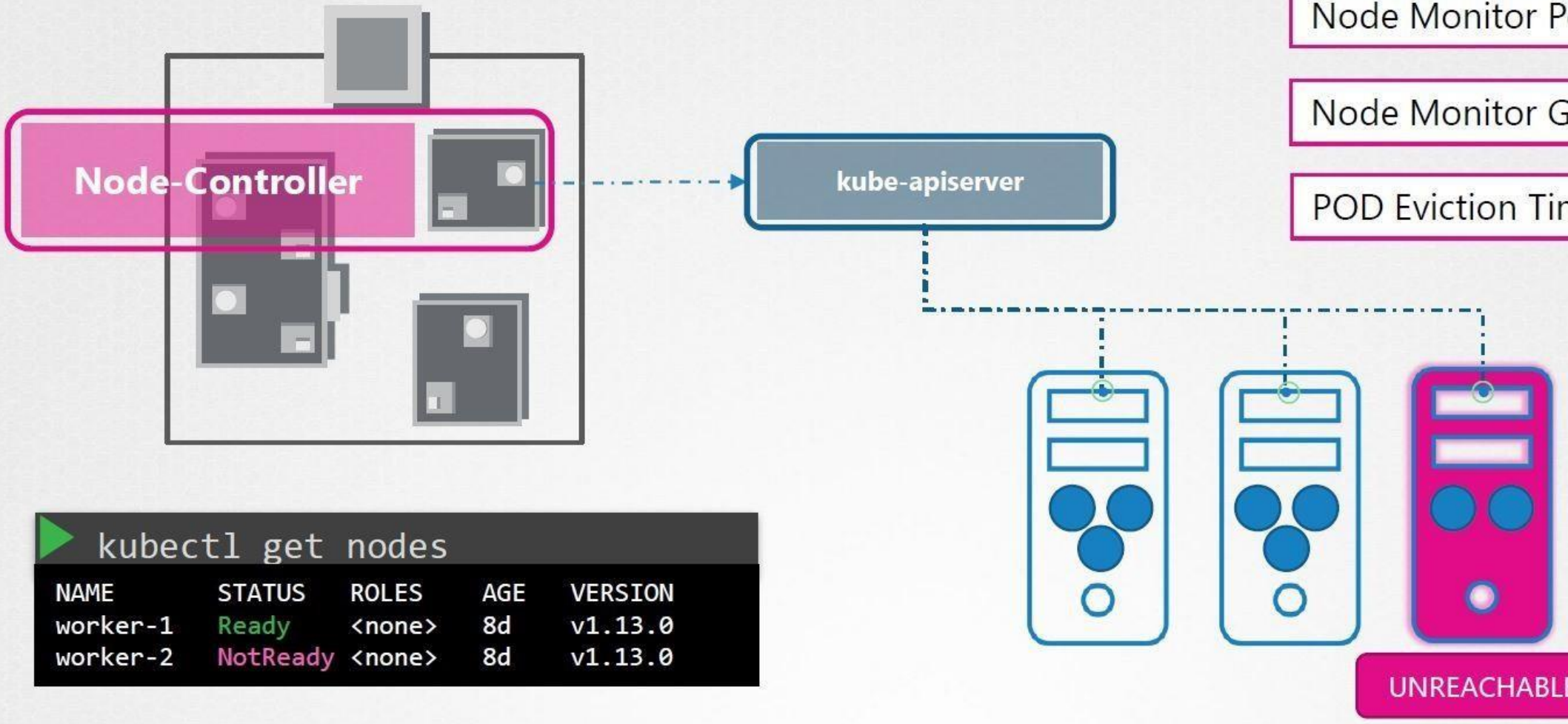
Watch Status

Remediate Situation

Node Monitor Period = 5s

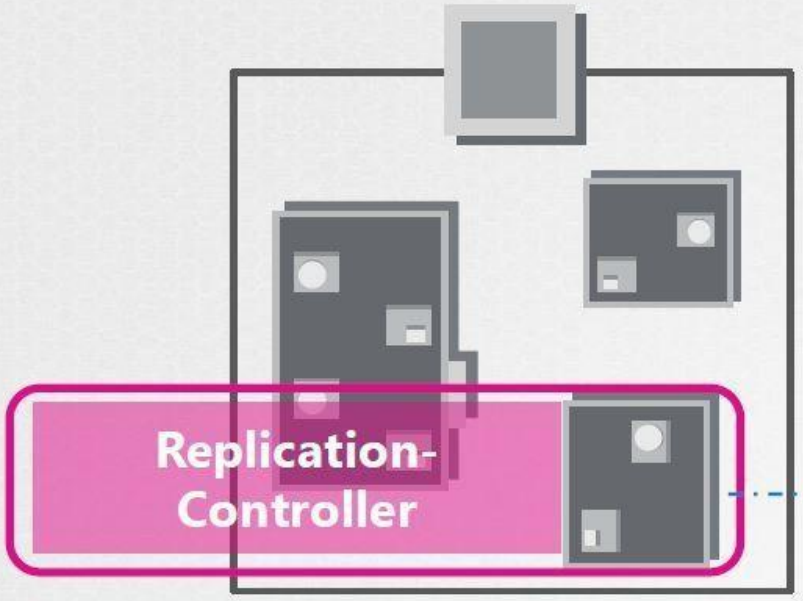
Node Monitor Grace Period = 40s

POD Eviction Timeout = 5m




```
kubectl get nodes
NAME       STATUS     ROLES    AGE   VERSION
worker-1   Ready      <none>   8d    v1.13.0
worker-2   NotReady   <none>   8d    v1.13.0
```

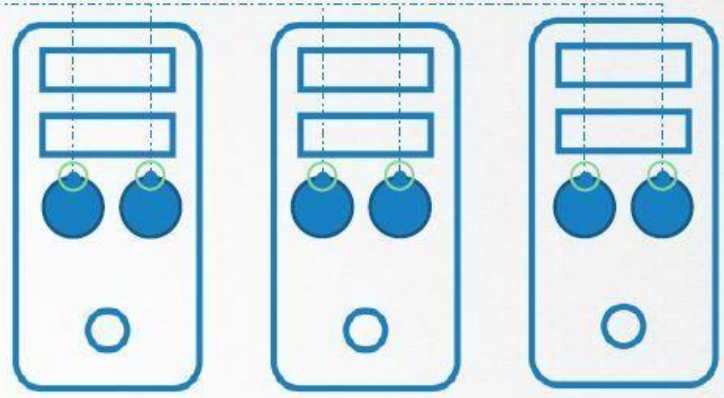
Controller



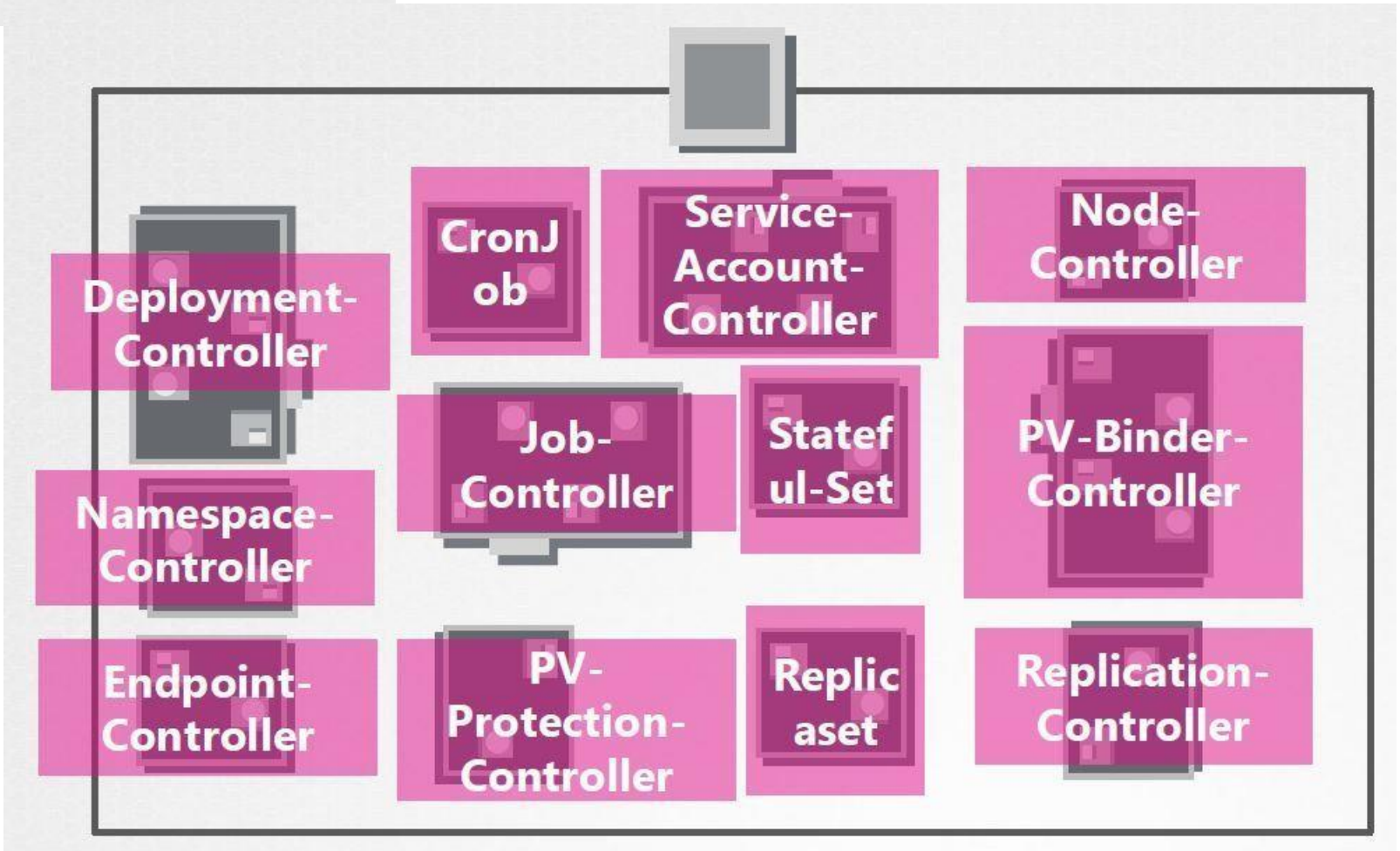
kube-apiserver

- Watch Status
- Remediate Situation
- Node Monitor Period = 5s
- Node Monitor Grace Period = 40s
- POD Eviction Timeout = 5m

POD



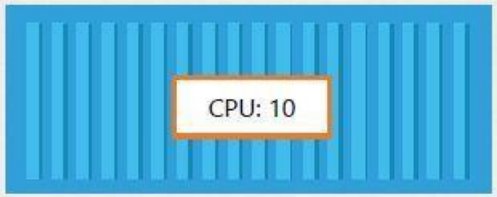
Controller



KUBE-SCHEDULER

This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

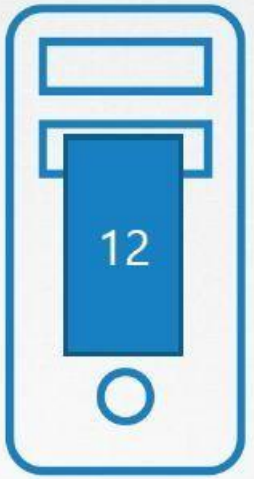
Kube-Scheduler



3rd - 12 - 10 = 2
4th - 16 - 10 = 6

1. Filter Nodes

2. Rank Nodes



CONTAINER RUNTIME

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

- Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

CONTAINER RUNTIME

containerd

rkt

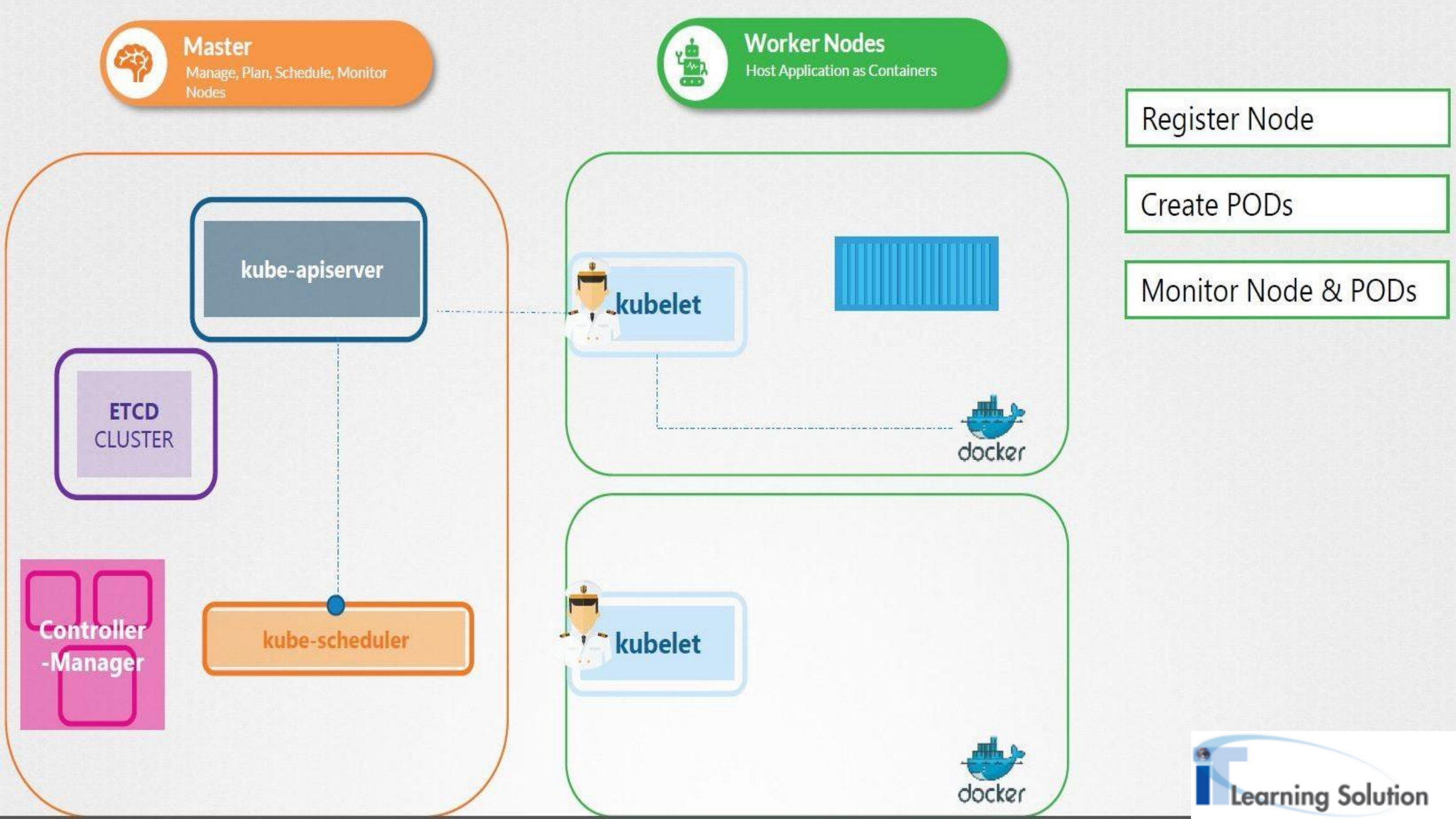
frakti

cri-o

KUBELET

It is the agent that runs on each node in the cluster. It is responsible for various tasks:

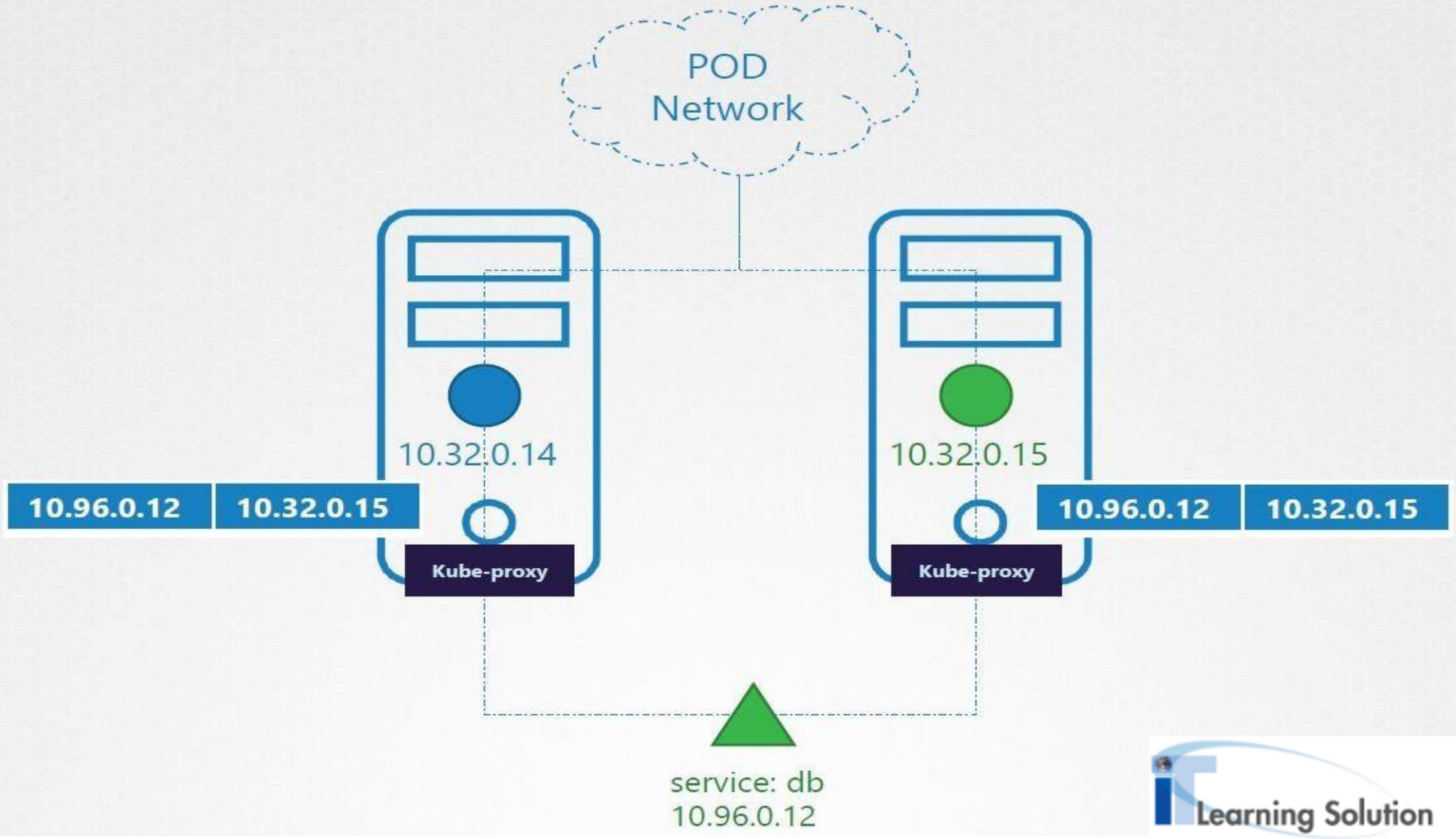
- making sure that the containers are running on the node as expected.
- for relaying information to and from control plane service through API server.
- It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work.
- The kubelet process then assumes responsibility for maintaining the state of work and the node server.
- It manages network rules, port forwarding, etc.


Master
Manage, Plan, Schedule, Monitor Nodes
Worker Nodes
Host Application as Containers
Register Node
Create PODs
Monitor Node & PODs
kube-apiserver
ETCD CLUSTER
Controller -Manager
kube-scheduler
kubelet
kubelet
docker
docker
Learning Solution

KUBE-PROXY

- This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers' health checkup, etc.

Kube-proxy



KUBERNETES INSTALLATION

REQUIREMENTES

One or more machines running one of:

- Ubuntu 16.04+
- Debian 9+
- CentOS 7
- Red Hat Enterprise Linux (RHEL) 7
- Fedora 25+
- HypriotOS v1.0.1+
- Container Linux (tested with 1800.6.0)

REQUIREMENTS

- 2 GB or more of RAM per machine (any less will leave little room for your apps)
- 2 CPUs or more
- Full network connectivity between all machines in the cluster (public or private network is fine)
- Unique hostname, MAC address, and product_uuid for every node.
- Certain ports are open on your machines.
- Swap disabled. You MUST disable swap in order for the kubelet to work properly.

CHECK REQUIRED PORTS

Control-plane node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services†	All

CONFIGURATION

ON ALL NODES

- Disable SWAP

#vim /etc/fstab

then remove the entry of swap or comment

#swapoff -a -> to disable swap

•Set selinux to permissive mode

#vim /etc/sysconfig/selinux

SELINUX=permissive

#setenforce 0

•Stop firewall service

#systemctl stop firewalld

#systemctl disable firewalld

INSTALL CONTAINER RUNTIME

ON ALL NODES

- #wget https://download.docker.com/linux/centos/docker-ce.repo -O /etc/yum.repos.d/docker-ce.repo
- # yum install docker-ce -y
- # systemctl start docker
- # systemctl enable docker
- # systemctl status docker

Installing kubeadm, kubelet and kubectl

- kubeadm: the command to bootstrap the cluster.
- kubelet: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- kubectl: the command line util to talk to your cluster.

DEBIAN BASED

- #apt-get update && sudo apt-get install -y apt-transport-https curl
- #curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
- #cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list

  deb https://apt.kubernetes.io/ kubernetes-xenial main

  EOF
- #apt-get update
- #apt-get install -y kubelet kubeadm kubectl
- #apt-mark hold kubelet kubeadm kubectl

RPM BASED

ON ALL NODES

#vim /etc/yum.repos.d/kubernetes.repo

```
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

#yum install kubelet kubectl kubeadm

#systemctl enable kubelet

#systemctl start kubelet

NETWORK SOLUTION

We have many types of Network Solution->

- Calico
- Cilium
- Contiv-VPP
- Kube-router
- Weave Net

(doc -> https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/)

INITIALIZE THE CLUSTER

- Now initialize the master node->

#kubeadm init --pod-network-cidr=10.244.0.0/16

- Then after execution of above command you will have some commands so do all showing commands to configure kube configuration

- Run the join command in every worker node

NETWORK SOLUTION

- we are using calico network for the installation

#kubectl apply -f https://docs.projectcalico.org/v3.14/manifests/calico.yaml

VERIFY INSTALLATION

- #kubectl get nodes
- #kubectl get pods
- #kubectl get pods -A

BASH COMPLETION

- #yum install bash-completion -y
- #cd ~/.kube
- #kubectl completion bash > kube.sh
- #source kube.sh

To make it permanent

- #vim $HOME/.bashrc
- Add a entry

source $HOME/.kube/kube.sh

NODES

A node is a worker machine in Kubernetes, previously known as a minion. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components.

NODE STATUS

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

A node's status contains the following information:

- Addresses
- Conditions
- Capacity and Allocatable
- Info

NODES STATE

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)
`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`
`DiskPressure`	`True` if pressure exists on the disk size – that is, if the disk capacity is low; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

KUBECTL

#kubectl run <name> --image=<name>

#kubectl cluster-info

#kubectl get nodes

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

#kubectl get pods

#kubectl describe pod <pod-name>

#kubectl get pods -A

#kubectl get pods -o wide

#kubectl get pods -A --show-labels

PODS

- A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it. For example, keeping a database container and data container in the same pod.

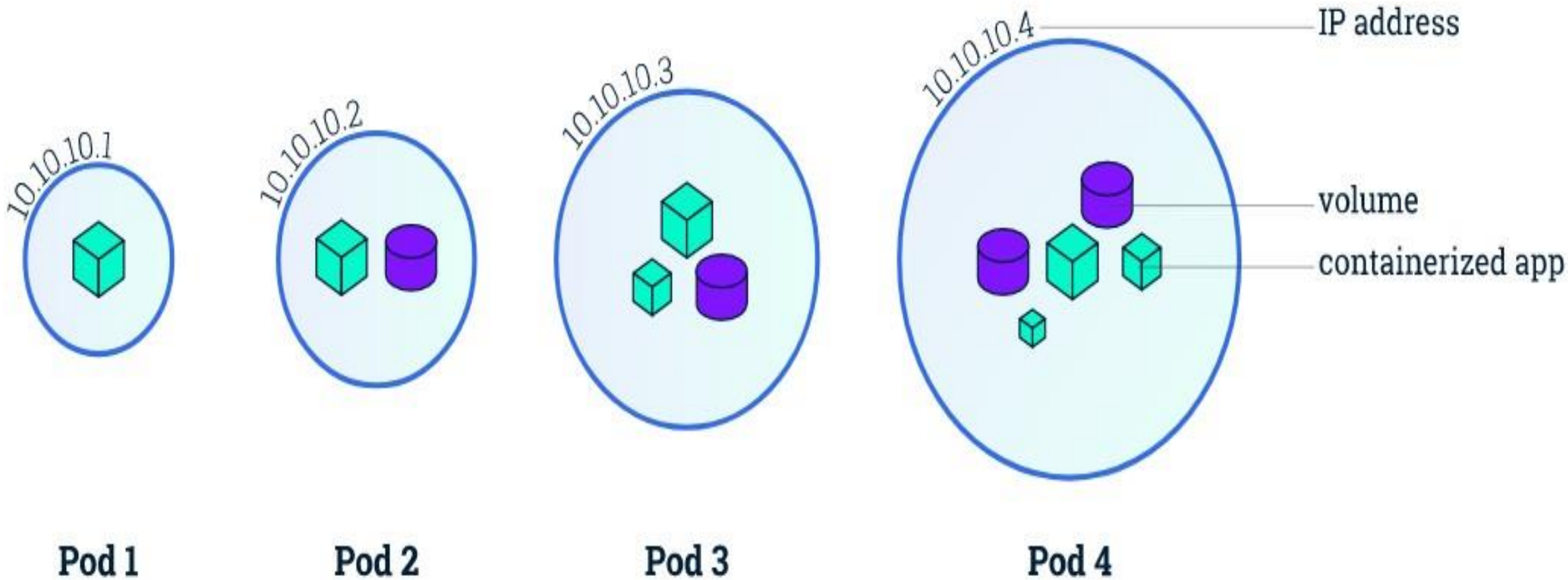
- There are two types of Pods –
  - Single container pod
  - Multi container pod

PODS

- Pods provide two kinds of shared resources for their constituent containers: networking and storage.

1)Networking-> Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports.

2)Storage -> A Pod can specify a set of shared storage Volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data.



HOW TO CREATE PODS

- #kubectl run <podname> --image=nginx
- #kubectl run <podname> --dry-run=client --image=nginx
- #kubectl get pods
- #kubectl get pods -A
- #kubectl get pods -o wide
- #kubectl get pods -A --show-labels
- #kubectl describe pods <podname>
- #kubectl explain <resource> like pod,deployment
- #kubectl exec -it <podname> bash

YAML INTRODUCTION

- Four fields are required to create yaml file->

1)apiVersion - Which version of the Kubernetes API you're using to create this object

2) kind - What kind of object you want to create

3)metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace

4) spec - What state you desire for the object

apiVersion

1) apiVersion: -> version of api depend on the module what we are creating

at the time of creating yaml file we need to specify version (like:- service then in apiversion we will use v1) use like below options

kind	**Version**
Pod	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

HOW TO CREATE POD

#vim pod.yml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: new-pod
5  spec:
6    containers:
7    - name: nginx-container
8      image: nginx
9
10
```

#kubectl create -f pod.yml
#kubectl create -f pod.yml --dry-run=client
#kubectl delete -f pod.yml

INIT CONTAINER

- Init Containers are containers that run before the main container runs with your containerized application
- They are specialized containers that runs before app containers in a pod.
- Init containers can contain utilities or setup scripts not present in an app image.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

NOTE:- If a Pod's init container fails, Kubernetes repeatedly restarts the Pod until the init container succeeds.

INIT CONTAINER

```yaml
apiVersion: v1
kind: Pod
metadata:
   name: mypod
spec:
     containers:
       - name: firstcontainer
         image: nginx
         args: ["sleep","50"]
       - name: secondcontainer
         image: nginx
     initContainers:
       - name: initcontainer
         image: nginx
         args: ["sleep","30"]
```

LABELS AND SELECTORS

Labels :

- Labels enable end users to map their own, custom, organizational structures onto system resources in a loosely coupled fashion.

Selectors :

- Labels do not provide uniqueness. In general, we can say many objects can carry the same labels. Labels selector are core grouping primitive in Kubernetes. They are used by the users to select a set of objects or to filter the tags.

EXAMLE OF LABEL

```yaml
apiVersion: v1
kind: Pod
metadata:
   name: example-pod
   labels:
     env: development
 spec:
   containers:
   - name: label-example
     image: nginx
     ports:
     - containerPort: 80
```

LABELS COMMANDS

#kubectl apply -f <filename>

#kubectl get pods

#kubectl get pods <podname>--show-labels

#kubectl label pod <podname> tier=backend

#kubectl get pod <podname> --show-labels

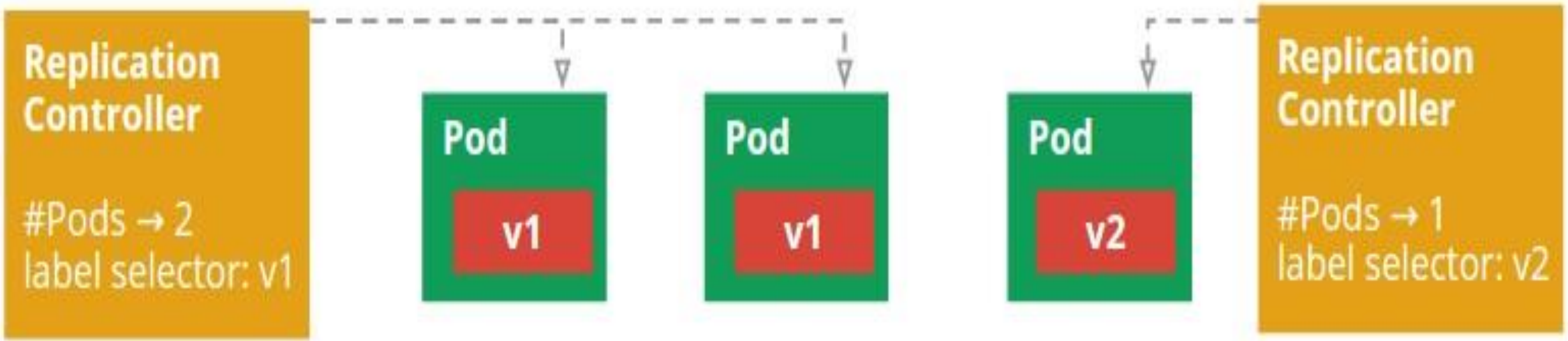
#kubectl label pod <podname> tier-

#kubectl label --overwrite pods <podname> env=prod

SELECTOR COMMANDS

- kubectl get pods -l env=prod
- kubectl get pods -l env!=prod

Kubernetes Replication Controllers



Behavior

- Keeps Pods running → Restarts Pods, desired state
- Gives direct control of Pod #s → Fine-grained control for scaling

Benefits

replica_controller.yml

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
   name: rc-nginx
spec:
   replicas: 3
   template:
       metadata:
            name: nginx
            labels:
              app: nginx
       spec:
         containers:
         - name: nginx
           image: nginx
```

Pod definition

#kubectl create -f replica_controller.yml

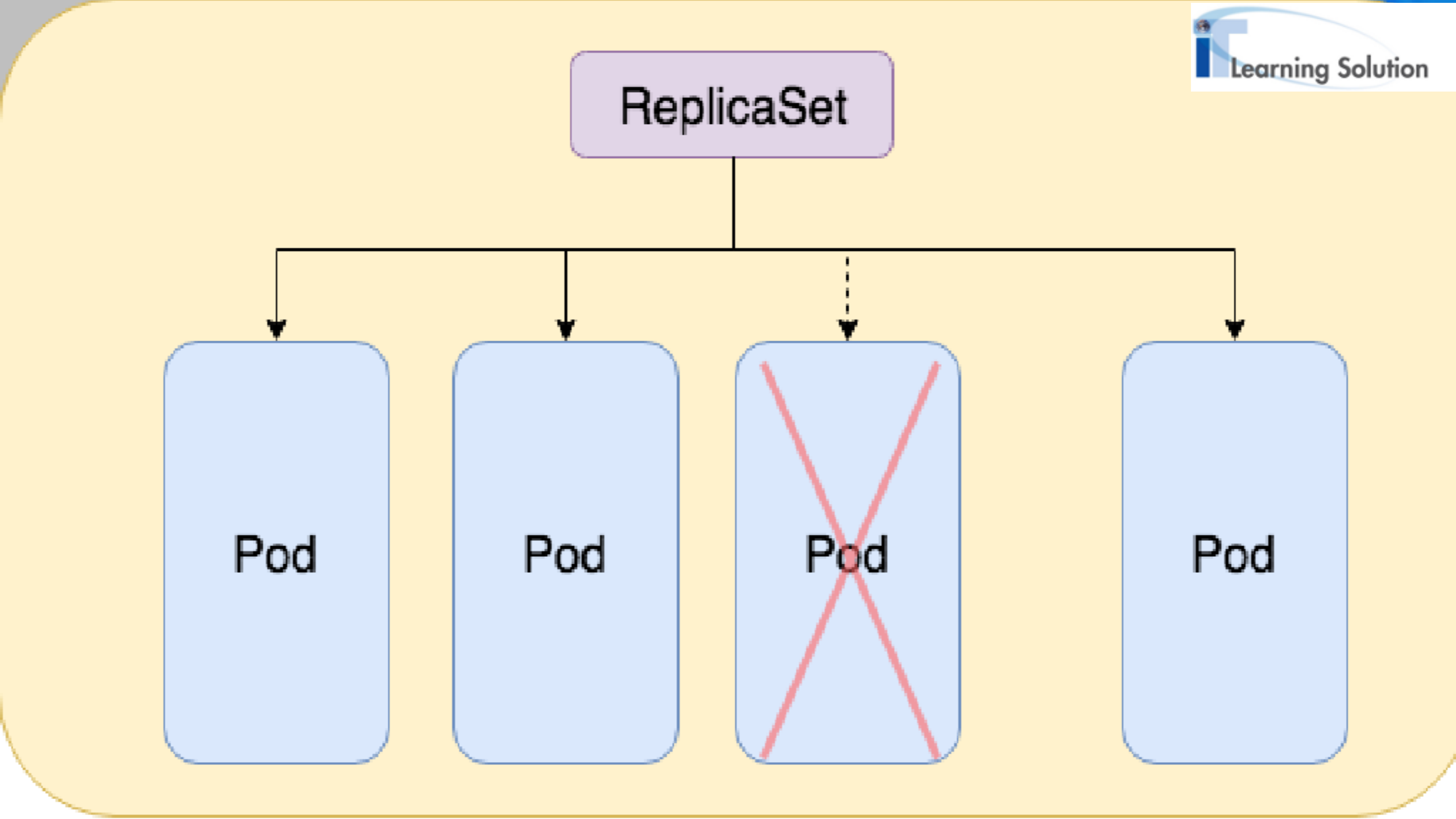
#kubectl get replicationcontroller
or
#kubectl get rc
#kubectl get pods
#kubectl get pods -o wide
#kubectl get pods -A --show-labels

ReplicaSet

- A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

- How ReplicaSet Works?

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.



replica_set.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Will match the labels from already created pods also

#kubectl create -f replica_set.yml

#kubectl get replicaset

or

#kubectl get rs
#kubectl get pods
#kubectl get pods -o wide
#kubectl get pods -A --show-labels

If we want to increase or decrease the replicas then just modify the definition file
replicas: 6

#kubectl replace -f replica_set.yml
or
#kubectl scale --replicas=6 -f replica_set.yml
or
#kubectl scale --replicas=6 replicaset rs-nginx

REPLICATION CONTROLLER VS REPLICA SET

- ReplicaSet requires a selector definition. The selector helps to identify what pods are already created.
- Selector definition is not required for replication controller but we can specify.
- For example the pods created before the creation of replicaset that match same labels specified in the selector the replica set will automatically also take those pods into consideration when creating the replica set.

SERVICE

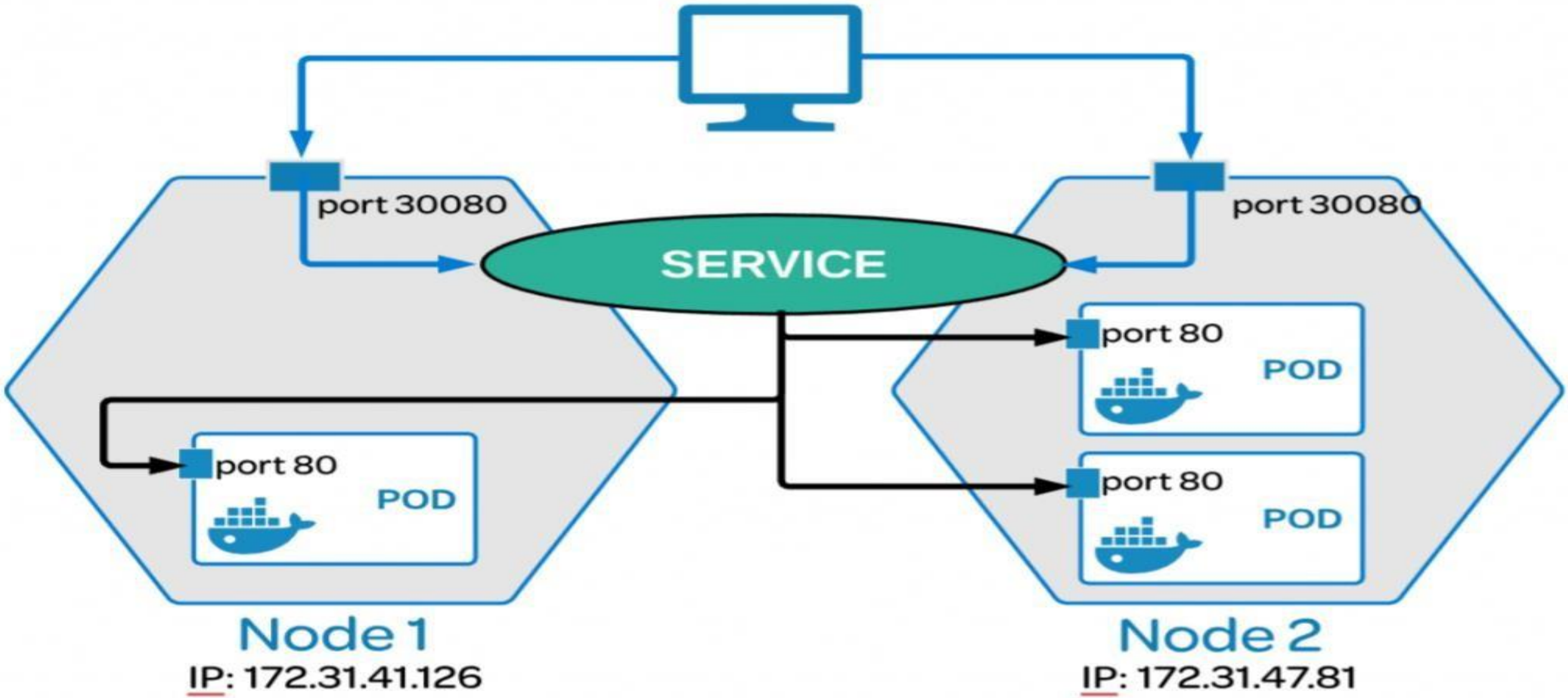
- Services are resources in Kubernetes (like pods, deployments, etc.) that provide a single point of access from the outside world, into your pod(s) which run your application.

- Each service has an IP address and a port that never change, and the client always knows where to find the pods due to the flat nat-less network.

Kubernetes Service

Learning Solution

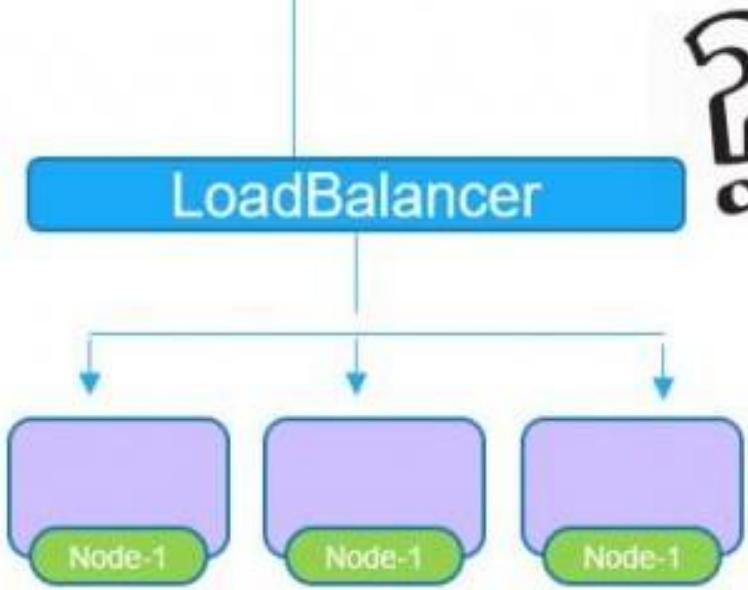
A service allows you to dynamically access a group of replica pods.



Types of Services

- Reachable within the cluster.
- Connects Frontend Pods to Backend Pods

- Exposing Frontend app to external world

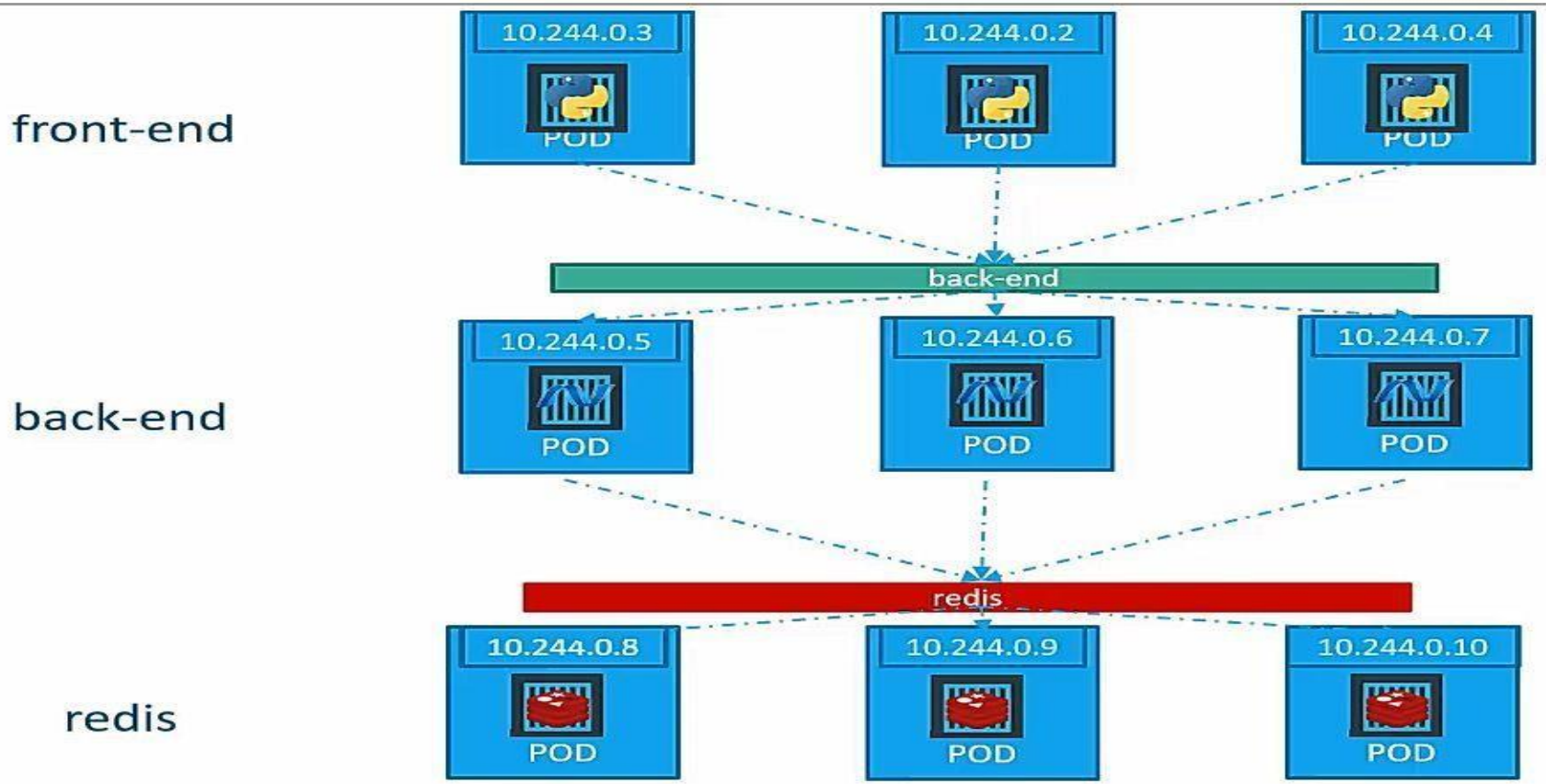


- Equally distribute the loads

CLUSTER IP

- ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.

ClusterIP


front-end
back-end
redis
10.244.0.3
POD
10.244.0.2
POD
10.244.0.4
POD
back-end
10.244.0.5
POD
10.244.0.6
POD
10.244.0.7
POD
redis
10.244.0.8
POD
10.244.0.9
POD
10.244.0.10
POD

APP

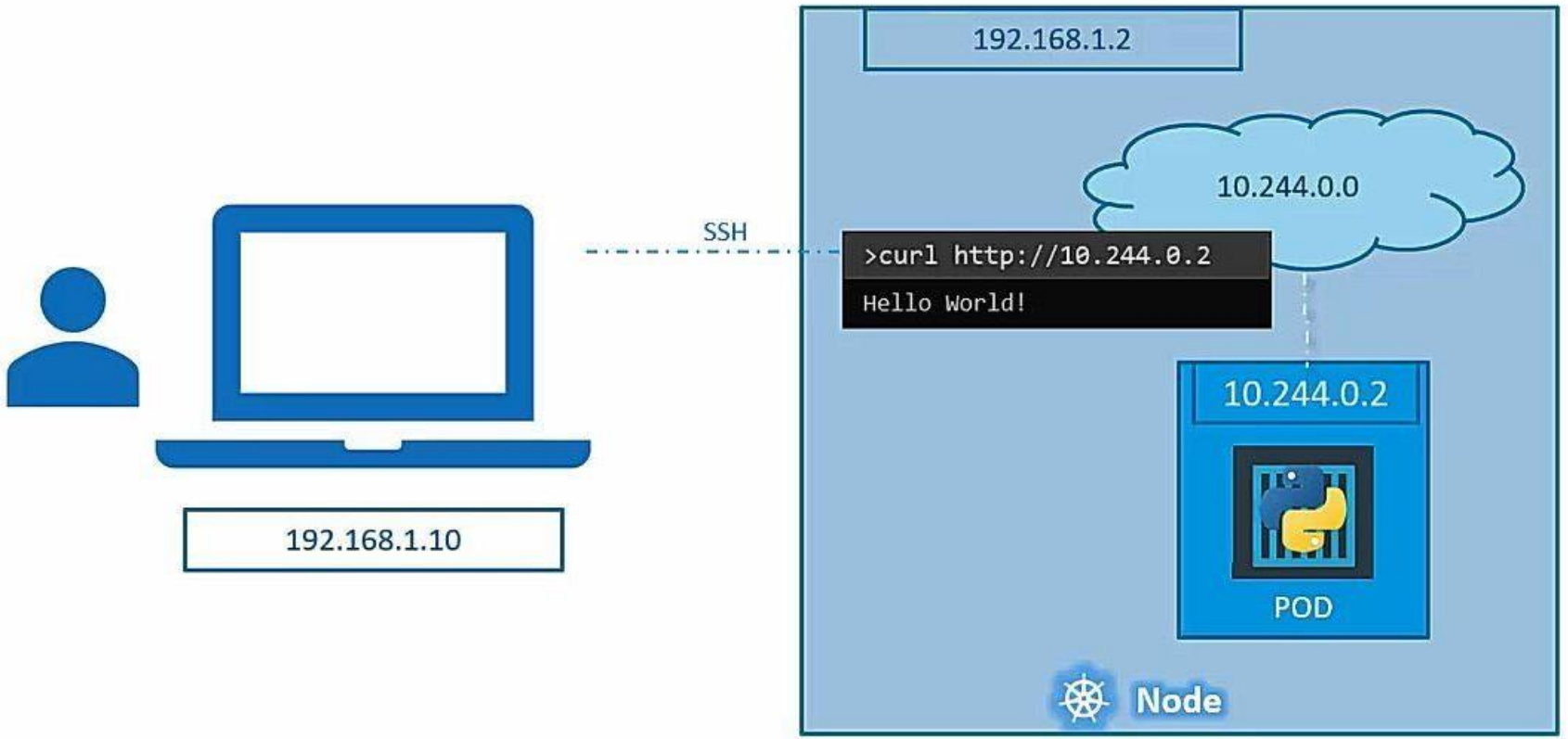
```yaml
apiVersion: v1
kind: Pod
metadata:
    name: my-pod
    labels:
        app: web
spec:
   containers:
   - name: web-container
     image: nginxdemos/hello:plain-text
     ports:
     - containerPort: 80
```

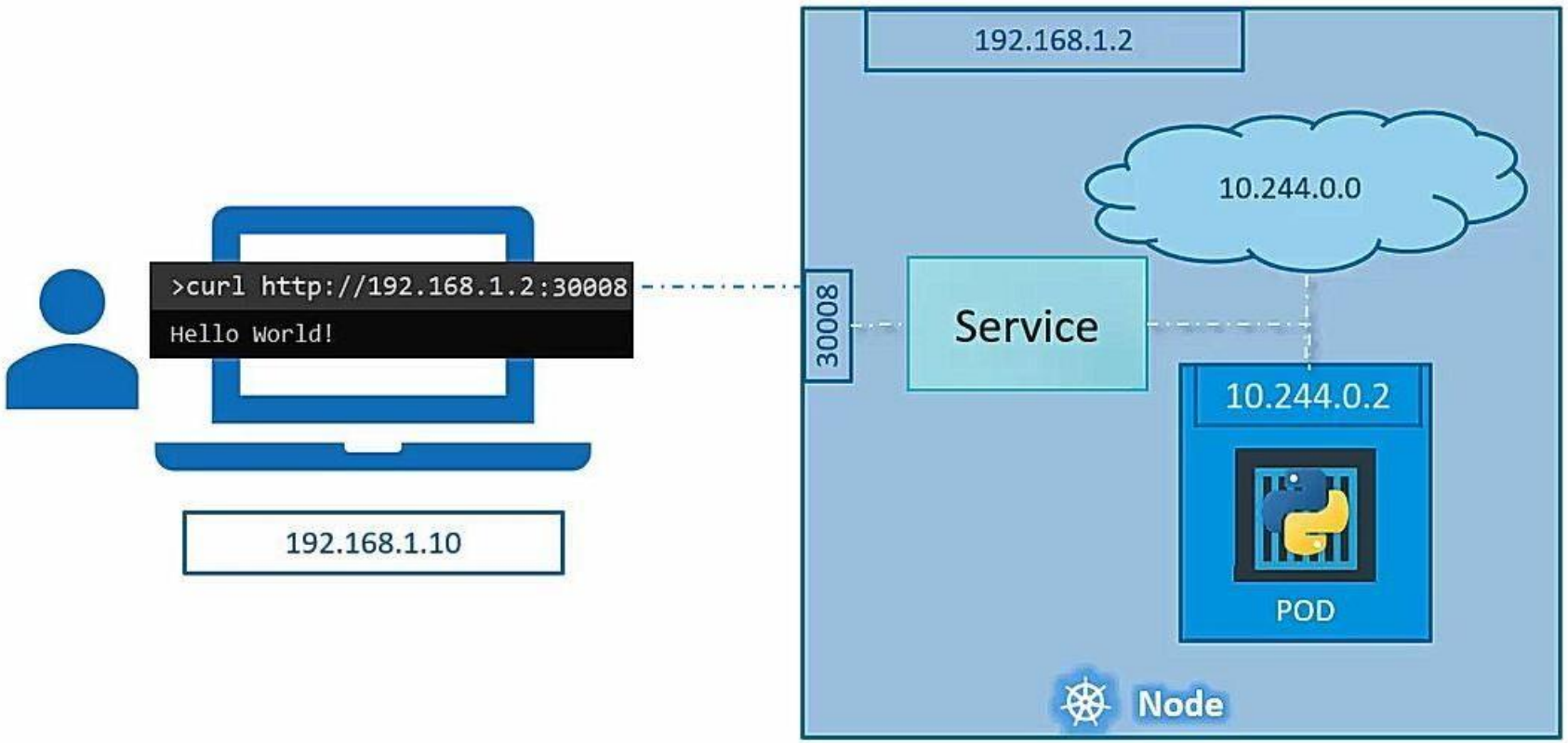
CLUSTER IP SERVICE

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-pod-service
spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
            app: web
```

NODE PORT

- NodePort: Exposes the Service on each Node’s IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You’ll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

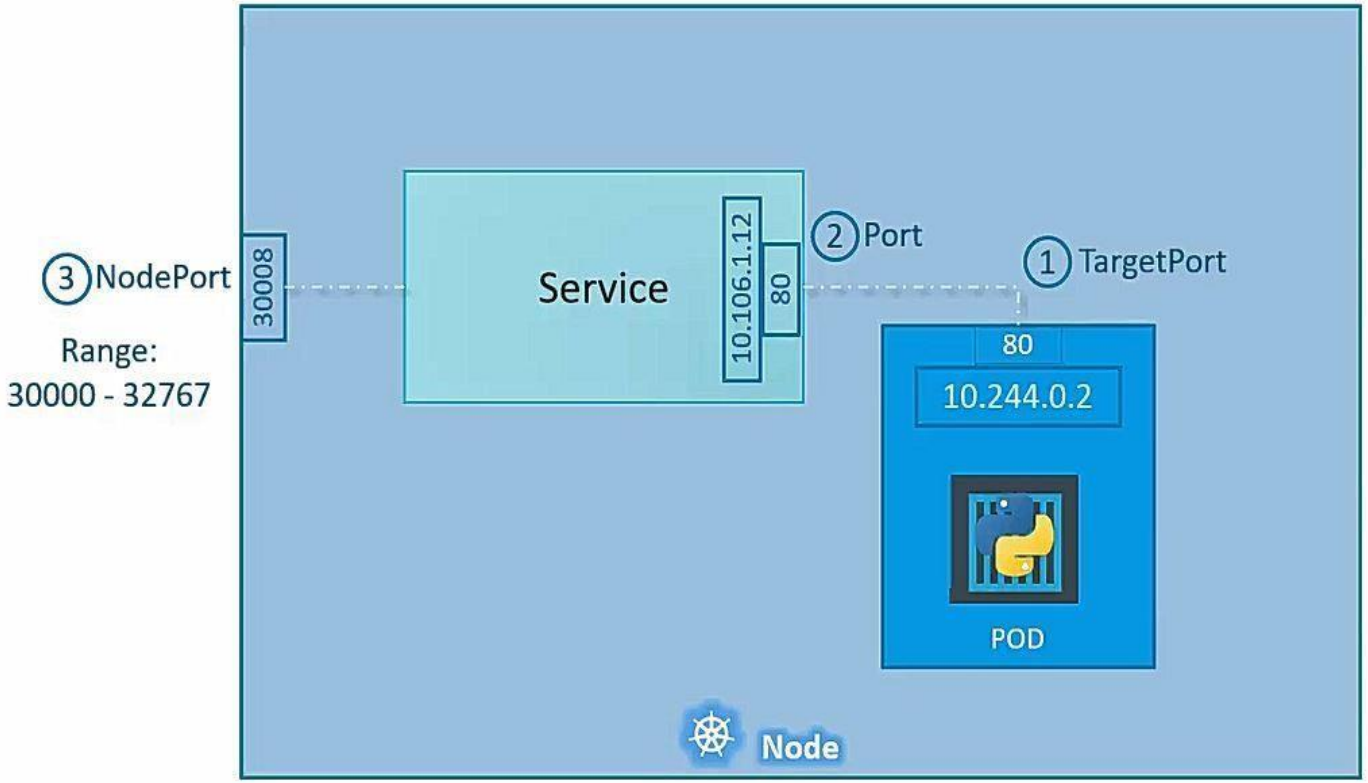




APP

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: my-pod
    labels:
        app: web
spec:
   containers:
   - name: web-container
     image: nginxdemos/hello:plain-text
     ports:
     - containerPort: 80
```

Service - NodePort



service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
      *port: 80
       nodePort: 30008
```

Service - NodePort

service-definition.yml

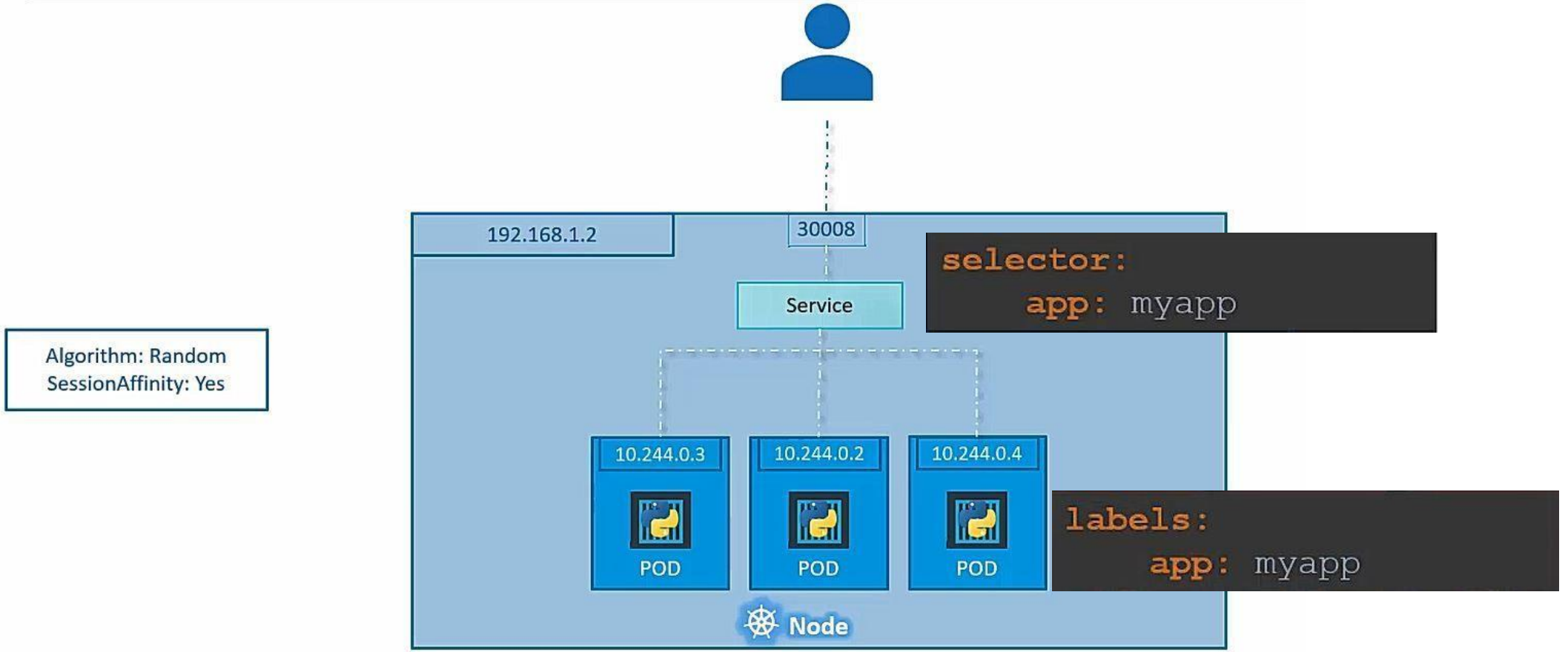
```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

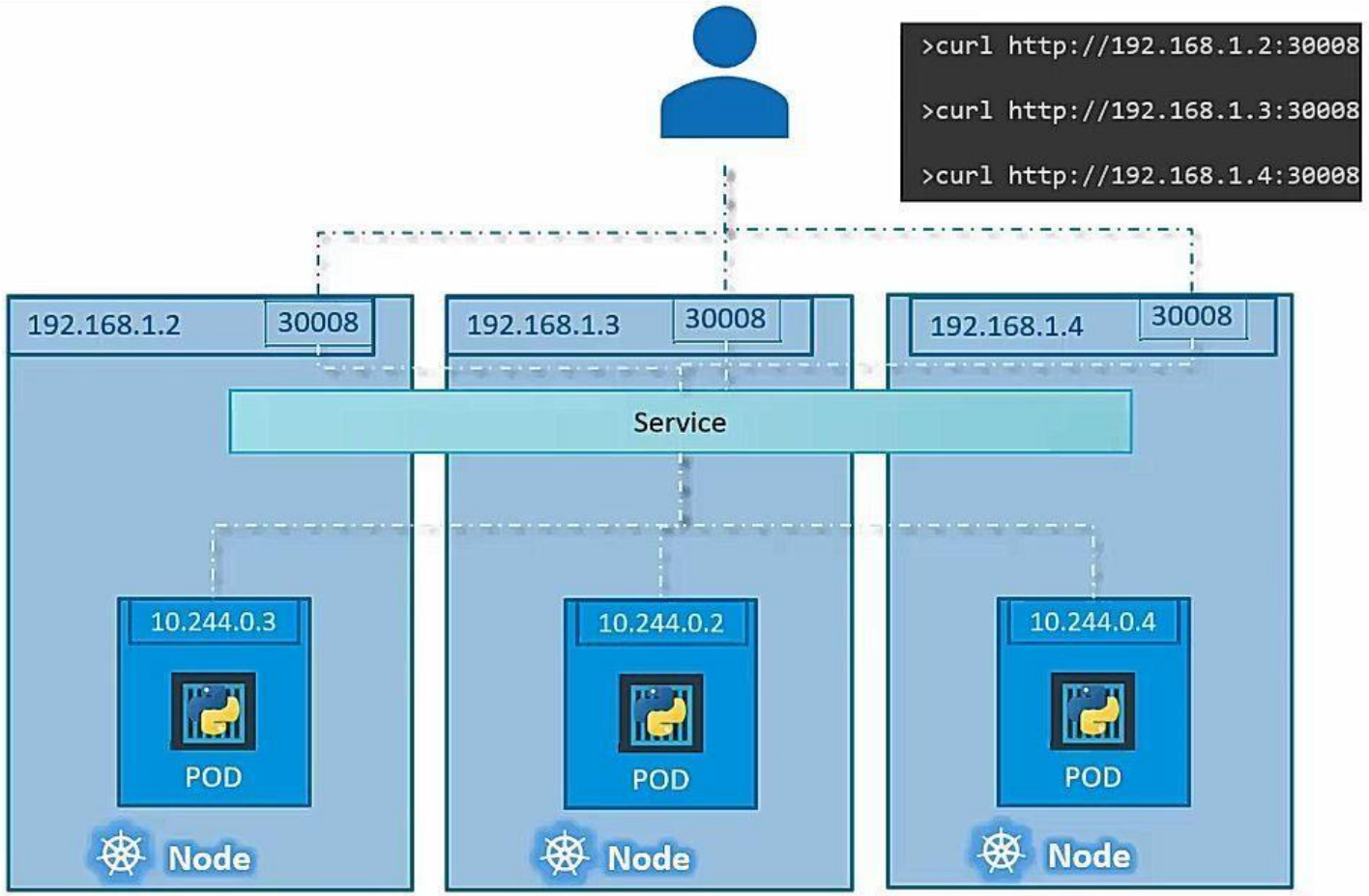
```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        16d
myapp-service   NodePort    10.106.127.123   <none>        80:30008/TCP   5m
```

```
> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

Service - NodePort



Service - NodePort



LOAD BALANCER

- LoadBalancer: Exposes the Service externally using a cloud provider’s load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

MANUAL SCHEDULING

SCHEDULER

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. The scheduler reaches this placement decision taking into account the scheduling principles described below.

SCHEDULING WITH KUBE SCHEDULER

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

- Filtering
- 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. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.

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.

MANUAL SCHEDULING

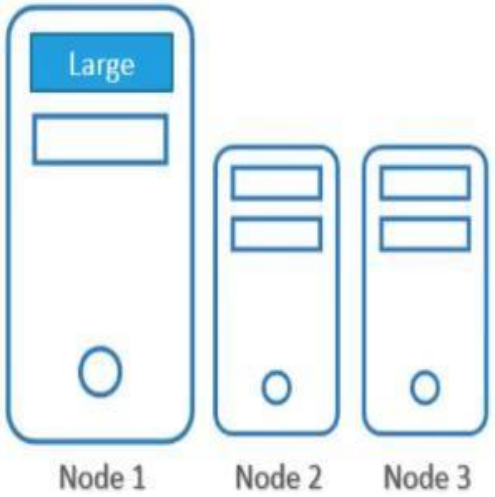
```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5    labels:
6      app: nginx
7  spec:
8    containers:
9    - name: nginx
10     image: nginx:1.7.9
11     ports:
12     - containerPort: 80
13   nodeName: node02
```

NODE SELECTOR (LABEL ON NODES)

Label Nodes

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

```
kubectl label nodes node-1 size=Large
```



NODE SELECTOR(LABEL ON PODS)

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: mypod
5  spec:
6    containers:
7    - name: data-processor
8      image: data-processor
9    nodeSelector:
10     size: Large
11
```

RESOURCE ALLOCATION ON CONTAINER

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
    - containerPort: 8080
    resources:
        requests:
            memory: "1Gi"
            cpu: 1
        limits:
            memory: "2Gi"
            cpu: 2
```

NOTE:- A container can not exceed 2 cpu limit if it tries to increase so it is known as throttle(limit) but it will continue working.

If container memory goes beyond 2Gi then it will terminated and create new if replica set is defined.

LIMIT CPU AND MEMORY FOR CONTAINER

```
1  apiVersion: v1
2  kind: LimitRange
3  metadata:
4      name: limit-mem-cpu-per-container
5  spec:
6      limits:
7      - max:
8          cpu: "800m"
9          memory: "1Gi"
10       min:
11         cpu: "100m"
12         memory: "99Mi"
13       default:
14         cpu: "700m"
15         memory: "900Mi"
16       defaultRequest:
17         cpu: "110m"
18         memory: "111Mi"
19       type: Container
```

#kubectl create -f <filename>
#kubectl describe limitrange/limit-mem-cpu-per-container

LIMITS MEMORY AND CPU FOR POD

```
1  apiVersion: v1
2  kind: LimitRange
3  metadata:
4      name: limit-mem-cpu-per-pod
5  spec:
6      limits:
7      - max:
8          cpu: "2"
9          memory: "2Gi"
10       type: Pod
11
12
13
```

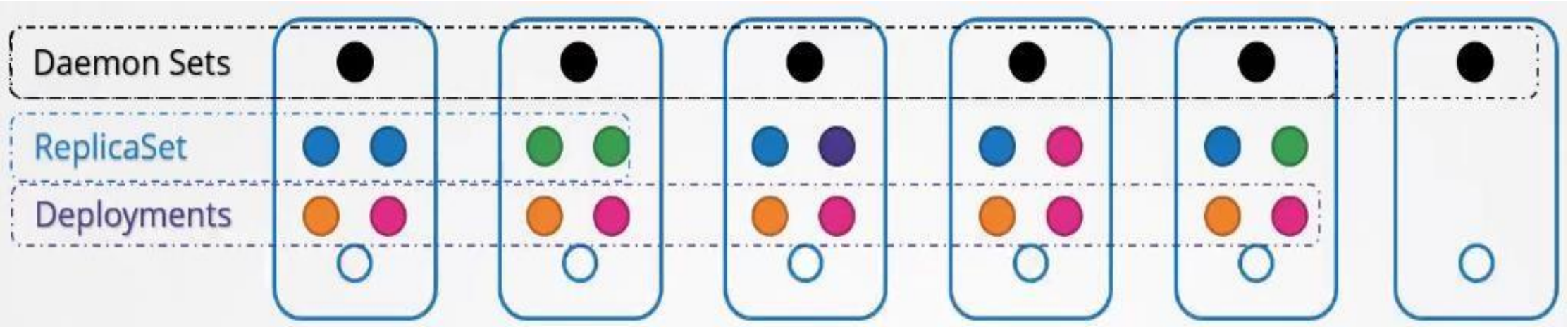
#kubectl create -f <filename>
#kubectl describe limitrange/limit-mem-cpu-per-pod

Daemon Set

- 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.



DAEMON SET

```
1  apiVersion: app/v1
2  kind: DaemonSet
3  metadata:
4    name: monitoring-daemon
5  spec:
6    selector:
7      matchLabels:
8          app: monitoring-agent
9    template:
10     metadata:
11         labels:
12             app: monitoring-agent
13     spec:
14         containers:
15         - name: monitoring
16           image: monitoring
17
```

Deployment

- A Deployment provides declarative updates for pods and replicasets.
- You describe a desired state in a Deployment, and the Deployment 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.

Uses of Deployment

- Create a Deployment to rollout a ReplicaSet
- Declare the new state of the Pods
- Rollback to an earlier Deployment revision
- Scale up the Deployment to facilitate more load
- Pause the Deployment
- Use the status of the Deployment
- Clean up older ReplicaSets

DEPLOYMENT STRATEGY

Two types of strategy->

1)Recreate -> in case of recreate lets say we have a pod and in pod there are 5 containers so it will first down all the containers then it will create new containers.

so in a production environment it's not a good strategy because of there is downtime.

2)Rolling Update -> in case of rolling update it will down one container then update one then down the second container then create second.

by default Kubernetes is using rolling update.

DEPLOYMENT

```yaml
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
```

ROLLOUT AND VERSIONING

- When we update any application so automatically it is called as revision so when we deploy any container so it takes as a revision 1 then when we update so revision2 and so on.

```
#kubectl rollout status deployment/myapp-deployment -> to check rollout status
#kubectl rollout history deployment/myapp-deployment -> to check history of rollout
```

DEPLOYMENT

To update manually->

#kubectl set image deployment/myapp-deployment nginx=nginx:1.7.1

To check deployment status and we can see strategy type->

#kubectl describe deployment myapp-deployment

•When you are doing upgrade so automatically it create a new replicasets.

#kubectl get rs

Now let's say we have some issue with update so we can rollout to previous version.->

#kubectl rollout undo deployment/myapp-dep

#kubectl get rs

STORAGE

Storage

Volumes

- Exposed at Pod level and backend different ways
- emptyDir: ephermal scratch directory that lives for the life of pod

Persistent Volumes

- Abstraction away from the storage provider(AWS, GCE)
- Have a lifecycle independent of any individual pod that uses it

Persistent Volume Claims

- Request for storage with specific details(size,access modes, etc.)

TYPES OF KUBERNETES VOLUME

Persistent Volumes

- By default, containers write to ephermal storage
- As a result, when a pod is terminated, all data written by it’s container is lost
- We can attach Persistent Volumes to pods which persist any data written to them

emptyDir

An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node

Persistent Volume (emptyDir)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Persistent Volumes

- Persistent Volumes can be provisioned either statically or dynamically.
  - Static: Apre-provisioned pool of volumes such as ISCSI or Fiber Channel
  - Dynamic: Volumes are created on demand by calling a storage provider's API such as Amazon EBS
- Persistent Volumes have a lifecycle independent of the pods that use them

Persistent Volume Claims

- Created by users to request a persistent volume
- Users can request various properties such as capacity and access modes (e.g. can be mounted once read/write or many times read-only)

CREATING PERSISTENT VOLUME

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4   name: pv-vol1
5  spec:
6   accessModes:
7      - ReadWriteOnce
8  capacity:
9     storage: 1Gi
10  hostPath:
11    path: "/tmp/data01"
12
13
14 ##three types of permissions-> 1) ReadOnlyMany 2) ReadWriteOnce 3) ReadWriteMany
15
16 #kubectl create -f <filename>
17 #kubectl get pv
18 #kubectl describe pv <pvname>
```

CREATING PERSISTENT VOLUME CLAIM

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4   name: myclaim
5  spec:
6   accessModes:
7    - ReadWriteOnce
8   resources:
9     requests:
10      storage: 500Mi
```

#kubectl create -f <filename>
#kubectl describe pvc
#kubectl describe pv <pvname>

USING PV AND PVC IN POD

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4   name: mypod
5   labels:
6      name: frontendhttp
7  spec:
8   containers:
9   - name: myweb
10    image: nginx
11    ports:
12    - containerPort: 80
13      name: "httpserver"
14    volumeMounts:
15    - mountPath: "/var/www/html"
16      name: myvol
17  volumes:
18  - name: myvol
19    persistentVolumeClaim:
20      claimName: myclaim          ##give in pvc yml
```

SECURITY

Kubernetes Authentication

- Authentication machanism handled by kubeapi server when we use kubectl command.

Transport Layer Security (TLS)

TLS

- Kubernetes provides a certificates.k8s.io API, which lets you provision TLS certificates signed by a Certificate Authority (CA) that you control. These CA and certificates can be used by your workloads to establish trust.

CERTIFICATE AUTHORITY (CA)

Client Certificates for Clients

admin.crt admin.key

admin

scheduler.crt scheduler.key

KUBE-SCHEDULER

controller-manager.crt controller-manager.key

KUBE-CONTROLLER-MANAGER

kube-proxy.crt kube-proxy.key

KUBE-PROXY

apiserver-kubelet-client.crt apiserver-kubelet-client.key

KUBE-API SERVER

apiserver-etcd-client.crt apiserver-etcd-client.key

KUBE-API SERVER

kubelet-client.crt kubelet-client.key

KUBELET SERVER

Server Certificates for Servers

etcdserver.crt etcdserver.key

ETCD SERVER

apiserver.crt apiserver.key

KUBE-API SERVER

kubelet.crt kubelet.key

KUBELET SERVER

Authorization

Authorization

In Kubernetes, you must be authenticated (logged in) before your request can be authorized (granted permission to access).

Kubernetes authorizes API requests using the API server. It evaluates all of the request attributes against all policies and allows or denies the request. All parts of an API request must be allowed by some policy in order to proceed. This means that permissions are denied by default.

Authorization Modes

- Node Based
  - grants permissions to kubelets based on the pods they are scheduled to run

- Attribute Based Access Control (ABAC)
  - Manual Manage Permissions in API

- Role Based Access Control (RBAC)
  - Roles and Role Bindings

- Webhook
  - manage third party authentication

Kube API Server Yaml

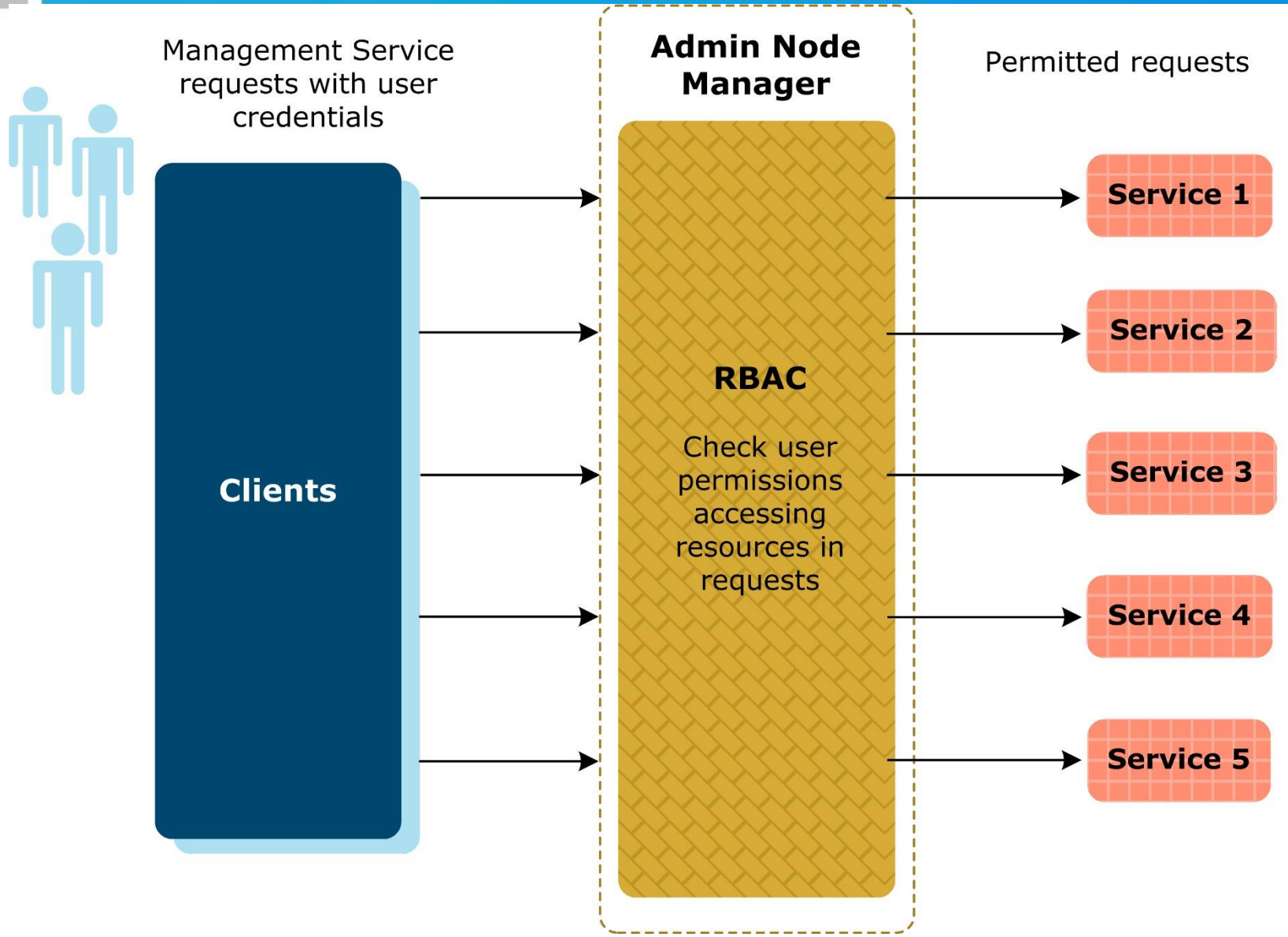
```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 172.25.230.143:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=172.25.230.143
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
```

Role-Based Access Control (RBAC) Authorization

What is RBAC?

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

RBAC



Managing Roles with Imperative Commands

#kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods

#kubectl create role pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod

#kubectl create role foo --verb=get,list,watch --resource=replicasets.apps

Verify

#kubectl get role

#kubectl describe role <rolenam>

Managing RoleBindings with Imperative

#kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob --namespace=acme

#kubectl create rolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp --namespace=acme

Verify

#kubectl get rolebinding

#kubectl describe rolebinding <name>

Check Access

#kubectl auth can-i <verb> <resource>

For example:

#kubectl auth can-i create pods

#kubectl auth can-i create pods --as gaurav

Note

Roles and Rolebindings are restricted to namespace.

If we want to give access cluster wide then we will use clusterrole and clusterrolebinding.

To check role access->

#kubectl api-resources --namespaced=true

To check clusterrole access->

#kubectl api-resources --namespaced=false

Managing Cluster Role with Imperative

#kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods

#kubectl create clusterrole pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod

Verify

#kubectl get clusterrole

#kubectl describe clusterrolebinding <name>

Managing ClusterRoleBinding with Imperative

#kubectl create clusterrolebinding root-cluster-admin-binding --clusterrole=cluster-admin --user=root

#kubectl create clusterrolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp

SECURITY CONTEXT

Security context defines privilege and access control settings for a Pod or Container.

SECURITY CONTEXT

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: web-pod
5  spec:
6      securityContext:
7          runAsUser: 1000
8      containers:
9      - name: ubuntu
10       image: ubuntu
11       command: ["sleep","3600"]
12
```

KOENIG
step forward

SECURITY CONTEXT

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Learning Solution

KOENIG
step forward

SECURITY CONTEXT

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

IT Learning Solution

Environment Variables

TYPES OF ENVIRONMENT VARIABLES

- Plain key value
- ConfigMap
- Secrets

PLAIN KEY VALUE

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: simple-webapp-color
5  spec:
6      containers:
7      - name: simple-webapp-color
8        image: simple-webapp-color
9        ports:
10         - containerPort: 8080
11       env:
12         - name: APP_COLOR
13           value: Pink
14
```

CONFIG MAP

A ConfigMap is a dictionary of configuration settings. This dictionary consists of key-value pairs of strings. Kubernetes provides these values to your containers. Like with other dictionaries (maps, hashes, ...) the key lets you get and set the configuration value.

CONFIG MAP

STEP 1: CREATE CONFIG-MAP

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: app-config
5  data:
6      APP_COLOR: blue
7      APP_MODE: prod
8
9
```

CONFIG MAP

STEP 2: inject configmap in pod file

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp-pod
5    labels:
6      app: myapp
7  spec:
8      containers:
9      - name: nginx-container
10       image: nginx:1.7.1
11       ports:
12         - containerPort: 8080
13       envFrom:
14         - configMapRef:
15             name: app-config
16
```

SECRETS

Secrets can be defined as Kubernetes objects used to store sensitive data such as user name and passwords with encryption.

There are multiple ways of creating secrets in Kubernetes.
Creating from txt files.
Creating from yaml file.

SECRETS

```
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: app-secret
5  data:
6      DB_Host: mysql
7      DB_User: root
8      DB_Password: password
9
10
11
```

in the above data we are specifying in a plain text so it's not a good idea so we can type it in a hash format. So we can convert in base64 format and type output in file.

```
#echo -n "mysql" | base64
bxiZcWw=      --> output
```

SECRETS

```
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: app-secret
5  data:
6      DB_Host: bxiZcWw=
7      DB_User: cb1Wx24x
8      DB_Password: ghTs7xD2
9
10
11
```

#kubectl get secrets
#kubectl describe secrets
#kubectl get secret app-secret -o yaml -> to show all information and encrypted password
#echo -n 'bxiZcWw= ' | base64 --decode -> to decode

SECRETS

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp-pod
5    labels:
6      app: myapp
7  spec:
8      containers:
9      - name: nginx-container
10       image: nginx:1.7.1
11       ports:
12         - containerPort: 8080
13       envFrom:
14         - secretRef:
15             name: app-secret
16
```

Logging and Monitoring

What are the kinds of logs we want to capture?

- Container Logs
- Host OS Logs
- Control Plane Logs
- Event Messages

Logging in Kubernetes and Containers

Helpful Command:

#kubectl logs <podname> -c <containername>

Monitoring

- Free
  - Kube-state-metrics- collects metrics from Kubernetes API
  - Prometheus
  - metricbeat – standard metric monitoring that is Kubernetes aware
  - Ganglia
  - And more……
- Third Party Services – Paid
  - Datadog
  - Honeycomb
  - Stackdriver with google monitoring
  - And more…

Setup Prometheus Monitoring Tool

Prometheus

Monitoring

Configuration

#git clone https://github.com/gauravkumar9130/kube-yaml.git

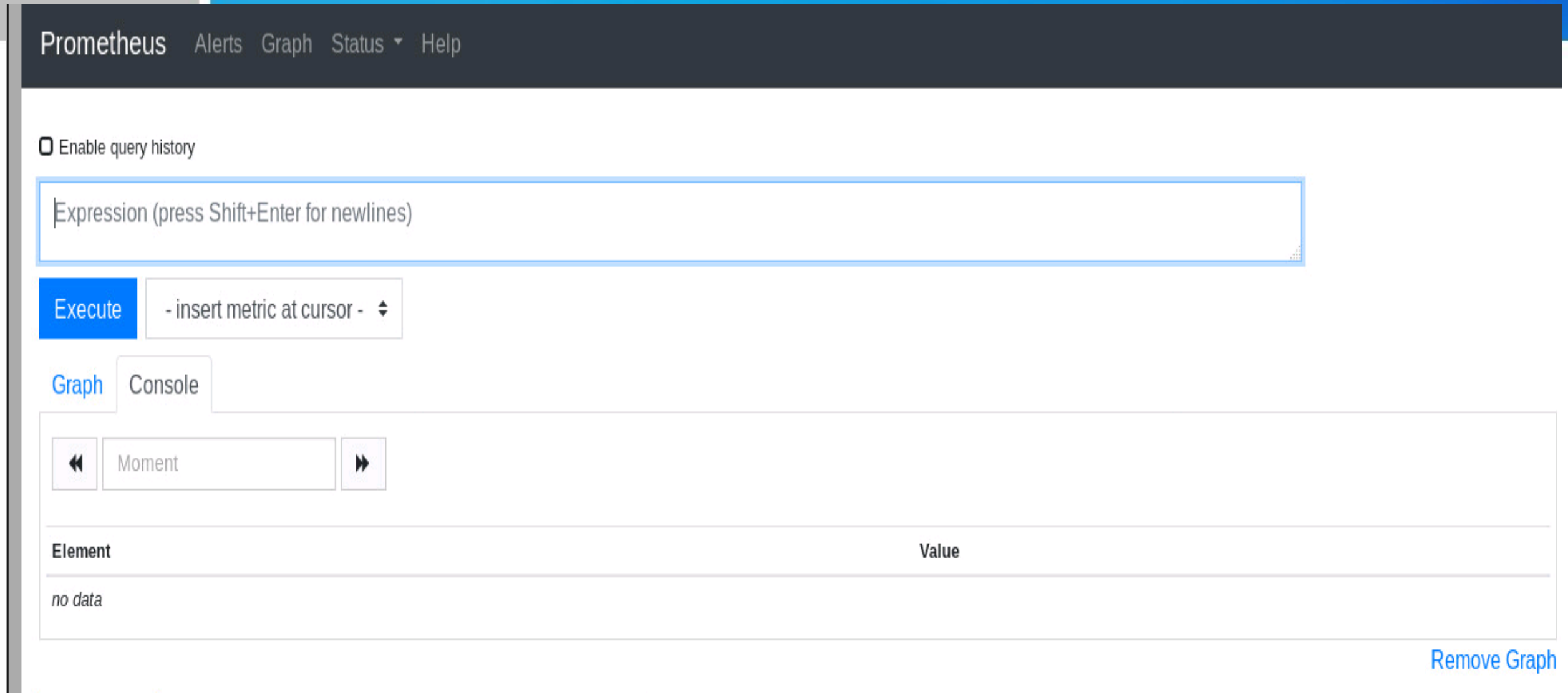
#cd kube-yaml

#kubectl create -f prometheus.yaml

#kubectl create -f kube-state-metrics-configs/.

#kubectl get pods -n monitoring

#kubectl get svc –n monitoring



NETWORKING

DNS

- Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

- Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain.

CUSTOM DNS FOR CONTAINER

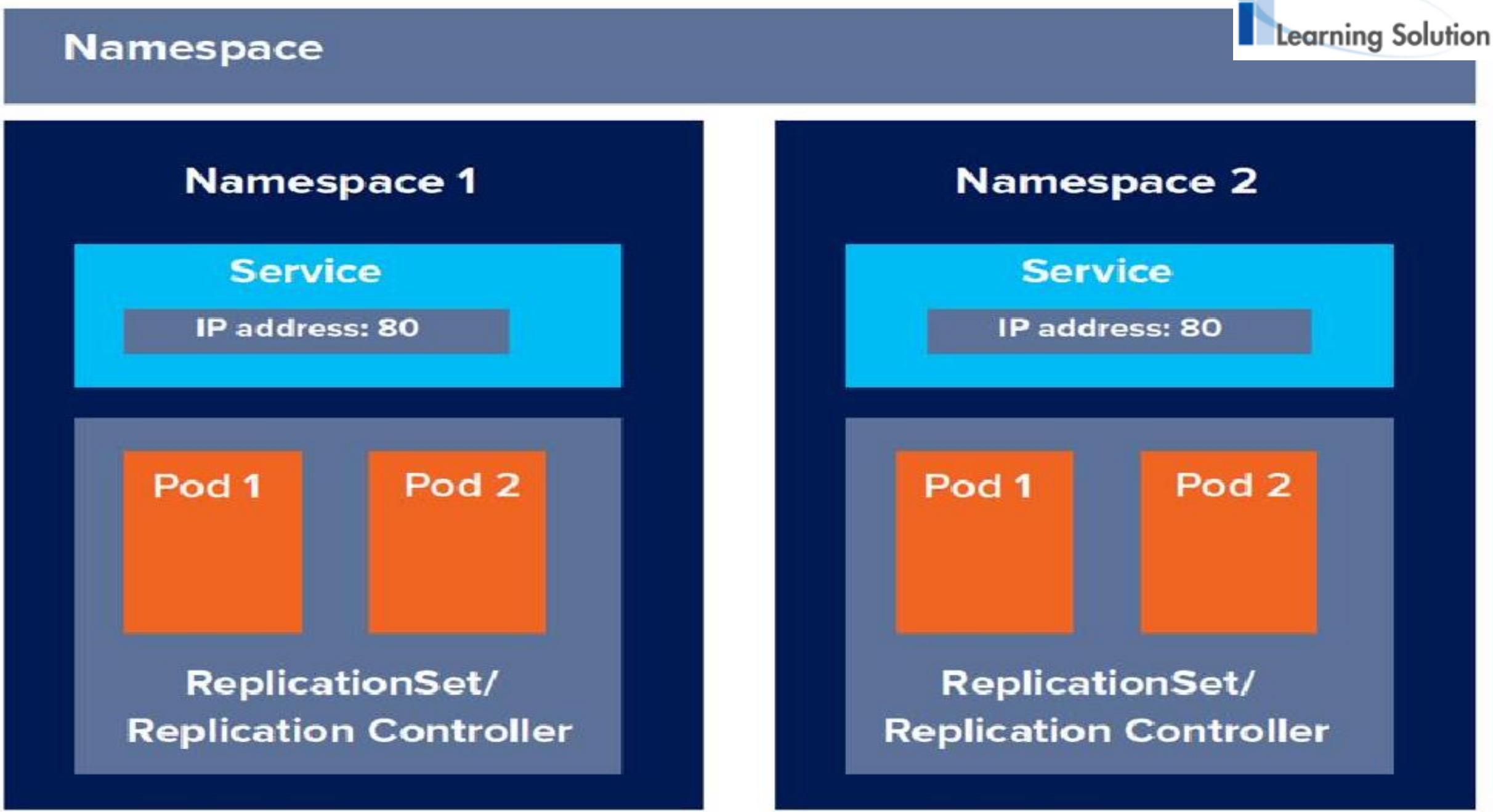
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
    args:
    - "/etc/hosts"
```

NAMESPACE

- Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.
- It offers an isolation for process interaction within OS.
- It limits the visibility that a process has on other process, network filesystem and userID component.
- Process from the containers or the host processes are not directly accessible from within this container process.

FUNCTIONALITY OF NAMESPACE

- Namespaces help pod-to-pod communication using the same namespace.
- Namespaces are virtual clusters that can sit on top of the same physical cluster.
- They provide logical separation between the teams and their environments.


Namespace
Namespace 1
Service
IP address: 80
Pod 1
Pod 2
ReplicationSet/
Replication Controller
Namespace 2
Service
IP address: 80
Pod 1
Pod 2
ReplicationSet/
Replication Controller

CREATE NAMESPACE

#vim name.yml

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: dev
5
```

```
[root@kmaster ~]# kubectl create -f name.yml
namespace/myname created
```

#kubectl create namespace dev

TO ADD PODS IN NAMESPACE

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: myapp-pod
    namespace: dev
    labels:
        app: myapp
        type: front-end
spec:
    containers:
    - name: nginx-container
      image: nginx
```

#kubectl get pods --namespace=dev

CONTROL THE NAMESPACES

kubectl create -f namespace.yml

kubectl get namespace

kubectl get namespace <Namespace name>

kubectl describe namespace <Namespace name>

#kubectl delete namespace <Namespace name>

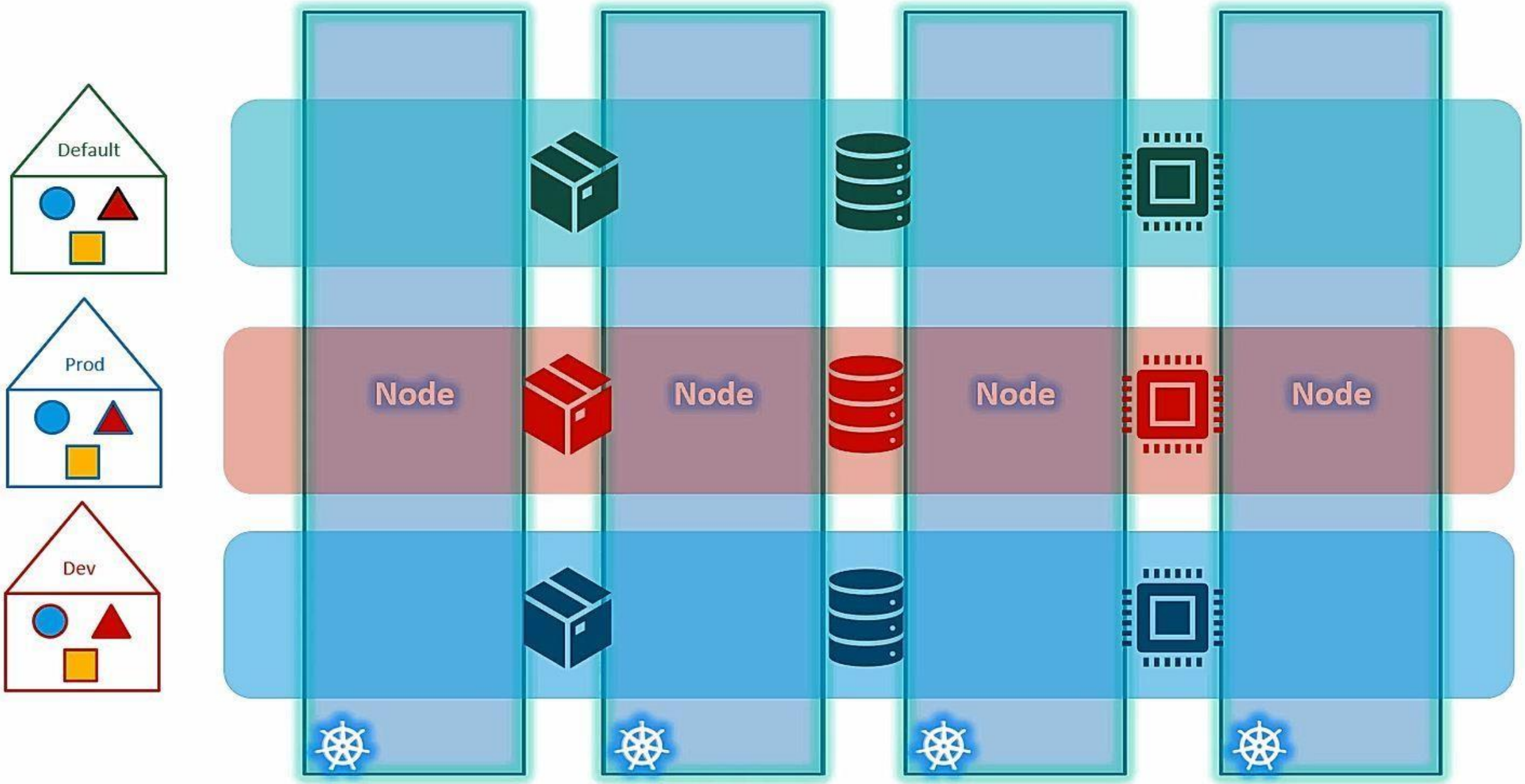
To switch from default namespace to another namespace:

#kubectl config set-context $(kubectl config current-context) --namespace=dev

To view pods from all namespaces

#kubectl get pods --all-namespaces

Namespace – Resource Limits



RESOURCE QUOTA IN NAMESPACE

```
1  apiVersion: v1
2  kind: ResourceQuota
3  metadata:
4      name: compute-quota
5      namespace: dev
6  spec:
7      hard:
8          pods: "10"
9          requests.cpu: "4"
10         requests.memory: 5Gi
11         limits.cpu: "10"
12         limits.memory: 10Gi
13
14
15 #kubectl create -f compute-quota.yml
16
```

Create Pod In given Namespace

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
    - containerPort: 8080
    resources:
        requests:
            memory: "1Gi"
            cpu: 1
        limits:
            memory: "2Gi"
            cpu: 2
```

CONTAINER NETWORK INTERFACE

- Kubernetes does not provide any default network implementation, rather it only defines the model and leaves to other tools to implement it. We can use **Calico** Solution.

NETWORK SOLUTION

We have popular Network Solution->

- Calico (https://www.tigera.io/blog/kubernetes-networking-with-calico/)

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

CALICO

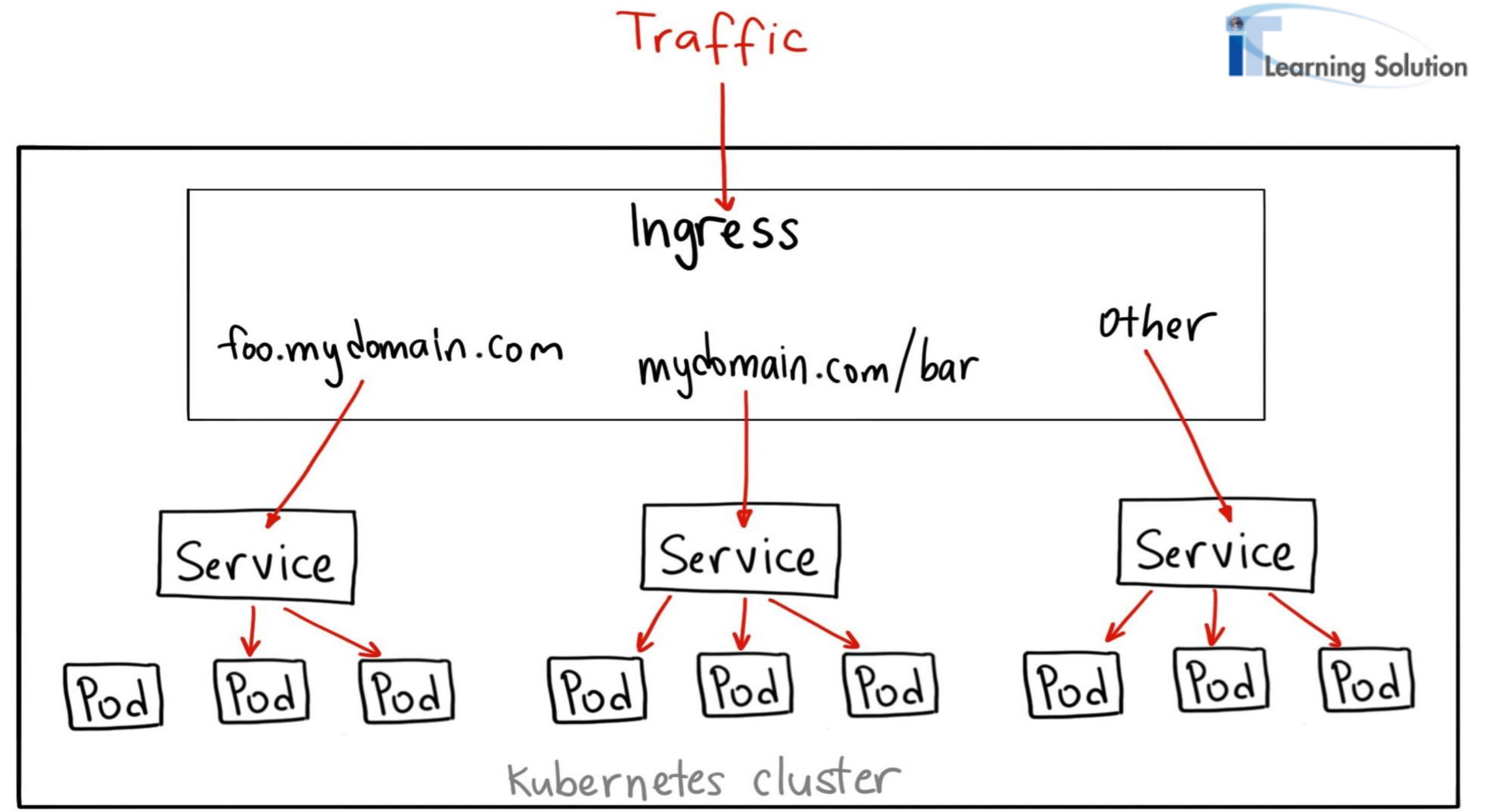
- Calico is a networking and network policy provider. Calico supports a flexible set of networking options so you can choose the most efficient option for your situation.

- By default, Calico uses any ipaddress other than you cluster ip, as the Pod network CIDR, though this can be configured in the calico.yaml file. For Calico to work correctly, you need to pass this same CIDR to the kubeadm init command using the **--pod-network-cidr** flag or via kubeadm's configuration.

- kubectl apply -f https://docs.projectcalico.org/v3.14/manifests/calico.yaml

INGRESS

- In Kubernetes, an Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster. You configure access by creating a collection of rules that define which inbound connections reach which services.


Traffic
Learning Solution
Ingress
foo.mydomain.com
mydomain.com/bar
Other
Service
Service
Service
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Kubernetes cluster

HOW TO DEPLOY INGRESS

- #git clone https://github.com/nginxinc/kubernetes-ingress.git
- #cd kubernetes-ingress
- #cd deployments

Create a namespace and service account->

- #kubectl apply -f common/ns-and-sa.yaml

Create a secret with a tls certificate and a key for the default server in nginx:-

- #kubectl apply -f common/default-server-secret.yaml

Set the behaviour of nginx-config (we can change)->

- #kubectl apply -f common/nginx-config.yaml

Configure RBAC->

- #kubectl apply -f rbac/rbac.yaml

HOW TO DEPLOY INGRESS

Deploy the ingress controller->

We have two options ->

- 1) deployment
- 2) daemon set -> we will use because it will be available in every container

#kubectl apply -f daemon-set/nginx-ingress.yaml

Verify

- #kubectl get namespace
- #kubectl get all -n nginx-ingress
- #kubectl get all -n nginx-ingress
- #kubectl get pods -n nginx-ingress -o wide

Host Based Ingress
(Hotel Application)

Application

- we are creating a website for :

  hotel.example.com

  With using hotel application and service

Application(HostBased)

#mkdir ingress-hostbased

Creating Namespace:

```
#vim create_namespace.yml
apiVersion: v1
kind: Namespace
metadata:
 name: hotel
```

Deploying Hotel Application

```
#vim app_deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hotel
 namespace: hotel
spec:
 replicas: 2
 selector:
  matchLabels:
   app: hotel
 template:
   metadata:
    labels:
      app: hotel
   spec:
     containers:
     - name: hotel
       image: nginxdemos/hello:plain-text
       ports:
       - containerPort: 80
```

Creating Service for Application

```
#vim service_for_hotel.yml
apiVersion: v1
kind: Service
metadata:
 name: hotel-svc
 namespace: hotel
spec:
 ports:
 - port: 80
   targetPort: 80
   protocol: TCP
   name: http
 selector:
   app: hotel
```

Create Ingress Rule(Host Based)

```
#vim ingress_rule.yml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: hotel-ingress
 namespace: hotel
spec:
 rules:
 - host: hotel.example.com
   http:
    paths:
    - path: /
      backend:
        serviceName: hotel-svc
        servicePort: 80
```

Verify

- #kubectl get all --namespace hotel
- #kubectl describe ingress hotel-ingress --namespace hotel

Add entry in host file because of we are not using any loadbalancer:

#vim /etc/hosts

<ipofworker> cafe.example.com

Path Based Ingress
(Cafe Application)

Application

- we are creating a website for : cafe.example.com

- which is having micro-services running

  café.example.com/tea -> tea-app

  café.example.com/coffee -> coffee-app

Create Namespace

```
#vim create_namespace.yml
apiVersion: v1
kind: Namespace
metadata:
 name: cafe
```

Create Coffee-App

```
#vim app_deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: coffee
 namespace: cafe
spec:
 replicas: 2
 selector:
  matchLabels:
   app: coffee
 template:
   metadata:
    labels:
      app: coffee
   spec:
     containers:
     - name: coffee
       image: nginxdemos/hello:plain-text
       ports:
       - containerPort: 80
```

Create Tea-App

```
#vim app_deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: tea
 namespace: cafe
spec:
 replicas: 2
 selector:
  matchLabels:
   app: tea
 template:
   metadata:
    labels:
     app: tea
   spec:
     containers:
     - name: tea
       image: nginxdemos/hello:plain-text
       ports:
       - containerPort: 80
```

Service for Coffee App

```
#vim service_for_coffee.yml
apiVersion: v1
kind: Service
metadata:
 name: coffee-svc
 namespace: cafe
spec:
 ports:
 - port: 80
   targetPort: 80
   protocol: TCP
   name: http
 selector:
   app: coffee
```

Service for Tea App

```
#vim service_for_tea.yml
apiVersion: v1
kind: Service
metadata:
 name: tea-svc
 namespace: cafe
spec:
 ports:
 - port: 80
   targetPort: 80
   protocol: TCP
   name: http
 selector:
   app: tea
```

Ingress Rule

```
#vim ingress_rule.yml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: cafe-ingress
 namespace: cafe
spec:
 rules:
 - host: cafe.example.com
   http:
    paths:
    - path: /tea
      backend:
        serviceName: tea-svc
        servicePort: 80
    - path: /coffee
        serviceName: coffee-svc
        servicePort: 80
```

Verify Application

#kubectl get ingress --namespace cafe

#kubectl describe ingress cafe-ingress --namespace cafe (show the Rules:)

#kubectl get pods -o wide -n café

Add entry in host file because of we are not using any loadbalancer

#vim /etc/hosts

- **ip of worker cafe.example.com**

Load Balancer

WHY?

- Kubernetes does not offer an implementation of network load-balancers (Services of type LoadBalancer) for bare metal clusters.

- LoadBalancers will remain in the “pending” state indefinitely when created.

METAL LOAD BALANCER

- MetalLB hooks into your Kubernetes cluster, and provides a network load-balancer implementation. In short, it allows you to create Kubernetes services of type “LoadBalancer” in clusters.

- It has two features that work together to provide this service: address allocation, and external announcement.

Prerequisites

#kubectl edit configmap -n kube-system kube-proxy

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "ipvs"
ipvs:
  strictARP: true
```

Installation By Manifests

```
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/namespace.yaml
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/metallb.yaml
# On first install only
kubectl create secret generic -n metallb-system memberlist --from-literal=secretkey="$(openssl rand -base64
128)"
```

https://metallb.universe.tf/installation/

Create ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.1.240-192.168.1.250
```

https://metallb.universe.tf/configuration/

LET'S DEPLOY A APPLICATAION

```
[root@master abc]# cat deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 4
 selector:
   matchLabels:
    app: nginx
 template:
   metadata:
    name: nginx-deployment
    labels:
      app: nginx
   spec:
    containers:
    - name: nginx-deployment
      image: nginx
      ports:
       - containerPort: 80
```

SERVICE FOR APPLICATION

```
[root@master abc]# cat service.yml
apiVersion: v1
kind: Service
metadata:
 name: nginx
spec:
 ports:
 - name: http
   port: 80
   protocol: TCP
   targetPort: 80
 selector:
  app: nginx
 type: LoadBalancer
[root@master abc]#
```

Verify

```
[root@master abc]# kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
kubernetes   ClusterIP      10.96.0.1       <none>           443/TCP        10h
nginx        LoadBalancer   10.98.253.120   172.25.230.240   80:31739/TCP   13m
[root@master abc]#
```

Deploying PHP Guestbook application with Redis

Application

- A single-instance Redis master to store guestbook entries
- Multiple replicated Redis instances to serve reads
- Multiple web frontend instances

Reference:

https://kubernetes.io/docs/tutorials/stateless-application/guestbook/

THANK YOU