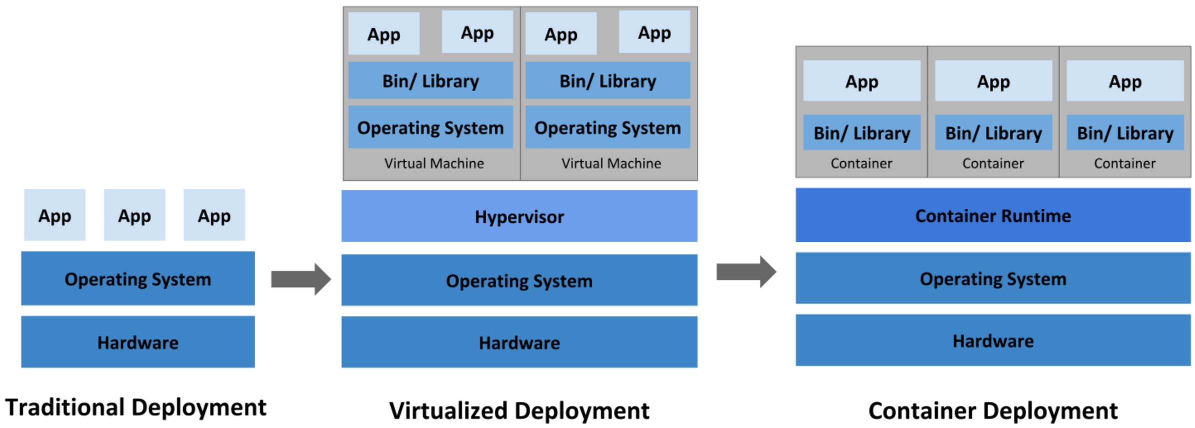
Containers recap

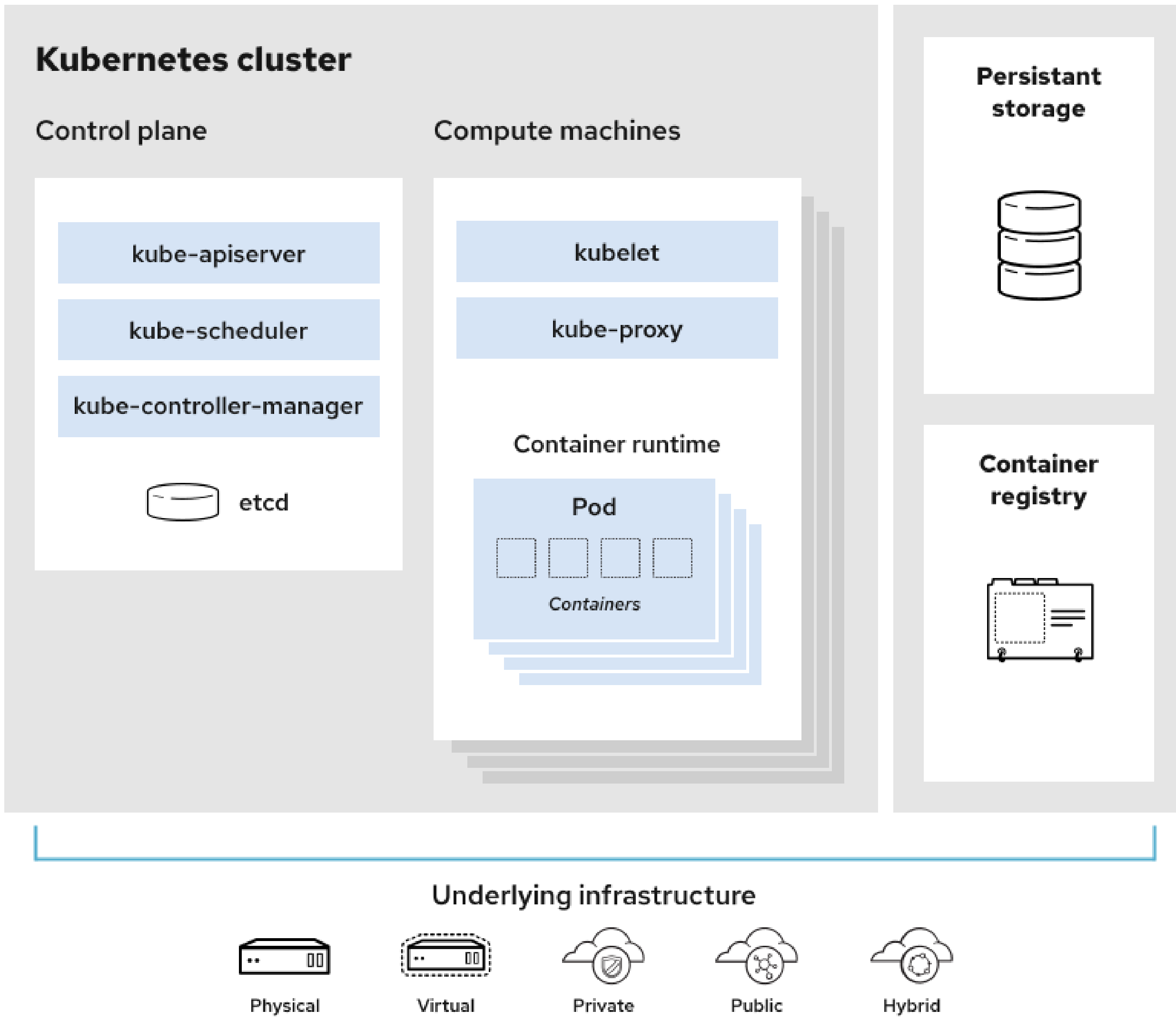


What is Kubernetes

Kubernetes (also known as k8s or “kube”) is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.
It’s based on Google’s internal project - Borg.

Kubernetes Architecture



- Control plane (master nodes)

Manages the workload

- Compute plane (worker nodes)

Runs the workload

Components of Kubernetes cluster

Master node - manages the cluster setup

- Etcd

Etcd is a key-value store database. Configuration data and information about the state of the cluster lives in etcd,

Fault-tolerant and distributed, etcd is designed to be the ultimate source of truth about your cluster.

- kube-scheduler

The scheduler considers the resource needs of a pod, such as CPU or memory, along with the health of the cluster. Then it schedules the pod to an appropriate compute node.

- Controller manager

Controllers take care of actually running the cluster, and the Kubernetes controller-manager contains several controller functions. For Example:

  - Node controller - Manages node related tasks like onboarding nodes, managing node failures, etc
  - Replication controller - Manages container. Make sure the desired number of containers run all the time.

- Kube-apiserver

Facilitates the interaction with the Kubernetes cluster.

The is the front end of the Kubernetes control plane, handling internal and external requests. The API server determines if a request is valid and, if it is, processes it. You can access the API through REST calls, through the kubectl command-line interface, or through other command-line tools such as kubeadm.

Worker node - Rus the container workload in the cluster setup

- Container runtime Engine

To run the containers, each compute node has a container runtime engine. Docker is one example, but Kubernetes supports other Open Container Initiative-compliant runtimes as well, For example:

  - Docker

    - Containerd
    - Rocket
    - CRI-O

- Kubelet

Each compute node contains a kubelet, a tiny application that communicates with the control plane. The kublet makes sure containers are running in a pod. When the control plane needs something to happen in a node, the kubelet executes the action.

- Kube-proxy

Each compute node also contains kube-proxy, a network proxy for facilitating Kubernetes networking services. The kube-proxy handles network communications inside or outside of your cluster—relying either on your operating system's packet filtering layer, or forwarding the traffic itself.

Pod

Pod is the atomic unit of scheduling in kubernetes cluster

Pod is a group of containers. Normally one container is run in one pod.
When two applications are tightly coupled, then only two containers may run in a single pod.
Usually it is avoided to run more than one container in one pod.

Pod is a wrapper to distribute the container.
Consider running multiple containers with port mapping from host:

docker run -d -p 80:80 httpd
docker run -d -p 80:80 httpd

The second run fails as port 80 on the host is already occupied with the earlier container. Hence another container needs to be run with different port mapping on host.

docker run -d -p 81:80 httpd

We need to manually track the free ports.

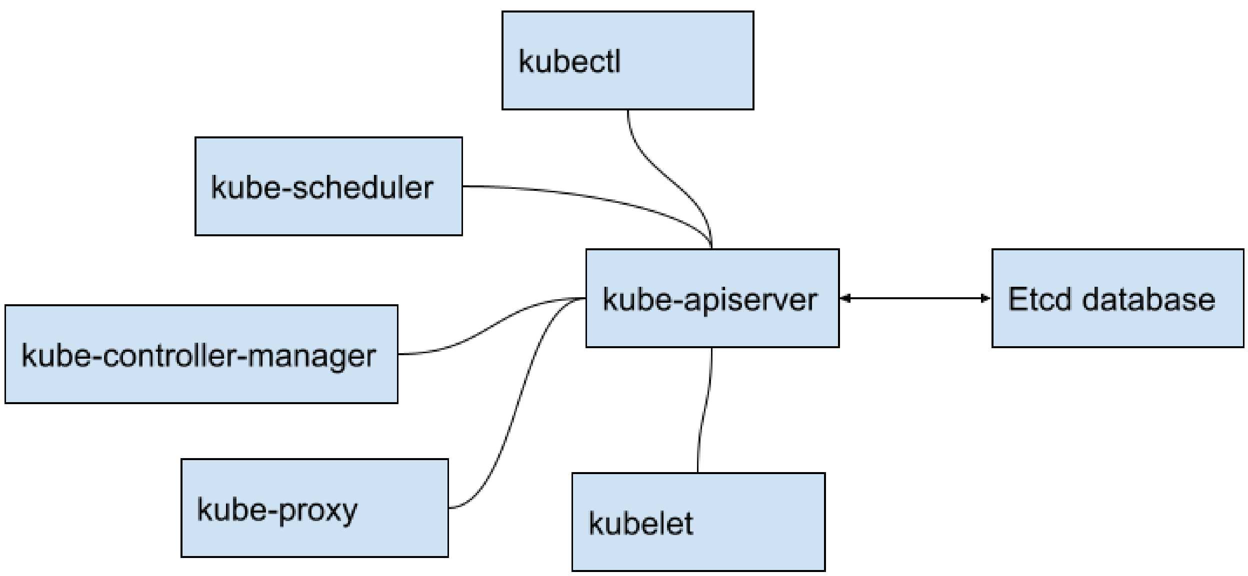
Container abstraction helps to handle this.

Pod gets it's own network namespace and virtual ethernet connection to connect underlying infrastructure. (We will look actual hands-on with this using kubernetes later)

Basic operations to run a workload in Kbernetes:
Package application as container
Wrap it in the pod
Deploy it using manifest file

Pods help for abstraction of container engine.
With use of pods as basic atomic unit, the underlying container engine can be easily changed.

Kube-apiserver



Consider any kuectl get command

1) **kubectl** generates authentication request to **kube-apiserver**
2) **Kube-apiserver** validates the request
3) **kube-apiserver** retrieve data from **etcd**
4) Data is displayed on the console by **kubectl**

APIs can also be invoked directly by sending HTTP request instead of running the kubectl command because api-server uses HTTP protocol for all communication

During the request for creating pod:

1) **kubectl** generates authentication request to **kube-apiserver**
2) **Kube-apiserver** validates the request
3) **kube-apiserver** creates a pod object without assigning it to a node
4) **kube-apiserver** updates the pod info in **etcd** database
5) **Kube-apiserver** updates the user that pod has been created
6) **kube-scheduler** continuously monitors the **kube-apiserver** and realizes there is new pod with no node assigned
7) **kube-scheduler** identifies the right node for the pod and communicated that back to **kube-apiserver**
8) **kube-apiserver** updates the info in **etcd**
9) **kube-apiserver** passes that info to the **kubelet** in the appropriate worker node
10) **kubelet** creates the pod in the node and instructs the container runtime engine to deploy the application image
11) once done **kubelet** updates the status back to the **kube-apiserver**
12) **kube-apiserver** updates the data back in the **etcd**

kube-apiserver at the center of all the tasks that need to be performed on the cluster.

KubeControllerManager

Manages various controllers in the setup.
Continuously monitors the status of different components on the cluster
In case of any failure, it takes corrective actions to bring the cluster back to its desired state.
For Example:
node-controller continuously monitors the status of the nodes via kube-apiserver
Heartbeat is sent every 5 seconds to monitor the health of the node
If the heartbeat is missed, the controller waits for 40 seconds before marking the node unreachable.
After the node is marked unreachable, the controller waits for another 5 mins to see if the node is coming back.
If a node does not come up after 5 mins, the controller manager moves the pods assigned to the broken node to the healthy nodes.
There are many controllers in a kubernetes cluster e.g. node-controller, replication controller, namespace controller, deployment controller, etc. All these controllers are packaged in to single kube-controller-manager process

Kube scheduler

Kube scheduler does not create the pod on the worker node.
Kube scheduler just decides which pod goes on which node. Kubelet creates the pod on the node.
Scheduler tries to identify the best node for the pod based on:
resource requirements
Taints
Affinity rules
Etc

Kubelet

Kubelet in the worker node registers the node with the cluster
When kube-apiserver sends request to load a container it sends request to container runtime engine e.g. docker to run the instance
Kubelet then regularly monitors that status of the pods and reports the status to kube-apiserver

Kubelet is the sole point of contact on the worker node for the master node.

Load or unload the pods, send status

Hands-on

kubectl get pods

kubectl get pods -A

kubectl run --image IMAGE_NAME POD_NAME

kubectl run --image nginx nginx

kubect create deployment NAME --image=IMAGE

kubectl create -f definition.yml

kubectl create -f definition.yml

kubectl delete pod NAME

kubectl deete deployment NAME

kubectl run NAME --image=NAME --dry-run=client -o yaml

kubectl edit pod NAME

kubectl apply -f file.yml

kubectl describe pod PODNAME

kubectl logs PODNAME

kubectl logs PODNAME CONTAINERNAME

pod.yml:
```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    app: myapp
    type: prod
spec:
  containers:
    - name: container-1
      image: redis
    - name: container-2
      image: nginx
```

Pod replication

Replication controller helps create multiple instances of a pod to provide high availability.
It also helps with scaling of application workload using load balancing.
What is Scale up (vertical scaling) VS scale out (horizontal scaling)?

Labels and Selectors are used to identify and use the pods in a specific replica set.
This can be used for pods outside of the replica set definition file as well.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-ha
  labels:
    app: myapp
    type: front-end
```

```
spec:
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
        type: frontend
    spec:
      containers:
        - name: container-1
          image: redis
  replicas: 3
  selector:
    matchLabels:
      type: frontend
```

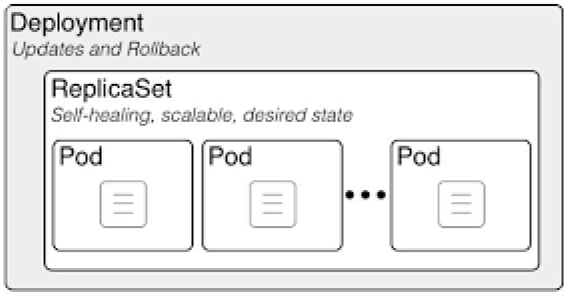
Kubectl create -f file.yml
Kubectl explain replicaset
Kubectl get replicaset
Kubectl get pods
Kubectl delete pod NAME
Kubectl get pods
Kubectl describe replicaset myapp-ha

Scale by changing the replicas number in aove config and then use below command
kubectl replace -f file.yml

Alternate options:

Kubectl scale –replicas=4 -f file.yml
Kubectl scal –replcas=5 replicaset myapp-ha

Deployments



Typical requirements form any production application:

- Replications - for high availability
- Seamless upgrade
- Rolling updates - updates one after other for seamless user experience
- Rollback updates - in case of failure, updates can be rolled back
- Pause - update resume capability

Same config as replicaset except
kind: Deployment

Kubectl explain deployment

Generate the sample config file using:
Kubectl create deployment –image=nginx nginx –dry-run=client -o 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:1.22
```

Kubectl apply -f file.yml
Kubectl get deployments
Kubectl get replicaset
Kubectl get pods
Kubectl get all
Kubectl describe deployment NAME
Kubectl describe pod name

Watch “kubectl get all”

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
Kubectl describe pod name

https://hub.docker.com/_/nginx/tags
Edit file and change to the newer version
Kubectl apply -f file.yml

Kubectl describe pod name
kubectl rollout status deployment nginx-deployment
kubectl rollout history deployment nginx-deployment
kubectl rollout undo deployment nginx-deployment --to-revision 1

Create custom image with required contents (optional)

1) Create a program (tobe used in custom image) which will consume system memory:

```
cat mem.c
/*
 * https://stackoverflow.com/a/1865536
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
	while(1)
	{
			printf ("\nAllocating...");
		void *m = malloc(1024*1024);
		memset(m,0,1024*1024);
			printf("done");
			printf ("\nPress any key to continue...");
```

```
            //sleep(2);
            getchar();
    }
    return 0;
}
```

```
$ gcc mem.c -o mem
```

2) Create a Dockerfile to generate a custom container with required packages and tools.

```
cat Dockerfile
from fedora
run dnf -y install iputils iproute
copy mem /usr/local/bin/mem
CMD sleep 100000
```

3) Create docker image and test is by running docker container

```
docker build -t my-fedora .

docker run --rm -ti --name new-fedora my-fedora bash
```

4) Login to https://hub.docker.com/ with your credentials and create a public repo with tag

5) Back to the system: tag newly created custom image and push it to docker hub:

```
$ docker login
$ docker image tag my-fedora:latest ashishkshah/my-fedora:test
$ docker push ashishkshah/my-fedora:test
```

6) Create kubernetes pod using this new image:
```
$ kubectl run --image ashishkshah/my-fedora:test mypod
```

7) Connect to the pod with bash shell and verify the tools embedded in it:

```
kubectl exec --stdin --tty mypod -- /bin/bash
$ mem
$ ping
$ ip
```

Namespace

```
kubectl create namespace myspace
```

```
kubectl get pods --namespace=myspace
kubectl create -f file.yml --namespace=myspace
kubectl get ns
kubectl get pods --all-namespaces
kubectl config current-context
kubectl config set-context kubernetes-admin@kubernetes --namespace=myspace
```

```
cat namespace.yml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: myspace
spec:
status: {}
```

Resource limit on Pod

1) Run the pod with custom image

```
cat pod-limits.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  containers:
  - name: app
    image: ashishkshah/myrepo:test
    resources:
      requests:
        memory: "8Mi"
      limits:
        memory: "16Mi"
```

2) Connect to the pod via bash shell (two sessions) and execute memory consumption utility /usr/local/bin/mem
3) Keep increasing the program's memory consumption and observe it in another session.
   $ watch “ps aux | grep mem | grep -v mem”

4) Watch the program getting terminated upon hitting limit

A request is the amount of that resource that the system will guarantee for the container, and Kubernetes will use this value to decide on which node to place the pod.
A limit is the maximum amount of resources that Kubernetes will allow the container to use.

Resource limit on namespace

```
cat resource-quota.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
spec:
  hard:
    cpu: "2"
    memory: "32Mi"
    pods: "4"
    replicationcontrollers: "2"
    resourcequotas: "1"
    services: "3"

kubectl describe quota

kubectl --namespace=myspace describe quota

# kubectl apply -f resource-quota.yml --namespace=myspace

kubectl --namespace=myspace describe quota

kubectl config set-context minikube --namespace=myspace
kubectl config get-contexts
kubectl describe quota

cat pod-limits-nginx.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  containers:
  - name: app
    image: nginx
```

```yaml
    resources:
      requests:
        memory: "8Mi"
        cpu: 2
      limits:
        memory: "16Mi"
        cpu: 2
```

Services in Kubernetes

In kubernetes, services are different mechanisms available for accessing pods.
By Default, pods created on kubernetes cluster are not accessible outside of the cluster and they can not communicate among themselves as well.

```
kubectl run --image nginx nginx --port 80
kubectl describe pod nginx | grep 'Node\|IP'
```

```
curl IP
curl IP <-- from node
```

```
kubectl port-forward pod-nginx  30005:80 --address 0.0.0.0
```

NOTE: port-forwarding is used throughout this hands-on activity for the demonstration purpose only. This is not a recommended way to expose any service from kubernetes cluster.

There are different types of services available in kubernetes cluster which enables the communication among the pods and communication outside the cluster.

1) ClusterIP
2) NodePort
3) LoadBalancer

ClusterIP

ClusterIP is the default service type.
This is used for inter service communication within the cluster.
Consider two deployment sets, front-end and back-end.
Communication between these two deployment sets is established using clusterip service.

Why do we need it? or why do we prefer this for inter pod communication within the cluster?

If we are using a pod's ip address for communication among the pods, when any pod is deleted or crashed due to some reason, the deployment set or replica set will spawn another pod to maintain the replica count.
The new pod spawned will have different ip addresses and hence it will not be easy to maintain the communication among the pods with new pods running with different ips.

To resolve this, we are creating a clusterip service which attaches itself to pods or replicas or daemonsets using labels and selectors.
In our example we will have two clusterip services for frontend and backend each.
The IP address of clusterip service is now being used for communication among the services instead of pods ip address.
Now even if the pods in replicasets fail and new pods are spawned with a new ip address, communication between the two services is not broken as it is done using clusterip service.

Kubernetes assigns a cluster-internal IP address to ClusterIP service.
Due to this the service is only reachable within the cluster.
The IP address of ClusterIP service can not be accessed from outside the cluster.

clusterip service enable connictivity between group of pods

kubectl create -f pod-client.yml

Try to access pod ip using curl

kubectl get all -o wide

Note the node where pod is runnning
From the node try to access service using curl

```
cat deployment-nginx.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-server-dep
  labels:
    app: nginx-server
spec:
  template:
    metadata:
      labels:
        app: nginx-server
```

```
    spec:
      containers:
        - name: frontend
          image: nginx
  replicas: 3
  selector:
    matchLabels:
      app: nginx-server
```

```
cat cip-service-nginx-dep.yml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cip-nginx-server-dep
  name: cip-nginx-server-dep
spec:
  ports:
  - name: "80"
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx-server
  type: ClusterIP
```

```
kubectl apply -f deployment-nginx.yml
kubectl apply -f cip-service-nginx-dep.yml
kubectl port-forward services/cip-nginx-server-dep  30005:80 --address 0.0.0.0

curl kubemaster:30005
```

Kill the pods one by one and make sure service is accessible via service ip

NodePort

NodePort service is an extension of ClusterIP service.
It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.

Limitation of clusterip service was it can not be used for communication outside the cluster.
Noteport service can be used for enabling access to the service outside of the cluster.

As the name may suggest, the NodePort service exposes the service(port) on each Node's IP.
The service can be accessed from outside the the cluster using nodeip:port
Port configured to listen on the node is mapped to the service port and it is further mapped to the port on the pod.

Node port must be in the range of 30000–32767. The ports can be allocated manually or kubernetes will take care of it if they are not manually assigned.

```
 cat np-service-nginx-server.yml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: np-service-server
  name: np-service-server
spec:
  ports:
  - name: "80"
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30009
  selector:
    app: nginx-server
  type: NodePort
```

```
kubectl apply -f np-service-nginx-server.yml
kubectl describe service np-service-server
```

Check the node where pod is provisioned

run curl kubeworker1:30005 from outside

LoadBalancer

The nodeport service has enabled external connectivity for the service. But still there is one problem with it.
The service is listening on the specified port on node's ip. That means if the pods in your replicaset are distributed among different nodes, all the nodes ip addresses will listen on the nodeport. Hence there will be a number of IP:PORT combinations available to access your service.

Needless to say there is no server side distribution of the traffic with this approach of accessing service. This approach is not practically used
for providing external access to the service but is used as an intermediate step.

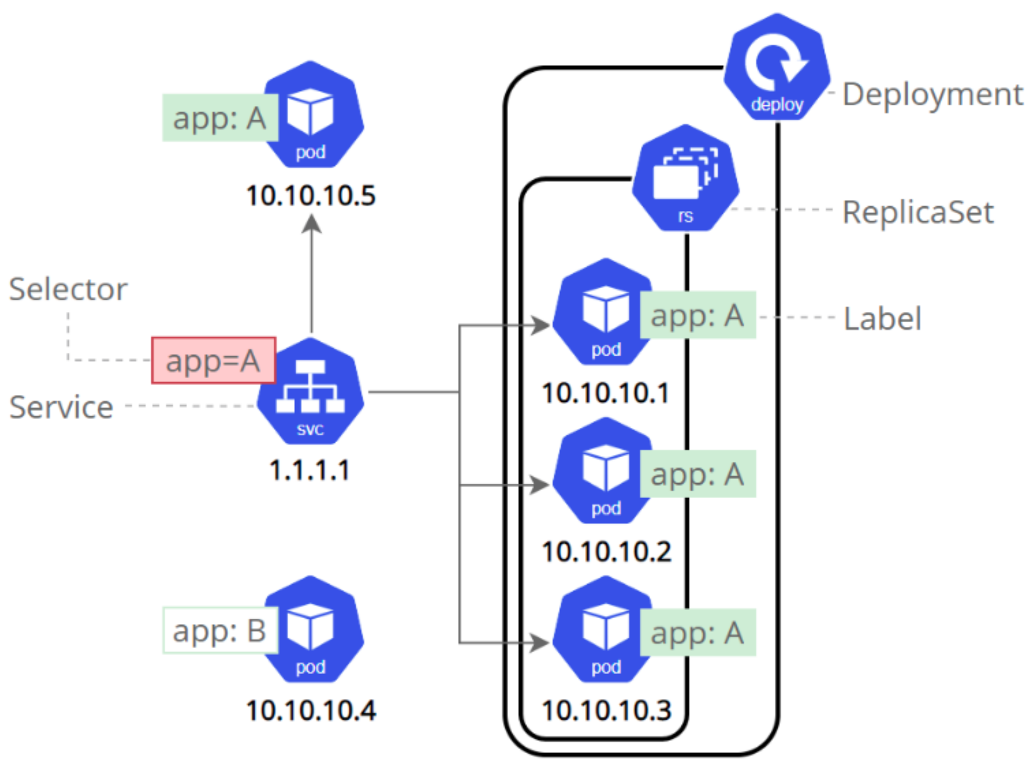
LoadBalancer service is an extension of NodePort service.
Loadbalancer integrates NodePort with cloud-based load balancers and provides a single ip:port combination for accessing service from external networks.
The load is then distributed by load balancer service to different nodes underneath.

Usually cloud-provide's load balancer service offering is used to exposes this service externally.
Each cloud provider like e.g. AWS, Azure, GCP, etc has its own native load balancer implementation.
This type of service is heavily dependent on the cloud provider.

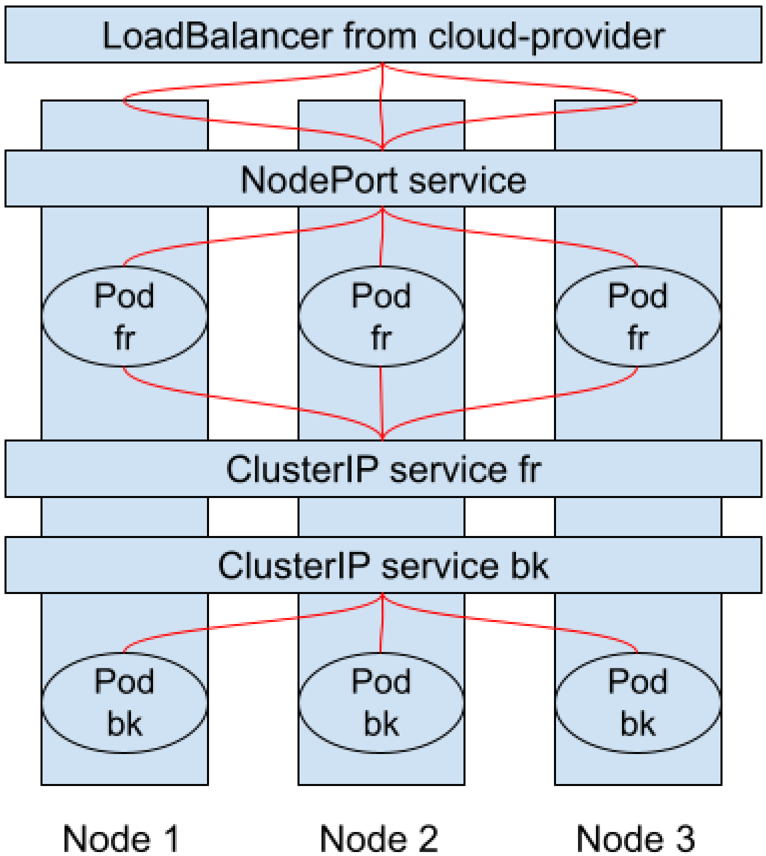
Load Balancer created by cloud provider routes the incoming requests to kubernetes services which sends the traffic further to backend pods.

Services summarized

Function:



Implementation:



Scheduling

Scheduling

A production kubernetes cluster consists of a number of worker nodes.

We run various types of workloads in any production environment.
There could be applications which need to be run on hardware with specific configuration, some applications may require specific tweaks in the OS for it to run effectively.
In the kubernetes cluster how can we guarantee that the pod for a specific application will always pick the suitable node to run.

Kubernetes has different scheduling capabilities to achieve this.

Manual Scheduling

Add below in pod’s specs section

nodeName: node01

We cannot move a running pod from one system to another.
We need to delete the pod from one node and re-create in other.

Selectively list the desired pods.

kubectl get pods --selector app=nginx-server

Pods with label app=nginx-server will only be listed with above command.

Taints and Tolerations

kubectl taint nodes node-name key=val:taint-effect

taint effects:
noschedule - pods will not be scheduled on the node
prefer no schedule - system will try to avoid placing pod on node but it's not guaranteed
noexecute - new pods will not scheduled on the node and existing pods will be deleted from it

kubectl taint nodes node1 app=redis:NoSchedule

Toleration:
added to pods
under specs, add section

```
tolerations:
- key: "app"
  operator: "Equal"
  value: "redis"
  effect: "NoSchedule"
```

taints and tolerations are not to tell the pod to go to a specific node but it is for nodes to accept only a certain pods.
If you want to restrict pods to certain nodes it can be node with node affinity.

taint on master node prevents any pods from being schedule on it

kuectl describe nodes kubemaster | grep Taint

kubectl taint node node01 app=nginx:NoSchedule
kubectl run nginx --image=nginx

kubectl describe pod nginx
see why pod is in pending state

kbectl run nginx2 --image=nginx --dry-run=client -o yaml

add section called

```
tolerations:
- key: "app"
  operator: "Equal"
  value: "redis"
  effect: "NoSchedule"
```

check taint on ctrl plain node
remove the taint on it
copy taint form describe command
kubectl taint node controlplane PasteTheTaintAndAppendDashToRemoveIt-

check the taint is removed now

cat deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-ha
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
        type: frontend
    spec:
      containers:
        - name: container-1
          image: nginx
      nodeName: node01
```

```yaml
  replicas: 4
  selector:
    matchLabels:
      type: frontend
```

```yaml
cat pod-nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-server-1
  labels:
    app: nginx-server
spec:
  containers:
    - name: container-1
      #image: ashishkshah/myrepo:test
      image: nginx
      ports:
        - containerPort: 80
  tolerations:
  - key: "app"
    operator: "Equal"
    value: "nginx-server"
    effect: "NoExecute"
```

Storage

As discussed during the docker session, docker images are layered one above other.
A new layer is created for every line (which modifieds the image) in the Dockerfile.

Old layers are reused from the cache when the action to be performed is the same in different Dockerfile. Thus it's faster and saves space as well.

Storage files in docker can be found at /var/lib/docker on host

Image layers are read only and can only be modified with image build.

creating container creates a writable layer on top of it
the rw layers are available only till the life of the container. When container is deleted this layer is lost hence data is ephemeral.

Docker uses a copy on write mechanism to modify the files in the image.

Any image file being modified is copied to the read-write layer first and then it is modified. Files in the image layer will not be modified as its read only layer.
This layer will only be modified upon docker build command to build the image.

To make the files modified persistent across container deletion we can use persistent volumes.

Two types of mounting
- Volume mount
- Bind mount

docker volume create myvol
creates vol directory at /var/lib/docker/volumes/myvol
docker run -v myvol:/var/lib/mysql mysql
mounts volume on host to mysql dir in container

if vol is not already created, docker will auto create it upon first use.

To use external storage on host instead of volume run:
docker run -v /data/mysql:/var/lib/mysql mysql
The path of mounted storage or directory on host is mentioned in the command instead of volume name.

alternate command (more preferred):
docker run --mount type:bind,source=/data/mysql,target=/var/lib/mysql mysql

Storage drivers responsible of doing all storage related operations like managing layers, maintaining files across the layers, etc.

aufz
zfs
btrfs
device mapper
overlay
overay2

Like Storage drivers manage the storage we have volume driver for volume management in docker
default volume driver plugin is local
There are other storage vendor specific drivers
e.g.
Azure FS storage, Convoy, gcw-docker, flusterfa, netapp, vmware vsphere storage, etc

Like CRI and CNI - CSI is developed to support multiple storage solutions.
different storage vendors have their own csi drivers. this is not k8s specific standard. It is a universal standard meant for any container orchestration tool to work with any storage vendor with supported plugins.

Volumes in k8s

```
cat pod-storage.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    app: nginx
    type: frontend
    subject: devops
spec:
  containers:
    - name: container-1
      image: nginx
      volumeMounts:
        - mountPath: /usr/share/nginx/html/
          name: nginx-home
          readOnly: false

  volumes:
    - name: nginx-home
      hostPath:
        path: /data
```

Check where pod is running and then create /data/index.html in that node (or use preferredNode option)

Persistent Volumes

In the above method, the user has to know the availability of the storage volume tobe used. This must not be the case. Users should not be bothered about finding the suitable storage space but just ask for it.
System administrator should be responsible for making the storage space available.

One level of abstraction is achieved in allocation of storage space using persistent volumes and claims.

System administrators create persistent volumes on the kubernetes cluster.
Users just claim the storage space from the available pool of persistent volumes.

PVs are visible throughout the entire cluster. They are not namespace specific.
Claims are specific to a namespace.

```
cat pv.yml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nginx-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /data
```

```
cat pvc.yml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nginx-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 50Mi
```

```
cat pod-pvc.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    app: nginx
    type: frontend
```

```
    subject: devops
spec:
  containers:
    - name: container-1
      image: nginx
      volumeMounts:
      - mountPath: /usr/share/nginx/html/
        name: mydrive
  volumes:
    - name: mydrive
      persistentVolumeClaim:
        claimName: nginx-claim
```

Lab setup

Minikube installation on Fedora OS

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-latest.x86_64.rpm
sudo rpm -Uvh minikube-latest.x86_64.rpm

dnf install docker
systemctl enable docker.service --now
Useradd kube
Passwd kube

usermod -aG docker $USER && newgrp docker
echo -e " kube    ALL=(ALL)       ALL" >> /etc/sudoers
Su - kube

$ minikube start --driver=docker
```

Install kubectl

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch
```

enabled=1
gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
sudo yum install -y kubectl
kubectl get pods -A

Bash auto completion:
kubectl completion bash | sudo tee /etc/bash_completion.d/kubectl > /dev/null
source /usr/share/bash-completion/bash_completion
Kubectl <tab> <tab>

Multi node kubernetes setup

Execute steps 1 to 6 on all the nodes

1) Set static ip, gateway and dns configuration:

nmcli connection show
nmcli

sudo nmcli con modify NAME ifname NAME ipv4.method manual ipv4.addresses 192.168.122.10/24 gw4 192.168.122.1

sudo nmcli con mod NAME ipv4.dns 192.168.122.1

nmcli con down NAME
nmcli con up NAME

sudo hostnamectl set-hostname master-node

reboot

2) Disable swap

rpm -e zram-generator-defaults zram-generator
swapoff -a
free

3) Configure repo

```
cat <<EOF > /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
EOF
```

4) Install packages and enable service

```
yum install -y docker kubelet kubeadm kubectl crio

systemctl enable docker --now
systemctl enable kubelet --now
systemctl enable crio --now
```

5) Configure firewall rules

```
sudo firewall-cmd --permanent --add-port=6443/tcp
sudo firewall-cmd --permanent --add-port=2379-2380/tcp
sudo firewall-cmd --permanent --add-port=10250/tcp
sudo firewall-cmd --permanent --add-port=10251/tcp
sudo firewall-cmd --permanent --add-port=10252/tcp
sudo firewall-cmd --permanent --add-port=10255/tcp
sudo firewall-cmd --reload
```

6) Enable bridge-nf-call-iptables in sysctl config file

```
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sysctl --system
```

7) Execute steps 1 to 6 on Mster node and then run below command:

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

sudo kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

8) Run steps 1 to 6 on worker node and then join the node to master

```
 kubeadm join 192.168.122.10:6443 --token fm2fjd.940uvjc3tqi6pgjx
--discovery-token-ca-cert-hash
sha256:a8989efa29ec4e6f4c343d104b3f66f84da855992d795ddc9082522bc39b9346
```

Resources:

https://kubernetes.io/docs/concepts/
https://www.redhat.com/en/topics/containers/what-is-kubernetes
https://www.redhat.com/en/topics/containers/kubernetes-architecture
https://minikube.sigs.k8s.io/docs/start/
https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-using-native-package-management
https://kubernetes.io/docs/tasks/tools/included/optional-kubectl-configs-bash-linux/
https://phoenixnap.com/kb/how-to-install-kubernetes-on-centos