1. Cluster Architecture an Install- 25%

1. 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

- **Creating Role:**
  - Rbac.authorization.k8s.io api Group
  - Always sets the namespace it belongs in
  -

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-and-pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
  resourcesNames: [“pod1”, “pod2”]
```

- **Creating ClusterRole:**

A ClusterRole can be used to grant the same permissions as a Role. Because ClusterRoles are cluster-scoped, you can also use them to grant access to:

- cluster-scoped resources (like nodes)
- non-resource endpoints (like `/healthz`)
- namespaced resources (like Pods), across all namespaces
  For example: you can use a ClusterRole to allow a particular user to run `kubectl get pods --all-namespaces`

Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]  #Nodes etc
  verbs: ["get", "watch", "list"]
```

- **Creating RoleBinding/ClusterRoleBinding**
  A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

  Here is an example of a RoleBinding that grants the "pod-reader" Role to the user "jane" within the "default" namespace. This allows "jane" to read pods in the "default" namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

  To grant permissions across a whole cluster, you can use a ClusterRoleBinding. The following ClusterRoleBinding allows any user in the group "manager" to read secrets in any namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
  Namespace: default     //optional
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

  To allow a subject to read `pods` and also access the `log` subresource for each of those Pods, you write:

```yaml
resources: ["pods", "pods/log"]
```

// Basic commands

```
Kubelet create -f <file-name>.yaml
kubectl get roles
kubelet get rolebindings
kubelet describe role <role_name>
kubectl describe rolebining <role_binding_name>
```

// Check Access:

```
kubectl auth can-i create deployments
kubectl auth can-i <operation> <object_name>            # syntax
kubectl auth can-i <operation> <object_name>  --as <user_name> -namespace <namepscae_name>
```

Command-line utilities

kubectl create role:

Creates a Role object defining permissions within a single namespace. Examples:

- Create a Role named "pod-reader" that allows users to perform `get`, `watch` and `list` on pods:

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

- Create a Role named "pod-reader" with resourceNames specified:

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

- Create a Role named "foo" with apiGroups specified:

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

- Create a Role named "foo" with subresource permissions:

```
kubectl create role foo --verb=get,list,watch --resource=pods,pods/status
```

- Create a Role named "my-component-lease-holder" with permissions to get/update a resource with a specific name:

```
kubectl create role my-component-lease-holder --verb=get,list,watch,update --resource=lease
--resource-name=my-component
```

kubectl create clusterrole

Creates a ClusterRole. Examples:

- Create a ClusterRole named "pod-reader" that allows user to perform `get`, `watch` and `list` on pods:

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

- Create a ClusterRole named "pod-reader" with resourceNames specified:

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

- Create a ClusterRole named "foo" with apiGroups specified:

```
kubectl create clusterrole foo --verb=get,list,watch --resource=replicasets.apps
```

- Create a ClusterRole named "foo" with subresource permissions:

```
kubectl create clusterrole foo --verb=get,list,watch --resource=pods,pods/status
```

- Create a ClusterRole named "foo" with nonResourceURL specified:

```
kubectl create clusterrole "foo" --verb=get --non-resource-url=/logs/*
```

- Create a ClusterRole named "monitoring" with an aggregationRule specified:

```
kubectl create clusterrole monitoring
--aggregation-rule="rbac.example.com/aggregate-to-monitoring=true"
```

kubectl create rolebinding

Grants a Role or ClusterRole within a specific namespace. Examples:

- Within the namespace "acme", grant the permissions in the "admin" ClusterRole to a user named "bob":

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

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to the service account in the namespace "acme" named "myapp":

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

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to a service account in the namespace "myappnamespace" named "myapp":

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

kubectl create clusterrolebinding

- Grants a ClusterRole across the entire cluster (all namespaces). Examples:
  Across the entire cluster, grant the permissions in the "cluster-admin" ClusterRole to a user named "root":

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

- Across the entire cluster, grant the permissions in the "system:node-proxier" ClusterRole to a user named "system:kube-proxy":

```
kubectl create clusterrolebinding kube-proxy-binding --clusterrole=system:node-proxier
--user=system:kube-proxy
```

- Across the entire cluster, grant the permissions in the "view" ClusterRole to a service account named "myapp" in the namespace "acme":

```
kubectl create clusterrolebinding myapp-view-binding --clusterrole=view
```

Example 1: *Create a new service account named "sa" in the development namespace. Create a cluster role called "pod-reader," having permission to get pod and list pods. "sa" should be able to get pod and list pods.*

```
kubectl create namespace development
kubecrl create serviceaccount sa -n development
kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods
kubectl create clusterrolebinding pod-reader --clusterrole=pod-reader --serviceaccount=development:sa
kubectl auth can-i list pods --development target --as system:serviceaccount:development:sa
```

Task 2:

1. Create two *Namespaces* `ns1` and `ns2`
   ```
   kubectl create ns ns1
   kubectl create ns ns2
   ```
2. Create *ServiceAccount* (*SA*) `pipeline` in both *Namespaces*
   ```
   kubectl create sa pipeline
   ```
3. These *SAs* should be allowed to view almost everything in the whole cluster. You can use the default `ClusterRole view` for this:
   ```
   kubectl create clusterrolebinding pipeline-view --clusterrole view --serviceaccount
   ns1:pipeline --serviceaccount ns2:pipeline
   ```
4. These *SAs* should be allowed to create and delete *Deployments* in *Namespaces* `ns1` and `ns2`
   ```
   Kubectl create clusterrolbinding pipeline-view –clusterrole=view –serviceaccount=ns1:pipeline
   –serviceaccount=ns2:pipeline
   ```
5. Verify everything using `kubectl auth can-i`
   ```
   Kubectl auth can-i  create deployment –as system:ns1:pipeline
   ```

2. Version upgrades on kubernetes cluster:

1. OS Upgrades:

- Whenever the node goes down, the pods that are running will go offline .
- Those pods that are part of the replica set will recreate these pods in other nodes and those which are not part of the replica set will not be scheduled . and after pod eviction times , only empty nodes will come up without any pods in that particular node.
- *Pod eviction Timing*: The time in which master nodes wait for the pod to come back to online state after it went offline, ~5 min default value
- So, if we are not sure about the nodes that come back online within 5 mins, we cannot do upgrades directly on the nodes.
- Safer way is, `drain` the nodes
- Doing this all the pods will be moved to other nodes safely.
- Also node is marked as `unSchedule` meaning no pods are scheduled on this node.
  ```
  > kubectl drain <node-name>
  > kubectl drain <node-name> --ignore-daemonsets  // in case daemonsets are available
  ```

- *Cordon*: it simply mark nodes as unschedule , unless like drain it moves pods to other nodes
  ```
  > kubectl cordon <node_name>
  ```

- *Uncordon* : to set back the node to normal scheduling
  ```
  > kubectl uncordon <node_name>
  ```

Notes:

- When the control plane does not have any taints , still when the node is cordoned , pods will be scheduled in controlplane.
- when nodes contain pods that are not part of a replica set, when trying to cordon it, it wont work and throws an error. when forcefulling drained it , that particular pod will be lost permanently

2. Kubernetes Software version:

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls.

- kubernetes consists of 3 parts:
  V1.11.3
  Major.minor.patch
  -Features - Bug Fixes
  -Functionalities

- Standard software release
- first release v1.0
- latest stable version v1.13.0
- alpha & beta release v1.10.0-alpha /v1.10.0.beta / v1.10.0

3. Cluster upgrades process:

Level 1: kube-apiserver
X v1.10

Level 2: controller-manager kube-scheduler kubectl
X-1 v1.9 or v1.10 X-1 v1.9 or v1.10 X+1 > X-1

Level 3 : kubelet kube-proxy
X-2 v1.8 or v1.9 or v1.10 X-2 v1.8 or v1.9 or v1.10

- so, kube-apiserver is the highest level and all lower level component versions cannot be higher than kube-apiserver.
- kubernetes always support latest 3 minor versions are available
- always good practice to release before release of 3rd version is good to upgrade cluster
- recommended practice to upgrade is 1 level higher than previous
- Upgrades involves 2 major steps:
  1. upgrade master node
  2. upgrade worker node

a. Upgrade master:

- Using kubeadm:

```
> kubeadm version
```

```
// Download needed version of kudeadm
# replace x in 1.23.x-00 with the latest patch version
> apt-mark unhold kubeadm && \
```

```
  apt-get update && apt-get install -y kubeadm=1.23.x-00 && \
  apt-mark hold kubeadm
```

// verify the downloaded version

```
> kubeadm version
```

// Check for the available plan to upgrade cluster

```
> kubeadm upgrade plan
```

// Apply the plan

```
> kubeadm upgrade apply <version>   # v1.23.x
```

- When the master is upgrading ,the master node goes down, no new scheduling of pods/app happens. and all the pods that are in worker nodes will continue running.
- when pod fails, no new pod is scheduled
- when the cluster is upgraded, functions become normal.
- **Other control plane nodes:**
  - `sudo kubeadm upgrade node`
  - Drain the nodes:

    `kubectl drain <node-to-drain> --ignore-daemonsets`
- **Upgrade kubelet and kubectl:**

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.23.x-00 kubectl=1.23.x-00 && \
apt-mark hold kubelet kubectl
```

  - Restart the kubelet:

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

  - Uncordon the node:

    Bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

b. Upgrade worker node:

I. strategy-1 :
- upgrade all worker nodes at a time
- requires downtime

II. strategy-2 :
- upgrade each node one by node
- while upgrading each node, all workloads will be shifted to other nodes.

III. Strategy-3:

- to add new node to the cluster
- move all workloads to the new node.
  - Upgrade kubeadm:

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.23.x-00 && \
apt-mark hold kubeadm
> sudo kubeadm upgrade node
```

  - Drain the nodes:

```
> kubectl drain <node-to-drain> --ignore-daemonsets   [run on master node]
```

  - **Upgrade kubelet and kubectl:**

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.23.x-00 kubectl=1.23.x-00 && \
apt-mark hold kubelet kubectl
```

  - Restart the kubelet:

```
> sudo systemctl daemon-reload
> sudo systemctl restart kubelet
```

  - Uncordon the node:

    Bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
> kubectl uncordon <node-to-drain>
```

4. **Implement etcd backup and restore:**

Scenario 1:

The master nodes in our cluster are planned for a regular maintenance reboot tonight. While we do not anticipate anything to go wrong, we are required to take the necessary backups. Take a snapshot of the ETCD database using the built-in snapshot functionality.

Store the backup file at location /opt/snapshot-pre-boot.db

Backup ETCD to /opt/snapshot-pre-boot.db and restore

Note: In this case, we are restoring the snapshot to a different directory but in the same server where we took the backup (the controlplane node) As a result, the only required option for the restore command is the --data-dir.

Next, update the /etc/kubernetes/manifests/etcd.yaml:

We have now restored the etcd snapshot to a new path on the controlplane - /var/lib/etcd-from-backup, so, the only change to be made in the YAML file, is to change the hostPath for the volume called etcd-data from old directory (/var/lib/etcd) to the new directory /var/lib/etcd-from-backup.

Solution:

```
> ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379
--cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt
--key=/etc/kubernetes/pki/etcd/server.key  snapshot save /opt/snapshot-pre-boot.db

> service kube-apiserver stop            # optional

> ETCDCTL_API=3 etcdctl --data-dir /var/lib/etcd-from-backup snapshot restore
/opt/snapshot-pre-boot.db

> service kube-apiserver start       # optional

> change/edit  /etc/kubernetes/manifests/etcd.yaml
```

Replace …

```
...
volumes:
...
...
- hostPath:
    path: /var/lib/etcd-from-backup   #new-data-dir. In my case, I have already configured it.
    type: DirectoryOrCreate
  name: etcd-data
```

Restart etcdb:

```
> systemctl daemon-reload
> service etcd restart
```

start kubeapi server:

```
> service kube-apiserver start
```

Troubleshoot:

Error: # failed to load Kubelet config file /var/lib/kubelet/config.yaml, error failed to read kubelet config file "/var/lib/kubelet/config.yaml", error: open /var/lib/kubelet/config.yaml: no such file or directory

Check logs: `journalctl -u kubelet`

You can execute **`kubeadm init phase kubelet-start`** to only invoke a particular step that will write the kubelet configuration file and environment file and then start the kubelet.

5. Use kubeadm to install K8s:

Steps:

1. Install docker
2. Install kubeadm /kubelet / kubectl
3. Initialize - master
4. Pod network - all nodes
5. Join Node - worker nodes

To create VMs using vagrant: https://github.com/kodekloudhub/certified-kubernetes-administrator-course/

```
=====================================================

IFNAME=$1
ADDRESS="$(ip -4 addr show $IFNAME | grep "inet" | head -1 |awk '{print $2}' |
cut -d/ -f1)"
sed -e "s/^.*${HOSTNAME}.*/${ADDRESS} ${HOSTNAME} ${HOSTNAME}.local/" -i
/etc/hosts

# remove ubuntu-bionic entry
sed -e '/^.*ubuntu-bionic.*/d' -i /etc/hosts
sed -i -e 's/#DNS=/DNS=8.8.8.8/' /etc/systemd/resolved.conf

# Update /etc/hosts about other hosts
cat >> /etc/hosts <<EOF
192.168.33.13 master
192.168.33.14 worker-1
192.168.33.15 worker-2
EOF

# https://kubernetes.io/docs/setup/production-environment/container-runtimes/#containerd
apt-get update
apt-get install containerd -y

mkdir -p /etc/containerd
containerd config default  /etc/containerd/config.toml

# https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
#install kubectl & kubeadm and kubelet
apt-get update &&  apt-get install -y apt-transport-https gnupg2 curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |  apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" |  tee -a
/etc/apt/sources.list.d/kubernetes.list
apt-get update
apt-get install -y kubelet kubeadm kubectl

apt-get install bash-completion
```

```
echo 'source <(kubectl completion bash)' >>~/.bashrc
#source /usr/share/bash-completion/bash_completion
kubectl completion bash >/etc/bash_completion.d/kubectl

# Set iptables bridging
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo echo '1' > /proc/sys/net/ipv4/ip_forward
sudo sysctl --system

#load a couple of necessary modules
sudo modprobe overlay
sudo modprobe br_netfilter
#disable swaping
#sed 's/#   /swap.*/#swap.img/' /etc/fstab
#sudo swapoff -a

service systemd-resolved restart
============================================================

# master as a regular user
# mkdir -p $HOME/.kube
# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
# sudo chown $(id -u):$(id -g) $HOME/.kube/config

# if coredns doesn't work , use flannel
```

Initializing your control-plane node

The control-plane node is the machine where the control plane components run, including etcd (the cluster database) and the API Server (which the kubectl command line tool communicates with).

```
kubeadm init --pod-network-cidr 10.244.0.0/16
--apiserver-advertise-address=192.168.33.13
```

// --apiserver-advertise-address = <MASTER_NODE_IP_ADDRESS> #To verify > ifconfig en0s8

Output looks as below:

To start using your cluster, you need to run the following as a regular user:

// Run this below commands on all nodes to be able to access kube files=>

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.56.2:6443 --token fcp4l1.u0h4p6xmf8kj8fas \
        --discovery-token-ca-cert-hash
sha256:e6aa08fe66cd32198bd3181488ae45300b78a023d52afc50273ed2375e18b987
```

Troubleshooting:

Kubelete connection refused:

docker info | grep Cgroup

The result of the above command would be something like this:
Cgroup Driver: cgroupfs Cgroup Version: 1

Then, update kubelet args (KUBELET_KUBECONFIG_ARGS) in /etc/systemd/system/kubelet.service.d/10-kubeadm.conf and add a --cgroup-driver flag corresponding to docker cgroup (in this case cgroupfs).

My config file looks like this after the modification:
... Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/e`tc/kubernetes/kubelet.conf --cgroup-driver=cgroupfs" ...

Finally, run kubeadm reset and then kubeadm init.

Installing POD network:

Several external projects provide Kubernetes Pod networks using CNI, some of which also support Network Policy.

See a list of add-ons that implement the Kubernetes networking model.

You can install a Pod network add-on with the following command on the control-plane node or a node that has the kubeconfig credentials:

```
> kubectl apply -f <add-on.yaml>
```

You can install only one Pod network per cluster.

Eg. to install weave plugin:

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

Note: Once this is done, now get into worker nodes and run the kubeadm join command as mentioned above

6. Manage High Availability k8s cluster:

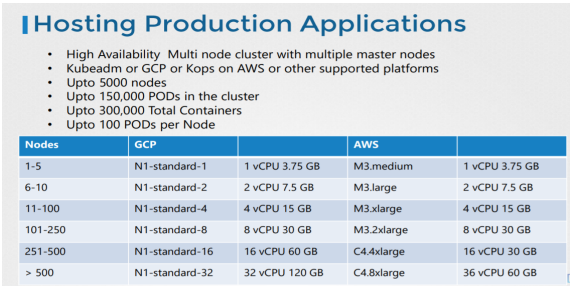
Designing K8S cluster:

What to be analysed?

- Purpose
- Education
  - Development & Testing
  - Hosting Production Applications
- Cloud or OnPrem?
- Workloads
  - How many?
  - What kind?
    - Web
    - Big Data/Analytics
- Application Resource Requirements
  - CPU Intensive
  - Memory Intensive
- Traffic
  - Heavy traffic
  - Burst Traffic

1. Purpose:

- Education
  - Minikube
  - Single node cluster with kubeadm/GCP/AWS
- Development & Testing
  - Multi-node cluster with a Single Master and Multiple workers
  - Setup using kubeadm tool or quick provision on GCP or AWS or AKS
- Hosting Production Applications



Nodes	GCP		AWS	
1-5	N1-standard-1	1 vCPU 3.75 GB	M3.medium	1 vCPU 3.75 GB
6-10	N1-standard-2	2 vCPU 7.5 GB	M3.large	2 vCPU 7.5 GB
11-100	N1-standard-4	4 vCPU 15 GB	M3.xlarge	4 vCPU 15 GB
101-250	N1-standard-8	8 vCPU 30 GB	M3.2xlarge	8 vCPU 30 GB
251-500	N1-standard-16	16 vCPU 60 GB	C4.4xlarge	16 vCPU 30 GB
> 500	N1-standard-32	32 vCPU 120 GB	C4.8xlarge	36 vCPU 60 GB

2. Cloud or OnPrem?

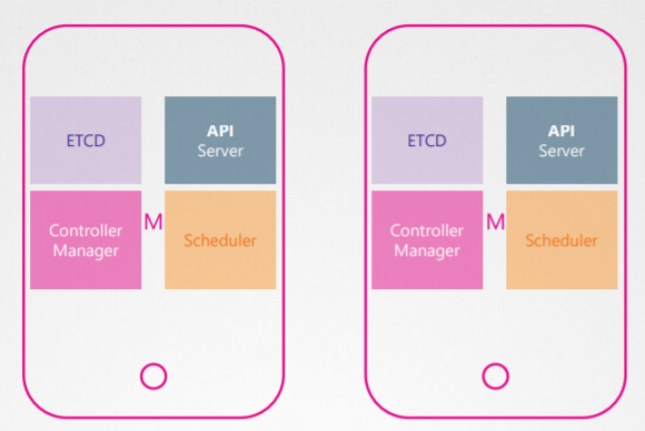
- Use Kubeadm for on-prem
- GKE for GCP

- Kops for AWS
- Azure Kubernetes Service(AKS) for Azure

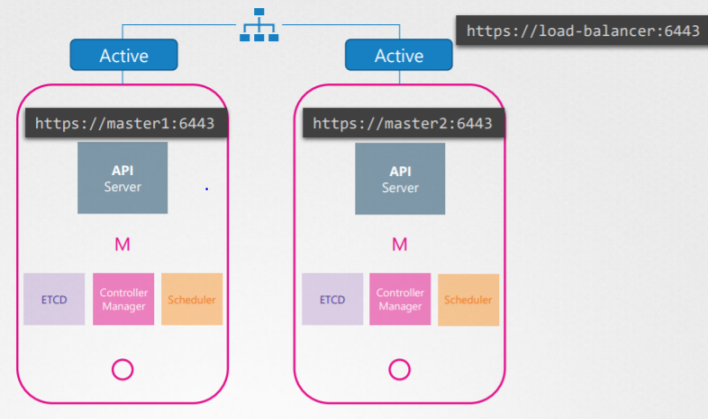
3. **Storage:**
    - High Performance – SSD Backed Storage
    - Multiple Concurrent connections – Network based storage
    - Persistent shared volumes for shared access across multiple PODs
    - Label nodes with specific disk types
    - Use Node Selectors to assign applications to nodes with specific disk types
4. **Nodes:**
    - Virtual or Physical Machines
    - Minimum of 4 Node Cluster (Size based on workload)
    - Master vs Worker Nodes
    - Linux X86_64 Architecture
    - Master nodes can host workloads
    - Best practice is to not host workloads on Master nodes

Maintaining Highly Available ControlPlane?

It is necessary to maintain HA for cluster control planes to have high fault tolerance in production.

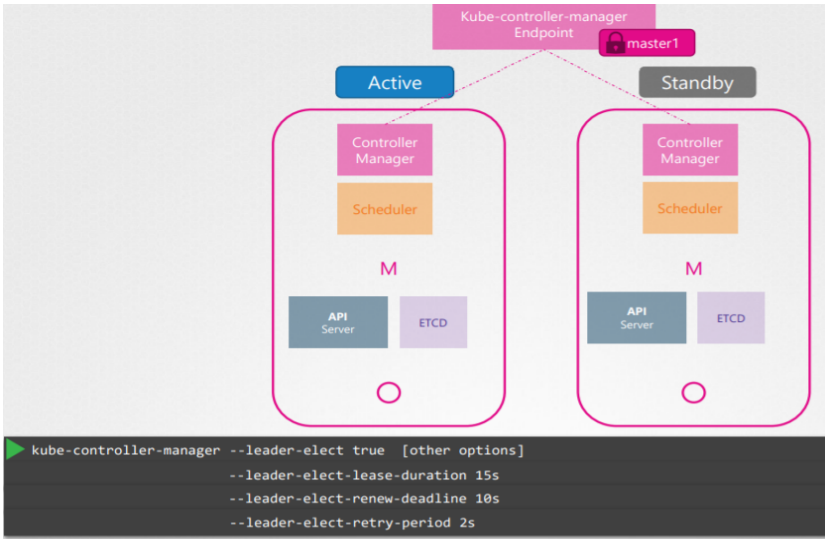


1. Create load balancer for kube-apiserve
    - We know that API server is responsible for receiving request and processing them and providing information to control plane. They work on 1 request at a time.
    - Create a kube-apiserver load balancer with a name that resolves to DNS.
        - In a cloud environment you should place your control plane nodes behind a TCP forwarding load balancer. This load balancer distributes traffic to all healthy control plane nodes in its target list. The health check for an apiserver is a TCP check on the port the kube-apiserver listens on (default value `:6443`).
        - The load balancer must be able to communicate with all control plane nodes on the apiserver port. It must also allow incoming traffic on its listening port.
        - Make sure the address of the load balancer always matches the address of kubeadm's `ControlPlaneEndpoint`.



2. Scheduler & controller manager:
   If multiple instances are present for this kind, then there will be duplicate work.
   This can be achieved by active and standby methods, when one is active the other will be in standby mode by applying a lock on the other instance.

   Whichever process first updates the endpoint will get the active status and the other will be locked.



   Even the scheduler follows the same method as mentioned above.

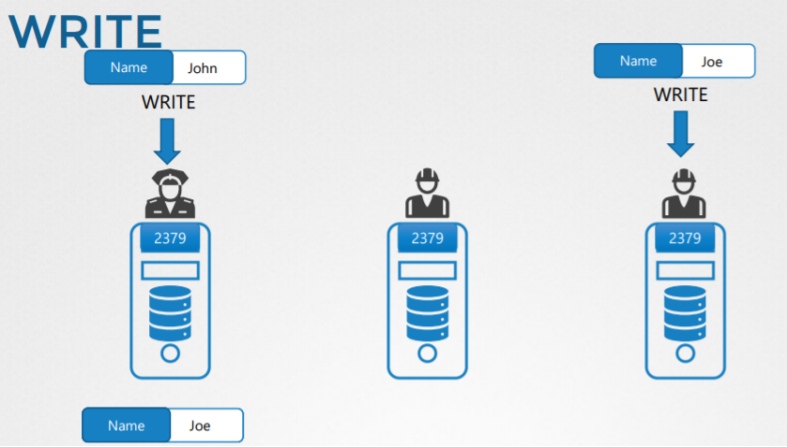
3. **ETCD in HA:**
   ETCD is a distributed reliable key-value store that is Simple, Secure & Fast.
   It is possible to have key-value stores in multiple instances .
   How to make sure data on all instances is consistent ?

   READ/WRITE

   Read? It is possible to get the data

   Write? What if the write request comes from 2 instances out of 4 instances ?

   Here it actually does not store them on all the instances instead it follows leader and other nodes become followers, where leader store that particular data and it makes sure that others get the data to store it and works vice versa.



Note: **Quorum = N/2 + 1** Minimum number of nodes needed for a cluster to function .

```
wget -q --https-only \
"https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"
tar -xvf etcd-v3.3.9-linux-amd64.tar.gz
mv etcd-v3.3.9-linux-amd64/etcd* /usr/local/bin/
mkdir -p /etc/etcd /var/lib/etcd
cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```