KUBERNETES COMPLETE

- Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications.
- A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

POD:

- A Pod is the basic execution unit of a Kubernetes application--the smallest and simplest unit in the Kubernetes object model that you create or deploy.
- **Each POD many have multiple containers. But "one-container-per-Pod" is recommended.
- Each Pod is meant to run a single instance of a given application.
- Pods provide two kinds of shared resources for their constituent containers: networking and storage.
- Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports.
- 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.
- A Pod is not a process, but an environment for running a container. A Pod persists until it is deleted.

Terminology and Controllers:

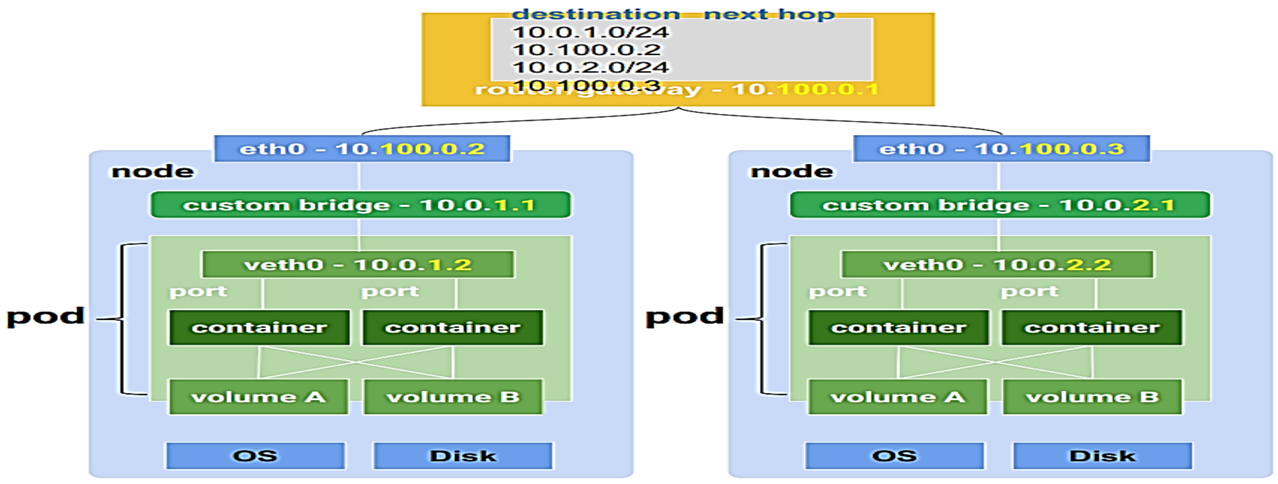
- **ReplicaSet:** the default, is a relatively simple type. It ensures the specified number of pods are running
- **Deployment:** is a declarative way of managing pods via ReplicaSet. Includes rollback and rolling update mechanisms
- **Daemonset:** is a way of ensuring each node will run an instance of a pod. Used for cluster services, like health monitoring and log forwarding
- **StatefulSet:** is tailored to managing pods that must persist or maintain state
- **Job and CronJob:** run short-lived jobs as a one-off or on a schedule.

NODES:

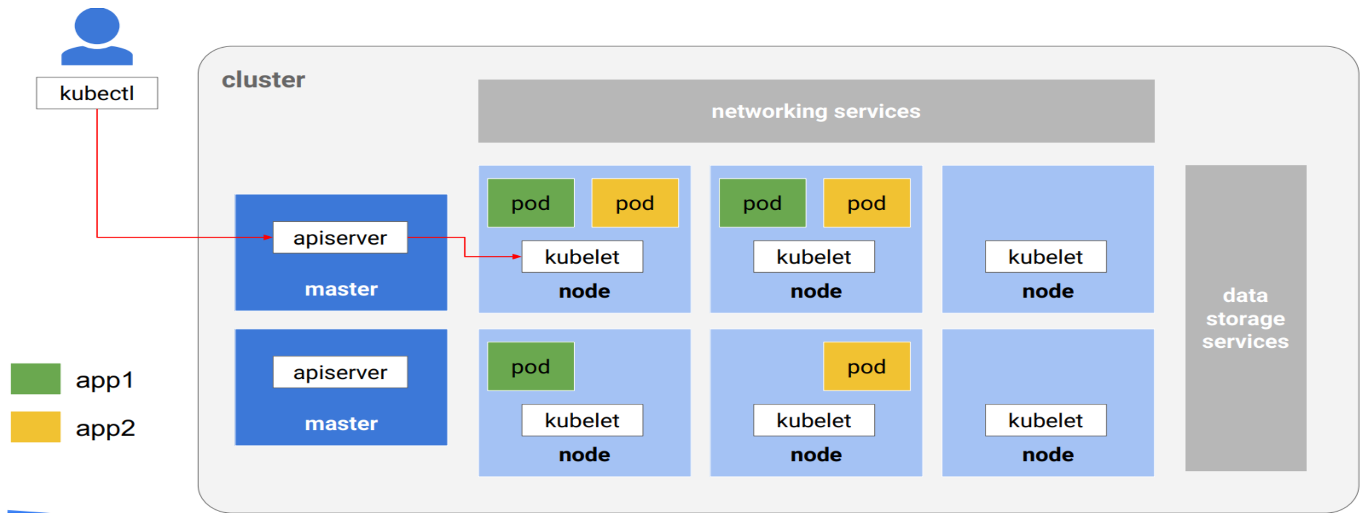
A node is a worker machine in Kubernetes, previously known as a minion. A node may be a VM or physical machine. Each node contains the services necessary to run pods and is managed by the master components. The services on a node include the container runtime, kubelet and kube-proxy. Node is externally created by cloud providers.

Kubernetes node Components interface:

Node controller: is a Kubernetes master component which manages various aspects of nodes like Node status, Node health, assign cidr block to node.
Kubelet: An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
kube-proxy: kube-proxy is a network proxy that runs on each node in your cluster. kube-proxy maintains network rules on nodes.
Container Runtime: The container runtime is the software that is responsible for running containers.
*****Heartbeats, sent by Kubernetes nodes, help determine the availability of a node.**

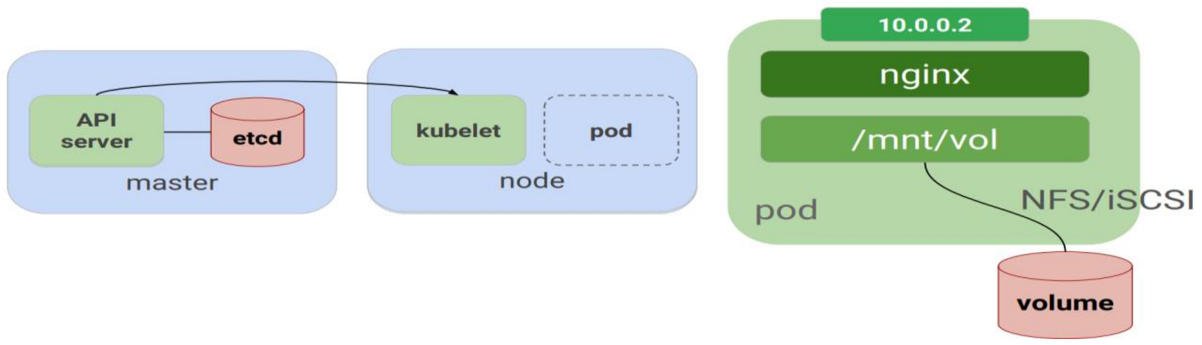


<u>Cluster:</u>
A cluster is a set of computers working as an instance managed by Kubernetes. You can have up to 5,000 nodes in a cluster. Regional clusters have masters and nodes spread across 3 zones for high availability and resilience from single zone failure and downtime during master upgrades.



CLUSTER >>> NODE >>> POD
CLUSTER = NODES + MASTER
NODE = PODS + Containers
POD = Containers
<u>Labels</u>:
Labels are arbitrary metadata you can attach to any object in the Kubernetes API. Labels tell you how to group these things to get an identity. This is the only way you can group things in Kubernetes.
EX: 4pods 3 Labels   labels APP: MYAPP; Phase: test/prod; Role: Front End/ Back End/

<u>Volumes:</u>
Volumes are a way for containers within a pod to share data, and they allow for Pods to be stateful. These are two very important concerns for production applications.
There are many different types of volumes in Kubernetes. Some of the volume types include long-lived persistent volumes, temporary, short-lived
NOTE: Kubernetes *persistent volumes* are administrator provisioned volumes. These are created with a particular filesystem, size, and identifying characteristics such as volume IDs and names.



https://kubernetes.io/docs/concepts/storage/volumes/
Different Volumes:

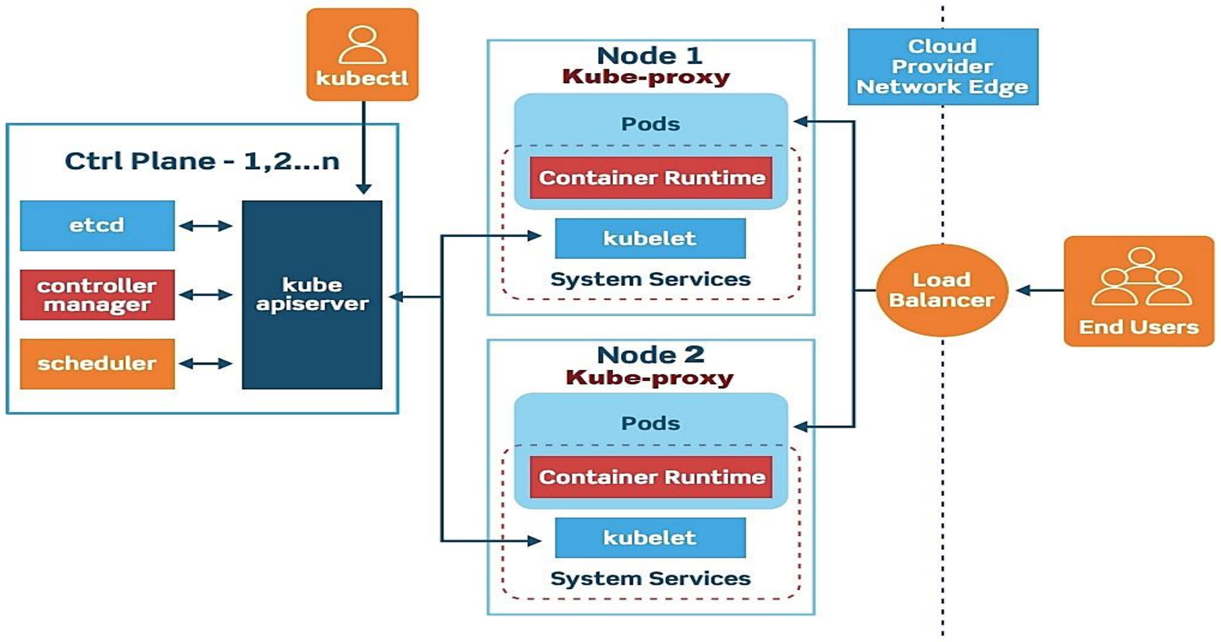
1. **LOCAL:** A local volume represents a mounted local storage disk or Dir. Used only for static provision of persistent volume. More durable than Host-path.
2. **AWS ELASTICBLOCKSTORE:** Mounts an Amazon Web Services (AWS) EBS Volume into your Pod. Nodes on which Pods are running must be AWS EC2 instances. Instances need to be in the same region and availability-zone.

3. **NFS (NETWORK FILE SYSTEM):** NFS volume allows an existing NFS (Network File System) share to be mounted into your Pod.
4. **CONFIGMAP:** The configMap resource provides a way to inject configuration data into Pods. The data stored in a ConfigMap object. You must create a ConfigMap before you can use it.
5. **FLOCKER:** Flocker is an open-source clustered Container data volume manager. A flocker volume allows a Flocker dataset to be mounted into a Pod.
6. **GCEPERSISTENTDISK:** A gcePersistentDisk volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. You must create a PD using gcloud before you can use it.
7. **PERSISTENTVOLUMECLAIM:** A persistentVolumeClaim volume is used to mount a PersistentVolume into a Pod. User dont need to know where they are created.

Kubernetes - Cluster Architecture:

Kubernetes - Master Machine Components:

1. **ETCD:** It stores the configuration information which can be used by each of the nodes in the cluster. It is a distributed key value Store which is accessible to all.
2. **API SERVER(MAIN):** The Kubernetes API server validates and configures data for the api objects which include pods, services, replication-controllers, and others. It helps in communication of pods by using kubectl
3. **CONTROLLER MANAGER:** It is responsible for maintaining desired states mentioned in the manifest.



4. **SCHEDULER:** It watches for new work tasks and assigns them to healthy nodes in the cluster. The scheduler is responsible for workload utilization and allocating pod to new node.

- Pods.yml ----> To create only pods
- ReplicationController ---> To create replication on pods
- Service ---> To expose Node port or for Load Balance
- ReplicaSet ---> To create pods or versioning or rolling updates without downtime. Used in Deployment

Kubernetes Options for Installation

1. **Bare-Metal Installations:**

Start from OS   Install Every Component   K8s the hard way   Other Managed Installations:

2. **Popular options**

I. MINIKUBE (DEVELOPER)
II. MICRO K8S (DEVELOPER)
III. KUBE-ADMIN
IV. KOPS

3. **Cloud-Provider Installations:**

GKE AKS EKS

HEAPSTER is a performance monitoring, metrics collection system, and aslo allows events and signal generated by cluster compatible with Kubernetes versions 1.0. 6 and above
MINIKUBE is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your laptop for users looking to try out Kubernetes or develop with it day-to-day.
KUBELET: is an agent service which runs on each node and enables the slave to communicate with master. So Kubelet works on the description of containers provided to it in the podspec and make sure that the containers describe in the PodSpec are healthy and running.
KUBECTL is the platform using which you can pass commands to the cluster. So it basically provide the cli to run commands against the kubernetes cluster with various ways to create and manage the kubernetes components.
KUBEPROXY: It runs on each node and can do simple TCP/UDP packet forwarding across backend network service.

REPLICA SET VS REPLICATION CONTROLLER

- Replica Set manifest works on "Set based Selector" Which means it will use matchLabels and matchExpression to replicate the pods.
  Ex: matchLabels: env in (prod, qa); matchExpression: tier notin (frontend, backend)
- Replication Controller manifest works on "Equity based Selector" Which means If the label name matches then only it creates replica for that label-selectors.
  EX: APP = Ngnix; env = prod ; tier != frontend

KUBECTL DRAIN: command is used to drain a specific node during maintenance. Once this command is given, the node goes for maintenance and is made unavailable to any user.
INGRESS:
Ingress exposes **HTTP and HTTPS** routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.
Ingress is used to route host with a dns name using **load balancer** or **nginix controller** with **rules** that are defined in the resource file(yaml).
An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name based virtual hosting.
An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type **Service.Type=NodePort** or **Service.Type=LoadBalancer**.
In order for the Ingress resource to work, the cluster must have an ingress controller running. Ingress controllers are not started automatically with a cluster
TYPE of Ingress Controllers:
1. **HAProxy Ingress** is a highly customizable community-driven ingress controller for HAProxy.
2. **NGINX**, Inc. offers support and maintenance for the NGINX Ingress Controller for Kubernetes.
3. **AKS Application Gateway Ingress Controller** is an ingress controller that enables ingress to AKS clusters using the Azure Application Gateway….. etc.

The Ingress spec has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests.
In Ingress **spec field rules** are written. Each rule contains:
1. **Host:** which matches with Domain name and route to specific url. No host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
2: **Path:** is the pages where it has to direct like index or contact or career...A list of paths (for example, **/testpath**), each of which has an associated backend defined with a serviceName and servicePort. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.
3. **Backend:** used to navigate to a service also a default-http-backend by default. Default backend is often configured in an Ingress controller to service any requests that do not match a path in the spec.

Process of ingress

1. Deploy: To deploy a cluster we use NGINX reverse proxy or HAProxy that is called Ingress Controller.
Yaml files needed in deployment:

- **Deploymen**t➔Kind: Deployment with replicas, container as nginx ingress controller, env with pod name, ports for controller & args for the path where nginix controller to start it.
- **ConfigMap**➔Kind: ConfigMap where nginx error log, ssl-protocols are placed.
- **Service**➔Kind: Service with service of type NodePort with selector as nginix-ingress
- **Auth**➔Kind: ServiceAccount with Roles, clusterRoles, RoleBindings

2. Configure: It contains URL routes, rules which are called Ingress Resources.
Resources
1. **Ingress** ➔Kind: Ingress where host, path, backend are defined

$ ***kubectl apply -f <File>***
$ ***kubectl edit ingress <name>*** #To update an existing Ingress to add a new Host
$ ***kubectl describe ingress <name>***

API versions:

Alpha level:(for example, v1alpha1). prone to errors but the user can drop for support to rectify errors at any time.
Beta level:(for example, v2beta3). Scripts present in this version will be firm since because they are completely tested
Stable level:(vX X is integer) Stable level versions get many updates often

RECREATE AND ROLLING UPDATEDEPLOYMENT STRATEGY:

- **Recreate** is used to kill all the running (existing) replication controllers and creates newer replication controllers. Recreate helps the user in faster deployment whereas it increases the downtime.
- **Rolling update** also helps the user to replace the existing replica controller to newer ones. But, the deployment time is slow and in fact, we could say, there is no deployment at all
- Create deployment:

KUBERNETES NAME SPACE:

- Namespaces are given to provide an identity to the user to differentiate them from the other users. Namespace assigned to a user must be unique
- Namespaces assist information exchange between pod to pod through the same namespace. Namespaces are a way to divide cluster resources between multiple users.
- Kubernetes starts with three initial namespaces:

DEFAULT: The default namespace for objects with no other namespace
KUBE-SYSTEM: The namespace for objects created by the Kubernetes system
KUBE-PUBLIC: This namespace is created automatically and is readable by all users

KUBERNETES LOAD BALANCING:

Internal load balancing: used to balance the loads automatically and allocates the pods within the necessary configuration
External load balancing: It transfers or drags the entire traffic from the external loads to backend pods.

SCALING KUBERNETES ON APPLICATION AND INFRASTRUCTURE LEVELS

Infrastructure Level: Utilization of resources like RAM and DISK

1. **Vertical Scaling of Kubernetes Nodes**: Dynamic allocation reserved resources. (Standalone node)
2. **Horizontal Auto-Scaling**: New nodes will be added to a cluster when RAM, CPU, I/O or Disk usage reaches certain levels.

Application Level:

1. **Horizontal pod auto-scalers (HPA):** If CPU consumption of all pods grows more than, say, 70%, HPA will schedule more pods, and when CPU consumption gets back to normal, deployment is scaled back to the original number of replicas.

```
$  kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=MINPODS] --max=MAXPODS [--cpu-percent=CPU]
$ kubectl autoscale deployment wordpress --cpu-percent=70 --min=1 --max=10 -n wp
```

KUBERNETES YAML:

API ---- KIND ---- Format

```
#### simple-pod.yml ###########
apiVersion: v1
kind: Pod
metadata:
  name: simple-pod
  labels:
    env: test
    app: gol
spec:
  containers:
  - name: nginx
    image: crsreddy1447/gol:1.0
    ports:
    - containerPort: 8080
```

```
###### Service ######
simple-pod-svc.yml:
apiVersion: v1
kind: Service
metadata:
  name: simple-svc
spec:
  selector:
    app: gol
  type:  NodePort
  ports:
  - name:  https
    port:  8080
    nodePort: 30002
    protocol: TCP
```

```
############################ ReplicationController ################################
simple-pod-replication.yml:
apiVersion: v1
kind: ReplicationController
metadata:
  name: simple-rc
spec:
  replicas: 3
  selector:
    app: gol
  template:
    metadata:
      labels:
        app: gol
        ver: "1.0"
    spec:
      containers:
      - name: simple-pod
        image: crsreddy1447/gol:1.0
        ports:
        - containerPort: 8080
```

```
######## ReplicaSet.yml ###########
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
```

```
    - name: php-redis                                          image: gcr.io/google_samples/gb-frontend:v3
```

#### Deployment.yml ##########

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hello-dep
 namespace: default
spec:
 replicas: 2
 strategy:
 type: RollingUpdate
 rollingUpdate:
  maxSurge: 1
  maxUnavailable: 25%
 selector:
  matchLabels:
   app: hello-dep
 template:
  metadata:
   labels:
    app: hello-dep
  spec:
   containers:
   - image: gcr.io/google-samples/hello-app:2.0
    imagePullPolicy: Always
    name: hello-dep
    ports:
    - containerPort: 8080
```

######### **HPA.yml** #############

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
 name: php-apache
spec:
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: php-apache
 minReplicas: 1
 maxReplicas: 10
 metrics:
 - type: Resource
  resource:
   name: cpu
   target:
    type: Utilization
    averageUtilization: 50
status:
 observedGeneration: 1
 lastScaleTime: <some-time>
 currentReplicas: 1
 desiredReplicas: 1
 currentMetrics:
 - type: Resource
  resource:
   name: cpu
   current:
    averageUtilization: 0
    averageValue: 0
```

****

KUBERNETES COMMANDS:

```
$ kubectl apply -f <pathofyaml>  # To execute the yaml file
$ kubectl delete -f <pathofyml>  # To delete the applied features
$ kubectl get <object-kind>
$ kubectl create -f deploy.yml   ----> Uses first time only
$ kubectl apply -f deploy.yml
$ kubectl describe deploy <app>
$ kubectl get rs  #  Details of replicaset
$ kubectl get pods # Details of POD
$ kubectl describe rs  # Full details  of Replica Set

$ kubectl get nodes -w # watch the output changes
$ kubectl get nodes -o wide # more info
$ kubectl get pods # List all the pods you created
```

$ kubectl get pods --all-namespaces   # Lists all the pods in the cluster irrespective of who created

*************Rolling Update To deployment** ******************************

$ kubectl apply -f deploy.yml –record # To record and apply the deploy
$ kubectl rollout status deployments <app name>
$ kubectl get deploy <app name>
$ kubectl rollout history deployments <app name>
$ kubectl get rs

********************UNDO Rolled Updates** ******************************

$ kubectl describe deploy <app-name>
$ kubectl rollout undo deployment <app-name> --to-revsion=1
$ kubectl get deploy
$ kubectl rollout status deployments <app name>