DevOps

① What is DevOps?

*) DevOps is a term that combines "Development (Dev) & Operations (Ops)" in the context of software development and IT operations.

*) It is set of practices, cultural philosophies, & tools aim to enhance collaboration & communication between software development (Dev) & IT Operations (Ops) teams.

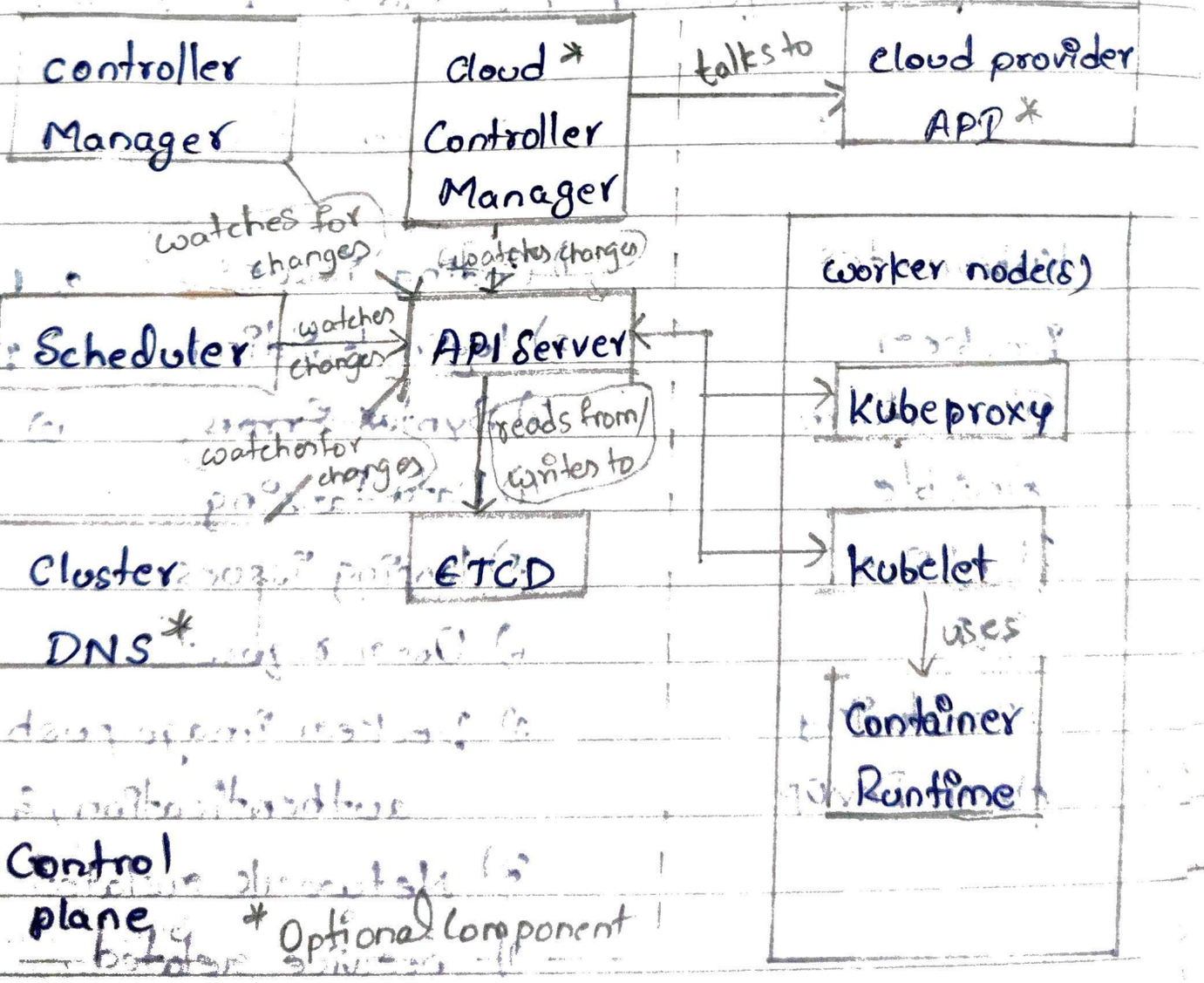
*) The goal of devops is to automate & streamline the process of building, testing, deploying, & maintaining software applications, ultimately leading to faster delivery, improved quality & reliable slw releases.

Imp Tools:	Service Connection issues:
Kubernetes	1) Coding issues
Docker	2) login Building
Terraform	3) Syntax Errors
Ansible	4) Data Parsing
Jenkins	5) looping issues
GIT	6) User request
Monitoring	7) Docker image push (connection, authentication, & authorization)
Azure DevOps	8) Network related
	9) Service related
	10) Port related
	11) Data integration
	12) Certificate related issue

Kubernetes

- Kubernetes, also known as "K8s", is an open-source container orchestration platform that automates the deployment, scaling, & management of containerized applications.
- It was originally developed by GOOGLE, but it is now maintained by the CNCF (cloud native computing Foundation). It is written in GO language
- K8S is a platform for managing containerized work

* Kubernetes Architecture:



Kubernetes Master (Control Plane):

This is the brain of the k8s cluster and manages the overall state of the system.

Master components are:

① kube-apiserver

→ The apiserver is a component of the k8s control plane that exposes the k8s API

→ It is the frontend of k8s control plane. Provides a REST API for the kubernetes control plane and handles requests from various k8s components and External Clients

→ It is tracking state of all cluster components & managing interactions blw them.

→ It receives request from kubectl, other k8s components, and external clients & process them accordingly.

② ETCD

→ It is key-value store for all cluster configuration data of the entire k8s cluster.

→ Consistent and highly available key value stove used for k8s backing store for all cluster data.

③ kube-controller manager:

→ It runs all built-in controllers, like node or replication controller

→ This component ensures that the desired state of the cluster is maintained by monitoring the state of various k8s objects and reconciling any differences

(eg: Replicasets, Deployments, Services)

→ It is responsible for managing the various controllers in the kubernetes cluster. These controllers include the Node Controller, Job controller End point slice controller & Service Account Controller

④ **kube-Scheduler**

- It distributes unscheduled workloads across the available work nodes.
- → This component assign Pods to worker node based on various factors such as resource requirements, node capacity, & affinity rules while scheduling pods.

⑤ **Cloud Controller manager** (optional)

→ Runs cloud controller process that take care of eg: Node autoscaling, creating DNS entries.

⑥ **Cluster DNS** * (optional)

→ provides in-cluster DNS for Pods & Services

Cloud Provider API * (optional)

API to manage cloud (AWS, Azure, GCP, ...) resources

Kubernetes Worker Nodes (Minions):

- These are the machines that run the application containers. Each worker node includes the following components: kubelet, Container Runtime, & kube-Proxy.

contains: kubelet, cAdvisor, services, pods & containers.

① Kubelet

→ It is an agent that runs on each node in the cluster.

→ It manages containers based on incoming Pod Specifications

→ It makesure that containers are running in a Pod. and it communicates with the API Server to receive Pod Specifications and ensures that the containers are running as expected.

② Container Runtime:

→ This is the software that runs the containerized applications (eg: Docker, containerd), which also responsible for running containers.

→ Runtime that implements the CRI, like CRI-O or containerd.

③ Kube-proxy

→ It accepts and controls network connections to the node's pods

→ It is a network proxy that runs on each node in your cluster, implementing part of the K8s Services concept.

→ Kube-proxy maintains network roles on nodes.

These network rules allow network communication to your pods from network sessions inside or outside of the cluster.

* Orchestration:

- A container Orchestration tool or framework can help you in such situations. It can help you automate all the deployment / management overhead.
- Once such Container Orchestration tool is "k8s"

* Key Components:

1) Pod:

A pod is the smallest deployable unit in k8s and represents a single instance of a running process in the cluster. A pod contains one or more containers.

2) Container:

A container is a light weight, standalone executable package that contains everything needed to run an application, including code, runtime, system, tools & libraries.

③ Service:

- A service is an abstraction that defines a service of pods and a policy for how to access them.
- Services provide a stable IP address and DNS name for a set of pods, allowing other parts of the application to access them.

④ ReplicaSet:

It ensures that a specified no. of replicas of a pod are running at all times. It takes care of autoscaling of the replicas based on demand.

⑤ Deployment:

A higher-level object that manages replicasets and provides declarative updates to the pods and ReplicaSets in the cluster.

⑥ ConfigMap:

A configuration store that holds configuration data in key-value pairs.

⑦ Secret:

A Secure way to store and manage sensitive information such as Passwords, API key and certificates

⑧ Volume:

A directory that is accessible to the containers running in a pod. Volumes can be used to store data or share files between containers.

Kubernetes Installations:

K8s has many installation options

→ Single node K8s cluster
- k3c
- kind
- Minikube

→ Multinode K8s cluster

Manual Installations:

- kube-adm ⇒
  - Install docker
  - Install CRI-dockerd
  - Install kubeadm
  - Install Flannel (Only on master)
  - Add other nodes to cluster (Only nod
- kube-spray
- KOps
- Microk8s

K8s as a Service

→ AWS ⇒ EKS (Elastic kubernetes Services)
Azure ⇒ AKS (Azure kubernetes Services)
GCP ⇒ GKE (Google kubernetes Engine)

- AWS CLI
- IAM User
- kubectl
- EKSCTL
- Install terraform

* What are the default namespaces in k8s cluster?

1) 'default' namespace
2) 'kube-system' Namespace
3) 'kube-public' Namespace
4) 'kube-node-lease' Namespace

* **K8s cluster policies:**

1. Network policies
2. Pod Security policies
3. ResourceQuotas
4. Pod Disruption Budgets
5. Image Pull policies
6. Role-Based Access Control (RBAC)

Pod Problems handles with containers

1. Initial crash : restart policy
2. Repeated crashes
3. crashloopbackoff state
4. Backoff reset

* **K8s Probe types:**

1. **Liveness Probe:**
   To check whether a container is alive and running as expected
2. **Readiness Probe:**
   To determine whether a container is ready to receive n/w traffic.
3. **StartupProbe:**
   To determine whether a container has started Successfully

* **K8s real-time challenges (K8s Problems)**

1) Multi Tenancy
2) Security
3) load Balancing
4) Observability
5) Data Stores
6) Deployment Strategies (Argo rollout)

* Container network interface (CNI):

It is a standardized way to configure networking for containers. It defines a set of APIs and plugin that allow container runtime like Docker, CRI-O, and containerd to create and manage network interface for containers.

The CNI architecture in k8s consists of 3 main components:

i) CNI Plugin
ii) CNI Configuration file
iii) CNI Runtime

CNI plugin:

⊙ It is responsible for creating and configuring n/w interfaces for containers.
⊙ There are many different CNI Plugins available, such as Flannel, Calico, & WeaveNet, which can be used to provide different types of networking capabilities such as overlay networks (or) direct host networking.

CNI configuration file:

⊙ It is a JSON file that specifies the network configuration for a particular pod (or) container.
⊙ This includes information such as the IP address, subnet, and gateway for the container.

[CNI Runtime]

- It is responsible for invoking the CNI Plugin and passing it the CNI configuration file. It is integrated with the container runtime and is responsible for creating and managing the network interfaces for containers.
- When a pod is created in K8s, the CNI Runtime is responsible for invoking the appropriate CNI plugin & passing it the CNI configuration file for the pod, & the containers are able to communicate with each other over the network.

* Kubernetes Networking Model:

⇒ K8s was built to run distributed systems over a cluster of machines.

⇒ The very nature of distributed systems makes networking a central and necessary component of K8s deployment, & understanding the K8s networking model will allow you to correctly run, monitor & troubleshoot ur applications running on K8s.

⇒ Kubernetes makes opinionated choices about how pods are networked. In particular, K8s dictates the following requirements on any network implementation

- All pods can communicate with all other pods without using NAT (Network Address Translation)
- All nodes (nodes) can communicate with all pods without NAT.

⊙ The IP that a pod see itself as is the same IP that others see it as.

⊙ Given these constraints, we are left the four distinct networking problems to solve.

① Container-to-container networking
② Pod-to-Pod networking
③ Pod-to-Service networking
④ Internet-to-Service networking

* Pod LifeCycle :-

A pod is the smallest and simplest unit of deployment, which is responsible for running one (or) more containers. The lifecycle of a pod can be divided into several phases, as follows:

① **Pending** :
When a pod is created, it enters the "Pending" Phase.

② **Running** :
Once the pod is assigned to a node & all containers have been created, it transitions into the "Running" Phase.

③ **Succeeded** :
If all the containers in a pod complete their tasks successfully & terminate, the pod transitions into the "Succeeded" Phase.

④ Failed:
If any container within the pod terminates with a failure status, the pod enters the "failure" Phase.

⑤ Unknown:
If k8s cannot determine the status of a pod for some reason, it enters the "Unknown" Phase.

⑥ Terminating:
When a pod is scheduled for deletion or if it is associated replication controller or deployment is scaled down, it enters the "Terminating" Phase.

Pod restart Policies:

⇒ Pod restart policies define how containers in a pod should be restarted in the event of a failure.
⇒ There are `3' restart policies that can be applied to a pod:

① Always
The always restart policy is the default. k8s will always restart the container when it terminates regardless of the exit status or termination.

② OnFailure:
The onfailure restart policy specifies that the container should be restarted only if it terminates with a non-zero exit code.

③ Never:

The never restart policy indicates that the container should never be restarted, regardless of the termination reason or exit code.

Pod can run 3 types of Containers:

① Containers:

This is the primary container within the pod that runs the main application (or) service. It typically encapsulates the business logic (or) functionality that the pod is designed to provide.

② Init Containers:

An init container is a special type of container that runs and completes before the application container starts.

③ Ephemeral Containers:

It is also called as "Side-car Container". A side-car container is an additional container that runs along side the application container within the same pod. Which runs supporting services isolated from the main applications.

* Container Status :-

As well as the pod overall; k8s tracks the state of each container inside the pod. You can use container lifecycle hooks to trigger events to run at certain points in a container lifecycle.

1. Waiting
2. Running
3. Terminated

* Replication Controller :-

- A replication controller is one of the earlier forms of pod replication management in k8s.
- It ensures that a specified no. of pod replicas are always running.
- It achieves this by creating (or) deleting pods based on the desired state defined in its application configuration.
- The replication controller uses a selector and a replication count to identify and maintain the correct number of Pod replicas.
- It also provides basic scaling capabilities, allowing you to scale the number of replicas up (or) down.
- However, it only supports equality-based selectors for Pod identification.

* Replica Set:

⊙ A replica set is an evolved version of the replication controller and offers more advanced features. It serves the same purpose as a Replication Controller, Ensuring a specified no. of Pod replicas are running.

⊙ Replica Set use the more flexible set-based selector to identify the pods they manage.

⊙ This allows for more complex selection criteria based on lables and labels values.

⊙ Replication Sets also support both scaling up & down, as well as scales to a specific no. of replicas.

* **Deployment:**

⇒ Deployment are primarily designed for managing stateless applications such as web servers (or) microservices. Deployment provides a declarative way to define & manage a set of replicated pods.

⇒ Deployments use a rolling update strategy by default, allowing pods to be updated one by one, minimizing downtime during updates.

⇒ Deployments generate unique labels and selectors for each, new revision, allowing rollbacks to previous versions if needed.

⇒ Scaling can be performed dynamically by changing the replica count in the deployment specification.

* **Statefull Set :-**

⇒ Statefull Sets are designed for managing stateful applications, such as databases, clustered systems, where each pod requires stable network identity & Persistant Storage

⇒ They assign unique and predictable hostnames and persistant volume claims (PVCs) to each pod, allowing Statefull applications to maintain their data and network identity during rescheduling (or) scaling.

⇒ Stateful Sets support ordered rolling updates, which means that pods are updated sequentially, one at a time, following a specified update strategy.

⇒ Scaling a StatefulSet typically involves adding (or) removing pods, and the pods retain their assigned identities & persistent storage

⇒ Stateful Set are useful for applications that require unique network identities, stable storage, ordered scaling, and ordered termination.

* [Job]

→ A Job is used to run a single or batch-like workload in kubernetes.

→ It creates one or more pods and ensures that the specified no. of pods successfully completes the assigned task before considering the Job as complete.

→ Jobs are commonly used for tasks that need to run to completion, such as data processing, batch jobs, or one-time tasks.

→ If a pod fails during execution, the job can be configured to automatically create a replacement pod, ensuring the desired no. of successfully completed tasks.

* [CronJob]

→ A cronjob is used to schedule & run jobs periodically based on a cron like schedule

→ It allows users to define a schedule using the cron syntax to specify when & how often a job should be executed.

→ Cronjobs are suitable for recurring tasks (or) Jobs that need to be executed at specific intervals, such as backups, log cleanup, or data synchronization

→ Each scheduled execution of a cronjob creates a new Job object, and the Job executes the defined task

→ Cronjobs can maintain a history of job execution,

allowing users to view & track the execution status & logs of past Job runs.

* **Daemonset:-**

- A Daemonset ensures that a copy of a pod runs on every node in the kubernetes cluster.
- It is used for deploying system-level daemons (or) agents that need to run each node such as log collectors, monitor agents, (or) network proxies
- When a new node is added to the cluster, a pod is automatically schedule and deployed on that node. When node is removed, the associated pod is terminated.
- Daemonsets provide a way to manage the lifecycle of a pod on each node without the need for manual intervention (or) scaling commands.

* **Service:**

A Service API resources are responsible for stitching your workloads together into an accessible load balanced service.

By default, workloads are only accessible within the container cluster, and they must be exposed externally using a either a load balancer (or) Node port service.

For development, internally accessible workloads can be accessed via proxy through the api-master using the kubectl proxy command.

① **ClusterIP:**

This is the default type. It exposes the service on an internal IP address reachable only within the cluster.

② **NodePort:**

This type exposes the service on a static port on each cluster node's IP address. It enables access to the service from outside the cluster.

③ **LoadBalancer:**

This type provisions a load balancer for the service and assigns it an external IP address. It is typically used in cloud environments that support external load balancers.

④ **ExternalName:**

This type maps the service to the contents of the external name field (eg: a DNS CNAME record). It does not create any endpoints but allows accessing an external service by its DNS NAME.

* **Ingress (layer 7 load balancing):**

Ingress exposes HTTP and HTTPS routes from outside the cluster to services with in the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

An Ingress may be configured to give services externally reachable URLs, loadbalance traffic, terminate SSL/TLS, and offer name-based Virtual hosting.

An Ingress controller is responsible for fulfilling the ingress usually with a load balancer, through it may also configure your edge router (or) additional frontends to help handle the traffic.

An Ingress doesnot expose Arbitraty Ports (or) Protocols. Exposing services other than HTTP & HTTPS to the Internet typically uses a service of the type service.

Type = Nodeport (or) Service Type = loadbalancer

* **Ingress Controller:**

It is responsible for implementing the rules defined in the Ingress resource.

It run as a separate component within the cluster and typically utilizes a loadbalancer (or) reverse proxy to manage incoming traffic.

* Ingress Class :-

It is an API object introduced in K8S 1.18 to provide a way to associate an ingress controller with specific ingress resources.

It allows for more flexible management of multiple ingress controllers within a cluster.

Ingress class was introduced, the associated b/w an ingress resource and an ingress controller was made implicitly based on annotations.

Each ingress controller would define its own set of annotations that needed to be added to the ingress resources to indicate which controller should handle it.

This approach had limitations when multiple ingress controllers were used as it required modifying the ingress resources of each controller.

* **Kubernetes Storage:**

To persist the data in the Read/Write layer, docker has volumes. K8s support volumes to persist the data. The types of volumes which are supported by K8s are:

① Volume:

→ This gives volume with the help of mnt namespace to container. Volume life cycle is equivalent to lifecycle of pod.

② Ephemeral Volume:

→ This is also temporary volume used for containers where they require any persistent storage across pod restarts/ creations.

③ Persistent Volume:

→ It is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using storage classes.

→ It is a resource in the cluster just like a node in a cluster resource. PVs are volume plugins like volumes, but have a lifecycle independent of any individual Pod that uses the PV.

→ This API object captures the details of the implementation of the storage, be that NFS, ISCSI, or a cloud-provider specific storage system.

* Persistant Volume Claim (PVC):
→ It is a request for storage by a user. It is similar to a pod. Pod consume node resources and PVCs consume PV resources.
→ Pods can request specific level of resources (CPU and memory). Claims can request specific size and access modes (eg: They can be mounted ReadWrite once, Read Only many, or Read write many see Access Modes).

* Overlay:-
⇒ It is commonly used in the context of container networking. K8s provides a built-in networking model called CNI, which allows different networking plugins to implement networking for containerized applications.
⇒ Overlay networking works in K8s:
① Pod-to-Pod Communication
② Overlay Network Configuration
③ Encapsulation & Tunneling
④ Networking Plugins.

* ConfigMap :-

→ ConfigMaps are used to store non-sensitive configuration data that can be shared among containers & pods.

→ They are created as K8s objects and can be defined using YAML (or) JSON.

→ ConfigMaps can store data as key-value pairs (or) as plain text.

→ Typical uses of configmaps include environment variables, Command-line arguments, configuration files, or any other kind of non-sensitive configuration data.

→ Configmaps are accessible by all pods in a Namespace and can be mounted as volumes (or) exposed as environment variables.

* Secret :-

- Secrets are used to store sensitive information, such as passwords, API keys, TLS certificate, or any other confidential data.
- Similar to ConfigMaps, secrets are created as K8S objects and can be defined using YAML or JSON.
- Secrets store data as key-value pairs, but the values are base 64 encoded by default to provide same level of encoding.

* Horizontal Pod Autoscaler (HPA):-

- HPA automatically adjusts the no. of pod replicas based on the observed CPU utilization or custom metrics
- It scales the no. of replicas of a deployment, replicaset (or) Statefulset to meet the desired utilization target.
- HPA can be configured with minimum & maximum replica limits to ensure the scalability and stability of the application.
- HPA periodically collects CPU utilization metrics from the pods & compares them against the defined target CPU utilization.

* Vertical Pod Autoscaler (VPA):

- VPA adjusts the resource requests & limits of individual pods based on their actual resource usage
- It collects resource utilization metrics from pods analyzes historical usage patterns, & suggests (or) automatically adjusts the resource requests & limits for each pod.
- VPA can adjust CPU & Memory resource requests & limits dynamically optimizing resource allocation for each pod.

* Taints:

→ A taint is a property assigned to a node that indicates a restriction or perference

→ A taint consists of a key-value pair & an effect. The effect can be one of three options:

① No schedule
② Prefer No schedule
③ No Execute

→ Taints are typically applied to nodes using the 'kubectl taint' command (or) by specifying them in the node's configuration.

* Tolerations:

→ A toleration is a setting defined in a pod's specification that allows it to tolerate nodes with specific taints

→ Tolerations are used to specify which taints a pod can tolerate, enabling it to be scheduled on nodes with those taints

→ A toleration consists of a key-value and an operator. The operators can be one of three options:

① Exists
② Equal
③ Does Not Exist

* **Labels:**

→ labels are key-value pairs attached to k8s resources such as pods, services, deployments, (or) nodes.

→ labels are typically used to organize resources, facilitate grouping, and enable efficient querying & filtering.

→ Multiple labels can be assigned to a resource, allowing for flexible classification & grouping of resources.

→ labels are used for various purpose, including selecting resources for deployment, defining service endpoints, or applying policies and annotations.

* **Selectors:-**

→ Selectors are used to define criteria for selecting resources based on their labels.

→ In k8s, selectors are associated with certain resources, types, such as services, deployments, replicaset, (or) ingress rules.

→ Selectors follow a specific syntax, usually expressed as a label selector, to match & filter resources based on label values.

→ Selectors are used in various k8s operations, such as deploying applicated to specific nodes or pods, exposing services to a subset of resources, (or) scaling replicas.

* **Node Selector:-**

A node selector specifies the simplest way to constrain Pods es with specific labels.

* Limits :-

→ Limits refer to the maximum amount of resources that a container can utilize.

→ They are defined at the container level and restrict the container's resource consumption.

→ Limits are typically set for CPU and memory resources but can also be applied to other resources like storage or network bandwidth.

→ When a container exceeds its specified limits, k8s takes action to control (or) throttle the container's resource usage.

→ Limits ensure resource fairness and stability within the k8s cluster.

* Requests :-

→ Requests specify the minimum amount of resources that a container requires to run effectively.

→ They are defined at the container level and indicate the container's resource demands.

→ Requests are typically set for CPU & Memory resources but can also be specified for other resources.

→ Requests aid in resource scheduling & capacity planning within the k8s cluster.

Replication Controller	Replica Set
① It is an older concept in K8s & has been around since the early version.	① It is the successor to replication controller & was introduced in K8s 1.2.
② It supports basic equality-based selectors for identifying the pods in manages.	② It supports both equality-based & set-based selector requirements
③ It doesn't have a built-in mechanism for managing rolling updates & rollbacks	③ Deployments provide a high-level abstraction that manages the replicaset & provides additional features like version rolling updates, rollbacks, scaling & version.
④ It is suitable for basic replication needs where you need to maintain specified no of replicas of your pods.	④ It is designed to be more flexible and powerful than Replication controllers.

Replication Controller	Replica Set
① It is an older concept in K8s & has been around since the early version.	① It is the successor to replication controller & was introduced in K8s 1.2.
② It supports basic equality-based selectors for identifying the pods in manages.	② It supports both equality- & set-based selector requirements
③ It doesn't have a built-in mechanism for managing rolling updates & rollbacks	③ Deployments provide a high-level abstraction that manages the replicaset & provides additional features like versioning, rolling updates, rollbacks, scaling & version.
④ It is suitable for basic replication needs where you need to maintain specified no. of replicas of your pods.	④ It is designed to be more flexible and powerful than Replication controllers.

Replicaset

① It ensures that a specified number of pod replicas are running in a cluster

② It doesn't have built in update strategies. It only ensures that the desired no. of replicas is maintained

③ You can manually scale the no. of replicas controlled by a replicaset using the 'kubectl scale'

Deployment

① It is a higher-level abstraction that manages Replicasets.

② It supports rolling updates, which allow you to update pod in a controlled manner. It supports roll backs to previous versions if issues are encountered

③ Similar to replicaset. It allows auto scale the no. of replicas

Statefulset

① It is used for managing stateful app where each pod instance has a persistant state.

② It provide guarantees for the ordering & uniqueness of pod creation & deletion, ensuring state r/w identifies & persistent storage for each pod.

③ It doesn't support rolling backs & only support rolling updates.

Deployment

① It is used for managing stateless applications or service that do not require persistent state

② Deployments are primarily focused on managing the replication & scaling of pods.

③ It provide features like rolling updates & rollbacks, making it easy to deploy new versions of applications.

ConfigMap

① ConfigMaps are used to store non-sensitive data, such as environment variables, configuration files & CLI arguments.

② Config map stores data as key-value pairs the values can be plain text.

③ ConfigMap data is stored in clear text.

Secret

① Secrets are used to store sensitive data such as passwords, API keys, TLS certificates, or any other confidential info.

② Secreet also stores data as key value pairs, but the values encoded or encrypted to maintain confidentiality.

③ Secrets are base 64-encoded by default.

Horizantal Pod Autoscaler(HPA)

① HPA scales the number of pod replicas horizantally.

②

② HPA adds or removes pod replicas to meet the desired utilization target

③ HPA is useful in scenarios wheres the workload can be parallelized & additional replicas can handle increased demand.

Vertical Pod Autoscaler(VPA)

① VPA scales individual pod resource requests & limits vertically

② VPA adjusts the resource requests & limits of individual pods.

③ VPA is useful when the work-load performance can be improved by adjusting the resource allocation within the pod.

* K8s Issues :-

1) Persistent Storage :

Managing persistent storage for stateful applications in K8s can be challenging. Configuring storage classes, persistent volumes(PVs), persistent volume claims(PVCs), as well as handling data, replication, backups, & disaster recovery, can pose difficulties.

2) Cluster upgrades and maintainance :

Upgrading and maintaining a kubernetes cluster can be complex, especially when dealing with production environments. Ensuring compatibility performing rolling upgrades, managing cluster add-ons and handling workloads migrations require careful planning and execution.

3) Monitoring & logging :-

Monitoring & troubleshooting K8s clusters can be complex, especially when dealing with distributed microservices architectures collecting & analyzing metrics, logs, & events from multiple pods & nodes, and ensures proper observability and alerting can be challenging.

4) Clusterscale & Performance :

As the size of a K8s cluster grows, managing scale & ensuring optimal performance can become complex. Cluster scalability load balancing, resource allocation, & optimization can pose challenges, especially when running resource-intensive (or) highly distributed applications.

Update EKS cluster:

i) aws eks update-kubeconfig --name <clustername>

ii) aws eks update-cluster-version --name <clustername>
--kubernetes-version <new-version>

Update EKS Nodes:

① **Drain Nodes:**

kubectl drain <node-name> -ignore-daemonsets

② **Terminate & Replace Nodes:**

aws autoscaling terminate-instance-in-auto-scaling-group
--instanceid <instance-id> -no-should-decrement-desired-capacity

③ **Uncordon Nodes:**

kubectl uncordon <node-name>

④ **Verify Node update:**

kubectl get Nodes

Change ConfigMap data:-

i) kubectl get configmaps

ii) kubectl patch configmap <configmap-name>
-p '{"data":{"key1":"newvalue1","key2":"newvalue2"}}'

iii) kubectl describe configmap <configmap-name>

* Kustomization K8s :-

Kustomization is nothing but a package manager of K8's. kustomize is a k8s configuration transformation tool that enables you to customize untemplated YAML files, leaving the original files untouched.

Kustomization is a configuration file used by kustomize, which is a built-in tool for customizing kubernetes applications. kustomize allows you to modify & manage kubernetes resources without directly modifying the original resource files.

It follows the principles of declarative configuration & Separation of concerns.

kubectl apply -k <.>

① Base :-

The base directory contains the original kubernetes resource files that service as the starting point for customization. It can include deployment service, config map, or any other valid K8s resources.

② Overlays :-

Overlays are directories that contains customization files that modify or extend the base resources. Multiple overlays can be applied to the same base, allowing you to create different configuration for different environments. Each overlays can selectively add, modify, or remove resources.

③ **Patches:**

patches are files that define to be applied to specific resources. There are different types of patches available, such as → adding annotations, changing labels, → modifying container images, (or) → updating configuration data.

④ **Resources:**

The kustomization file specifies the resources to include in the final configuration.

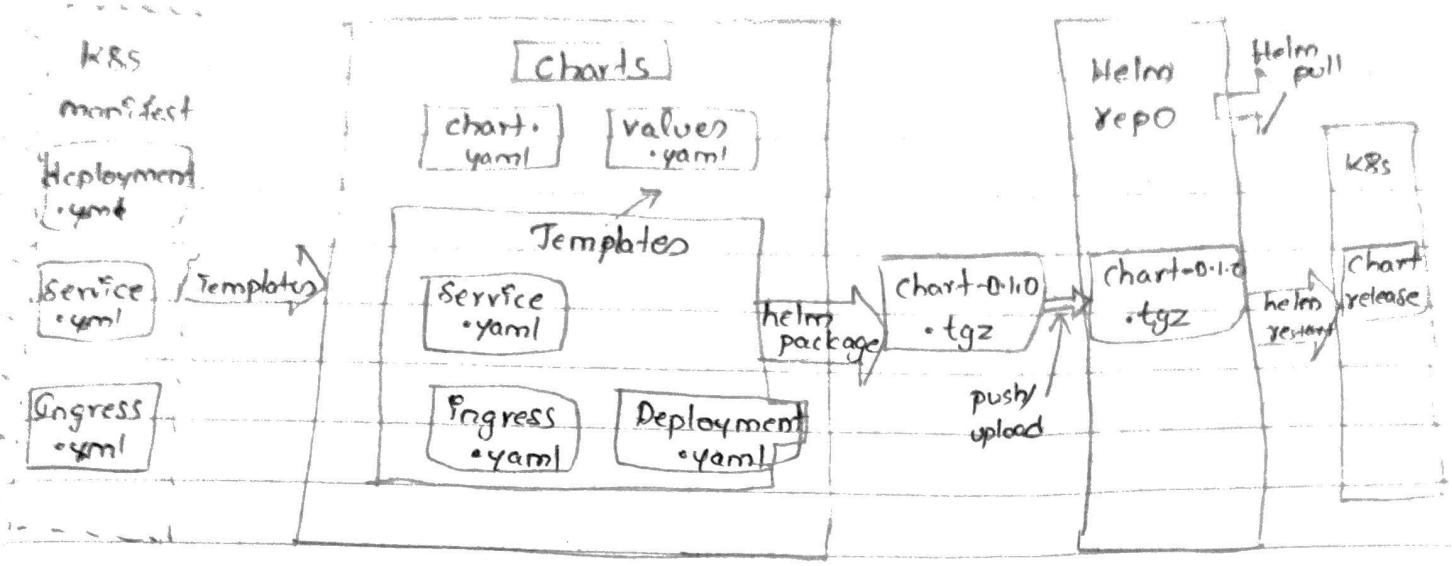
* **Helm Charts :-**

⇒ Helm charts are nothing but a package manager for k8s.

⇒ we have to combine all the YAML files (deployment.yml, service.yml, & ingress.yml) and then deploy.

* `helm search <chart-name>` In this we have plenty of charts for reference.

* `helm install <chart-name>`
* `helm upgrade <chart-name>`
* `helm rollback <chart-name>`



* Status Code :-

① 1XX ⇒ Information (It indicates that the inital part of the client's request has been received and understood by the server)

② 2XX ⇒ Success (The Server returned the requested resource)

③ 3XX ⇒ Redirection (This indicates that the requested resource has multiple representations available, & the client should choose one)

④ 4XX ⇒ Client Errors (or) Bad Request
(The server could not understand the request due to invalid syntax or other client errors)

⑤ 5XX ⇒ Server Side errors (The server encountered an unexpected error while processing the request)

* "CrashloopBackOff" error:

It is a common error message you might encounter in K8s when a container in a pod repeatedly crashes immediately after being restarted.

This error indicates that the pod is stuck in a restart loop, resulting in a continuous cycle of container crashes & restart attempts.

The "CrashloopBackOff" error can occur due to various reasons including misconfiguration, resource limitations, dependency issues, (or) application errors.

① Check the pod logs:
`kubectl logs <pod-name> -c <container-name>`
② Verify resource allocation
③ Validate dependencies
④ Confirm container readiness
⑤ Test container locally
⑥ Review kubernetes events
`kubectl describe pod <Pod-name>`
⑦ Update application or container configuration

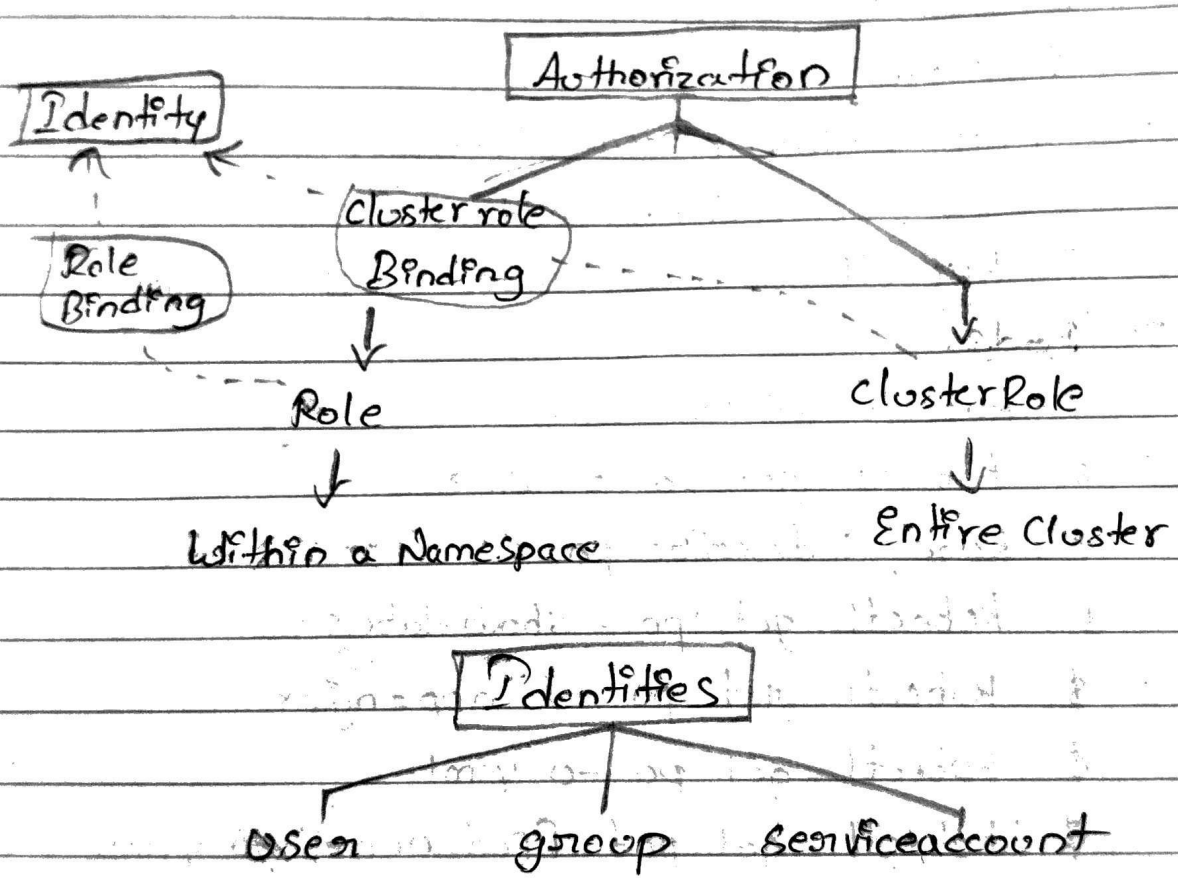
To investigate → 'CrashloopBackoff' issue

i) check logs → kubectl logs <name of pod>
ii) Inspect events → kubectl describe pod <podname>
iii) Review configuration
iv) check resource limits
v) Debug application

* [RBAC (Role Based Access Control)]-

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

RBAC authorization uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure policies through the k8s API



* kubernetes Commands:

Viewing Resource Information:

-> Nodes

$ kubectl get no

$ kubectl get no -o wide

$ kubectl describe no

$ kubectl get no -o yaml

$ kubectl get node --selector=[label_name]

$ kubectl get node -o
jsonpath='{.items[*].status.addresses[?(@.type
=="ExtenalIP")].address}'

$ kubectl top node [node_name]

⇒ Pods

$ kubectl get po

$ kubectl get po -o wide

$ kubectl describe po

$ kubectl get po --show-labels

$ kubectl get po -l app=nginx

$ kubectl get po -o yaml

$ kubectl get pod [pod_name] -o yaml --export

$ kubectl get pod [pod_name] -o yaml --export
> nameoffile.yaml

$ kubectl get pods --field-selector status.phase=
Running

⇒ **Namespaces**

```
$ kubectl get ns
$ kubectl get ns -o yaml
$ kubectl describe ns
```

⇒ **Deployments**

```
$ kubectl get deploy
$ kubectl describe deploy
$ kubectl get deploy -o wide
$ kubectl get deploy -o yaml
```

⇒ **Services**

```
$ kubectl get svc
$ kubectl describe svc
$ kubectl get svc -o wide
$ kubectl get svc -o yaml
$ kubectl get svc --show-labels
```

⇒ **DaemonSets**

```
$ kubectl get ds
$ kubectl get ds --all-namespaces
$ kubectl describe ds [daemonset_name] -n [namespace_name]
$ kubectl get ds [ds_name] -n [ns_name] -o yaml
```

⇒ **Events**

```
$ kubectl get events
$ kubectl get events -n kube-system
$ kubectl get events -w
```

Logs

```
$ kubectl logs [pod_name]
$ kubectl logs --since=1h [pod_name]
$ kubectl logs --tail=20 [pod_name]
$ kubectl logs -f -c [container_name] [pod-name]
$ kubeitl logs [pod_name] > pod.log
```

Service Accounts

```
$ kubectl get sa
$ kubectl get sa -o yaml
$ kubectl get serviceaccounts default -o yaml >
    ./sa.yaml
$ kubectl replace serviceaccount default -f
    ./sa.yaml
```

Replicasets

```
$ kubectl get rs
$ kubectl describe rs
$ kubectl get rs -o wide
$ kubectl get rs -o yaml
```

Roles

```
$ kubectl get roles --all-namespaces
$ kubectl get roles --all-namespaces -o yaml
```

Secrets

```
$ kubectl get secrets
$ kubectl get secrets --all-namspaces
$ kubectl get secrets -o yaml
```

ConfigMaps

```
$ kubectl get cm
$ kubectl get cm --all-namespaces
$ kubectl get cm --all-namespaces -o yaml
```

Ingress

```
$ kubectl get ing
$ kubectl get ing --all-namespaces
```

PersistentVolume

```
$ kubectl get pv
$ kubectl describe pv
```

PersistentVolumeClaim

```
$ kubectl get pvc
$ kubectl describe pvc
```

StorageClass

```
$ kubectl get sc
$ kubectl get sc -o yaml
```

Multiple Resources

```
$ kubectl get svc,po
$ kubectl get deploy,po
$ kubectl get all
$ kubectl get all --all-namespaces
```

Changing Imperative Attributes :-

- **Taints**
  $ kubectl taint [node_name] [taint_name]
- **Labels**
  $ kubectl label [node_name] disktype=ssd
  $ kubectl label [pod_name] env=pod
- **Cordon / Uncordon**
  $ kubectl cordon [node_name]
  $ kubectl uncordon [node_name]
- **Drain**
  $ kubectl drain [node_name]
- **Nodes / Pods**
  $ kubectl delete node [node_name]
  $ kubectl delete pod [pod_name]
  $ kubectl edit node [node_name]
  $ kubectl edit pod [pod_name]
- **Deployments / Namespaces**
  $ kubectl edit deploy [deploy_name]
  $ kubectl delete deploy [deploy_name]
  $ kubectl expose deploy [deploy_name] --port=80 --type=NodePort
  $ kubectl scale deploy [deploy_name] --replicas=5
  $ delete kubectl delete ns

$ kubectl edit ns [ns_name]

- [Services]

$ kubectl edit svc [svc_name]
$ kubectl delete svc [svc_name]

=> [DaemonSets]

$ kubectl edit ds [ds_name] -n kube-system
$ kubectl delete ds [ds_name]

=> [ServiceAccounts]

$ kubectl edit sa [sa_name]
$ kubectl delete sa [sa_name]

=> [Annotate]

$ kubectl annotate po [pod_name] [annotation]
$ kubectl annotate no [node_name]

③ Requests :-

=> [API Call]

$ kubectl get --raw /apis/metrics.k8s.io/

=> [Cluster Info]

$ kubectl config
$ kubectl cluster-info
$ kubectl get componentstatuses

6. Adding Resources :-

=> Creating a pod

```
$ kubectl create -f [name_of_file]
$ kubectl apply -f [name_of_file]
$ kubectl run [pod_name] --image=nginx
    --restart=Never
$ kubectl run [pod_name] --image=nginx --restart=Never
$ kubectl run [pod_name] --generator=run-pod/v1
    --image=nginx
```

=> Creating a Service

```
$ kubectl create svc nodeport [svc_name]
    --tcp=8080:80
```

=> Creating a Deployment

```
$ kubectl create -f [name_of_file]
$ kubectl apply -f [name_of_file]
$ kubectl create deploy [deploy-name]
    --image=nginx
```

=> Interactive Pod

```
$ kubectl run [pod_name] --image=busybox
    --rm -it --restart=Never --sh
```

⇒ **Output YAML to a file**

```
$ kubectl create deploy [deploy_name] --image=nginx
    --dry-run -o yaml > deploy.yaml
$ kubectl get po [pod_name] -o yaml --export
    > pod.yaml
```

⇒ **Getting Help**

```
$ kubectl -h
$ kubectl create -h
$ kubectl run -h
$ kubectl explain deploy.spec
```

kubernetes

① apiversions

container: v1

configb: batch/v1

Daemonset: apps/v1

Deployment: apps/v1

Job: batch/v1

Replicaset: apps/v1

Replication controller: v1

Statefullset: apps/v1

Service: v1

Ingress: networking.k8s.io/v1

① Pod

apiversion: v1

kind: pod

metadata:

name: activity

Spec:

containers:

-image: apline

command:

-sleep

args:

-1d

name: pod

② Replicaset

```
---
apiversion : app/v1
Kind : Replicabet
Metadata :
   name : nginx-rs
   labels :
      app : nginx
Spec :
   minReady Seconds : 3
   replicas : 4
   Selector :
     match
       -key : app
        Operator : In
        values :
          - nginx
          - webservers
   template :
     metadata :
        name : nginx-pod
        labels :
          app : nginx
          ver : "1.23"
```

```
Spec :
  containers :
   - name : nginx
     image : nginx : 1.23
     Ports :
       -containerport : 80
       protocol : TCP
```

Ingress

```
apiVersion : networking.k8s.io/v1
kind : Ingress
metadata :
    name : my-ingress
    annotations :
        nginx.ingress.kubernetes.io/rewrite-target: /
spec :
  rules :
    - host : myappexample.com
      http :
        paths :
          - path : /FOO
            path type : prefix
            backend :
              service :
                name: myapp-foo
                port :
          - path : /bar
            pathtype : prefix
            backend :
              Service :
                name :
                port :
---
```

④ Service

```
---
apiversion: v1
Kind : service
metadata :
   name : nginx
Spec :
  type : clusterIP
  selector :
     app : nginx
     Ver : "1.0"
  Ports :
     - name : webport
       port : 35000
       Targetport : 80
```