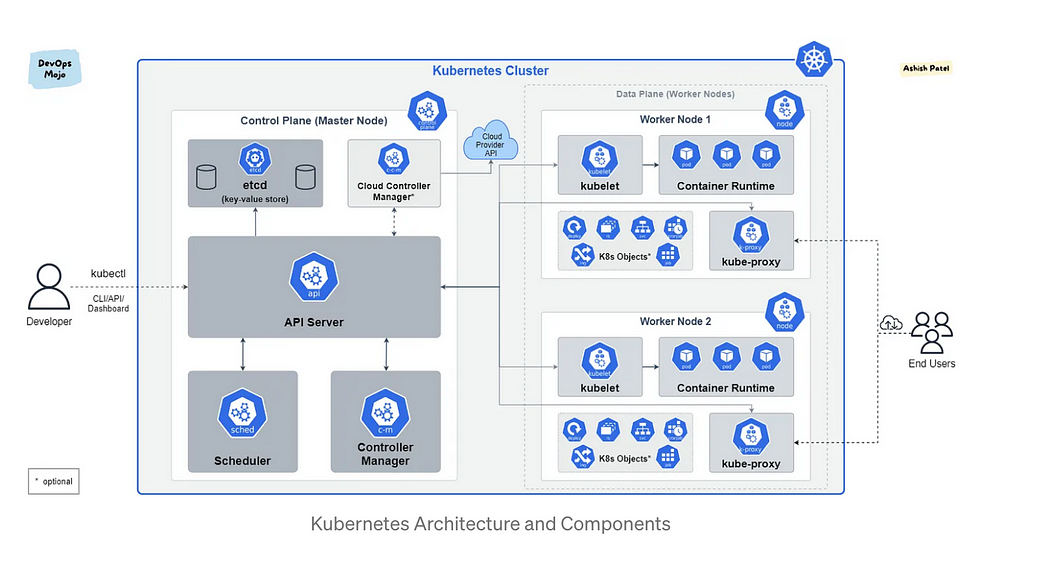
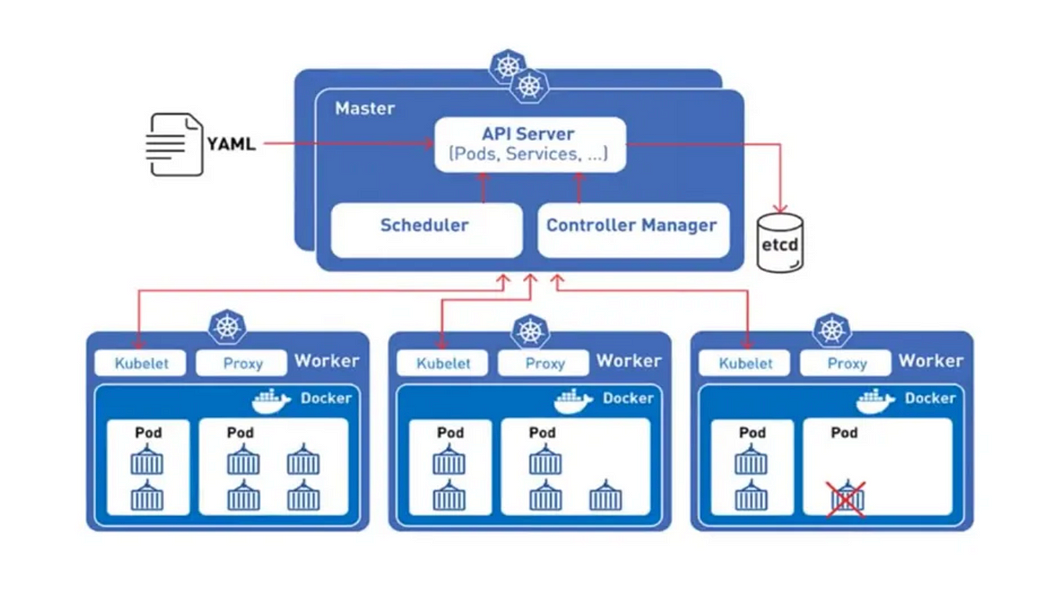
Kubernetes Notes

# **Kubernetes Architecture**

* Kubernetes is a powerful container management tool that automates the deployment and management of containers. It can run and coordinate containerized applications across a cluster of machines. It is a platform designed to completely manage the life cycle of containerized applications and services using methods that provide predictability, scalability, and high availability. Kubernetes can define how your applications should run and how application interact with other applications or the outside world.





* Kubernetes deployment is called a cluster. A Kubernetes cluster consists of at least one main (control) plane, and one or more worker machines, called nodes. Both the control planes and node instances can be physical devices, virtual machines, or instances in the cloud.

**What are containers?**

* Stuff like your application code, dependent libraries, and its dependencies all the way up to the kernel — is a container. The key concept here is isolation. Isolate all your stuff from the rest so that you have better control of them. There are three types of isolation provided by containers
* Workspace isolation (Process, Network)
* Resource isolation (CPU, Memory)
* File system isolation (Union File System)

# What features Does K8s Offer?

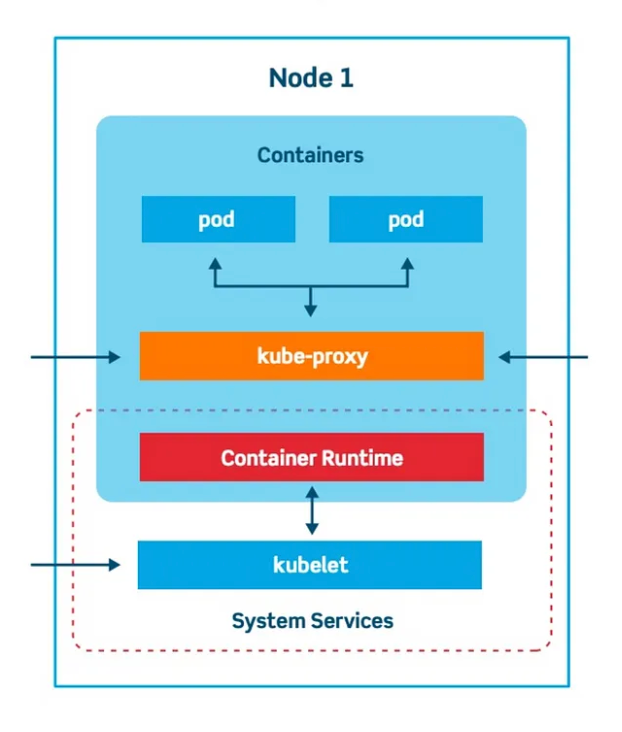
* **Service discovery and load balancing**  
  Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
* **Scaling**  
  Auto-scaling: Automatically change the number of running containers, based on CPU utilisation or other application-provided metrics.
* **Self-healing**  
  Kubernetes restarts containers that fail, replaces containers, kills containers that don’t respond to your user-defined health check, and doesn’t advertise them to clients until they are ready to serve.
* **Storage orchestration**  
  Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
* **Automated rollouts and rollbacks**  
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
* **Automatic bin packing**  
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
* **Secret and configuration management**  
  Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
* Assures **high availability** with zero downtime
* Reliable infrastructure to support **data recovery** with ease

**Why you need Kubernetes?**

* Containers are a good way to bundle and run your applications. They provide you with an easy way to package and deploy services, allow for process isolation, immutability, efficient resource utilization, and are lightweight in creation.
* Managing containers for production is challenging:  
  - You need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start.  
  - You can end up with dozens, even thousands of containers over time. These containers need to be deployed, managed and updated.
* It will be very difficult to do all things manually, right? Kubernetes is the solution! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and fail-over for your application, provides deployment patterns, and more.
* Package up the app and let something else manage it for us
* Not worry about the containers management
* App Containers can’t live with single server deployment
* Inter-host communication of containers
* Deploying and updating software at scale
* Better management through modularity
* Auto healing
* Logging and Monitoring

# Worker Node

* Every worker as shown in picture below has these 3 key processes:
* **Container Runtime**
* **kubelet**
* **kube-proxy**



Worker Node

**kubelet**

* **kubelet** is a primary node-agent of the worker node, which interacts with both node and the container in the given worker node
* The kubelet is responsible for
* Maintaining a set of pods, which are composed of one or more containers, on a local system.
* For registering a node with a Kubernetes cluster, sending events and pod status, and reporting resource utilization.
* Within a Kubernetes cluster, the kubelet watches for PodSpecs via the Kubernetes API server.
* A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the API server) and ensures that the containers described in those PodSpecs are running and healthy.

**Kube-proxy:**

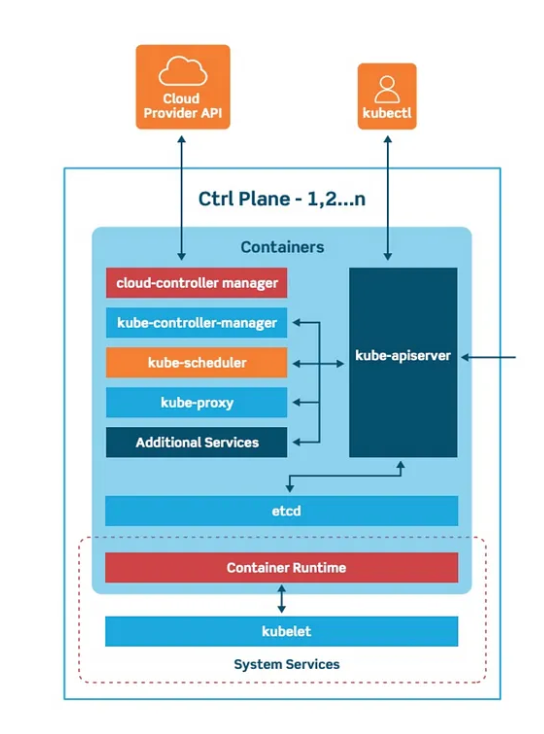
* K8s cluster can have multiple worker nodes and each node has multiple pods running, so if one has to access this pod, they can do so via Kube-proxy. In order to access the pod via k8s services, there are certain network policies, that allow network communication to your Pods from network sessions inside or outside of your cluster. These rules are handled via kube-proxy**.***kube-proxy has an intelligent algorithm to forward network traffics required for pod access which minimizes the overhead and makes service communication more performant*

**Container runtime**

* Every Microservice module(micro-app) you deploy is packaged into a single pod that has its own container runtime. One needs to install a container runtime into each worker node in the cluster so that Pods can run there.
* **Some of the container runtime examples are,**
* containerd
* CRI-O
* Docker
* The container runtime is the software that is responsible for running containers (in Pods).
* To run the containers, each worker node has a container runtime engine.
* It pulls images from a container image registry and starts and stops containers.

# Master Node

* *Who manages these worker nodes, to ensure that they are always up and running? How does the K8s cluster know which pods should be scheduled and which one should be dropped or restarted? How does the k8s cluster know the resource level requirements of each container app?*— — — Well the answer lies in the concept of Master Node — -



Master Node

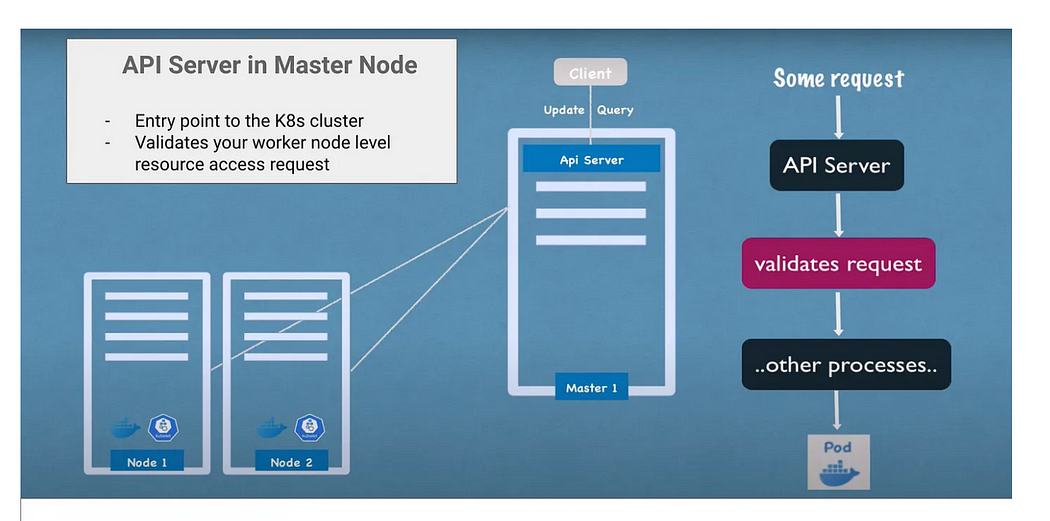
* The **master node** is also known as a **control plane** that is responsible to manage worker/slave nodes efficiently. They interact with the worker node to
* Schedule the pods
* Monitor the worker nodes/Pods
* Start/restart the pods
* Manage the new worker nodes joining the cluster

**Master Node Processes:**

* The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability. You can scale up and scale down cluster by adding and removing nodes.
* Every master nodes in the K8s cluster runs the following key processes
* **kube-apiserver**
* **kubectl: kube-controller-manager**
* **kube-scheduler**
* **etcd**

**kube-apiserver:**

* It is the main gateway to access the k8s cluster and act as the main gatekeeper for client level authentication or we can say that the kube-apiserver is the front end for the Kubernetes control plane.



kube-api server in master node

* *So whenever you want to*
* Deploy any new app
* Schedule any pods or
* Create any new services
* Query status or the health of your worker nodes
* You need to make a request to the API server of the master node which in turn validates your requests before you get access to the processes in worker nodes. kube-apiserver is designed to scale horizontally — that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances

**kube-scheduler in K8s Master Node:**

* It schedules pods to worker nodes..Every time as a K8s admin/developer if you want to schedule a new pod on the worker node, you need to send the request to the master API server which in turn will make a call to the kube-scheduler process. The scheduler here will intelligently decide on which worker node this pod should be placed. This is a key Control plane component that keep a watch for newly created Pods with no assigned worker node, and selects a worker node for them to be scheduled and run on.  
  This decision to which worker node the newly created pod should be accommodated in based on the resource level availability of each node. Scheduler does the resource level query and makes critical scheduling decisions. The real execution of the scheduler level decision is done by the kubelet process in the given worker node.
* **Factors taken into account for scheduling decisions include:**
* Individual and collective resource requirements
* Hardware/Software/Policy constraints
* Node affinity and anti-affinity specifications,
* Data locality, inter-workload interference, and deadlines.

**kube-controller-manager(Kubectl):**

* It is one of the critical processes in a master node that monitors the status of any worker node level failures. It keeps a close watch over the event like  
  :: Crashing of any pods in the worker node  
  and, requests the scheduler to restart or reschedule any dead /failed pods, after detecting such event.
* These control manager component of master control planer has following types of controllers:
* **Node controller**: Responsible to respond when any worker node goes down
* **Replication controller**: It ensures that the request to maintain the correct replica count of any pod deployment is always taken care
* **Endpoints controller**: Populates the Endpoints object viz. Joins, Services & Pods
* **Service Account & Token controllers**: Create default accounts and API access tokens for new namespaces created in the worker node.
* It runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

**etcd**

* etcd*in the master control plane is responsible to store every kind of cluster-level change in the form of key-value pairs.*It is a consistent, distributed, and highly-available key value store. It can be part of the control plane, or, it can be configured externally.
* It can be easily seen as a brain of the k8s cluster which keeps the log of every minute details of changes occurring in the cluster.
* For example, if any pod crashes in the worker node and it has to be rescheduled, the same gets stored in **etcd** as key-value pair, also the event of pod rescheduling on the node is also logged here.

**Cloud Controller Manager**

* The cloud-controller-manager is a Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider’s API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.
* By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.
* The cloud-controller-manager is structured using a plugin mechanism that allows different cloud providers to integrate their platforms with Kubernetes.

**kubelet and kube-proxy difference**

**kubelet**

* It is an agent that runs on each node in the cluster.
* It acts as a conduit between the API server and the node.
* It makes sure that containers are running in a Pod and they are healthy.
* It instantiates and executes Pods.
* It watches API Server for work tasks.
* It gets instructions from master and reports back to Masters.

**kube-proxy**

* It is networking component that plays vital role in networking.
* It manages IP translation and routing.
* It is a network proxy that runs on each node in cluster.
* It maintains network rules on nodes. These network rules allow network communication to Pods from inside or outside of cluster.
* It ensure each Pod gets unique IP address.
* It makes possible that all containers in a pod share a single IP.
* It facilitating Kubernetes networking services and load-balancing across all pods in a service.
* It deals with individual host sub-netting and ensure that the services are available to external parties.

# Kubernetes Objects

* *Kubernetes Objects* are persistent entities in the Kubernetes system that represent state of your cluster. By creating an object, you’re effectively telling the Kubernetes system what you want your cluster’s workload to look like; this is your cluster’s *desired state*.
* Kubernetes objects can be distinguished between workload-oriented objects that are used for handling container workloads and infrastructure-oriented objects, that for example handle configuration, networking and security. Some of these objects can be put into a namespace, while others are available across the whole cluster.
* we can describe these objects in the popular data-serialization language YAML and send them to the api-server, where they get validated before they are created.



Kubernetes Objects

**K8s components :**

* **Pods**: Smallest unit of k8s, which is an abstraction of the container application. Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
* **Nodes:** The machines on Kubernetes. They can be either a physical or a virtual machine. Pods are scheduled and orchestrated to run on nodes.
* **Deployment** provides declarative updates for Pods and ReplicaSets. You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
* **ReplicaSet’s** purpose is to maintain a stable set of replica Pods (Identical Pods) running at any given time.
* **Services & Ingress:**To manage external communication between nodes and internal pod level communication
* **ConfigMaps**: To manage the end-point URLs required by the pods/ DB’s
* **Secrets**[:](https://services.azuga.com/metadata/azuga-metadata/v3/metadata/drivers) To keep app-level passwords and secret keys securely using based64 encoding
* **Statefulsets**: StatefulSet is used to manage stateful applications with persistent storage. Storage stays associated with replacement pods. Volumes persist when pods are deleted. Pods are deployed/updated individually and in order. Each Pod has own storage. A StatefulSet maintains a sticky identity for each of their Pods.
* **DaemonSet** ensures that all (or some, matching a node selector) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.  
  Some typical uses of a DaemonSet are:  
  - Running a cluster storage daemon on every node.  
  - Running a logs collection daemon on every node.  
  - Running a node monitoring daemon on every node.
* **Job** runs pods that perform a completable task. Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.
* **CronJob** creates Jobs on a repeating schedule. CronJobs are meant for performing regular scheduled actions such as backups, report generation, and so on.
* **Namespaces** provides a mechanism for isolating groups of resources within a single cluster.Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, PersistentVolumes, etc)*.

**Service**

* Service is used to expose an application deployed on a set of pods using a single endpoint. i.e. It maps a fixed IP address to a logical group of pods.
* Service provides stable networking for pods (ephemeral pods) by bringing stable IP addresses and DNS names, and provide a way to Kubernetes to configuring a proxy to forward traffic to a set of pods.
* Service enables communication between nodes, pods, and users of app, both internal and external, to the cluster. Service also provides load balancing when you have Pod replicas.
* There are four types of Kubernetes services — ClusterIP, NodePort, LoadBalancer and ExternalName.

**Ingress**

* Ingress manages external access to the services in a cluster, typically HTTP/S.
* Ingress may provide load balancing, SSL termination and name-based virtual hosting.
* Ingress exposes one or more services to external clients through a single externally reachable IP address.
* Traffic routing is controlled by rules defined on the Ingress resource.
* Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer.

**Network Policy (NetPol)**

* Network Policy isolates the network between pods by specifying which pods can connect to each other, and other network endpoints.
* Network Policy works on OSI Layer 3 and Layer 4.
* Network policies are implemented by the [network plugin](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/).

**Endpoint**

* Endpoint defines which pods (or other servers) are exposed through a service.
* For Kubernetes *Services*, Kubernetes creates a *Endpoint* object. This endpoint will have the ip address mapping of the pods. This is created automatically for services with a defined selector.
* Endpoints can also be used to connect to external services like they were internal to the kubernetes cluster.

**EndpointSlice**

* EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster.
* EndpointSlices offer a more scalable and extensible alternative to Endpoints.
* EndpointSlice contains references to a set of network endpoints.

**EndpointSlice**

* EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster.
* EndpointSlices offer a more scalable and extensible alternative to Endpoints.
* EndpointSlice contains references to a set of network endpoints

**Persistent Volumes (PV)**

* PV is a low level representation of a storage volume. It is an abstraction for the physical storage device that is attached to the cluster.
* PV can be mounted into a pod through a PVC.
* PV is resource in the cluster that can be provisioned dynamically using Storage Classes, or they can be explicitly created by a cluster administrator.
* PV is independent of the lifecycle of the Pods. It means that data represented by a PV continue to exist as the cluster changes and as Pods are deleted and recreated.

**Persistent Volume Claim (PVC)**

* PVC is binding between a Pod and PV. Pod request the Volume through the PVC.
* PVC is the request to provision persistent storage with a specific type and configuration.
* PVCs describe the storage capacity and characteristics a pod requires, and the cluster attempts to match the request and provision the desired persistent volume.
* PVC must be in same namespace as the Pod. For each Pod, a PVC makes a storage consumption request within a namespace.
* PVC is similar to a Pod. Pods consume node resources and PVC consume PV resources.

**Storage Classes (SC)**

* StorageClass allows dynamic provisioning of Persistent Volumes, when PVC claims it.
* StorageClass abstracts underlying storage provider.
* StorageClass is used in conjunction with PVC that allow Pods to dynamically request a new storage.

**ConfigMaps**

* ConfigMap is used to store non-confidential data in key-value pairs.
* Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.
* ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

**Secrets**

* Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.
* Secret allows you store and manage a small amount of sensitive information such as passwords, tokens, keys, SSH keys, etc.
* With Secret, you don’t need to include confidential data in your application code.
* Secrets can be created independently of the Pods that use them.

**Service Account**

* Kubernetes uses Service Accounts to authenticate and authorize requests by pods to the Kubernetes API server.
* Service Account provides an identity for processes that run in a Pod.

**Role**

* Role defines what can be done to Kubernetes Resources.
* Role contains one or more rules that represent a set of permissions.
* Roles are namespaced, meaning Roles work within the constraints of a namespace.
* After creating a Role, you assign it to a user or group of users by creating a RoleBinding.

**ClusterRole**

* ClusterRole works the same as Role, but they are applied to the cluster as a whole.
* ClusterRoles are not bound to a specific namespace. ClusterRole give access across more than one namespace or all namespaces.
* After creating a ClusterRole, you assign it to a user or group of users by creating a RoleBinding or ClusterRoleBinding.
* ClusterRoles are typically used with service accounts.

**RoleBinding**

* RoleBinding is used for granting permission to a Subject.
* RoleBinding holds a list of *subjects* (users, groups, or service accounts), and a reference to the role being granted.
* Role and RoleBinding are used in namespaced scoped.
* RoleBinding may reference any Role in the same namespace.
* Defines who can perform the actions defined in a Role or ClusterRole (within a namespace).

**ClusterRoleBinding**

* ClusterRole and ClusterRoleBinding function like Role and RoleBinding, except they have wider scope.
* RoleBinding grants permissions within a specific namespace, whereas a ClusterRoleBinding grants access cluster-wide and to multiple namespaces.
* ClusterRoleBinding is binding or associating a ClusterRole with a Subject (users, groups, or service accounts).

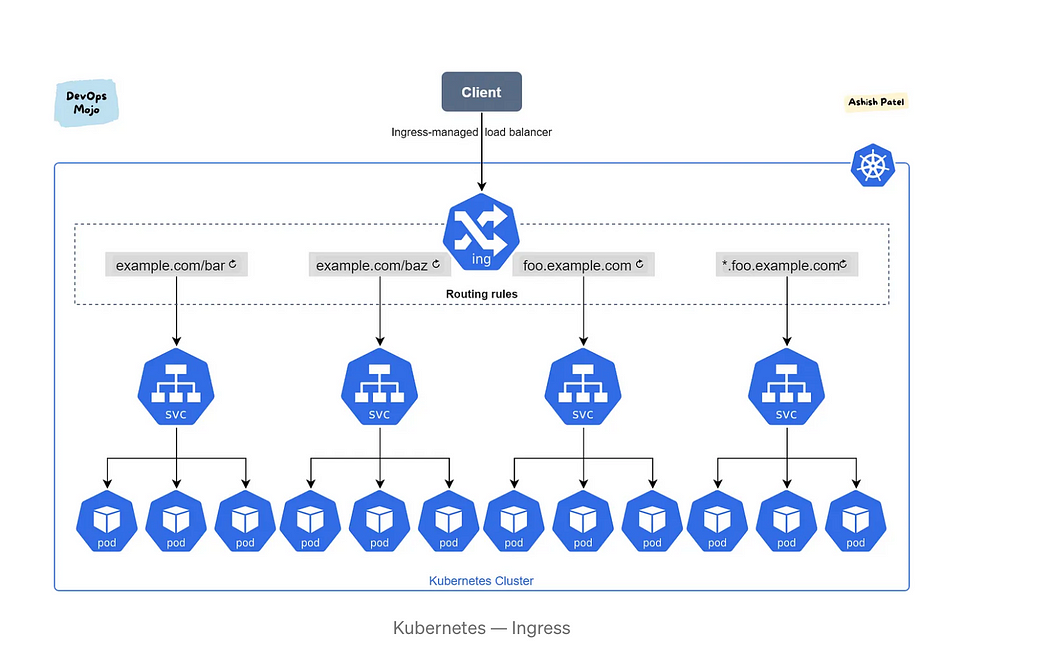
**Horizontal Pod Autoscaler (HPA)**

* HPA automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.
* HPA automatically scales number of pod replicas based on CPU usage or another metric.
* HPA controls the scale of a Deployment and its ReplicaSet.

**Pod Disruption Budget (PDB)**

* PDB limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions.
* PDB can temporarily halt the eviction process if the number of replicas of an application falls below the declared threshold. Eviction process will continue once the number of available replicas is over the threshold.
* PDB defines the minimum number of pods that must remain running when evacuating nodes.

**Ingress**



* [Ingress](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#ingress-v1-networking-k8s-io) 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. You can configure access by creating a collection of rules that define which inbound connections reach which services.
* An [Ingress controller](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.
* The Ingress [spec](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status) has all the information needed to configure a load balancer or proxy server. It contains a list of rules matched against all incoming requests. Ingress provides routing rules to manage external users’ access to the services in a Kubernetes cluster, typically via HTTPS/HTTP. With Ingress, you can easily set up rules for routing traffic without creating a bunch of Load Balancers or exposing each service on the node. This makes it the best option to use in production environments.
* *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 NodePort or LoadBalancer.*

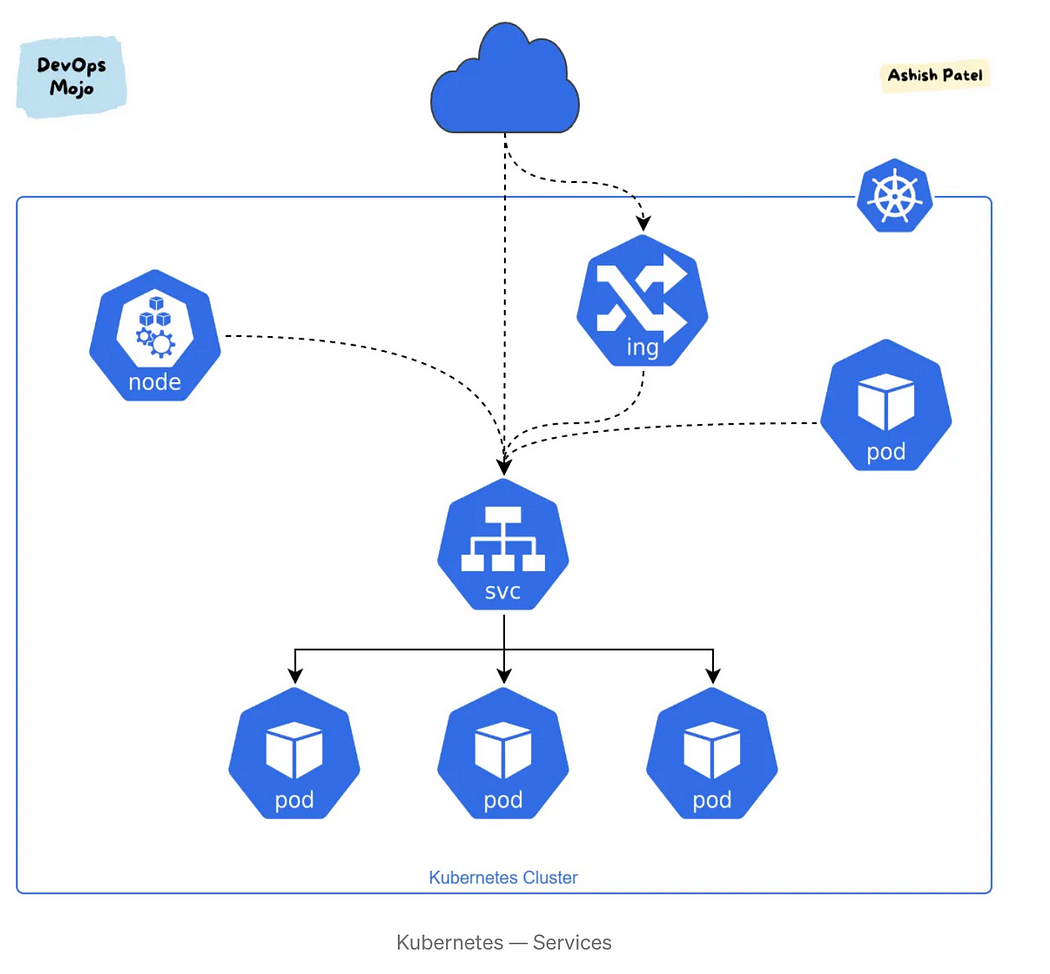
**Ingress controllers**

* Ingress controller is an application that runs in a cluster and configures an HTTP load balancer according to Ingress resources. The load balancer can be a software load balancer running in the cluster or a hardware or cloud load balancer running externally. Different load balancers require different Ingress controller implementations.
* In order to Ingress resource work, the cluster must have an ingress controller running.
* You can deploy any number of ingress controllers within a cluster.
* There are many different Ingress controllers, and there’s support for cloud-native load balancers (from GCP, AWS, and Azure).  
  e.g., Nginx, Ambassador, EnRoute, HAProxy, AWS ALB, AKS Application Gateway

**Use Cases**

* Externally reachable URLs for applications deployed in Kubernetes clusters.
* Load balancing rules and traffic, as well as TLS/SSL termination for each hostname, such as foo.example.com.
* Content-based routing:  
  1. *Host-based routing*. For example, routing requests with the host header foo.example.com to one group of services and the host header bar.example.com to another group.  
  2. *Path-based routing*. For example, routing requests with the URI that starts with /serviceA to service A and requests with the URI that starts with /serviceB to service B.

**Services**



**What is a Service in Kubernetes?**

* Kubernetes service is a Kubernetes object that provides stable networking for pods and provide a way to Kubernetes to configuring a proxy to forward traffic to a set of pods.
* It enables communication between nodes, pods, and users of app, both internal and external, to the cluster. Service also provides load balancing when you have Pod replicas.
* Services are a good abstraction for loose coupling for communication within the cluster, but also from external services like a browser request coming to the cluster.

**How Kubernetes Services Works?**

* Kubernetes assigns IP address to service on creation, just like a node or pod. These addresses get assigned from a service cluster’s IP range. Service is also assigned a DNS name based on the service name, and an IP port.
* Instead of static IP address-based assignments, Services use selectors (or labels) to define which pods uses which service. Services select Pods based on their labels. When a network request is made to the service, it selects all Pods in the cluster matching the service’s selector, chooses one of them, and forwards the network request to it.
* Service automatically discovers a new pod with labels that match the selector. This process seamlessly adds new pods to the service, and at the same time, removes terminated pods from the cluster.
* Kubernetes Service Types

1. ClusterIP
2. NodePort
3. LoadBalancer
4. ExternalName

**Kubernetes Service Key Points**

* Service gives stable networking endpoints for a set of Pods.
* Service gets single IP, DNS and Port that never change.
* Service enables how pods talk to each other inside the cluster.
* Service enables how pods expose to the world/outside of cluster.
* Service offers load balancing, naming and discovery isolation.
* Service only sends traffic to healthy Pods and hide unreliable Pods.
* Service can do session affinity.
* Service supports TCP and UDP.
* Service is Load Balancer in Cloud Kubernetes services.

apiVersion: v1  
kind: Service  
metadata:  
 name: my-service  
spec:  
 selector:  
 app: MyApp  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 9376

* As soon as a new pod appears in a cluster, with labels matching with Service’s selector, the app=MyApp in the example above - Service will start sending traffic to it.

**Endpoints**

* Usually, we don’t see Endpoints objects when using Kubernetes Services, as they are working under the hood, similarly to [ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/" \t "_blank) which are “hidden” behind Kubernetes Deployments.

**What else does a Kubernetes cluster need?**

**Container Registry**

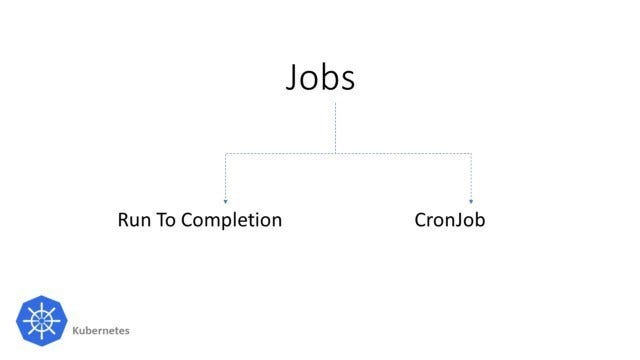
* The container images that Kubernetes relies on are stored in a container registry. This can be a registry you configure, or a third party registry like
* Docker Hub
* Amazon Elastic Container Registry (*ECR*)
* Azure Container Registry (*ACR*)
* Google Container Registry (*GCR*)

# Kubernetes Jobs

Kubernetes Jobs are created to run pods for a short period of time which are running for completion as opposed to different objects in Kubernetes like Replicasets, Replication Controllers, Demonesets, which is run continuously.

In Kubernetes, we have two types of Jobs

* **Run to completion**
* **Schedulers**



## Job (Run to completion)

It ensures to create one or more pods. It can also parallelly run one or more pods at the same time. It also refers as Jobs in Kubernetes

Let’s start to create Jobs and understand different methods to use it

1. Create Job.yaml file



2. Let’s deploy Job using this command

* kubectl create -f job.yaml

3. To check the job status

* kubetl get jobs

4. Check for pod initiate by Job

* kubectl get pods

Create multiple pod and parallelism on Job

Here is an example to run multiple pods with parallelism

* apiVersion: batch/v1  
  kind: Job  
  metadata:  
   name: parallel-job  
   generateName: kube-job-  
  # namespace: prod  
  spec:  
   completions: 5  
   parallelism: 2  
   ttlSecondsAfterFinished: 1  
   template:  
   spec:  
   containers:  
   - name: busybox  
   image: busybox  
   command: ["echo", "Kubernetes Job"]  
   restartPolicy: Never

Here we can also use [TTL Controler](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/) to cleanup completed Jobs

* ttlSecondsAfterFinished: 1

When running multiple Jobs with the same name it will generate an error stating that a job with the same name exists. To overcome this issue, we can use genrateName field in the metadata section This will generate pods with prefix Kube-job- with numbers

* generateName: kube-job-

## CronJob (Schedulers)

Running Cronjob on Kubernetes is as simple as we are running crontab on Linux.

It runs a job periodically on a given schedule, written in [Cron](https://en.wikipedia.org/wiki/Cron) format.

Note: Time is based on the timezone of the master where the job is initiated.

Here in this example, we will create Cronjob that will run a Linux image at the scheduled time.

We will also understand the best ways to schedule the cronjob in Kubernetes.

Creating a YAML for cronjob  
  


This YAML will schedule a pod ever one minute

schedule: schedule time of its jobs to be created and executed

concurrencyPolicy:Forbid The Cron job does not allow concurrent runs; if it is time for a new job run and the previous job run hasn't finished yet, the Cron job skips the new job run. So it can be used to keep our cluster clean environment.

successfulJobsHistoryLimit: 1  
failedJobsHistoryLimit: 1

It specifies how many completed and failed jobs should be kept in a cluster.

Here we kept limit 1 for both fields to keep our environment clean.