1. **Basic concepts**

Kubernetes, often abbreviated as K8s, is an open-source platform designed to automate deploying, scaling, and managing containerized applications.

It provides a robust framework for ensuring application reliability, scalability, and high availability by orchestrating containers across a cluster of machines. Kubernetes uses key concepts like **pods** (the smallest deployable units), **nodes** (machines in the cluster), and **services** (for load balancing and networking).

It offers features like self-healing (automatically restarts failed containers), horizontal scaling, rolling updates, and secret/configuration management.

By abstracting underlying infrastructure complexities, Kubernetes enables developers to focus on application development while ensuring efficient resource utilization and consistent application performance.

1. **A diagram of a diagram of a company

   Description automatically generatedChallenges without orchestration**

**orchestration** refers to the automated management and coordination of containerized applications across a cluster of machines. It ensures that containers are deployed, scaled, networked, and maintained according to the desired state defined by the user.

1. **Features of the Kubernetes**
2. **Automated Scheduling**: Kubernetes provides advanced scheduler to launch container on cluster nodes based on their resource requirements and other constraints
3. **Healing Capabilities**: Kubernetes allows to replace and reschedules containers when node dies. Kubernetes doesn’t allow containers to be used until they get ready.
4. **Auto Upgrade and Rollback**: Kubernetes rolls out changes to the application or its configuration  
   Monitoring Application ensure that Kubernetes doesn’t kill all instance at that time   
   If something goes wrong, with Kubernetes you can roll back the change.
5. **Horizontal Scaling**: Kubernetes can scale up and scale down the application as per the requirements with a simple command, using a UI, or automatically based on CPU usage.

**Storage Orchestration**: With Kubernetes, you can mount the storage system of your choice. You can either opt for local storage or choose a public cloud provider.

In Kubernetes, containers are ephemeral by default, meaning that their data is lost when they stop or restart. To persist data, you can mount external storage to containers using **volumes**. Kubernetes supports a variety of storage systems, which can be:

**Local Storage:**

* **Definition:** Storage provided by the physical disk on the node (machine) where the container is running.
* **Use Case:** Useful for workloads that require high performance and data that doesn’t need to be shared across nodes.
* **Examples:** Local SSDs or HDDs available on the host machine.

**Public Cloud Provider Storage:**

1. **Definition:** Storage services offered by cloud providers.
2. **Use Case:** Ideal for distributed applications that need scalable, reliable, and shared storage across multiple nodes.
3. **Examples:**
   * Amazon EBS (Elastic Block Store) for AWS
   * Google Persistent Disks for GCP
4. **Secret and Configuration Management**: Kubernetes can help you deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

**Secret and Configuration Management** refers to the capability to securely manage sensitive data (like passwords, API keys, or certificates) and application configuration (like environment variables) in a way that is dynamic and separate from application code or container images. You can deploy a new secret or configuration or update an existing one without needing to rebuild the container image or redeploy the application.

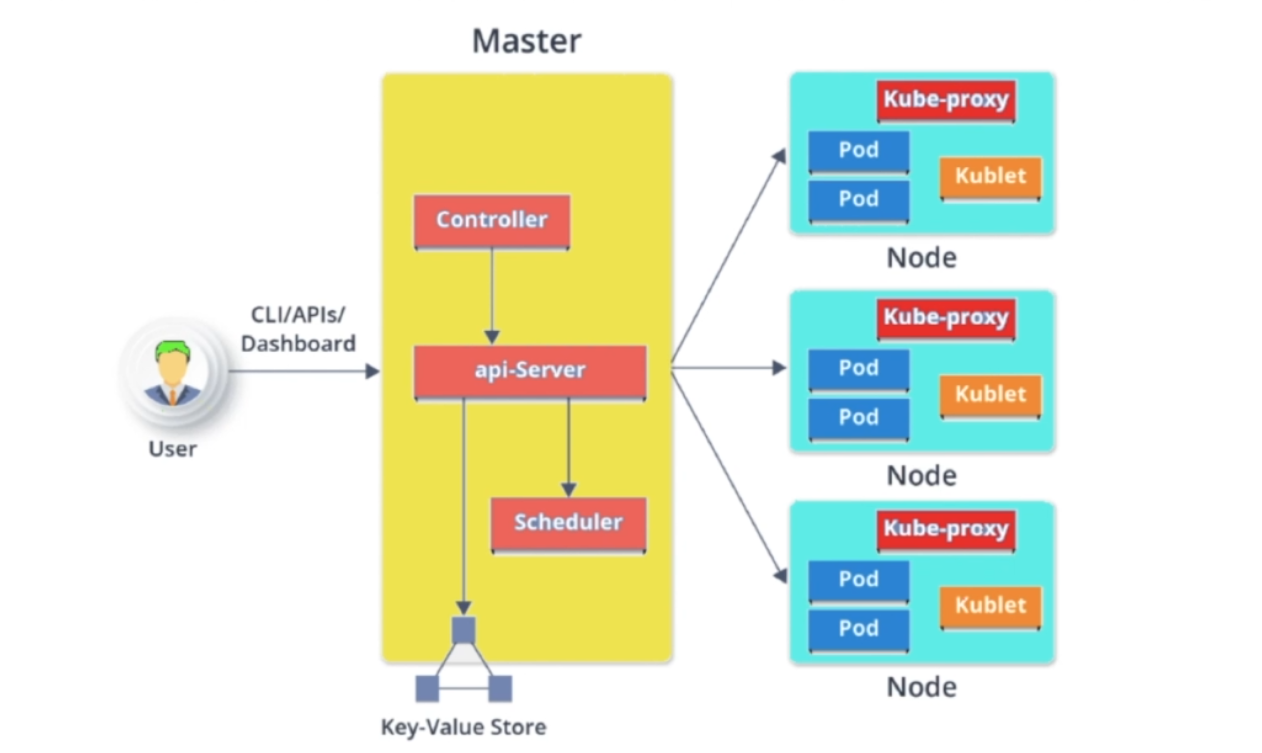
Kubernetes has a built-in mechanism to securely store and transmit sensitive data. Instead of hardcoding secrets in application code or including them in Docker images, they are stored as **Secrets** objects, which are encrypted at rest (if configured) and accessed by applications securely.

These secrets are injected into pods as environment variables or mounted as files, reducing the risk of accidental exposure.

1. **Kubernetes Architecture**

Kubernetes follow the Master-Slave(worker) Node architecture

Master Node: Responsible for the management of Kubernetes cluster. Its is mainly the entry point for all administrative tasks

More than one master nodes can be there in Kubernetes cluster

* **CLI/API/Dashboard:** In Kubernetes architecture, the **CLI (kubectl)** allows direct command-line interaction with the cluster, the **API Server** acts as the central communication hub for all operations between components, and the **Dashboard** provides a user-friendly web interface to monitor and manage cluster resources visually.  
    
  **kubectl** is the command-line interface (CLI) tool for interacting with Kubernetes clusters, enabling you to deploy, manage, and troubleshoot applications running on the cluster. It communicates with the Kubernetes API server to execute commands, such as creating or updating resources like pods, deployments, services, and namespaces.
* **Master**: The **master** in Kubernetes architecture refers to the **control plane**, which manages and orchestrates the cluster by making all critical decisions and ensuring the desired state of the system.It consist of components: Controller, api-server, schedular. The master nodes oversee cluster-wide functions like scaling, updates, and failover while communicating with worker nodes to execute workloads.  
  Multiple masters can be possible.  
    
  In multi master node system, **single master node** will be commanding node for own workers and other masters too.  
    
  Main Master node uses **etcd** to manage the workers and other master nodes
* **API Server:** The **API Server** in Kubernetes is a critical component of the control plane that exposes the Kubernetes REST API, acting as the gateway for all external and internal interactions with the cluster.  
    
  It serves as the central point for all communication, receiving requests from users, services, or other components, and then processing and validating them.  
    
  These CRUD operations are performed using **Kubernetes objects**, which represent the configuration and state of the resources. The API Server acts as the interface through which users, applications, or other cluster components can interact with and manipulate these objects, ensuring the cluster maintains the desired state as specified by users.  
    
  In Kubernetes, the **resulting state of the cluster** is stored in **etcd**, which is a distributed **key-value store**. This means that **etcd** functions as the central source of truth for all Kubernetes cluster data. It stores and maintains the **desired state** (what the system should look like) and the **current state** (what the system actually looks like) of various resources such as pods, deployments, services, and configurations.
* **Controller:** In Kubernetes, a **controller** is a control loop that continuously monitors the state of the cluster and works to ensure that the current state matches the desired state defined by the user.  
    
  Controllers are responsible for handling the lifecycle of specific resources, such as pods, nodes, and deployments, and performing necessary actions when discrepancies are detected.  
    
  Key types of controllers include the **ReplicaSet Controller** (which ensures that the specified number of pod replicas are running), the **Deployment Controller** (which manages updates and rollbacks of applications), the **Node Controller** (which monitors the health of nodes), and the **StatefulSet Controller** (which handles the management of stateful applications).|  
    
  Each controller is continuously monitoring its assigned resource, and when a change or failure occurs (e.g., a pod crashes or a node goes down), the controller automatically takes corrective actions, such as rescheduling pods, scaling resources, or replacing failed components, to maintain the desired state of the system.  
    
  It performs lifecycle functions such as namespace creation, event garbage collection, node garbage collection  
    
  Controller watches the desired state of the objects it manages and watches their current states through the API server
* **Scheduler:** It is a control plane component responsible for determining which node a newly created pod should be placed on, based on available resources and constraints.  
    
  It continuously monitors the cluster for unscheduled pods and selects the most appropriate node for each pod, considering factors like resource requirements (CPU, memory), node capacity, affinity rules (e.g., where pods should or should not run), taints and tolerations (which nodes should not run certain pods), and other factors such as node health and priority.
* **Worker Node:** It’s a physical server or a vm which runs the applications using pods  
    
  Worker nodes contain all the necessary services to manage the networking between the container, communicate with the master node and assign resources to the scheduled container
* **Kublet:** It is an agent which communicates with the master node and executes on nodes or the worker nodes. Bridge between master and slave. Kublet gets the configuration of a pod from the API server and ensures that the described containers are up and running.  
    
    
  It monitors the state of the node and the pods running on it and ensures that the containers within those pods are started, stopped, and running properly.  
    
  The kubelet also manages node-level tasks like volume mounting and managing resources such as CPU and memory limits for containers, making it a crucial agent for managing the operational health of nodes in a Kubernetes cluster.
* **Pods:** A **pod** in Kubernetes is the smallest and simplest deployable unit that encapsulates one or more containers, along with their shared storage, network, and configurations.  
    
  Pods are the basic unit of execution in a Kubernetes cluster and ensure that containers that need to work together are tightly coupled, sharing the same **IP address**, **hostname**, and **storage volumes**. Containers within a pod can communicate with each other directly via **localhost**, and they are scheduled on a **node** by Kubernetes.  
    
  Pods can be managed and scaled using higher-level Kubernetes objects like **Deployments** or **ReplicaSets**, which provide automatic updates, scaling, and recovery.  
    
  **A pod can run multiple containers**, which share resources like networking and storage. **A node (machine) can run multiple pods**, where each pod is isolated from others and can have its own set of containers.
* **Kube-Proxy:** The kube-proxy is a key component in Kubernetes that runs on each node and manages network traffic, ensuring that requests to services are correctly routed to the appropriate pods.  
    
  It is responsible for implementing the service abstraction by maintaining network rules on each node, allowing communication between pods across different nodes and exposing services to external traffic.  
    
  It manages subnetting at the individual node level, ensuring that each pod can communicate with others regardless of which node they are on. kube-proxy implements network rules using iptables or IPVS (IP Virtual Server), which determine how traffic is routed to services and their corresponding pods.  
    
  Kubernetes nodes are usually part of a large network, and each node (machine) within the cluster typically gets its own IP address within this larger network. Within the node, pods also receive their own unique IP addresses, but these addresses are isolated and don’t directly interfere with the addresses of other nodes or pods. Subnetting here is used to control the range of IP addresses assigned to pods and nodes, ensuring that each node’s local network can handle traffic correctly and efficiently.  
    
  Kube-Proxy acts as a network proxy and a load balancer for a service on a single worker node. A network **proxy** refers to its role in managing and routing network traffic between external clients (or other services) and the appropriate **pods** in the Kubernetes cluster. When a request is made to a **Kubernetes service** (which represents a set of pods), kube-proxy intercepts the request and checks where it should go. It then forwards the request to one of the pods that is part of the service, based on load-balancing rules or routing logic.
* **Key-value store:** key-value store is a distributed, highly available data storage system used to store all the cluster state information.

The most common key-value store used in Kubernetes is **etcd**, which is part of the Kubernetes control plane. This store holds crucial data such as **configuration information**, **cluster metadata**, **secrets**, **API object configurations**, and **state information** for all Kubernetes components, including nodes, pods, services, deployments, and more.

1. **Kubernetes vs Docker Swarm**

| Feature | Kubernetes | Docker Swarm |
| --- | --- | --- |
| Orchestration | Advanced, full-fledged container orchestration platform | Simpler container orchestration solution focused on ease of use |
| Complexity | More complex, provides a broader feature set | Simpler and easier to set up and manage |
| Scalability | Highly scalable, supports large clusters with thousands of nodes and pods | Limited scalability, better suited for smaller clusters |
| Architecture | Master-worker architecture with multiple components (API server, scheduler, controller manager, etc.) | Simpler, with manager and worker nodes, with fewer components |
| Load Balancing | Built-in sophisticated load balancing for services across the cluster | Built-in, simpler load balancing, limited compared to Kubernetes |
| Service Discovery | Native service discovery and DNS integration | Native service discovery, but limited in feature |

1. **Ways to setup**

Kubernetes has ability to run anywhere  
  
Integration required for certain cloud providers like aws and gcp  
  
Volumes and ELBs(External load balancer) work with certain cloud providers  
AWS,GCP has this feature  
  
First demo will be with Minikube to spin up single machine with Kubernetes cluster  
  
Setup Kubernetes on AWS using kops  
  
Kops can be used to spin a highly available production cluster  
  
Really recommend to try the setup yourself,once

**Setup Kubernetes with minikube**

Things to remember:

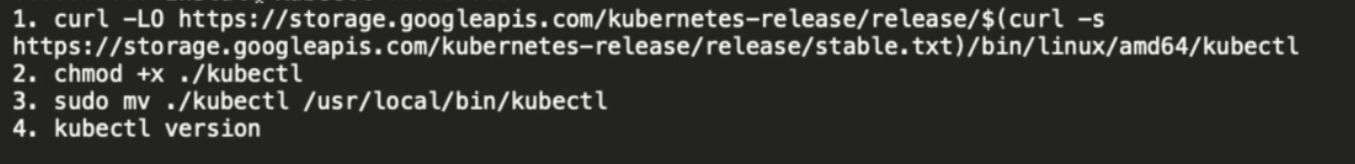
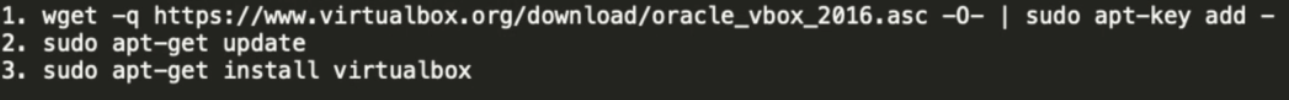
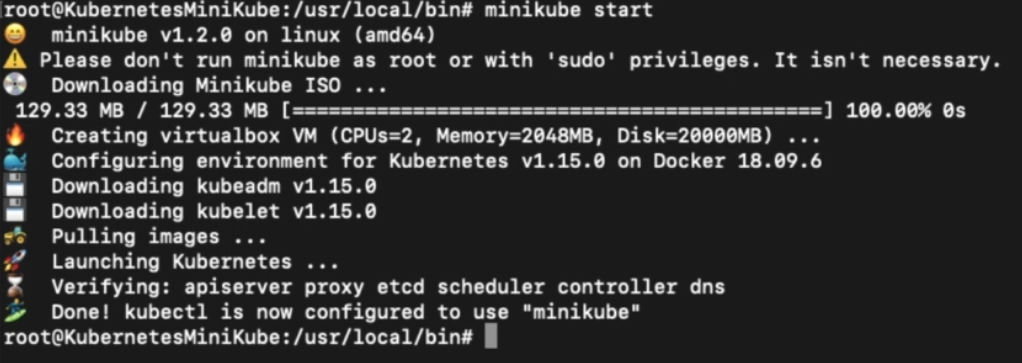
Minikube is a lightweight tool that lets you set up and run a single-node Kubernetes cluster locally on your machine, making it ideal for learning, development, and testing.Minikube Runs a single node Kubernetes Cluster inside the linux VM

Minikube is used for development Purposes

Minikube cant be used in production,as Its one node machine

We need atleast 2gb ram and 2 core cpu

1. Create a linux machine from digital ocean and ssh it
2. Run df -k to see the space available
3. A computer screen with white text

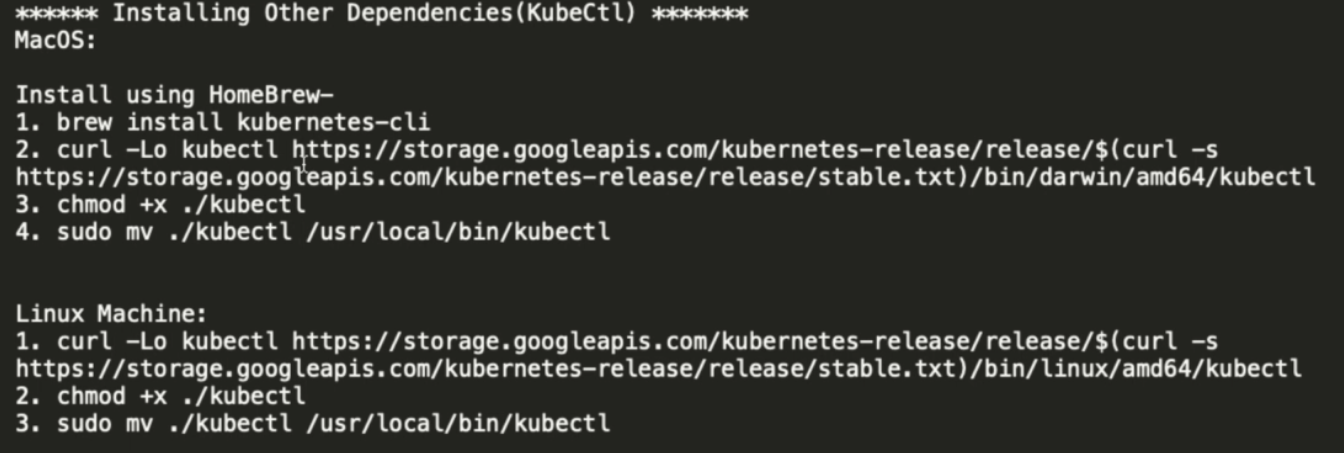
   Description automatically generatedFirst install docker CE edition
4. Install KubeCtl: **kubectl** is the command-line interface (CLI) tool for interacting with Kubernetes clusters, enabling you to deploy, manage, and troubleshoot applications running on the cluster. It communicates with the Kubernetes API server to execute commands, such as creating or updating resources like pods, deployments, services, and namespaces.
5. Install minikube:
6. Install VirtualBox: Minikube uses VirtualBox (or similar tools) to create a virtual machine (VM) that isolates the Kubernetes cluster from your host system. This prevents conflicts between the cluster's environment and the host's operating system.
7. Run the command:  
   minikube start  
     
   The command minikube start initializes and starts a local Kubernetes cluster using Minikube. It creates a single-node Kubernetes cluster by setting up the necessary control plane components (API server, scheduler, controller-manager, etc.) and worker node processes. It uses a default or specified driver (like VirtualBox, Docker, or KVM) to provision the cluster.
8. A screenshot of a computer program

   Description automatically generatedInteract Cluster using Kubectl
9. **Kops Introduction**

Kops (Kubernetes Operations) is an open-source command-line tool that simplifies the creation, management, and scaling of production-grade Kubernetes clusters on cloud providers like AWS, GCP, and others. It automates cluster provisioning by managing infrastructure components, including virtual machines, networking, storage, and load balancers, while offering support for HA (High Availability) configurations.  
  
Kops generates configuration files, applies updates seamlessly, and integrates with tools like Terraform for managing cluster resources as code. It supports Kubernetes upgrades, rolling updates, and security configurations. With built-in validation and SSH key management, Kops is ideal for deploying robust and secure Kubernetes clusters efficiently.  
  
Kops is an official Kubernetes project for managing production grade Kubernetes clusters. Kops is currently the best tool to deploy Kubernetes clusters to AWS  
  
Kops has commands for creating clusters, updating settings, and applying the changes  
  
Kops automates a large part of operating Kubernetes on AWS and **only works for Mac/Linux**

1. **Set Up AWS for Kops**
   1. A black background with white text

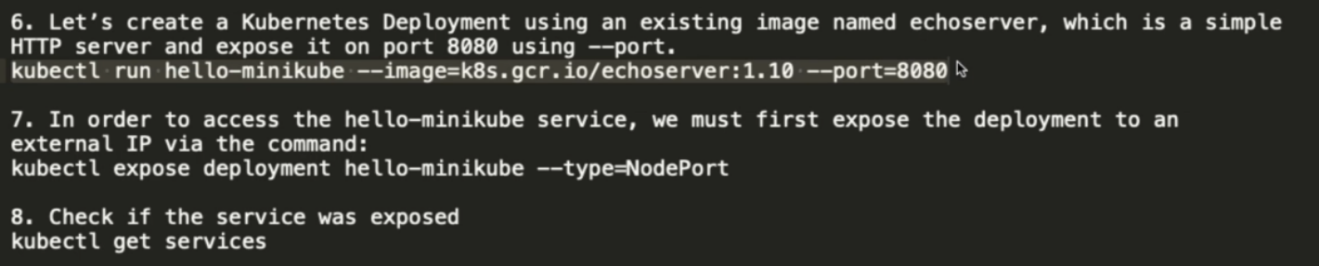
      Description automatically generatedInstall kops on linux
   2. A black screen with white text

      Description automatically generatedInstall on macOS
   3. Install the dependencies(kubectl) required for it:
   4. A screen shot of a computer program

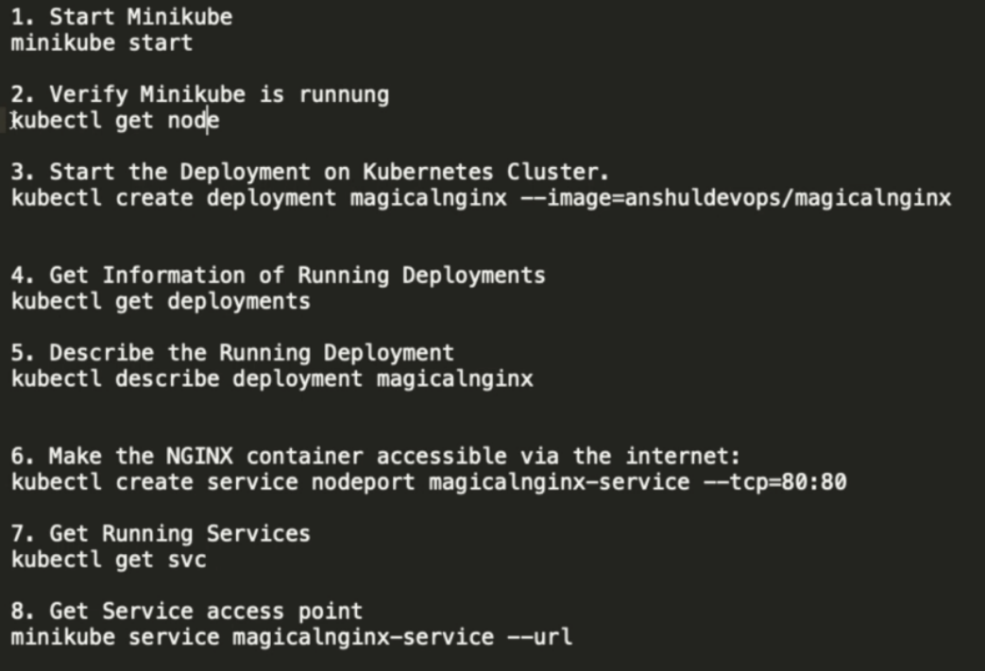
      Description automatically generatedInstall Python PIP: Python and pip are needed when setting up AWS for Kops because they enable the installation and management of essential tools and libraries, like the **AWS Command Line Interface (CLI)** and **boto3** (the AWS SDK for Python). These tools are crucial for interacting with AWS services, such as storing the Kops cluster state in S3, configuring DNS with Route 53, and provisioning infrastructure like EC2 instances.
   5. A screen shot of a computer

      Description automatically generatedInstall AWS CLI: AWS CLI (Command Line Interface) is essential when working with Kops for Kubernetes setup on AWS because it provides the tools needed to interact with AWS services directly from the command line.
   6. Setup on the machine is done. Now we need to setup on the aws side. First you need to create/manage aws account. Then you need to setup IAM permission for kops. Create an user(kops) and give them permission  
        
      **Permission required for kops user:**  
      **Amazon EC2 Full Access  
      Amazon Route53 Full Access  
      AmazinS3 Full Access  
      IAM Full Access  
      Amazon VPC Full Access**
   7. After creating a iam user with access type: Programmatic access , after this create a group for that user with a tag named kops. Then aws will give you a access key ID and secret
   8. Go to your machine terminal and type command: aws configure  
        
      it will ask you for access key id and aws secret key.
   9. Now we need to create a s3 bucket for the **KOPS\_STATE\_STORE**. The KOPS\_STATE\_STORE is an essential environment variable in Kops that specifies the location where cluster configuration and state information are stored. Typically, this is an Amazon S3 bucket (or equivalent object storage like GCS for Google Cloud). This storage location serves as a centralized repository for all cluster-related data, including configuration files, secrets, manifests, and metadata required to create, update, and maintain the Kubernetes cluster.  
        
      it is like etcd  
        
      get the fastest region for deploy the s3 bucket  
        
      User can use [www.cloudping.info](http://www.cloudping.info) to choose the fastest region per their location
   10. Go to services and storage and s3 > create bucket> bucket name> select region>leave other as default.
   11. Once the s3 bucket is ready, kops cluster must be **valid DNS names. (With Kops 1.6.2 or later, DNS configuration is optional)**For a company like Facebook setting up Kubernetes with Kops, the cluster must have a valid DNS name, such as k8s-cluster.facebook.com, which follows the rules of a fully qualified domain name (FQDN). This DNS name is essential for Kops to configure DNS records using AWS Route 53, enabling the cluster's components, like the API server, to be discoverable and accessible.  
         
       For example, if your cluster is named k8s-cluster.example.com, Kops will configure Route 53 to associate this name with the IP or load balancer of the API server, enabling users and services to connect to the cluster through this DNS name. This simplifies communication and ensures the cluster is properly integrated into the AWS ecosystem.  
         
       With **Kops version 1.6.2 or later**, DNS configuration using external services like AWS Route 53 becomes **optional** if the cluster name ends with .k8s.local. In this case, Kops automatically sets up an internal DNS configuration, bypassing the need for external DNS management. This is useful for private or development clusters where external DNS is unnecessary. The .k8s.local suffix triggers Kops to use a built-in DNS resolver or configure a private DNS setup that works within the cluster's VPC, simplifying the process for environments that do not require public DNS resolution.
   12. Verify kubectl is installed
   13. Generate SSH keys: ssh -f .ssh/id\_rsa  
         
       The command ssh-keygen -f ~/.ssh/id\_rsa generates a pair of SSH keys—a private key (id\_rsa) and a public key (id\_rsa.pub)—and stores them in the .ssh directory of your home folder.  
         
       SSH keys are used to securely authenticate and establish a connection between your local machine and the remote instances (e.g., EC2 instances) created by Kops. This ensures that only authorized users can access the servers without requiring a password.
   14. A computer screen shot of white text

       Description automatically generatedCreate a cluster  
         
       Explanation:  
         
        **--name=<cluster-name>**:  
       Specifies the name of the cluster. This must be a valid DNS name (e.g., mycluster.k8s.local or mycluster.example.com).
       * + - **--state=s3://<kops-state-store>**:  
             Points to the S3 bucket where the cluster's state and configuration will be stored (e.g., s3://my-kops-bucket).
           - **--zones=<aws-availability-zone>**:  
             Specifies the AWS availability zone(s) where the cluster will be deployed (e.g., us-east-1a).
           - **--node-count=<number-of-nodes>**:  
             Defines the number of worker nodes to create in the cluster (e.g., 3).
           - **--node-size=<instance-type-for-nodes>**:  
             Specifies the EC2 instance type for worker nodes (e.g., t3.medium).
           - **--master-size=<instance-type-for-master>**:  
             Specifies the EC2 instance type for master nodes (e.g., t3.medium).We need to create EC2 instance.  
               
             The EC2 instances use the public key provided by your local machine to authenticate your SSH connection.
           - **--dns-zone=<dns-hosted-zone>**:  
             Specifies the Route 53 DNS hosted zone if using a custom domain (e.g., example.com). For .k8s.local clusters, this can be skipped.
   15. A computer screen with white text

       Description automatically generatedIf you want to create a Kubernetes cluster using Kops **without configuring external DNS**, you can use a cluster name with the .k8s.local suffix.
   16. After cluster creation we should validate the cluster by command: kops validate cluster
   17. Cluster has been create now you need to create deployment using and existing image
   18. But if we copy the IP of ec2 instance and the port that we get from the cluster (we can get it from kubectl get status) in the browser, it will be unable to load.   
         
       It is because of the Security group of EC2 instance . We have to modify the security group and in the inbound traffic we have to add the cluster port and who can access to “everyone”
   19. A black background with white text

       Description automatically generatedDeleting the cluster
2. **Run First Custom Image on Local Kubernetes**

For that we need a docker image on the docker hub

The image is coming from the hub.docker.com

1. A screenshot of a computer program

   Description automatically generated**Run custom Image on AWS Kubernetes**

can create a new domain through route53.   
 Display the output in json, yaml format

kubectl create service loadbalancer magicalnginx –-cp=80:80  
  
This creates a **Service** of type LoadBalancer in Kubernetes. A Load alancer Service exposes the application to external traffic by provisioning a cloud load balancer (e.g., an AWS ELB or GCP load balancer).

1. **Basics of Kubernetes**

**Kubernetes Node Workflow**

A diagram of a server

Description automatically generatedIn Kubernetes, a **node** is a worker machine (physical or virtual) that runs application workloads as Pods. Each node runs essential components: the **kubelet**, which ensures containers are running as expected; the **kube-proxy**, which manages networking for Pods and routes traffic within the cluster; and a container runtime (e.g., Docker or containerd) to run the containers.  
  
Nodes are managed by the **control plane**, which schedules workloads, monitors health, and ensures desired state. Nodes communicate with the control plane to report resource usage, health, and events. They can host multiple Pods, which share the node's resources.  
  
If a node fails, Kubernetes reschedules affected Pods to healthy nodes. Nodes are dynamically added or removed to maintain cluster scalability and availability, and they are critical for the distributed nature of Kubernetes.  
  
Single master can manage upto 5000 worker nodes.

When a user accesses a Kubernetes Service using a domain name, the request is routed to the cloud provider's **load balancer** created by the Service of type LoadBalancer. The load balancer receives the request and forwards it to one of the Kubernetes worker nodes in the cluster. Within the worker node, **iptables**, a Linux kernel utility, is used to manage network routing rules. It directs the incoming traffic to the appropriate Pod running the application. Kubernetes automatically configures **iptables rules** for the Service, ensuring that traffic is evenly distributed across all Pods matching the Service's selector, even if they are running on different nodes. This seamless redirection ensures high availability and load balancing of the application.  
  
**iptables** is a Linux utility used for packet filtering and NAT (Network Address Translation). It allows administrators to define rules for how incoming, outgoing, and forwarded network packets should be handled. These rules are stored in a series of tables (filter, nat, mangle, etc.), which consist of chains (like INPUT, OUTPUT, FORWARD) that hold the rules. In Kubernetes, **iptables** rules are dynamically configured by kube-proxy to direct traffic to the correct Pods.

**Scaling Pods in Kubernetes**

You can scale the pods Vertically or Horizontally

**A Stateless Application can be scaled up horizontally**  
  
A **stateless application** is one that does not retain session data or state between requests, treating each request independently and relying on external sources (like databases) for necessary information. This design makes it ideal for **horizontal scaling**, where additional instances of the application are added to handle increased traffic, rather than upgrading a single server’s resources (vertical scaling).  
  
For example, imagine an online store’s search feature. When you search for a product, the request is processed independently, so the system can add more servers to handle searches as more users join.

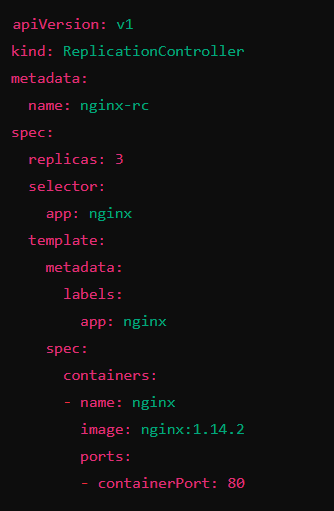
all traditional database are stateful services, they have file system that can’t be split over the multiple instances

**A Stateful Application can be scaled up Vertically**

A **stateful application** is one that keeps track of user data or session information, meaning its functionality depends on maintaining a consistent state. Because this state is tied to specific resources, it is challenging to scale horizontally by adding more instances. Instead, stateful applications are typically scaled **vertically**, which means improving the performance of a single instance by upgrading its hardware resources, such as CPU, memory, or storage, to handle more load while preserving the state.  
  
For instance, if a bank’s database server experiences heavy transaction traffic, the bank may upgrade the server's CPU, RAM, or storage to handle more queries and larger datasets instead of deploying multiple database instances. This ensures that the stateful data—such as account balances or transaction logs—remains accurate and synchronized while accommodating higher workloads.  
  
**Scaling in Kubernetes can be done using Replication Controller**

A **Replication Controller** in Kubernetes is a resource that ensures a specified number of identical Pods are always running. It automatically manages the creation, deletion, and scaling of Pods to match the desired replica count. If a Pod fails, the Replication Controller will create a new one to replace it, ensuring the system remains stable and reliable. It is useful for ensuring high availability and fault tolerance by maintaining the desired state of the application. However, starting from Kubernetes version 1.8, the **ReplicaSet** has largely replaced the Replication Controller, offering more advanced features like managing rolling updates, though the core functionality remains the same.  
  
It ensures that a specified number of pod replicas are running at any one time.  
  
If there are too many pods, the replication Controller terminates the extra ones and add if needed. It can manage multiple pods on single node or multiple pods on multiple nodes.

**Lab Scaling Pods in Kubernetes with Replication Controller**

* ****Create Replication Controller Manifest(YAML) file.  
    
  A Replication Controller **manifest** file in Kubernetes is a YAML or JSON configuration file used to define the desired state for a set of identical Pods. It includes key details like the replica count, which specifies how many instances of the Pod should be running, and the Pod template, which outlines the specifications for the Pods to be created (e.g., container images, environment variables, ports). **Name**: The ReplicationController is named nginx-rc, uniquely identifying it in the cluster.

**Replica Count**: Ensures exactly 3 nginx Pods are always running.

**Selector**: Matches and manages Pods with the label app: nginx.

**Pod Template**:

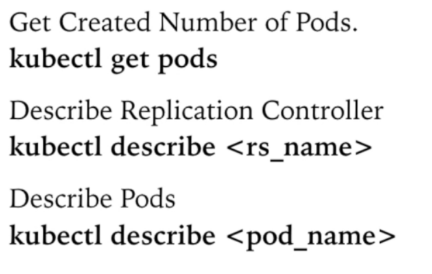
Creates Pods running nginx:1.14.2.

Exposes port 80 for the nginx web server.

**Functionality**: Automatically replaces failed or deleted Pods to maintain 3 replicas, ensuring high availability.

* Minikube start
* **A screenshot of a computer

  Description automatically generatedA screenshot of a computer program

  Description automatically generated**Execute Replication Controller manifest file  
  kubectl create -f <filename>

**This command is for replication controller . the replica sets have different commands**

kubectl apply -f <replica-set-manifest.yaml>

**A screenshot of a computer screen

Description automatically generated**

**Replica Set**

Replica Set is an enhanced version of replication controller. Replica Set purpose is to maintain a stable set of replica pods running at any given time .

**Difference Between Replication Controller and ReplicaSet**

| **Feature** | **Replication Controller** | **Replica Set** |
| --- | --- | --- |
| **Introduction** | Older Kubernetes resource for managing Pods. | Newer, more advanced resource introduced in later versions. |
| **Selector Matching** | Supports equality-based selectors (e.g., key=value). | Supports both equality-based and set-based selectors (e.g., key in (value1, value2)). |
| **Usage in Deployments** | Not used by Deployments. | Used by Deployments for managing rolling updates and other features. |
| **Advanced Features** | Basic functionality; lacks rolling updates or versioning. | Supports rolling updates, versioning, and enhanced flexibility. |
| **Adoption** | Largely deprecated but still supported for legacy setups. | The recommended resource for Pod management in modern Kubernetes. |

In Kubernetes, **selector matching** refers to how the system identifies and matches Pods that should be managed by controllers like **Replication Controllers** or **ReplicaSets**. A **Replication Controller** only supports **equality-based selectors**, which means it can match Pods based on simple key-value pairs (e.g., app=nginx). However, a **ReplicaSet** is more flexible and supports both **equality-based** and **set-based selectors**. With set-based selectors, you can specify conditions like matching a key with a set of values, such as app in (nginx, apache)  
  
Imagine you're a store manager who needs to organize products. In the case of a **Replication Controller**, you can only match products that are labeled exactly as "type=electronics." However, with a **ReplicaSet**, you have more flexibility: you can match products that are labeled as either "type=electronics" or "type=furniture" using a set-based selector, or you can even match all products except those labeled A screenshot of a computer program

Description automatically generated"type=clothing" using advanced set-based operations.

**Deployment**

A deployment declaration in Kubernetes allows user to do app deployments and updates

In Kubernetes, a **Deployment** is a higher-level resource used to manage the lifecycle of applications, ensuring that a specific number of identical Pods are running, updated, and scaled in a controlled way. It provides features like **rolling updates** (gradually updating Pods to minimize downtime), **rollback** (reverting to a previous version if an update fails), and **self-healing** (automatically replacing failed Pods). A Deployment is typically used with **ReplicaSets** to manage Pod replicas, and it allows you to declaratively define your application state, making it easier to scale and maintain applications with minimal manual intervention.

A close-up of a document

Description automatically generatedImagine you run an online store that uses a web application deployed on Kubernetes. You have a Deployment for your web server with 5 replicas to handle incoming customer traffic. If you want to update the application to a newer version, Kubernetes will perform a **rolling update**, updating one Pod at a time to ensure there’s no downtime for customers. If something goes wrong with the update, you can roll back to the previous version, ensuring your store remains functional and stable. This makes managing your application easier, just like how a store manager would control product stock and make incremental changes to the store layout without disrupting business operations.

A screenshot of a computer program

Description automatically generated  
  
  
**Services**

In Kubernetes, **Services** are abstractions that expose a group of **Pods** to the network, enabling communication between components within a cluster or with external clients. They provide a stable IP and DNS name, ensuring connectivity even as Pods are dynamically created or terminated.  
  
Services operate through **selectors**, matching Pods using labels to define the backend endpoints. Key types include **ClusterIP** (internal communication within the cluster), **NodePort** (external access via a port on each node), **LoadBalancer** (integrates with cloud provider load balancers for external access), and **ExternalName** (maps to an external DNS name).  
  
Services rely on kube-proxy for routing traffic to the appropriate Pods and can use **headless services** (without a cluster IP) to directly expose Pod IPs for advanced use cases like stateful applications. This design ensures scalability, reliability, and consistent communication across Kubernetes workloads.  
 **Kubectl Expose** command create a service for pods so that they can be accessible externally.  
  
Creating a service will create an end point for pods.

A screenshot of a computer

Description automatically generatedThe set of pods targeted by a service is usually determined by a selector in manifest file

**metadata**:

* Defines the Service name (my-web-app-service) and labels.

**spec.selector**:

* Matches Pods with the label app: my-web-app to route traffic.

**ports**:

* Maps the Service's port (80) to the targetPort (8080) of the Pods.

**type**:

* Defines the Service type. Here, ClusterIP exposes the Service only within the cluster.

**Labels**

In Kubernetes, **labels** are key-value pairs that are attached to resources like Pods, Services, Deployments, and Nodes to organize and identify them. Labels allow you to categorize and select resources based on specific attributes, such as app: my-web-app or environment: production. These labels are used by Kubernetes components to group and select resources for various purposes, such as routing traffic or applying configurations.  
  
Labels are flexible, can be added or modified at any time, and are crucial for querying resources using commands like kubectl get pods -l app=my-web-app.

1. **Service Discovery using DNS**

Service discovery in Kubernetes using DNS is a mechanism that allows services within a cluster to find and communicate with each other using human-readable names instead of IP addresses, which may change. Kubernetes automatically assigns each Service a DNS name following the pattern <service-name>.A diagram of a service

Description automatically generated

**Scenario**: Suppose you have two Services in your Kubernetes cluster:

**backend-service** (handles business logic).

**frontend-service** (serves the UI and talks to the backend).

**Backend Deployment and Service**:

Deployment creates Pods running a backend app.

A Service named backend-service exposes these Pods.

Kubernetes assigns the DNS name: backend-service.default.svc.cluster.local (assuming it's in the default namespace).

**Frontend Deployment and Service**:

Another Deployment creates Pods running the frontend app.

A Service named frontend-service exposes these Pods.

The frontend needs to call the backend for data.

**DNS in Action**:

From a frontend Pod, the app sends a request to http://backend-service:8080.

Kubernetes' CoreDNS resolves backend-service to its **ClusterIP** (e.g., 10.96.0.1) and forwards the request to one of the backend Pods.

The communication happens without needing fixed IPs or manual configurations.

When a frontend app sends a request to backend-service, it doesn’t need to know the exact IP address of the backend Pods. Instead:

1. **CoreDNS Role**: Kubernetes includes a built-in DNS system called **CoreDNS**. Think of it as a phonebook that translates the name backend-service into the backend service's internal IP address (called **ClusterIP**), such as 10.96.0.1.
2. **ClusterIP**: This IP is a stable, virtual IP for the backend service. It acts as a single entry point for all requests to the backend Pods.
3. **Request Forwarding**: After resolving the name, Kubernetes uses the backend service's built-in load balancer to pick one of the backend Pods and send the request to it. This ensures the request reaches a working backend Pod without the frontend worrying about specific IPs.
4. **Config Map Vs Secrets**

| **Aspect** | **ConfigMap** | **Secret** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Purpose** | Stores non-sensitive configuration data like environment variables or config files. | Stores sensitive information like passwords, API keys, or certificates. |

|  |  |  |
| --- | --- | --- |
| **Data Encoding** | Data is stored as plain text. | Data is stored in Base64-encoded format. |

|  |  |  |
| --- | --- | --- |
| **Security** | Not encrypted; readable by anyone with access to the cluster. | More secure; requires specific permissions to access. |

|  |  |  |
| --- | --- | --- |
| **Usage Example** | Application configuration (e.g., log levels, external URLs). | Secrets like database credentials or TLS certificates. |

|  |  |  |
| --- | --- | --- |
| **Default Storage** | Stored in etcd as plain text. | Stored in etcd as Base64-encoded text. |

|  |  |  |
| --- | --- | --- |
| **Access in Pods** | Mounted as files or exposed as environment variables. | Mounted as files or exposed as environment variables. |

1. **Ingress**

An **Ingress** in Kubernetes is an API object that manages external access to services within a cluster, typically HTTP and HTTPS traffic. It acts as an intelligent layer-7 load balancer, allowing you to define routing rules to direct requests to the appropriate services based on URLs, hostnames, or paths.

Unlike Services like NodePort or LoadBalancer, which expose applications with basic mechanisms, Ingress provides more advanced features like SSL termination, custom domains, and rewriting paths.

To use Ingress, you need an **Ingress Controller** (e.g., NGINX, Traefik) deployed in your cluster, as it handles the actual traffic routing. Ingress simplifies exposing multiple services under a single external IP or domain name, enabling scalable and efficient web application deployment.

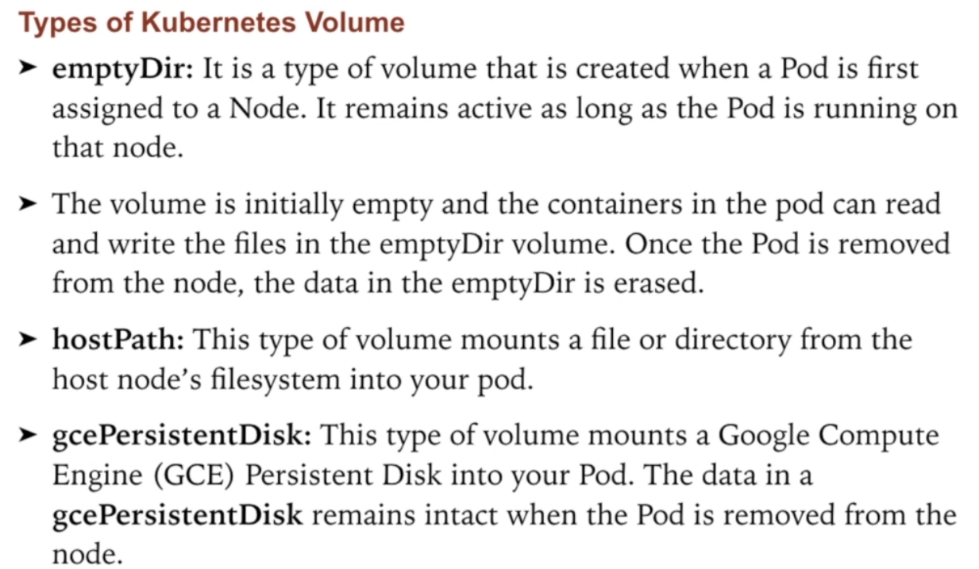
1. **Volumes**

A **volume** in Kubernetes is a way to provide persistent or shared storage to Pods, overcoming the limitation that data inside a container is ephemeral and lost when the container restarts.

Volumes are mounted into containers and can store data that persists across container restarts within a Pod's lifecycle. Kubernetes supports multiple volume types, including **emptyDir** (temporary storage tied to a Pod), **hostPath** (uses the node’s filesystem), and **persistentVolumeClaim (PVC)** for durable storage linked to PersistentVolumes (PVs), enabling storage across Pod lifecycles. Some volumes, like **configMap** and **secret**, are used for injecting configuration and sensitive data into Pods.

Volumes in Kubernetes allow user to store data outside the container

**A screenshot of a computer

Description automatically generatedEmptyDir**

An **emptyDir** in Kubernetes is a volume type that provides temporary storage shared across containers in the same Pod. It is created when the Pod is assigned to a node and lasts for the Pod's entire lifecycle. The data in an emptyDir is initially empty and can be used for purposes like caching, temporary file sharing, or log processing between containers.

If a container within the Pod restarts, the data in the emptyDir remains intact, but if the Pod itself is deleted or rescheduled to a different node, the data is lost. By default, emptyDir uses the node’s default storage (e.g., disk), but it can be configured to use memory (medium: Memory) for faster access and storage that doesn't persist on disk. This makes emptyDir ideal for temporary, non-persistent data needs.