**What is Monolithic Architecture and Microservices Architecture?**

**Monolithic Architecture:** Imagine a restaurant where everything happens in one big kitchen. This kitchen handles taking orders, cooking food, and serving customers all in a single place. In this scenario, if the kitchen gets too crowded or if there’s a problem with one part of the kitchen, it can affect the entire restaurant’s operation. If the chef is sick, the entire kitchen may come to a halt, impacting the entire dining experience.

**Microservices Architecture:** Now, consider a food delivery service like Zomato or Swiggy. Instead of one big kitchen, they have a network of different restaurants, each specializing in a specific type of regional food or cuisine. When you place an order, it’s not prepared in a single kitchen rather, each restaurant (microservice) prepares its own portion of the order. These portions are then assembled and delivered to you. If one restaurant has an issue, it doesn’t necessarily impact the others. For example, if the burger place is busy, it won’t affect the rolls restaurant’s ability to fulfil orders.

**Kubernetes Origins and Open Source:**

Kubernetes was created by Google in 2013 in Golang. Initially, Kubernetes was not open source but in 2014, google introduced Kubernetes open source and donated to CNCF.

**Languages Supported by Kubernetes**:

Kubernetes supports both YAML and JSON for configuration.

**Features of Kubernetes:**

● **AutoScaling:** Kubernetes supports two types of autoscaling horizontal and vertical scaling for large-scale production environments which helps to reduce the downtime of the applications.

● **Auto Healing:** Kubernetes supports auto healing which means if the containers fail or are stopped due to any issues, with the help of Kubernetes components (which will talk in upcoming days), containers will automatically repair or heal and run again properly.

● **Load Balancing:** With the help of load balancing, Kubernetes distributes the traffic between two or more containers.

● **Platform Independent:** Kubernetes can work on any type of infrastructure whether it’s On-premises, Virtual Machines, or any Cloud.

● **Fault Tolerance:** Kubernetes helps to notify nodes or pods failures and create new pods or containers as soon as possible

● **Rollback:** You can switch to the previous version.

● **Health Monitoring of Containers:** Regularly check the health of the monitor and if any container fails, create a new container.

● **Orchestration:** Suppose, three containers are running on different networks (On-premises, Virtual Machines, and On the Cloud). Kubernetes can create one cluster,

that has all three running containers from different networks.

**Alternatives of Kubernetes:**

● **Docker Swarm**

● **Apache Mesos**

● **OpenShift**

● **Nomad, etc**

**Kubernetes Architecture by Kubernetes:**

Kubernetes follows client-server architecture where the Master Node and Worker node exist which constitutes a ‘Kubernetes Cluster’. We can have multiple worker nodes and Master nodes according to the requirement.

**Control Plane**

The control plane components, including the API server, etcd, scheduler, and controller manager, are typically found on the master node(s) of a Kubernetes cluster. These components are responsible for managing and controlling the cluster.

**Master Node**

The master node is responsible for the entire Kubernetes cluster and manages all the activities inside the cluster in which master nodes communicate with the worker node to run the applications on the containers smoothly. Master Node has four primary components which help to manage all the things that we have discussed earlier:

1. **API Server:** In Simple terms, after installing the **Kubectl** on the master node developers run the commands to create pods. So, the command will go to the API Server, and then, the API Server forwards it to that component which will help to create the pods. In other words, the API Server is an entry point for any Kubernetes task where the API Server follows the hierarchical approach to implement the things.

2. **Etcd:** Etcd is like a database that stores all the pieces of information of the Master node and Worker node (entire cluster) such as Pods IP, Nodes, networking configs, etc. Etcd stored data in key-value pair. The data comes from the API Server to store in etc.

3. **Controller Manager:** The controller Manager collects the data/information from the API Server of the Kubernetes cluster like the desired state of the cluster and then decides what to do by sending the instructions to the API Server.

4. **Kube Scheduler:** Once the API Server gathers the information from the Controller Manager, the API Server notifies the Scheduler to perform the respective task such as increasing the number of pods, etc. After getting notified, the Scheduler takes action on the provided work.

**Worker Node**

The Worker Node is the mediator who manages and takes care of the containers and communicates with the Master Node which gives the instructions to assign the resources to the containers scheduled. A Kubernetes cluster can have multiple worker nodes to scale resources as needed.

The Worker Node contains four components that help to manage containers and communicate with the Master Node:

1. **Kubelet:** kubelet is the primary component of the Worker Node which manages the Pods and regularly checks whether the pod is running or not. If pods are not working properly, then kubelet creates a new pod and replaces it with the previous one because the failed pod can’t be restarted hence, the IP of the pod might be changed. Also, kubelet gets the details related to pods from the API Server which exists on the Master Node.
2. **Kube-proxy**: kube-proxy contains all the network configuration of the entire cluster such as pod IP, etc. Kube-proxy takes care of the load balancing and routing which comes under networking configuration. Kube-proxy gets the information about pods from the API Server which exists on Master Node.
3. **Pods:** A pod is a very small unit that contains a container or multiple containers where the application is deployed. Pod has a Public or Private IP range that distributes the proper IP to the containers. It’s good to have one container under each pod.
4. **Container Engine**: To provide the runtime environment to the container, Container Engine is used. In Kubernetes, the Container engine directly interacts with container runtime which is responsible for creating and managing the containers. There are a lot of Container engines present in the market such as CRI-O, containerd, rkt(rocket), etc. But Docker is one of the most used and trusted Container Engine.

**Why Set Up Minikube?**

Minikube provides an excellent environment for learning and experimenting with Kubernetes without the need for a full-scale cluster. It’s perfect for developers and enthusiasts who want to get hands-on experience with Kubernetes in a controlled environment.

**Kubeconfig**

● **Purpose:** Kubeconfig files are used for cluster access and authentication. Kubeconfig defines how ‘kubectl’ or any other Kubernetes clients interact with the Kubernetes cluster.

● **Contents:** The Kubeconfig file contains information about the cluster, user credentials, certificates, and context.

● **Usage:** Kubeconfig files are used by Administrators, developers, or CI/CD systems to authenticate the Kubernetes cluster. They decide who can access and how to access the cluster.

*Kubeconfig files can be stored in the user’s home directory* ***(~/.kube/config)*** *or specified using the KUBECONFIG environment variable.*

Example of a Kubeconfig file:

apiVersion: v1

kind: Config

clusters:

- name: my-cluster

cluster:

server: https://<CLUSTER\_ENDPOINT>

certificate-authority-data: <BASE64\_ENCODED\_CA\_CERT>

users:

- name: my-user

user:

client-certificate-data: <BASE64\_ENCODED\_CLIENT\_CERT>

client-key-data: <BASE64\_ENCODED\_CLIENT\_KEY>

contexts:

- name: my-context

context:

cluster: my-cluster

user: my-user

current-context: my-context

**Service File**

● **Purpose:** Service files contain all information about networking. The service file defines how networking will be handled on the cluster. Also, the Service file enabled the load balancing option for the applications which is a premium feature of Kubernetes.

● **Contents:** The service file specifies the service’s name, type (ClusterIP, NodePort, LoadBalancer, etc [Discuss in Upcoming Blogs]), and selectors to route traffic to pods.

● **Usage:** Service files are used by developers.

**Note**: Services can also be used for internal communication between Pods within the cluster, not just for exposing applications externally.

**Example of a service file**:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: my-app

ports:

- protocol: TCP

port: 80

targetPort: 8080

**Deploying Your First Nodejs Application on Kubernetes Cluster**