KUBERNETES ROADMAP 2024

Making your Kubernetes journey simple!

Learn Prerequisites

Basic Linux Commands
Containerization Concepts
Networking Fundamentals
YAML Syntax
Cloud Fundamentals

Kubernetes Architecture

Kubernetes Cluster
Kubernetes Master Node / Controle Plane: API Server, Scheduler, Kube Controller Manager, Cloud Controller Manager, etcd
Kubernetes Worker Nodes/ Data Plane: Kubelet, Kube-proxy, Container runtime

Kubernetes Cluster Set-up

Self-Hosted Solutions
Turnkey Solutions
Managed Kubernetes Services
Kubernetes Distributions

Set-up & Learn Kubernetes Cli

Install Kubectl
k9s
kubectx/kubens

Kubernetes Concepts

Basics: Namespace, Pods, ReplicaSet, Deployment, DaemonSet, StatefulSet, Jobs and CronJobs

Resource Management: Measurement, Setting Resource Requests and Limits, limit Namespaces

Auto Scaling: HPA, VPA, CA
Storage: Volume, CSI, PVC, PV

Services and Networking

Pod Networking, Services, Ingress, KVN

Scheduling

Affinity and Anti-Affinity,
Taints and Tolerations , Annotations

Monitoring & Logging

Kubernetes Dashboard, Prometheus, Grafana, ELK stack etc

Security

Kubernetes Secrets, RBAC, KNS

Click to Watch the video

@Sandip Das

What is Kubernetes?

Kubernetes is a powerful open-source orchestration tool, designed to help us manage microservices and containerized applications across a distributed cluster of computing nodes.

Using Kubernetes we can automate many of the manual processes involved in deploying, managing, and scaling containerized applications.

Kubernetes aims to hide the complexity of managing containers through the use of several key capabilities, such as REST APIs and declarative templates that can manage the entire lifecycle.

Prerequisites?

- **Basic Linux Commands:** Familiarize yourself with Linux command-line operations.
- **Containerization Concepts:** Understand Docker and how containers work.
- **Networking Fundamentals:** Grasp the basics of networking, including DNS, load balancing, and port mapping.
- **YAML Syntax:** Learn how to write and understand YAML files.
- **Cloud Fundamentals:** Get a basic understanding of cloud services and providers like AWS, Azure, or GCP.

History of Kubernetes

Birth of the Borg System

2003-2004

From Borg to Omega

2013

Google Introduces Kubernetes

2014

Kube v1.0 & CNCF

2015

Kubernetes Goes Mainstream!

2016

Enterprise Adoption & Support

2017

More Cloud Adoptions - GKE, EKS, and AKS become generally available

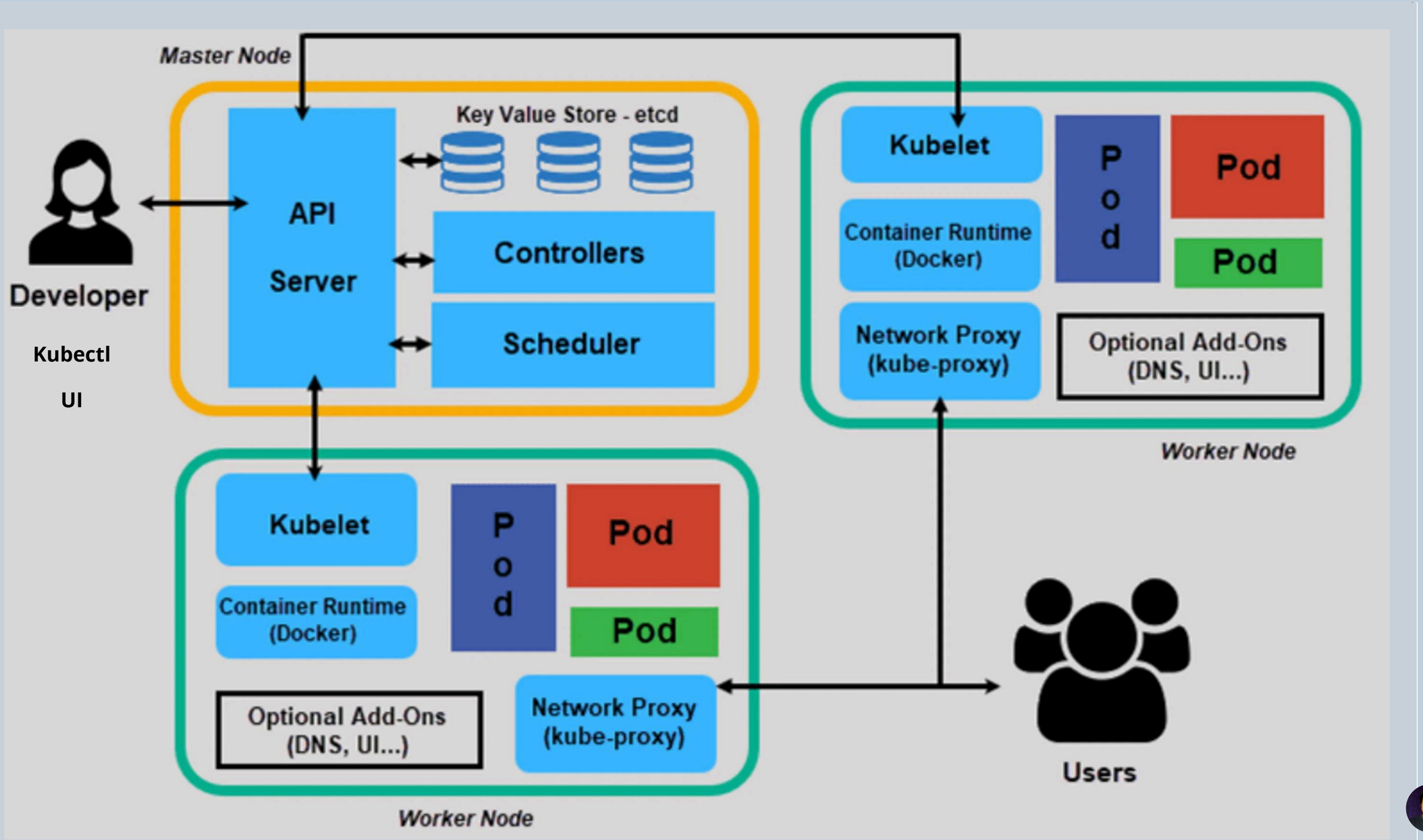
2018

And the story continued....

@Sandip Das

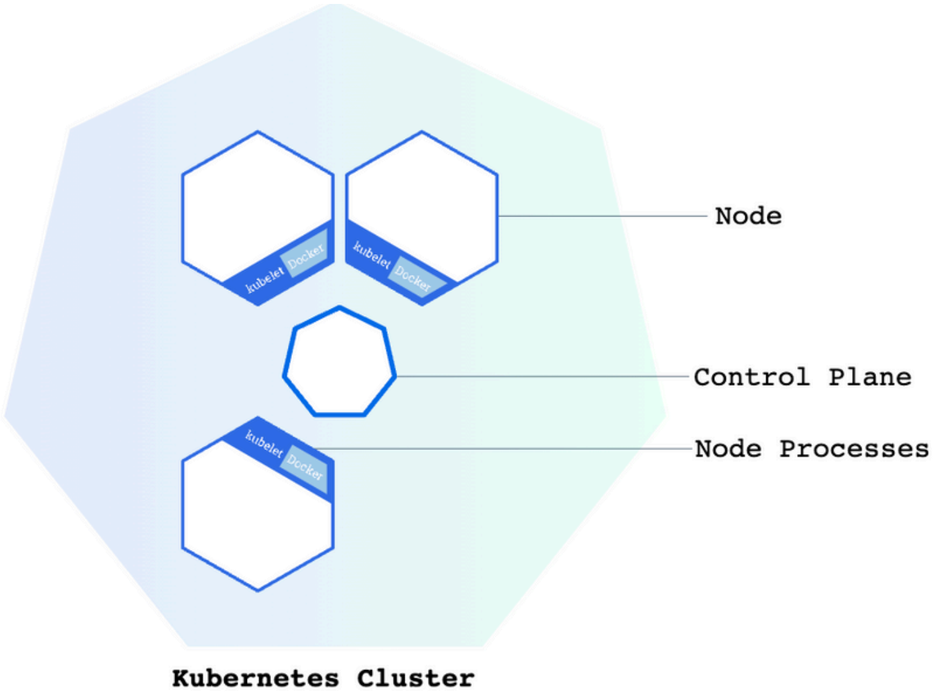
Kubernetes Architecture

Kubernetes Architecture



@Sandip Das

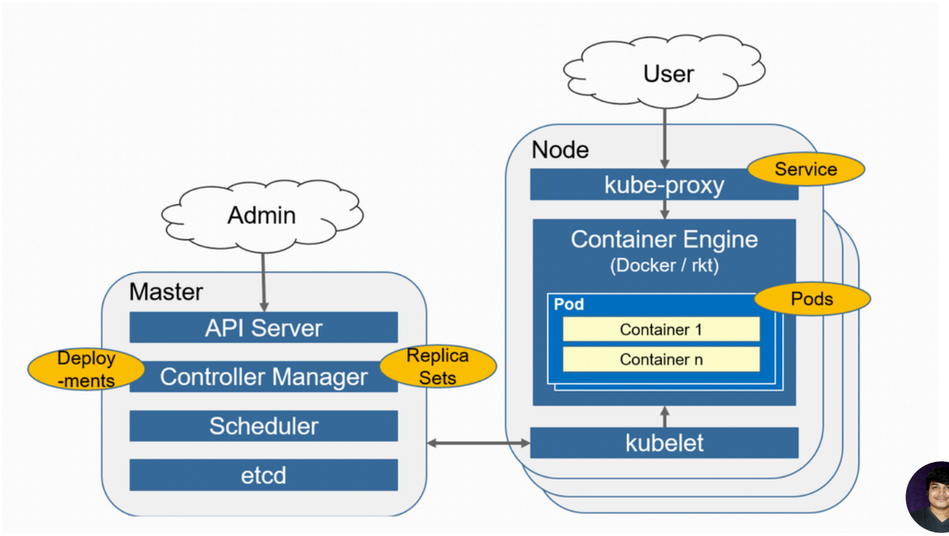
Kubernetes Cluster



What is Kubernetes Cluster?

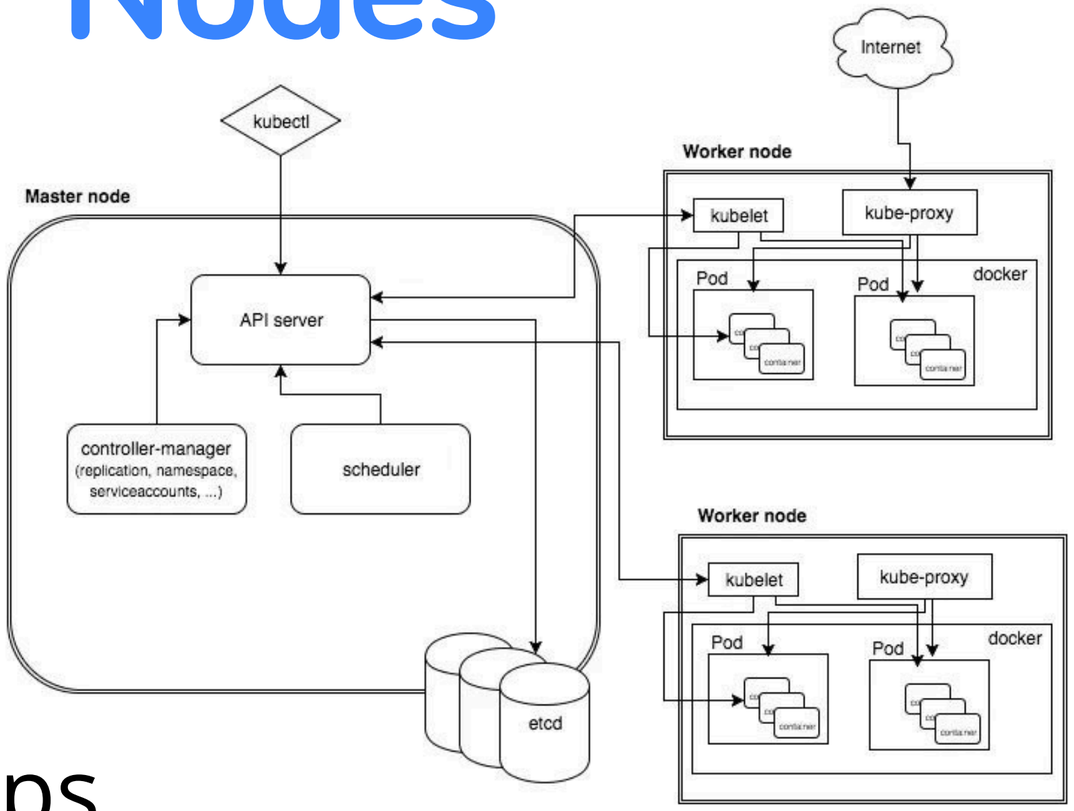
A Kubernetes cluster is a collection of nodes on which workloads can run, these nodes can be physical (bare metal) machines, or virtual machines (VMs), or serverless compute systems like Amazon Fargate.

Kubernetes cluster enables us to schedule and run containers across a collection of nodes



@Sandip Das

Kubernetes Nodes



What is Kubernetes Nodes?

Kubernetes runs your workload by placing containers into Pods to run on Nodes

A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

Type of Nodes:

Master Node: The master node controls the state of the cluster. It does Scheduling and scaling applications, Maintaining a cluster's state.

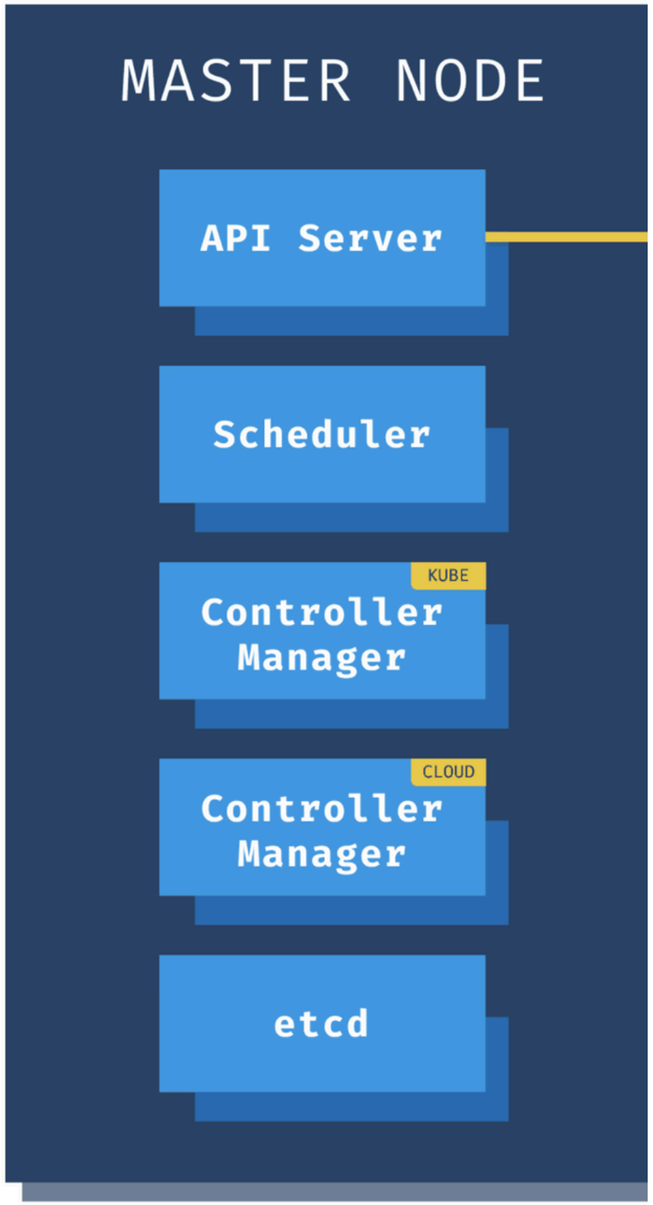
Worker Node: The worker nodes are the components that run the applications. Worker nodes perform tasks assigned by the master node.

Tips

1. **There must be a minimum of one master node and one worker node for a Kubernetes cluster to be operational**
2. **Cluster name must be domain compliant**
3. **We can make any node as un-schedulable using the "cordon" command e.g. kubectl cordon $NODENAME and to make node schedulable again, use the "uncordon" e.g. kubectl uncordon $NODENAME**

@Sandip Das

Master Node(s)



What is Master Node(s)?

This node hosts the Kubernetes control plane and manages the cluster. It acts as the “brains” of the cluster

Components

API Server: It's the entry point or gateway for REST / kubectl. It receives all REST requests for modifications to pods, services, replication sets/controllers, and others i.e serving as a frontend to the cluster. It's the only coponent communicates with the etcd

Scheduler: It schedules pods to worker nodes. It reads the service’s operational requirements and schedules it on the best fit node.

Kube Controller Manager: It always evaluates the current vs the desired state and checks for a change in configurations, if any change in the configuration occurs, it spots the change, starts working to get the desired state.

Cloud Controller Manager: This is responsible for managing controller processes with dependencies on the underlying cloud provider

etcd: a simple, distributed key value storage which is used to store the Kubernetes cluster data (such as number of pods, their state, namespace, etc),

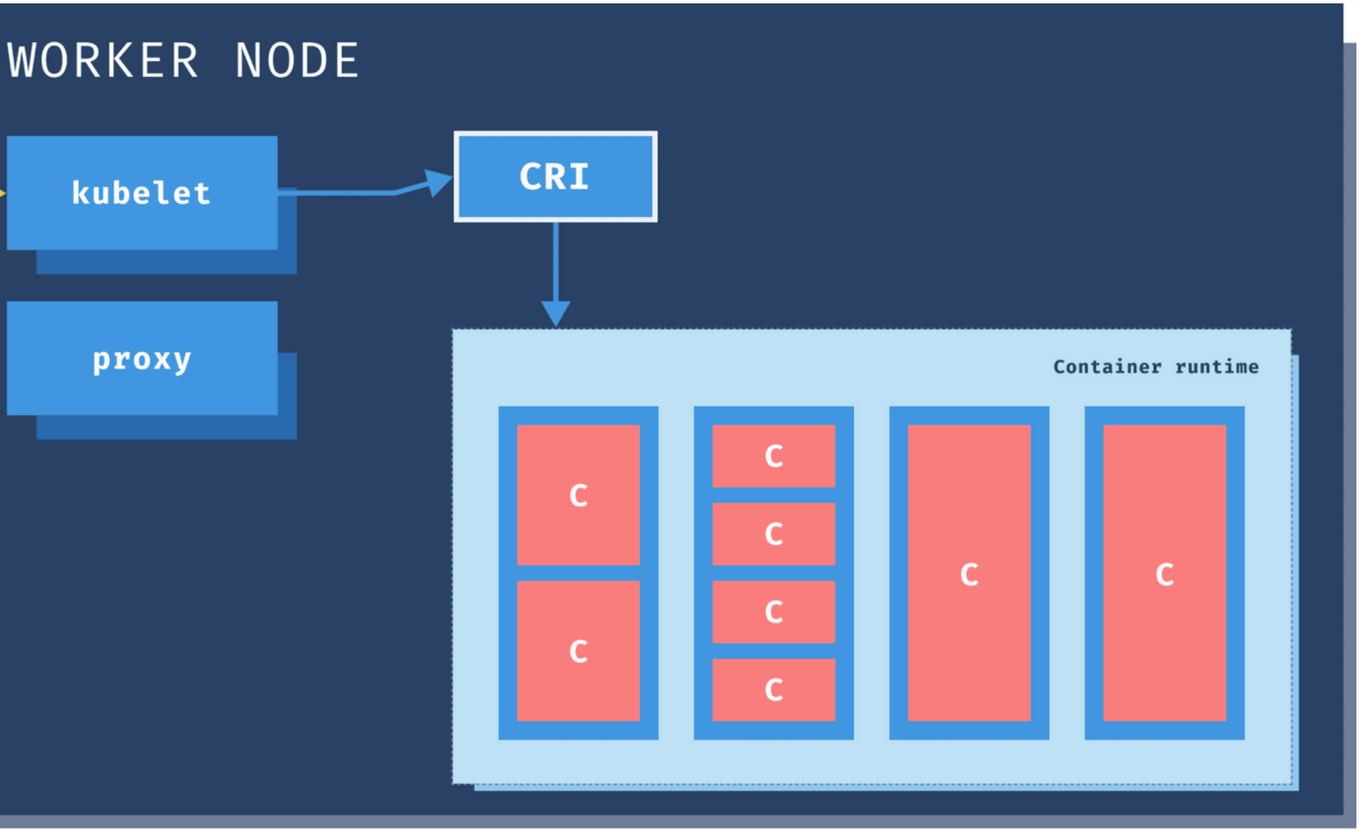
@Sandip Das

@Sandip Das

Worker Node(s)

What is Worker Node(s)?

These are the machines/nodes which host the data plane, where containers (workloads) are deployed. Every node in the cluster must run a container runtime such as Docker, as well as the below-mentioned components.

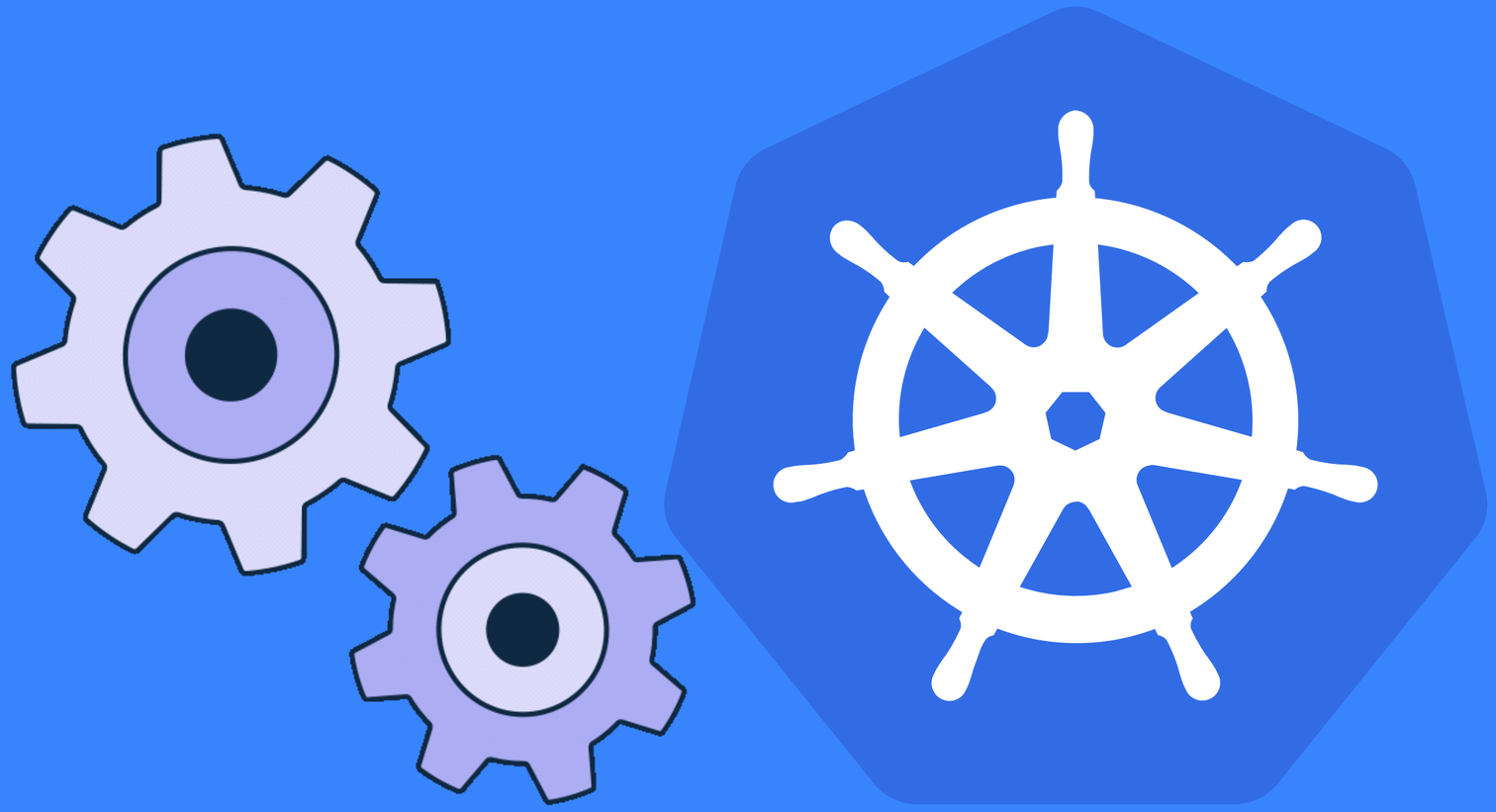


Components

Kubelet: This is responsible for the running state of each node, regularly taking in new or modified pod specifications (primarily through the kube-apiserver), and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running

Kube-proxy: It's a proxy service implementation that manages IP translation and routing, such as network proxy and a load balancer, that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.

Container: This resides inside a pod. The container is the lowest level of a micro-service, which holds the running application, libraries, and their dependencies. Containers can be exposed to the world through an external IP address. Kubernetes has supported Docker containers since its first version. In July 2016 the rkt container engine was added



Kubernetes Cluster-setup

Difference ways to set-up Kubernetes Cluster

Self-Hosted Solutions

Turnkey Solutions

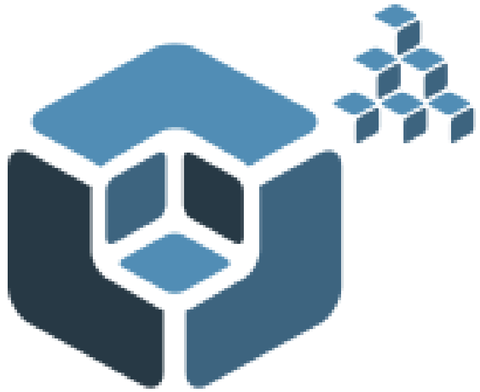
Managed Kubernetes Services

Kubernetes Distributions

@Sandip Das

1. Self-Hosted Solutions

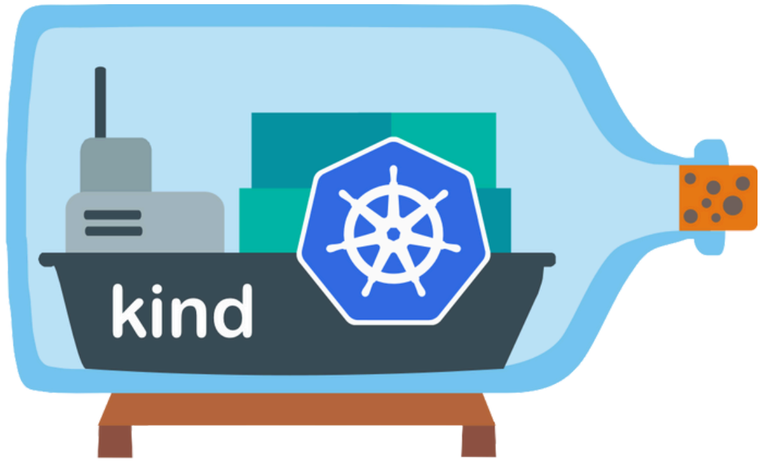
- **Kubeadm**: A tool provided by the Kubernetes project for easily setting up a Kubernetes cluster. You manage both the control plane and worker nodes. Follow this article for steps.
- **Kubespray**: A community project that uses Ansible playbooks to deploy and manage Kubernetes clusters. Follow this article for steps.
- **k0s**: A lightweight, single-binary Kubernetes distribution that can be used for creating Kubernetes clusters with minimal overhead. Follow this article for steps.
- **MicroK8s**: A lightweight, single-node Kubernetes solution for local development by Canonical (Ubuntu). Follow this article for steps.
- **k3s**: A lightweight Kubernetes distribution optimized for edge and IoT devices by Rancher Labs. Follow this article for steps.



@Sandip Das

2. Turnkey/Minimum Effort Solutions

- **Minikube**: A local Kubernetes cluster for development purposes, which runs on a single node. Follow this article to get started.
- **Kind (Kubernetes IN Docker)**: A tool for running local Kubernetes clusters using Docker container nodes. Ideal for testing Kubernetes clusters and CI/CD workflows. Follow this article to get started
- **Docker Desktop Kubernetes**: Kubernetes can be enabled within Docker Desktop, providing a local, single-node cluster for development purposes. Follow this article to get started.
- **Vagrant + kubeadm**: Use Vagrant to provision VMs and kubeadm to set up a Kubernetes cluster for development or testing. Follow this article to get started.




@Sandip Das

3. Managed Kubernetes Services

- **AWS Elastic Kubernetes Service (EKS)**: Fully managed service by AWS that handles most of the control plane operations.
- **Google Kubernetes Engine (GKE)**: Managed Kubernetes service by Google Cloud, offering features like auto-upgrades and node pools.
- **Azure Kubernetes Service (AKS)**: Managed Kubernetes service by Azure, providing integrated CI/CD with Azure DevOps.
- **IBM Cloud Kubernetes Service**: Managed Kubernetes service by IBM Cloud.
- **Oracle Kubernetes Engine (OKE)**: Managed Kubernetes service by Oracle Cloud Infrastructure.


@Sandip Das

4. Kubernetes Distributions

- **OpenShift**: An enterprise Kubernetes platform by Red Hat, with additional tools and services for enterprise use.
- **Tanzu Kubernetes Grid**: VMware's enterprise-grade Kubernetes distribution, part of their Tanzu suite.
- **Charmed Kubernetes**: Canonical's Kubernetes distribution provides a complete, production-grade solution on Ubuntu.


@Sandip Das

Kubernetes Concepts

CLI

kubectl

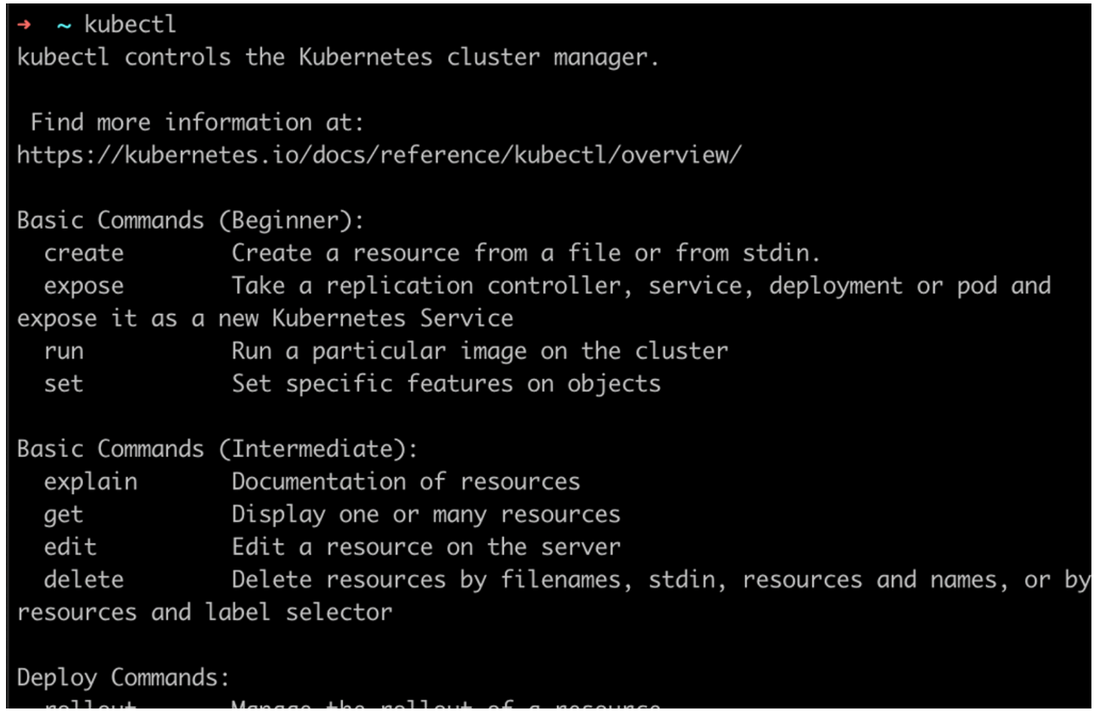
kubectl command is a line tool that interacts with kube-apiserver and send commands to the master node. Each command is converted into an API call.

Syntax

kubectl [command] [TYPE] [NAME] [flags]

Examples

kubectl apply -f ./my-manifest.yaml ***# create resource(s)***
kubectl get services ***# List all services in the namespace***
kubectl get pods --all-namespaces ***# List all pods in all namespaces***
kubectl get pods -o wide ***# List all pods in the current namespace, with more details***
kubectl get deployment my-dep ***# List a particular deployment***
kubectl get pods ***# List all pods in the namespace***
kubectl get pod my-pod -o yaml ***# Get a pod's YAML***



Click to see Kubectl Cheatsheet

@Sandip Das

Other Options?

Some useful kubectl alternatives:

K9s: A terminal-based UI for managing Kubernetes clusters.

- **Sample Commands (there is a lot more to it, check here):**
  - k9s: Launch the K9s terminal UI.
  - :q: Quit the UI.
  - :pods: View and manage all pods.

kubectx/kubens: Tools for switching between Kubernetes clusters (kubectx) and namespaces (kubens).

- **Useful Commands:**
  - kubectx: List all available contexts.
  - kubectx <context_name>: Switch to a specific context.
  - kubens <namespace_name>: Switch to a specific namespace.

Lens: A Kubernetes IDE for managing clusters visually.

- **Useful Commands**:
  - **lens**: Launch the Lens application.
  - Navigate through clusters, nodes, pods, and more using the UI.

@Sandip Das

Running Workload

Namespace

What are Namespace?

In Kubernetes, namespaces provides a mechanism for isolating groups of resources within a single cluster.

A virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a resource quota to avoid consuming more than its share of the physical cluster's overall resources.

To create a Namespace, run the below command:
kubectl apply -f simple-namespace.yaml

To know more about Namespace, click here

simple-namespace.yaml

```
apiVersion: v1
kind: Namespce
metadata
   name: analytics
```

@Sandip Das

Pods

What are Pods?

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

It generally refers to one or more containers that should be controlled as a single application.

A pod encapsulates application containers, storage resources, a unique network ID, and other configurations on how to run the containers.

To create a Pod, run the below command:
kubectl apply -f simple-pod.yaml

To know more about Pods, click here

simple-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

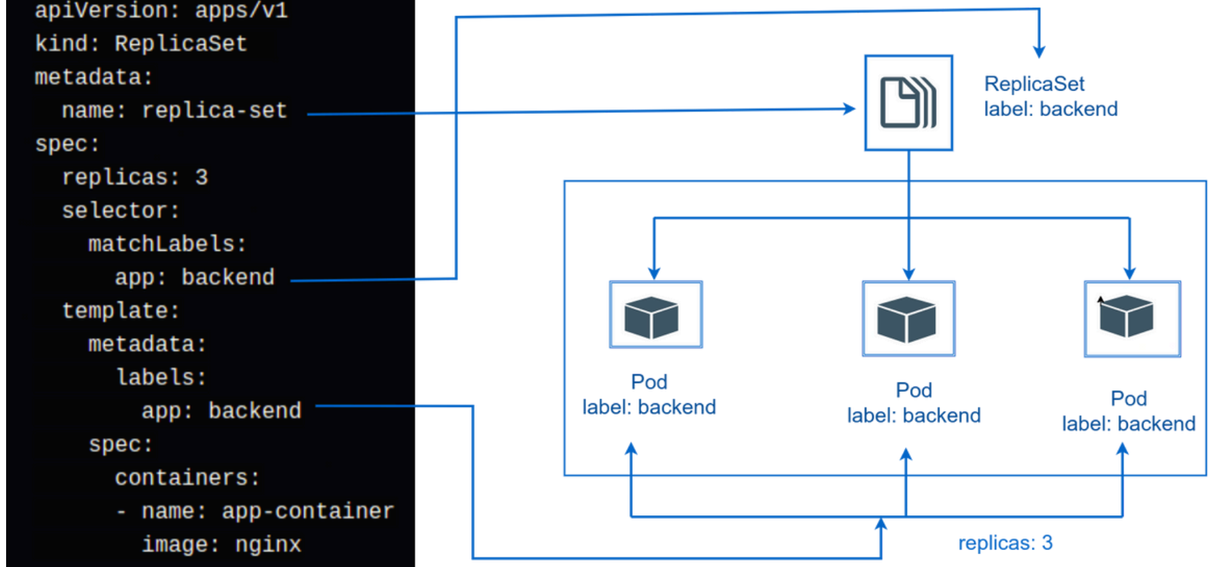
@Sandip Das

ReplicaSet

@Sandip Das

- **A ReplicaSet is an API object that ensures a specified number of pod replicas are maintained. If a Pod fails or is deleted, the ReplicaSet will create a new one to maintain the desired count.**
- **Use Cases: Ensuring high availability of Pods, scaling applications, and maintaining a stable set of replica Pods running at any given time.**

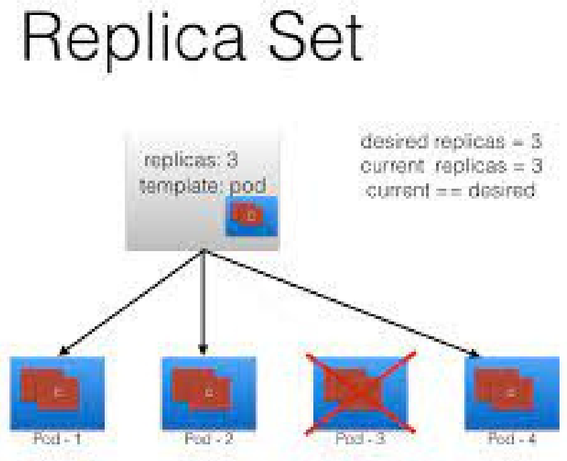
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replica-set
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: app-container
        image: nginx
```



ReplicaSets are an essential tool for managing your Kubernetes cluster's resources and ensuring that your applications are highly available. By defining a desired number of pod replicas, you can ensure that your application is resilient to failures and can scale to handle increased traffic.

One common use case for ReplicaSets is to maintain a stable set of replica Pods running at any given time. This is particularly useful for applications that require a certain level of availability, such as web servers or databases. By defining a ReplicaSet with a desired number of replicas, you can ensure that the necessary resources are always available to handle user requests.

ReplicaSets are also useful for scaling applications horizontally. By increasing the number of replicas in a ReplicaSet, you can handle increased traffic or workloads without overloading individual Pods. This can help ensure that your application remains responsive and available to users.



Deployment

@Sandip Das

What are Deployment?

A Deployment provides declarative updates for Pods and ReplicaSets.

It describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

To create a Deployment, run the below command:
kubectl apply -f simple-deployment.yaml

To know more about Deployment, click here

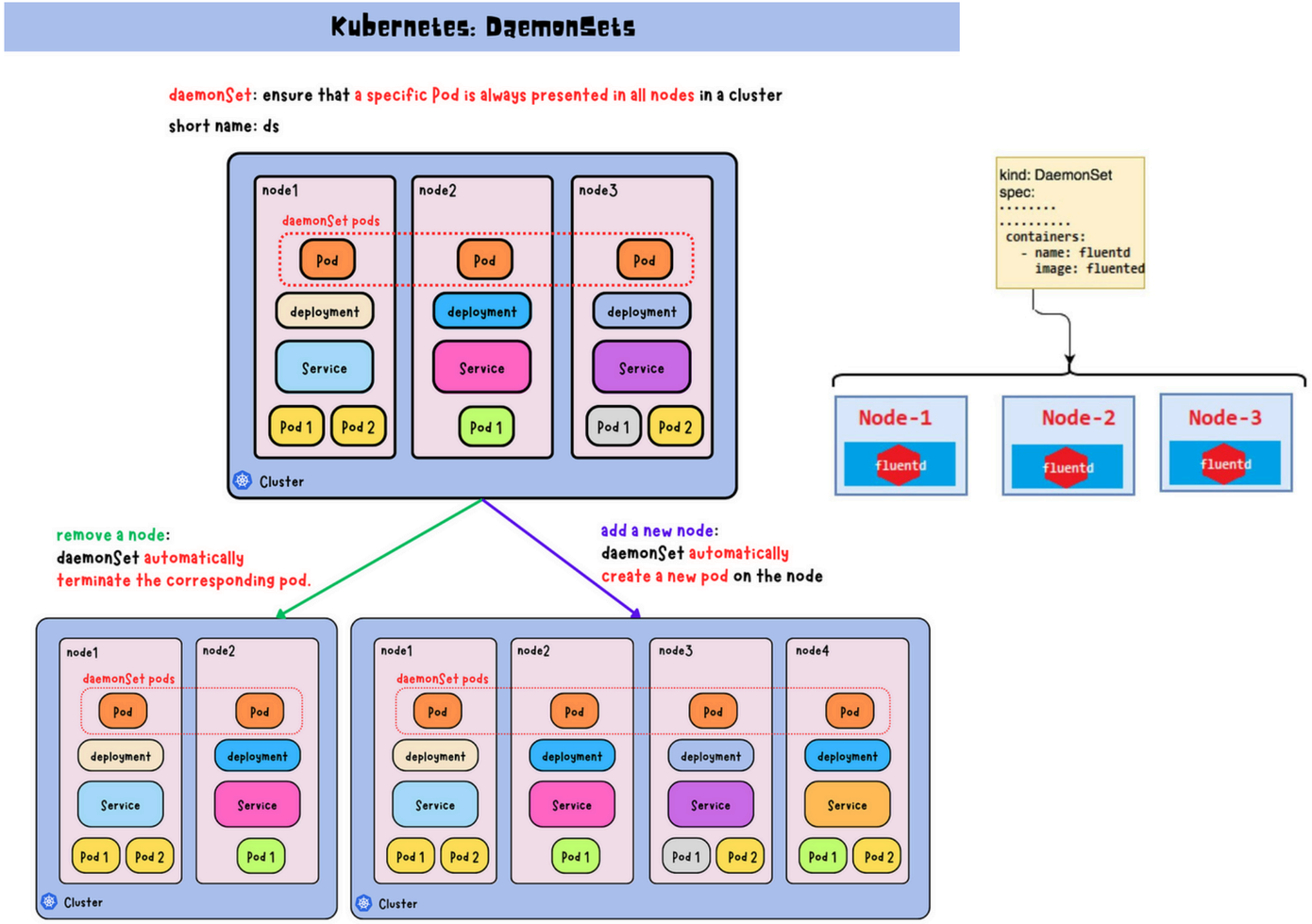
simple-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

DaemonSet

A DaemonSet ensures that all (or some) nodes in the cluster run a copy of a specific Pod. As nodes are added or removed from the cluster, the DaemonSet adjusts the number of Pods to ensure coverage on every node.

Use Cases: Running cluster storage daemons, log collectors, and monitoring agents on every node.



DaemonSet.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.12.0
        env:
        - name: FLUENT_ELASTICSEARCH_HOST
          value: "elasticsearch"
        - name: FLUENT_ELASTICSEARCH_PORT
          value: "9200"
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

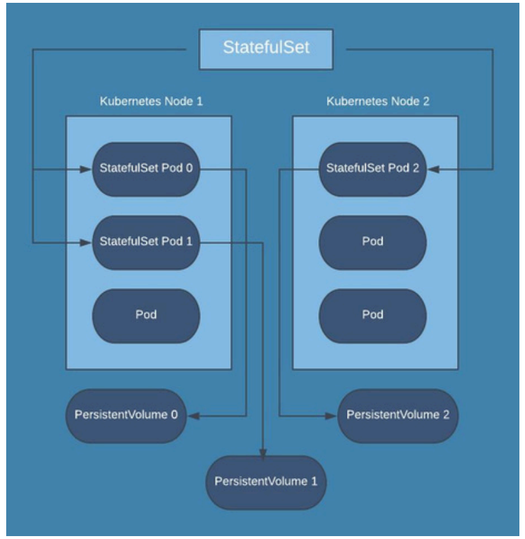
@Sandip Das

StatefulSet

StatefulSet is a Kubernetes workload API object used to manage stateful applications, ensuring that the pods are uniquely identifiable, and maintain the same network identity and storage even after rescheduling.

Use Cases:

1. **Databases**: StatefulSets are ideal for databases like MySQL or PostgreSQL, where each instance needs persistent storage and a stable network identity.
   - **Example**: Deploying a MySQL cluster where each pod retains its own persistent volume.
2. **Distributed Systems**: Used for systems like Apache Kafka or Zookeeper, which require unique identities and stable storage for proper functioning.
   - **Example**: Deploying Kafka brokers where each broker must maintain its own data.
3. **Persistent Storage**: Applications that require stable storage that persists across pod rescheduling, like a content management system (CMS).
   - **Example**: Deploying WordPress with a persistent backend database.



StatefulSet.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: "mysql"
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        ports:
        - containerPort: 3306
          name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "yourpassword"
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-persistent-storage
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

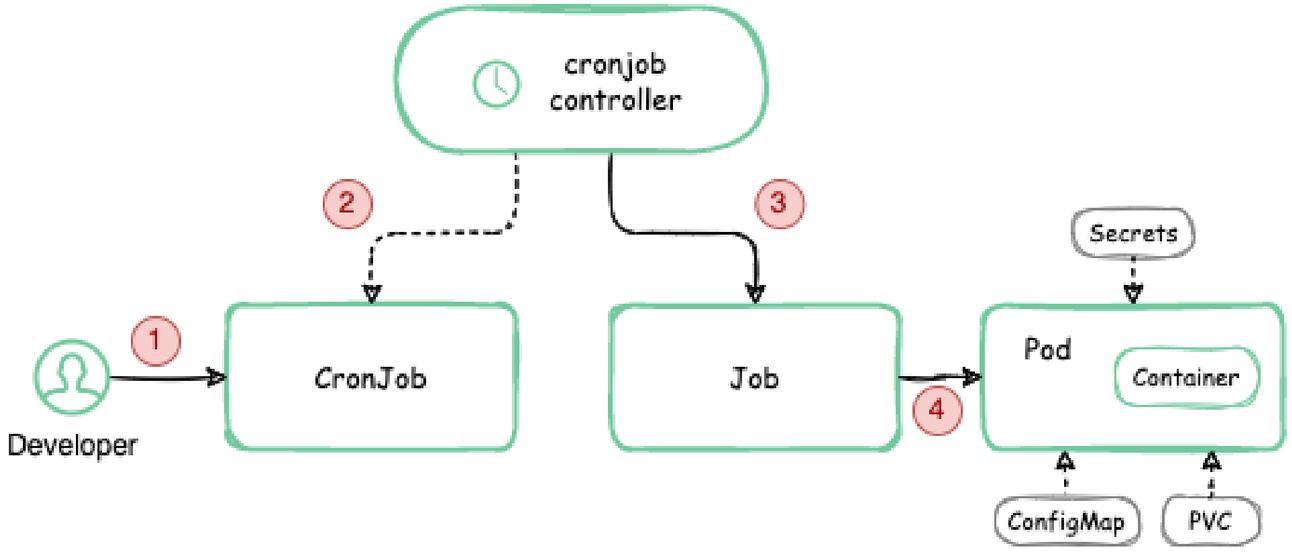
@Sandip Das

Jobs and CronJobs

Jobs:
Definition: A Job creates one or more Pods and ensures that a specified number of them successfully terminate. It's useful for tasks that run to completion, like batch tasks.
Features: Tracks the success of the task, retries failed tasks, and can be set to run multiple tasks in parallel.
CronJobs:
Definition: A CronJob manages time-based Jobs, i.e., it runs jobs on a scheduled basis, much like the cron utility in Unix-like systems.
Features: Specify the schedule in cron format, and the CronJob will run the task at the specified times.

Use Cases:

1. **Data Processing**: Running a batch processing task, like processing logs or generating reports.
   - Example: A Job that processes a log file and generates a summary report.
2. **Database Migration:** Executing database migration scripts during application deployment.
   - Example: A Job that applies database schema changes in a MySQL database.
3. **Backup Operations**: Performing scheduled backups of databases or other data stores.
   - Example: A Job that takes a backup of a PostgreSQL database and stores it in an S3 bucket.



sample_job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migration
spec:
  template:
    metadata:
      name: db-migration
    spec:
      containers:
      - name: migrate
        image: mysql:8.0
        command: ["sh", "-c", "mysql -h your-db-host -u root -
pyourpassword yourdb < /migrations/migrate.sql"]
        volumeMounts:
        - name: migration-scripts
          mountPath: /migrations
      restartPolicy: Never
  backoffLimit: 4
  volumes:
  - name: migration-scripts
    configMap:
      name: migration-config
```


@Sandip Das

ConfigMap

A Kubernetes ConfigMap is a resource used to store non-confidential data in key-value pairs, which can be consumed by pods or used to configure system applications without hard-coding configuration data into the application's source code.

Use Cases:

1. Storing configuration settings like database URLs or external service endpoints.
2. Providing environment-specific configurations to applications, enabling the same application to run differently in development, testing, and production environments.
    -

configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  appSettings.json: |
    {
      "database": "sql.example.com",
      "port": "5432",
      "maxConnections": 100
    }
```

@Sandip Das

Resource Management

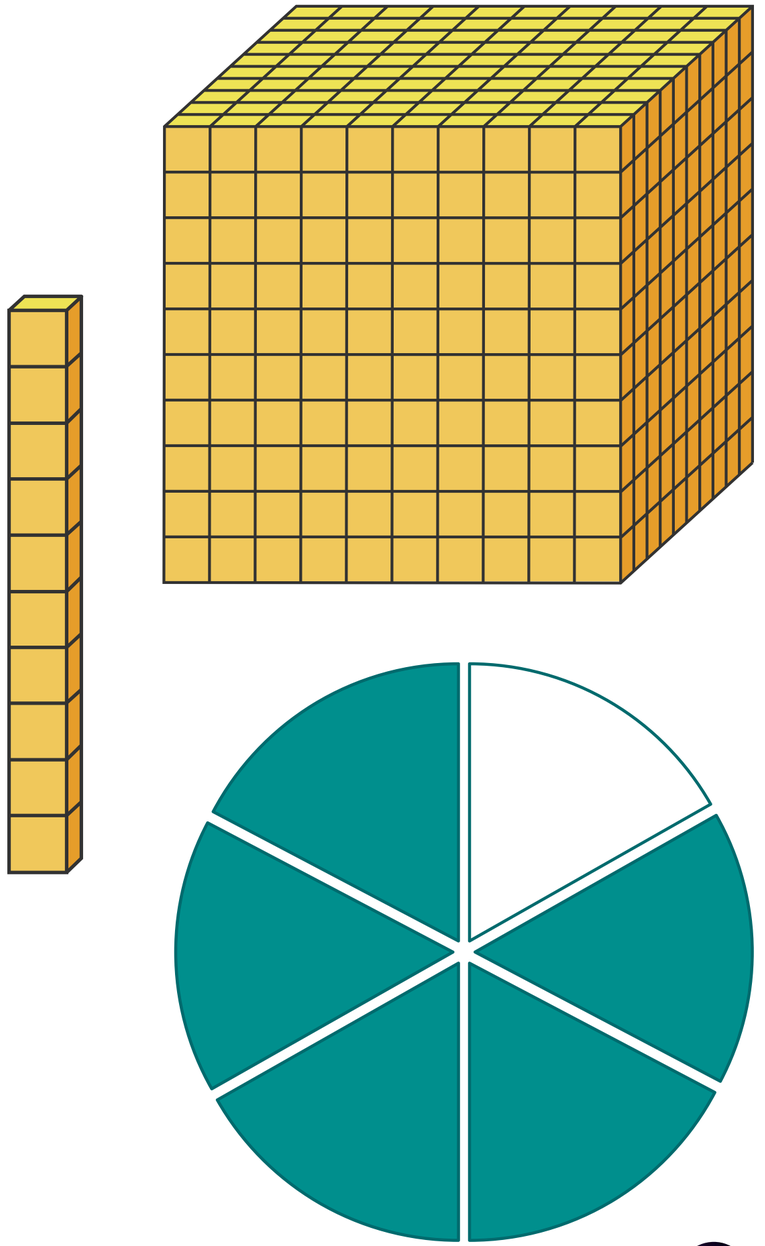
Resource Units

CPU Resources

- Unit: CPU is specified in units of Kubernetes CPUs. One Kubernetes CPU (1 cpu) is equivalent to:
  - 1 AWS vCPU
  - 1 GCP Core
  - 1 Azure vCore
  - 1 Hyperthread on a bare-metal Intel processor with Hyper-Threading
- Formats:
  - Millicpus: Fractions of a CPU can be expressed in decimal (e.g., 0.5) or as millicpus (e.g., 500m where "m" stands for milli). 1 CPU is equivalent to 1000m (millicpus).

Memory Resources

- Unit: Memory is specified in bytes. Kubernetes uses suffixes to represent power-of-two multipliers:
  - Ki: Kibibyte (Ki = 2^10 = 1,024 bytes)
  - Mi: Mebibyte (Mi = 2^20 = 1,048,576 bytes)
  - Gi: Gibibyte (Gi = 2^30 = 1,073,741,824 bytes)
  - Ti: Tebibyte (Ti = 2^40 bytes)
  - Pi: Pebibyte (Pi = 2^50 bytes)
  - Ei: Exbibyte (Ei = 2^60 bytes)
- **Decimal Units**: Kubernetes also supports the SI units for memory (powers of ten), but they are less common:
  - K: Kilobyte (K = 10^3 = 1,000 bytes)
  - M: Megabyte (M = 10^6 = 1,000,000 bytes)
  - G: Gigabyte (G = 10^9 = 1,000,000,000 bytes)
  - T: Terabyte (T = 10^12 bytes)
  - P: Petabyte (P = 10^15 bytes)
  - E: Exabyte (E = 10^18 bytes)



@Sandip Das

Setting Resource Requests and Limits

Resource requests and **limits** are used to control CPU and memory resources that a pod can use.

Requests specify the amount of resources a **container is guaranteed to have** and are used by the scheduler to decide on which node to place the pod.

Limits, on the other hand, **ensure a container never goes above a certain value**, **preventing it from using all of a node's resources**. Here's how you can set resource requests and limits in a pod definition.

request_limit_example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
  - name: example-container
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"  # 250 millicpu or 0.25 CPU
      limits:
        memory: "128Mi"
        cpu: "500m"
```

In this example:

- The CPU request is set at 250 millicpu (or 0.25 of a CPU), and the limit is set at 500 millicpu (or 0.5 of a CPU).
- The memory request is set at 64 MiB, with a limit of 128 MiB.

@Sandip Das

Assigning Quotas to Namespaces

Resource quotas are a way to limit the total amount of memory and CPU resources that can be used by all pods in a namespace.

This is useful for multi-tenant clusters where resource utilization needs to be controlled across different teams or projects.

Here’s an example of how to create a resource quota.

In this example, the resource quota:

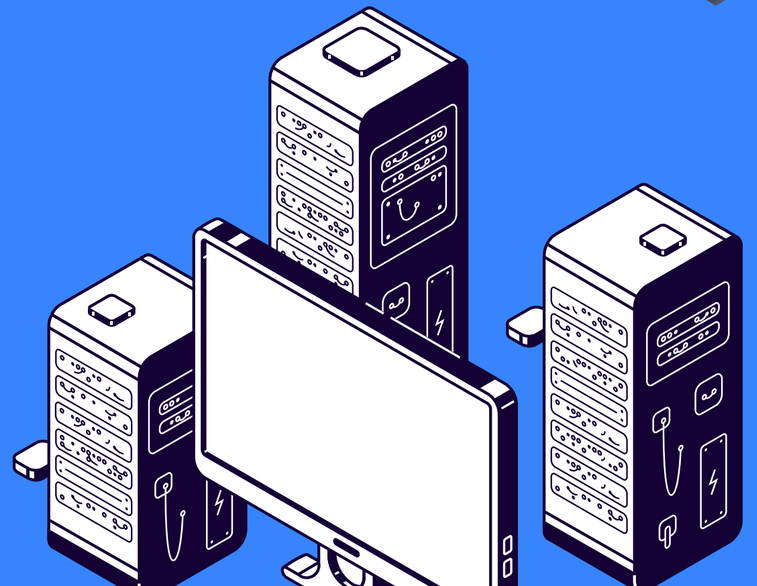
- Limits the total CPU request across all pods in the dev namespace to 1 CPU, with a total limit of 2 CPUs.
- Sets the total memory request limit to 1 GiB and the total memory limit to 2 GiB.
- Restricts the total number of pods to 10 in the dev namespace.

request_limit_namespace_example.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-quota
  namespace: dev
spec:
  hard:
    requests.cpu: "1"    # Total CPU requests across all pods
    requests.memory: 1Gi  # Total memory requests across all
pods
    limits.cpu: "2"     # Total CPU limits across all pods
    limits.memory: 2Gi   # Total memory limits across all pods
    pods: "10"          # Total number of pods that can be created
```

@Sandip Das

Auto Scaling



Horizontal Pod Autoscaler (HPA)

What is HPA?

HPA automatically adjusts the number of pod replicas in a deployment, replicaset, statefulset, or any other pod controller, based on observed CPU utilization (or, with custom metrics support, other application-provided metrics).

How it works?

- **Metrics Monitoring**: HPA collects metrics from either the Kubernetes metrics server (for CPU/memory usage) or custom metrics APIs (for other metrics).
- **Decision Making**: It compares the current metrics value with the target value you set.
- **Scaling Actions**: If the current value exceeds the target, HPA increases the pod replicas. If it's below the target, it reduces the replicas.

hpa.yaml

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-deployment
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

@Sandip Das

Vertical Pod Autoscaler (VPA)

What is VPA?

VPA allocates the optimal amount of CPU and memory to pods, adjusting their resource requests on the fly. It is useful when you are unsure about the resource needs of your application.

How it works?

- **Monitoring:** VPA observes the historical and current resource usage of pods.
- **Recommendation:** Based on this, it calculates and provides recommended CPU and memory settings.
- **Application:** VPA can automatically update the resource requests of pods in running state or only apply these recommendations when pods are restarted, depending on the update policy.

vpa.yaml

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       my-deployment
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        minAllowed:
          cpu: "100m"
          memory: "100Mi"
        maxAllowed:
          cpu: "1"
          memory: "500Mi"
```

@Sandip Das

Cluster Autoscaler

What is CA?

Cluster Autoscaler in Kubernetes is a tool that automatically adjusts the size of a Kubernetes cluster when one of the following conditions is true:

1. There are pods that failed to run in the cluster due to insufficient resources.
2. There are nodes in the cluster that have been underutilized for an extended period of time and their pods can be placed on other existing nodes.

How it works?

- **Scale Out**: If there are pods that cannot be scheduled on any of the existing nodes due to resource constraints, the Cluster Autoscaler communicates with the cloud provider to spin up new nodes.
- **Scale In**: Conversely, if nodes in the cluster are underutilized (based on configurations and thresholds you set) and their workloads can be safely moved to other nodes, Cluster Autoscaler will terminate such nodes to reduce resource wastage and cost.

Key Components

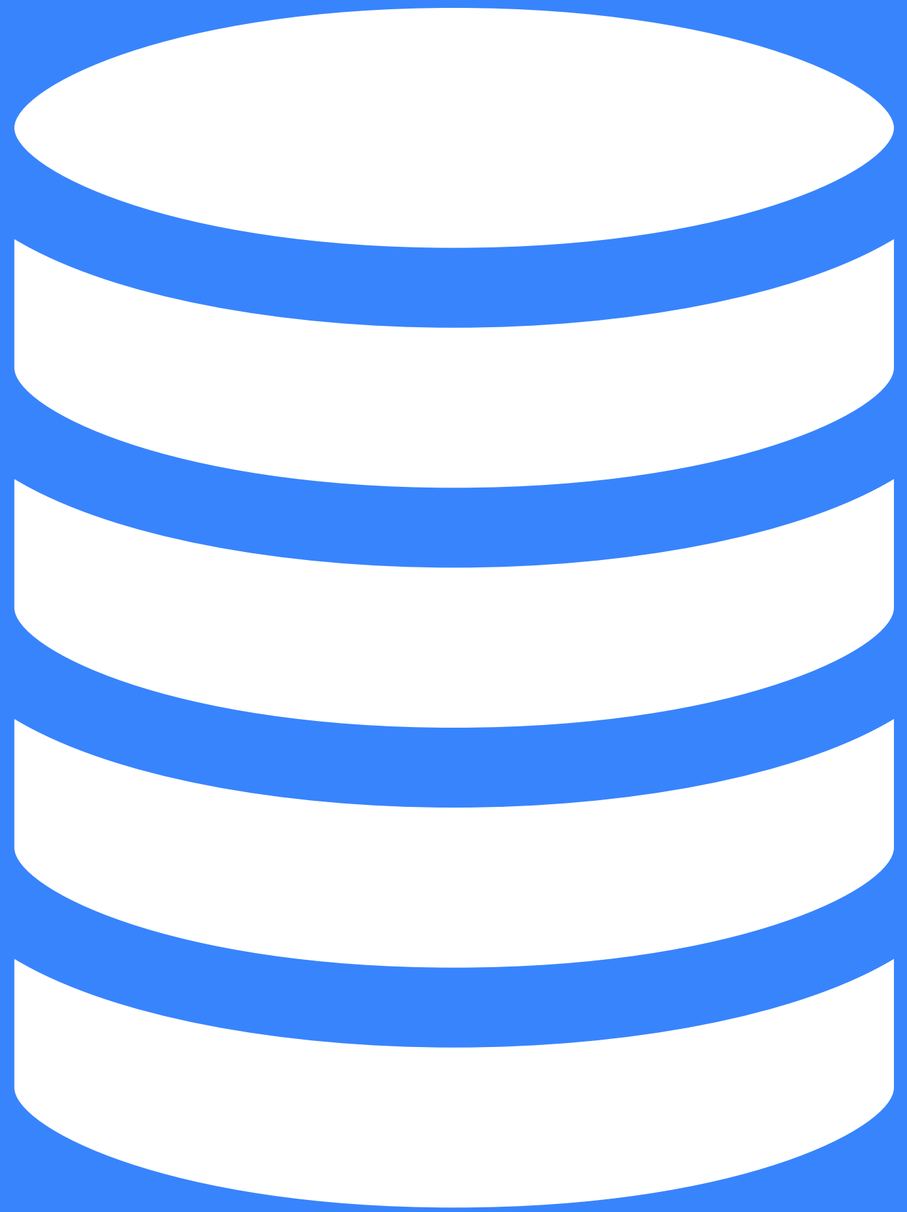
- **Cloud Provider Integration:** Cluster Autoscaler must integrate with your cloud provider to manage the lifecycle of nodes. It supports most major cloud providers, such as AWS, Azure, GCP, and others.
- **Resource Monitoring**: It relies on metrics like CPU and memory usage provided by services like the Kubernetes Metrics Server to make scaling decisions.
- **Pod Disruption Budgets (PDBs)**: It respects PDBs to ensure that the autoscaling process doesn't violate the application's high availability requirements.

ca.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    spec:
      containers:
      - image: k8s.gcr.io/cluster-autoscaler:v1.18.0
        name: cluster-autoscaler
        resources:
          limits:
            cpu: 100m
            memory: 300Mi
        command:
        - ./cluster-autoscaler
        - --v=4
        - --cloud-provider=aws
        - --skip-nodes-with-local-storage=false
        - --expander=least-waste
        - --nodes=1:10:my-node-group
        env:
          - name: AWS_REGION
            value: "us-west-2"
          - name: AWS_ACCESS_KEY_ID
            valueFrom:
              secretKeyRef:
                name: aws-secret
                key: access-key
          - name: AWS_SECRET_ACCESS_KEY
            valueFrom:
              secretKeyRef:
                name: aws-secret
                key: secret-key
        volumeMounts:
        - name: ssl-certs
          mountPath: /etc/ssl/certs/ca-certificates.crt
          readOnly: true
      volumes:
      - name: ssl-certs
        hostPath:
          path: /etc/ssl/certs/ca-certificates.crt
```

@Sandip Das

Storage



Volume

What are Volume?

Volume in a simple sense is a directory containing data, just similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod.
Using Volume Kubernetes guarantees data is preserved across container restarts

There are many types of volumes: emptyDir, hostPath, gcePersistentDisk, awsElasticBlockStore, nfs, iscsi, flocker, glusterfs, rbd, cephfs, gitRepo, secret, persistentVolumeClaim, downwardAPI, azureDiskVolume,

To create a Volume, run the below command:
kubectl apply -f simple-volume.yaml

To know more about Volume, click here

simple-volume.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
   name: pv0001
   labels:
      type: local
spec:
   capacity:
      storage: 10Gi
   accessModes:
      - ReadWriteOnce
      hostPath:
         path: "/tmp/data01"
```

@Sandip Das

Persistent Volume Claim (PVC)

What are PVC?

A PVC is a request for storage by a user in Kubernetes, allowing pods to claim persistent storage. It abstracts storage resources and binds to a Persistent Volume (PV) that meets the requested storage size and access modes.

Use Cases?

Dynamic storage provisioning, data persistence for applications like databases, and shared storage across pods.

Useful Commands

Get All PVCs:

kubectl get pvc

Get PVC Details:

kubectl describe pvc <pvc-name>

Delete a PVC:

kubectl delete pvc <pvc-name>

Get PVCs in All Namespaces:

kubectl get pvc --all-namespaces

simple-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

@Sandip Das

Persistent Volume (PV)

What are PVC?

A PV is a piece of storage in the cluster provisioned by an administrator or dynamically by a storage class, which provides storage resources to a PVC.

Use Cases?

Providing durable storage to applications, supporting various storage backends like NFS, AWS EBS, or GCE PD, and facilitating data persistence.

Useful Commands

Get All PVs:

kubectl get pv

Get PV Details:

kubectl describe pv <pv-name>

Delete a PV:

kubectl delete pv <pv-name>

Watch PVS:

kubectl get pv --watch

simple-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /path/to/nfs
    server: nfs-server.example.com
```

@Sandip Das

Persistent Volume (PV) Key Conceps

Status

- **Available**: The PV is available for binding to a Persistent Volume Claim (PVC). It is not yet bound to any PVC and is ready to be claimed.
- **Bound**: The PV is bound to a PVC. This indicates that the PV is in use and has been successfully claimed by a PVC.
- **Released**: The PVC that was bound to this PV has been deleted, but the PV is not yet reclaimed by the cluster. The data on the PV might still be present, depending on the reclaim policy.
- **Failed**:The PV has failed to be properly released or reclaimed. This state usually requires manual intervention to clean up or troubleshoot the issue.
- **Terminating**: The PV is in the process of being deleted from the cluster. This state is visible when a delete command has been issued for the PV but it has not yet completed the deletion process.
- **Released (Soft Deleted)**:When a PVC is deleted, the PV enters a "Released" state, but if the reclaim policy is set to "Delete," the PV might be cleaned up by the storage backend, marking it as soft deleted before the actual deletion happens.

Reclaim Policies

The reclaim policy of a PV dictates what happens to the PV after the PVC bound to it is deleted:

- **Retain**: The PV retains its data after the PVC is deleted. It needs to be manually cleaned up or reused.
- **Delete**: The PV and its associated storage are automatically deleted when the PVC is deleted.
- **Recycle**: The PV is scrubbed (data wiped) and made available for a new claim. This policy is deprecated and not commonly used in modern Kubernetes setups.

@Sandip Das

CSI (Container Storage Interface)

What are CSI?

The Container Storage Interface (CSI) is an initiative to unify the storage interface of a Container Orchestration System (like Kubernetes) with various storage backends. Prior to CSI, Kubernetes had provider-specific plugins, which led to maintenance and scalability issues. CSI abstracts these details away, allowing storage providers to create plugins that work across multiple orchestration systems without requiring changes to the core code of Kubernetes.

CSI Drivers

- **AWS EBS CSI Driver**: Supports the integration of AWS Elastic Block Store (EBS) into Kubernetes, providing robust and scalable block storage for stateful applications.
- **Google Persistent Disk CSI Driver**: Facilitates the use of Google Compute Engine Persistent Disk as a native storage option in Kubernetes, supporting both standard and SSD-backed storage.
- **Azure Disk CSI Driver**: Allows Azure Disks and Azure File shares to be used as persistent volumes in Kubernetes, supporting a variety of disk types and configurations.
- **NFS CSI Driver:** Provides a way to use NFS (Network File System) resources as persistent volumes in Kubernetes. This is widely used for shared filesystems across multiple pods.
- **OpenEBS**: A leading open-source project that provides a variety of storage solutions for Kubernetes, including dynamic local PV provisioning and high availability.
- **Ceph CSI:** Enables Ceph RBD (Block Storage) and CephFS (File System) to be used as persistent storage in Kubernetes environments, supporting features like snapshotting and encryption.
- **VMware vSphere CSI Drive**r: Integrates VMware's vSphere storage, including VMware-specific storage capabilities like policy-based management and data-at-rest encryption, into Kubernetes.

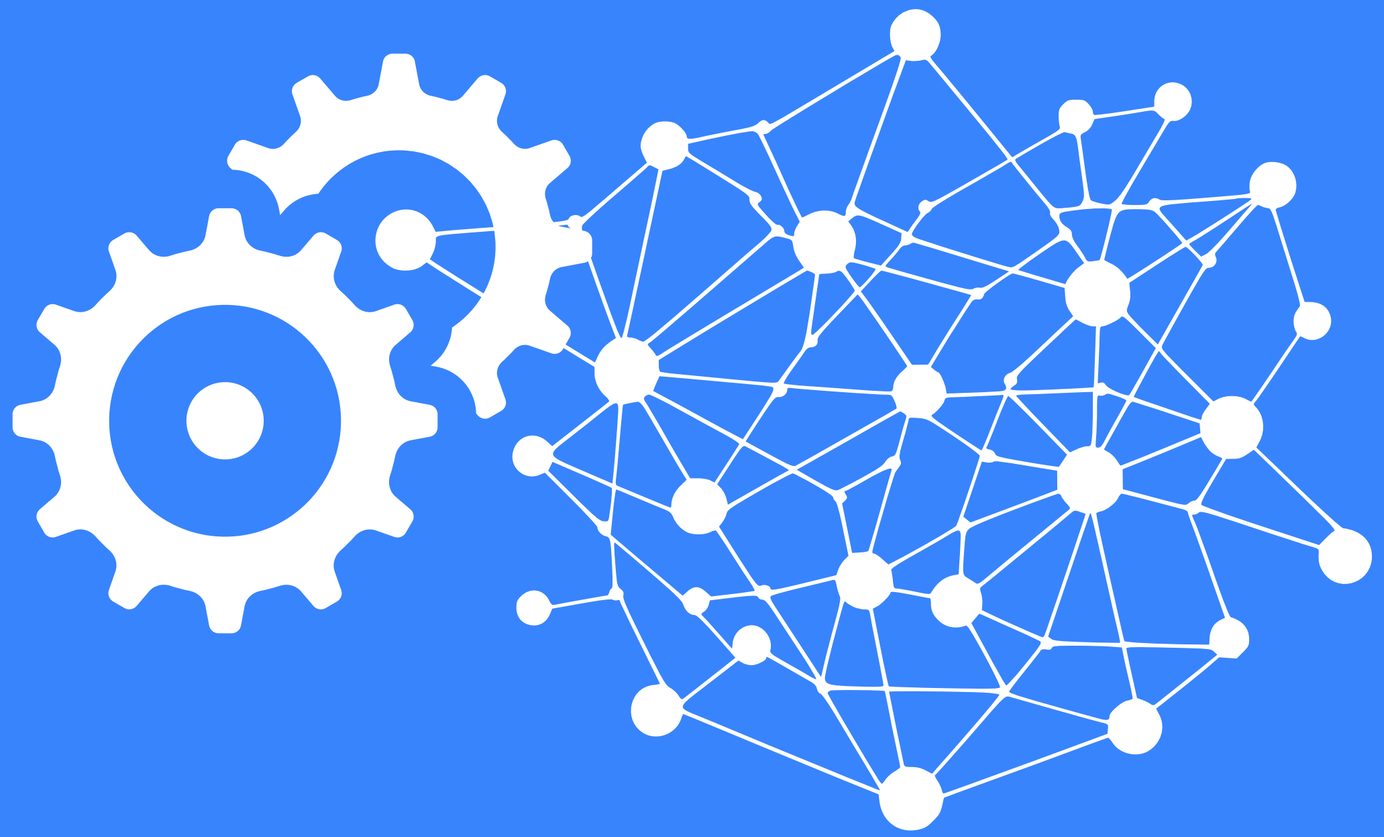
```
apiVersion: v1
kind: List
items:

# Storage Class for AWS EBS gp3
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    name: ebs-gp3-sc
  provisioner: ebs.csi.aws.com
  parameters:
    type: gp3
    fsType: ext4
    throughput: "125"  # MiB/s, adjust according to needs
    iops: "3000"      # IOPS, adjust according to needs
  reclaimPolicy: Delete
  allowVolumeExpansion: true
  volumeBindingMode: WaitForFirstConsumer

# Persistent Volume Claim
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: ebs-gp3-pvc
  spec:
    accessModes:
      - ReadWriteOnce
    storageClassName: ebs-gp3-sc
    resources:
      requests:
        storage: 10Gi

# Pod using the PVC
- apiVersion: v1
  kind: Pod
  metadata:
    name: mygp3pod
  spec:
    containers:
    - name: nginx-container
      image: nginx
      volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: myvolume
    volumes:
    - name: myvolume
      persistentVolumeClaim:
        claimName: ebs-gp3-pvc
```

@Sandip Das



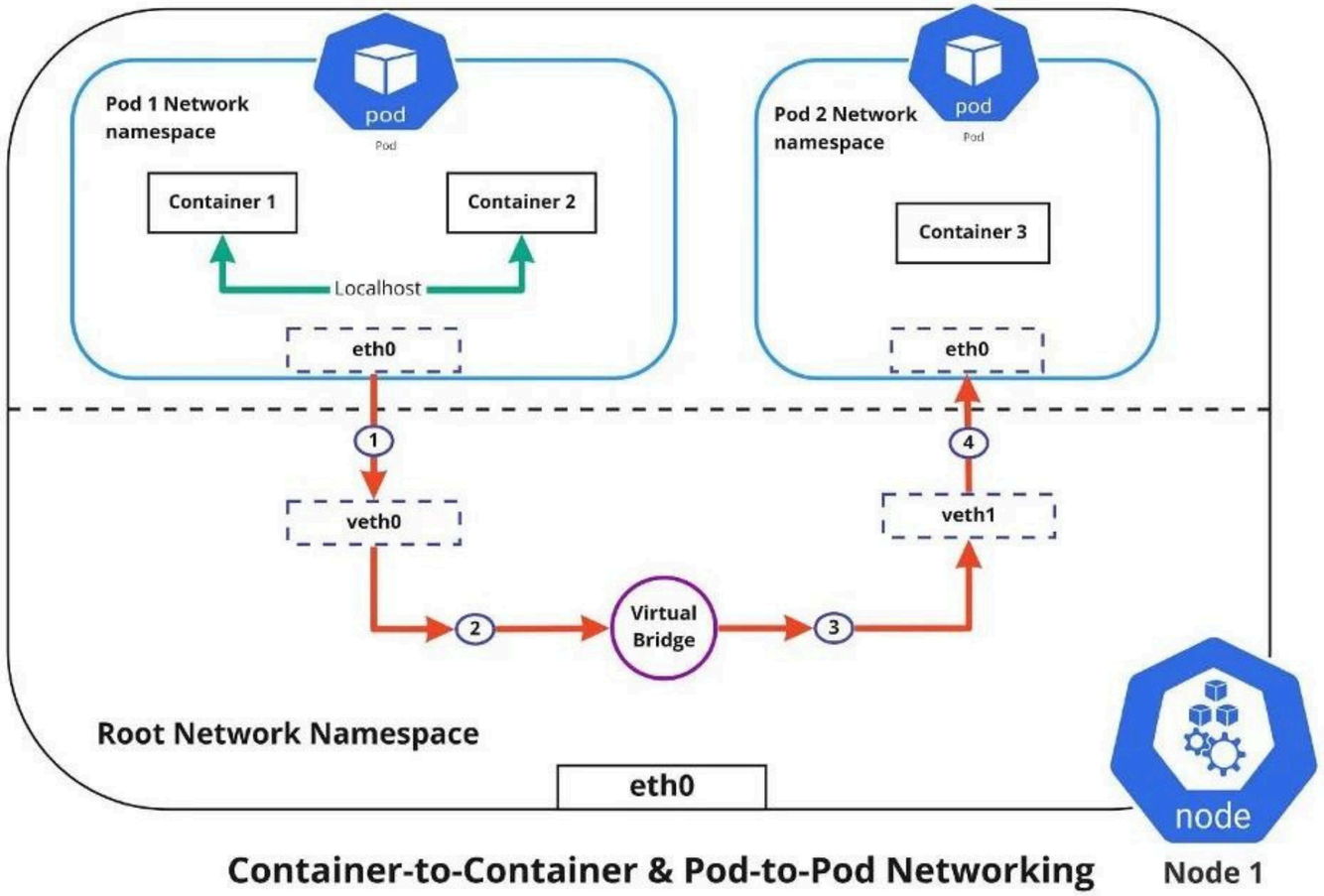
Services and Networking

Pod Networking

- **In Kubernetes, every Pod is assigned a unique IP address within the cluster, allowing them to communicate with each other. This internal network is established and managed by networking plugins.**

- **Use Cases: Ensuring that Pods can communicate with each other and with other resources in the cluster, regardless of which node they reside on**

This unique IP address assignment is particularly useful when it comes to ensuring that Pods can communicate with each other and with other resources in the cluster, regardless of which node they reside on. This is because these IP addresses are reachable from any other node in the cluster, making communication between Pods and other resources seamless. Additionally, this internal network is established and managed by networking plugins, which provide a high level of flexibility and customization to meet the specific needs of each cluster. Whether you're running a small, single-node cluster or a large, multi-node cluster, Kubernetes provides the tools you need to ensure that your Pods can communicate effectively and efficiently.



@Sandip Das

Services

What are Services?

It's an abstract way to expose an application running on a set of Pods as a network service.

Pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive forever. Instead, a service represents a logical set of pods and acts as a gateway, allowing to send requests to the service without needing to keep track of which physical pods actually make up the service.

Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

To create a Service, run the below command:
kubectl apply -f simple-service.yaml

To know more about services, Click here

simple-service.yaml

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

@Sandip Das

Ingress

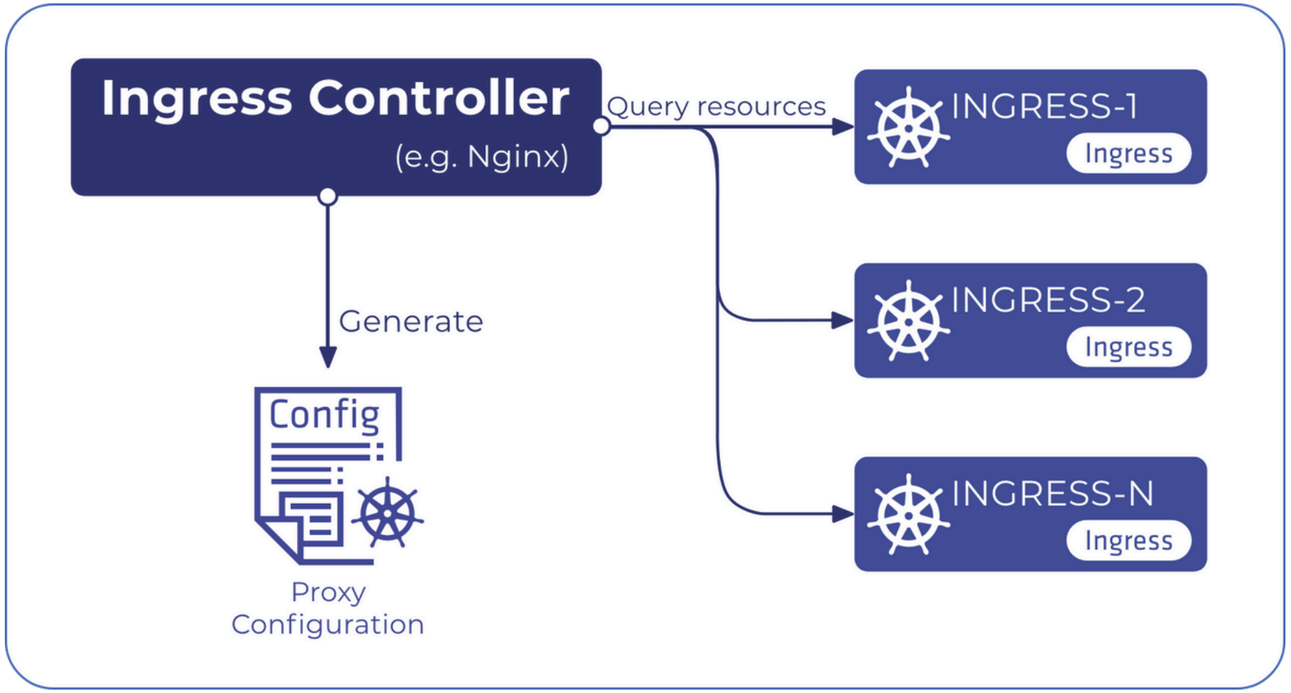
In Kubernetes, an Ingress is an API object that manages external access to the services in a cluster, typically HTTP and HTTPS. It provides HTTP and HTTPS routing to services based on a set of rules, including hostnames and paths.

Use Cases: Exposing multiple services under a single IP address, SSL/TLS termination, name-based virtual hosting, and more.

In addition to managing external access to services, Ingress also allows for load balancing and can be configured to work with different load balancing algorithms. It also enables the use of multiple SSL certificates and provides a way to manage them centrally.

Another advantage of using Ingress is the ease of managing and configuring routes for traffic. It allows for easy modification and addition of routing rules without the need to modify individual services or the load balancer.

Overall, Ingress is a powerful tool in managing external access and routing for Kubernetes services. Its flexibility and ease of use make it a popular choice among developers and system administrators alike.



@Sandip Das

Kubernetes Virtual Networking

Kubernetes virtual networks

Kubernetes virtual networks provide isolated and configurable network environments for pods and services within a cluster, facilitating secure and efficient intercommunication.

Example

Here's a list of popular tools used in Kubernetes for various networking functions:

1. **CNI Plugins (Pod Networking):**
   - **Flannel**: A simple and easy-to-configure layer 3 network fabric designed for Kubernetes.
   - **Weave Net**: Provides a resilient and simple to use network for Kubernetes and Docker container environments.
2. **Network Policies:**
   - **Kube-router:** Provides pod networking, load balancing, and network policy in Kubernetes using standard Linux networking tools like iptables and IPVS.
3. **Service Networking:**
   - **MetalLB**: A load balancer for Kubernetes, typically used when running on bare-metal to provide LoadBalancer type services.
   - **Traefik**: Not only a reverse proxy and load balancer but can also be used as an Ingress controller to manage access to services.
4. **Ingress Controllers:**
   - **NGINX Ingress Controller**: Manages access to services inside a Kubernetes cluster using NGINX as a reverse proxy and load balancer.
   - **HAProxy Ingress Controller**: An Ingress controller that uses HAProxy to manage external access to HTTP services within a Kubernetes cluster.
5. **Overlay Networks:**
   - **VXLAN (Virtual Extensible LAN)**: Often used as an overlay network protocol to create a logical network for pods across multiple nodes.
   - **Istio**: While primarily known for service mesh capabilities, Istio also facilitates an overlay network for secure communication between microservices.

@Sandip Das

Security

Kubernetes Secrets

Kubernetes secrets are used to store and manage sensitive information such as passwords, OAuth tokens, and SSH keys. They help keep such information safe by ensuring it's not hard-coded in your application or stored in the source code. Below are some examples of how to create and use Kubernetes secrets.

1. Creating a Secret:

```
kubectl create secret generic my-secret --from-literal=username=my-app-user --from-literal=password=s3cr3tpassw0rd
```

2. Create Using a YAML File

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: bXktYXBwLXVzZXI=  # base64 encoded value of 'my-app-user'
  password: czNjcjN0cGFzc3cwcmQ=  # base64 encoded value of 's3cr3tpassw0rd'
kubectl apply -f my-secret.yaml
```

3. List secrets

```
kubectl get secrets my-secret -o yaml
```

4. Decoding Secret Values

```
kubectl get secret my-secret -o jsonpath="{.data.username}" | base64 --decode
```

5. Delete secrets:

```
kubectl delete secret my-secret
```

Using Secrets as Environment Variables
pod.yaml

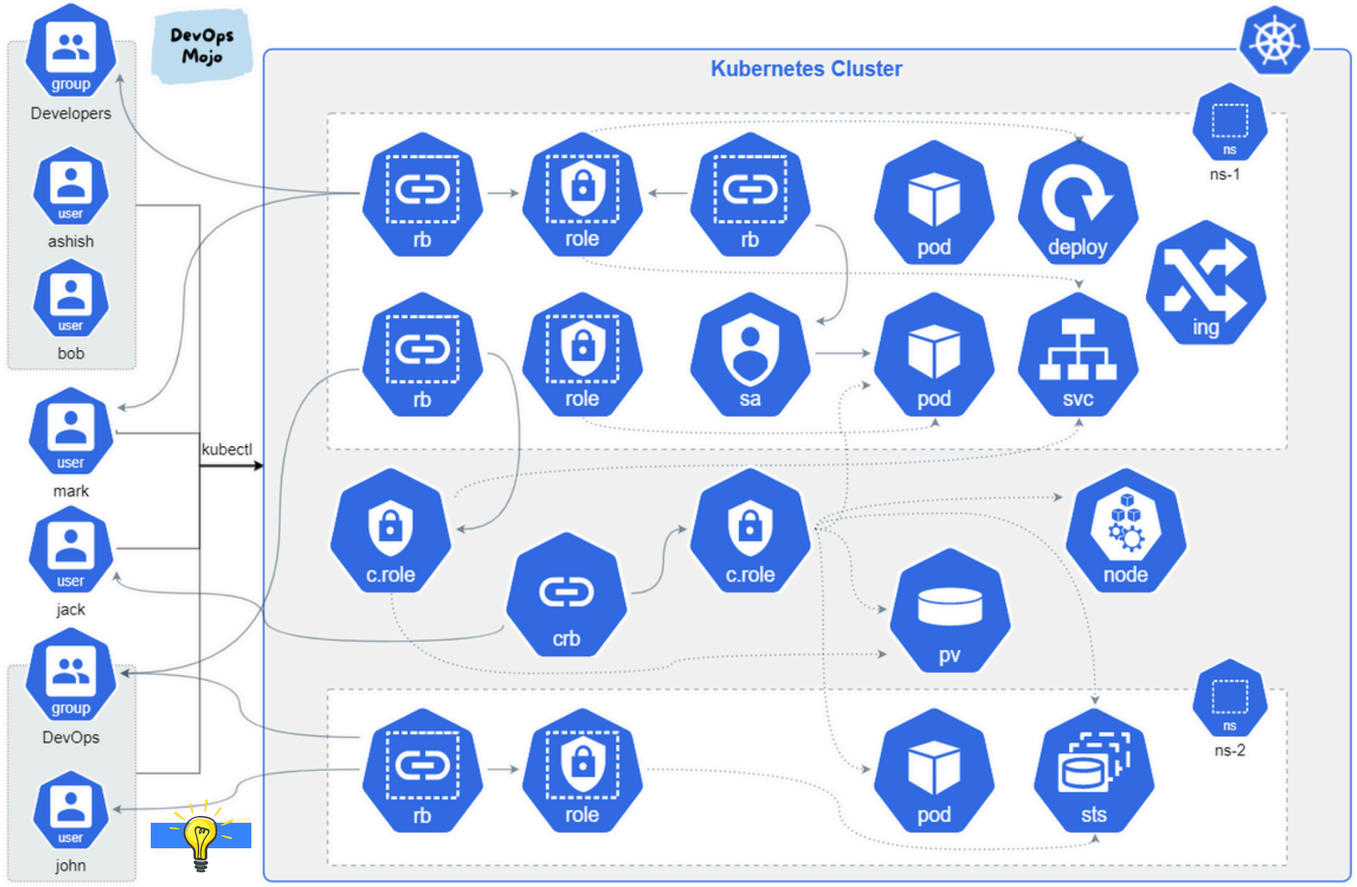
```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    env:
    - name: USERNAME
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: username
    - name: PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: password
```

@Sandip Das

Role-Based Access Control (RBAC)

RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise.

- Features:
  - Roles: Defines permissions on resources (like Pods, Services). Can be namespaced or cluster-wide (ClusterRoles).
  - RoleBindings: Associates roles with users or groups. Can be namespaced or cluster-wide (ClusterRoleBindings).
  - Allows fine-grained access control to Kubernetes API resources.



@Sandip Das

Kubernetes Network Security

Kubernetes network security involves protecting the communication channels and data within a Kubernetes cluster from unauthorized access and threats. It ensures that only legitimate traffic can flow between containers, services, and the outside world.

Here are some useful cloud-native tools for Kubernetes network security:

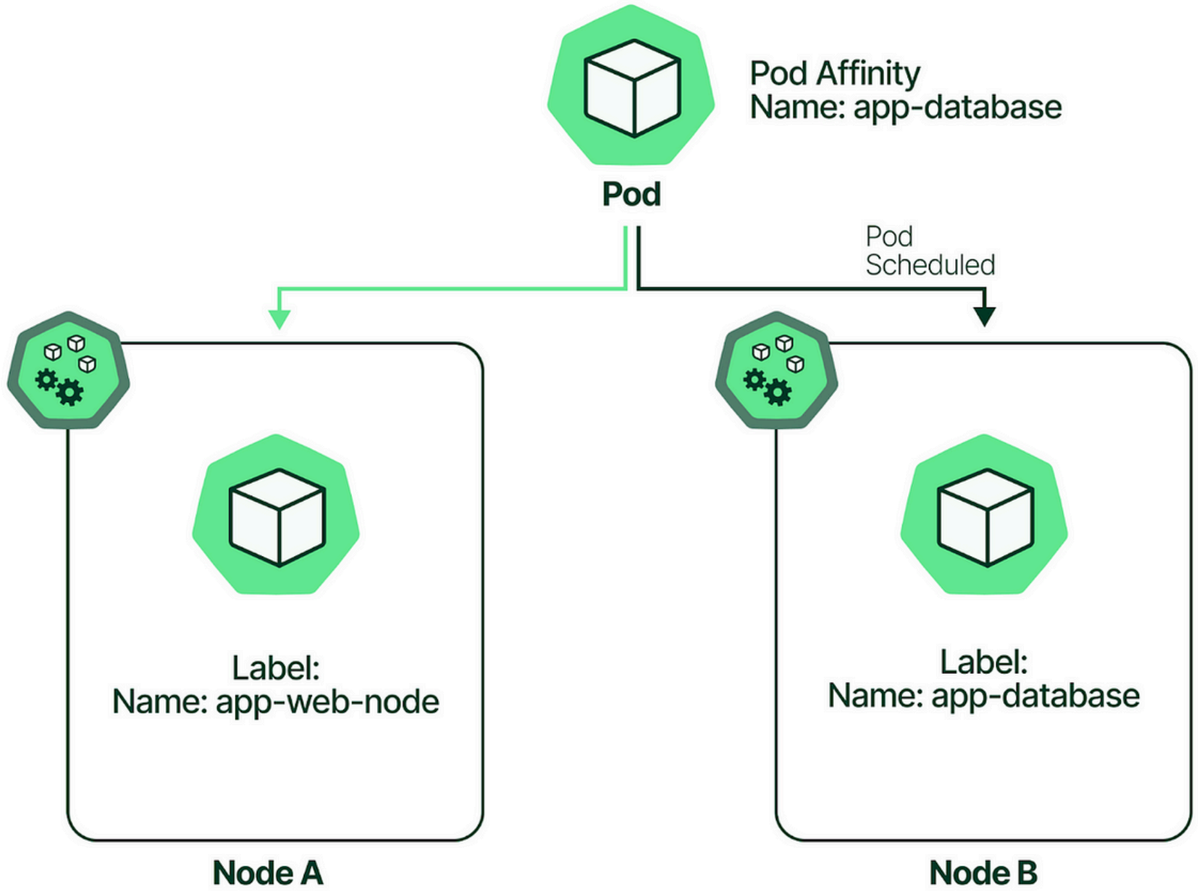
1. **Cilium**: Leverages eBPF technology to provide highly scalable network security policies, API-aware security, and visibility.
2. **Tigera Secure**: Extends Calico with enterprise-grade security features for compliance and threat detection.
3. **Linkerd**: Another service mesh that provides automatic mTLS (mutual TLS) to secure traffic between services.
4. **Kube-router**: Provides network routing, firewalling, and load balancing capabilities using standard Linux networking tools.
5. **Aqua Security**: Focuses on security throughout the application lifecycle, from development to production, specifically tailored for containerized environments.
6. **Falco**: A cloud-native runtime security project to detect and alert on anomalous activities in your applications and containers.
7. **Opa (Open Policy Agent):** Provides a high-level declarative language to specify security policies and enforces those policies across the Kubernetes stack.

@Sandip Das

Scheduling

Affinity and Anti-Affinity

- **Affinity**:
  - Allows you to specify rules about how pods should be scheduled relative to other pods.
  - **Features**: Can be used to place pods on the same node (or spread them out) based on labels and conditions.
- **Anti-Affinity:**
  - Ensures that two pods don't end up on the same node.
  - **Features**: Useful for ensuring high availability by spreading replicas of a service across nodes or racks.

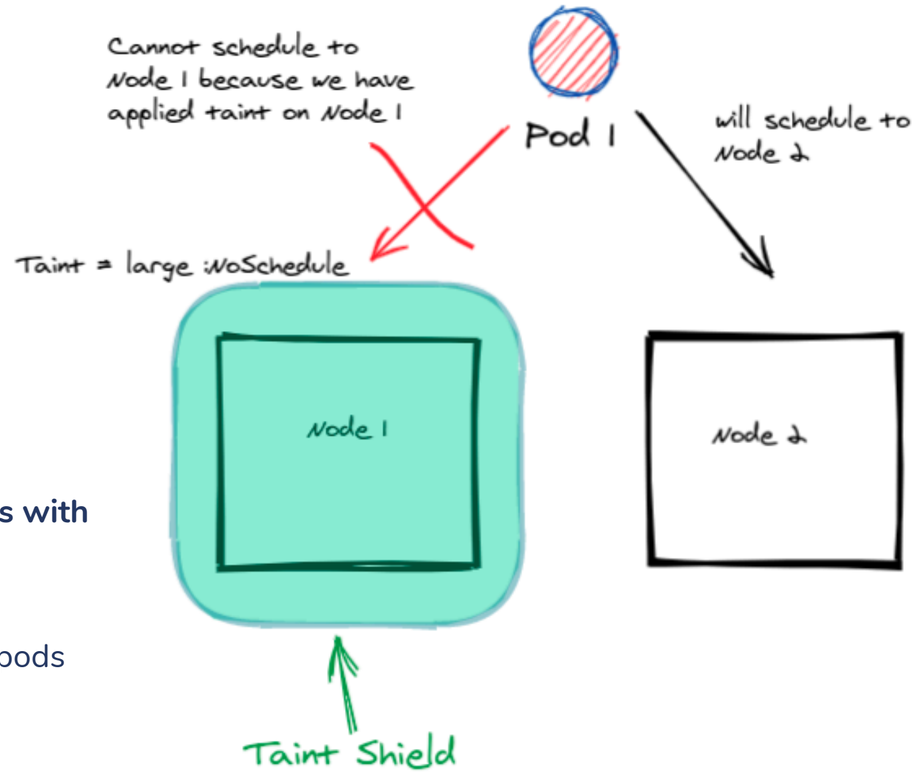


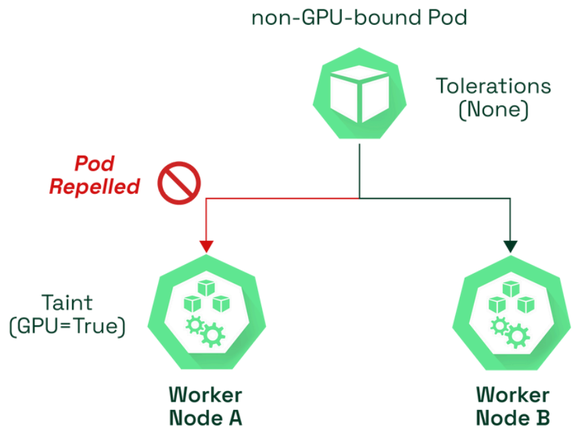
Taints and Tolerations

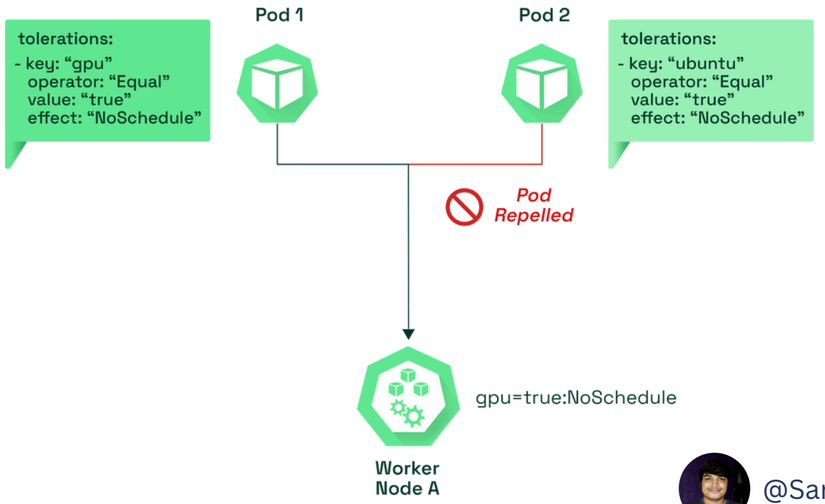
Taints:
Taints allow a node to "**repel**" **a set of pods based on key-value pairs.**
Useful for designating nodes for specific purposes, like dedicated nodes or GPU nodes.

Tolerations:
Tolerations are applied to **pods and allow (but do not require) the pods to schedule onto nodes with matching taints.**

Works in conjunction with taints to ensure pods are scheduled appropriately. For example, only pods with a specific toleration can be scheduled on a node with a specific taint.







@Sandip Das

Annotations

In Kubernetes, **annotations are a mechanism to attach arbitrary non-identifying metadata to objects**. Unlike labels, which are used to identify and select objects, annotations are not used to select and find objects. Annotations can be used to store and retrieve additional information about Kubernetes objects, which can be beneficial for various purposes.

Key Characteristics of Annotations:

1. **Arbitrary Metadata**: Annotations can store any data, including structured data, as long as it can be serialized into a string format.
2. **Non-identifying**: While labels are used to identify and group objects, annotations are not meant for identifying and selecting objects.
3. **Not for Filtering**: Unlike labels, annotations are not used by Kubernetes when filtering objects.
4. **Tool-specific Fields**: Annotations are often used to store fields that are specific to certain tools or systems. For example, a tool might use annotations to store a timestamp of the last backup.

Common Use Cases for Annotations:

1. **Build**, release, or image information: Storing version details, release IDs, or image hashes.
2. **Pointers to logging, monitoring, analytics, or audit repositories:** For example, a trace identifier.
3. **Client library or tool information**: Storing data about tools or libraries interacting with an object.
4. **Debugging purposes**: Annotations can be used to store debugging information.
5. **Other information:** Any other information that might be helpful but is not suitable for labels.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    example.com/icon-url: "https://example.com/icon.png"
    example.com/some-note: "This is an example annotation"
spec:
  containers:
  - name: sample-container
    image: nginx
```

Annotations, like labels, are key-value pairs. Here's an example of how you might set an annotation when creating or modifying a Kubernetes object:

@Sandip Das

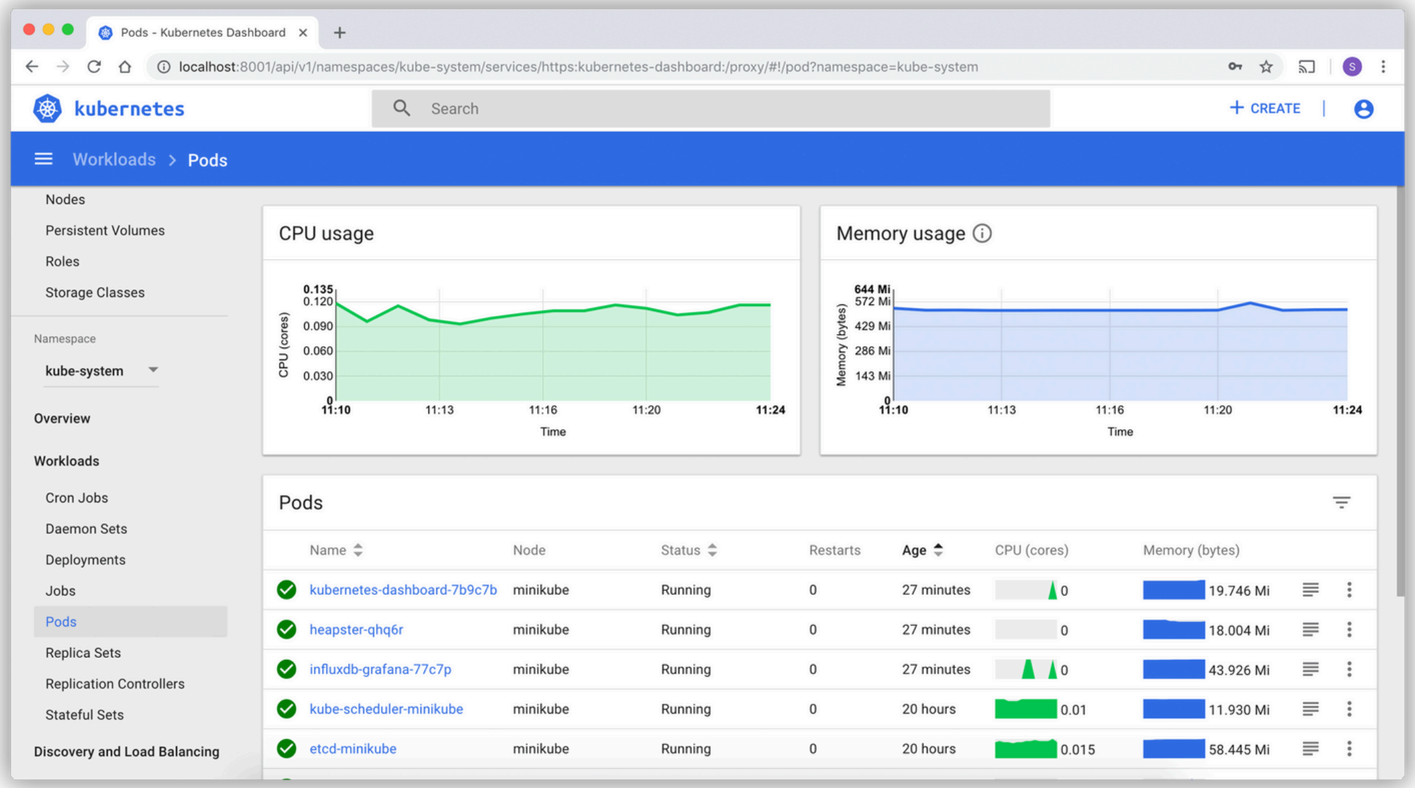
Monitoring

Kubernetes Dashboard

The Kubernetes Dashboard is a web-based user interface that allows users to manage and monitor Kubernetes clusters. It provides an overview of applications running in the cluster, as well as detailed information about cluster resources.

Use Cases: Visualizing cluster state, monitoring cluster resources, managing workloads, and troubleshooting issues.

The Kubernetes Dashboard is an essential tool for managing and monitoring Kubernetes clusters. It offers a graphical representation of the cluster's state and provides users with detailed information about the resources being used. The dashboard is also perfect for managing workloads and troubleshooting issues that may arise. With its user-friendly interface, even novice users can easily navigate the dashboard and quickly get up to speed with what's happening in their cluster. Whether you're a developer, system administrator, or just someone interested in learning more about Kubernetes, the Dashboard is a must-have tool in your arsenal.



@Sandip Das

Useful Logging & Monitoring Tools

Monitoring Tools:

Prometheus: An open-source systems monitoring and alerting toolkit. Ideal for monitoring Kubernetes clusters, providing real-time metrics, and integrating with Grafana for visualization.

Thanos: A highly available Prometheus setup with long-term storage capabilities. Used for scalable and durable monitoring across multiple Kubernetes clusters with Prometheus.

Grafana: An open-source platform for monitoring and observability. Used for visualizing time-series data from Prometheus, InfluxDB, and other data sources, with customizable dashboards.

Nagios: A widely used open-source monitoring system. Suitable for monitoring system health, network devices, and services with alerting capabilities.

Zabbix: An enterprise-grade open-source monitoring solution. Used for monitoring servers, networks, applications, and cloud services with built-in notification and alerting.

Datadog:
A cloud-based monitoring and analytics platform. Ideal for full-stack monitoring, including infrastructure, applications, and logs, with real-time analytics.

New Relic:
A cloud-based observability platform. Used for application performance monitoring (APM), infrastructure monitoring, and end-user experience tracking.

Sensu: A flexible monitoring pipeline with a focus on observability. Suitable for monitoring infrastructure and applications across hybrid cloud environments with automated response.

Logging Tools:

Elasticsearch: A distributed, RESTful search and analytics engine. Often used as the backbone for log analysis and searching in combination with Logstash and Kibana (ELK Stack).

Logstash: An open-source server-side data processing pipeline. Used for ingesting, transforming, and shipping logs and events from various sources into Elasticsearch.

Kibana: A data visualization and exploration tool for Elasticsearch. It provides powerful visualizations and dashboards for log data, helping with analysis and monitoring.

Fluentd: An open-source data collector for unified logging. Used for collecting logs from various sources and forwarding them to multiple outputs, including Elasticsearch and cloud storage.

Graylog: An open-source log management platform. Used for aggregating, indexing, and analyzing log data, with real-time search and dashboards.

Splunk: A comprehensive platform for searching, monitoring, and analyzing machine-generated data. Often used for enterprise-grade log management, security event monitoring, and big data analytics.

Papertrail: A cloud-based log management service. Ideal for aggregating logs from various sources for real-time viewing, searching, and alerts.

S**umo Logic:** A cloud-native, machine data analytics service. Used for real-time log management and security monitoring across hybrid cloud environments.

@Sandip Das

Thank You

@LearnTechWithSandip

SUBSCRIBE

For staying till the end

Follow on: Sandip Das