**Kubernetes Overview**

**What is Kubernetes?**

**Kubernetes, also known as "K8s", is an open-source container orchestration platform developed by Google. It is designed to automate the deployment, scaling, and management of containerized applications across a cluster of nodes. Kubernetes provides a consistent and reliable way to manage applications, whether they are running on-premises, in the cloud, or in hybrid environments.**

**Why Kubernetes?**

**Kubernetes exists to manage containerized applications at scale, providing automation, scalability, and resilience. It addresses several challenges that arise when running containers in production:**

1. **Automated Deployment & Scaling**
   * **Kubernetes automates rolling updates and rollbacks.**
   * **It can scale applications up or down based on demand (e.g., CPU/memory usage).**
2. **Load Balancing & Traffic Management**
   * **Kubernetes distributes network traffic to ensure stable application performance.**
3. **Self-Healing & Fault Tolerance**
   * **If a container crashes, Kubernetes automatically restarts it.**
   * **If a node fails, Kubernetes shifts workloads to healthy nodes.**
4. **Resource Optimization**
   * **Kubernetes efficiently schedules workloads to maximize resource utilization.**
   * **It optimizes CPU and memory usage across multiple applications.**
5. **Service Discovery & Networking**
   * **Kubernetes provides built-in DNS-based service discovery.**
   * **Services can communicate with each other using stable network addresses.**
6. **Portability & Flexibility**
   * **Kubernetes is cloud-agnostic and works on AWS, Azure, GCP, on-premises, and hybrid environments.**
   * **It supports multiple container runtimes (Docker, containerd, CRI-O).**
7. **Security & Configuration Management**
   * **Kubernetes manages sensitive data using Secrets.**
   * **It isolates workloads for better security.**

**What is Microservices?**

**Microservices is an architectural pattern where an application is structured as a collection of small, independently deployable services. Each service performs a specific function and communicates with other services over well-defined APIs.**

**Architecture & Components of Kubernetes**

**Master Node Components**

**1. API Server**

**The API server is the front end of the Kubernetes control plane. It exposes the Kubernetes API and handles requests from users and other components.**

**2. Scheduler**

**The scheduler assigns pods to nodes based on resource availability and other factors.**

**3. Controller Manager**

**The controller manager ensures the desired state of resources is maintained. Some important controllers are:**

* **Node Controller: Monitors node health.**
* **Replication Controller: Ensures the desired number of pods are running.**
* **Endpoint Controller: Manages services and their endpoints.**
* **Job Controller: Manages one-off tasks.**
* **DaemonSet Controller: Ensures a pod runs on all or specified nodes.**

**4. etcd Database**

**A consistent and highly-available key-value store used as the backing store for all cluster data.**

**Worker Node Components**

**1. Kubelet**

**An agent running on each node, ensuring containers are running as specified in a Pod.**

**2. Kube-proxy**

**A network proxy that maintains network rules for pods and handles communication between services and pods.**

**Kubernetes Workload Components**

**Pods**

**A Pod is the smallest deployable unit in Kubernetes and can contain one or more containers. Pods are managed by higher-level objects like Deployments and ReplicaSets.**

**Example Pod Manifest:**

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: my-pod**

**spec:**

**containers:**

**- name: my-container**

**image: nginx:latest**

**ports:**

**- containerPort: 8080**

**Deployments**

**A Deployment is an object that manages a set of Replica Pods, allowing for declarative updates and scaling.**

**Example Deployment Manifest:**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: my-deployment**

**spec:**

**replicas: 4**

**selector:**

**matchLabels:**

**app: my-app**

**template:**

**metadata:**

**labels:**

**app: my-app**

**spec:**

**containers:**

**- name: my-container**

**image: nginx:latest**

**ports:**

**- containerPort: 8080**

**ReplicaSets**

**A ReplicaSet ensures that a specified number of replica Pods are running at any given time.**

**Example ReplicaSet Manifest:**

**apiVersion: apps/v1**

**kind: ReplicaSet**

**metadata:**

**name: my-replicaset**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: my-app**

**template:**

**metadata:**

**labels:**

**app: my-app**

**spec:**

**containers:**

**- name: my-container**

**image: nginx:latest**

**ports:**

**- containerPort: 8080**

**Kubernetes Services**

**ClusterIP**

**The default service type that exposes the service on a cluster-internal IP, making it reachable only from within the cluster.**

**Example ClusterIP Service Manifest:**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-app-cip**

**spec:**

**type: ClusterIP**

**ports:**

**- port: 80**

**targetPort: 80**

**protocol: TCP**

**selector:**

**app: my-app-cip**

**NodePort**

**Exposes the service on each node’s IP at a static port. This makes the service accessible from outside the cluster.**

**Example NodePort Service Manifest:**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-app-np**

**spec:**

**type: NodePort**

**ports:**

**- port: 80**

**targetPort: 80**

**protocol: TCP**

**selector:**

**app: my-app-np**

**LoadBalancer**

**Exposes the service externally using a cloud provider’s load balancer. Automatically creates a NodePort and ClusterIP service.**

**Example LoadBalancer Service Manifest:**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-app-lb**

**spec:**

**type: LoadBalancer**

**ports:**

**- port: 80**

**targetPort: 80**

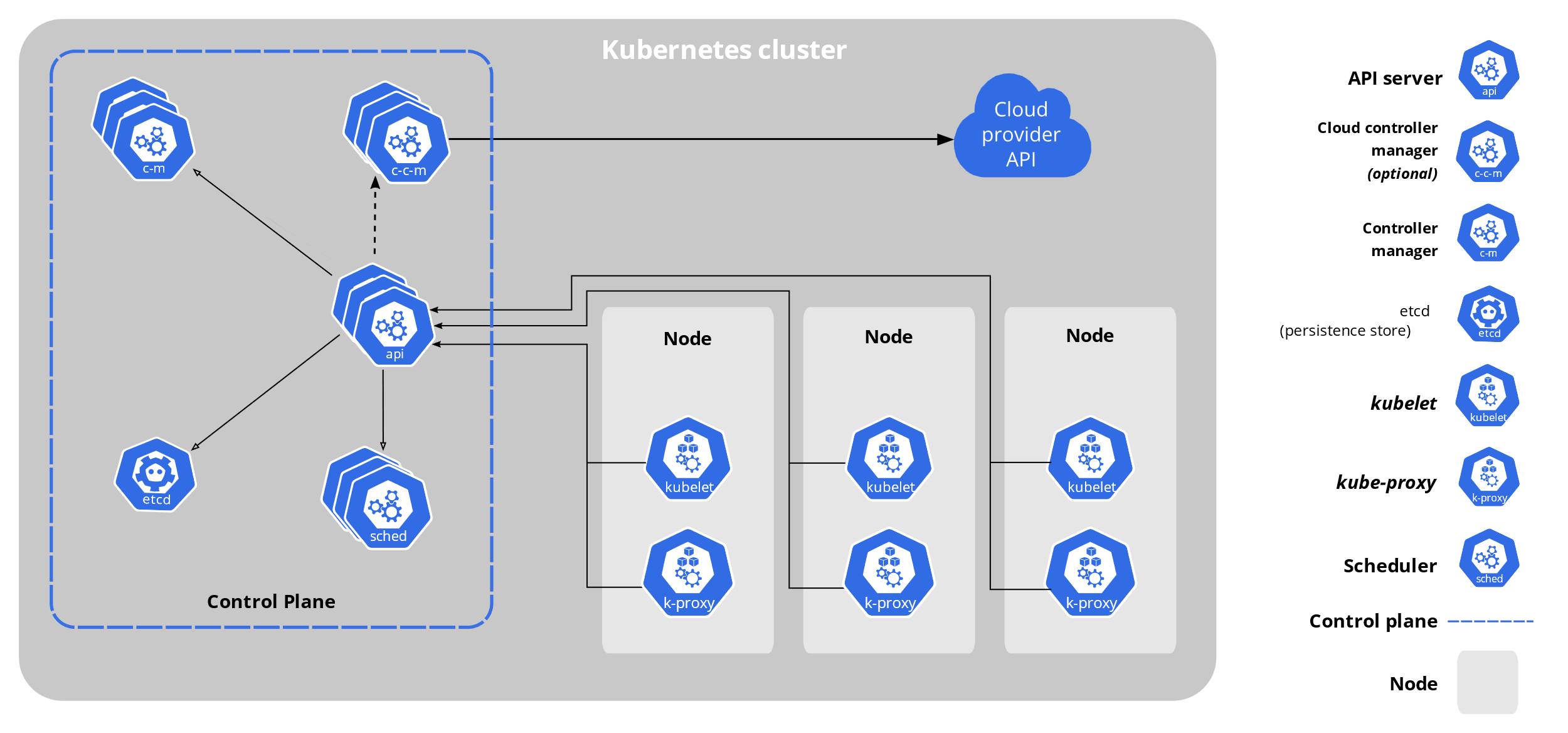
**protocol: TCP**

**selector:**

**app: my-app-lb**

**Architecture & Components of Kubernetes**

**=**

****

**Master Node Components**

**API Server:** The API server serves as the front end of the Kubernetes control plane, exposing the Kubernetes API. It’s the main point of interaction for administrators and users to communicate with the cluster. All requests to the cluster go through the API server, which manages requests, validates them, and provides access to the Kubernetes resources.

**Scheduler:** The scheduler is responsible for assigning pods (units of computation) to nodes. When a new pod is created, it doesn’t automatically have a node to run on. The scheduler monitors for these unscheduled pods, assesses resource availability, and decides which node will be the best fit for each pod. This process is crucial to optimizing resource usage across the cluster.

**Controller Manager**: This component runs various controllers that help keep the cluster’s state in sync. Each controller watches over certain resources or tasks and ensures that the desired state matches the current state. Examples include the Node Controller, which manages node statuses, and the Replication Controller, which ensures the desired number of pod replicas are running at any time.

In Kubernetes, the Controller Manager runs multiple types of controllers, each responsible for monitoring and managing the state of various resources within the cluster. Here are some key controllers managed by the Controller Manager:

Node Controller: Monitors the health and availability of nodes in the cluster. If a node goes down, the Node Controller detects this and initiates actions to prevent pods from being scheduled on that node.

Replication Controller: Ensures that a specified number of pod replicas are running at any given time. If a pod fails or is terminated, the Replication Controller creates new ones to maintain the desired replica count.

Endpoint Controller: Manages Endpoints objects, which connect Services to the appropriate set of Pods. This ensures that clients can access the correct instances of an application or service.

Service Account and Token Controller: Creates default Service Accounts for newly created namespaces and manages API tokens associated with them, allowing applications to authenticate and access the Kubernetes API securely.

Job Controller: Ensures that Job resources complete their specified tasks. It manages the execution of Pods for one-off tasks, ensuring that they finish as expected.

DaemonSet Controller: Ensures that a specific pod runs on all or specific nodes. This is useful for services that need to be available on every node, such as logging or monitoring agents.

ReplicaSet Controller: Similar to the Replication Controller but with more advanced features, it also ensures that a specific number of pod replicas are running. This is commonly used by Deployments for managing stateless applications.

Deployment Controller: Manages Deployment resources, enabling rolling updates, rollbacks, and maintaining desired states for pods in a more controlled manner.

StatefulSet Controller: Manages StatefulSet resources, which are designed for stateful applications that require stable network identities, persistent storage, and ordered deployment or scaling.

Garbage Collector Controller: Cleans up resources that are no longer in use, such as deleted objects or unused replica sets, ensuring efficient resource utilization.

Horizontal Pod Autoscaler (HPA) Controller: Monitors CPU utilization or custom metrics for a set of pods and adjusts the number of replicas automatically to match the desired load.

CronJob Controller: Manages CronJob resources, which are Jobs scheduled to run at specified times, similar to cron tasks in Linux.

Each of these controllers continuously watches the state of the cluster through the API server and takes necessary actions to align the actual state with the desired state defined in Kubernetes manifests.

These master node components work together to manage and maintain the cluster's desired state, allowing Kubernetes to efficiently orchestrate containerized applicationsEtcd Database: Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

**Worker Node Component**

**Kubelet:**

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

**Kube-proxy:**

• kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

**Workload Components**

Pods

Deployments

Services

Ingress

ConfigMaps

Secrets

Namespaces

Persistent Volumes

Persistent Volume Claims

**PODs**

• In Kubernetes, a "pod" is the smallest and most basic unit of deployment.

• It represents a single instance of a running process within the cluster. A pod encapsulates one or more containers, storage resources, network configurations, and other options required to run a specific set of containers together.

• Pods are typically created and managed using higher- level abstractions such as Deployments, ReplicaSets, or StatefulSets, which provide additional features like scaling, rolling updates, and self-healing capabilities.

• Pods provide several benefits, including resource isolation, flexible deployment strategies, easy scaling, and enhanced reliability.

Pod manifest

apiVersion: v1

kind: Pod

metadata:

  name: my-pod

spec:

  containers:

  - name: my-container

    image: nginx: latest

    ports:

    - containerPort: 8080

**Deployments**

* In Kubernetes, a "Deployment" is an object that provides declarative updates and management for a set of replica Pods.
* When you create a Deployment, you specify the desired state by defining the container images, number of replicas, and other configuration parameters.
* Kubernetes then ensures that the actual state matches the desired state.
* If there are any discrepancies, Kubernetes automatically takes action to reconcile the state, creating or deleting Pods as necessary.

**Deployment manifest**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: my-deployment

spec:

  replicas: 4

  selector:

    matchLabels:

      app: my-app

  template:

    metadata:

      labels:

        app: my-app

    spec:

      containers:

      - name: my-container

        image: nginx:latest

        ports:

        - containerPort: 8080

**ReplicaSets**

* In Kubernetes, a "ReplicaSet" is an object that ensures a specified number of replica Pods are running at any given time.
* It is responsible for maintaining the desired replica count and managing the lifecycle of the Pods.
* ReplicaSets are typically used to manage stateless applications where individual instances of the application can be treated as interchangeable.
* They help in achieving high availability and scalability by automatically scaling the number of replicas up or down based on the defined specifications.
* When you create a ReplicaSet, you specify the desired number of replicas and provide a template for creating the Pods.

**ReplicaSet manifest**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: my-replicaset

spec:

  replicas: 3

  selector:

    matchLabels:

      app: my-app

  template:

    metadata:

      labels:

        app: my-app

    spec:

      containers:

      - name: my-container

        image: nginx:latest

        ports:

        - containerPort: 8080

**Key Differences Between ReplicaSets and Deployments**

| **Feature** | **ReplicaSet** | **Deployment** |
| --- | --- | --- |
| **Primary Purpose** | Ensures a specified number of identical Pods are running at all times | Manages ReplicaSets and allows for declarative updates to Pods, enabling rollouts and rollbacks |
| **Updates** | Limited to scaling (increasing/decreasing replicas) | Supports rolling updates, canary deployments, and rollbacks for seamless application updates |
| **Declarative State** | Focused on maintaining a stable count of Pods with no built-in update strategy | Allows for declarative, versioned management of application state, making it easy to update applications with minimal downtime |
| **History Management** | None | Stores previous ReplicaSets for rollback and history tracking |
| **Self-healing** | Yes, ensures Pods are recreated if they fail | Yes, along with sophisticated mechanisms to handle failed updates during rollouts |

**Service components:**

Kubernetes ServiceTypes allow you to specify what kind of Service you want. The default is ClusterIP.

Type values and their behaviors are:

**ClusterIP**: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType. (To talk to other nodes in the cluster)­­

apiVersion: apps/v1

kind: Deployment

metadata:

  name: my-deployment-cip

spec:

  replicas: 3

  selector:

    matchLabels:

      app: my-app-cip

  template:

    metadata:

      labels:

        app: my-app-cip

    spec:

      containers:

      - name: my-container

        image: nginx:latest

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-app-cip

  labels:

    app: my-app-cip

spec:

  type: ClusterIP

  ports:

  - port: 80

    targetPort: 80

    protocol: TCP

  selector:

    app: my-app-cip

NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>. (The entrpoint for node)

apiVersion: apps/v1

kind: Deployment

metadata:

  name: my-deployment-np

spec:

  replicas: 3

  selector:

    matchLabels:

      app: my-app-np

  template:

    metadata:

      labels:

        app: my-app-np

    spec:

      containers:

      - name: my-container

        image: nginx:latest

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-app-np

  labels:

    app: my-app-np

spec:

  type: NodePort

  ports:

  - port: 80

    targetPort: 80

    protocol: TCP

  selector:

    app: my-app-np

LoadBalancer: Exposes the Service externally using cloud provider’s load balancer. NodePort and ClusterIP services, to which the external load balancer routes, are automatically created.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: my-deployment-lb

spec:

  replicas: 3

  selector:

    matchLabels:

      app: my-app-lb

  template:

    metadata:

      labels:

        app: my-app-lb

    spec:

      containers:

      - name: my-container

        image: nginx:latest

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-app-lb

  labels:

    app: my-app-lb

spec:

  type: LoadBalancer

  ports:

  - port: 80

    targetPort: 80

    protocol: TCP

  selector:

    app: my-app-lb

**labels in metadata**

* **Purpose**: Labels in metadata are key-value pairs added to Kubernetes objects like Pods, ReplicaSets, Deployments, etc., to provide identifying metadata.
* **In your YAML**:

yaml

Copy code

labels:

app: guestbook

tier: frontend

Here, the labels app: guestbook and tier: frontend are attached to the ReplicaSet as metadata. These labels serve to categorize the ReplicaSet by specifying it belongs to the "guestbook" app and it’s part of the "frontend" tier.

* **Usage**: Labels can be used for querying and organizing resources across Kubernetes. For instance, you can get all objects with app=guestbook by running kubectl get <resource-type> -l app=guestbook.

**matchLabels in selector**

* **Purpose**: matchLabels is used within the selector section to define criteria for selecting Pods that the ReplicaSet should manage.
* **In your YAML**:

yaml

Copy code

selector:

matchLabels:

tier: frontend

The matchLabels here indicates that the ReplicaSet will target and manage all Pods that have the label tier: frontend. This ensures the ReplicaSet knows which Pods to maintain in the desired state (replicas) specified (3 Pods in this case).

* **Usage**: matchLabels ensures that only Pods with specific labels are managed by the ReplicaSet, linking the ReplicaSet to its managed Pods based on labels.

https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0