1. Kubernetes Basics

What is Kubernetes?

Kubernetes (often abbreviated as K8s, where 8 stands for the eight letters between K and s) is an open-source platform for automating the deployment, scaling, and management of containerized applications. Originally developed by Google, it is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes abstracts the underlying infrastructure, allowing you to manage applications across multiple nodes (servers) in a cluster.

Why Kubernetes?

- **Portability**: Works across on-premises, public clouds (AWS, Azure, GCP), and hybrid environments.
- **Scalability**: Automatically scales applications based on demand.
- **Resilience**: Self-heals by restarting failed containers or redistributing workloads.
- **Declarative Configuration**: Define the desired state of your application, and Kubernetes ensures it.
- **Ecosystem**: Integrates with tools like Helm, Prometheus, and ArgoCD for advanced workflows.

Core Concepts

- **Cluster**: A set of nodes (machines) that run containerized applications. A cluster consists of a control plane (manages the cluster) and worker nodes (run applications).
- **Node**: A single machine (physical or virtual) in the cluster. The control plane nodes manage the cluster, while worker nodes run application workloads.
- **Pod**: The smallest deployable unit in Kubernetes, typically containing one or more containers that share storage, networking, and a specification for how to run.
- **Container**: A lightweight, portable unit that packages an application and its dependencies.
- **Control Plane**: Components like the API server, scheduler, controller manager, and etcd that manage the cluster's state.
- **Kubelet**: An agent running on each node, ensuring containers in pods are running as expected.
- **Kube-Proxy**: Runs on each node to manage network rules for communication between pods and services.

Example

Imagine running a web application. Without Kubernetes, you'd manually deploy containers to servers, configure networking, and handle scaling. With Kubernetes, you define a configuration (e.g., YAML file) specifying how many instances of the web app to run, and Kubernetes handles deployment, scaling, and failover automatically.

---

2. Minikube

What is Minikube?

Minikube is a tool that runs a single-node Kubernetes cluster locally on your machine, ideal for learning, development, and testing. It creates a virtual machine (or container) to simulate a Kubernetes cluster.

Why Use Minikube?

- **Ease of Setup**: No need for a full cloud-based cluster.
- **Local Development**: Test Kubernetes configurations without incurring cloud costs.
- **Learning Tool**: Perfect for understanding Kubernetes concepts hands-on.

How Minikube Works

Minikube sets up a single-node cluster with all Kubernetes components (control plane and worker node) running in a VM or container. It supports drivers like Docker, VirtualBox, or HyperKit to run the cluster.

Setup and Usage

1. **Install Minikube**:

   ```
   # macOS
   brew install minikube
   # Linux
   curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
   sudo install minikube-linux-amd64 /usr/local/bin/minikube
   ```

2. **Start Minikube**:

   ```
   minikube start
   ```

   This creates a local cluster with default settings (e.g., 2 CPUs, 2GB RAM).

3. **Interact with Cluster**: Use `kubectl` to manage the cluster:

   ```
   kubectl get pods
   ```

4. **Access Dashboard**:

   ```
   minikube dashboard
   ```

   Opens a web-based UI for visualizing cluster resources.

Example

To deploy a simple Nginx web server:

```
kubectl create deployment nginx --image=nginx
kubectl expose deployment nginx --type=NodePort --port=80
minikube service nginx
```

This deploys Nginx, exposes it as a service, and opens the URL in your browser.

Advanced Minikube Features

- **Add-ons**: Enable features like Ingress, Metrics Server, or Helm with `minikube addons enable <addon-name>`.
- **Multi-Node Clusters**: Simulate a multi-node cluster with `minikube start --nodes 2`.
- **Resource Configuration**: Customize CPU and memory with `minikube start --cpus 4 --memory 4096`.

---

3. Kubernetes Pods

What is a Pod?

A pod is the smallest deployable unit in Kubernetes, representing one or more containers that share resources like storage, networking, and a lifecycle. Pods are ephemeral, meaning they can be created or destroyed dynamically.

Key Characteristics

- **Shared Context**: Containers in a pod share the same network namespace (localhost communication) and storage volumes.
- **Single Purpose**: Pods typically run a single primary container, but sidecar containers (e.g., for logging) can be included.
- **Ephemeral**: Pods are not meant to be long-lived; they are managed by higher-level controllers like Deployments.

Pod Lifecycle

1. **Pending**: Pod is created but not yet running.
2. **Running**: Containers are running.
3. **Succeeded/Failed**: Pod's containers have terminated (successfully or with errors).
4. **CrashLoopBackOff**: A container fails repeatedly and Kubernetes retries.

Example: Pod Definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Apply with:

```
kubectl apply -f pod.yaml
```

Multi-Container Pods

A pod can include sidecar containers for tasks like logging or monitoring:

```yaml
spec:
  containers:
  - name: main-app
    image: my-app:latest
  - name: log-agent
    image: fluentd:latest
    volumeMounts:
    - name: logs
      mountPath: /logs
  volumes:
  - name: logs
    emptyDir: {}
```

Here, the `log-agent` container collects logs from the `main-app` container.

Advanced Pod Features

- **Init Containers**: Run setup tasks before the main containers start.

  ```yaml
  spec:
    initContainers:
    - name: init-task
      image: busybox
      command: ['sh', '-c', 'echo "Setup complete" > /tmp/setup']
  ```

- **Resource Limits**:

```
spec:
  containers:
  - name: my-container
    image: nginx
    resources:
      limits:
        cpu: "500m"
        memory: "512Mi"
      requests:
        cpu: "200m"
        memory: "256Mi"
```

- **Liveness/Readiness Probes**: Ensure containers are healthy and ready to serve traffic:

```
livenessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```

---

4. Kubernetes Deployments

What is a Deployment?

A Deployment is a Kubernetes resource that manages a set of identical pods, ensuring they are running, scaled, and updated as per the desired state. Deployments are ideal for stateless applications.

Key Features

- **Scaling**: Adjust the number of pod replicas.
- **Rolling Updates**: Update application versions without downtime.
- **Rollback**: Revert to a previous version if an update fails.
- **Self-Healing**: Replace failed pods automatically.

Example: Deployment Definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
```

```yaml
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:1.14
        ports:
        - containerPort: 80
```

Apply with:

```
kubectl apply -f deployment.yaml
```

Scaling a Deployment

```
kubectl scale deployment my-app --replicas=5
```

Rolling Updates

Update the image version:

```yaml
spec:
  template:
    spec:
      containers:
      - name: my-container
        image: nginx:1.15
```

Apply the update:

```
kubectl apply -f deployment.yaml
```

Kubernetes performs a rolling update, replacing old pods with new ones gradually.

Rollback

If the update fails:

```
kubectl rollout undo deployment/my-app
```

Advanced Deployment Strategies

- **Recreate**: Terminate all old pods before starting new ones (causes downtime).

- **Blue-Green**: Deploy a new version alongside the old one, then switch traffic.
- **Canary**: Deploy a new version to a subset of pods and gradually shift traffic.

---

5. Kubernetes Services

What is a Service?

A Service is an abstraction that defines a logical set of pods and a policy to access them. It provides stable networking for pods, which are ephemeral.

Service Types

1. **ClusterIP** (default):
   - Exposes the service within the cluster.
   - Example: Internal API accessible only by other pods.

   ```yaml
   apiVersion: v1
   kind: Service
   metadata:
     name: my-service
   spec:
     selector:
       app: my-app
     ports:
     - protocol: TCP
       port: 80
       targetPort: 8080
     type: ClusterIP
   ```
2. **NodePort**:
   - Exposes the service on a specific port of each node (30000–32767 range).
   - Example: Access a web app via `<node-ip>:<node-port>`.

   ```yaml
   spec:
     type: NodePort
     ports:
     - port: 80
       targetPort: 8080
       nodePort: 30007
   ```
3. **LoadBalancer**:
   - Exposes the service externally via a cloud provider's load balancer.
   - Example: Public-facing web app on AWS ELB.

   ```yaml
   spec:
     type: LoadBalancer
     ports:
   ```

```
    - port: 80
      targetPort: 8080
```

4. **ExternalName**:
    - Maps a Kubernetes service to an external DNS name without creating a proxy.

```
spec:
  type: ExternalName
  externalName: api.example.com
```

Headless Services

For stateful applications (e.g., databases), a headless service (`clusterIP: None`) allows direct access to individual pods:

```
spec:
  clusterIP: None
  selector:
    app: my-db
```

Example

Expose a Deployment as a ClusterIP service:

```
kubectl expose deployment my-app --port=80 --target-port=8080 --type=ClusterIP
```

---

6. Kubernetes Networking

Core Concepts

Kubernetes networking ensures pods can communicate with each other, services, and external systems. Key principles: - **Pod-to-Pod Communication**: Every pod gets a unique IP address, and pods can communicate directly within the cluster. - **Service Networking**: Services provide stable IPs and DNS names for pods. - **External Access**: Ingress or LoadBalancer exposes services to the outside world. - **CNI Plugins**: Kubernetes uses Container Network Interface (CNI) plugins like Calico, Flannel, or Weave for networking.

Networking Model

- **Pod IP**: Each pod gets a cluster-unique IP.
- **Service IP**: A virtual IP for accessing a group of pods.
- **DNS**: Kubernetes runs CoreDNS for internal DNS resolution (e.g., `my-service.default.svc.cluster.local`).
- **Network Policies**: Control traffic between pods using `NetworkPolicy` resources.

Example: Network Policy

Restrict traffic to a pod to only allow specific sources:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-specific
spec:
  podSelector:
    matchLabels:
      app: my-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 80
```

CNI Plugins

- **Calico**: Supports advanced network policies and encryption.
- **Flannel**: Simple overlay network for basic pod communication.
- **Weave**: Provides service discovery and encryption.

Advanced Networking

- **Ingress**: Manages external HTTP/HTTPS traffic with path-based routing.

  ```yaml
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: my-ingress
  spec:
    rules:
    - host: example.com
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: my-service
  ```

```yaml
            port:
              number: 80
```

- **Service Mesh** (e.g., Istio, Linkerd): Adds observability, security, and traffic management.

---

7. ArgoCD Intro

What is ArgoCD?

ArgoCD is a declarative, GitOps-based continuous delivery tool for Kubernetes. It synchronizes Kubernetes resources defined in Git repositories with the cluster's state.

Why ArgoCD?

- **GitOps**: Use Git as the single source of truth for infrastructure and application configuration.
- **Automation**: Automatically syncs changes from Git to the cluster.
- **Auditability**: Track changes via Git history.
- **Multi-Cluster**: Manage multiple Kubernetes clusters.

How ArgoCD Works

1. **Git Repository**: Store Kubernetes manifests (YAML files) or Helm charts.
2. **ArgoCD Controller**: Monitors the Git repo and compares it to the cluster state.
3. **Sync Process**: Applies changes to the cluster to match the Git state.
4. **UI/CLI**: Provides a web interface and CLI for managing applications.

Setup

1. Install ArgoCD:

   ```
   kubectl create namespace argocd
   kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/ma
   ```

2. Access the UI:

   ```
   kubectl port-forward svc/argocd-server -n argocd 8080:443
   ```

3. Log in using the CLI:

   ```
   argocd login localhost:8080
   ```

Example: Deploy an Application

Define an ArgoCD application:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/my-org/my-repo.git
    path: manifests
    targetRevision: HEAD
  destination:
    server: https://kubernetes.default.svc
    namespace: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Apply with:

```
kubectl apply -f application.yaml
```

GitOps with ArgoCD

- **Declarative**: Define the desired state in Git.
- **Pull-Based**: ArgoCD pulls changes from Git, unlike push-based CI/CD.
- **Drift Detection**: ArgoCD detects and corrects deviations in the cluster state.

Advanced ArgoCD Features

- **ApplicationSets**: Manage multiple applications across clusters.
- **RBAC**: Fine-grained access control for teams.
- **Custom Resources**: Extend ArgoCD with plugins for custom workflows.

---

8. GitOps

What is GitOps?

GitOps is a methodology for managing infrastructure and applications using Git as the single source of truth. Changes are made via pull requests, and tools like ArgoCD apply them to the cluster.

Principles

1. **Declarative Configuration**: Infrastructure and apps are defined in Git.
2. **Version Control**: All changes are tracked in Git.
3. **Automated Delivery**: CI/CD tools apply changes automatically.
4. **Observability**: Monitor and correct drift between Git and cluster state.

Benefits

- **Consistency**: Same configuration across environments.
- **Auditability**: Git history tracks who made changes and why.
- **Collaboration**: Use pull requests for team reviews.

Example Workflow

1. Developer updates a Kubernetes manifest in a Git repo.
2. A pull request is created and merged after review.
3. ArgoCD detects the change and applies it to the cluster.

Tools

- **ArgoCD**: Kubernetes-native GitOps tool.
- **Flux**: Another popular GitOps tool.
- **Jenkins X**: Combines CI/CD with GitOps.

---

9. Kubernetes Service Exposure

Methods to Expose Services

1. **ClusterIP**: Internal access within the cluster.
2. **NodePort**: Expose on a node's IP and port.
3. **LoadBalancer**: Use a cloud provider's load balancer.
4. **Ingress**: HTTP/HTTPS routing with advanced features.

Ingress Example

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

Requires an Ingress controller (e.g., NGINX Ingress).

External DNS

Automate DNS record creation with tools like ExternalDNS:

```
metadata:
  annotations:
    external-dns.alpha.kubernetes.io/hostname: app.example.com
```

---

10. Kubernetes App Deployment

Steps to Deploy an Application

1. **Create a Deployment**: Define the app's pods and replicas.
2. **Expose the Deployment**: Create a Service (e.g., ClusterIP or LoadBalancer).
3. **Configure Ingress**: Route external traffic (if needed).
4. **Apply Configurations**: Use ConfigMaps or Secrets for app settings.
5. **Monitor and Scale**: Use metrics and Horizontal Pod Autoscaler (HPA).

Example: Full Deployment

```
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
```

```yaml
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:1.0
        ports:
        - containerPort: 8080
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: db-host
---
# Service
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 8080
  type: ClusterIP
---
# Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
spec:
  rules:
  - host: my-app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app-service
            port:
              number: 80
```

Apply with:

```
kubectl apply -f app.yaml
```

---

11. kubectl Debugging

Key Commands

1. **kubectl describe**:
   - Provides detailed information about a resource, including events.
   - Example:
     ```
     kubectl describe pod my-pod
     ```
     Shows pod status, events, and container details.
2. **kubectl logs**:
   - Retrieves logs from a container in a pod.
   - Example:
     ```
     kubectl logs my-pod -c my-container
     ```
     Use `-f` for streaming logs: `kubectl logs -f my-pod`.
3. **kubectl exec**:
   - Run commands inside a container.
   - Example:
     ```
     kubectl exec -it my-pod -- bash
     ```
4. **kubectl rollout**:
   - Manage deployment updates and rollbacks.
   - Check status:
     `kubectl rollout status deployment/my-app`
   - Rollback:
     `kubectl rollout undo deployment/my-app`

Debugging Workflow

1. Check pod status: `kubectl get pods`.
2. Describe pod for events: `kubectl describe pod my-pod`.
3. View logs: `kubectl logs my-pod`.
4. Inspect container: `kubectl exec -it my-pod -- /bin/sh`.
5. Check cluster resources: `kubectl get all`.

---

12. Kubernetes ConfigMaps

What is a ConfigMap?

A ConfigMap stores configuration data as key-value pairs, decoupling configuration from application code.

Use Cases

- Environment variables.

- Configuration files.
- Command-line arguments.

Example: ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  db-host: mysql-service
  log-level: debug
```

Use in a pod:

```
spec:
  containers:
  - name: my-app
    image: my-app:latest
    env:
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: db-host
```

ConfigMap as a Volume

Mount a ConfigMap as a file:

```
spec:
  containers:
  - name: my-app
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-config
```

---

13. Kubernetes StatefulSet

What is a StatefulSet?

StatefulSet is a Kubernetes controller for managing stateful applications (e.g., databases) that require stable network identities and persistent storage.

Key Features

- **Stable Network Identity**: Each pod has a unique, predictable name (e.g., `mysql-0`, `mysql-1`).
- **Ordered Deployment**: Pods are created and deleted in order.
- **Persistent Storage**: Each pod gets its own PersistentVolumeClaim (PVC).

Example: StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Requires a headless service:

```
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: mysql
spec:
  clusterIP: None
  selector:
    app: mysql
  ports:
  - port: 3306
```

---

14. Helm Charts

What is Helm?

Helm is a package manager for Kubernetes, allowing you to define, install, and manage applications as reusable "charts."

Helm Chart Structure

- **Chart.yaml**: Metadata about the chart.
- **values.yaml**: Default configuration values.
- **templates/**: Kubernetes manifests with templating.
- **charts/**: Dependencies.

Example: Simple Helm Chart

```
helm create my-chart
```

This creates a chart with a Deployment, Service, and Ingress.

Installing a Chart

```
helm install my-release my-chart
```

Customizing Values

Override defaults in `values.yaml`:

```yaml
replicaCount: 3
image:
  repository: nginx
  tag: "1.16"
```

Install with custom values:

```
helm install my-release my-chart -f custom-values.yaml
```

Advanced Helm Features

- **Dependencies**: Include other charts as dependencies.
- **Hooks**: Run scripts at specific lifecycle events (e.g., pre-install).
- **Templating**: Use Go templates for dynamic manifests.

---

15. Kubernetes PersistentVolumeClaims (PVCs)

What is a PVC?

A PersistentVolumeClaim (PVC) is a request for storage by a pod. It abstracts the underlying storage (PersistentVolume, or PV) provided by the cluster.

Key Components

- **PersistentVolume (PV)**: A piece of storage provisioned by an admin or dynamically by a StorageClass.
- **PersistentVolumeClaim (PVC)**: A user's request for storage, bound to a PV.
- **StorageClass**: Defines storage types (e.g., SSD, HDD) and provisioning rules.

Example: PVC

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

Use in a pod:

```yaml
spec:
  containers:
  - name: my-app
    image: my-app:latest
    volumeMounts:
    - name: storage
      mountPath: /data
  volumes:
  - name: storage
```

```
    persistentVolumeClaim:
      claimName: my-pvc
```

Dynamic Provisioning

Use a StorageClass to provision PVs automatically:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

---

16. EKS Solana Deployment

What is EKS?

Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service on AWS. It simplifies cluster management by handling the control plane.

Deploying Solana on EKS

Solana is a high-performance blockchain platform. Deploying it on EKS involves: 1. **EKS Cluster Setup**: `bash eksctl create cluster --name solana-cluster --region us-west-2 --nodegroup-name workers --node-type t3.large --nodes 3` 2. **Solana Helm Chart**: Use a Helm chart for Solana (community or custom). `bash helm repo add solana https://solana.github.io/helm-charts helm install solana solana/solana --set replicas=3` 3. **Storage**: Use PVCs with an EBS-backed StorageClass for validator data. 4. **Networking**: Expose Solana nodes via LoadBalancer or Ingress.

Example: Solana Validator

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: solana-validator
spec:
  serviceName: solana
  replicas: 1
  selector:
    matchLabels:
      app: solana
  template:
    metadata:
      labels:
        app: solana
    spec:
      containers:
      - name: solana
        image: solana:1.10
        args: ["--identity", "/data/identity.json"]
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 100Gi
      storageClassName: gp2
```

---

17. EKS Networking

EKS Networking Model

EKS uses the AWS VPC CNI plugin by default, assigning pod IPs from the VPC subnet.

Key Configurations

- **VPC and Subnets**: Ensure subnets have enough IPs for pods.
- **Security Groups**: Control traffic to nodes and pods.
- **Network Policies**: Use Calico for fine-grained control if needed.

Example: Calico on EKS

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

External Access

Use AWS LoadBalancer or Ingress (e.g., ALB Ingress Controller):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```yaml
  name: my-ingress
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
spec:
  ingressClassName: alb
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

---

18. EKS Monitoring

Tools

1. **Prometheus**: Collects and stores metrics.
2. **Grafana**: Visualizes metrics.
3. **Prometheus Operator**: Simplifies Prometheus deployment.

Prometheus

Prometheus scrapes metrics from Kubernetes components and applications.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  replicas: 2
  serviceMonitorSelector:
    matchLabels:
      app: my-app
```

Grafana

Deploy Grafana to visualize Prometheus metrics:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm install grafana grafana/grafana
```

Prometheus Operator

Automates Prometheus setup:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm install prometheus prometheus-community/kube-prometheus-stack
```

Example: ServiceMonitor

Monitor a service with Prometheus:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: my-app-monitor
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  endpoints:
  - port: web
    path: /metrics
```

---

19. EKS Load Balancer

AWS Load Balancer Controller

Manages AWS Elastic Load Balancers (ALB/NLB) for Kubernetes services.

Setup

1. Install the controller:

   ```
   helm repo add eks https://aws.github.io/eks-charts
   helm install aws-load-balancer-controller eks/aws-load-balancer-controller -n kube-syst
   ```

2. Create a LoadBalancer service:

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: my-service
     annotations:
       service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
   spec:
     selector:
   ```

```
    app: my-app
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
```

---

20. Additional Topics

Horizontal Pod Autoscaler (HPA)

Automatically scales pods based on metrics like CPU or memory.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Kubernetes Secrets

Store sensitive data like passwords or API keys:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  password: cGFzc3dvcmQ= # base64-encoded
```

Use in a pod:

```
spec:
  containers:
```

```
  - name: my-app
    env:
    - name: PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: password
```

Kubernetes RBAC

Role-Based Access Control (RBAC) manages permissions:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
subjects:
- kind: User
  name: user1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

From Beginner to Advanced

Beginner

- Understand core concepts (pods, services, deployments).
- Set up Minikube and deploy simple applications.
- Use `kubectl` for basic operations (`get`, `describe`, `logs`).

Intermediate

- Work with ConfigMaps, Secrets, and StatefulSets.

- Use Helm for managing applications.
- Configure networking (Ingress, Network Policies).

Advanced

- Implement GitOps with ArgoCD.
- Deploy complex apps (e.g., Solana) on EKS.
- Set up monitoring with Prometheus and Grafana.
- Use HPA and RBAC for scalability and security.

Next Steps

- Explore service meshes (Istio, Linkerd).
- Learn advanced storage (CSI drivers).
- Experiment with multi-cluster setups.
- Contribute to CNCF projects or build custom operators.

This guide covers Kubernetes comprehensively, with practical examples and advanced concepts. Let me know if you want to dive deeper into any specific topic!