Must-Know Kubernetes Features for Every DevOps Engineer

1. Autoscaling

- **Question:** What is Kubernetes autoscaling, and how does it help with workload management?
- **Answer:** Kubernetes autoscaling automatically adjusts the number of pods and resources based on real-time application demand, ensuring optimal resource utilization. It includes Horizontal Pod Autoscaler (HPA) for scaling pods, Vertical Pod Autoscaler (VPA) for adjusting pod resources, and Cluster Autoscaler for scaling nodes.

Example: This HPA scales the my-app deployment between 2 & 10 replicas based on CPU usage.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

---

2. Helm Charts

- **Question:** How do Helm charts simplify application deployment in Kubernetes?
- **Answer:** Helm is a package manager for Kubernetes that uses pre-configured templates, known as charts, to deploy applications efficiently. It reduces the need for manually writing multiple YAML files, thereby minimizing errors and saving time.

Helm Chart Structure:

```
• my-helm-chart/
• │── charts/            # Subcharts (optional)
• │── templates/         # Kubernetes manifests (YAML) with templating
• │   │── deployment.yaml
• │   │── service.yaml
• │   │── ingress.yaml
•   │   ├── _helpers.tpl
•   ├── values.yaml         # Default values for the templates
•   ├── Chart.yaml          # Metadata about the chart
•   ├── README.md           # Documentation
•
```

---

3. Network Policies

- **Question:** Why are Kubernetes network policies important for security?
- **Answer:** Network policies define rules for pod communication, restricting unauthorized access between pods or external services. This helps in enhancing security by ensuring only approved communications take place within the cluster.

Example: Only frontend pods can communicate with backend pods on port 8080.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
      ports:
        - protocol: TCP
          port: 8080
```

---

4. Persistent Volumes and Persistent Volume Claims

- **Question:** How does Kubernetes handle persistent storage for stateful applications?
- **Answer:** Persistent Volumes (PVs) provide stable storage resources, while Persistent Volume Claims (PVCs) allow applications to request storage dynamically. This ensures that critical data remains available even if pods restart or are deleted.

Example: Creating a Persistent Volume (PV) and PVC

```yaml
apiVersion: v1
kind: PersistentVolume
```

```yaml
metadata:
  name: my-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
```

PVC:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi  # Requesting 5Gi storage
  storageClassName: manual
```

Using PVC in a Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-pvc
spec:
  containers:
    - name: my-app
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: storage-volume
  volumes:
    - name: storage-volume
      persistentVolumeClaim:
        claimName: my-pvc
```

---

5. Ingress Controllers

- **Question:** What is the role of an Ingress controller in Kubernetes?

- **Answer:** An Ingress controller manages external access to Kubernetes services by directing HTTP and HTTPS traffic based on predefined rules. It simplifies service exposure without requiring separate load balancers for each service

Create a TLS Secret (SSL Certificate):

```
kubectl create secret tls my-tls-secret \ --cert=path/to/tls.crt \ --
key=path/to/tls.key
```

Example: Path-Based Routing:

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

Ingress with TLS Enabled

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-ingress
spec:
  tls:
    - hosts:
        - secure.example.com
      secretName: my-tls-secret
  rules:
```

```yaml
    - host: secure.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: secure-service
                port:
                  number: 443
```

Example: Backend Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

---

6. ConfigMaps and Secrets

- **Question:** How do ConfigMaps and Secrets enhance application configuration management?
- **Answer:** ConfigMaps store non-sensitive configuration data, while Secrets store sensitive data like passwords and API keys. These features separate configuration from application code, improving security and flexibility.

Creating a ConfigMap (YAML):

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_HOST: "mysql-service"
  DATABASE_PORT: "3306"
  LOG_LEVEL: "debug"
```

Using ConfigMap in a Pod

```yaml
kind: Pod
metadata:
  name: pod-using-configmap
spec:
  containers:
    - name: my-app
      image: nginx
      envFrom:
        - configMapRef:
            name: app-config
```

Creating a Secret (YAML):

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DATABASE_PASSWORD: bXlzZWNyZXQ=  # Base64 encoded "mysecret"
  API_KEY: c3VwZXJzZWNyZXQ=        # Base64 encoded "supersecret"
```

Mounting Secret as a Volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-mounting-secret
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: app-secret
  containers:
    - name: my-app
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret
```

---

7. Service Mesh

- **Question:** What benefits does a service mesh provide in microservices architecture?
- **Answer:** A service mesh manages service-to-service communication by handling traffic routing, security, and observability. Tools like Istio or Linkerd enable encrypted communication and intelligent traffic management.

---

8. Role-Based Access Control (RBAC)

- **Question:** How does Kubernetes RBAC improve security?
- **Answer:** RBAC defines granular permissions for users and services, ensuring that only authorized entities can access specific resources. This prevents unauthorized modifications or security breaches.

Creating a Role (Namespace-Specific):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: dev
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
```

RoleBinding assigns a **Role** to a user or service account within a namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: dev
subjects:
  - kind: User
    name: dev-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

---

9. Pod Disruption Budgets (PDBs)

- **Question:** Why are Pod Disruption Budgets essential for maintaining application availability?
- **Answer:** PDBs define the minimum number of pods that must remain available during updates or maintenance. This prevents downtime and ensures continuous service availability.
- For **stateful applications** like databases (MySQL, MongoDB, Redis), use **PDB with StatefulSets** to prevent disruption

Example: PDB for Redis StatefulSet

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: redis-pdb
  namespace: default
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: redis
```

Example: Redis StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:latest
```

---

10. StatefulSets

- **Question:** What is the purpose of StatefulSets in Kubernetes?
- **Answer:** StatefulSets manage stateful applications by ensuring stable network identities, persistent storage, and ordered deployments. They are ideal for applications like databases where pod identity and data persistence are crucial.

Example: Deploying a MySQL Database with StatefulSet:

- **Replicas: 3** → Creates 3 pods (mysql-0, mysql-1, mysql-2).
- **Pod DNS Names** → Each pod gets a hostname like mysql-0.mysql.default.svc.cluster.local.
- **Persistent Storage** → Each pod gets its own volume (mysql-0, mysql-1 won't share storage).

```yaml
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: "mypassword"
          ports:
            - containerPort: 3306
          volumeMounts:
            - name: mysql-storage
              mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: mysql-storage
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

Creating a Headless Service for StatefulSet:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  clusterIP: None
  selector:
    app: mysql
  ports:
    - port: 3306
```

11. DaemonSets

- **Question:** How do DaemonSets differ from Deployments in Kubernetes?
- **Answer:** DaemonSets ensure that a specific pod runs on every node in a cluster, typically for system-level tasks like log collection and monitoring. Unlike Deployments, they do not scale up or down based on demand.

---

12. Jobs and CronJobs

- **Question:** When should you use Jobs and CronJobs in Kubernetes?
- **Answer:** Jobs are used for running tasks that execute once and then terminate, such as data processing or batch jobs. CronJobs automate recurring tasks, such as database backups, by scheduling Jobs at specified intervals.

Example: Running a One-Time Job

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: backup-job
spec:
  template:
    spec:
      containers:
        - name: backup
          image: alpine
          command: ["sh", "-c", "echo 'Backing up database'"]
      restartPolicy: Never
```

Example: Running a CronJob Every 5 Minutes

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: scheduled-job
spec:
  schedule: "*/5 * * * *"  # Every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: cron-job
              image: busybox
              command: ["sh", "-c", "echo 'Running scheduled task'"]
          restartPolicy: OnFailure
```

13. Kubernetes Namespaces

- **Question:** What is the significance of Namespaces in Kubernetes?
- **Answer:** Namespaces provide logical isolation within a cluster, allowing teams to organize resources efficiently and manage multi-tenant environments without conflicts.

Creating a Namespace

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev

```

14. Kubernetes Init Containers

- **Question:** How do Init Containers differ from regular application containers?
- **Answer:** Init Containers run before application containers to perform setup tasks, such as downloading dependencies or configuring the environment, ensuring that the main application starts correctly.

Example: Init Container Waiting for a Database

```yaml

apiVersion: v1
kind: Pod
metadata:
  name: init-container-example
spec:
  initContainers:
    - name: init-db
      image: busybox
      command: ["sh", "-c", "until nc -z db-service 3306; do echo waiting for database; sleep 2; done"]
  containers:
    - name: my-app
      image: nginx
```

15. Kubernetes Probes (Liveness, Readiness, and Startup Probes)

- **Question:** What is the purpose of Kubernetes probes, and how do they improve application stability?

- **Answer:** Kubernetes uses probes to check the health of pods. Liveness probes restart failing containers, Readiness probes control traffic routing to only healthy pods, and Startup probes ensure proper initialization before allowing traffic.

Example:
LivenessProbe: Restarts the pod if it becomes unresponsive.
ReadinessProbe: Ensures the pod is ready before sending traffic.
StartupProbe: Ensures full initialization before marking it healthy.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: probe-demo
spec:
  containers:
    - name: my-app
      image: my-app:latest
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
      startupProbe:
        httpGet:
          path: /startup
          port: 8080
        failureThreshold: 30
        periodSeconds: 5
```

---

16. Kubernetes Resource Quotas and Limits

- **Question:** How do Resource Quotas and Limits help manage cluster resources?
- **Answer:** Resource Quotas restrict overall resource consumption per namespace, while Limits define the maximum CPU and memory that a pod or container can use, ensuring fair resource distribution.

Example: Setting Resource Quotas for a Namespace

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
  namespace: dev
spec:
  hard:
    pods: "10"               # Max 10 pods in the namespace
    requests.cpu: "4"        # Max 4 CPU requests
    requests.memory: 8Gi     # Max 8Gi memory requests
    limits.cpu: "8"          # Max 8 CPU usage
    limits.memory: 16Gi      # Max 16Gi memory usage
```

Example: Assigning CPU/Memory Requests and Limits to a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limited-pod
  namespace: dev
spec:
  containers:
    - name: my-app
      image: nginx
      resources:
        requests:
          cpu: "250m"     # Guarantees 0.25 CPU core
          memory: "256Mi"   # Guarantees 256Mi memory
        limits:
          cpu: "500m"     # Maximum 0.5 CPU core
          memory: "512Mi"   # Maximum 512Mi memory
```

---

17. Kubernetes Service Types

- **Question:** What are the different types of Kubernetes services, and when should they be used?
- **Answer:** Kubernetes offers four service types:
    - **ClusterIP** (default) for internal communication.
    - **NodePort** for exposing services on a static port of each node.
    - **LoadBalancer** for exposing services externally via cloud provider load balancers.
    - **ExternalName** for aliasing services outside the cluster.

18.Kubernetes Pod Lifecycle

- **Question:** What are the different phases of a Kubernetes pod lifecycle?
- **Answer:** A pod goes through various phases:
    - **Pending** (waiting for resources).
    - **Running** (active execution).
    - **Succeeded** (completed successfully).

  - **Failed** (terminated due to failure).
  - **Unknown** (status retrieval failure)

19.Kubernetes Affinity and Anti-Affinity Rules

- **Question:** How do affinity and anti-affinity rules improve pod scheduling in Kubernetes?
- **Answer:** Affinity rules allow you to specify which nodes a pod should run on based on labels, while anti-affinity rules prevent pods from running on the same node. This helps in optimizing resource utilization and improving fault tolerance. For example, you can ensure that pods of the same application are spread across different nodes to avoid a single point of failure.

Example: Running Microservices Together:

Ensures web-app **runs on the same node** as a backend pod.

Uses topologyKey: "kubernetes.io/hostname" → Ensures co-location **on the same node**.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchLabels:
                  app: backend
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: web-app
          image: nginx
```

20. Kubernetes Blue-Green and Canary Deployments

- **Question:** How do Blue-Green and Canary deployments improve application rollouts?
- **Answer:**

    - **Blue-Green Deployment:** Runs two identical environments, allowing seamless traffic switching during updates.
    - **Canary Deployment:** Gradually introduces new versions to a small user group, minimizing the risk of failures.

---

21. Kubernetes Horizontal Scaling vs. Vertical Scaling

- **Question:** What is the difference between Horizontal and Vertical Scaling in Kubernetes?
- **Answer:**
    - **Horizontal Scaling (HPA):** Increases or decreases the number of pods based on demand.
    - **Vertical Scaling (VPA):** Adjusts pod CPU/memory resources dynamically.

22. Kubernetes Security Contexts

- **Question:** How do Security Contexts enhance Kubernetes workload security?
- **Answer:** Security Contexts define privilege levels for pods and containers, enforcing restrictions like running as a non-root user or setting read-only filesystems.

---

23. Kubernetes Sidecar Containers

- **Question:** What are Sidecar Containers, and how are they used in Kubernetes?
- **Answer:** Sidecar Containers run alongside the main application container in a pod, performing auxiliary functions like logging, monitoring, or security.

---

24. Kubernetes Cost Optimization Strategies

- **Question:** What are effective strategies to optimize Kubernetes cost in cloud environments?
- **Answer:**
    - Use **Spot Instances** for non-critical workloads.
    - Enable **Cluster Autoscaler** to scale down unused nodes.
    - Implement **Resource Requests & Limits** to prevent over-provisioning.
    - Use **Vertical Pod Autoscaler (VPA)** to optimize pod resource allocation dynamically.

---

25. Kubernetes Zero-Trust Security Model

- **Question:** How can Kubernetes implement a Zero-Trust security model?
- **Answer:**
    - Enforce **RBAC with least privilege** policies.
    - Implement **Network Policies** to limit internal pod communications.
    - Use **mTLS (mutual TLS)** via a service mesh like Istio.
    - Apply **OPA/Gatekeeper** to enforce policy-as-code.

26. Kubernetes Image Scanning with Trivy or Clair

- **Question:** Why is container image scanning important in Kubernetes security?
- **Answer:**
  - Scans container images for vulnerabilities before deployment.
  - **Trivy and Clair** integrate with CI/CD pipelines for automated security checks.
  - Prevents running containers with known CVEs (Common Vulnerabilities and Exposures).

---

27. Kubernetes Immutable Deployments with GitOps

- **Question:** How does GitOps improve Kubernetes deployment security?
- **Answer:**
  - Uses **declarative configurations stored in Git**.
  - Ensures only authorized changes are applied.
  - Tools like **ArgoCD and Flux** continuously sync Kubernetes with Git state.

---

28. Monitoring a Sample Application

- **Question:** How can you set up and integrate Prometheus with Grafana for monitoring Kubernetes applications?
- **Answer:**
  Prometheus collects and stores Kubernetes metrics by scraping `/metrics` endpoints from applications, nodes, and services. Grafana visualizes this data through customizable dashboards, enabling real-time monitoring of CPU, memory, pods, and network usage. Together, they provide a powerful observability stack, with Prometheus handling data collection and alerting, while Grafana offers rich visualization and analysis.

install Prometheus and Grafana using Helm:

Step 1: Add the Prometheus Helm Repository

helm repo update

Step 2: Install Prometheus

helm install prometheus prometheus-community/kube-prometheus-stack

Example: **Monitoring a Sample Application**

Let's assume you have a simple web application that exposes metrics on port 8000. Here's how you can set up monitoring for it:

Step 1: Deploy the Application

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
```

```yaml
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:latest
        ports:
        - containerPort: 8000
```

Step 3: Create a ServiceMonitor

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    app: my-app
  ports:
  - name: web
    port: 8000
    targetPort: 8000
```

Step 3: Create a ServiceMonitor

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: my-app-monitor
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: my-app
  endpoints:
  - port: web
    interval: 30s
```