Kubernetes for Beginners (Step by Step)

Introduction to Kubernetes

What is Kubernetes?

In Simple Terms: Kubernetes (often shortened to K8s) is a powerful system designed to manage containerized applications across many computers. Think of it as an orchestra conductor for your software, making sure everything runs in harmony.

Technical Definition: Kubernetes is an open-source container orchestration platform. It automates the deployment, scaling, and management of applications packaged within containers.

Real-World Example: Imagine your website is made up of several smaller parts (like a shopping cart, product display, and search bar). Kubernetes acts like the conductor, keeping all these parts running

smoothly, scaling up when traffic increases, and restarting them if something goes wrong - all without manual intervention.

History and Evolution of Kubernetes

Google Roots: Kubernetes has its origins in Google's internal system called Borg. Borg was responsible for managing Google's massive data centers and workloads like Gmail and Google Search.

Open-Source Birth: In 2014, Google decided to release Kubernetes as an open-source project. This meant anyone could use and contribute to its development.

Community Power: Since then, Kubernetes has exploded in popularity. A huge community of developers and companies now support the project, making it more powerful and versatile with each release.

Benefits of Kubernetes

Efficient Resource Use: Kubernetes packs your applications tightly onto servers, making the best use of your hardware investment. This can save you money!

Scalability: Need to handle a sudden surge in website visitors? Kubernetes can automatically add more copies of your application with just a few commands.

Resiliency: If a server fails, or your application crashes, Kubernetes

steps in to restart things, minimizing downtime for your users.

Developer Friendliness: Kubernetes makes it easier to deploy new versions of your application without disrupting your users by using concepts like rolling updates.

Cloud Flexibility: Kubernetes runs seamlessly on your own servers, in the cloud (like AWS, Azure, Google Cloud), or a combination of both!

Kubernetes Architecture

Understanding how Kubernetes works its magic means understanding its two main parts:

Master Node: The control center of your Kubernetes cluster. It's like the brain that makes all the important decisions.

Worker Nodes: The workhorses that actually run your containerized applications. Imagine them as the muscles that do the heavy lifting.

Master Node

Let's break down the critical components within the Master Node:

API Server: The heart of everything! It's the doorway for communicating with Kubernetes, whether you're a user typing in commands or other system components.

Syntax: kubectl get pods (A command to list running pods)

Controller Manager: A collection of "housekeepers" responsible for

maintaining the desired state of your cluster. They watch for things like failed pods and take corrective action.

Scheduler: The smart matchmaker that decides where to place new pods based on available resources and requirements. It ensures your applications get the resources they need.

etcd: The super-reliable data store where Kubernetes keeps all its important information about the cluster's state. It's like the cluster's memory.

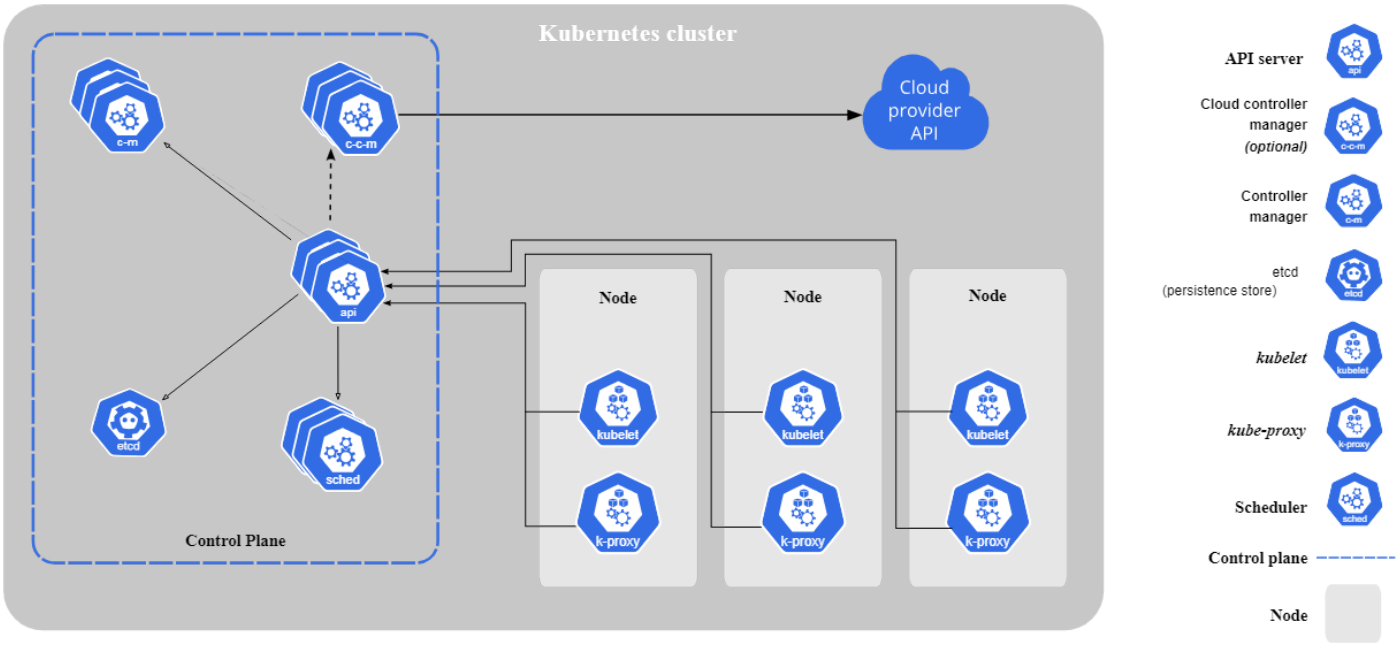
Worker Nodes

Now let's switch to the Worker Nodes:

Kubelet: The main Kubernetes agent running on each node. It talks to the master's API server, ensuring that the containers described in your instructions are up and running.

Kube-proxy: This component handles networking. Think of it as a traffic manager within the cluster, making sure your applications can communicate with each other and the outside world.

Containers: These are the heart of it all! Your applications run packaged neatly inside containers, isolated and ready to do their work.



Simplified Diagram

Real-World Example

Imagine a bustling factory:

The Master Node: This is the management office, keeping plans, tracking inventory, and assigning tasks.

The Worker Nodes: These are the factory floor with machines and conveyor belts running the production.

Containers: These are individually packaged items moving through the factory, ready to be assembled into a final product.

Kubernetes Concepts

Pods

The Atomic Unit: A Pod is the smallest deployable unit in Kubernetes. Think of it as a cozy house for one or more closely related containers.

Why Pods? Containers within a Pod share the same network space, storage, and can communicate directly with each other as if they were on the same machine.

Real-World Example: Your website application might need a main container running your web server and another container to handle image processing. These would be packaged together as a single Pod.

Pod Lifecycle: Pods go through a lifecycle:

- **Pending:** Pod is created but not yet scheduled.
- **Running:** Pod scheduled to a node and containers are running.
- **Succeeded:** All containers in the Pod completed their tasks.
- **Failed:** One or more containers crashed.

Multi-Container Pods

Tightly-Coupled Teamwork: Sometimes, applications need helper containers in addition to the main one. For instance, a logging container to collect logs from your main application. This is where multi-container pods shine!

Services

Networking Gateways: Pods can come and go, making them difficult targets for direct communication. Services provide a stable address (IP and port) that routes traffic to the correct Pods, even as they change.

Service Types

- **ClusterIP (default):** Exposes the service internally within the cluster.
- **NodePort:** Exposes the service on each node's IP at a static port, making it accessible from the outside.
- **LoadBalancer:** On cloud providers, this provisions an external load balancer to direct traffic.
- Service Discovery Kubernetes automatically provides environment variables with service details to Pods, so they can easily discover each other.
  ExternalName: Maps a service to a name external to the cluster (useful for referencing external databases).

Deployments

Desired State Management: Deployments are like blueprints - you tell Kubernetes what your application should look like (e.g., I want 3 replicas of this web server), and it works relentlessly to maintain that state.

Updating Deployments: Easy rollouts! Modify your Deployment configuration (like the image tag for your containers), and Kubernetes performs a rolling update, gradually replacing old Pod versions with new ones.

Rollbacks: Did an update go wrong? Rollback with a single command to revert to a previous working version.

Volumes

Data Persistence: Containers are ephemeral; their storage disappears when they stop. Volumes provide storage that outlives individual Pods.

Persistent Volumes (PV): Storage provisioned by an admin, could be cloud storage (AWS EBS, Azure Disk) or network storage (NFS).

Persistent Volume Claims (PVC): Requests by Pods to use storage. Kubernetes matches the request with a suitable PV.

ConfigMaps and Secrets

Configuring Applications: ConfigMaps store non-sensitive configuration data like URLs, database connection strings, or feature flags. You mount these into Pods as environment variables or files.

Securing Sensitive Data: Secrets, similar to ConfigMaps, are designed specifically to store sensitive information like passwords, API keys, or tokens. They get extra security measures.

Kubernetes Installation and Setup

Local Development Environments

These are great for learning, testing, and development before deploying your applications to production.

Minikube

The Solo Act: Minikube spins up a single-node Kubernetes cluster inside a virtual machine on your computer.

Perfect for: Getting your feet wet with Kubernetes and experimenting locally.

How to Install:

> You'll need a virtualization tool like VirtualBox or Hyper-V. Detailed instructions are on the Minikube website: https://minikube.sigs.k8s.io/docs/start/

Docker Desktop

Integrated Powerhouse: If you already use Docker Desktop for container development, good news! It comes with a built-in Kubernetes cluster.

Best for: Developers comfortable with Docker, who want an all-in-one solution.

How to Enable: Go to Docker Desktop's settings and check the "Enable Kubernetes" box.

Cloud-Based Kubernetes

These are managed offerings from major cloud providers, taking care of the complex cluster setup for you.

Amazon Elastic Kubernetes Service (EKS)

- **AWS Integration:** EKS tightly integrates with other AWS services like load balancers and storage.
- Good for: Teams already invested in the AWS ecosystem.

Google Kubernetes Engine (GKE)

- **Google's Expertise:** GKE leverages Google's experience from inventing Kubernetes. It boasts smooth setup and robust features.
- Good for: Anyone using Google Cloud Platform, or those who want a well-established, managed Kubernetes solution.

Microsoft Azure Kubernetes Service (AKS)

- **Azure Fit:** AKS seamlessly connects with Azure's cloud ecosystem.
- Good for: Teams heavily reliant on Microsoft Azure services.

Important Considerations:

Cost: Cloud-based Kubernetes will incur charges from your cloud provider. Local options are usually free, but may have limited resources.

Control: Cloud providers simplify cluster management, but you have less direct control compared to setting up your own cluster from scratch.

Integration: Think about how well the Kubernetes service needs to work with other tools and services you are already using.

Getting Hands-On

Each cloud provider offers step-by-step guides for setting up their Kubernetes service:

1. **Amazon EKS:** https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html

2. **Google GKE:** https://cloud.google.com/kubernetes-engine/docs/quickstart
3. **Microsoft AKS:** https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough

Kubernetes Networking

Networking in Kubernetes is designed to be simple for developers yet flexible enough to handle complex scenarios. Here's a breakdown of core concepts:

Pod Networking

Pod Networking Fundamentals: Every Pod in your cluster gets its own unique IP address. This lets Pods within the cluster communicate with each other directly, like a miniature private network.

- **Pods have their own IP:** Each Pod in your Kubernetes cluster receives a unique IP address. This means Pods can communicate directly with each other using simple networking, regardless of which node they reside on.

- **Pods on the same node:** Communication between Pods on the same node is exceptionally efficient. They talk directly, without extra routing hops.

Kubernetes' Secret Sauce: Kubernetes uses something called the Container Network Interface (CNI) to provide this networking magic. Popular CNI plugins include Calico, Flannel, and Weave Net.

Real-World Example: Think of Pods like apartments in a building. Each apartment (Pod) has its own unique address, allowing residents (containers) to easily communicate and share resources within the apartment, or with other apartments in the building.

Service Networking

Connecting the Dots: Pods come and go, but Services provide fixed virtual IP addresses and a DNS name. Clients (other Pods or external users) connect to the stable Service, and Kubernetes routes traffic to the correct, healthy Pods behind it.

How it Works: A Service acts like a load balancer. When a client tries to connect to the Service, Kubernetes intelligently routes it to a healthy Pod associated with that Service.

Real-World Example: Imagine a Service like a hotel reception desk. It doesn't matter which specific room (Pod) you eventually get, the reception desk provides a consistent way to find a room (or a working Pod).

Ingress

Exposing Services to the Outside: Services are great for internal communication, but what about allowing traffic from outside the cluster to reach your applications? That's where Ingress comes in.

The Gateway: An Ingress defines rules to route external traffic to your Services. This usually involves an Ingress controller, which is a specific component that implements these rules.

Real-World Example: Think of an Ingress like an airport customs checkpoint. It controls and directs traffic flowing into your cluster (which is like a country in this analogy).

Advanced Ingress Features

Beyond Basic Routing: Ingress controllers often bring a treasure trove of features:

- **Path-based Routing:** Send traffic to different Services based on the URL path (e.g., /shop goes to the shopping cart service, /blog to the blog service).
- **TLS Termination:** Decrypt HTTPS traffic at the Ingress level for easier certificate management and improved security.
- **Rate Limiting:** Protect your Services from being overwhelmed with requests.
- **Authentication:** Add an authentication layer in front of your applications.

Popular Ingress Controllers:

- **Nginx Ingress Controller:** A popular and versatile choice.
- **Traefik:** Known for ease of use and integration with Let's Encrypt (for free TLS certificates).

- **HAProxy:** Provides high-performance and advanced load balancing features.

Network Policies

Firewall Rules Inside Your Cluster: Network Policies let you define fine-grained rules about which Pods can communicate with each other. This is crucial for security!

Syntax Example (Basic):

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-pods-only
spec:
  podSelector:
    matchLabels:
      role: web-server
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: db
```

Real-World Example: You might create a policy that only allows Pods with the label "database" to receive incoming traffic from Pods with the

label "web-frontend". This adds a layer of security within your Kubernetes environment.

Network Security Deep Dive

Network Policies in Action: Let's get more specific with an example. Imagine you have a three-tier application:

- **Web Frontend Pods:** (labeled role: frontend)
- **Application Backend Pods:** (labeled role: backend)
- **Database Pods:** (labeled role: database)

A good starting point might be a Network Policy like this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: secure-myapp
spec:
  podSelector:
    matchLabels:
      role: frontend # Selects only our frontend Pods
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: backend # Allows traffic ONLY from backend Pods
```

Key Points to Understand

- **Flat Network:** Kubernetes uses a flat network model. There's no need to deal with complex subnets or port mapping between nodes,

simplifying setup and maintenance.

- **CNI (Container Network Interface):** Kubernetes uses CNI plugins to provide networking for Pods. Popular plugins include Calico, Flannel, and Weave Net.

Kubernetes Workload Management

Scaling Applications

Manual Scaling: The most basic way to scale is to change the number of replicas in your Deployment:

Syntax: kubectl scale deployment my-web-app --replicas=5

Real-World Example: Your website gets mentioned on a popular news site and traffic surges. You quickly increase the replicas of your web-server pods to handle the load.

Autoscaling

Automation FTW: The Horizontal Pod Autoscaler (HPA) automatically scales your Pods based on metrics like CPU utilization or memory usage.

Syntax Example:

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: my-web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-web-app
  minReplicas: 2
  maxReplicas: 8
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Real-World Example: Think of autoscaling like a thermostat for your app. It adjusts the number of replicas up or down to maintain a healthy level of resource usage.

Job and CronJob

Tasks vs. Ongoing Work:

- **Jobs:** Run a task to completion (like a big data processing batch or a database backup script). If the Pod fails, Kubernetes will try again until it succeeds.
- **CronJobs:** Run tasks on a schedule (like a daily report generation or a weekly cleanup job).

Real-World Example: You might have a Job to generate financial reports at the end of the month, or a CronJob to send out a daily newsletter to subscribers.

Kubernetes Security

It's Not Just About Technology: Security involves people, processes, and yes, Kubernetes features too!

Role-Based Access Control (RBAC)

Who Can Do What: RBAC lets you fine-tune permissions within your cluster. You define:

Roles: Describe what actions are allowed (e.g., "Can view pods," "Can create deployments").

RoleBindings: Attach Roles to users or groups.

Example Syntax (Simple):

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-access
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Pod Security Policies (PSPs)

Locking Down Containers: PSPs define what a Pod is and isn't allowed to do.

Example controls:

- Prevent running privileged containers
- Restrict the volumes a Pod can use
- Enforce specific Linux security settings

Image Security

Know Your Containers: It's crucial to:

- **Start from Trusted Base Images:** Use official or well-vetted images.
- **Scan for Vulnerabilities:** Use tools to identify known vulnerabilities (CVEs) in your images.

- **Continuously Update:** Security patches for the OS and libraries within your container images are constantly being released.

Kubernetes Monitoring and Logging

Why It Matters: Imagine driving a car without a dashboard. You wouldn't know how fast you're going, if you're running low on fuel, or if there's an engine warning light flashing – that's what running a Kubernetes cluster without monitoring would be like.

Metrics Server

The Metrics Server is a core Kubernetes component that gathers resource usage stats (think CPU and memory) for your nodes and pods. It provides a starting point for monitoring.

Prometheus

The Metrics Powerhouse: Prometheus is a popular open-source monitoring system that excels at:

- **Collecting Metrics:** It pulls (scrapes) metrics from various targets, including the Kubernetes Metrics Server.
- **Storing Metrics:** Prometheus uses a time-series database, making it perfect for tracking trends.
- **Powerful Queries:** A specialized query language (PromQL) lets you slice and dice your metrics data.

Real-World Example: Imagine tracking the average CPU usage of all

web-server pods in your cluster over the last hour. Prometheus makes that query possible.

Grafana

Visualizing the Data: Grafana is a dashboarding tool that connects to Prometheus (and other data sources) to create beautiful and informative visualizations.

Dashboards: Design custom dashboards showing:

- Line graphs for resource usage over time
- Gauges for current values
- Alerting when thresholds are exceeded

Real-World Example: A dashboard showing the top 5 pods by memory usage, the overall health of your cluster, and graphs tracking requests per second to your web servers.

ELK Stack

Focus on Logs: While Prometheus excels at numerical metrics, the ELK stack is all about logs:

- **Elasticsearch:** A powerful search and analytics engine for storing logs.
- **Logstash:** Collects and processes logs from various sources, including Kubernetes pods.
- **Kibana:** Provides a visual interface to explore and analyze logs in Elasticsearch.

Real-World Example: Use the ELK stack to search for specific error messages across all your application logs, or to visualize patterns of login attempts (helpful for security).

Kubernetes Ecosystem and Tools

Helm

- **App Store for Kubernetes:** Helm simplifies the process of installing, managing, and upgrading applications on your cluster. Think of it as the apt-get or yum for Kubernetes.
- **Charts:** Helm uses pre-packaged bundles called "charts" that contain all the necessary Kubernetes object definitions (Deployments, Services, etc.) to deploy an application.

Real-World Example: Instead of manually creating Kubernetes objects to deploy WordPress, you could use a Helm chart and have it all set up with a single command like helm install stable/wordpress

Syntax Example:
> helm search repo wordpress # Search for WordPress charts

> helm install my-wordpress stable/wordpress # Install a chart

> helm upgrade my-wordpress stable/wordpress # Upgrade a release

Istio

Traffic Master: Istio is a service mesh. It adds a layer on top of your existing Pods, providing:

- Intelligent Traffic Routing: Fine-grained control, split traffic for canary releases or A/B testing.
- Resiliency: Circuit breakers, retries, and timeouts.
- Observability: Metrics and tracing of requests within your cluster.
- Security: Built-in encryption and authentication between services.

Real-World Example: You want to release version 2 of your app, but only route 10% of the traffic to it initially. Istio makes this easy!

Knative

Simplifying Serverless: Knative builds upon Kubernetes to enable serverless workloads. Key ideas:

- Scale to Zero: Knative can scale your applications down to zero Pods when no requests are incoming, saving you resources.
- Traffic Management: Split traffic between different revisions of your app.

Real-World Example: You have an image processing function that only runs when users upload new images. Knative would be a great fit, spinning up your function when needed and scaling down when idle.

Linkerd

Alternative Service Mesh: Linkerd offers similar features to Istio,

focusing on simplicity and lightweight performance.

Real-World Use Case: When you need advanced traffic management and observability, but your primary concern is minimal overhead on your cluster.

13 most commonly used command for Kubernetes 🎯

Here are 13 of the most commonly used kubectl commands for managing a real production Kubernetes environment, along with explanations and common use cases:

✅Core Management

#1 kubectl get

Lists Kubernetes resources.

Include:

→ kubectl get pods (list pods)

→ kubectl get deployments (list deployments)

→ kubectl get services (list services)

→ kubectl get all (list most resources in a namespace)

#2 kubectl describe

Provides detailed information about a resource.

Include:

→ kubectl describe pod my-pod

→ kubectl describe node my-node

#3 kubectl create

Creates resources from files or standard input. Often used with YAML manifest files:

→ kubectl create -f my-deployment.yaml

#4 kubectl apply

Creates or updates resources based on a configuration file. The safer way to manage changes in production as it keeps track of applied changes.

→ kubectl apply -f my-deployment.yaml (apply a deployment definition)

#5 kubectl delete

Deletes resources. Use with caution! Examples:

→ kubectl delete pod my-pod

→ kubectl delete service my-service

✅Debugging and Troubleshooting

#6 kubectl logs

Gets logs from a container within a pod:

→ kubectl logs my-pod

→ kubectl logs my-pod -c my-container (specify a container)

#7 kubectl exec

Executes a command inside a container. Powerful for debugging:

→ kubectl exec -it my-pod -- bash (interactive shell)

#8 kubectl port-forward

Forwards a local port to a port on a pod. Useful for accessing services not directly exposed:

→ kubectl port-forward my-pod 8080:80

✅Monitoring and Analysis

#9 kubectl top

Displays resource usage (CPU/memory) for pods and nodes: → kubectl top pod (pod resource usage) → kubectl top node (node resource usage)

#10 kubectl explain

Describes the fields of a Kubernetes resource:

→ kubectl explain pod → kubectl explain pod.spec (more specific)

✅Managing Workloads

#11 kubectl rollout

Management for deployments, including updates and rollbacks:

→ kubectl rollout status deployment/my-deployment → kubectl rollout undo deployment/my-deployment

#12 kubectl scale

Scales the number of replicas in a deployment or replica set:

→ kubectl scale deployment/my-deployment --replicas=5

#13 kubectl edit

Edits a resource's configuration directly on the cluster. Use with caution in production.

→ kubectl edit deployment my-deployment

Important Notes:

→ Namespaces: Most commands can be used with -n <namespace> to target specific namespaces

→ Output Formats: Use -o wide, -o yaml, or -o json with commands like get and describe to control output formatting.

Kubernetes

60 Most Important Question

Kubernetes Interview Questions for DevOps Profile

✅ For Absolute Beginners (20 Questions)

1. What is Kubernetes, and why is it important for container orchestration?
2. Explain the concept of a Kubernetes pod.
3. How does Kubernetes handle container scaling?
4. What is a Kubernetes service, and why is it useful?
5. Describe the role of a Kubernetes controller.
6. What are labels and selectors in Kubernetes?
7. How do you create a deployment in Kubernetes?
8. What is a Kubernetes namespace, and why would you use it?
9. How does Kubernetes manage secrets and configuration data?
10. What is the difference between a StatefulSet and a Deployment in Kubernetes?
11. What is the difference between a Kubernetes deployment and a Kubernetes pod?
12. How do you expose a Kubernetes service externally?

13. What are liveness and readiness probes in Kubernetes, and why are they important?
14. Describe the concept of a Kubernetes secret and its use cases.
15. How can you upgrade a Kubernetes cluster to a new version?
16. What is a Kubernetes persistent volume (PV), and how does it differ from a persistent volume claim (PVC)?
17. How do you manage configuration files (such as YAML manifests) for Kubernetes resources?
18. What is the purpose of a Kubernetes init container?
19. Explain the concept of a Kubernetes daemon set.
20. How can you perform rolling restarts for pods in a Kubernetes deployment?

✅ For Intermediate (20 Questions, 2-6 Years of Experience)

1. Explain the concept of a Kubernetes replica set.
2. How do you perform rolling updates in Kubernetes?
3. What is a Kubernetes ingress, and how does it work?
4. Describe the role of a Kubernetes scheduler.
5. How do you troubleshoot a pod that is not running as expected?
6. What are Kubernetes resource quotas, and how do they impact cluster management?
7. How can you secure communication between Kubernetes components?
8. What is a Kubernetes ConfigMap, and how do you use it?
9. Discuss the benefits of using Helm for managing Kubernetes applications.
10. How do you monitor and visualize Kubernetes cluster health?
11. What is a Kubernetes StatefulSet, and when would you use it?

12. How do you handle secrets and sensitive data in Kubernetes securely?
13. Discuss the benefits and drawbacks of using Helm charts for application deployment.
14. What is a Kubernetes custom resource (CR), and how can you create one?
15. How do you set resource limits and requests for containers in a Kubernetes pod?
16. Describe the process of setting up a Kubernetes ingress controller.
17. What is the role of a Kubernetes network policy, and how does it enhance security?
18. How can you horizontally autoscale a Kubernetes deployment based on CPU utilization?
19. Explain the concept of Kubernetes affinity and anti-affinity rules.
20. How do you troubleshoot and diagnose performance issues in a Kubernetes cluster?

✅ For Experienced Professionals (20 Questions, 7+ Years of Experience)

1. Explain the concept of a Kubernetes custom resource definition (CRD).
2. How do you handle rolling back a failed deployment in Kubernetes?
3. Discuss the challenges of managing stateful applications in Kubernetes.
4. What is the role of a Kubernetes operator?

5. How can you optimize resource utilization in a large-scale Kubernetes cluster?
6. Describe the process of setting up a multi-cluster Kubernetes federation.
7. How do you implement network policies in Kubernetes for security?
8. What are the differences between Helm v2 and Helm v3?
9. Discuss the use of Kubernetes Operators for managing databases.
10. How would you design a highly available and fault-tolerant Kubernetes architecture?
11. What is the role of a Kubernetes admission controller, and how can you customize it?
12. Discuss the use of Kubernetes custom metrics for autoscaling.
13. How do you manage multi-tenancy in a large Kubernetes cluster?
14. What is the difference between a Kubernetes job and a Kubernetes cron job?
15. How can you achieve high availability for etcd in a Kubernetes control plane?
16. Describe the process of setting up a Kubernetes federation for global deployments.
17. What are the best practices for securing Kubernetes API server endpoints?
18. How do you handle rolling updates for stateful applications in Kubernetes?
19. Discuss the use of Kubernetes Operators for managing complex applications.
20. How would you design a disaster recovery strategy for a critical Kubernetes workload?

Kubernetes : 5 Mini Projects to start with

Project 1: Scalable Web Application

Requirement: Create a highly available website that can handle varying traffic loads.

Summary: Deploy a containerized web application, use load balancing for distribution, and configure autoscaling rules to handle surges in requests.

Tools/Services:

- Docker (or another container engine) for building container images.
- Kubernetes Services for load balancing.
- Horizontal Pod Autoscalers for automatic scaling.

Step to Follow:

Step 1: Containerize Your Web Application

- **Create a Dockerfile:** This is like a recipe for building your application's image. Here's a simple example using Nginx (a popular web server):

```
Dockerfile

FROM nginx:alpine
COPY /src/html /usr/share/nginx/html
```

where:

FROM nginx:alpine: Starts with a small Nginx image.

COPY /src/html /usr/share/nginx/html: Copies your web content (index.html, etc.).

- **Build the Image:** Run this in your project directory:

  # docker build -t my-web-app .

Step 2: Deployment Time!

- **Deployment File:** A YAML file defines your Kubernetes Deployment (manages your app's pods). Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-web-app
  template:
    metadata:
      labels:
        app: my-web-app
    spec:
      containers:
      - name: web-app
        image: my-web-app
        ports:
        - containerPort: 80
```

Where:
replicas: 2 (Starts 2 pods for your app.)

selector: Labels (help Kubernetes manage the pods)

- **Create it:** Use kubectl (your Kubernetes controller):

kubectl apply -f deployment.yaml

Step 3: Expose with a Service

- **Service File:** Another YAML defining a 'Service', responsible for balancing traffic to your pods.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  selector:
    app: my-web-app
  type: LoadBalancer
  ports:
  - port: 80
```

where:
type: LoadBalancer (Makes the service accessible externally (the way this works depends on your cloud provider, if any)).

- **Apply it:**
  # kubectl apply -f service.yaml

Step 4: Autoscaling (Optional but Awesome)

- **Define Rules:** A Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pods based on metrics like CPU usage.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app-deployment
  minReplicas: 2
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

- **Let's activate:**

```
# kubectl apply -f hpa.yaml
```

Congrats! You've got the basics of a scalable web app.

Project 2: Deploy a Self-Healing Microservice

Requirement: Ensure a microservice is always available, even in case of pod failures.

Summary: Deploy a microservice with multiple replicas, utilize health checks (liveness and readiness probes), and set up automatic redeployment of failed pods.

Tools/Services:

- Docker for building images.
- Kubernetes Deployments to manage multiple replicas.
- Kubernetes Liveness & Readiness probes for health monitoring.

Project 3: Database with Persistent Storage

Requirement: Deploy a database that retains data even if pods are rescheduled.

Summary: Provision a database (e.g., PostgreSQL, MySQL), and attach persistent storage to hold data securely.

Tools/Services:

- Docker for containerizing the database.
- Kubernetes Persistent Volumes & Persistent Volume Claims for storage management.
- A database of your choice.

Project 4: CI/CD Pipeline with Jenkins

Requirement: Automate build, test, and deployment of a containerized app.

Summary: Set up Jenkins within Kubernetes and create a pipeline that pulls code changes, builds a new container image, tests it, and deploys the update to an environment.

Tools/Services:

- Docker for building container images.
- Jenkins deployed as a Kubernetes Pod.
- A source code repository (e.g., GitLab, GitHub).

Project 5: Centralized Logging with Fluentd

Requirement: Collect and store log data from all the containers in your Kubernetes cluster.

Summary: Deploy Fluentd as a DaemonSet to gather logs from containers, and ship them to a storage and analysis platform like Elasticsearch.

Tools/Services:

- Fluentd deployed in Kubernetes.
- Elasticsearch (can be outside Kubernetes) for log storage and analysis.
- Kibana (optional) for log visualization.

Bonus

(Hands-on Lab & Other documents)

Azure Hands-on Labs (Guide)
https://techyoutube.com/index.php/2024/01/26/12-azure-hands-on-labs-elevate-your-expertise-now/

AWS Hands-on Labs (Guide)
https://techyoutube.com/index.php/2023/12/26/24-aws-hands-on-labs-elevate-your-expertise-now/

Hope you find this document helpful for your DevOps Learning.

For more such free content you can check :

https://techyoutube.com

Now, to Support, just follow me on below socials (No Cheating Please)

Telegram: https://t.me/LearnDevOpsForFree

Twitter: https://twitter.com/techyoutbe

Youtube: https://www.youtube.com/@T3Ptech