

Real DevOps Questions

Asked In Interview Answered



By DevOps Shack

[Click Here for DevOps & DevSecOps BootCamp](#)

DevOps Questions Asked In Interview Q&A

1. You've been asked to set up a highly available Kubernetes cluster for a production environment. How would you approach it?

Setting up a **Highly Available (HA) Kubernetes cluster** means building a system that ensures no single point of failure. This involves planning the cluster control plane, etcd database, networking, and worker nodes with redundancy, failover, and load balancing.

Key Components in HA Setup:

1. Control Plane Redundancy

- Minimum 3 control plane (master) nodes to ensure quorum in etcd
- Spread across availability zones (AZs) in cloud environments

2. etcd Cluster Setup

- Distributed key-value store
- Minimum 3 odd-numbered nodes (5 recommended for larger clusters)
- Secure communication with TLS
- etcdctl and etcd metrics are used to monitor health

3. Load Balancer in front of Control Planes

- External (e.g., AWS ELB or HAProxy)
- Accepts requests to kube-apiserver and forwards to healthy masters

4. Worker Nodes

- Minimum 2 or more
- Join using kubeadm join or managed via autoscaling groups
- Run workloads; need to be distributed across AZs

5. Networking

- Use CNI plugins (Calico, Flannel, Cilium) for inter-pod communication
- Configure pod CIDR ranges carefully

6. Storage

- Dynamic storage provisioners (EBS, GCP PD, etc.)
- PersistentVolumes backed by cloud-native or NFS storage

7. Ingress & DNS

- Ingress controller (Nginx, Traefik)
- CoreDNS deployed as a replica set

8. HA Tools

- Use **kubeadm HA mode** or **Kubespray** for setup
- Can also use **K3s** with embedded etcd for lightweight clusters

Example kubeadm HA Architecture:

- 3 Master nodes
- 1 HAProxy Load balancer (or AWS NLB)
- 3 etcd nodes (can be co-located or separate)
- Multiple worker nodes
- TLS certificates and RBAC configured

Commands (using kubeadm):

```
# Initialize first master with HA flags
```

```
kubeadm init --control-plane-endpoint "LB_DNS:6443" \  
--upload-certs --pod-network-cidr=192.168.0.0/16
```

```
# Join other masters
```

```
kubeadm join LB_DNS:6443 --token <token> \  
--discovery-token-ca-cert-hash sha256:<hash> \  
--control-plane --certificate-key <key>
```

```
# Join worker
```

```
kubeadm join LB_DNS:6443 --token <token> \  
--discovery-token-ca-cert-hash sha256:<hash>
```

2. How do you synchronize or set the correct system time on a Linux server?

Incorrect system time can lead to SSL/TLS issues, misaligned logs, and broken cron jobs. It's critical for distributed systems like Kubernetes, CI/CD, or time-based secrets.

Tools to Use:

- ntpd – Older method, daemon-based
- chronyd – Modern, faster synchronization
- timedatectl – Unified CLI to control time

Step-by-step with chrony:

```
# Install chrony
```

```
sudo apt install chrony -y
```

```
# Check NTP source configuration
```

```
cat /etc/chrony/chrony.conf
```

```
# Restart the service
```

```
sudo systemctl restart chrony
```

```
# Verify NTP is syncing
```

```
chronyc tracking
```

```
chronyc sources
```

Use timedatectl to set or sync:

```
# Check current time and NTP status
```

```
timedatectl
```

```
# Enable automatic NTP
```

```
timedatectl set-ntp true
```

If a specific time server is needed (e.g., ntp1.example.com):

```
sudo nano /etc/systemd/timesyncd.conf
```

```
# Add: NTP=ntp1.example.com
```

```
sudo systemctl restart systemd-timesyncd
```

3. What is the use of top and htop commands in Linux? When would you use each?

Both are used for real-time **system resource monitoring**, especially for CPU, memory, and process management.

top (built-in):

```
top
```

- Interactive CLI-based process viewer
- Displays processes by CPU usage
- Press k to kill, P to sort by CPU, M for memory

htop (advanced, colorful, more interactive):

```
sudo apt install htop
```

```
htop
```

- Better UI with color-coded graphs
- Navigate with arrow keys
- Tree view of process hierarchies
- Easily filter/search with /

When to use what?

Scenario	Use
Minimal Linux servers (no htop)	top
Need interactive tree view & mouse	htop
Remote server with no extras	top
Debugging hung processes	htop

4. What is the difference between head and tail commands? Write the syntax for each.

Both are used for **reading a specific portion** of files from either the beginning (head) or end (tail).

head:

Shows **first N lines**

```
head -n 10 filename.log # First 10 lines
```

tail:

Shows **last N lines**

```
tail -n 10 filename.log # Last 10 lines
```

Bonus: Real-time log monitoring

```
tail -f /var/log/syslog # Follow live logs
```

5. How can you interact with a Kubernetes cluster without using kubectl?

Even if kubectl is not installed, you can **interact with Kubernetes directly via its REST API**.

Option 1: Use curl with kubeconfig

```
export K8S_API=https://<cluster-ip>:6443
curl --cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/admin.crt \
--key /etc/kubernetes/pki/admin.key \
$K8S_API/api/v1/namespaces
```

Option 2: Use client libraries (Python)

```
from kubernetes import client, config
config.load_kube_config()
```

```
v1 = client.CoreV1Api()
print(v1.list_pod_for_all_namespaces())
```

Option 3: Use Dashboard, Lens, or K9s

- Kubernetes Dashboard: GUI-based
 - Lens IDE: Cross-platform Kubernetes GUI
 - K9s: Terminal UI for Kubernetes
-

6. What methods can be used to set up a Kubernetes cluster? Which one would you choose and why?

Method	Description	Use Case
kubeadm	Manual install; lets you build clusters from scratch	Full control, best for learning & custom setup
kOps	Production-ready clusters on AWS	Best for AWS users with HA needs
RKE	Rancher Kubernetes Engine for easy setup	Easy Docker-based provisioning
Minikube	Local, single-node cluster	Best for local testing
EKS/GKE/AKS	Managed by AWS/GCP/Azure	Enterprise-ready, less operational overhead

Why I'd choose managed Kubernetes (EKS/GKE/AKS):

- Automated upgrades
- Scalability
- Integrated logging, monitoring, and IAM
- Great for production workloads

7. Explain a complete CI/CD pipeline and its stages.

A CI/CD pipeline automates the software development lifecycle — from code commit to deployment. It ensures quality, consistency, and speed in delivery.

CI/CD Pipeline Stages (Breakdown)

1. Code Commit (Trigger Stage)

- Developers push code to Git repositories (GitHub, GitLab, Bitbucket).
- A webhook triggers the CI/CD pipeline.

git push origin feature/login

2. Build Stage

- Source code is compiled (Java, C#, etc.)
- Dependencies are downloaded.

-
- Output: executable artifact (e.g., .jar, .dll, .js, .pyc)

Example (Java with Maven):

```
mvn clean package
```

Output: target/app.jar

3. Test Stage

- Unit Tests: Validate small code blocks
- Integration Tests: Check service integrations
- Code Coverage tools: JaCoCo, Istanbul

```
npm test      # Node.js
```

```
pytest      # Python
```

```
mvn test      # Java
```

4. Security & Linting

- Static Analysis: SonarQube
- Secrets Detection: Gitleaks
- Vulnerability Scanning: Trivy, Snyk

```
trivy image myapp:latest
```

5. Packaging

- Build Docker image (for containerized apps)

```
docker build -t myorg/app:v1 .
```

6. Artifact Upload

- Push to a central registry
 - Docker Hub, ECR → container images
 - Nexus, Artifactory → .jar, .zip

```
docker push myorg/app:v1
```

7. Deployment Stage

- Kubernetes: kubectl apply -f
 - Helm: helm upgrade
 - Serverless: AWS Lambda deploy
 - SSH/Ansible for bare-metal
-

8. Monitoring/Notification

- Post-deploy tests, health checks
 - Slack/email alerts, Prometheus metrics
-

Tools Involved:

Stage	Tools
VCS	GitHub, GitLab
CI Engine	Jenkins, GitLab CI, GitHub Actions
Build Tools	Maven, npm, pip
Test Frameworks	JUnit, Mocha, PyTest
Security	Trivy, SonarQube, Gitleaks
Docker Registry	Docker Hub, ECR
CD Tools	ArgoCD, Helm, kubectl
Alerts/Monitoring	Slack, Prometheus, Grafana

8. Write a CI/CD pipeline for a Node.js application and explain each stage.

Here's a sample CI/CD flow using **GitHub Actions** for a Node.js app:

```
name: Node.js CI/CD Pipeline
```

```
on:  
  push:  
    branches: [ "main" ]
```

```
jobs:
```

```
build-test:  
  runs-on: ubuntu-latest  
  steps:  
    - uses: actions/checkout@v2  
    - name: Set up Node  
      uses: actions/setup-node@v2  
      with:  
        node-version: '18'  
  
    - name: Install dependencies  
      run: npm install  
  
    - name: Run tests  
      run: npm test  
  
    - name: Run linter  
      run: npm run lint  
  
docker:  
  needs: build-test  
  runs-on: ubuntu-latest  
  steps:  
    - uses: actions/checkout@v2  
  
    - name: Build Docker image  
      run: docker build -t myorg/nodeapp:${{ github.sha }} .  
  
    - name: Push to DockerHub  
      run: |  
        echo "${{ secrets.DOCKER_PASS }}" | docker login -u ${{ secrets.DOCKER_USER }} --  
        password-stdin  
        docker push myorg/nodeapp:${{ github.sha }}  
  
deploy:  
  needs: docker  
  runs-on: ubuntu-latest  
  steps:  
    - name: Deploy to Kubernetes  
      run: kubectl apply -f k8s/deployment.yaml
```

Each job (build-test, docker, deploy) runs in sequence.

9. How would you set up Node.js on a Linux system for a new project?

 **Steps:**

```
# Install prerequisites
```

```
sudo apt update
```

```
sudo apt install curl -y
```

```
# Install Node.js via NodeSource
```

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
```

```
sudo apt install nodejs -y
```

```
# Verify installation
```

```
node -v
```

```
npm -v
```

 **Initialize a new project:**

```
mkdir myapp && cd myapp
```

```
npm init -y
```

 **Create an app:**

```
// index.js
```

```
const express = require('express');
```

```
const app = express();
```

```
app.get('/', (req, res) => res.send('Hello World'));
```

```
app.listen(3000, () => console.log('Running on 3000'));
```

```
npm install express
```

```
node index.js
```

10. What is the purpose of tail and head in log monitoring? Provide syntax examples.

- head → First few lines of a file**

```
head -n 50 /var/log/syslog # Shows top 50 lines
```

- tail → Last few lines (great for latest logs)**

```
tail -n 50 /var/log/syslog # Last 50 lines
```

```
# Live log monitoring
```

```
tail -f /var/log/nginx/access.log
```

- Combine with grep**

```
tail -f /var/log/syslog | grep "error"
```

Used to:

- Monitor live logs
- Check startup issues
- Trace crash logs or shutdowns

11. A production server is showing the wrong time. How would you sync with an NTP server?

Step-by-step with chronyd:

```
# Install chrony
```

```
sudo apt install chrony
```

```
# Edit config
```

```
sudo nano /etc/chrony/chrony.conf
```

```
# Add preferred NTP servers
```

```
# Restart & enable
```

```
sudo systemctl restart chrony
```

```
sudo systemctl enable chrony
```

```
# Verify
```

```
chronyc tracking
```

```
timedatectl
```

You can also force sync:

```
sudo chronyc makestep
```

Or using ntpdate (one-shot):

```
sudo ntpdate time.google.com
```

12. You're troubleshooting a large log file and need to view the top and bottom. What is the use of head and tail?

Log files can be GBs in size. You don't want to open them in an editor like vim. Instead:

```
head -n 100 /var/log/app.log # See earliest logs
```

```
tail -n 100 /var/log/app.log # See latest logs
```

Use in combination:

```
cat /var/log/app.log | head -n 50
```

```
cat /var/log/app.log | tail -n 50
```

Live tailing:

```
tail -f /var/log/app.log
```

Use with grep, awk, or sed to filter important events.