DevOps Interview Deep Dive

Advanced Docker, Kubernetes & Cloud Engineering

Production-Ready Solutions & Best Practices

Sandhya | DevOps Engineer

Update Config Without Rebuilding Images 🔄

Question 11: Decoupling Configuration from Container Images

The Problem

Hardcoding configuration inside Docker images creates **rigid, environment-specific builds**. Every config change (DB credentials, API endpoints, feature flags) forces a complete rebuild, retag, and redeploy cycle — violating the **"build once, deploy everywhere"** principle.

> 💡 **Core Principle**
> Images should be **immutable artifacts** containing only code and dependencies. Configuration is **injected at runtime** based on the environment.

Industry-Standard Solutions

1. **Environment Variables:** Pass via `-e` flag or `env_file` in docker-compose. Kubernetes: `env` / `envFrom` in pod specs.

2. **ConfigMaps (K8s):** Store non-sensitive config. Mount as volumes or inject as env vars. Update ConfigMap → rolling restart picks up changes.

Practical Example

```
# ❌ BAD: Hardcoded in Dockerfile FROM node:18 COPY . /app ENV DB_HOST=prod-db.example.com CMD ["npm", "start"] # ✅ GOOD: Config via environment FROM node:18 COPY . /app # No ENV — injected at runtime CMD ["npm", "start"]
```

```
# Kubernetes ConfigMap apiVersion: v1 kind: ConfigMap metadata: name: app-config data: DB_HOST: "prod-db.example.com" LOG_LEVEL: "info" # Inject into Pod envFrom: - configMapRef: name: app-config
```

Comparison

Method	Use Case	Update Speed
Env Vars	Simple key-value	Instant (restart pod)
ConfigMap	Complex config, files	Propagates in ~1 min

3 **Secrets (K8s):** For sensitive data (passwords, keys). Base64 encoded, RBAC-protected. External: AWS Secrets Manager, Vault.

4 **Volume Mounts:** Mount config files from host or orchestrator. Decouples file-based config (nginx.conf, app.yaml) from image.

Method	Use Case	Update Speed
Secrets	Credentials, certs	Secure, RBAC

✅ **Best Practice**

Use **12-factor app methodology**: Store config in the environment. Never commit secrets to Git. Use tools like `sealed-secrets` or `external-secrets-operator` for GitOps workflows.

Reduce Docker Image Size 📦

Question 12: Optimization Techniques for Production Images

Why Image Size Matters

Large images increase **pull time, storage costs, attack surface, and deployment latency**. A 2GB image vs 100MB image means 20x slower pulls across a 100-node cluster — directly impacting scaling speed and costs.

Proven Optimization Strategies

1. **Multi-Stage Builds:** Build in one stage (includes compilers, tools), copy artifacts to slim runtime image. Example: Build Go binary with `golang:1.21`, run in `alpine`.

2. **Use Minimal Base Images:** `alpine` (5MB), `distroless` (no shell, minimal attack surface), or `scratch` (for static binaries).

3. **Layer Optimization:** Combine RUN commands with &&. Clean up in same layer (`apt-get clean && rm -rf /var/lib/apt/lists/*`).

Multi-Stage Build Example

```
# Stage 1: Build FROM golang:1.21 AS builder WORKDIR /app COPY . . RUN go build -o myapp # Stage 2: Runtime (only 15MB!) FROM alpine:3.18 RUN apk --no-cache add ca-certificates COPY --from=builder /app/myapp /myapp CMD ["/myapp"] # Result: 1.2GB build image → 15MB runtime
```

Layer Optimization

```
# ❌ BAD: Each RUN creates a layer RUN apt-get update RUN apt-get install -y curl RUN apt-get clean # ✅ GOOD: Single layer, cleanup included RUN apt-get update && \ apt-get install -y curl && \ apt-get clean && \ rm -rf /var/lib/apt/lists/*
```

Size Comparison

4 **.dockerignore:** Exclude `node_modules`, `.git`, test files, docs from build context.

5 **Remove Dev Dependencies:** For Node: `npm ci --production`. For Python: `pip install --no-cache-dir`.

Approach	Size	Use Case
ubuntu:22.04 base	~800MB	❌ Avoid for prod
node:18 (full)	~900MB	Dev only
node:18-alpine	~170MB	✅ Production
distroless/nodejs	~120MB	✅ Secure prod
Multi-stage Go	~15MB	✅ Best for static

⚠️ **Trade-off**

`alpine` uses `musl libc` instead of `glibc`. Some binaries may fail. `distroless` has no shell — debugging requires ephemeral containers or `kubectl debug`.

Deploying to Amazon EKS

Question 13: End-to-End EKS Deployment Workflow

Deployment Overview

EKS (Elastic Kubernetes Service) is AWS-managed Kubernetes. Deployment involves **cluster provisioning, IAM/RBAC setup, containerized app deployment, ingress/service exposure, and monitoring**. Here's the production workflow I follow:

Step-by-Step Process

1 **Provision EKS Cluster:** Use Terraform or `eksctl`. Define VPC, subnets, node groups (on-demand/spot), OIDC provider for IRSA.

```
eksctl create cluster --name prod-cluster --region us-west-2 --nodegroup-name workers --node-type t3.medium --nodes 3
```

2 **Configure kubectl:** Update kubeconfig to point to EKS cluster.

3 **Containerize App:** Build Docker image, push to ECR.

4 **Deploy to EKS:** Apply Kubernetes manifests (Deployment, Service, Ingress).

```
# deployment.yaml apiVersion: apps/v1 kind: Deployment metadata: name: myapp spec: replicas: 3 selector: matchLabels: app: myapp template: metadata: labels: app: myapp spec: containers: - name: myapp image: 123456.dkr.ecr.us-west-2.amazonaws.com/myapp:v1.0 ports: - containerPort: 8080
```

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

5 **Setup Ingress:** Install AWS Load Balancer Controller, create Ingress resource → ALB auto-provisioned.

6 **Monitoring:** Deploy Prometheus, Grafana, CloudWatch Container Insights for metrics/logs.

```
aws eks update-kubeconfig --name prod-cluster --region
us-west-2
```

3 **Build & Push Image:** Tag image with ECR registry URL, authenticate, push.

```
docker build -t myapp:v1.0 .
docker tag myapp:v1.0 123456.dkr.ecr.us-west-
2.amazonaws.com/myapp:v1.0
aws ecr get-login-password | docker login --username
AWS --password-stdin 123456.dkr.ecr.us-west-
2.amazonaws.com
docker push 123456.dkr.ecr.us-west-
2.amazonaws.com/myapp:v1.0
```

🔐 IAM Best Practice

Use **IRSA (IAM Roles for Service Accounts)** instead of node IAM roles. Annotate service account with IAM role ARN — fine-grained permissions per pod.

Troubleshooting Slow Service

Question 14: Isolating Performance Issues in Microservices

Diagnostic Approach

When one service degrades while others remain healthy, follow a **layered investigation**: application → container → network → infrastructure. Avoid assumptions — metrics drive diagnosis.

Step-by-Step Troubleshooting

1 **Check Pod Health:** `kubectl get pods` — look for CrashLoopBackOff, OOMKilled, high restart count.

2 **Resource Metrics:** `kubectl top pod` — CPU/memory usage. If near limits, it's resource starvation.

3 **Application Logs:** `kubectl logs <pod> --tail=100` — errors, timeouts, slow queries.

4 **Describe Pod:** `kubectl describe pod <name>` — events show scheduling issues, failed probes, volume mount

7 **Node Health:** `kubectl top nodes` — node resource pressure. Check disk I/O with `iostat` on node.

8 **APM Tracing:** Use Jaeger, Datadog, or AWS X-Ray to trace request flow — identify slow downstream calls.

Common Culprits

Symptom	Root Cause	Fix
High CPU	Infinite loop, inefficient code	Profile app, optimize
High memory	Memory leak, large datasets	Heap dump analysis
OOMKilled	Limit too low	Increase memory limit
Slow response	DB query taking 5s+	Add index, optimize query
Connection timeout	Network policy blocking	Fix NetworkPolicy rules

errors.

5 **Network Latency:** Test inter-service connectivity: `kubectl exec -it <pod> -- curl http://other-service`. Check DNS resolution time.

6 **Database Connections:** If service talks to DB, check connection pool exhaustion, slow queries, lock contention.

Symptom	Root Cause	Fix

🔍 Pro Tip

Use `kubectl debug` to inject ephemeral container with debugging tools (curl, netstat, strace) into running pod without modifying image.

✅ Best Practice

Set up **SLOs (Service Level Objectives)** and alerts on P95/P99 latency. Detect slowness before users complain.

Blue-Green Deployment Strategy

Question 15: Zero-Downtime Deployments with Instant Rollback

How Blue-Green Works

Blue-Green deployment maintains **two identical production environments**: **Blue (current)** and **Green (new version)**. Traffic routes to Blue. Deploy new version to Green, test, then **switch traffic atomically**. If issues arise, switch back instantly — zero downtime, instant rollback.

Implementation Flow

1. **Blue Environment Running:** All traffic goes to Blue (v1.0). Green is idle or non-existent.

2. **Deploy to Green:** Provision Green environment with v2.0. Run smoke tests, health checks.

3. **Switch Traffic:** Update load balancer / service selector to point to Green. Traffic now hits v2.0.

4. **Monitor Green:** Watch metrics, logs, errors for 15-30

Kubernetes Implementation

```
# Blue Deployment (v1.0) apiVersion: apps/v1 kind:
Deployment metadata: name: myapp-blue spec:
replicas: 3 selector: matchLabels: app: myapp
version: blue template: spec: containers: - image:
myapp:v1.0 # Service points to Blue selector:
version: blue # Change to 'green' to switch
```

Benefits vs Challenges

Benefits	Challenges
✅ Zero downtime	⚠️ 2x infrastructure cost during switch
✅ Instant rollback	⚠️ Database migrations risky (schema divergence)
✅ Full testing in prod-like env	⚠️ Stateful apps harder (sessions, caches)
✅ Simple logic	⚠️ Requires traffic routing control

min.

5 **Rollback or Proceed:** If issues → switch back to Blue. If stable → keep Green, decommission Blue.

⚠️ Database Challenge

If Blue and Green use same DB, migrations must be **backward compatible**. Use **expand-contract pattern**: add new columns, deprecate old ones later.

✅ Cloud Implementation

AWS: Use **Route 53 weighted routing** or **ALB target groups**. GCP: **Traffic Director**. Azure: **Traffic Manager**.

Toughest EKS Challenge 💪

Question 16: Real-World Production War Story

The Challenge: Node Group AutoScaling Thrashing

Scenario: Production EKS cluster with Cluster Autoscaler and HPA. During traffic spikes, pods scaled up → new nodes launched. But pods failed to schedule due to **insufficient IP addresses** in VPC subnets. Nodes joined but stayed NotReady. CA kept launching more nodes, hitting AWS account limits.

Impact: Cascading failures. Existing pods evicted due to resource pressure. New deployments stuck in Pending. On-call escalation. Revenue-impacting downtime.

Root Cause Analysis

- **VPC Subnet Sizing:** /24 subnets → 251 usable IPs. With 50 nodes × 16 pods/node = 800 IPs needed. We ran out.
- **ENI Exhaustion:** Each node consumes one ENI. AWS limits per instance type. We hit the limit.

Solution & Learnings

1. **Immediate Fix:** Manually terminated NotReady nodes. Disabled CA temporarily. Scaled down non-critical workloads.
2. **VPC Redesign:** Created new /20 subnets (4096 IPs). Migrated workloads using blue-green cluster strategy.
3. **CNI Optimization:** Enabled **VPC CNI prefix delegation** — increases IPs per node by 10x without using more ENIs.
4. **Monitoring:** Added CloudWatch alarms on available IPs per subnet. Grafana dashboard for IP allocation trends.
5. **CA Tuning:** Set `--max-nodes-total` limit. Configured scale-down delay to prevent thrashing.

💡 Key Takeaway

- **Cluster Autoscaler Threshing:** CA launched nodes, but they couldn't get IPs → stayed NotReady → CA launched more → loop.

Always calculate **IP requirements** before sizing VPC subnets: `(nodes × pods_per_node) + overhead`. For EKS, use **/19 or larger** subnets in production.

🔧 VPC CNI Prefix Delegation

Enabled via: `kubectl set env daemonset aws-node -n kube-system ENABLE_PREFIX_DELEGATION=true`. Increases max pods/node from 29 to 110 on m5.large.

Managing Infrastructure with Terraform

Question 17: IaC Workflows and Real-World Challenges

Terraform Workflow

I use Terraform for **declarative infrastructure**: VPCs, subnets, EKS clusters, RDS, S3, IAM. Code lives in Git, changes reviewed via PR, applied via CI/CD (GitHub Actions or GitLab CI).

Standard Workflow

1. **Write/Modify .tf Files:** Define resources in HCL. Use modules for reusability.
2. **terraform init:** Initialize backend (S3 + DynamoDB for state locking), download providers.
3. **terraform plan:** Generate execution plan. Review in PR — team approval required.
4. **terraform apply:** Apply changes. Automated in CI for dev/staging, manual approval for prod.

Challenges Encountered

State Drift — Challenge

Manual changes in AWS Console caused drift. Fixed: `terraform refresh` + import. Enforced policy: "All changes via Terraform."

State Locking Failures — Challenge

Concurrent `apply` from two devs → DynamoDB lock conflict. Solution: Enforce CI/CD only, disable local applies for prod.

Destroy Dependencies — Challenge

Destroying VPC failed — subnets still referenced by EKS. Solution: Use `depends_on` explicitly, destroy in reverse order.

Module Versioning — Challenge

Updated shared module broke 3 environments. Fix: Pin module versions: `source = "git::...?ref=v1.2.0"`.

5 **State Management:** Remote backend in S3, state locking via DynamoDB, versioning enabled.

✅ Best Practices

Use **workspaces** or separate state files per environment. Enable **state encryption** at rest. Run `terraform fmt` and `tflint` in CI. Use **Terraform Cloud** or **Atlantis** for GitOps automation.

Monitoring Tools & Components 📊

Question 18: Observability Stack in Production

Monitoring Stack I Use

Tool	Purpose	Metrics
Prometheus	Metrics collection	Time-series data, scrapes exporters
Grafana	Visualization	Dashboards, alerting
ELK Stack	Log aggregation	Centralized logs, search
Jaeger	Distributed tracing	Request flow, latency breakdown
CloudWatch	AWS-native	EC2, RDS, Lambda metrics/logs
Datadog	APM + Infrastructure	Full-stack observability

Components Monitored

1 **Cluster Health:** Node CPU/memory/disk, pod count, pending pods, failed deployments.

4 **Database:** Connection pool size, slow queries, replication lag (RDS).

5 **Network:** Ingress/egress bandwidth, DNS query latency, service mesh metrics (if using Istio).

6 **Logs:** Error rates, log volume, specific error patterns (regex alerts).

Alert Examples

```
# Prometheus Alert Rule groups: - name: pod_alerts rules: - alert: HighPodMemory expr: container_memory_usage_bytes / container_spec_memory_limit_bytes > 0.9 for: 5m labels: severity: warning annotations: summary: "Pod {{ $labels.pod }} using 90%+ memory"
```

💡 Golden Signals

2 **Application Metrics:** Request rate, latency (P50/P95/P99), error rate, throughput.

3 **Resource Utilization:** Container CPU/memory limits vs usage, PVC storage.

Focus on **Latency, Traffic, Errors, Saturation** (Google SRE book). These four metrics catch 90% of issues early.

Best Practice

Use **kube-state-metrics** for Kubernetes object metrics. Deploy **node-exporter** for host metrics. Set up **PagerDuty** integration for critical alerts.

Troubleshooting Slow Server 🔍

Question 19: Systematic Performance Investigation

Investigation Workflow

Slow server = symptom, not root cause. Follow **USE Method** (Utilization, Saturation, Errors) across all resources: CPU, memory, disk, network.

Step-by-Step Diagnosis

1 **CPU Check:** `top` or `htop` — identify high CPU processes. Check load average vs core count.

```
uptime # Load avg: 8.5 on 4-core → overloaded
top -o %CPU # Sort by CPU usage
```

2 **Memory Check:** `free -h` — look for swap usage (indicates RAM exhaustion).

```
free -h # If swap used → memory pressure
vmstat 1 # Monitor swapping in real-time
```

5 **Process Inspection:** `ps aux` — zombie processes, runaway scripts.

```
ps aux --sort=-%mem | head -20 # Memory hogs
pstree -p # Process tree
```

6 **Logs:** `/var/log/syslog`, `/var/log/messages` — OOM killer, hardware errors.

```
dmesg | grep -i oom # OOM kills
journalctl -p err -since "1 hour ago" # Recent errors
```

7 **Application Level:** Check app logs, database slow query log, connection pool exhaustion.

Common Scenarios

3 **Disk I/O:** `iostat -x 1` — check %util. Near 100% = disk bottleneck.

```
iostat -x 1 # %util column
iotop # Which process doing I/O
```

4 **Network:** `iftop` or `nethogs` — bandwidth saturation, packet loss.

```
iftop # Live bandwidth usage
netstat -s | grep -i error # Packet errors
```

Symptom	Diagnosis	Fix
Load avg > cores	CPU saturation	Scale horizontally or optimize code
Swap usage high	RAM exhausted	Add RAM or kill memory leaks
Disk %util 100%	I/O bottleneck	Faster disk (SSD), optimize queries
Network drops	Packet loss	Check NIC, switch, bandwidth
OOM in logs	Process killed	Increase limits or fix leak

🔧 Advanced Tools

`strace -p <PID>` — trace system calls. `perf top` — CPU profiling. `lsof` — open files/sockets.

Summary & Key Takeaways 🎯

Essential DevOps Principles for Production Success

Core Competencies Covered

- ✅ **Configuration Management:** Decouple config from images using env vars, ConfigMaps, Secrets.
- ✅ **Image Optimization:** Multi-stage builds, minimal base images, layer optimization → 10x smaller images.
- ✅ **EKS Deployment:** End-to-end workflow from cluster provisioning to production deployment.
- ✅ **Troubleshooting:** Systematic diagnosis using metrics, logs, and layered investigation.
- ✅ **Blue-Green Strategy:** Zero-downtime deployments with instant rollback capability.
- ✅ **Production Challenges:** Real-world EKS IP exhaustion — planning and monitoring are critical.

Universal Best Practices

🎯 Automation First — Principle

Automate everything: CI/CD, infrastructure provisioning, scaling, backups. Manual operations don't scale.

📊 Observability is Non-Negotiable — Principle

If you can't measure it, you can't improve it. Metrics, logs, traces = production confidence.

🔒 Security by Default — Principle

Least privilege IAM, secrets encryption, network policies, container scanning. Security isn't optional.

🔄 Immutable Infrastructure — Principle

Never patch running systems. Deploy new versions. Cattle, not pets.

🚀 Final Thought

- ✅ **Terraform IaC:** GitOps workflow, state management, and common pitfalls to avoid.
- ✅ **Observability:** Comprehensive monitoring stack — Prometheus, Grafana, ELK, tracing.
- ✅ **Server Performance:** USE method for resource diagnosis — CPU, memory, disk, network.

DevOps is 20% tools, 80% culture. Focus on **collaboration, continuous improvement, and blameless postmortems**. Technology changes, but principles endure.