

DevOps Fundamentals You Must Own

Interview-ready answers with scenarios, troubleshooting flow, and practical commands

Prepared for interview practice. These answers are written in a practical SRE/DevOps style: explain the system behavior first, then give a structured debugging path, then show commands. Adapt service names, namespaces, cloud tooling, and thresholds to your environment.

How to use this guide

- Start with the interview answer (30-60 second explanation).
- Use the scenario to sound experienced and grounded.
- Mention 2-4 commands only in live interviews unless asked to go deeper.
- State trade-offs and blast radius before changing production settings.

Quick command toolbox (Linux/Kubernetes/Terraform)

CPU / memory / IO / network

```
top
htop
vmstat 1
iostat -xz 1
mpstat -P ALL 1
pidstat -dur 1
ss -s
ss -tanp
sar -u 1 5
sar -d 1 5
```

Kubernetes

```
kubectl get pods -A
kubectl describe pod <pod> -n <ns>
kubectl logs <pod> -n <ns> --previous
kubectl get events -n <ns> --sort-by=.lastTimestamp
kubectl top pod -n <ns>
kubectl rollout status deploy/<name> -n <ns>
```

Terraform

```
terraform fmt -check
terraform validate
terraform plan
terraform plan -refresh-only
terraform state list
terraform state show <addr>
```

Interview tip: explain what signal each command gives you. Example: iostat shows device saturation/await, while vmstat shows run queue and IO wait.

1. Real difference between CPU bottleneck vs I/O bottleneck, and how to identify each

Interview answer: A CPU bottleneck means the workload is compute-bound - requests wait because cores are busy executing instructions. An I/O bottleneck means the workload is waiting on an external resource - disk, network, database, DNS, or remote API. The fix is different: CPU bottlenecks usually need code optimization, parallelism tuning, or more CPU; I/O bottlenecks need latency reduction, batching, caching, queue tuning, or dependency scaling.

Scenario: Checkout API latency spikes from 120 ms to 2 s. CPU looks moderate at node level, but p99 is bad. We confirm app threads are waiting on database queries, not burning CPU.

How I would approach it

- Check symptoms first: high response time and timeouts do not automatically mean CPU pressure.
- Look at CPU saturation indicators: run queue, per-core utilization, user/sys split, throttling (in containers).
- Look at wait indicators: io wait, disk await, DB query latency, network retransmits, DNS resolution time.
- Identify where time is spent per request - app compute, DB, remote API, filesystem, locks.
- Pick a fix based on the constraint: optimize SQL and connection pool before scaling app replicas if DB is the real bottleneck.

Practical commands / checks

```
# CPU-bound signals
mpstat -P ALL 1
pidstat -u -p <pid> 1
top -H -p <pid>
kubectl top pod -n <ns> <pod>
kubectl describe pod <pod> -n <ns> | grep -i -A3 throttling

# IO-bound signals (host)
vmstat 1          # wa column (IO wait), r queue
iostat -xz 1      # util, await, svctm-like indicators
pidstat -d -p <pid> 1
iotop -oPa        # if available

# Network / dependency waiting
ss -s
ss -tanp | grep :443 | head
curl -w 'dns:%{time_namelookup} connect:%{time_connect} ttfb:%{time_starttransfer}
total:%{time_total}\n' -o /dev/null -s https://service.example
dig api.internal.example
```

What can break production

- Mistaking high CPU usage for CPU bottleneck - 70 percent CPU can still be fine if latency is low and no run queue backlog exists.
- Node averages hide hot threads - one core can be saturated while overall CPU looks normal.
- Scaling app replicas can worsen I/O bottlenecks by increasing load on DB/cache/downstream APIs.

Answer in interviews using the phrase: 'I first determine whether the service is spending time computing or waiting.'

2. Why systems fail even when CPU and memory look fine

Interview answer: Because most incidents are not caused by raw CPU or RAM exhaustion. Failures often come from saturation in queues, thread pools, connection pools, file descriptors, DNS, downstream dependencies, locks, rate limits, storage latency, kernel limits, or bad deploy/config changes. CPU and memory are only two signals in a larger system.

Scenario: Users get 504s after a feature release. Nodes show 35 percent CPU and 55 percent memory. Root cause turns out to be DB connection pool exhaustion plus retry storm from the new code path.

How I would approach it

- Check error budget signals first: status codes, timeout rates, p95/p99 latency, queue depth, retries.
- Look for resource types beyond CPU/RAM: file descriptors, sockets, conntrack, ports, DB pool, thread pool, disk latency.
- Check recent changes: deploy, config, feature flag, secret rotation, cert renewal, ingress rule.
- Inspect dependencies and backpressure behavior - one slow dependency can cascade into widespread failures.
- Verify kernel and runtime limits (ulimit, epoll, conntrack, max open files) and whether the process hit them.

Practical commands / checks

```
# Common hidden saturation checks
ulimit -n
cat /proc/sys/fs/file-max
lsof -p <pid> | wc -l
ss -s
ss -tan state time-wait | wc -l
cat /proc/sys/net/netfilter/nf_conntrack_count 2>/dev/null
cat /proc/sys/net/netfilter/nf_conntrack_max 2>/dev/null
```

```
# Application and platform clues
journalctl -u <service> --since '15 min ago'
dmesg | tail -100
kubectl get events -A --sort-by=.lastTimestamp | tail -50
kubectl describe ingress <name> -n <ns>
kubectl describe pod <pod> -n <ns>
```

What can break production

- Over-trusting dashboards with only CPU and memory panels.
- Ignoring queue depth and retries - retries can amplify the outage.
- Assuming 'no infra issue' means 'application bug only'; often it is interaction between both.

3. What actually happens when a Kubernetes Pod restarts

Interview answer: In practice, people often say 'pod restarted' when the container inside the pod restarted. The kubelet detects a container exit and restarts it based on restartPolicy, with exponential backoff (CrashLoopBackOff is the backoff state). The Pod object may stay the same while container restartCount increases. If the controller replaces the Pod (new ReplicaSet rollout, eviction, node issue), that is a new Pod with a new UID.

Scenario: A Java app OOMs during startup. The pod name stays the same, restartCount increases, and status shows CrashLoopBackOff. The kubelet is restarting the container with backoff while probes and events show failure details.

How I would approach it

- Differentiate container restart vs Pod replacement. Check Pod UID and restartCount.
- Read current and previous container logs. Previous logs are critical in crash loops.
- Check events for OOMKilled, probe failures, image pull issues, volume mount errors, or failed init containers.
- Confirm whether init containers completed and whether startup dependencies (DB, config, secret) are available.
- If it is a rollout issue, compare old vs new ReplicaSet, image tag, config checksum, and changes in env vars.

Practical commands / checks

```
kubectl get pod <pod> -n <ns> -o wide
kubectl get pod <pod> -n <ns> -o
  jsonpath='{.metadata.uid}{"\n"}{.status.containerStatuses[*].restartCount}{"\n"}'

kubectl describe pod <pod> -n <ns>
kubectl logs <pod> -n <ns> -c <container>
kubectl logs <pod> -n <ns> -c <container> --previous
kubectl get events -n <ns> --sort-by=.lastTimestamp | tail -30

# Check rollout context
kubectl rollout history deploy/<name> -n <ns>
kubectl get rs -n <ns> -l app=<app>
kubectl diff -f deployment.yaml
```

What can break production

- Only checking current logs and missing the crash reason in --previous logs.
- Assuming emptyDir persists across Pod replacement - it survives container restart in same Pod, but not new Pod creation.
- Changing many things at once while debugging a crash loop.

4. Difference between readiness and liveness probes, and how each can break production

Interview answer: Readiness controls whether a pod receives traffic. Liveness controls whether Kubernetes restarts the container. Readiness should fail when the app cannot serve requests correctly; liveness should fail only when the

process is unhealthy and cannot recover on its own. Bad probe design can create outages by either sending traffic too early or restarting healthy pods repeatedly.

Scenario: A service starts in 40 seconds after warming cache. Liveness probe is too aggressive and kills it every 15 seconds. In another case, readiness checks only 'process is up' so traffic is routed before DB migrations complete.

How I would approach it

- Use readiness to gate traffic during startup, dependency warmup, drain, and partial outages.
- Use liveness only for deadlocks or unrecoverable states. Avoid coupling it to transient downstream dependency errors.
- Tune initialDelaySeconds, timeoutSeconds, periodSeconds, and failureThreshold to match app behavior.
- Use startupProbe for slow-starting apps to protect liveness during initialization.
- During incidents, inspect probe events and compare probe endpoint latency vs normal request latency.

Practical commands / checks

```
kubectl describe pod <pod> -n <ns> | sed -n '/Liveness:/,/Readiness:/p'
kubectl get pod <pod> -n <ns> -o
  jsonpath='{.status.containerStatuses[*].state}{"\n"}'
kubectl logs <pod> -n <ns> --previous
kubectl exec -it <pod> -n <ns> -- sh -c 'wget -qO- http://127.0.0.1:8080/healthz
|| true'
kubectl exec -it <pod> -n <ns> -- sh -c 'wget -qO- http://127.0.0.1:8080/ready ||
true'
kubectl get events -n <ns> --sort-by=.lastTimestamp | grep -i probe
```

What can break production

- Liveness endpoint calling DB/cache and failing during transient dependency slowness - causes restart storm.
- Readiness endpoint too shallow - returns 200 before app is actually able to serve.
- Probe timeout too low for GC pauses or cold starts.

Good interview line: 'Readiness protects users from a pod; liveness protects the pod from itself.'

5. What problem infrastructure as code solves, and when it becomes dangerous

Interview answer: IaC solves manual drift, tribal knowledge, and unreproducible infrastructure by making changes versioned, reviewable, testable, and repeatable. It becomes dangerous when teams treat it as magic and apply large changes without plan review, environment isolation, state protection, or blast-radius controls.

Scenario: A team manually edits a security group in production to fix an incident. Two weeks later, Terraform apply removes the manual rule and causes another outage because the change was never codified.

How I would approach it

- Core value: desired state in code, reviewed via pull request, applied consistently across environments.
- Use modules and policies to standardize networking, IAM, tagging, backups, and monitoring.
- Protect production with small plans, approvals, targeted rollouts, and policy checks.
- Detect and reconcile drift intentionally; do not let emergency manual changes remain undocumented.
- Treat credentials, secrets, and state as sensitive assets because state can include resource IDs and secret values.

Practical commands / checks

```
# Safer IaC workflow
terraform fmt -check
terraform validate
terraform plan -out=tfplan
terraform show -no-color tfplan | less

# Drift check without changing resources
terraform plan -refresh-only

# Review changed resources only (JSON for automation)
terraform show -json tfplan > tfplan.json
```

What can break production

- Single repo apply touching too many environments at once.
- No remote backend locking - concurrent applies corrupt state.
- Secrets leaking into code or state outputs.

6. How Terraform state works, and what happens when it drifts

Interview answer: Terraform state maps Terraform resource addresses to real infrastructure objects and stores last-known attributes. During plan/apply, Terraform compares configuration, state, and provider-read actual values. Drift means reality changed outside Terraform (manual changes, autoscaling, cloud-side updates), so plan results may be surprising until state is refreshed and code is corrected.

Scenario: An engineer manually changes an RDS parameter and subnet attachment in AWS console. Next Terraform plan shows unexpected updates and possible replacement. We run refresh-only, review drift, codify approved changes, and avoid accidental recreation.

How I would approach it

- State is not the source of truth by itself - it is Terraform's memory. Providers read the live system during refresh.
- Store state remotely with locking and versioning (for example S3 + DynamoDB lock or Terraform Cloud).
- When drift is detected, decide: accept drift into code, revert manual change, or import/move state entries.
- Use lifecycle settings carefully (ignore_changes, prevent_destroy) only when you understand trade-offs.
- Back up state before state surgery, and never edit state files casually.

Practical commands / checks

```
# Inspect and reconcile drift
terraform state list
terraform state show aws_db_instance.main
terraform plan -refresh-only
terraform apply -refresh-only # if your process allows and after review

# Import existing resources
terraform import aws_s3_bucket.logs my-prod-logs-bucket

# Refactor addresses safely
terraform state mv module.old.aws_iam_role.app module.new.aws_iam_role.app

# Remove orphaned state mapping (does not destroy remote object)
terraform state rm aws_instance.legacy
```

What can break production

- Using ignore_changes to hide important drift forever.
- Blindly applying a plan that includes replacement of stateful resources.
- Manual state edits without backup and peer review.

7. Difference between horizontal and vertical scaling, and when to avoid autoscaling

Interview answer: Horizontal scaling adds more instances or pods. Vertical scaling increases CPU/RAM of existing instances. Horizontal scaling improves redundancy and throughput for stateless services, while vertical scaling can help stateful or memory-bound workloads with limited horizontal partitioning. Avoid autoscaling when scaling creates instability, cost explosions, or pushes the bottleneck into a fixed dependency like a database.

Scenario: Traffic spikes 5x during a campaign. App pods autoscale from 10 to 80, but DB CPU hits 95 percent and latency collapses. Real fix is protecting DB with caching, query optimization, and rate limiting instead of unlimited app HPA growth.

How I would approach it

- Find the true bottleneck first. Autoscaling the wrong tier can amplify failures downstream.
- Horizontal scale is preferred for stateless APIs if sessions/state are externalized.
- Vertical scale is useful for JVM memory pressure, databases, and workloads with high shared state or licensing constraints.
- Avoid autoscaling when startup is slow, requests are non-idempotent, queues are not backpressured, or metrics are noisy.
- Use guardrails: max replicas, scale-up/down stabilization, PDBs, and dependency-aware limits.

Practical commands / checks

```
# Kubernetes autoscaling checks
kubectl get hpa -A
kubectl describe hpa <name> -n <ns>
```

```
kubectl top pod -n <ns>
kubectl top node
kubectl get pdb -n <ns>
kubectl describe deploy <name> -n <ns>
```

```
# Queue/dependency signals (examples)
redis-cli info stats
psql -c "select now(), count(*) from pg_stat_activity;"
```

What can break production

- Autoscaling on CPU only when the actual limiter is request latency or queue depth.
- No cooldown/stabilization causing scale oscillation (thrashing).
- Scaling app tier without scaling connection pool limits, DB capacity, or cache nodes.

8. Why deployments succeed but users still see errors

Interview answer: A deployment can succeed at the orchestrator level while the user journey fails. Kubernetes reports rollout success when pods become ready, but users can still hit errors due to bad routing, incompatible schema changes, stale caches, missing secrets, partial rollout, client-side caching, CDN issues, or dependency mismatches.

Scenario: Deployment status is green, but users see 500s on checkout. Pods are ready, yet the new app version expects a DB column that the migration job failed to create in one region.

How I would approach it

- Separate platform success from business success: rollout status is not equal to user success.
- Check post-deploy KPIs: error rate, p95 latency, conversion, queue lag, and dependency health.
- Validate all layers: Ingress/LB routing, app version skew, DB migration status, feature flags, config maps, secrets, and CDN cache.
- Compare successful vs failing requests by path, region, tenant, and app version.
- Use progressive delivery or canaries with automated rollback based on user-facing metrics.

Practical commands / checks

```
kubectl rollout status deploy/<name> -n <ns>
kubectl get pods -n <ns> -l app=<app> -o wide
kubectl logs deploy/<name> -n <ns> --tail=200
kubectl describe ingress <name> -n <ns>
kubectl get endpointslice -n <ns> | grep <service>

# Validate from user path perspective
curl -I https://app.example.com/health
curl -H 'Host: app.example.com' http://<ingress-lb-ip>/critical-path
kubectl exec -it <pod> -n <ns> -- printenv | sort | grep -E 'FEATURE_|DB_|REDIS_'
```

What can break production

- Only checking deployment status and not checking error rates after release.
- Running non-backward-compatible schema changes with mixed app versions.
- Readiness endpoint too shallow, marking pods ready before key dependencies are reachable.

9. How to decide between rollback vs hotfix during an incident

Interview answer: Default to the fastest safe path to restore service. Roll back when the incident is clearly linked to a recent change and rollback is low risk, fast, and well understood. Hotfix when rollback is impossible, risky (for example schema already migrated), or when the issue is in data/config that needs forward correction. The decision is about time-to-recovery and blast radius, not ego.

Scenario: A release introduces a CPU spike and elevated 500s. If the change is isolated and schema is backward compatible, rollback is fastest. In a different incident, a destructive migration already ran, so rollback would break reads; we apply a hotfix and feature-flag disable.

How I would approach it

- Ask first: what changed, when, and can we safely revert all of it?
- Estimate recovery time for rollback vs hotfix, including validation time.
- Check compatibility constraints: DB schema, message formats, cache keys, API contracts, and data migrations.
- Prefer feature flags and config toggles when they can stop the bleeding quickly.

- Document the decision and assign one owner to drive recovery while others gather evidence.

Practical commands / checks

```
# Rollback paths (Kubernetes)
kubectl rollout history deploy/<name> -n <ns>
kubectl rollout undo deploy/<name> -n <ns>
kubectl rollout status deploy/<name> -n <ns>

# Config/feature flag mitigation (examples)
kubectl set env deploy/<name> FEATURE_X=false -n <ns>
kubectl rollout restart deploy/<name> -n <ns>

# Verify impact
kubectl get pods -n <ns> -l app=<app>
kubectl logs deploy/<name> -n <ns> --since=10m | tail -100
```

What can break production

- Choosing hotfix because 'we can fix it quickly' while users continue failing.
- Rolling back app code after irreversible schema/data changes without compatibility checks.
- Making multiple emergency changes simultaneously and losing auditability.

Interview phrase: 'I optimize for MTTR first, then for root cause and permanent fix.'

10. Difference between metrics, logs, and traces, and when to start with each

Interview answer: Metrics are numeric time-series summaries (fast for detection and trends). Logs are discrete event records with detail (good for errors and context). Traces follow a request across services (best for latency breakdown and dependency attribution). Start with metrics to detect and scope, then use traces to find where time is spent, then logs to inspect exact failures. But if you already know a specific error signature, logs may be the fastest entry point.

Scenario: Users report intermittent slow checkout. Metrics show p99 latency spike and errors in one region. Traces reveal most time in payment service calling a third-party API. Logs then confirm timeout and retry behavior with a specific partner endpoint.

How I would approach it

- Metrics answer: 'Is there a problem? How big? When did it start? Which service/region/tenant?'
- Traces answer: 'Where is the request spending time across hops?'
- Logs answer: 'What exactly failed in this request/process and with what context?'
- Use correlation IDs and trace IDs in logs so you can pivot quickly between tools.
- Do not rely on one signal alone; combine at least two before making risky production changes.

Practical commands / checks

```
# Metric first (examples)
# Prometheus query examples (adapt labels)
rate(http_requests_total{job="checkout",status=~"5.."}[5m])
histogram_quantile(0.99,
  sum(rate(http_request_duration_seconds_bucket{job="checkout"}[5m])) by (le))
sum(rate(container_cpu_usage_seconds_total{namespace="prod"}[5m])) by (pod)

# Logs (Kubernetes)
kubectl logs deploy/checkout -n prod --since=15m | grep -i 'timeout\|error'

# Trace correlation (example pattern)
# Find trace_id in logs, then open it in Jaeger/Tempo/X-Ray and inspect spans
```

What can break production

- Starting with logs across all services before scoping the blast radius - too noisy.
- Metrics with high cardinality labels causing dashboard and alert pain.
- Missing correlation IDs makes traces and logs hard to connect.

Final interview tips

- Use a structure: symptom -> scope -> hypothesis -> checks -> mitigation -> prevention.
- Say what you would measure before and after the fix.
- Mention blast radius and rollback plan whenever you propose a production change.
- Be explicit about trade-offs (speed vs safety, temporary mitigation vs permanent fix).
- If you do not know an exact command, explain the signal you need and where you would get it.

Practice exercise

Pick any recent incident from your work and answer these ten questions against that real event. This makes your interview answers sound natural and credible.

Use this as a mock interview sheet: answer each question in under 60 seconds, then go deeper with one real example and 2-3 commands.