

Production Scenarios — SRE · DevOps · Platform

Platform, SRE & DevOps interview guide

Questions	171 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/sre_devops_scenarios_guide.html
Printed	2026-10-21

Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

171 real production scenarios across SRE, DevOps, and Platform Engineering — the "something is broken, walk me through it" questions senior interviews actually use. Each follows a **STAR diagnostic path**: **Symptom** → **Investigation** → **Root cause** → **Fix** → **Prevention**, closing with an interview tip. Filter by domain or use Interview Mode to drill.

S1 **Your main API's error rate jumped from 0.1% to 8% three minutes ago. You're the on-call. What do you do first?**

Incident Triage

🗣 Spoken answer · ~69 s

My first move is to mitigate, not to understand everything. I confirm it's real from more than one signal, then ask the key question: what changed three minutes ago? I check deploys, config and feature flag changes, infrastructure events, and traffic shifts. If it correlates with a deploy, I roll back right away, because that's the fastest, lowest-risk fix. In parallel I scope the impact: is it all endpoints or one, all regions or one, one customer segment, one dependency? The error types, like five hundreds versus timeouts versus a specific downstream, tell me where to dig. I declare an incident if it's customer-impacting, post a first status update, and call in help so I'm not alone. If it isn't a deploy, I look at dependency health, saturation of the database or connection pools, and upstream traffic spikes, and consider mitigations like failing over, shedding load, or disabling a non-critical feature. Once it's stable, I preserve evidence, write the timeline, and run a blameless postmortem.

Likely follow-ups

- The deploy was two hours ago, not three minutes. Does that change your approach?
- How do you decide between rollback and roll-forward?

🚩 Error-rate spike



● SYMPTOM

You get paged: SLO burn rate is 14x on the checkout API. Customers are seeing intermittent 500s. No one has touched anything... that you know of.

● INVESTIGATION

```
# 1. Don't diagnose yet – coordinate + quantify
```

```
ack page; declare incident; open #inc channel; become IC
```

```
# 2. Quantify blast radius
```

```
which SLI? which endpoints? % of users? region-specific?
```

```
# 3. THE highest-yield question:
```

```
what changed in the last ~30 min? (deploy / config / flag / migration)
```

```
kubectl rollout history deploy/checkout # recent deploy?
```

```
git log --since='30 min ago' / check CD pipeline + flag changes
```

~80% of incidents are change-induced, so the deploy/config/flag timeline is the first place to look. In parallel a second responder checks dependency health (DB, downstream APIs) and the four golden signals on the dashboard.

● ROOT CAUSE

A canary deploy 4 minutes ago shipped a config that pointed 10% of traffic at a DB replica that was lagging — those requests timed out and returned 500s. The error rate (8%) roughly matched the canary slice plus retries.

● FIX / MITIGATION

```
# Mitigate BEFORE root-causing – restore service first
```

```
kubectl rollout undo deploy/checkout # roll back the canary
```

```
# confirm the SLI recovers within a minute or two
```

● PREVENTION

Once the SLI recovers, de-escalate the page (recovered ≠ resolved) and investigate the replica-lag bug calmly. First stakeholder update goes out with a 'next update by HH:MM' commitment.

Interview tip: The thing they're testing is your **reflex order**: coordinate (IC + channel) → quantify → 'what changed?' → mitigate → communicate. Say out loud 'I restore service before finding root cause — I'd roll back the most recent change and confirm the SLI recovers first.' That single sentence is the SRE signal.

Official docs

- [Google SRE Workbook: On-Call](#)
- [Google SRE Book: Effective Troubleshooting](#)

S2 A service is 'up' (health checks green, dashboards green) but users insist it's broken. How do you reconcile that?

Incident Triage

Spoken answer · ~83 s

When users say it's broken and the dashboards are green, I trust the user signal over the dashboard and look for the gap in what I'm measuring. First I reproduce the issue as a real user would, running the real checkout journey end to end, not hitting the health endpoint. Health checks usually test that the process is alive, not that the checkout works. Then I compare what the health check covers with what the user path touches: a dependency such as the payment provider, a database path, a specific region, a feature flag, or a client version. I look at user-facing signals: error rate by endpoint, support ticket patterns, and client-side or real user monitoring data, which may show failures that server metrics don't, for example requests failing at the CDN, the load balancer, or in the browser. I slice by region, device, and user cohort, since a partial failure averages out in global dashboards. Once found, I mitigate, then fix the monitoring so this class of failure is detected: add an SLI measured from the user's perspective, synthetic checks of the real journey, and deeper health checks. Green dashboards that miss real breakage are themselves a finding.

Likely follow-ups

- What would you add to catch this class of problem next time?
- Why can a health check pass while checkout fails?

SYMPTOM

Support is flooded with 'checkout fails' reports. Every internal dashboard is green. The health-check endpoint returns 200. Leadership wants to know if it's real.

INVESTIGATION

```
# Trust the user signal over the green dashboard – find the gap
```

```
# Reproduce as a real user would (not the health check):
```

```
curl -v https://app/checkout # full real journey, not /healthz
```

```
# Compare what the health check tests vs what users actually do
```

```
# Check per-dimension: region, AZ, app version, device, customer tier
```

```
# Look at REAL request success, not synthetic /healthz
```

This is a gray failure: monitoring tests something different from what users do. The health check pings a trivial endpoint that doesn't touch the broken dependency, so it stays green while the real path fails.

ROOT CAUSE

The health check only verified the process was up; it never exercised the payment-gateway call. A cert on the payment gateway expired, so every real checkout failed — but /healthz never called it, so every probe passed.

● FIX / MITIGATION

```
# Shift the broken subset out / fix the cert
```

```
# immediate: rotate/renew the expired cert
```

```
# widen the health check to exercise the real critical path
```

● PREVENTION

Make health checks (and synthetic probes) exercise the actual user journey, not a trivial liveness ping. Add a synthetic 'checkout' transaction that would have caught this with zero traffic. Measure SLIs from the user's vantage point.

Interview tip: The headline insight: **when users and dashboards disagree, believe the users — your monitoring has a blind spot.** Name it as a 'gray failure,' and propose end-to-end/synthetic checks that mirror real journeys. Bonus points for naming cert expiry as a classic silent killer.

Official docs

- [AWS Builders' Library: Implementing health checks](#)
- [Grafana documentation](#)

S3 Latency spiked to 5x normal but CPU, memory, and disk all look fine. Where do you look?

Incident Triage

Spoken answer · ~73 s

If the usual host resources look fine, the time is going somewhere that isn't compute, so I look at waiting. First, dependencies: a slow database query, a slow downstream service, or a cache miss storm. Traces show where time is spent, and the slow query log and connection pool metrics are where I look next. Second, saturation of things that aren't CPU: thread pools, connection pools, file descriptors, queue depth, or lock contention, because requests queueing before they run looks like latency with idle CPU. Third, the network: packet loss, DNS delays, retransmits, or a bad path, and TLS handshakes if connection reuse broke. Fourth, runtime issues like garbage collection pauses, or CPU throttling in containers, where usage looks low but the CFS quota is limiting the pod. I'd also check what changed, whether a deploy, config, or a shift in traffic mix. My method is to use distributed traces or RED metrics to find the slow segment, then narrow down, and compare p50 against p99 to see whether it's everyone or only the tail.

Likely follow-ups

- p50 is fine but p99 is 5x worse. What does that suggest?
- How would you prove CPU throttling in a container?

SYMPTOM

P99 latency went from 200ms to 1s. The host metrics (CPU ~30%, memory fine, disk I/O low) say everything is healthy. The service is clearly slow.

INVESTIGATION

```
# 'Resources fine but slow' = it's WAITING, not working
```

```
# Find what it's blocked ON (off-CPU), not what burns CPU
```

```
ss -ti # per-socket RTT/retransmits to dependencies
```

```
# trace a slow request end-to-end (distributed tracing)
```

```
# which span got slow? a downstream call? a lock? DNS?
```

```
strace -p <pid> -T -e trace=network,futex # where it stalls
```

```
# check connection pool saturation + downstream P99
```

Low resource usage with high latency means the process is blocked waiting — on a downstream dependency, a lock, DNS, or an exhausted connection pool. A flame graph would show little on-CPU; off-CPU analysis shows the wait.

● ROOT CAUSE

A downstream dependency's P99 had degraded, and the service had no per-dependency timeout. Requests piled up waiting; by Little's Law ($L = \lambda \times W$) the rising wait time (W) inflated in-flight concurrency (L) until the connection pool was nearly exhausted — so even fast requests queued for a free connection.

● FIX / MITIGATION

```
# Mitigate: protect the caller from the slow dependency
```

```
# add/lower a timeout on the downstream call
```

```
# enable a circuit breaker so it fails fast instead of piling up
```

```
# scale the pool only as a stopgap (treats symptom, not cause)
```

● PREVENTION

Bound every network call with a timeout, add circuit breakers, and size connection pools with Little's Law in mind. Add tracing so the slow hop is obvious next time instead of requiring strace in prod.

Interview tip: Reframe immediately: '**resources fine but slow means it's waiting, so I do off-CPU analysis, not resource hunting.**' List the usual culprits — downstream latency, lock contention, DNS timeouts, pool exhaustion — and tie pool exhaustion to Little's Law. That off-CPU mindset separates senior debugging from 'check top'.

Official docs

- [Google SRE Book: Effective Troubleshooting](#)

S4 An incident started 20 minutes ago and three engineers are all investigating in parallel, stepping on each other. How do you take control?

Incident Triage

Spoken answer · ~78 s

This is a coordination failure, so I take control before anyone touches anything else. First, establish command: declare a single incident commander, myself or someone I assign, as the one decision-maker. One incident channel becomes the single source of truth. I ask everyone to stop making changes and report what they have done so far and what they plan, because uncoordinated parallel changes, a rollback, a live config edit, and pod restarts at once, make it impossible to know what helped or hurt. Then I assign clear roles: an operations lead, a communications lead, and a scribe recording a timeline. I decide on a hypothesis and the next action, usually the safest mitigation first, like completing the rollback, and assign others to investigate separate workstreams without changing production until the IC agrees. Changes are announced in the channel before they're made. I set a short sync cadence, say every ten minutes, and keep communicating status to stakeholders. When the SLI recovers, I verify, then close the incident. Afterwards, a blameless postmortem covers the coordination gaps, such as missing roles and no clear commander, so it's better next time.

Likely follow-ups

- Why stop all changes before deciding on a plan?
- What roles do you assign immediately?

SYMPTOM

You join a chaotic incident call. One engineer is rolling back, another is editing config live, a third is restarting pods. No one knows what the others are doing. The SLI is still bad.

INVESTIGATION

Establish command structure – this is a coordination failure

1. Declare a single IC (you, or assign one) – one decision-maker

2. One incident channel = single source of truth

3. STOP uncoordinated changes – they can collide/mask each other

4. Assign clear workstreams; a scribe timestamps actions

5. Short sync cadence (every ~10-15 min): each lead reports

The technical problem is secondary right now — the real failure is that simultaneous, uncoordinated changes make it impossible to know what's helping, and one person's rollback can collide with another's hotfix. An IC's core job is preventing conflicting simultaneous actions.

ROOT CAUSE

Two of the parallel changes were actually fighting each other: the live config edit kept reverting the rollback. Neither engineer knew the other was acting, so the SLI never stabilized.

● FIX / MITIGATION

```
# Freeze changes, then act deliberately under the IC
```

```
# pick ONE mitigation (the rollback), let it complete, observe
```

```
# everyone else stops changing prod until the IC coordinates
```

● PREVENTION

Adopt the Incident Command System for anything above a minor: one IC who coordinates but isn't hands-on-keyboard, defined roles (ops/comms/scribe), one channel, explicit workstream ownership. Run game days so the team practices this before a real SEV1.

Interview tip: They want to see you **impose structure under chaos**: a single IC, one source of truth, and stopping uncoordinated changes. The killer line: 'two people fixing the same thing simultaneously is dangerous — the IC's job is to prevent conflicting actions, not to debug.' Note the IC can be junior; command is a skill, not seniority.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

You roll back a bad deploy but the error rate doesn't recover. The system seems stuck. What's happening?

Spoken answer · ~76 s

If rolling back the bad code doesn't restore service, then the trigger is gone but the system is staying down. That points to a metastable failure: a self-sustaining feedback loop. I look for the loop. A retry storm, where failures cause retries that add load that causes more failures. Or a full queue or backlog that must drain, connection pools exhausted, caches that are cold and stampeding the database, or a dependency that's overloaded and can't recover while traffic stays high. Adding replicas doesn't help and can make it worse, because they hit the same overloaded dependency. The recovery is to reduce load rather than add capacity: shed load or rate-limit at the edge, disable retries or open circuit breakers, pause non-critical traffic and batch jobs, and let the bottleneck recover. Then restore traffic gradually, perhaps by percentage, while watching the SLI, and warm caches. Check for leftover state from the deploy too, such as a migration or a poisoned message in a queue. Afterwards, the postmortem addresses the amplifiers: retry budgets with backoff and jitter, load shedding, and capacity for cold-start conditions.

Likely follow-ups

- Why does adding replicas not help in a metastable failure?
- How do you bring traffic back gradually?

SYMPTOM

A deploy caused an outage. You rolled it back immediately — but five minutes later the error rate is still pinned high even though the bad code is gone. Adding replicas isn't helping either.

INVESTIGATION

```
# The trigger is gone but the system stays down -> metastable failure
```

```
# Look for a self-sustaining feedback loop:
```

```
# - retry storm: failures -> retries -> more load -> more failures
```

```
# - full queues / saturated connection pools that won't drain
```

```
# - thundering herd of clients reconnecting in lockstep
```

```
ss -s; check queue depth, retry rates, LB backlog
```

```
# adding capacity FEEDS the overloaded dependency more traffic
```

This is a metastable failure: a brief trigger pushed the system into a state its own retries now sustain. Recovery is counter-intuitive — adding servers or retrying harder pours fuel on the fire because the new capacity just generates more load on the saturated component.

● ROOT CAUSE

Clients (and internal retries) kept hammering the recovering service in lockstep. Each failure spawned retries that re-saturated it the instant it tried to come back, so it never escaped the bad state — classic retry-storm-driven metastability.

● FIX / MITIGATION

```
# Recovery = REDUCE load, not add capacity
```

```
# shed load at the edge (return 429/503 to most traffic)
```

```
# temporarily disable retries / drain the queues
```

```
# let the system reach a healthy steady state, THEN ramp traffic back gradually
```

● PREVENTION

Add retry budgets (cap retries as a fraction of traffic), exponential backoff with jitter, and circuit breakers so a blip can't become a storm. Build load-shedding so the system can drain. These break the synchronization that causes metastability.

Interview tip: The counter-intuitive answer is the whole point: **'during an overload-driven outage, the instinct to add servers deepens it — recovery is about shedding load, not adding supply.'** Name it metastable failure, identify the retry storm, and prescribe load shedding + retry budgets + backoff/jitter. Few candidates know this; it's a strong senior signal.

📖 Official docs

- [AWS Builders' Library: Ensuring rollback safety during deployments](#)

S6 **Disk is filling on a production host and will be full in ~30 minutes. df shows 95%, but du of the obvious directories doesn't add up. What's going on?**

Incident Triage

🗣 Spoken answer · ~78 s

When df says ninety-five percent but du can't find the space, the classic explanation is deleted-but-open files: a process still holds a file that was deleted, so the space isn't freed and du can't see it. I confirm with lsof, listing deleted files held open, using lsof plus L1 or checking for deleted entries. Typically it's a log file that was removed or rotated without the process being told to reopen it. The fix with thirty minutes of runway: truncate the file through the process's file descriptor under proc, or restart or signal the process to release the file, whichever is safe, which frees the space immediately. Other explanations for a df and du mismatch: a mount point hiding data under it, because files existed in the directory before another filesystem was mounted over it; sparse files; or inode exhaustion, which I check with df minus i. If the space is real, I find large files with du across the filesystem, ignoring other mounts, and look at what's growing. Longer term: fix log rotation, add alerts based on a predicted fill time, and fix the application writing excessively.

Likely follow-ups

- How do you free space held by a deleted open file without restarting?
- What else can make df and du disagree?

● SYMPTOM

A 'disk 95% full' alert fires (good — it's predictive, not at 100%). You run du on /var and /home and the numbers are far smaller than what df reports as used. Space is vanishing but you can't see where.

● INVESTIGATION

```
# df > du mismatch = space held by deleted-but-open files
```

```
df -h / # filesystem reports 95% used
```

```
du -xh / | sort -rh | head # adds up to much less -> the tell
```

```
# A process holds a deleted file open; blocks stay allocated
```

```
lsof +L1 # files with link count 0 still open
```

```
lsof | grep deleted # shows the holding PID + fd
```

The classic fingerprint: df (filesystem accounting) far exceeds du (visible files). A process is still holding a file that was rm'd — usually a log file deleted by a bad rotation while the app keeps writing to its open fd. The blocks can't be freed until that fd closes.

● ROOT CAUSE

logrotate removed the app's log file but the app wasn't signaled to reopen, so it kept writing to the now-deleted inode. The file is invisible to du (no name) but its gigabytes are still allocated, visible only via lsof.

● FIX / MITIGATION

```
# Reclaim WITHOUT killing the app
```

```
: > /proc/<pid>/fd/<fd-number> # truncate the still-open deleted file
```

```
# or, if acceptable, signal/restart the app to release the fd
```

● PREVENTION

Fix log rotation to use copytruncate or a post-rotate reload signal so the app reopens its log. Add predictive disk alerts (you already had one — that's why this isn't an outage). Watch the journal, Docker, and old kernels too.

Interview tip: This is a top-tier Linux triage question. Diagnose with the **df-vs-du mismatch** → `lsof +L1`; recover by truncating the proc fd (no restart needed). Mention copytruncate/postrotate reload as the permanent fix. Calling out that the alert fired at 95% (predictive lead time) not 100% shows capacity-planning maturity.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S7 Every request to your service suddenly takes exactly 5 seconds, then succeeds. Nothing is erroring. What's the prime suspect?

Incident Triage

Spoken answer · ~77 s

A flat delay of almost exactly five seconds, with requests succeeding afterwards, is the signature of a DNS timeout. The default resolver timeout is often five seconds, after which it retries or falls back and succeeds. A consistent round number is a strong hint that the delay is a timeout, not load. I check `resolv.conf` for the options, search domains and `ndots`, because a high `ndots` value multiplies lookups. I time DNS directly with `dig` or `getent` from the affected hosts, and check whether the resolver or a particular nameserver in the list is slow or dropping packets. In Kubernetes, the classic cause is the conntrack race on UDP DNS, which drops one of the parallel A and AAAA queries, and the client waits the five seconds; CoreDNS overload or a NetworkPolicy dropping DNS are others. Fixes: node-local DNS cache, reducing `ndots` or using fully qualified names, adding `single-request-reopen`, fixing or removing a bad nameserver, or scaling CoreDNS. Other fixed delays have different fingerprints: one second or three seconds suggest TCP SYN retransmits. I confirm with packet capture, looking at DNS queries and their retries.

Likely follow-ups

- What other fixed delays point to specific timeouts?
- How would you confirm DNS is the cause with a packet capture?

SYMPTOM

Latency didn't spike randomly — it's pinned at almost exactly 5.0 seconds for a large fraction of requests, which then complete successfully. A suspiciously round, consistent number.

INVESTIGATION

```
# A consistent ~5s flat delay screams DNS timeout
```

```
# Default resolver timeout is often 5s before fallback/retry
```

```
cat /etc/resolv.conf      # options timeout / attempts / ndots
```

```
# high 'ndots' -> many failing lookups before the real one
```

```
dig / getent hosts <dependency>  # is resolution slow/failing?
```

```
# trace a request: which span eats 5s? (likely a name lookup)
```

A flat, round delay (5s, 10s) is the signature of timeouts, not organic slowness — most often DNS. A misconfigured resolver, a high `ndots` value causing many failed lookups, or one dead nameserver in the list makes every request wait out the timeout before succeeding via fallback.

ROOT CAUSE

A high `ndots:5` setting (common in some Kubernetes configs) caused each hostname to be tried with several search-domain suffixes first — each suffixed lookup hit a slow/failing resolver and waited ~5s before the real query succeeded.

● FIX / MITIGATION

```
# Mitigate the DNS path
```

```
# lower ndots / add a trailing dot to FQDNs to skip search domains
```

```
# fix or remove the unhealthy nameserver
```

```
# add a node-local DNS cache to absorb lookups
```

● PREVENTION

Use FQDNs (trailing dot) for known hosts, tune `ndots`, run a local DNS cache (e.g. `NodeLocal DNSCache` in K8s), and monitor resolver latency. DNS is a dependency — give it timeouts and observability like any other.

Interview tip: The pattern recognition is the win: '**a consistent flat 5-second delay is almost always a DNS timeout, not organic latency.**' Mention `ndots` and search-domain expansion (very common in Kubernetes), one-dead-nameserver, and the fix (FQDNs, local DNS cache). Recognizing 'round number = timeout' is exactly the instinct interviewers probe.

Official docs

- [Kubernetes: DNS for Services and Pods](#)
- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

S8 **An alert is paging every 20 minutes overnight, but each time the on-call looks, everything is fine and it auto-resolves. How do you handle it?**

Incident Triage

 Spoken answer · ~80 s

I ask two questions: is the alert bad, or is the system flapping? An alert that pages every twenty minutes and auto-resolves, with nothing to do, is noise that burns out the on-call, and silencing it is tempting but loses the signal. On the alert side, a threshold sitting at the normal operating boundary needs fixing: add a for duration so the condition must persist before firing, add hysteresis so it fires at one level and resolves at a lower one, evaluate over a longer window or smooth the metric, or better, convert it to a symptom-based alert, like SLO burn rate, instead of a raw CPU number. If the alert isn't actionable, make it a ticket or delete it. On the system side, the flapping may itself be real: a periodic job, a cron spike, garbage collection, or an autoscaler thrashing every twenty minutes. The twenty-minute regularity is a clue, so I correlate with scheduled jobs. If it reveals a real periodic issue, I fix that. I don't just silence it, but I do immediately reduce the overnight impact, perhaps by routing to a ticket until fixed, and track alert quality afterwards.

Likely follow-ups

- What does the 20-minute regularity suggest?
- When is it right to delete an alert rather than tune it?

 SYMPTOM

The overnight on-call is exhausted — a CPU alert fires, they wake up, check, it's already back to normal, repeat. It's flapping right at the threshold. They want to just silence it.

 INVESTIGATION

```
# Two questions: is the ALERT bad, or is the SYSTEM flapping?
```

```
# Alert-side fixes for a threshold sitting at the normal boundary:
```

```
for: 5m           # require the condition to PERSIST before firing
```

```
hysteresis: fire above X, resolve below Y (Y < X)
```

```
# evaluate over a longer window / use burn rate, not instantaneous
```

```
# Then ask: WHY is CPU brushing the line every 20 min?
```

```
# a cron? a GC cycle? a batch job? a service near its limit?
```

A flapping alert usually means the threshold sits right at normal operating range. The immediate fix is a for: duration and/or hysteresis so a momentary brush doesn't page. But a chronically flapping metric can also signal a system genuinely teetering at its capacity edge — silencing without investigating hides a brewing problem.

● ROOT CAUSE

A periodic batch job spiked CPU just over the alert line every 20 minutes. The alert was instantaneous (no duration), so each brief spike paged. The system wasn't actually unhealthy — but the alert was also masking that the box had little headroom.

● FIX / MITIGATION

```
# Fix the alert AND check the headroom
```

```
# add 'for: 5m' so transient spikes don't page
```

```
# move the batch job's schedule / nice/ionice it
```

```
# verify the host has enough headroom (target ~60-70% util)
```

● PREVENTION

Run a monthly alert review: every page that fired should be urgent, actionable, and need a human — if not, downgrade it to a ticket or delete it. Track % actionable as a team-health metric. A shrinking, more-actionable pager is a maturing practice.

Interview tip: Don't just say 'silence it.' Show the dual lens: **fix the alert (duration/hysteresis/burn-rate) AND ask whether the flap reveals a real capacity problem.** The senior framing: 'a noisy pager is an engineering bug, not the cost of doing business' — and pager fatigue is itself a reliability risk because an alert-numbed responder misses the real incident.

Official docs

- [Google SRE Workbook: On-Call](#)

S9 **A change was deployed successfully, the deploy pipeline went green, but production is now subtly broken. How do you respond?**

Incident Triage

 Spoken answer · ~77 s

If the pipeline is green but production is subtly broken, I treat it as an incident and mitigate first. I roll back to the known-good version, using `kubectl rollout undo` or reverting the release, and confirm that the user-facing SLI recovers. Then I analyse why the checks didn't catch it, because a green deploy only means the rollout completed and pods became ready, not that the behaviour is correct. Typical gaps: tests that don't cover the broken path or run against unrealistic data; health checks that only test liveness; a change that depends on production data, configuration, or a dependency version; feature flags or config differences between environments; and no post-deploy verification or SLO-based canary analysis. I look at what changed, the diff, config, and dependency versions, and compare with staging. Preventive actions: add automated tests for the failing behaviour, add smoke tests and synthetic checks after deployment, adopt canary releases with automatic analysis on error rate and business metrics, not just infrastructure metrics, and make rollback one click. Also instrument the broken feature so I'd see wrong results quickly. The postmortem records the gap.

Likely follow-ups

- What post-deploy verification would have caught this?
- Why isn't a ready pod proof that the feature works?

 SYMPTOM

CI/CD reports success. The rollout completed, all pods are Ready, no alerts fired. But a key feature is returning wrong results for some users — the deploy 'succeeded' yet broke prod.

 INVESTIGATION

```
# 'Deploy succeeded but broke prod' -> the pipeline's checks were insufficient
```

```
# Treat it as an incident; mitigate first
```

```
kubectl rollout undo deploy/<svc> # roll back to the known-good version
```

```
# confirm the wrong-results behavior stops
```

```
# THEN investigate why: the deploy was healthy but the BEHAVIOR was wrong
```

```
# look at what the change actually altered (logic, data shape, flag)
```

A green pipeline only proves the deploy mechanism worked and pods are live — it says nothing about correctness. The gap is missing behavioral/canary validation: there were no SLO-gated canary checks comparing the new version's real outputs against the old.

 ROOT CAUSE

The change altered a calculation that passed all unit tests but produced wrong results against real production data shapes the tests didn't cover. Liveness/readiness probes were green because the process was fine — the logic was wrong, which probes never check.

● FIX / MITIGATION

```
# Roll back, then close the validation gap
```

```
# revert to prior version (instant if blue-green / simple if rolling)
```

```
# add a metrics-gated canary so the next bad change is caught at 1%
```

● PREVENTION

Add progressive delivery: canary to 1% of traffic, watch SLOs/error budget and key business metrics, auto-rollback on regression before most users are affected. Canary catches what staging and unit tests miss — real traffic, real data, real scale.

Interview tip: Two messages land: **'deploy succeeded' only means the mechanism worked, not that behavior is correct**, and the fix is a metrics-gated canary with auto-rollback. Mention that probes check liveness/readiness, never correctness — so a logically-wrong-but-healthy deploy sails through. Roll back first, validate later.

Official docs

- [Google SRE Workbook: Canarying Releases](#)
- [Martin Fowler: CanaryRelease](#)

S10 Half your requests fail and half succeed, seemingly at random, right after a routine deploy. What does the 50/50 split tell you?

Incident Triage

Spoken answer · ~80 s

A clean fifty-fifty split with no pattern by user or endpoint points to version or instance skew. Requests are being load balanced between two kinds of backend, and one kind is bad. First I check for mixed versions: `kubectl get pods` with wide output, looking at image versions, and rollout status to see if the rollout is stuck half done, with old and new pods both serving. If new pods are broken, for example a bad config or an incompatible schema change, half the traffic hitting them fails. Or the setup is two replicas, one of which is unhealthy but still passing its readiness check, so the load balancer sends it half the requests. I compare per-pod error rates to identify the bad instance, check its logs, and compare its environment variables and config with the healthy one. Another possibility is two backends in a load balancer pool, or two availability zones, one broken. Mitigation: remove the bad pod from the Service or roll back the deploy. Then fix the readiness probe so a broken instance doesn't receive traffic, and use canary rollout with automated analysis so a skewed deploy is caught early.

Likely follow-ups

- How do you find the one bad instance quickly?
- Why might a broken pod still pass its readiness probe?

SYMPTOM

Post-deploy, almost exactly half of requests return errors and half are fine, with no pattern by user or endpoint — it looks random. The deploy itself reported success.

INVESTIGATION

```
# A clean ~50/50 split points at a VERSION or INSTANCE skew
```

```
kubectl get pods -l app=svc -o wide # mixed image versions?
```

```
kubectl rollout status deploy/svc # rollout stuck half-done?
```

```
# or: 2 replicas, one healthy one broken -> LB sends 50% to the bad one
```

```
# or: 2 config versions, half the fleet has bad config
```

```
# check: are the failures all hitting one subset of pods?
```

A near-perfect 50/50 (or N/M) split that looks 'random' usually isn't random — it's a fleet partitioned into two states by the load balancer. Half the instances run the bad version/config and round-robin routing sends half the traffic there.

ROOT CAUSE

A rolling deploy got stuck halfway: half the pods updated to a build with a broken env var, half stayed on the old good build. The Service load-balanced evenly across both, so exactly the requests routed to the new pods failed.

● FIX / MITIGATION

```
# Mitigate by removing the bad half
```

```
# roll back the deployment (or scale the bad ReplicaSet to 0)
```

```
kubectl rollout undo deploy/svc
```

```
# or cordon/delete the broken pods so LB only hits healthy ones
```

● PREVENTION

Use readiness probes that actually validate the app can serve (so broken new pods never enter rotation), and metrics-gated canaries that halt a rollout when the new cohort's error rate rises — instead of a rollout blindly proceeding to 50%.

Interview tip: The pattern-recognition point: **a clean ~50/50 'random' failure split is almost never random — it's an instance/version skew with the LB splitting traffic across a good and a bad cohort.** Tie it to a stuck rolling deploy and to readiness probes that should have kept the broken pods out of rotation.

📖 Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S11 You're paged for high memory on a host. By the time you log in, free memory is 'low' but the app is fine. Is this an incident?

Incident Triage

Spoken answer · ~78 s

Probably not an incident. The key is to not read the free column. Linux uses spare memory as page cache, shown in buff slash cache, and that is reclaimable when applications need it. The number that matters is available, not free. If available is healthy and the application is fine, the alert is a false alarm caused by the wrong metric. Real memory pressure shows up as swapping, with vmstat showing swap in and swap out activity, as high major page faults, as the OOM killer entries in dmesg, or as allocation stalls and latency. So I check those. If there's no swapping and no OOM events, I'd conclude it's normal. The action is to fix the alert: use available memory, or better, alert on pressure indicators like swap activity, PSI memory pressure stall information, or OOM kills, not on free memory. I'd also check the trend: if available memory is steadily declining, there might be a leak that will become a problem later, so I'd keep an eye on it. And I'd document this in the runbook so the next on-call doesn't wake up for it.

Likely follow-ups

- Why is 'available' a better metric than 'free'?
- What signals indicate genuine memory pressure?

SYMPTOM

Alert: 'available memory < 10%'. You log in, run `free -h`, and most memory shows as used — very little 'free'. The application is responding normally. Is something wrong?

INVESTIGATION

```
free -h          # don't read the 'free' column!
```

```
# Linux uses spare RAM as page cache (buff/cache) – reclaimable
```

```
# The number that matters is 'available', not 'free'
```

```
# Real pressure shows as SWAPPING, not low free:
```

```
vmstat 1         # si/so columns = swap in/out activity
```

```
cat /proc/pressure/memory # PSI: % time stalled on memory
```

```
dmesg | grep -i oom      # any OOM kills? (past pressure->failure)
```

Low 'free' memory is normal and healthy — Linux fills unused RAM with reclaimable page cache to speed up I/O. The alert was likely written against the wrong metric ('free' instead of 'available'). Real memory pressure manifests as swap activity, page scanning, and major faults, not merely low free.

ROOT CAUSE

The alert thresholded on 'free' memory, which is almost always low on a busy Linux host because of page cache. There was no actual pressure: 'available' was healthy, no swapping, no OOM. It was a false alarm from a badly-defined metric.

● FIX / MITIGATION

```
# No real incident – fix the ALERT
```

```
# rethreshold on 'available' memory and/or swap activity (si/so)
```

```
# alert on PSI memory pressure for a modern, accurate signal
```

● PREVENTION

Alert on the right signal: MemAvailable, swap in/out rate, or PSI (Pressure Stall Information) — not raw free. This is also a symptom-vs-cause lesson: page on user-facing impact, not on a raw resource number that doesn't predict harm.

Interview tip: The classic correction: '**low free RAM is normal — Linux caches in spare memory; watch available and swap activity, not free.**' Conclude it's a false alarm caused by a bad metric, and propose alerting on available/PSI/swap. RSS-vs-VSZ and buff/cache-is-reclaimable are great supporting details.

Official docs

- [Linux kernel docs: Control Group v2](#)

A downstream dependency you call is intermittently slow. Your service is now also slow and starting to drop requests. How do you stop the bleeding?

Spoken answer · ~68 s

This is the start of a cascading failure, so the goal is to stop the slow dependency from dragging us down. The mechanism is that our threads or connections wait on the slow calls, pools fill, queues grow, and now we look slow and begin dropping requests, and callers retry, which makes it worse. Immediate mitigations: apply or tighten timeouts so we stop waiting, enable a circuit breaker so we fail fast instead of piling up, reduce or stop retries, and shed non-critical load. If the dependency is non-essential, I degrade gracefully by serving cached or default data, or disabling the feature that needs it. If it's essential, I scale our side carefully, since more instances can mean more load on a struggling dependency. I contact the dependency's owners in parallel. Longer term: bulkheads so one dependency can't exhaust shared pools, timeout budgets across the call chain, retry budgets with jitter, caching, SLOs on dependencies, and game-day tests of exactly this failure.

Likely follow-ups

- How do you size a timeout budget across a call chain?
- When is scaling out the wrong response?

SYMPTOM

A third-party API you depend on has degraded — its latency is erratic. Your service makes synchronous calls to it, and now your own threads are tied up waiting, your latency is climbing, and you're starting to shed real requests.

INVESTIGATION

```
# A slow dependency is dragging you down via resource exhaustion
```

```
# 'Slow' is often worse than 'down' — it ties up your threads
```

```
# Immediate levers:
```

```
# 1. timeout: bound the call so threads don't hang (e.g. 1-2s)
```

```
# 2. circuit breaker: trip after N failures -> fail fast, stop calling
```

```
# 3. fallback: serve cached/default data for the degraded feature
```

```
# 4. bulkhead: isolate that dependency's thread/conn pool
```

Without a timeout, each slow call holds a thread until it finally returns; rising latency multiplies in-flight concurrency until your pools exhaust and you can't serve anything — the dependency's failure has cascaded into yours. The defenses (timeout, breaker, bulkhead, fallback) all exist to contain that blast

radius.

● ROOT CAUSE

Your client had a very long (or no) timeout on the third-party call and shared one thread pool across all work. The slow dependency consumed the whole pool, so even requests that didn't need that dependency couldn't get a thread.

● FIX / MITIGATION

```
# Contain it now
```

```
# set an aggressive timeout on the dependency call
```

```
# open the circuit breaker to fail fast + serve a fallback
```

```
# (if non-critical) degrade gracefully: drop that feature, keep core flow
```

● PREVENTION

Bulkhead each dependency with its own pool so one can't starve the rest. Add circuit breakers + timeouts + fallbacks as defense-in-depth. Rank dependencies by criticality: optional ones should fail open (degrade), critical ones need redundancy.

Interview tip: Key insight: **'slow is worse than down — an unbounded slow dependency exhausts your threads and cascades.'** Prescribe the layered set (timeout + circuit breaker + bulkhead + fallback) and explain graceful degradation: the checkout must survive even if recommendations are down. Defense-in-depth, not one pattern.

Official docs

- [Google SRE Book: Emergency Response](#)

S13 It's 3am, you're paged for a SEV1, and you can't figure out the cause after 15 minutes. What now?

Incident Triage

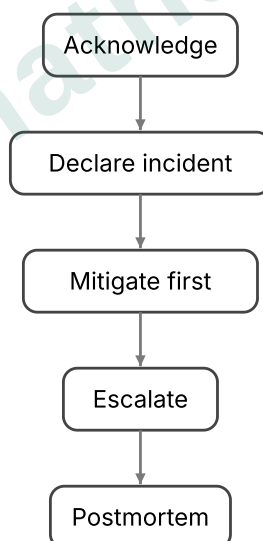
Spoken answer · ~71 s

At fifteen minutes with no cause, the right move is to stop going solo. I escalate: page the service owner, a secondary, or an expert for the suspected area, because fresh eyes and parallel investigation beat heroics. I make sure the incident has a clear commander so someone is coordinating while others dig, and I check that customers and stakeholders have a current status update, since silence causes its own problems. Meanwhile I shift from finding the cause to reducing impact: can we roll back anything recent, fail over, scale up, shed load, or disable the suspect feature, even without proving the cause? Mitigation doesn't require root cause. I also re-examine my assumptions, list what changed in the last day, and ask what would look different if each hypothesis were true. I keep notes so tired people don't repeat checks. Afterwards, the postmortem should ask what made this hard to diagnose, such as missing dashboards or unclear runbooks, and fix that, because debugging at 3 a.m. is a system design problem.

Likely follow-ups

- Who do you escalate to first and why?
- What would you change afterwards so this is easier to diagnose?

SEV1 at 3am



SYMPTOM

A full outage. You're on-call, alone, and 15 minutes in you still don't understand why. The pressure is mounting. Users are down. What's the right move?

INVESTIGATION

```
# Two parallel tracks: ESCALATE and MITIGATE BROADLY
```

```
# Escalating is a strength, not a failure:
```

```
# pull in secondary / specialist (DBA, network) via escalation policy
```

```
# get more hands + the person who knows this system
```

```
# Mitigate broadly even without root cause:
```

```
# 'what changed in the last hour?' -> roll it back blindly if plausible
```

```
# failover to another region/AZ; restart the suspect service
```

```
# shed load if it's overload-shaped
```

After a time-box (you've hit it), the priorities are getting help and restoring service — not continuing to solo-debug while users suffer. Broad mitigations (roll back recent changes, failover, restart) can restore service even before you understand the cause.

● ROOT CAUSE

In this case the on-call escalated, and the secondary immediately recognized a pattern from a prior incident — a specific dependency's connection pool — that the primary hadn't seen. Escalation, not more solo effort, was the fastest path.

● FIX / MITIGATION

```
# Restore first via broad mitigation, diagnose after
```

```
# roll back the most recent change even if unproven
```

```
# failover to a healthy region/replica
```

```
# once recovered: full investigation + postmortem
```

● PREVENTION

Set explicit escalation policies (timeout to secondary, clear path to specialists) and a culture where escalating early is praised. Maintain runbooks and run game days so 3am procedures are muscle memory, not improvisation.

Interview tip: They want to hear that you'll **escalate without ego and mitigate broadly under a time-box**, rather than heroically soldiering on. 'Escalating is the fastest path to the right expertise, not an admission of failure.' Pair with mitigate-before-diagnose: roll back / failover to restore service even without root cause.

Official docs

- [Google SRE Workbook: Incident Response](#)

S14 After an incident is resolved, what do you do to make sure it doesn't happen again — and how do you run the review?

Incident Triage

Spoken answer · ~77 s

Once service is restored, the work isn't done: if I just move on, the same incident recurs next month. I drive a blameless postmortem that produces tracked action items. Blameless means focusing on the system, never punishing individuals, so people share what really happened. The document covers a summary, the impact in users, duration, and error budget consumed, a timestamped timeline, the contributing factors, plural, since incidents rarely have a single cause, what went well, what went poorly, and where we got lucky. The most important part is the action items: each has an owner, a priority, and a ticket in the normal backlog, and they address detection, mitigation, and prevention. I hold the review soon after, while memory is fresh, with the people involved and adjacent teams, and share the write-up widely so others learn. Then I make sure the actions actually happen: track completion, review overdue items, and link repeat incidents to unfinished actions. I use the error budget policy to justify prioritising reliability work. Finally, I verify the fix, with a test or a chaos experiment, so I know it works.

Likely follow-ups

- Why list contributing factors rather than a single root cause?
- How do you make sure action items get completed?

SYMPTOM

Service is restored. The fire is out. Leadership says 'great, move on.' You know that's how the same incident recurs next month. What's the right follow-through?

INVESTIGATION

```
# Drive a BLAMELESS postmortem -> tracked action items
```

```
# Blameless = focus on the system, never punish individuals
```

```
# Document: summary, impact (users/duration/budget burned),
```

```
# timeline, contributing factors (plural!), what went well/poorly,
```

```
# where we got lucky, action items (each w/ owner + priority + ticket)
```

```
# Ask 'how did the system ALLOW this?' not 'who broke it?'
```

The incident isn't closed when service is restored — it's closed when the action items that prevent recurrence are done. Blamelessness isn't about being nice; it's about getting the truth, because engineers who fear punishment hide the details you need to fix the system.

● ROOT CAUSE

The trigger was 'someone ran the wrong command,' but the real finding was systemic: a destructive command had no confirmation, no dry-run, and no guardrail. The fix is changing the system, not the person — who did exactly what the system invited.

● FIX / MITIGATION

Convert learnings into prevention

every action item: named owner + priority + linked ticket

high-priority (SEV1) actions get scheduled, not 'best effort'

track completion rate; tie repeat incidents to skipped actions

● PREVENTION

Aggregate postmortems quarterly to find systemic patterns ('40% of our SEV2s are config pushes with no validation' → build the guardrail for the whole class). Use the error budget policy to force reliability work above features when the budget is burned.

Interview tip: Emphasize **blameless** (truth over blame), **contributing factors** not a single root cause (complex systems fail from combinations — Swiss cheese), and **action items with owners or they die**. The standout line: 'human error is a symptom — the real finding is why a dangerous action had no guardrail.' That's the systems-thinking view interviewers want.

Official docs

- [Google SRE Book: Postmortem Culture](#)
- [Atlassian: How to run a blameless postmortem](#)

🗣 Spoken answer · ~81 s

Flat traffic with CPU suddenly pinned at one hundred percent and no deploy means something is doing more work per request. I profile what's burning CPU. On the host, `top` and `pidstat` show which process; `perf top` shows the hottest functions system-wide; and `perf record` on the process with call graphs, turned into a flame graph, shows exactly where the time goes. I also split user versus system time and check for steal, since high system time points to the kernel, for example network or syscalls, and steal means a noisy neighbour. Common causes: a hot loop or regex gone wrong on certain input, garbage collection thrash from memory pressure, a cache that expired or got cleared so every request recomputes, a dependency slow enough that the app is spinning or retrying, a thundering herd, or data growth making a query or computation heavier. Also check if something changed outside deploys: a config flag, a data change, a certificate, a cron job, or a crypto miner. Mitigation: scale out or restart to buy time, or roll back a flag, while continuing the analysis. Then fix the cause and add profiling and CPU saturation alerts.

Likely follow-ups

- How do you tell user CPU from system CPU problems?
- What would a flame graph show you?

● SYMPTOM

Traffic is flat versus yesterday, yet every app server is at 100% CPU and latency is climbing. Nothing was deployed. The CPU burn appeared over about ten minutes.

● INVESTIGATION

```
# Same traffic + sudden CPU spike = something is doing MORE work per request
```

```
# Profile what's burning CPU (on-CPU analysis):
```

```
perf top                                # live hottest functions system-wide
```

```
perf record -F 99 -p <pid> -g -- sleep 30; perf report
```

```
# turn samples into a flame graph -> widest tower = the hot path
```

```
# also check: GC thrash? a hot loop? regex catastrophic backtracking?
```

```
# any recent DATA change (not code)? a new input pattern?
```

Flat traffic with a CPU spike means each unit of work suddenly costs more — a hot code path, GC pressure, or pathological input. A flame graph localizes it fast; the surprise is it's often triggered by data, not a deploy.

● ROOT CAUSE

A new customer started sending inputs that hit catastrophic regex backtracking in a validation function. Traffic volume was unchanged, but those requests burned exponentially more CPU, saturating every core.

● FIX / MITIGATION

```
# Mitigate the hot path
```

```
# block/throttle the pathological input pattern at the edge
```

```
# hotfix or disable the offending regex/feature flag
```

```
# scale out as a stopgap (buys time, doesn't fix per-request cost)
```

● PREVENTION

Add CPU profiling that's cheap enough for production (eBPF/continuous profiling) so the hot path is one click away. Fuzz/validate inputs and bound regex complexity. Alert on CPU-per-request, not just CPU, to catch efficiency regressions.

Interview tip: The reframe: **same traffic + more CPU = each request got more expensive — profile the hot path, don't just scale.** Mention flame graphs (on-CPU) and that the trigger is frequently *data*, not code (a new input pattern, catastrophic regex). Continuous/eBPF profiling on prod is the senior tooling answer.

📖 Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S16 Two services depend on each other and both are now failing. How do you untangle a suspected cascading failure?

Incident Triage

Spoken answer · ~80 s

For two services depending on each other and both failing, I need to find the origin of the cascade and break the feedback loop. First, timing: which degraded first? Trace timestamps, metrics, and the deploy and config logs show where it began. Second, look for the loop: A waits on B, which waits on A, leading to mutual starvation of threads or connection pools, plus retries amplifying the load. I check resource saturation in each: thread pools, connection pools, and queue depths. To break the loop quickly: stop the amplification and shed load. Disable or limit retries, open circuit breakers, apply rate limiting, and reduce the traffic into the pair. If one service is clearly the origin, for example a bad deploy, roll it back. I might temporarily break the circular dependency by degrading the feature that makes the callback, or restarting in a controlled order after isolating them. Then restore traffic gradually. Longer term: the circular dependency is a design smell, so remove or make it asynchronous; add timeouts shorter than the caller's, bulkheads per dependency, circuit breakers, and test with fault injection. The postmortem should map the dependency graph.

Likely follow-ups

- Why is a circular dependency between services a design smell?
- How do shorter timeouts on callees prevent cascades?

SYMPTOM

Service A calls B; B calls A for some operations. Both are throwing errors and high latency simultaneously. It's unclear which one started it — they're dragging each other down.

INVESTIGATION

Find the ORIGIN of the cascade + break the feedback loop

Look at timing: which degraded FIRST? (trace timestamps, deploy log)

Look for the loop: A waits on B waits on A -> mutual starvation

Check retries: are they amplifying load between the two?

Circuit breaker state, thread pools, queue depth on both

Cascading failures spread through dependencies and retries; a circular dependency makes it worse because each service's failure feeds the other. The job is to find where it started and break the amplifying loop, not to fix both at once.

ROOT CAUSE

B slowed first (a DB issue). A's calls to B timed out and retried aggressively, doubling load on B; meanwhile A's threads filled waiting on B, so A's own B-bound calls failed too. The retries between them sustained the cascade after the original DB blip passed.

● FIX / MITIGATION

```
# Break the loop, then fix the origin
```

```
# trip circuit breakers between A and B -> fail fast, stop amplifying
```

```
# disable/limit retries temporarily; shed load
```

```
# fix the origin (B's DB), then restore traffic gradually
```

● PREVENTION

Avoid circular dependencies in design; where unavoidable, mandate timeouts, circuit breakers, retry budgets, and bulkheads so one service can't drag the other down. Map the dependency graph so the critical path and cycles are known before an incident.

Interview tip: Show you can **find the cascade's origin and break the amplifying loop** rather than firefighting both endpoints. Name retry storms as the amplifier and circuit breakers + retry budgets as the circuit-breaker. Calling out the circular dependency as a design smell is a strong architecture signal.

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)

S17 A bug only reproduces under production load — never in staging. How do you debug it safely?

Incident Triage

🗣 Spoken answer · ~81 s

A bug that only appears under production load usually means hitting a limit or a race that needs concurrency. I can't break production to investigate, so I check safely and live. The usual suspects: file descriptor limits, with too many open files; ephemeral port exhaustion or many connections in TIME_WAIT; connection pool or thread pool exhaustion; lock contention and race conditions; garbage collection pauses; and conntrack table limits. I use read-only observation tools: ulimit and open file counts, ss for connection states, pool metrics, thread dumps, profilers with low overhead, distributed traces for the failing requests, and logs filtered by the failure signature. To find what differs from staging, I compare configuration, data volume, concurrency, and dependencies. Then I try to reproduce it safely: replay production traffic or a sample against a staging environment, load test with realistic concurrency and data volumes, or use a shadow or canary instance receiving a small share of real traffic with extra diagnostics enabled. I can also add temporary targeted instrumentation behind a feature flag. Once I find the limit, I raise it or fix the race, and add a load test that reproduces it to prevent regression.

Likely follow-ups

- What limits commonly appear only under production load?
- How can you safely get more diagnostics from a production instance?

● SYMPTOM

Users hit an intermittent failure in prod. You cannot reproduce it in staging no matter what. It clearly correlates with high load, but you can't just break prod to investigate.

● INVESTIGATION

```
# 'Only under load' = hitting a LIMIT or a race that needs concurrency
```

```
# Usual suspects (check live, under load):
```

```
ulimit -n / LimitNOFILE      # 'too many open files'?
```

```
ss -tan state time-wait | wc -l  # ephemeral port / TIME_WAIT exhaustion
```

```
conntrack -C                  # nf_conntrack table full?
```

```
# connection/thread pool exhaustion; lock contention; race conditions
```

```
# capture continuous metrics so you have data WHEN it happens
```

Load-only bugs are usually a saturated resource or a concurrency race that needs real parallelism to surface. Since you can't reproduce in staging, you instrument prod with continuous sampling and correlate the failure timestamps with whichever resource saturates.

● ROOT CAUSE

Under peak concurrency the service exhausted its database connection pool: requests beyond the pool size queued and timed out. Staging never had enough concurrent load to fill the pool, so it never appeared there.

● FIX / MITIGATION

```
# Mitigate + capture
```

```
# raise the limit/pool as a stopgap; add backpressure
```

```
# add tracing/metrics around pool acquisition to confirm
```

```
# load-test staging to the real peak to reproduce next time
```

● PREVENTION

Make staging load-representative (realistic concurrency, data sizes). Add soak and spike load tests in CI. Check limits before app logic for 'only under load' bugs. Continuous profiling/metrics make intermittent prod issues debuggable without breaking anything.

Interview tip: The mental model: **'only under load' = a limit hit or a concurrency race; check limits before app logic**. Rattle off fd limits, ephemeral ports/TIME_WAIT, conntrack, pool exhaustion. The method that impresses: continuous metric capture + correlate-with-saturation, and load-testing to reproduce — you can't debug intermittent issues without historical data.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S18 **You suspect a recent config change caused an outage, but the change 'looks fine' and passed review. How do you confirm and act?**

Incident Triage

 Spoken answer · ~79 s

A strong correlation between a config push and the start of an outage is strong evidence, and review isn't proof: a reviewed change can still be wrong. So I mitigate first by reverting the config, regardless of how harmless the diff looks. Arguing about whether it can be the config while users are down is the wrong use of time. Reverting is cheap and reversible, and if the outage recovers, that confirms causation; if it doesn't, I've lost little and eliminated a suspect. I use version control to find the exact change, and check whether the config was applied everywhere at once or progressively, and whether the effect lines up with the rollout timing. Then I figure out why the diff looked fine yet broke production: a value that behaves differently at scale, an interaction with another setting, an environment-specific difference, a parse or default-value change, or a change to a dependency that read the config. Afterwards, improve the process: validate configs automatically, roll out progressively with canaries and automatic rollback, keep configs in version control with a clear change log visible on dashboards, and test config changes like code.

Likely follow-ups

- Why revert first instead of proving the config is the cause?
- How should configuration changes be rolled out safely?

 SYMPTOM

An outage started right after a config push. The diff looks harmless and two people approved it. Someone argues 'it can't be the config, it passed review.' Meanwhile users are down.

 INVESTIGATION

```
# Correlation with a change is strong evidence – act, then prove
```

```
# Mitigate first: revert the config regardless of how it 'looks'
```

```
# (a reviewed change can still be wrong – review isn't proof)
```

```
git revert <config-commit> # or roll back the config version
```

```
# confirm the SLI recovers -> that's your proof it WAS the config
```

```
# THEN understand WHY the 'harmless' diff broke things
```

Timing correlation with a change is the single highest-yield signal, and 'it passed review' is not evidence it's correct — reviewers miss things. The fastest path to truth is to revert and watch the SLI: recovery confirms causation faster than arguing over the diff.

 ROOT CAUSE

The config changed a value that was syntactically valid and semantically reasonable in isolation, but interacted with another setting to, e.g., shrink a connection pool below what peak load needed. No reviewer simulated the interaction under load.

● FIX / MITIGATION

```
# Revert, confirm, then prevent
```

```
# revert the config; verify recovery
```

```
# add validation that would have caught the interaction
```

● PREVENTION

Validate config in CI (linting, schema, and ideally a canary that applies it to a small slice first). Treat config changes with the same rigor as code — they cause a large share of incidents. Make config rollbacks as fast and routine as code rollbacks.

Interview tip: Don't get trapped defending the review. **Timing correlation beats 'it looked fine'; revert and let the SLI recovery prove causation.** Stress that config changes are a top incident cause and deserve canary + validation, and that 'passed review' never equals 'correct.' Action over debate when users are down.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S19 Your monitoring/dashboards themselves went dark during an incident — you're flying blind. What now?

Incident Triage

Spoken answer · ~80 s

If the monitoring itself goes dark mid-incident, I'm flying blind, so the first move is to get any independent signal that doesn't rely on the broken pipeline. I probe the service directly: curl the critical path myself, run synthetic checks from outside, and look at the real thing as a user. I use out-of-band sources: cloud provider metrics, load balancer and CDN dashboards, the cloud console, host-level tools like top and logs directly on the machines, and customer reports or support volume as a proxy for impact. Meanwhile I get the monitoring back, treating it as a parallel workstream with its own owner: check whether the pipeline is overloaded, perhaps by the incident's own metric or log volume, or by cardinality, whether its storage is full, or whether it shares a failure domain with what is broken, for instance the same cluster or network. Given the uncertainty, I take more conservative actions and lean on safe mitigations. Afterwards, improve resilience: run monitoring on separate infrastructure from what it monitors, have a dead man's switch alert so silence pages me, keep a minimal out-of-band monitoring path, and practise operating without dashboards in game days.

Likely follow-ups

- Why should monitoring not share a failure domain with the monitored system?
- What is a dead man's switch alert?

SYMPTOM

Mid-incident, your Grafana/metrics pipeline stops updating. The dashboards freeze. You no longer know if things are getting better or worse. The monitoring is down too.

INVESTIGATION

```
# The scariest failure: you can't see. Get ANY independent signal
```

```
# Don't trust the broken pipeline – find out-of-band signals:
```

```
# - direct probes: curl the service health/critical path yourself
```

```
# - cloud provider's own metrics/console (independent of your stack)
```

```
# - logs at the source (tail on the host) if the log pipeline is up
```

```
# - external/synthetic monitor (independent vendor) + status of dependencies
```

```
# - the users themselves (support volume) as a crude SLI
```

When observability fails during an incident you must fall back to independent signals — direct probes, cloud-native metrics, source logs, external synthetics — because the system you'd normally trust is exactly what's broken. Restore a minimal signal before making blind changes.

● ROOT CAUSE

The metrics backend OOM'd from a cardinality explosion: a new deploy added a high-cardinality label (a request ID) to a metric, creating millions of series that crashed Prometheus — ironically taking out monitoring during the incident it was needed for.

● FIX / MITIGATION

```
# Get a signal, then restore monitoring
```

```
# probe the service directly to confirm current state
```

```
# drop/rollback the high-cardinality metric label; restart the TSDB
```

```
# verify dashboards return before relying on them again
```

● PREVENTION

Have an independent, external monitor (different vendor/path) and a dead-man's-switch heartbeat so you detect 'monitoring is down.' Guard metric cardinality in CI. Monitor the monitor — the worst outages are the ones your dashboards can't see.

Interview tip: Two strong points: **have an independent/out-of-band signal because your own monitoring can fail**, and recognize **cardinality explosion** (an unbounded label like `request_id`) as a classic way to OOM Prometheus. Mention a dead-man's-switch to detect monitoring silence. 'Monitor the monitor' is the memorable line.

📖 Official docs

- [Grafana documentation](#)

 Spoken answer · ~74 s

The immediate fix is obvious: rotate the certificate and confirm handshakes recover. But the interviewer wants more because certificate expiry is a predictable, preventable failure class, and this is the second time this year. So I address the whole class. First, inventory: find every certificate and its expiry, not just this one, across load balancers, ingress controllers, internal services, mTLS, code-signing, and third-party certs, using scans, cloud certificate inventories, and cert-manager. Second, automate renewal: use ACME and cert-manager or the cloud's managed certificates, so nothing depends on a human remembering. Third, monitor expiry: export days-to-expiry metrics, such as from the blackbox exporter or x509 exporter, and alert well ahead, say at thirty, fourteen, and seven days, to a channel that gets read, with an owner for each cert. Fourth, test that renewal really works, including that the new certificate is actually loaded, since many services need a reload or restart to pick it up. Fifth, record ownership and a runbook. Finally the postmortem asks why the earlier incident's actions didn't prevent this repeat, and fixes that process gap.

Likely follow-ups

- How do you make sure a renewed cert is actually picked up by the service?
- What expiry alert thresholds would you set?

 SYMPTOM

TLS handshakes are failing across a service — an expired certificate. The immediate fix (renew/rotate) is obvious. But this is the second time this year. The interviewer wants more than 'renew it.'

 INVESTIGATION

```
# Immediate: rotate the cert; confirm handshakes recover
```

```
# But cert expiry is a PREDICTABLE, preventable failure class
```

```
# Find every cert + its expiry, not just this one:
```

```
# inventory certs (services, internal mTLS, load balancers, clients)
```

```
# check for OTHER certs about to expire (don't get paged again next week)
```

```
echo | openssl s_client -connect host:443 2>/dev/null | openssl x509 -noout -dates
```

Renewing fixes this instance, but expired certs are a self-inflicted, fully predictable outage — the real response is eliminating the entire class with automation and lead-time alerting, plus checking the rest of your fleet for the next ticking cert.

 ROOT CAUSE

A manually-issued certificate had no automated renewal and no expiry monitoring, so it silently aged out. Nobody knew until handshakes broke. The 'second time this year' confirms it's a process gap, not bad luck.

● FIX / MITIGATION

```
# Eliminate the class, not just the instance
```

```
# rotate now; scan for other soon-to-expire certs and pre-renew
```

```
# automate issuance + renewal (e.g. cert-manager / ACME / managed TLS)
```

```
# alert on expiry with weeks of lead time
```

● PREVENTION

Automate certificate lifecycle (auto-renewal) so humans are never in the critical path, and alert well before expiry. Treat cert expiry like capacity exhaustion — a predictable date you should never be surprised by. Inventory all certs including internal mTLS.

Interview tip: Going beyond 'renew it' is the test: **cert expiry is a predictable, preventable failure class — automate renewal and alert with lead time, and sweep for other soon-to-expire certs now.** Framing a 'boring' outage as a process/automation gap (not bad luck) shows operational maturity.

Official docs

- [cert-manager documentation](#)

S21 Requests are failing intermittently and the errors are spread across random users with no pattern. How do you find the needle?

Incident Triage

Spoken answer · ~82 s

A low, steady, scattered failure rate of about two percent with no obvious pattern usually has a hidden common dimension, so the job is to slice the failures until one appears. I slice by every dimension I can: host or pod, availability zone, node, a specific replica, backend shard, client version, user agent, region, request size, or time of day. Look for a skew: if two percent of requests fail and exactly one out of fifty pods is bad, that's a two percent failure rate. Other hidden causes: a single bad instance or node, a faulty AZ, one backend shard or database replica with problems, a particular upstream dependency node, an intermittent network issue such as packet loss or a flaky link, a rare race condition, a retry that masks and then exposes problems, or a cache node. Distributed tracing helps by comparing failing and succeeding traces side by side. Logs: group errors by message signature, since varied errors may share a root. I also check for clock or time-related issues and connection reuse. Once I find the dimension, I remove the bad component from rotation, then fix it. The more data I can slice, the faster.

Likely follow-ups

- How do you use traces to compare failing and successful requests?
- What would a 2% error rate with 50 replicas suggest?

SYMPTOM

~2% of requests fail, scattered across users, endpoints, and times with no obvious pattern. It's not a clean cohort split. Logs show varied errors. It feels like noise but it's real and steady.

INVESTIGATION

```
# Low, steady, scattered failures -> find the hidden common dimension
```

```
# Slice by EVERY dimension to find what the failures share:
```

```
# host/pod? AZ? node? a specific replica? a backend shard?
```

```
# Use a trace/correlation ID to follow individual failures
```

```
kubectl get pods -o wide # do failures map to one node/AZ?
```

```
# check: one bad replica, one degraded shard, one flaky NIC/node
```

'Random and scattered' usually means the failures DO share a dimension you haven't sliced by yet — a single bad pod, a degraded AZ, or one unhealthy backend shard that only a fraction of requests happen to hit. High-cardinality slicing (or tracing) reveals the hidden cohort.

ROOT CAUSE

One pod out of twenty had a degraded local disk causing occasional I/O errors. Only the ~5% of requests routed to that pod were affected, scattered across users — so it looked random until failures were grouped by pod name.

● FIX / MITIGATION

```
# Remove the bad subset
```

```
# identify and drain/delete the unhealthy pod/node
```

```
kubectl drain <node> --ignore-daemonsets # or delete the pod
```

```
# verify the scattered failures stop
```

● PREVENTION

Readiness probes and node problem detectors should eject a degraded instance automatically. High-cardinality observability (per-pod, per-AZ labels) and tracing let you group 'random' failures by their true common factor in seconds.

Interview tip: The insight: **'random' failures usually aren't — they share a dimension you haven't grouped by (a bad pod, AZ, or shard)**. Prescribe slicing by every dimension / using trace IDs, then ejecting the bad subset. This is the gray-failure-adjacent skill of finding the hidden cohort behind apparent noise.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S22 After a database failover, the app is up but some users see stale or missing data. What happened and what do you do?

Incident Triage

Spoken answer · ~80 s

If the app is up after a database failover, but some users see stale or missing data, I suspect replication lag. The failover promoted a replica; if replication was asynchronous, the writes that hadn't reached the replica at the moment the primary died were lost, and the new primary doesn't have them. I determine the replication mode and how much lag there was at failover time, by looking at replication metrics and logs. Then assess the damage: which writes were lost, over what window? I can sometimes recover them from the old primary's disk or its write-ahead log or binlog, if it's still accessible, or from application logs, queues, or event streams, and reconcile, and communicate to affected customers. I also check for secondary issues: caches holding stale data that should be invalidated, read replicas lagging behind the new primary, and applications with in-flight transactions. Prevention is a design decision: synchronous or semi-synchronous replication for data that can't be lost, trading latency; or at least a quorum commit; plus idempotent writes, so clients can safely retry. Also define the RPO, test failover regularly including data checks, and avoid split-brain with fencing.

Likely follow-ups

- What does RPO mean for an async replica?
- How do you avoid split-brain after a failover?

SYMPTOM

The primary database failed and automatically promoted a replica. The app recovered — but support reports some recent writes are missing or users see old data. The failover 'worked' but data looks wrong.

INVESTIGATION

```
# Replica promotion + missing recent writes = replication lag / async loss
```

```
# Determine the replication mode and lag at failover time:
```

```
# async replication -> the un-replicated tail is LOST on promotion (nonzero RPO)
```

```
# check replication lag metrics around the failover moment
```

```
# also rule OUT split-brain: did two nodes accept writes?
```

```
# (split-brain = divergent data, far worse than simple lag)
```

With asynchronous replication, a failover promotes a replica that may be slightly behind the dead primary, so the most recent writes that hadn't replicated are gone — a real, expected nonzero RPO. The urgent secondary check is split-brain: if fencing failed and two nodes both took writes, you have divergent data to reconcile.

● ROOT CAUSE

The replica was promoted while a few seconds behind the primary under async replication. Those few seconds of writes (recent orders) were never replicated and were lost when the old primary was fenced — the failover behaved as designed, but the RPO wasn't zero.

● FIX / MITIGATION

```
# Assess and reconcile
```

```
# quantify what was lost (the un-replicated window)
```

```
# recover lost writes from app logs / event stream / idempotent replay if possible
```

```
# confirm only ONE primary now accepts writes (no split-brain)
```

● PREVENTION

Decide RPO deliberately: synchronous replication for zero-loss-critical data (e.g. payments) at a latency cost, async where some loss is tolerable. Ensure failover uses quorum/fencing to prevent split-brain. Test DR failover regularly to measure real RPO/RTO.

Interview tip: Show you understand **async replication means nonzero RPO — a failover can lose the un-replicated tail**, and that **split-brain (two primaries) is the more dangerous possibility to rule out**. Tie the choice to data criticality (sync for money, async otherwise) and to fencing/quorum. Stateful failure modes separate senior candidates.

📖 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [Google SRE Book: Managing Critical State \(consensus\)](#)

Spoken answer · ~83 s

Traffic dropping to almost zero with no errors, and everything else green, is suspicious, not reassuring. The error rate is fine only because there are barely any requests to fail. Usually the failure is upstream: requests aren't reaching me. I work backward from my service toward the user. Is the load balancer healthy and routing to my targets? Are DNS records correct, did a record change or TTL expire pointing somewhere wrong? Is the CDN or WAF blocking or returning cached errors? Did a certificate expire at the edge, or a firewall, security group, or network route change? Is an upstream service or client not calling me, for example because of a failed deploy elsewhere, a feature flag turning off a call path, or an outage in the calling system? Is a client library, mobile app version, or a queue consumer stuck? I also check whether the traffic drop is real: maybe the metrics pipeline broke, so I verify with a direct source such as load balancer logs. Business metrics, like orders per minute, tell me the user impact. Then I mitigate by fixing the upstream break. I also alert on traffic anomalies and low-traffic conditions, not just errors.

Likely follow-ups

- Why isn't a zero error rate reassuring when traffic is zero?
- What alerts would catch a sudden traffic drop?

SYMPTOM

Your request rate fell off a cliff to almost nothing. Error rate is fine (there are barely any requests to error). Latency looks great. Everything is 'green' — but the traffic graph is alarming.

INVESTIGATION

- # Near-zero traffic with no errors is suspicious, not reassuring
- # The failure is UPSTREAM of you -> you're not getting requests
- # Work BACKWARD toward the user:
- # is the load balancer / ingress routing to you? health checks passing?
- # is DNS resolving to the right place? did a DNS/routing change happen?
- # is an upstream gateway/CDN down or misrouting?
- # a dead man's switch should have caught 'expected traffic stopped'

Sudden silence is often worse than errors: it means requests aren't reaching you at all, so your own green metrics are meaningless. The break is upstream — DNS, load balancer, ingress, CDN, or a gateway — and you diagnose by walking backward from your service toward the user.

● ROOT CAUSE

A DNS or ingress change routed traffic away from the service (or a CDN/gateway in front went down). Requests never arrived, so the service saw no load and no errors — looking perfectly healthy while being effectively unreachable.

● FIX / MITIGATION

```
# Restore the path to the service
```

```
# fix the DNS/ingress/LB misroute or recover the upstream gateway
```

```
# verify real user requests are arriving again (traffic returns)
```

● PREVENTION

Alert on absence with a dead-man's-switch: 'expected traffic/heartbeat stopped' should page, since absence-of-errors won't. Measure SLIs as close to the user as possible (at the LB/edge), not just inside your service, so an upstream cutoff is visible.

Interview tip: The counter-intuitive point: **a sudden traffic drop with no errors is a red flag — the failure is upstream and your green metrics are blind to it.** Diagnose by walking backward toward the user (DNS/LB/ingress/CDN). The prevention is a dead-man's-switch alerting on the *absence* of expected traffic. Silence needs monitoring too.

📖 Official docs

- [Prometheus: Alertmanager](#)

 Spoken answer · ~78 s

A vague report like the site is slow needs to become a precise problem statement before I investigate, otherwise I'll thrash. I quantify first by asking: what exactly is slow, which page or journey, and how slow compared with normal? Since when, and does it correlate with a deploy, a config change, or a traffic event? Who is affected and where: all users or some, which regions, devices, networks, or browsers? Is it consistent or intermittent? Then I check data: dashboards for latency percentiles of the relevant endpoints, error rates, traffic, and saturation; real user monitoring to see actual client-side performance; and synthetic checks. I try to reproduce it myself, measuring where time goes: DNS, connection, TLS, server time, and download, using browser developer tools or curl timing. I break latency down by layer: client, CDN, load balancer, application, database, and dependencies, using traces. I narrow by comparing slow and fast requests. Once I have a specific, measurable symptom, I'm in the standard flow: hypothesise, test, mitigate. I report back to the requester with what I found and the current status. Better reporting templates help next time.

Likely follow-ups

- How do you break down where time is spent for a slow page?
- What questions do you ask the reporter first?

 SYMPTOM

A vague report: 'the site feels slow.' No specifics — which page, which users, since when, how slow. You need to convert this into something you can actually investigate without thrashing.

 INVESTIGATION

Turn vagueness into a precise problem statement FIRST

Quantify before theorizing:

WHAT exactly is slow? (which journey/endpoint)

SINCE WHEN? (correlate with a change)

WHO/WHERE? (all users? one region/device/tier?)

HOW slow vs normal? (is it even outside normal variance?)

Then: gather evidence (metrics/traces/logs) BEFORE hypotheses

change ONE variable at a time; verify assumptions, don't trust them

Disciplined debugging starts by stating the problem precisely and gathering evidence before forming theories — otherwise you pattern-match to a past incident and fix the wrong thing. 'Slow' must become 'checkout P95 is 3x normal for EU users since 14:00,' which points somewhere.

● ROOT CAUSE

Once quantified, 'slow' turned out to be 'image-heavy pages slow for one region since a CDN config change an hour ago' — a precise statement that immediately implicated the CDN change, which a scattershot investigation would have missed.

● FIX / MITIGATION

```
# Investigate the now-precise problem
```

```
# with the quantified statement, correlate to the CDN change + roll back
```

```
# confirm latency returns to baseline for that journey/region
```

● PREVENTION

Bisect the problem space (which layer? which region? which change?), change one thing at a time, and verify assumptions ('is it actually the CDN?') rather than trusting them. 'What changed?' is the highest-yield first question once the problem is scoped.

Interview tip: This tests **debugging temperament**: refuse to theorize until the problem is quantified (what/since-when/who/how-much), gather evidence before hypotheses, bisect, and change one variable at a time. 'What changed?' beats deep theory. The discipline — not a specific tool — is what they're screening for here.

Official docs

- [GitHub Actions: Caching dependencies](#)

S25 A previously-fixed incident has recurred — the exact same failure you wrote a postmortem for last month. How do you handle it?

Incident Triage

Spoken answer · ~81 s

First, mitigate: I have the runbook from last time, so I use it to restore service. Then the real question is why it recurred. I pull the prior postmortem and check whether the action items were completed. Almost always the answer is that the actions weren't done, were deprioritised, or were done but didn't actually address the cause. If they weren't done, it's a process failure: reliability work lost to feature work, which tells me the action tracking and prioritisation need fixing. I escalate this, because a repeat incident is a strong argument for prioritising those actions, and the error budget policy provides the lever. If they were done but ineffective, the original analysis was incomplete, perhaps it stopped at the proximate cause, or the fix addressed the symptom, so I redo the root cause analysis more thoroughly, looking at contributing factors and detection. I also verify fixes, with tests or chaos experiments, so I know they work. In the new postmortem, I explicitly document the recurrence and link to the earlier one. I make action items owned, scheduled, and tracked publicly, and report completion rates to leadership. The aim is learning, not blame.

Likely follow-ups

- How do you find out whether last month's actions were completed?
- What does a repeat incident tell you about the original postmortem?

SYMPTOM

You're paged and immediately recognize it: this is the same outage you had a postmortem for four weeks ago. The 'fix' clearly didn't hold, or was never done. Users are down again.

INVESTIGATION

Mitigate now (you have the runbook from last time – use it)

Then the real question: WHY did it recur?

pull the prior postmortem -> were the action items completed?

almost always: the action item was deprioritized/never shipped

OR the fix addressed a symptom, not a contributing factor

trace the recurrence directly back to the skipped/incomplete action

A recurrence is a process failure as much as a technical one — you already knew how to prevent it. Mitigate fast using the existing runbook, then make the recurrence visible: it almost always traces to a postmortem action item that was never completed because reliability work lost to feature work.

ROOT CAUSE

The postmortem's top action item (add a guardrail/validation) was created with no owner and slipped behind feature work, so it was never shipped. The latent condition remained and the same trigger reproduced the same outage.

● FIX / MITIGATION

Fix it for real this time + fix the process

apply the prior fix and actually complete the action item now

escalate: a repeat SEV is a forcing function to prioritize it

● PREVENTION

Every action item needs a named owner, a priority, and a tracked ticket in the normal backlog — an action without an owner is a wish. Use the error budget policy as the lever: a repeat incident justifies promoting reliability work above features. Track action-item completion as a health metric.

Interview tip: The senior move: **a recurrence is a process failure — trace it to the skipped postmortem action item and use the repeat as a forcing function to finally prioritize it.** Stress 'action items without owners die' and that the error budget policy exists precisely to stop reliability work from perpetually losing to features.

Official docs

- [Google SRE Book: Postmortem Culture](#)
- [Google SRE Workbook: Postmortem Culture](#)

 Spoken answer · ~81 s

For CrashLoopBackOff, the pod's container starts, crashes, and Kubernetes restarts it with growing delay. I start with `kubectl describe pod` to read the events and the last state, including the exit code and reason. Then `kubectl logs` with the previous flag, because the current container may have just restarted; I need the logs of the crashed one. If an init container is failing, I check its logs with the `container` flag. The common causes: an application error on startup such as a missing or wrong environment variable, config, or secret; a failing dependency connection such as a database not reachable; a bad command or entrypoint; a permission or file-not-found problem; an OOM kill, shown as exit code 137 and reason `OOMKilled`; or a liveness probe killing a slow-starting app, where a startup probe would help. I fix by correcting the configuration or secret, raising the memory limit if OOM, adjusting probes, or fixing the image. To debug further, I can override the command to `sleep` and `exec in`, or use an ephemeral debug container. Then I redeploy and watch rollout status. If it's production and a new version caused it, I roll back first.

Likely follow-ups

- How do you tell an OOM kill from an application crash?
- How do you debug a container that crashes too quickly to inspect?

 SYMPTOM

A deployment's pods keep restarting — status shows `CrashLoopBackOff` with an ever-growing backoff delay. The app never stays up long enough to serve traffic.

 INVESTIGATION

```
kubectl describe pod <pod>      # events: why is it restarting?
```

```
kubectl logs <pod> --previous  # logs from the CRASHED container, not the new one
```

```
kubectl logs <pod> -c <init>    # init container failing?
```

```
# Common causes:
```

```
# - app exits on startup (bad config/env, missing secret, can't reach dep)
```

```
# - failing liveness probe killing a slow-starting app
```

```
# - OOMKilled (check 'Last State: Terminated, Reason: OOMKilled')
```

```
kubectl get pod <pod> -o jsonpath='{.status.containerStatuses[0].lastState}'
```

CrashLoopBackOff just means 'the container keeps exiting and Kubernetes keeps restarting it with backoff.' The cause is in the previous container's logs and the pod events — most often a startup failure (bad config/missing dependency), an OOMKill, or a liveness probe killing an app that simply boots slowly.

● ROOT CAUSE

The app needed ~40s to warm up, but the liveness probe had a 10s initial delay — so kubelet killed it mid-boot, every time. It wasn't actually broken; the probe never let it finish starting.

● FIX / MITIGATION

```
# Fix matches the cause
```

```
# slow start -> add a startupProbe (gates liveness) or raise initialDelaySeconds
```

```
# OOMKilled -> raise memory limit or fix the leak
```

```
# bad config -> fix env/secret/configmap; redeploy
```

● PREVENTION

Use a startupProbe for slow-booting apps so liveness doesn't kill them during warmup. Keep liveness cheap and local (never check dependencies in liveness — a dep blip then restarts every pod). Right-size memory limits from observed usage.

Interview tip: Walk the method: describe for events + logs --previous for the crash. Then map cause→fix. The classic trap to name: **liveness probe killing a slow-starting app — use a startupProbe, and never put dependency checks in liveness** (that turns a dependency blip into a mass-restart outage). OOMKilled via lastState is the other big one.

Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Kubernetes: Debug Pods](#)

Spoken answer · ~78 s

A pod stuck in Pending means the scheduler can't place it. There are no logs because the container never started, so the answer is in the events: `kubectl describe pod`, and read the events at the bottom, which say exactly why. The common messages: insufficient CPU or memory means no node has room for the pod's requests, so I check requests versus node allocatable and either lower them or add capacity, and verify the cluster autoscaler is working. Node selector, affinity, or taint messages mean the pod's constraints don't match any node, so I check labels, tolerations, and affinity rules. An unbound PersistentVolumeClaim means a storage problem: no StorageClass, provisioner failing, or zone mismatch. Also a quota exceeded in the namespace, a missing PriorityClass, or no nodes available at all. For PVCs with `WaitForFirstConsumer` the pod's scheduling constraints and the volume's zone must align. The describe output normally lists how many nodes failed for each reason, like zero of ten nodes are available with a breakdown. After fixing the cause, the scheduler places the pod automatically. For capacity, I check why the autoscaler didn't add a node.

Likely follow-ups

- What does 'didn't match node selector' tell you?
- How can storage zone affect scheduling?

SYMPTOM

You deploy, but the pod sits in Pending indefinitely. It never lands on a node. No logs (the container never started). The scheduler just won't place it.

INVESTIGATION

```
kubectl describe pod <pod> # the Events at the bottom tell you EXACTLY why
```

```
# Common scheduler messages + meaning:
```

```
# 'Insufficient cpu/memory' -> no node has room for the requests
```

```
# 'node(s) didn't match node selector/affinity' -> selector/affinity too strict
```

```
# 'node(s) had taint that the pod didn't tolerate' -> missing toleration
```

```
# 'no nodes available' / unbound PVC -> storage can't be provisioned
```

```
kubectl get nodes; kubectl top nodes # is the cluster actually full?
```

A Pending pod is a scheduling failure, and `kubectl describe pod` states the reason verbatim in its Events. The usual causes: resource requests no node can satisfy, node selector/affinity/taint mismatches, or an unbound PersistentVolumeClaim.

● ROOT CAUSE

The pod requested more memory than any node had free (requests too high), and the cluster autoscaler couldn't add a node fast enough (or had hit a quota). So the scheduler had nowhere to place it.

● FIX / MITIGATION

```
# Resolve based on the Events reason
```

```
# requests too high -> right-size requests, or add/enable node capacity
```

```
# taint/affinity -> add toleration or relax the selector
```

```
# unbound PVC -> fix the StorageClass / provisioner
```

```
kubectl get events --sort-by=.lastTimestamp
```

● PREVENTION

Right-size requests from real usage (too-high requests waste capacity and block scheduling). Ensure the cluster autoscaler has headroom and quota. Watch for over-strict affinity/taints. Predictive capacity alerts catch 'cluster is filling up' before pods go Pending.

Interview tip: The whole answer hinges on one habit: **kubectl describe pod** tells you the exact scheduling reason in Events — read it. Then map the message (Insufficient cpu/mem, taint, affinity, unbound PVC) to the fix. Mentioning that over-high requests both waste capacity AND cause Pending shows right-sizing judgment.

Official docs

- [Kubernetes: Debug Pods](#)

Spoken answer · ~85 s

To tell a leak from a bad limit from a traffic spike, I look at the shape of memory over time. First, `kubectl describe` confirms OOMKilled as the last state. Then from metrics: a steady climb that never plateaus, even at constant load, indicates a leak; the pods die at about the same age or after a similar number of requests. A flat usage that jumps when load increases and then hits the limit suggests a traffic or workload-driven need, or a limit that is simply too low for the legitimate working set. A sawtooth pattern with garbage collection indicates a heap close to the limit, or a runtime not sized for the container. Correlate with request rate, deploys, and specific endpoints or job types. Also account for non-heap memory: native memory, thread stacks, page cache, and tmpfs volumes, which count towards the limit. For a leak, I take heap dumps or profiles before the kill, and use memory profilers to find the growing objects, then fix the code. For a bad limit, I set requests and limits from measured peak plus headroom. For a spike, I scale out, add backpressure, and bound queues. Alerts on memory approaching the limit give early warning.

Likely follow-ups

- What does a sawtooth memory pattern suggest?
- What non-heap memory counts toward a container's limit?

SYMPTOM

A service's pods restart with 'OOMKilled'. Sometimes it's minutes after start, sometimes hours. You need to distinguish a genuine memory leak from an under-set limit or a load-driven spike.

INVESTIGATION

```
kubectl describe pod <pod> # Last State: Terminated, Reason: OOMKilled
```

```
# Look at the memory-over-time SHAPE (from metrics):
```

```
# steady climb that never plateaus -> LEAK
```

```
# flat then OOM under load -> limit too low for peak working set
```

```
# sawtooth (climbs, GC, climbs) near the limit -> limit too tight for GC headroom
```

```
kubectl top pod <pod>; check container_memory_working_set_bytes
```

```
# exceeding the MEMORY limit = OOMKill (vs CPU limit = throttle)
```

OOMKilled means the container exceeded its memory limit. The memory-over-time shape distinguishes the cause: a never-plateauing climb is a leak; a flat baseline that only OOMs under load means the limit is below the real working set at peak; a sawtooth hugging the limit means insufficient GC headroom.

● ROOT CAUSE

The graph showed a steady, unbounded climb regardless of load — a genuine memory leak. The limit just determined how long until each pod died; raising it only delayed the inevitable OOMKill.

● FIX / MITIGATION

```
# Fix per shape
```

```
# leak -> fix the code (the limit only changes time-to-death)
```

```
# under-set limit -> raise to cover peak working set + headroom
```

```
# GC sawtooth -> give headroom or tune the runtime/heap
```

```
# stopgap: more replicas spreads load but won't cure a leak
```

● PREVENTION

Right-size memory limits from observed working set (P95+headroom) since the penalty is a hard kill, not throttling. Add leak detection (memory trend alerts). Remember CPU over-limit throttles (latency), but memory over-limit kills — treat memory headroom as non-negotiable.

Interview tip: The diagnostic skill is reading the **memory-over-time shape: unbounded climb = leak (raising the limit just delays death), flat-then-spike = limit too low for peak, sawtooth = GC headroom**. And the K8s fact: **memory over-limit OOMKills, CPU over-limit throttles**. That distinction is frequently tested.

📖 Official docs

- [Kubernetes: Debug Pods](#)
- [Kubernetes: Determine the reason for Pod failure](#)
- [Kubernetes: Resource management for Pods & containers](#)

 Spoken answer · ~79 s

When a rolling update is stuck, new pods are Running but never Ready, so the rollout can't progress, and the old pods correctly keep serving. That's the safe behaviour. I confirm with rollout status, then describe the new pod to see why it isn't Ready, and read its logs for startup errors. The readiness probe is failing for the new pods, which has several possible causes: the new version has a bug or a startup failure; the probe configuration is wrong, such as the wrong path, port, or too short a timeout; a dependency the new version needs, like a new config value, secret, or database schema change, isn't available; or the probe works but the app is slow to initialise and needs a startup probe or a longer initial delay. I compare configuration with the old version. If production is at risk, or the cause isn't quickly clear, I roll back with rollout undo; the rollout controls like progressDeadlineSeconds will mark it failed after the timeout. Then I fix the cause, redeploy, and add safeguards: pipelines that wait on rollout status and fail on timeout, and canary analysis.

Likely follow-ups

- Why does a stuck rollout keep the old pods serving?
- What does progressDeadlineSeconds do?

 SYMPTOM

You triggered a rolling update. The rollout is stuck partway: new pods are Running but never Ready, so the rollout won't progress, and (correctly) the old pods keep serving traffic.

 INVESTIGATION

```
kubectl rollout status deploy/<svc> # confirms it's stuck
```

```
kubectl describe pod <new-pod> # why isn't it Ready?
```

```
kubectl logs <new-pod> # app errors on startup?
```

```
# Readiness probe is failing for the NEW version -> good! it's protecting you
```

```
# - new build can't reach a dependency it needs to report ready
```

```
# - readiness endpoint/path changed; probe now wrong
```

```
# - new config/secret missing -> app starts but never becomes ready
```

The rollout is stuck because the new pods' readiness probe never passes, so Kubernetes correctly refuses to send them traffic or proceed (and keeps the old pods up). This is the system protecting you from shipping a broken version — the question is why the new version can't become ready.

● ROOT CAUSE

The new build required a config key that wasn't added to the ConfigMap, so it started but its readiness check (which verified that config) kept failing. `maxUnavailable` was 0, so the rollout safely halted with old pods still serving.

● FIX / MITIGATION

```
# Fix the new version, or roll back
```

```
# add the missing config/secret and let the new pods go Ready
```

```
# or: kubectl rollout undo deploy/<svc> (abandon the bad rollout)
```

```
# tune maxSurge/maxUnavailable for desired rollout safety
```

● PREVENTION

This is readiness probes working as designed — a broken new version never receives traffic. Keep `maxUnavailable` low so a bad rollout halts instead of replacing healthy pods. Add metrics-gated canary analysis to auto-abort on regression, not just on probe failure.

Interview tip: Reframe the 'stuck' rollout as a **feature, not a bug: the readiness probe is correctly refusing to route traffic to a broken new version, so the old pods keep serving**. Diagnose why new pods aren't Ready, then `fix-forward` or `rollout undo`. Note `maxUnavailable: 0` is what makes the failure safe.

📖 Official docs

- [Kubernetes: Deployments](#)
- [Kubernetes: Deployment strategies \(rolling update\)](#)

Spoken answer · ~85 s

For ImagePullBackOff, I start with `kubectl describe pod`, where the events show the exact pull error, and decode it. Not found or manifest unknown means the image name or tag is wrong, a typo, or the tag was never pushed. Unauthorized or denied means missing or expired registry credentials: I check that the `imagePullSecret` exists in the namespace and is referenced by the pod or its service account, and that the credentials are valid; on cloud registries, check the node's or workload's IAM permissions. No such host or timeout means a DNS or network problem: egress rules, NAT, a proxy, or a firewall or network policy blocking the registry. A rate limit error, such as with Docker Hub, calls for authenticated pulls or a mirror or pull-through cache. I verify by trying to pull the same image from a node or a workstation with the same credentials. Other causes are an architecture mismatch, for example an arm64 image on an amd64 node, and an image that's too big or a registry outage. After the fix, the kubelet retries automatically, or I delete the pod. To prevent it, I pin images by digest, test the pull in CI, and use a reliable private registry.

Likely follow-ups

- How would you avoid Docker Hub rate limits in a cluster?
- How does a pull secret get used by a pod?

SYMPTOM

A new pod is stuck with `ErrImagePull / ImagePullBackOff`. The container never starts because the image won't download.

INVESTIGATION

```
kubectl describe pod <pod> # Events show the exact pull error
```

```
# Decode the message:
```

```
# 'not found' / 'manifest unknown' -> wrong image name/tag (typo, missing tag)
```

```
# 'unauthorized' / 'denied' -> missing/expired registry credentials (imagePullSecret)
```

```
# 'no such host' / timeout -> can't reach the registry (network/DNS/firewall)
```

```
# rate limit (e.g. Docker Hub) -> too many pulls, need auth or a mirror
```

```
kubectl get secret <pull-secret> -o yaml # creds present + correct?
```

`ImagePullBackOff` means the kubelet can't fetch the image, and the describe Events name the exact reason: a bad image reference, missing/expired registry credentials, registry unreachability, or rate limiting. You read the message rather than guessing.

● ROOT CAUSE

The image tag referenced a build that was never pushed (a CI step failed silently), so the registry returned 'manifest unknown.' The deployment pointed at an image that didn't exist.

● FIX / MITIGATION

```
# Fix per the pull error
```

```
# wrong tag -> correct the image reference; ensure CI actually pushed it
```

```
# unauthorized -> create/refresh the imagePullSecret + attach to the SA
```

```
# unreachable -> fix network/DNS to the registry; use a pull-through mirror
```

```
# rate limited -> authenticate or use a registry mirror/cache
```

● PREVENTION

Use immutable, verified tags and ensure the image is pushed before the deploy references it (gate deploy on successful push). Manage pull secrets centrally; run a registry mirror/cache to avoid rate limits and registry outages on the critical path.

Interview tip: Again it's read-the-Events: **describe shows the precise pull failure — name (manifest unknown), auth (unauthorized), network (no such host), or rate limit.** The standout systemic point: deploys should be gated on the image actually being pushed, and a pull-through mirror removes the registry from your critical path.

📖 Official docs

- [Kubernetes: Debug Pods](#)
- [Kubernetes: Images \(pull policy, registries\)](#)
- [Kubernetes: Pull an image from a private registry](#)

S31 Service-to-service calls inside the cluster are failing intermittently. How do you debug Kubernetes networking?

Kubernetes

 Spoken answer · ~81 s

Intermittent failures in service-to-service calls inside the cluster call for working up the stack from DNS to endpoints to pods. First, does the Service have the right endpoints? `kubectl get endpoints` for service B shows which pod IPs are included; if some backing pods are unhealthy but still listed, or the endpoint list flaps, calls fail intermittently. I check the pods behind it, wide output, for restarts, readiness, and which node they're on, since a bad node can cause a subset of failures. Then DNS: from a debug pod, test resolution repeatedly, and look for slow or failed lookups, CoreDNS health, and the `ndots` and `conntrack` five-second problem. Then the network: NetworkPolicies, CNI problems, and kube-proxy rules, plus `conntrack` table exhaustion on a node, which causes random drops. I test connectivity directly to pod IPs versus the Service IP to separate kube-proxy from the application. Also consider connection reuse and keepalive issues, where a connection to a terminated pod stays in a pool, producing errors after rollouts. I also look at timeouts and retries configuration. Tracing or a service mesh's metrics show which hop is failing. Fix the cause and add retries with backoff.

Likely follow-ups

- How do you tell a kube-proxy problem from an application problem?
- What causes errors after pods terminate while clients hold pooled connections?

SYMPTOM

Service A calls Service B by its cluster DNS name. Most calls work; some fail with connection errors or timeouts. It's intermittent and doesn't correlate with a deploy.

INVESTIGATION

```
# Work up the stack from DNS to endpoints to pods
```

```
kubectl get endpoints <svc-b> # do the Service's endpoints include healthy pods only?
```

```
kubectl get pods -l app=b -o wide # are some backing pods unhealthy?
```

```
# DNS inside the cluster:
```

```
kubectl exec <pod-a> -- nslookup svc-b # resolves? (CoreDNS healthy?)
```

```
# is one endpoint a bad pod? readiness should remove it but check
```

```
# NetworkPolicy blocking some paths? kubectl get networkpolicy
```

Intermittent in-cluster failures usually mean the Service is load-balancing across a mix of healthy and unhealthy endpoints, or DNS (CoreDNS) is flaky, or a NetworkPolicy is partially blocking. Checking which pods back the Service (endpoints) and whether they're all healthy is the fastest split.

● ROOT CAUSE

One of Service B's pods was unhealthy but its readiness probe was missing/misconfigured, so it stayed in the Service's endpoint list. kube-proxy load-balanced ~1/N of calls to the bad pod, producing intermittent failures.

● FIX / MITIGATION

```
# Remove the bad endpoint
```

```
# add/fix readiness probe so unhealthy pods leave the endpoint list
```

```
# delete/drain the bad pod; confirm endpoints now list only healthy pods
```

```
# (if DNS) scale/tune CoreDNS + add NodeLocal DNSCache
```

● PREVENTION

Correct readiness probes keep unhealthy pods out of Service endpoints automatically. Run NodeLocal DNSCache and right-size CoreDNS for DNS reliability. Audit NetworkPolicies. Understand that a K8s Service is just kube-proxy load-balancing across its endpoints.

Interview tip: Show the layered K8s-network method: **endpoints → pod health → DNS (CoreDNS) → NetworkPolicy**. The most common cause is a **missing readiness probe leaving an unhealthy pod in the Service's endpoint list**, so kube-proxy sends a fraction of traffic to it. Tie it back to the 'random failures share a dimension' pattern.

📖 Official docs

- [Kubernetes: Debug Services](#)

S32 Your cluster autoscaler isn't adding nodes even though pods are Pending. Why might that be?

Kubernetes

Spoken answer · ~86 s

When pods are Pending for insufficient CPU but no nodes appear, I look at why the cluster autoscaler isn't scaling up. I read its logs in kube-system, because it logs its decisions and the reasons. Common causes: the node group is already at its maximum size; the cloud quota or capacity is exhausted, so instance creation fails; the pending pod wouldn't fit even on a new empty node because its requests are larger than any instance type in the group; the pod's constraints, such as node selectors, affinity, taints, or zone requirements for a volume, don't match any node group the autoscaler can scale; the pod has a PVC bound to a zone with no node group; or there are autoscaler annotations or settings like a safe-to-evict issue or a scale-up backoff after earlier failures. Also check that the autoscaler has the right IAM permissions and that its node groups are discovered by tags. Another possibility is that the pods aren't Pending for resources the autoscaler can fix at all, such as a missing PVC or an unsatisfiable anti-affinity. The fix depends: raise max size or quota, add a node group or instance type, adjust the pod's constraints, or correct permissions. Karpenter changes some of these dynamics.

Likely follow-ups

- Why would a Pending pod not trigger a scale-up?
- Where does the cluster autoscaler record its reasoning?

SYMPTOM

Pods are Pending for 'Insufficient cpu', which should trigger the cluster autoscaler to add a node. But no new nodes appear. The cluster just stays full with Pending pods.

INVESTIGATION

```
# Why might the autoscaler NOT scale up?
```

```
kubectl -n kube-system logs <cluster-autoscaler-pod> # it logs its decisions
```

```
# Common reasons:
```

```
# - node group at its MAX size (can't grow further)
```

```
# - cloud quota / capacity limit hit (no instances available)
```

```
# - pod can't fit on ANY node type the autoscaler can add (too-big requests)
```

```
# - pod has affinity/taint constraints no scalable node group satisfies
```

```
# - autoscaler not managing that node group / misconfigured
```

The cluster autoscaler only adds nodes if a Pending pod would actually fit on a node it's allowed to create. It won't scale if the node group is at max, cloud quota is exhausted, the pod's requests exceed any addable node, or affinity/taints exclude the scalable groups. Its logs state the reason.

● ROOT CAUSE

The node group had hit its configured maximum size, so the autoscaler had nothing to do — it logged that scale-up was capped. The cluster genuinely couldn't grow without raising the max or quota.

● FIX / MITIGATION

```
# Resolve the scaling block
```

```
# raise the node group max (and cloud quota) if capacity is needed
```

```
# right-size pod requests so they fit an addable node type
```

```
# relax affinity/taints, or add a node group that satisfies them
```

● PREVENTION

Monitor autoscaler scale-up failures and node-group headroom. Forecast capacity so you raise limits/quota before hitting them. For sharp spikes, pre-provision headroom — autoscaling has provisioning lag and won't save you from an instant surge.

Interview tip: Demonstrate you know the autoscaler's preconditions: **it only adds a node if the Pending pod would fit on a node it's allowed to create** — blocked by node-group max, cloud quota, too-big requests, or affinity/taints. Read its logs. The capacity-planning kicker: autoscaling lags, so pre-provision for known spikes.

Official docs

- [Kubernetes: Debug Pods](#)
- [Kubernetes Autoscaler: Cluster Autoscaler](#)

S33 A node goes NotReady and its pods aren't being rescheduled fast enough. What's happening and what do you do?

Kubernetes

 Spoken answer · ~78 s

When a node goes NotReady, pods aren't rescheduled immediately by design. The node controller waits: after the kubelet stops posting heartbeats, the node is marked NotReady after about forty seconds, and then the pods on it have a toleration for the not-ready and unreachable taints for three hundred seconds by default. Only after that are the pods evicted and replacements created elsewhere. During that time the pods are still listed as Running even though they're unreachable, though Service endpoints stop routing to them earlier because their readiness fails. So capacity is degraded for around five minutes, deliberately, to avoid flapping on brief blips. First I find out why the node is NotReady with `kubectl describe node: memory, disk or PID pressure, kubelet down, a network issue, or the instance itself dead`. To speed recovery, I can lower `tolerationSeconds` for latency-critical workloads, or manually cordon and drain the node or delete the node object if it's definitely gone, which triggers rescheduling. Stateful workloads with `ReadWriteOnce` volumes add delay while the volume detaches. Longer term: enough spare capacity and replicas spread across zones, and health-based auto-repair for nodes.

Likely follow-ups

- Why does Kubernetes wait five minutes before evicting pods?
- How would you speed up recovery for a critical service?

SYMPTOM

A node shows NotReady. The pods on it are still listed as Running for several minutes, and replacement pods are slow to come up elsewhere — so capacity is degraded longer than you'd expect.

INVESTIGATION

```
kubectl get nodes          # node NotReady since when?
```

```
kubectl describe node <node>  # conditions: MemoryPressure? DiskPressure? kubelet down?
```

```
# Why pods aren't rescheduled instantly:
```

```
# - the node's pods are marked for eviction only after a toleration timeout
```

```
# (default ~5 min for node.kubernetes.io/unreachable)
```

```
# - StatefulSet pods won't be force-deleted (could violate at-most-one)
```

```
kubectl get pods -o wide --field-selector spec.nodeName=<node>
```

A NotReady node doesn't instantly relocate its pods: Kubernetes waits out the unreachable toleration (≈5 min by default) before evicting, to avoid thrashing on a transient blip. For StatefulSets it's even more conservative (it won't risk two instances). So there's an inherent recovery delay you must account for.

● ROOT CAUSE

The node's kubelet stopped responding (a hardware/network issue). Pods stayed 'Running' in the API until the 5-minute unreachable toleration elapsed, then they were rescheduled — the delay was the eviction timeout, not a bug.

● FIX / MITIGATION

```
# Recover capacity
```

```
# cordon + drain the node if it's coming back; delete it if it's dead
```

```
kubectl drain <node> --ignore-daemonsets --delete-emptydir-data
```

```
# for faster failover: tune toleration seconds (carefully) + keep headroom
```

```
# ensure enough replicas/headroom to absorb a node loss (N+1)
```

● PREVENTION

Run N+1 capacity so losing a node doesn't degrade service, with PodDisruptionBudgets and anti-affinity so replicas aren't all on one node. Understand the unreachable-toleration delay. Use node problem detection to catch failing nodes early.

Interview tip: Explain the **built-in ~5-minute unreachable toleration that delays rescheduling** (intentional, to avoid thrashing on transient blips) and that StatefulSets are even more cautious. The reliability lesson: **run N+1 with anti-affinity + PDBs so a node loss is absorbed**, since relocation isn't instant.

Official docs

- [Kubernetes: Node-pressure eviction](#)
- [Kubernetes: Taints and tolerations](#)

S34 Your app needs persistent storage in Kubernetes but pods keep losing data on restart. What's misconfigured?

Kubernetes

 Spoken answer · ~79 s

If a stateful app loses data on every pod restart, storage isn't actually durable. I check what the pod mounts: `kubectl describe pod` and `get pvc`. The most common mistake is using `emptyDir`, which is ephemeral and wiped when the pod is deleted or rescheduled, instead of a `PersistentVolumeClaim`. Or the app writes to the container's own filesystem, not the mounted path, perhaps because the `mountPath` doesn't match where the app stores data. Other problems: the PVC exists but the reclaim policy is `Delete` and the claim was deleted, destroying the volume; a `Deployment` used instead of a `StatefulSet`, so each replacement pod gets a fresh volume or a shared one that doesn't suit the app; a `StorageClass` with the wrong type; or the volume being re-provisioned on each deploy because the PVC name changes with Helm templating. The fix: use a `StatefulSet` with `volumeClaimTemplates`, so each pod keeps its own PVC across restarts; verify the `mountPath` is the data directory; set the reclaim policy to `Retain` for important data; and take regular snapshots and backups. I validate by writing data, deleting the pod, and confirming the data returns.

Likely follow-ups

- Why is a `StatefulSet` better than a `Deployment` for a database?
- What happens to a PV when its claim is deleted with reclaim policy `Delete`?

SYMPTOM

A stateful app (e.g. a database) loses its data whenever its pod restarts or reschedules. Each new pod comes up empty. Storage clearly isn't persisting across pod lifecycles.

INVESTIGATION

```
# Data vanishing on restart -> not actually using durable storage
```

```
kubectl get pvc; kubectl describe pod <pod> # is a PVC mounted, or emptyDir?
```

```
# Common misconfigurations:
```

```
# - using emptyDir (ephemeral, dies with the pod) instead of a PVC
```

```
# - a Deployment (shared/rescheduled) instead of a StatefulSet for stateful apps
```

```
# - PVC bound but pod reschedules to another zone where the volume can't attach
```

```
# - writing to the container filesystem (lost on restart) not the mount
```

Data loss across restarts means the pod isn't backed by a `PersistentVolume` — typically it's using `emptyDir` (ephemeral by definition) or writing to the container's own filesystem. Stateful workloads also need a `StatefulSet` (stable identity + per-replica PVC), not a `Deployment`.

● ROOT CAUSE

The app used an `emptyDir` volume, which exists only for the pod's lifetime and is wiped when the pod is deleted/rescheduled. It needed a `PersistentVolumeClaim` backed by durable storage, managed by a `StatefulSet`.

● FIX / MITIGATION

```
# Use durable, identity-stable storage
```

```
# switch to a StatefulSet with volumeClaimTemplates (per-replica PVC)
```

```
# back the PVC with a real StorageClass (network/block storage)
```

```
# ensure the app writes to the mounted volume path, not container fs
```

● PREVENTION

Use `StatefulSets` + `PVCs` for stateful apps; reserve `emptyDir` for scratch. Mind zone affinity so a volume and its pod stay co-located. For databases, prefer a managed service or an operator that handles replication/backups — running stateful systems on K8s is hard.

Interview tip: The crisp diagnosis: **data lost on restart = ephemeral storage (`emptyDir` / container fs), not a `PersistentVolume`**. Stateful apps need a **`StatefulSet` with `volumeClaimTemplates`** for stable identity + durable per-replica `PVCs`. Noting that running databases on K8s is genuinely hard (prefer managed/operator) shows real-world judgment.

📖 Official docs

- [Kubernetes: Persistent Volumes](#)
- [Kubernetes: StatefulSets](#)

Spoken answer · ~81 s

CPU throttling happens against the limit within each scheduling period, a hundred milliseconds in the Linux CFS, not against the average. A bursty application that needs a lot of CPU for a few milliseconds within a period can exhaust its quota early in the period and then be paused for the rest of it, even though its average usage over a second or a minute looks modest. That pause shows up as latency spikes, especially at P99. The commonly viewed metric, such as `kubectl top`, shows averages, which hide this. Multi-threaded applications make it worse: many threads can burn through the quota in a few milliseconds. To detect it, look at the CFS throttling metrics: the number of throttled periods and throttled seconds, for example `container_cpu_cfs_throttled_periods_total` in Prometheus. If throttling is high, fix it by raising the CPU limit, or removing it and relying on requests, which many teams do for latency-sensitive services; reduce the number of threads or tune the runtime, for example setting `GOMAXPROCS` or JVM CPU settings to respect the container; or spread the load. Also make sure requests are accurate so scheduling is fair. Newer kernels have improved CFS bandwidth behaviour.

Likely follow-ups

- Which metric shows CPU throttling?
- Why might you remove CPU limits for latency-sensitive services?

SYMPTOM

A latency-sensitive service has P99 spikes. But CPU usage on the pods looks modest — well under the limit on average. Yet you suspect throttling. How can a 'low CPU' pod be throttled?

INVESTIGATION

```
# CFS throttling happens against the LIMIT per 100ms quota window, not the average
```

```
# A bursty app can be throttled while AVERAGE CPU looks low:
```

```
kubectl top pod # shows average – misleading for bursty workloads
```

```
# check the real signal:
```

```
container_cpu_cfs_throttled_periods_total / cfs_periods_total
```

```
# if throttled_periods is high -> you ARE being throttled despite low avg
```

```
# a CPU limit of e.g. 500m caps each 100ms window at 50ms of CPU
```

Kubernetes CPU limits are enforced by CFS quota per ~100ms window. A bursty service that needs a quick spike of CPU gets throttled the moment it exceeds its per-window quota — even if its average usage is far below the limit. So average CPU looks fine while latency suffers from throttling.

● ROOT CAUSE

The pod had a low CPU limit (e.g. 500m). During request bursts it briefly needed more than its 50ms-per-100ms quota and got throttled, adding latency — while the per-minute average stayed low and innocent-looking.

● FIX / MITIGATION

```
# Relieve the throttling
```

```
# raise or remove the CPU LIMIT (requests still guarantee scheduling)
```

```
# many teams set CPU requests but NO CPU limit to avoid throttling
```

```
# confirm cfs_throttled_periods drops + P99 improves
```

● PREVENTION

Set CPU requests (for scheduling/guarantees) but consider omitting CPU limits to avoid CFS throttling on bursty latency-sensitive workloads, while keeping memory limits (memory over-limit kills, CPU over-limit only throttles). Monitor throttled_periods, not just average CPU.

Interview tip: The non-obvious insight: **CPU limits throttle per 100ms CFS window, so a bursty pod is throttled even with low average CPU.** Watch `cfs_throttled_periods`, not `top`. The common remediation — **CPU requests but no CPU limit, keep memory limits** — plus 'CPU throttles, memory kills' is exactly the senior nuance interviewers reward.

📖 Official docs

- [Kubernetes: Resource management for Pods & containers](#)
- [Linux kernel docs: CFS scheduler](#)

 Spoken answer · ~83 s

When a new version misbehaves, I compare rollback options by speed and safety. The fastest is `kubectl rollout undo` for the Deployment, reverting to the previous ReplicaSet, or to a specific revision. It's quick, but the gotchas: it diverges from Git, so a GitOps controller with self-heal will re-apply the bad version, unless I revert in Git or pause sync; rollout history must be retained; and it only reverts the pod template, not other resources or ConfigMaps changed in the release. For blue-green deployments, flipping the Service selector or load balancer back to the old environment is instant, but needs the old environment still running. With canary or progressive delivery, aborting the rollout shifts traffic back, and tools can do it automatically on SLO regression. With Helm, `helm rollback` reverts to a previous release revision, including config. In GitOps, `git revert` and `sync` is the cleanest and auditable approach, but a little slower. Gotchas that apply to all: database migrations and data changes don't roll back with the code, so schema changes must be backward compatible; feature flags may be faster than redeploying; and I verify recovery via the SLI after rolling back, not just that the command succeeded.

Likely follow-ups

- Why does a GitOps controller undo `kubectl rollout undo`?
- What can't a rollback undo?

 SYMPTOM

A new version is misbehaving in production. You need to revert quickly. The interviewer wants to know your rollback options in Kubernetes and the trade-offs/gotchas of each.

 INVESTIGATION

```
# Options, fastest + safest first
```

```
kubectl rollout undo deploy/<svc> # revert to previous ReplicaSet
```

```
kubectl rollout undo deploy/<svc> --to-revision=N # specific revision
```

```
# blue-green: flip the Service selector back to the old version (instant)
```

```
# canary: just stop the rollout / scale canary to 0
```

```
# GitOps: git revert the manifest -> controller reconciles back
```

```
# GOTCHA: schema/data migrations may NOT be backward-compatible!
```

Kubernetes makes the compute rollback easy (`rollout undo` / flip selector / `git revert`), and the fastest is whatever requires the least change — flipping a blue-green selector is near-instant. The dangerous gotcha is state: if the new version ran a non-backward-compatible DB migration, rolling back the code

can break against the migrated schema.

● ROOT CAUSE

The new version had applied a schema migration that dropped a column the old version still reads. Rolling back the pods alone caused the old code to fail against the new schema — the rollback wasn't safe because state had changed irreversibly.

● FIX / MITIGATION

```
# Roll back safely considering state
```

```
# code-only change -> rollout undo / selector flip is safe + instant
```

```
# with migrations -> only the expand/contract pattern makes rollback safe
```

```
# if a destructive migration ran -> may need forward-fix, not rollback
```

● PREVENTION

Use expand/contract migrations so every intermediate state runs under both old and new code, keeping rollback always safe. Pair canary + auto-rollback for fast reverts. Treat data/schema as the hard part of rollback — code rollback is the easy 20%.

Interview tip: Show the menu (rollout undo, selector flip, git revert, stop canary) AND the critical caveat: **code rolls back easily, but a non-backward-compatible migration makes rollback unsafe.** The senior fix is **expand/contract migrations** so every state is runnable under both versions. State, not compute, is what breaks rollbacks.

Official docs

- [AWS Builders' Library: Ensuring rollback safety during deployments](#)

S37 A DaemonSet pod is failing on just one node out of fifty. How do you approach a node-specific failure?

Kubernetes

 Spoken answer · ~82 s

When a DaemonSet pod fails on exactly one node out of fifty, with the identical spec everywhere, the node differs, not the pod. I start with the pod on the bad node: describe it and read its logs, including the previous container's. Then I describe the node and compare it with a healthy one: conditions such as disk or memory pressure, kubelet and kernel versions, container runtime, labels, taints, and allocatable resources. Typical node-specific causes: a full disk or inode exhaustion, so the pod can't write; a missing or different kernel module or sysctl setting, which matters for networking and monitoring agents; a hostPath that doesn't exist or has wrong permissions on that node; a different OS image or kernel version from a partial upgrade; a stale or corrupted container image cache; port conflicts with something else on that host; or resource exhaustion by other workloads. I also check SELinux or AppArmor differences and cloud-level problems like a bad instance. Once I find the difference, I fix the node or, if it's quicker, cordon and drain it and replace it, since nodes should be disposable. Then I add node-level checks, like node problem detector, to catch drift.

Likely follow-ups

- How do you systematically compare a bad node with a healthy one?
- Why is replacing the node often better than repairing it?

SYMPTOM

A DaemonSet (one pod per node) runs fine on 49 nodes but crashes or fails on exactly one. The pod spec is identical everywhere, so why does one node behave differently?

INVESTIGATION

```
# Identical spec, one bad node -> the NODE differs, not the pod
```

```
kubectl describe pod <pod-on-bad-node>; kubectl logs <pod> --previous
```

```
kubectl describe node <bad-node> # conditions, kernel, kubelet version, taints
```

```
# Compare the bad node to a good one:
```

```
# - different kernel/OS/driver version? disk full? clock skew?
```

```
# - hardware fault (bad disk/NIC)? resource pressure unique to it?
```

```
# - leftover state from a prior workload? a missing host mount/file?
```

When one node out of many fails an identical workload, the variable is the node, not the manifest. You diagnose by diffing the bad node against a healthy one — kernel/driver versions, disk/clock state, hardware health, or leftover host state.

● ROOT CAUSE

The bad node had a full root disk (from a previous workload's leftover files), so the DaemonSet pod couldn't write its host-path data and crashed. The other 49 nodes had free space; only this node's local state differed.

● FIX / MITIGATION

```
# Repair or replace the node
```

```
# fix the node-specific issue (clear disk / fix driver / replace hardware)
```

```
# or cordon + drain + delete the node and let it be reprovisioned
```

```
kubectl cordon <node>; kubectl drain <node> --ignore-daemonsets
```

● PREVENTION

Use node problem detection and immutable, reprovisioned nodes (cattle, not pets) so a sick node is replaced rather than nursed. Monitor per-node disk/health. In immutable infra, the answer to a weird node is often 'delete and let it rebuild from a golden image.'

Interview tip: The principle: **identical workload failing on one node means the node is the variable — diff the bad node against a healthy one** (kernel, disk, hardware, leftover state). The cloud-native disposition: **treat nodes as cattle — cordon/drain/replace rather than nurse a snowflake node.**

Official docs

- [Kubernetes: DaemonSet](#)

S38 Your pods can't reach an external service (e.g. a third-party API) though the rest of the internet works. What do you check?

Kubernetes

 Spoken answer · ~85 s

If pods can reach most of the internet but not one specific external API, I narrow down whether it's DNS, egress routing, policy, or the remote side. DNS resolves, so I test from the pod: `curl verbose` to see whether it's refused or timing out, which differ. Timeout suggests packets are dropped: a NetworkPolicy egress rule, a cloud firewall or security group, or a NAT gateway or proxy that blocks that destination or port. Refused suggests the remote side or a firewall actively rejecting. I check whether the destination uses a different port or protocol than the working traffic. A frequent cause: the partner API allow-lists source IPs, and pods egress through NAT addresses that aren't on their list, or the egress IP changed, or traffic takes a different path with a different source IP. Also TLS problems, such as certificate verification failing, a missing CA bundle in the image, or an MTU issue where large packets are dropped, so small requests work and large ones hang. I compare with a test from a node and from outside the cluster to see whether it's specific to the pod network. Then fix by adjusting the policy or firewall, fixing egress IPs, or contacting the partner.

Likely follow-ups

- How can an MTU problem look like an intermittent connectivity failure?
- Why might a partner API reject only traffic from the cluster?

SYMPTOM

Pods can reach most external endpoints, but calls to one external API consistently fail (timeouts or refused). DNS resolves. Other external traffic from the same pods is fine.

INVESTIGATION

```
# Narrow: is it DNS, egress routing, policy, or the remote side?
```

```
kubectl exec <pod> -- nslookup api.partner.com # resolves correctly?
```

```
kubectl exec <pod> -- curl -v https://api.partner.com # refused vs timeout?
```

```
# refused = reached it, nothing/blocked at port; timeout = dropped in transit
```

```
kubectl get networkpolicy # an egress NetworkPolicy blocking this dest?
```

```
# egress path: NAT gateway IP -> is it on the partner's allowlist?
```

```
# is the partner rate-limiting/blocking your egress IP?
```

Since only one destination fails while general internet works, the issue is specific to that path: an egress NetworkPolicy, the partner allowlisting your NAT/egress IP, or the partner blocking/rate-limiting you. The refused-vs-timeout distinction tells you whether you're reaching them at all.

● ROOT CAUSE

The partner API allowlists client IPs, and the cluster's egress NAT gateway IP had changed (or a new node pool used a different NAT), so the partner silently dropped the traffic — timeouts, while every non-allowlisted destination worked fine.

● FIX / MITIGATION

```
# Restore the egress path
```

```
# pin a stable egress IP (NAT gateway) and get it allowlisted by the partner
```

```
# or fix the egress NetworkPolicy to permit the destination
```

```
# confirm with curl from the pod that calls now succeed
```

● PREVENTION

Use a stable, known egress IP (dedicated NAT) for partners that allowlist, and document/monitor it. Audit egress NetworkPolicies. Treat third-party dependencies with circuit breakers + timeouts so their blocking doesn't cascade into your service.

Interview tip: Key reasoning: **only one external destination failing points to a path-specific cause — egress policy, NAT/egress-IP allowlisting, or remote-side blocking** — and **refused vs timeout localizes whether you even reach them**. The egress-IP-allowlist gotcha (IP changed → partner drops you) is a very common real-world cause.

📖 Official docs

- [RFC 3022: Traditional NAT](#)
- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

S39 A Helm/manifest deploy succeeded but the new config didn't take effect in the running pods. Why?

Kubernetes

 Spoken answer · ~77 s

If the ConfigMap shows new values but pods still use the old config, the pods haven't restarted. Environment variables from a ConfigMap are read once when the container starts, so changing the ConfigMap changes nothing for running pods until they restart. Mounted files do get updated eventually by the kubelet, but not for subPath mounts, and the application must re-read the file anyway, which many don't. I confirm by checking that the pod started before the change. The fix is to trigger a rollout: `kubectl rollout restart`, or better, make config changes automatically cause a rolling update by hashing the config into a pod template annotation, which Helm does with a checksum of the ConfigMap, or using a tool like Reloader that watches ConfigMaps and restarts workloads. Another robust approach is immutable ConfigMaps with versioned names, so each config change creates a new object and updates the Deployment reference, producing a normal rolling update that can be rolled back. After the rollout, I verify the pods have the new values. The pipeline should include this step, and also wait on rollout status to confirm success.

Likely follow-ups

- Why don't environment variables update when a ConfigMap changes?
- How does a checksum annotation trigger a rollout?

SYMPTOM

You updated a ConfigMap (or Helm values) and applied it. The deploy reports success, the ConfigMap shows the new values — but the running pods are still using the OLD config.

INVESTIGATION

```
# ConfigMap changed but pods didn't restart -> they still hold old config

kubectl get configmap <cm> -o yaml    # confirms new values ARE applied

kubectl describe pod <pod>             # but pod started BEFORE the change

# Why old config persists:

# - env-var/volume ConfigMaps are read at pod START; updating the CM

#   does NOT restart pods or re-inject env vars

# - no rollout was triggered because the Deployment spec didn't change

# - (mounted CMs update the file eventually, but the app must re-read it)
```

Changing a ConfigMap doesn't restart pods. Env-var-injected config is read once at container start, so existing pods keep the old values until they're recreated. Since the Deployment's pod template didn't change, no rollout was triggered — the new config sits applied-but-unused.

● ROOT CAUSE

The config was consumed as environment variables. Updating the ConfigMap changed the stored object but didn't recreate any pods, so every running pod still had the env vars captured at its original start time.

● FIX / MITIGATION

```
# Trigger a rollout so pods pick up the new config
```

```
kubectl rollout restart deploy/<svc> # recreate pods -> re-read config
```

```
# better: make config changes auto-trigger a rollout
```

```
# (checksum/config annotation on the pod template, or a reloader)
```

● PREVENTION

Add a config checksum annotation to the pod template (Helm pattern) so a ConfigMap change rolls the Deployment automatically, or run a config-reloader. Treat config as part of the deploy that must trigger a rollout — applied-but-not-restarted is a silent gotcha.

Interview tip: The gotcha: **updating a ConfigMap does not restart pods — env-var config is read once at start, so a rollout must be triggered.** Fix now with `rollout restart`; fix permanently with a **config checksum annotation** (or a reloader) so config changes auto-roll the Deployment. Knowing mounted vs env-var CM behavior is the depth signal.

Official docs

- [Helm documentation](#)

 Spoken answer · ~78 s

One workload starving others on shared nodes is a missing isolation and governance problem. I first diagnose the contention: which pods are consuming the node's CPU or I/O, using `kubectl top` across namespaces and node-level tools, and whether requests and limits are set; BestEffort or poorly sized pods are the usual culprits. Then I fix the governance. Set accurate requests and limits so the scheduler and kernel cgroups enforce fair sharing; use `LimitRange` and `ResourceQuota` per namespace so a tenant can't consume unbounded resources. QoS classes protect critical pods, so give important services `Guaranteed` or well-sized `Burstable` status, and batch jobs lower priority with `PriorityClasses` so they're evicted or preempted first. For stronger isolation, put noisy workloads on dedicated node pools using taints and tolerations, or separate the batch workload entirely, with its own autoscaling and spot capacity. Shared resources that cgroups don't control well, like disk I/O, network, and memory bandwidth, may need I/O limits or separate nodes. I also add monitoring and alerts on node saturation and per-tenant usage, and showback so teams own their consumption. Pod anti-affinity can keep noisy and sensitive workloads apart.

Likely follow-ups

- Which resources are hard to isolate with cgroups?
- How do `PriorityClasses` help protect critical services?

 SYMPTOM

On a multi-tenant cluster, one team's batch job periodically spikes and starves CPU/I/O on shared nodes, hurting latency for unrelated services co-located there.

 INVESTIGATION

```
# One workload starving others = missing isolation / resource governance
```

```
# Diagnose the contention:
```

```
kubectl top pods --all-namespaces # who's consuming the node's resources?
```

```
# check: are requests/limits set? are QoS classes correct?
```

```
# Guilty workload likely has no/low requests -> overcommits the node
```

```
# QoS: Guaranteed (req==lim) > Burstable > BestEffort (evicted first)
```

A noisy neighbor on shared nodes is a resource-isolation failure: without proper requests/limits the offending pod overcommits the node and starves co-located pods. Kubernetes QoS classes and cgroup limits exist precisely to contain this — the batch job needs governance.

● ROOT CAUSE

The batch job ran with no resource requests (BestEffort QoS) and burst hard, consuming CPU and I/O that the scheduler never accounted for. Latency-sensitive services on the same nodes were starved during its spikes.

● FIX / MITIGATION

```
# Impose isolation
```

```
# set proper requests/limits on the batch job (cap its blast radius)
```

```
# raise critical services to Guaranteed QoS (req==limit)
```

```
# isolate by design: dedicated node pool/taints for batch vs latency-sensitive
```

```
# use PriorityClasses + anti-affinity to separate workloads
```

● PREVENTION

Enforce ResourceQuotas and LimitRanges per namespace so no tenant can overcommit. Separate batch and latency-sensitive workloads onto different node pools (bulkheading). Use QoS/PriorityClasses deliberately. Isolation by design beats firefighting contention.

Interview tip: Frame it as **resource isolation: the noisy job lacks requests/limits and overcommits the node — fix with requests/limits, QoS classes, quotas, and bulkheading onto separate node pools.** This is the cgroups/bulkhead pattern applied to multi-tenancy: contain the blast radius so one tenant can't starve the rest.

📖 Official docs

- [Kubernetes: Pod Quality of Service classes](#)
- [Kubernetes: Jobs](#)

S41 A secret was accidentally committed in a container image / manifest. What's your incident response?

Kubernetes

Spoken answer · ~83 s

If a secret was committed into a container image or manifest, I treat it as a security incident: the secret is compromised, assume it's leaked, because anyone who could pull the image or read the repository could have extracted it, and removing it from the latest version doesn't remove it from history or old layers. Step one: rotate the secret immediately, generating a new credential and revoking the old, which is the only reliable remediation. Step two: assess exposure: who or what could access the image or repository, and for how long; check access logs and the provider's audit logs for use of the credential by unknown parties, and look for any suspicious activity. Step three: clean up: remove the secret from the code, rebuild the image without it, and delete the old image tags and layers from the registry; remove it from Git history if appropriate, though rotation matters more. Step four: prevent recurrence: use a secrets manager and inject secrets at runtime, add secret scanning in pre-commit hooks and CI for code and images, use short-lived credentials and workload identity, and add .dockerignore and multi-stage builds. Document it in a postmortem. Notify security and, if required, affected parties.

Likely follow-ups

- Why isn't deleting the commit or image enough?
- What would you scan for in CI to prevent this?

SYMPTOM

During review you discover an API key (or DB password) was baked into a container image and pushed to the registry — and referenced in a manifest in Git. It may have been there for days.

INVESTIGATION

```
# Treat as a security incident: the secret is COMPROMISED, assume leaked
```

```
# 1. ROTATE the secret immediately (it must be considered exposed)
```

```
# 2. Assess exposure: who/what could have pulled the image or seen the repo?
```

```
# 3. Check for misuse (access logs for that credential)
```

```
# DON'T just delete the commit/image -> it persists in history/layers/caches
```

```
git log -p / image layers retain it; rotation is the only real fix
```

A leaked secret can't be un-leaked by deleting the file — it survives in Git history, image layers, registry caches, and anyone's local copies. The only effective response is to rotate (invalidate) the credential immediately and treat it as compromised, then investigate exposure and misuse.

ROOT CAUSE

The secret was embedded as an image layer and referenced in a committed manifest. Removing the commit wouldn't help — the credential lived in pushed image layers and Git history that others could already have, so it had to be rotated.

● FIX / MITIGATION

```
# Rotate, then prevent recurrence
```

```
# rotate/revoke the credential NOW; confirm no active misuse
```

```
# remove from history/layers as cleanup (but rotation is the real fix)
```

```
# move secrets to a secrets manager injected at runtime, never baked in
```

● PREVENTION

Never bake secrets into images or commit them — inject at runtime from a secrets manager (Vault/cloud secrets) via the workload's identity. Add pre-commit/CI scanning (gitleaks) to block secret commits. Use least-privilege, short-lived credentials so a leak's blast radius and lifetime are small.

Interview tip: The critical instinct: **a committed secret is compromised — rotate immediately; deleting the commit/image does NOT help because it persists in history, layers, and caches.** Then move secrets to runtime injection + add CI secret scanning. Treating it as a security incident (rotate-first) rather than a cleanup task is the maturity signal.

Official docs

- [Google SRE Book: Managing Incidents](#)
- [Google SRE Workbook: Incident Response](#)

Spoken answer · ~80 s

To find out why pods are being evicted, I start with the events: `kubectl get events` sorted by time, filtered for `evict`, and describe the evicted pod, which states the reason. Then I describe the node to see conditions like memory pressure, disk pressure, or PID pressure. The main causes are node resource pressure: the kubelet evicts pods when the node runs low on memory, ephemeral disk, or inodes, using QoS classes so `BestEffort` and then over-requesting `Burstable` pods go first. Another cause is exceeding ephemeral storage limits, where a pod writing too many logs or temp files is evicted. Priority-based preemption by higher-priority pods is a different cause, as are node drains and taints with `NoExecute`. To stop it: set accurate requests so the scheduler places pods realistically and the pods aren't classified `BestEffort`; set ephemeral-storage requests and limits, and fix log and temp file growth; reserve resources for the system with `kube-reserved` and `system-reserved`; add capacity or autoscaling; clean up image garbage on nodes with image GC thresholds; and use `PodDisruptionBudgets` for voluntary evictions. Clean up the leftover evicted pod objects, and alert on node pressure before eviction starts.

Likely follow-ups

- How does the kubelet decide which pod to evict first?
- What is the difference between eviction and preemption?

SYMPTOM

Pods are getting Evicted and rescheduled, disrupting service. It's not a deploy. The evictions happen on certain nodes under certain conditions.

INVESTIGATION

```
kubectl get events --sort-by=.lastTimestamp | grep -i evict
```

```
kubectl describe pod <evicted-pod> # eviction reason
```

```
kubectl describe node <node> # MemoryPressure / DiskPressure / PIDPressure?
```

```
# Eviction causes:
```

```
# - node resource pressure -> kubelet evicts (BestEffort/Burstable first)
```

```
# - node disk/inode pressure (image/log/emptyDir bloat)
```

```
# - pod exceeding ephemeral-storage limit
```

```
# - preemption by higher-PriorityClass pods
```

Evictions come from node-level resource pressure (memory/disk/PID) where the kubelet reclaims by evicting low-QoS pods first, or from preemption by higher-priority pods. The node conditions and pod events name which pressure triggered it. QoS class determines who gets evicted.

● ROOT CAUSE

Nodes hit DiskPressure because container logs and emptyDir volumes filled the node's disk, so the kubelet evicted BestEffort pods (which had no resource requests) to reclaim space — repeatedly, as the disk refilled.

● FIX / MITIGATION

```
# Relieve pressure + protect important pods
```

```
# clear the disk pressure (log rotation, prune images, size emptyDir)
```

```
# set requests so important pods are Guaranteed/Burstable, not BestEffort
```

```
# set ephemeral-storage requests/limits; add node disk headroom/alerts
```

● PREVENTION

Set resource requests so critical pods have a protective QoS class (Guaranteed pods are evicted last).
Manage node disk (log rotation, image GC, ephemeral-storage limits) and alert on node pressure predictively.
PriorityClasses protect critical workloads from preemption.

Interview tip: Explain that **evictions stem from node pressure (memory/disk/PID) or preemption, and QoS class decides who's evicted first — BestEffort pods go first**. The fix combines relieving the pressure (often disk: logs/images/emptyDir) with setting requests/PriorityClasses so important pods are protected. Predictive node-pressure alerting prevents the surprise.

Official docs

- [Kubernetes: Node-pressure eviction](#)

S43 You need zero-downtime deploys but users occasionally get errors during rollouts. What's causing the brief errors?

Kubernetes

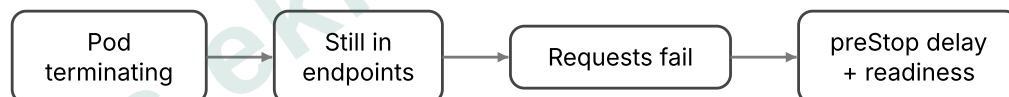
🗣 Spoken answer · ~78 s

Brief errors during otherwise zero-downtime rollouts usually mean pods are terminated before they've stopped receiving traffic. There's a race: when a pod is deleted, it gets SIGTERM and is removed from the Service endpoints at about the same time, but the endpoint removal takes a moment to propagate to kube-proxy, ingress controllers, and load balancers. So for a few seconds traffic can still be sent to a pod that is already shutting down, producing connection resets or 502s. And if the application exits immediately on SIGTERM, in-flight requests are cut. The fixes: add a preStop hook with a short sleep, for example five to fifteen seconds, so the pod keeps serving while endpoints propagate; handle SIGTERM gracefully by stopping new connections, finishing in-flight requests, and then exiting, within `terminationGracePeriodSeconds`; make sure readiness probes are accurate so new pods only get traffic when ready; use `maxUnavailable` zero with a surge so capacity never drops; and for load balancers, configure connection draining or deregistration delay that exceeds the shutdown time. Keepalive settings matter too: clients holding connections to terminated pods will see errors. Test with load during a rollout.

Likely follow-ups

- Why does a preStop sleep help?
- What does `terminationGracePeriodSeconds` control?

🚩 Deploy 502 window



● SYMPTOM

Rollouts are supposed to be zero-downtime, but during each deploy a small number of requests fail for a few seconds — connection resets or 502s — then it's clean again. Almost zero-downtime, but not quite.

● INVESTIGATION

```
# Brief errors during rollout = pods terminated before they stop receiving traffic
```

```
# The race: pod gets SIGTERM + is removed from endpoints, but in-flight
```

```
# requests / the LB's connection list aren't drained gracefully
```

```
# Checklist of the usual gaps:
```

```
# - no preStop hook / no grace period -> in-flight requests cut off
```

```
# - app doesn't handle SIGTERM (finish in-flight + stop accepting)
```

```
# - readiness not flipped to NotReady before shutdown (still gets new traffic)
```

```
# - LB/endpoint propagation lag races the pod shutdown
```

True zero-downtime needs graceful connection draining on shutdown. The brief errors come from a race: a terminating pod is killed (or stops listening) while it's still receiving traffic because endpoint removal and in-flight requests weren't drained. SIGTERM handling + a preStop delay close the gap.

● ROOT CAUSE

On termination, pods received SIGTERM and exited immediately, but it took a moment for them to be removed from Service endpoints and for the LB to stop sending connections — so requests in that window hit a pod that had already stopped accepting them, yielding 502s/resets.

● FIX / MITIGATION

```
# Make shutdown graceful
```

```
# add a preStop hook sleep (a few seconds) so endpoint removal propagates
```

```
# handle SIGTERM: stop accepting new, finish in-flight, then exit
```

```
# set terminationGracePeriodSeconds generously; flip readiness to false first
```

```
# ensure LB connection draining is enabled
```

● PREVENTION

Implement graceful shutdown: SIGTERM → stop accepting new requests → drain in-flight → exit, with a preStop delay covering endpoint/LB propagation and an adequate grace period. This mirrors how the platform expects shutdown to work (SIGTERM → grace → SIGKILL).

Interview tip: The subtlety: **'almost zero-downtime' errors are a termination race — pods stop serving before they're drained from endpoints/LB.** The fix is graceful shutdown: **preStop delay + SIGTERM handling (finish in-flight) + adequate grace period.** Tying it to the SIGTERM→grace→SIGKILL model shows you understand orchestrated shutdown deeply.

Official docs

- [Kubernetes: Deployments](#)

Spoken answer · ~81 s

The ALB's CloudWatch metrics split the blame for me. The metric for ELB 5XX count means the load balancer itself generated the error: there was no healthy target, the connection to the target timed out, or the surge queue was full. That points to target health, capacity, or timeouts. The metric for target 5XX count means the targets returned the 5xx, so the problem is in my application. I also look at the specific codes: 502 means a bad response from the target, such as the target closing the connection unexpectedly, perhaps due to keepalive mismatch; 503 means no healthy targets; 504 means the target timed out against the load balancer's idle timeout. Then I check target group health, how many targets are healthy and why they fail health checks, the target response time, and the request count per target. I look at access logs to see which targets and paths produce errors. If errors are from targets, I move to the application: logs, recent deploys, resource saturation, and dependencies. If they're from the ALB, I check scaling, health check settings, security groups, idle timeout versus application keepalive, and deployment events. Alarm on both separately.

Likely follow-ups

- What does a 502 versus a 504 from an ALB tell you?
- Why should the application keepalive exceed the ALB idle timeout?

SYMPTOM

Users get 5xx from your application load balancer. You need to localize fast: is the ALB itself failing, or are the backend targets returning errors?

INVESTIGATION

```
# ALB CloudWatch metrics split the blame for you:
```

```
# HTTPCode_ELB_5XX_Count -> the ALB generated the 5xx (no healthy target,
```

```
# timeout to target, or surge queue full) = infra/target-health side
```

```
# HTTPCode_Target_5XX_Count -> your APP returned the 5xx = application side
```

```
# Also check: TargetResponseTime, healthy host count, RejectedConnectionCount
```

```
# 502 = bad/closed response from target; 503 = no healthy targets;
```

```
# 504 = target timed out (slower than ALB idle timeout)
```

The ALB's CloudWatch metrics directly separate ELB-generated 5xx (no healthy targets, target timeout, surge queue full) from Target-generated 5xx (your app erroring). The specific code also tells the story: 503 = no healthy targets, 502 = bad target response, 504 = target too slow.

● ROOT CAUSE

HTTPCode_ELB_5XX with a healthy-host count of zero showed the ALB had no healthy targets — the app's health check was failing (it hit a dependency that was down), so the ALB returned 503 even though the app processes were 'running.'

● FIX / MITIGATION

```
# Fix per the split
```

```
# ELB 503/no healthy targets -> fix target health checks / unhealthy targets
```

```
# Target 5xx -> it's your app: roll back / debug the application
```

```
# 504 -> target slower than ALB idle timeout: fix latency or raise timeout
```

● PREVENTION

Make ALB target health checks reflect real readiness (not a trivial ping) but keep them resilient to brief dependency blips. Alert separately on ELB-5xx vs Target-5xx. Understand the code meanings so the metric immediately points to infra vs app.

Interview tip: The core skill: **use HTTPCode_ELB_5XX vs HTTPCode_Target_5XX to split load-balancer faults from application faults**, and read the code (503 = no healthy targets, 502 = bad response, 504 = target timeout). This instant infra-vs-app triage is exactly what AWS production interviews probe.

📖 Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Cloudflare Learning: Load balancing](#)
- [Elastic Load Balancing](#)

S45 An EC2 instance suddenly can't reach the internet (or a dependency) though it worked yesterday. How do you debug AWS networking?

Cloud / AWS

 Spoken answer · ~80 s

When an EC2 instance suddenly loses connectivity, I walk the AWS network path layer by layer and ask what changed since yesterday. First the security group: does it allow the outbound traffic? Security groups are stateful, so return traffic is automatic. Second the network ACL on the subnet: it's stateless, so it must allow both directions, including ephemeral return ports, and a deny rule added recently is a classic culprit. Third the route table: is there a default route to the NAT gateway or internet gateway, or did a route get deleted or become a blackhole? Fourth the NAT gateway itself: is it healthy, in the right subnet, with an elastic IP, and not hitting port allocation limits? Then, for private endpoints, the VPC endpoint policy and DNS settings. I use VPC Reachability Analyzer to test the path, VPC Flow Logs to see accept or reject records, and CloudTrail to find who changed a security group, route, or ACL. On the instance I check DNS resolution and the OS firewall. The usual answer is a recent infrastructure change, so I revert or fix it and codify the rule in Terraform to prevent drift.

Likely follow-ups

- Why do NACLs need ephemeral port rules but security groups don't?
- How do VPC Flow Logs help you here?

SYMPTOM

An EC2 instance that's been working fine can no longer reach an external dependency. No code changed on the box. The connectivity just broke.

INVESTIGATION

```
# Walk the AWS network path layer by layer
```

```
# 1. Security Group: does it allow the OUTBOUND traffic? (SGs are stateful)
```

```
# 2. NACL: subnet-level, STATELESS -> must allow BOTH directions incl. ephemeral ports
```

```
# 3. Route table: is there a route to 0.0.0.0/0 via IGW (public) or NAT (private)?
```

```
# 4. NAT gateway / IGW healthy? (private subnet needs NAT for egress)
```

```
# 5. DNS (route 53 resolver) resolving? 6. is the dependency itself up?
```

```
ip route get <dst>; curl -v <dep> # refused vs timeout on the box
```

AWS connectivity is a stack of independent controls — Security Group (stateful, instance-level), NACL (stateless, subnet-level), route table, and NAT/IGW. You check each in order. A change to any one (often a NACL or route edit) silently breaks egress.

● ROOT CAUSE

A NACL was edited to add a deny rule, and because NACLs are stateless, the change blocked the ephemeral return ports for outbound connections — so new connections timed out. The instance and SG were fine; the subnet NACL broke it.

● FIX / MITIGATION

```
# Restore the path at the broken layer
```

```
# fix the NACL to allow return traffic (ephemeral ports) – NACLs are stateless!
```

```
# verify SG egress, route to NAT/IGW, NAT health
```

```
# confirm connectivity returns
```

● PREVENTION

Prefer Security Groups (stateful, simpler) for most controls and use NACLs sparingly — their statelessness (must allow return traffic explicitly) is a classic footgun. Manage network config as code (IaC) with review so a manual NACL/route edit can't silently break egress.

Interview tip: Demonstrate the layered AWS path: **SG (stateful) → NACL (stateless!) → route table → NAT/IGW → DNS → dependency**. The footgun to name: **NACLs are stateless, so a rule change can block ephemeral return ports** — a very common 'it worked yesterday' cause. Refused-vs-timeout on the box localizes it further.

Official docs

- [Amazon EC2 User Guide](#)

 Spoken answer · ~75 s

For a surprise AWS bill, I first find the driver, then stop the spend, then prevent a repeat. In Cost Explorer I group by service, then by usage type, region, linked account, and tag, and compare against last month to see what's new or grew. Common causes are data transfer, especially cross-AZ or NAT gateway processing; runaway logging in CloudWatch; forgotten or oversized instances; unattached volumes and old snapshots; a misconfigured autoscaler or retry loop; and a compromised credential mining crypto in an unused region. Cost Anomaly Detection and the Cost and Usage Report help isolate it. To stop the bleed: terminate or resize the offending resources, fix the loop or leak, and if there's any sign of compromise, treat it as a security incident, rotating credentials. Then prevention: enforce cost allocation tags, set budgets with alerts at several thresholds, enable anomaly detection, add service control policies to block unused regions, review rightsizing and commitments like savings plans, and give teams visibility with showback so they own their spend. I also document the root cause in a short postmortem.

Likely follow-ups

- How would you detect a crypto-mining compromise from the bill?
- Why are NAT gateway charges a frequent surprise?

 SYMPTOM

Finance flags that the cloud bill jumped sharply versus last month with no obvious reason. You need to find what's driving it and stop further spend fast.

 INVESTIGATION

Find the driver: which service, account, region, and dimension?

Cost Explorer: group by service, then by usage type / region / tag

Look for the delta vs last month -> what's NEW or grew?

Usual spike causes:

- data transfer (esp. cross-AZ / cross-region / NAT gateway egress)

- a runaway resource (forgotten large instances, unbounded autoscaling)

- storage growth (un-lifecycle S3, orphaned EBS/snapshots)

- logging/observability ingestion exploding

A bill spike is debugged like an incident: localize via Cost Explorer (group by service, then usage type/region/tag) to find the delta versus last month, which points at a specific driver — frequently data transfer (cross-AZ/NAT egress), a runaway/forgotten resource, or storage/observability growth.

● ROOT CAUSE

A misconfigured service was sending large traffic cross-AZ (and through the NAT gateway), so data-transfer charges exploded — an invisible cost that doesn't show up as 'compute' and is easy to miss without grouping by usage type.

● FIX / MITIGATION

```
# Stop the bleed, then optimize
```

```
# kill/right-size the runaway resource; fix the cross-AZ traffic pattern
```

```
# add S3 lifecycle policies; clean orphaned EBS/snapshots
```

```
# set budgets + anomaly detection so the NEXT spike alerts early
```

● PREVENTION

Tag resources for cost attribution, set AWS Budgets + Cost Anomaly Detection (alert on spikes early), and review data-transfer architecture (keep chatty traffic in-AZ, use VPC endpoints to avoid NAT egress). FinOps accountability: make teams see their own spend.

Interview tip: Treat cost like an incident: **localize with Cost Explorer (group by service → usage type → tag), find the delta, stop the driver.** The frequently-missed culprit is **data transfer (cross-AZ / NAT egress)**, not compute. Prevention = budgets + anomaly detection + cost tagging. This FinOps awareness is increasingly expected of SRE/platform roles.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S47 An application gets 'AccessDenied' calling an AWS API despite 'having the right IAM role.' How do you debug it?

Cloud / AWS

 Spoken answer · ~76 s

AccessDenied despite having the right role means one of IAM's several evaluation layers is saying no. First I verify identity with sts get-caller-identity, because the app may not be using the role everyone assumes; maybe it picks up instance credentials, environment variables, or a different profile. Then I check each layer. The role's identity policy: does it allow this exact action and resource, including the object path with a slash-star suffix for S3? Then everything that can deny: an explicit deny anywhere, a permissions boundary, a service control policy at the organization level, and a session policy. For resource-based policies like an S3 bucket policy, check that they don't deny, and for cross-account access both sides must allow. For S3 specifically, check Block Public Access, VPC endpoint policies, and KMS key policies if the object is encrypted, since the role also needs kms Decrypt. Conditions on source IP, VPC, MFA, or tags are another trap. Tools: the IAM policy simulator, CloudTrail, whose error message often names the denying policy type, and Access Analyzer. Once found, I make the minimal fix, keeping least privilege.

Likely follow-ups

- What does an explicit deny do to an allow elsewhere?
- Why might KMS cause an S3 AccessDenied?

SYMPTOM

Your app, running with an IAM role, gets AccessDenied calling an AWS API (say, reading from S3). Everyone insists the role 'has the permission.' It still fails.

INVESTIGATION

```
# IAM is deny-by-default with multiple evaluation layers – check each
```

```
# 1. Is the app even USING the role you think? (which identity is it?)
```

```
aws sts get-caller-identity # confirms the ACTUAL principal in use
```

```
# 2. Does an EXPLICIT DENY override the allow? (SCP, permission boundary,
```

```
# resource policy, session policy – explicit deny always wins)
```

```
# 3. Resource policy (e.g. S3 bucket policy / KMS key policy) also allow it?
```

```
# 4. For cross-account: BOTH identity policy AND resource policy must allow
```

```
# decode the denied message; use IAM Policy Simulator / Access Analyzer
```

IAM is deny-by-default and evaluates several layers — identity policy, resource policy, SCPs, permission boundaries, session policies — where any explicit deny wins. 'Has the permission' usually means the identity policy allows it, but an SCP/boundary/resource policy is denying, or the app isn't using the role you assume.

● ROOT CAUSE

An Organizations SCP (or a permission boundary) explicitly denied the action at the account level, overriding the role's allow. The role's policy looked correct in isolation, but the higher-level guardrail vetoed it — explicit deny beats any allow.

● FIX / MITIGATION

```
# Resolve by finding the denying layer
```

```
# get-caller-identity to confirm the principal
```

```
# check SCPs, permission boundaries, and resource (bucket/key) policies
```

```
# adjust the layer that denies (or the role) following least privilege
```

● PREVENTION

Use the IAM Policy Simulator and Access Analyzer to reason about effective permissions across layers. Prefer IRSA/instance roles (not long-lived keys), least privilege, and understand the evaluation order: explicit deny > explicit allow > implicit deny.

Interview tip: Show you know **IAM is multi-layered and deny-by-default, where explicit deny (SCP, boundary, resource policy) overrides any allow**. First verify the actual principal with `get-caller-identity`, then find the denying layer. 'The role has it' ignoring SCPs/boundaries is the classic misunderstanding this question targets.

Official docs

- [AWS IAM: Roles](#)

Spoken answer · ~74 s

With RDS I can't SSH in, so I use native observability. First CloudWatch: CPU, freeable memory, read and write IOPS and latency, database connections, and for gp2 volumes the burst balance, because running out of IOPS credits causes a sudden slowdown. Then Performance Insights, which shows database load broken down by top SQL, wait events, and users, so I immediately see whether the load is CPU, I/O, or locks. Enhanced Monitoring gives OS-level metrics. Then I look at the database itself: slow query log, missing indexes, a changed query plan, lock contention or long-running transactions, and connection exhaustion from an app opening too many connections, where a pool or RDS Proxy helps. I correlate with deploys and traffic: a new query pattern, a missing index in a new feature, or autovacuum and table bloat on Postgres. Replica lag matters if reads go to replicas. Fixes range from adding an index or rewriting a query, to scaling the instance class or storage IOPS, adding read replicas or caching, or tuning parameters. I measure again afterwards to confirm it helped.

Likely follow-ups

- What would Performance Insights tell you that CloudWatch doesn't?
- Why does gp2 burst balance cause sudden slowdowns?

SYMPTOM

Your RDS-backed app slowed down. The database is the suspected bottleneck. You can't SSH into a managed DB, so how do you diagnose it?

INVESTIGATION

```
# Use RDS-native observability (no SSH on managed DBs)
```

```
# CloudWatch: CPUUtilization, FreeableMemory, Read/WriteIOPS, DBConnections,
```

```
# ReadLatency/WriteLatency, BurstBalance (gp2 IOPS credits!)
```

```
# Performance Insights: top SQL by load, wait events (lock? IO? CPU?)
```

```
# Common causes:
```

```
# - a slow/expensive query or missing index (a new query pattern)
```

```
# - connection exhaustion (too many app instances x pool size > max_connections)
```

```
# - IOPS/burst-credit exhaustion; storage autoscaling lag
```

Managed databases are diagnosed through their metrics and query insights, not the host. Performance Insights surfaces the top SQL and wait events (telling you if it's CPU-, IO-, or lock-bound), while CloudWatch shows connections, IOPS, and burst-credit exhaustion — the usual RDS culprits.

● ROOT CAUSE

DBConnections was pegged at the max: too many app instances each with a connection pool exceeded RDS max_connections, so new queries queued waiting for a connection — the DB wasn't CPU-bound, it was connection-starved.

● FIX / MITIGATION

Relieve the bottleneck

connection exhaustion -> add a connection proxy (RDS Proxy) / cut pool sizes

slow query -> add the missing index / fix the query

IOPS exhaustion -> move to gp3/provisioned IOPS; scale storage

read-heavy -> add read replicas (mind replication lag)

● PREVENTION

Front RDS with a connection proxy to absorb connection churn, right-size pools ($\text{app_instances} \times \text{pool} \leq \text{max_connections}$), use Performance Insights continuously, and pick storage (gp3/io2) for the IOPS profile. Read replicas for read scaling, mindful of lag for read-after-write.

Interview tip: For managed DBs, lead with **Performance Insights (top SQL + wait events)** and **CloudWatch (connections, IOPS, burst credits)** since you can't SSH in. The most common non-obvious cause is **connection-pool exhaustion ($\text{instances} \times \text{pool} > \text{max_connections}$)**, fixed with RDS Proxy / smaller pools — not a slow query. Burst-credit exhaustion on gp2 is another classic.

Official docs

- [Amazon RDS User Guide](#)

 Spoken answer · ~78 s

If an Auto Scaling group isn't scaling during a surge, I check reaction speed, limits, and whether instances can actually launch. First, is it scaling on the right metric? CPU may look fine while the real bottleneck is queue depth or latency, so target tracking on request count per target or queue backlog is better. Are the thresholds, evaluation periods, and cooldowns too conservative, making it react in minutes when the surge takes seconds? Then limits: has the group hit its maximum size, the account's EC2 vCPU quota, or a lack of capacity for the instance type in the AZ? Check the scaling activity history, which shows failed launches with reasons, such as an invalid AMI, a missing IAM profile, subnet IP exhaustion, or insufficient capacity. Another factor is instance warm-up: even when scaling works, new instances need time to boot, register with the load balancer, and pass health checks, so you need headroom or predictive and scheduled scaling for known peaks, plus warm pools. Fixes: better metrics and policies, higher limits, multiple instance types and AZs, and alarms on scaling failures. Load test the scaling.

Likely follow-ups

- Why is request count per target often a better scaling metric than CPU?
- What are warm pools?

 SYMPTOM

Traffic surged and latency is climbing, but the Auto Scaling Group isn't adding instances fast enough (or at all). Capacity is stuck while demand grows.

 INVESTIGATION

```
# Why an ASG fails to scale out in time:
```

```
# - scaling on the WRONG metric (CPU, but the bottleneck is queue depth/latency)
```

```
# - scaling policy thresholds/cooldowns too conservative -> slow reaction
```

```
# - hit the ASG MAX size (can't grow further)
```

```
# - hit account/instance-type capacity or quota limits (no instances available)
```

```
# - launch/boot + warmup time too long for a sharp spike (provisioning lag)
```

```
# - new instances failing health checks -> ASG terminates them in a loop
```

ASGs react with a lag and only within their limits. Failures to scale come from scaling on a metric that doesn't track the real bottleneck, conservative thresholds/cooldowns, hitting max size or capacity/quota, slow boot+warmup versus a sharp spike, or new instances failing health checks and churning.

● ROOT CAUSE

The ASG was at its configured max size, so it physically couldn't add more — and even below that, the scale-out was triggered on CPU while the actual bottleneck was request queue depth, so it under-reacted to the surge.

● FIX / MITIGATION

```
# Fix scaling responsiveness + limits
```

```
# raise ASG max + ensure instance-type capacity/quota headroom
```

```
# scale on the bottleneck signal (req/s, queue depth) not just CPU
```

```
# pre-provision/pre-warm for KNOWN spikes (autoscaling lags sharp surges)
```

```
# tune cooldowns; fix health checks so new instances stay up
```

● PREVENTION

For known spikes, pre-provision capacity — reactive autoscaling can't match a sharp surge due to boot/warmup lag. Scale on the metric that predicts user-facing latency (queue depth for workers), keep max/quota headroom, and load-test the scaling behavior before the event.

Interview tip: Cover both **limits (max size, quota)** and **responsiveness (wrong metric, cooldowns, boot lag)**.

The senior capacity insight: **autoscaling has provisioning lag, so pre-provision for known spikes and scale on the bottleneck signal (queue depth), not CPU**. Health-check churn terminating new instances is a sneaky cause worth naming.

Official docs

- [Amazon EC2 Auto Scaling](#)

Spoken answer · ~75 s

Multi-AZ only helps if the remaining zones can absorb the loss, and in practice that is often untrue. Investigating, the first question is capacity: were the other AZs sized to carry the full load? With three AZs each running at seventy percent, losing one pushes the others past a hundred. You need N plus one headroom or fast autoscaling. Second, did traffic actually drain from the bad AZ? Health checks may have been too lenient, so the load balancer kept sending requests to unhealthy targets, or DNS caching kept clients pinned. Third, hidden single-AZ dependencies: a database primary, a NAT gateway, a cache, or a single-AZ queue, so the app was multi-AZ but its dependency wasn't. Fourth, cross-AZ imbalance, for example an uneven instance distribution, or sticky sessions. Fifth, failover time: how long does database failover or leader election take, and were clients retrying correctly? Prevention: test AZ loss deliberately with game days or fault injection, size for it, use zone-aware health checks and connection draining, ensure each tier is truly multi-AZ, and alert on the per-AZ error rate.

Likely follow-ups

- How much headroom do you need to survive losing one of three AZs?
- How would you test AZ failure safely?

SYMPTOM

Your service runs across multiple AZs for resilience. One AZ had an incident — and your service degraded anyway, which shouldn't happen if multi-AZ failover works. Why?

INVESTIGATION

```
# Multi-AZ only helps if you can actually SHED the bad AZ
```

```
# Investigate why the AZ loss wasn't absorbed:
```

```
# - capacity: were the other AZs sized to carry the full load? (often NOT)
```

```
# - was traffic actually drained from the bad AZ? (health checks failing fast?)
```

```
# - a SHARED singleton dependency in the bad AZ (a primary DB, a cache)
```

```
# - cross-AZ calls still routed INTO the failing AZ
```

```
# - 'gray' AZ failure: slow/degraded, not cleanly down -> health checks passed
```

Multi-AZ protects you only if the surviving AZs have the capacity to carry full load AND traffic is actually drained from the bad AZ. Common gaps: the other AZs weren't sized for N+1 at peak, a singleton dependency lived in the failing AZ, or a gray (degraded-not-dead) AZ kept passing health checks so traffic kept flowing to it.

● ROOT CAUSE

The primary database lived in the affected AZ and didn't fail over cleanly, and the other AZs were sized for normal split load, not for absorbing 100% — so when the bad AZ degraded, the remaining capacity was insufficient and a shared dependency was impaired.

● FIX / MITIGATION

```
# Restore + correct the gaps
```

```
# drain the bad AZ (fail over the DB, pull its targets out)
```

```
# ensure surviving AZs can carry full load (capacity for AZ loss)
```

```
# eliminate single-AZ singletons; verify cross-AZ failover
```

● PREVENTION

Size capacity so any one AZ can be lost at peak (plan for the failure domain, not the average). Remove single-AZ singletons (multi-AZ databases with tested failover). Use health checks that detect gray failures so traffic actually drains. Run game days that kill an AZ to validate it.

Interview tip: The nuance: **multi-AZ isn't automatic resilience — surviving AZs must be sized to carry full load, traffic must actually drain, and no singleton can live in one AZ.** Gray AZ failures (degraded, health checks still pass) are a classic gap. The capacity rule: size for losing a failure domain at peak, and game-day it.

📖 Official docs

- [Amazon RDS: Multi-AZ deployments](#)

Spoken answer · ~77 s

Lambda has its own failure modes, so I read the Lambda metrics first. Throttles mean I hit a concurrency limit, either the account-wide limit or a function's reserved concurrency; the fix is to raise limits, request an increase, or smooth the load with a queue. Duration near the timeout points to cold starts or a slow downstream, very often a database that can't handle the sudden number of connections, since each concurrent Lambda opens its own. ConcurrentExecutions shows whether the spike is real. For cold starts, look at init duration in logs and consider provisioned concurrency, smaller packages, and keeping initialisation light. For the database, use RDS Proxy or a pooled data API, and cap concurrency so Lambda doesn't overwhelm it. Also check memory settings, since memory also sets CPU, so more memory can shorten duration, and X-Ray traces show where time is spent. For async invocations, check retries and dead-letter queues, and for stream sources the batch size and iterator age. Longer term: put a queue like SQS in front for buffering, set sensible timeouts, make handlers idempotent, and alarm on throttles and duration.

Likely follow-ups

- Why can a Lambda spike overwhelm a database?
- When would you use provisioned concurrency?

SYMPTOM

A Lambda-based endpoint starts timing out and throwing throttling errors during a traffic increase. It worked fine at low volume.

INVESTIGATION

```
# Serverless-specific failure modes
```

```
# CloudWatch + Lambda metrics:
```

```
# Throttles -> hit concurrency limit (account/function reserved concurrency)
```

```
# Duration near timeout -> cold starts or a slow downstream (often the DB)
```

```
# ConcurrentExecutions -> approaching the ceiling
```

```
# Cold starts: spiky latency on scale-up (VPC-attached / large package = worse)
```

```
# Downstream: Lambda scales FAST and can overwhelm a non-serverless DB
```

```
# (connection storm to RDS) -> the DB throttles, Lambda times out waiting
```

Lambda failures under load are usually concurrency throttling (hit the function/account limit) or a downstream that can't keep up. A subtle one: Lambda scales out far faster than a traditional database can accept connections, so a surge spawns a connection storm that overwhelms the DB, and Lambdas time out waiting.

● ROOT CAUSE

A traffic spike scaled Lambda to hundreds of concurrent executions, each opening a DB connection — instantly exhausting RDS max_connections. The DB rejected/queued connections, so Lambdas blocked and hit their timeout; some invocations were also throttled at the concurrency ceiling.

● FIX / MITIGATION

```
# Fix concurrency + downstream pressure
```

```
# raise/managed concurrency limits (reserved + provisioned concurrency)
```

```
# put RDS Proxy between Lambda and the DB to pool/limit connections
```

```
# provisioned concurrency to cut cold starts on critical paths
```

```
# add backoff; cap fan-out so Lambda can't overwhelm downstreams
```

● PREVENTION

Use RDS Proxy (or a connection-pooling layer) so fast-scaling Lambda doesn't storm a database. Set reserved/provisioned concurrency deliberately. Mind cold starts for latency-sensitive paths. Recognize that serverless shifts the bottleneck to whatever it calls.

Interview tip: Name the serverless-specific traps: **concurrency throttling, cold starts, and — the big one — Lambda scaling faster than a database accepts connections (connection storm), fixed with RDS Proxy.** The insight that 'serverless moves the bottleneck to its downstreams' shows you understand the architecture, not just the metrics.

Official docs

- [AWS Lambda: Concurrency](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [AWS Lambda Developer Guide](#)

Spoken answer · ~77 s

When S3 plus CloudFront shows stale content after a deploy, the new files are in the bucket but some cache layer is serving the old version. The layers: the CloudFront edge cache still holds the old objects until the TTL expires; the browser cache honours Cache-Control or Expires headers; and sometimes an intermediate or service worker cache. I check response headers: X-Cache says hit or miss from CloudFront, Age shows how long it has been cached, and Cache-Control shows the TTL I set. Fixes: create a CloudFront invalidation for changed paths, which works but is slow and has a cost at scale; however the better practice is content-hashed filenames for assets, like app dot abc123 dot js, with a very long cache lifetime, so a new deploy produces new URLs and nothing needs invalidating. The entry point, index.html, gets a short TTL or no-cache so it always references the latest hashed assets. Also check that the deploy synced to the right bucket and path, the origin path setting, and that the distribution isn't pointing at a different origin. Automate invalidation of index.html in the pipeline.

Likely follow-ups

- Why are hashed filenames better than invalidations?
- How do you read the X-Cache and Age headers?

SYMPTOM

You deployed updated static assets to S3 behind CloudFront, but users still see the old version. The new files are definitely in the bucket. The CDN is serving stale content.

INVESTIGATION

```
# CDN/cache layers serve stale content until invalidated/expired

# Where the old version can persist:
#   - CloudFront edge cache still holding old objects (TTL not expired)
#   - browser cache (Cache-Control headers)
#   - you uploaded but didn't INVALIDATE the CDN

# Check the served object's headers (Age, Cache-Control, X-Cache HIT/MISS)

aws cloudfront create-invalidation --paths '/*' # force edge refresh
```

Static deploys behind a CDN are a caching problem: the new files are in S3, but CloudFront edges keep serving cached copies until their TTL expires or you explicitly invalidate. Browsers add another cache layer. The fix is invalidation plus correct cache-control headers.

● ROOT CAUSE

The deploy updated S3 but didn't invalidate CloudFront, and the objects had a long TTL — so edge locations kept serving the old cached version until you forced an invalidation.

● FIX / MITIGATION

```
# Refresh the caches correctly
```

```
# create a CloudFront invalidation for changed paths (or use versioned filenames)
```

```
# set appropriate Cache-Control / TTLs
```

```
# verify X-Cache shows a MISS/refetch and Age resets
```

● PREVENTION

Use content-hashed/versioned filenames (e.g. app.[hash].js) so new deploys are new URLs that never collide with cached old ones — the cleanest pattern, avoiding invalidations entirely. Set deliberate cache-control TTLs. Make CDN invalidation part of the deploy pipeline if you don't version filenames.

Interview tip: The diagnosis: **new files in S3 but stale at the edge = CDN/browser caching — invalidate or rely on TTL**. The superior prevention is **content-hashed filenames** so each deploy produces new URLs (no invalidation needed, no stale-cache risk). Reading X-Cache/Age headers to confirm HIT vs MISS shows hands-on CDN debugging.

Official docs

- [Amazon CloudFront](#)
- [Amazon S3 User Guide](#)

 Spoken answer · ~78 s

For a stateful migration I plan around state and reversibility, not just compute. First, stand up the new environment in parallel without touching the old one. Second, replicate data continuously from old to new, using database replication, AWS DMS, or snapshots plus change capture, so the copy is nearly current, rather than a one-shot copy during downtime. Third, validate: run checksums and row counts, and test the new stack with shadow or replayed traffic. Fourth, cut over with a short planned window: freeze or briefly stop writes, let replication drain to zero lag, promote the new database, and switch traffic through DNS with a low TTL lowered in advance, or via a load balancer or traffic weighting. I move traffic gradually where possible, canary first. Fifth, keep rollback ready: keep the old environment running, and ideally reverse replication, so I can fail back without data loss. Beyond that: communicate the window, rehearse the whole thing in staging with a timed runbook, define success metrics, watch the SLIs, and only decommission the old environment after a bake period. Don't forget secrets, IAM, networking, and dependencies in the new region.

Likely follow-ups

- Why lower the DNS TTL before cutover?
- How do you make the migration reversible?

 SYMPTOM

Leadership wants a stateful service (with a database) moved to a new region or onto new instance types, with as little downtime as possible. How do you approach the migration safely?

 INVESTIGATION

- # Plan around STATE + reversibility, not just compute
- # 1. Stand up the new environment in parallel (don't mutate the old)
- # 2. Replicate data continuously old -> new (ongoing sync, not a one-shot copy)
- # 3. Validate the new env under shadow/canary traffic
- # 4. Cut over at a low-traffic window: brief write-freeze or dual-write,
- # final sync, flip traffic (DNS/LB), verify
- # 5. Keep the old env warm as instant rollback until confident

Stateful migrations hinge on moving data safely and keeping a rollback path. You build the new environment in parallel, replicate data continuously, validate with real traffic, then cut over during a quiet window with a final sync — keeping the old environment as an instant fallback rather than a one-way leap.

● ROOT CAUSE

The risky part is the data: a naive 'snapshot, copy, switch' loses writes that happen during the copy. Continuous replication plus a short final-sync (with a brief write freeze or dual-write) bounds data loss and keeps RPO near zero.

● FIX / MITIGATION

```
# Execute the cutover safely
```

```
# continuous replication -> final sync at a low-traffic window
```

```
# flip traffic via DNS/LB (low TTL so you can flip back fast)
```

```
# keep old env warm for instant rollback; decommission only after soak
```

● PREVENTION

Use low DNS TTLs for a fast flip-back, validate with shadow/canary traffic before cutover, and never decommission the source until the target has soaked. Test the rollback path. Treat the migration like a deploy: reversible, incremental, validated — with data as the hard part.

Interview tip: Show that **state and reversibility drive the plan**: parallel new env, continuous replication (not a lossy one-shot copy), final sync with a brief freeze/dual-write, low-TTL traffic flip, and the old env kept warm for instant rollback. Emphasizing 'never burn the boats until the new env soaks' and bounding RPO is the senior signal.

📖 Official docs

- [Amazon EC2 User Guide](#)

Spoken answer · ~81 s

If Terraform wants to destroy and recreate a production database, I stop and don't apply. A replacement of a stateful resource means data loss, so the plan needs understanding first. I read the plan to find which attribute forces replacement; the output marks it as forces replacement. Often it's an innocent-looking change such as a renamed identifier, a changed engine setting, an availability zone, or a provider upgrade that changed behaviour. Then I choose a safe path. If the change was unintended, I revert it in code. If I need the change, I look for an in-place alternative or a modify operation. If the resource was renamed in code, I use a moved block or terraform state mv so Terraform maps it to the existing resource, instead of recreating it. If state drifted, I import the resource. Protective measures: take a final snapshot first, set lifecycle prevent_destroy on stateful resources, enable deletion protection on the database itself, require plan review in CI before apply, and restrict who can apply. If a real replacement is needed, I plan it as a migration with replication and cutover, not a destroy. Always review plans for destroy counts.

Likely follow-ups

- What do prevent_destroy and deletion protection each guard against?
- How do moved blocks avoid recreation after a rename?

SYMPTOM

You run terraform plan for a routine change and the output says it will DESTROY and recreate your production RDS instance (or another stateful resource). Applying this would wipe data. What do you do?

INVESTIGATION

```
# STOP. A destroy/recreate of a stateful resource = data loss. Do NOT apply.
```

```
# Understand WHY Terraform wants to replace it (a 'ForceNew' attribute changed):
```

```
terraform plan # read the '# forces replacement' annotations carefully
```

```
# common triggers: changing an immutable attribute (name, AZ, engine,
```

```
# subnet group, encryption setting) -> provider must recreate
```

```
# verify in state vs config what diverged
```

A destroy/recreate plan on a stateful resource is a red alert — applying it deletes the data. Terraform shows which attribute 'forces replacement' (an immutable field changed). You never apply blindly; you find the offending attribute and avoid the recreate, often by not changing the immutable field or by moving state.

● ROOT CAUSE

A change to an immutable attribute (e.g. the DB subnet group, name, or an encryption setting) triggered Terraform's force-replacement. The intended change was harmless, but it touched a field the provider can only change by recreating the resource.

● FIX / MITIGATION

```
# Avoid the destroy safely
```

```
# revert/avoid changing the immutable attribute, OR
```

```
# use lifecycle { prevent_destroy = true } as a guardrail on stateful resources
```

```
# if the change is truly needed: do a controlled migration, not an in-place destroy
```

```
# target the apply narrowly; never auto-apply stateful destroys
```

● PREVENTION

Add lifecycle prevent_destroy to critical stateful resources, require manual review/approval for any plan showing destroys (CI policy / OPA), and separate stateful resources into their own state with extra guardrails. Plan output must be read, not rubber-stamped.

Interview tip: The instinct is everything: **a destroy/recreate of a stateful resource means data loss — never apply it; find the immutable attribute forcing replacement.** Mention prevent_destroy and CI policy that blocks destroys without approval. This tests IaC discipline and whether you actually read plan output rather than auto-applying.

Official docs

- [Terraform: lifecycle meta-argument](#)

S55 A spot instance interruption took down part of your workload. How should you have architected for it?

Cloud / AWS

Spoken answer · ~74 s

Spot instances can be reclaimed with two minutes' notice, so the workload must be designed to tolerate loss. What should have been in place: handle the interruption notice, via the instance metadata or EventBridge event, by draining connections, checkpointing state, and finishing or handing off in-flight work. Keep the workload stateless or store state externally so losing an instance loses nothing. Diversify: use many instance types and AZs in the capacity pool, with an allocation strategy like price-capacity-optimized, so a single pool's reclaim doesn't take out everything. Keep a baseline of on-demand or reserved capacity for the critical part and use spot for the burst or fault-tolerant part. For batch work, make jobs small, idempotent, and retryable, with queues so interrupted work returns to the queue. In Kubernetes, use a termination handler and pod disruption budgets, and spread across nodes. Enable capacity rebalance recommendations to replace instances proactively, and let autoscaling replace lost capacity quickly. Finally, test the interruption with fault injection and monitor interruption rates. Cost savings are only real if the design absorbs the reclaim.

Likely follow-ups

- What workloads are a poor fit for Spot?
- How does capacity diversification reduce risk?

SYMPTOM

You run some workload on Spot instances for cost savings. AWS reclaimed a batch of them with a 2-minute warning, and part of your service went down because it wasn't ready for the interruption.

INVESTIGATION

```
# Spot = cheap but can be reclaimed anytime (2-min warning) -> design for it
```

```
# What should have been in place:
```

```
# - handle the interruption notice (drain, checkpoint, finish in-flight)
```

```
# - only run INTERRUPTIBLE workloads on Spot (stateless, batch, fault-tolerant)
```

```
# - diversify instance types/AZs so one capacity pool's reclamation isn't total
```

```
# - mix Spot with On-Demand/Reserved for a stable baseline
```

```
# - keep critical/stateful work OFF pure Spot
```

Spot interruptions are expected behavior, not a failure — the architecture must tolerate sudden reclamation. The fixes: only run fault-tolerant/stateless work on Spot, react to the 2-minute interruption notice (drain/checkpoint), diversify across instance types/AZs, and keep a stable On-Demand baseline for

critical capacity.

● ROOT CAUSE

The workload ran entirely on a single Spot instance type in one AZ with no interruption handling. When AWS reclaimed that capacity pool, a large fraction vanished at once and in-flight work was lost — there was no baseline and no draining.

● FIX / MITIGATION

```
# Re-architect for interruption tolerance
```

```
# handle the 2-min interruption notice: drain + checkpoint + reschedule
```

```
# diversify instance types/AZs; mix Spot + On-Demand baseline
```

```
# keep stateful/critical components off pure Spot
```

● PREVENTION

Use Spot for stateless, interruptible, diversified workloads with a guaranteed On-Demand/Reserved baseline, and always consume the interruption notice to drain gracefully. This is the bulkhead/graceful-degradation pattern applied to capacity: a reclamation should degrade, not break.

Interview tip: The framing: **Spot reclamation is expected — design for it (handle the 2-min notice, diversify pools/AZs, keep an On-Demand baseline, run only fault-tolerant work on Spot).** Treating an interruption as a design assumption rather than an incident, and keeping stateful/critical work off pure Spot, is the cost-vs-reliability judgment they're after.

Official docs

- [Amazon EC2 User Guide](#)

 Spoken answer · ~75 s

Active-active with asynchronous cross-region replication makes inconsistency inherent, because distance forces async replication, which brings lag and conflicts. So the real decision is which consistency each piece of data needs. For data that must be strongly consistent, like account balances or inventory, avoid multi-writer: designate one home region per record or per user, route writes there, and accept higher latency for remote users. For data that can be eventually consistent, make it safe: use conflict resolution strategies such as last-writer-wins with care, version vectors, or CRDTs for mergeable data like counters and sets. Reduce visible staleness with session affinity so a user keeps hitting one region, read-your-writes guarantees by reading from the region you wrote to, and showing pending state in the UI. Monitor replication lag and alert on it, and set a lag budget. Design writes to be idempotent and avoid conflicting concurrent updates where possible, for example by partitioning ownership. Technologies like DynamoDB global tables, Spanner, or CockroachDB offer different trade-offs, with Spanner paying latency for strong consistency. It's the CAP and PACELC trade-off, so be explicit about it.

Likely follow-ups

- How does a home-region-per-user design avoid conflicts?
- When is last-writer-wins dangerous?

 SYMPTOM

You run active-active across two regions for availability. Users sometimes see stale or conflicting data because writes in one region take time to replicate to the other. Replication lag is causing inconsistency.

 INVESTIGATION

```
# Active-active + async cross-region replication => inconsistency is INHERENT
```

```
# Physics: cross-region distance forces ASYNC replication -> lag + conflicts
```

```
# Decide per data type (CAP/PACELC):
```

```
# - strong consistency needed (money, inventory) -> route writes to one region
```

```
# (single-writer) or use a globally-consistent store (higher latency)
```

```
# - availability/latency over consistency (sessions, carts) -> AP + reconcile
```

```
# read-your-writes: pin a user's session to their write region
```

In active-active across regions, the speed of light forces asynchronous replication, so lag and write conflicts are inherent — you can't make it both fully consistent and low-latency (CAP/PACELC). The resolution is per-data-type: strong consistency for money (single-writer or global store), availability-with-

reconciliation for engagement data.

● ROOT CAUSE

Both regions accepted writes to the same records, and async replication meant a user could write in one region and read stale data from the other before it replicated — and concurrent writes to the same key created conflicts needing resolution.

● FIX / MITIGATION

```
# Choose consistency per data flow
```

```
# critical data -> single-writer region or a strongly-consistent global DB
```

```
# tolerant data -> accept eventual consistency + conflict resolution (LWW/CRDT)
```

```
# pin sessions for read-your-writes; surface 'eventually consistent' in UX
```

● PREVENTION

Don't pick one consistency model for the whole system — choose per data flow: consistency for ledgers/inventory, availability for sessions/carts. Use single-writer or globally-consistent stores where correctness matters. Active-active multi-region is powerful but trades away simple consistency.

Interview tip: The senior judgment: **active-active cross-region forces async replication, so inconsistency is inherent — choose consistency per data type (CAP/PACELC), not for 'the system'!** Money gets a single-writer/strong store; engagement data gets eventual consistency + conflict resolution. Read-your-writes via session pinning is a great concrete technique.

Official docs

- [Amazon S3: Replication](#)
- [PostgreSQL: High availability, load balancing, replication](#)

S57 How do you investigate and remediate a publicly-exposed cloud resource (e.g. an open S3 bucket or security group)?

Cloud / AWS

 Spoken answer · ~76 s

A publicly exposed bucket or security group is a security incident, so I assume exposure and work in order: contain, assess, remediate, prevent. Contain immediately: enable Block Public Access on the bucket or tighten the security group to known CIDRs, which is fast and reversible. Assess: what was exposed and for how long? CloudTrail, S3 server access logs or data events, and VPC Flow Logs show who accessed it; check whether sensitive data was involved, and whether credentials or keys were in the exposed data. If access by unknown parties is found, escalate to the security team, rotate any exposed secrets, and follow legal and compliance obligations for notification. Remediate: fix the root cause, which is usually a misconfigured template, a manual console change, or a permissive default. Prevent: guardrails at the account level, such as S3 Block Public Access enforced organisation-wide, service control policies, and AWS Config rules or Security Hub with auto-remediation to detect and fix public resources; infrastructure as code with policy-as-code checks in CI, like OPA or Checkov; and least-privilege reviews. Finally, a blameless postmortem with tracked actions.

Likely follow-ups

- How do you determine whether the data was actually accessed?
- What organisation-level controls prevent public buckets?

SYMPTOM

A scan (or worse, a researcher) reports that an S3 bucket is world-readable, or a security group allows 0.0.0.0/0 to a sensitive port. Sensitive data or access may be exposed.

INVESTIGATION

```
# Treat as a security incident: assume exposure -> contain, assess, remediate
```

```
# 1. CONTAIN immediately: lock down the resource
```

```
# (block public access on the bucket / tighten the SG to known CIDRs)
```

```
# 2. ASSESS exposure: access logs – was the data actually accessed? by whom?
```

```
# (S3 access logs / CloudTrail / VPC flow logs)
```

```
# 3. If accessed: data-breach response (rotate any exposed secrets, notify per policy)
```

```
# 4. find others like it: scan the whole estate for the same misconfig
```

An exposed cloud resource is a security incident: contain it immediately (block public access / tighten the SG), then determine from access logs whether anything was actually reached, escalate to breach response if so, and crucially scan for the same misconfiguration elsewhere — one open bucket usually

means a systemic gap.

● ROOT CAUSE

A bucket policy / ACL left public access enabled (or an SG rule used 0.0.0.0/0 for convenience and was never tightened). The remediation isn't just this one resource — it's the absence of a guardrail preventing public exposure across the org.

● FIX / MITIGATION

```
# Remediate the instance and the class
```

```
# enable account-level S3 Block Public Access; remove 0.0.0.0/0 SG rules
```

```
# rotate anything that may have been exposed
```

```
# add preventive guardrails so it can't recur (SCPs, Config rules, IaC policy)
```

● PREVENTION

Prevent the whole class: account-level S3 Block Public Access, SCPs that forbid public exposure, AWS Config/Access Analyzer rules that detect and alert (or auto-remediate) public resources, and IaC policy checks (OPA/tfsec) in CI so a public resource can't be merged. Least privilege by default.

Interview tip: Respond as a security incident — **contain first, assess exposure via logs, then remediate the instance AND the class**. The systemic move is **preventive guardrails (S3 Block Public Access, SCPs, Config rules, IaC policy in CI)** so it can't recur. 'One open bucket means scan for all of them' shows you fix classes, not instances.

Official docs

- [Amazon VPC: Security groups](#)
- [Amazon S3 User Guide](#)

Spoken answer · ~79 s

When the problem is the cloud provider, I can't fix it, so I focus on mitigation, communication, and learning. First confirm it's the provider: check the status page and the provider's health dashboard, and correlate with my own signals so I'm not blaming the provider for my bug. Then mitigate what I control. If the service is multi-region, fail over traffic to a healthy region, using DNS or global load balancing. Otherwise, degrade gracefully: serve cached or read-only content, disable non-essential features, shed load, and queue writes for later. Avoid control-plane operations during the incident, such as launching instances or changing IAM, since the control plane is often the first thing to degrade; rely on the data plane and pre-provisioned capacity. Communicate: update my own status page and stakeholders regularly, tell customers what we know and what we're doing, and open a support case with the provider. Be careful with failover: not all dependencies may exist in the other region. After recovery, run a postmortem: which components lacked redundancy, was the dependency on one region acceptable for the SLO, and did our failover actually work? Test regional failover regularly.

Likely follow-ups

- Why avoid control-plane changes during a provider outage?
- How do you decide whether multi-region is worth the cost?

SYMPTOM

AWS (or your provider) is having a regional service degradation — it's on their status page. Your service depends on the affected components and is degraded. The root cause is the provider, not you.

INVESTIGATION

You can't fix the provider — focus on mitigation + comms + learning

1. Confirm it's the provider (status page + your own signals correlate)

2. Mitigate what YOU control:

- fail over to an unaffected region (if you're multi-region)

- degrade gracefully: shed dependence on the affected service

- serve cached/static fallback for the broken capability

3. Communicate honestly to users (status page); set update cadence

4. Don't burn the team chasing a root cause you can't change

When the provider is the cause you can't fix it, so the response is mitigation of what you control (failover, graceful degradation, fallbacks), honest communication, and not exhausting the team hunting a root cause that isn't yours. Whether you can actually fail over is determined by architecture decisions made long before.

● ROOT CAUSE

A specific regional service (say, a managed queue or a specific AZ) degraded. Single-region services were stuck; the question becomes whether you'd invested in multi-region or graceful degradation ahead of time to ride it out.

● FIX / MITIGATION

```
# Mitigate within your control + communicate
```

```
# fail over to a healthy region / degrade the affected feature gracefully
```

```
# fall back to cached/default behavior; shed the broken dependency
```

```
# keep users informed; postmortem your DEPENDENCY design afterward
```

● PREVENTION

Architect so a provider/region incident degrades rather than breaks: multi-region for critical paths, graceful degradation and fallbacks for provider dependencies, and counting dependency failures against your own SLO so you're incentivized to add resilience. Even provider outages are partly your design choice.

Interview tip: The mature stance: **you can't fix the provider — mitigate what you control (failover, degradation, fallbacks), communicate honestly, and don't burn the team on someone else's root cause.** The key follow-through: **count dependency outages against your SLO** so they drive investment in multi-region/graceful-degradation. Whether you can ride it out was decided at design time.

Official docs

- [AWS Builders' Library: Avoiding fallback in distributed systems](#)

 Spoken answer · ~81 s

Connection refused is a specific, useful signal, different from a timeout. It means the packet reached the host, but nothing is listening on that port, or something actively sent a TCP reset. So the network path is basically fine, and fast errors tell me I got there. I check: is the service actually running? Use `systemctl status`, `ps`, and check recent crashes. Is it listening on the right port and the right interface? `ss` minus `tlnp` shows listening sockets; a service bound to `127.0.0.1` only will refuse external connections, a very common mistake, and so will the wrong port number after a config change. If it's a container or Kubernetes, check that the container port, Service `targetPort`, and endpoints match, and that the pod is Ready. Check whether a local firewall is configured to reject with a reset instead of drop. Behind a load balancer, there may be no healthy targets, or the service restarted and hasn't yet bound its port. A listen backlog overflow can also cause resets. Fix by starting the service, correcting the bind address or port, or fixing the deployment, then verify with a connection test from the client side.

Likely follow-ups

- Why does binding to `127.0.0.1` cause refused connections from other hosts?
- How does refused differ from a firewall DROP?

 SYMPTOM

Clients trying to reach your service get 'connection refused' — fast, immediate errors (not hangs). What does the refused (vs timeout) distinction tell you, and where do you look?

 INVESTIGATION

```
# 'Refused' is a SPECIFIC, useful signal (different from 'timeout')
```

```
# Refused = the packet REACHED the host, but nothing is listening on that
```

```
# port (or it actively sent a TCP RST). You got there; no one answered.
```

```
# (Timeout, by contrast = packets silently dropped: firewall/route/host down)
```

```
# So check the SERVER side:
```

```
ss -tlnp | grep :<port> # is the process actually listening? on which address?
```

```
# is it bound to 127.0.0.1 instead of 0.0.0.0? (works locally, refuses remotely)
```

'Connection refused' is precise: the client reached the host but nothing accepted on that port (or got an RST). That rules out firewalls/routing (which cause timeouts) and points at the listener — the service is down, on the wrong port, or bound to localhost instead of all interfaces.

● ROOT CAUSE

The service was bound to 127.0.0.1 instead of 0.0.0.0, so it worked when tested locally on the box but refused every remote connection — the port was open only to loopback.

● FIX / MITIGATION

```
# Fix the listener
```

```
# bind the service to 0.0.0.0 (or the correct interface), not 127.0.0.1
```

```
# confirm it's listening on the expected port (ss -tlnp)
```

```
# restart/redeploy; verify remote clients now connect
```

● PREVENTION

Monitor that the service is listening on the expected address:port (synthetic check). The refused-vs-timeout distinction should be in every responder's mental model — it instantly separates 'nothing listening' from 'traffic blocked.' The 127.0.0.1-vs-0.0.0.0 binding bug is a perennial classic.

Interview tip: Nail the distinction: **refused = reached the host, nothing listening (or RST); timeout = packets dropped (firewall/route/down)**. Refused sends you to the server's listener — and the canonical cause is **binding to 127.0.0.1 instead of 0.0.0.0** (works locally, refuses remotely). This single distinction speeds up countless network incidents.

Official docs

- [RFC 9293: TCP](#)

Spoken answer · ~78 s

A timeout means a silent drop somewhere on the path, so my job is to localise where packets die. The method is capturing on both ends. On the client, I run `tcpdump` for the destination and see whether the SYN goes out and whether a SYN-ACK ever returns. On the server, I capture on the same port: if the SYN never arrives, packets are being dropped in the path between; if the SYN arrives but the SYN-ACK doesn't get back, I check return routing or asymmetry. If it arrives and the server replies, the problem is on the way back. Then I narrow down. Cloud: security groups and network ACLs, route tables, and flow logs showing reject records. Network: firewalls, load balancer health, or a middlebox. Host: iptables or nftables drops, which I can see with counters, a full SYN backlog, or the conntrack table being full. `traceroute` or `mtr` shows where the path stops, though remember some hops deprioritise ICMP. MTU black holes cause hangs on larger packets only. Once localised to a hop or rule, fix it, and add monitoring that tests that path regularly.

Likely follow-ups

- How does capturing on both ends localise the drop?
- What does a SYN with no SYN-ACK tell you?

SYMPTOM

Clients get timeouts reaching your service — connections just hang and eventually fail, with no refused/RST. Packets are vanishing somewhere between client and server.

INVESTIGATION

```
# Timeout = silent drop. Find WHERE in the path packets die.
```

```
# Capture on BOTH ends to localize:
```

```
tcpdump -i any -nn 'tcp and host <server> and port  
'
```

```
# - client sends SYN but server NEVER sees it -> dropped in between
```

```
# (firewall, security group/NACL, routing, network ACL)
```

```
# - server sees SYN, replies SYN-ACK, client never gets it -> return path broken
```

```
mtr -rwzbc 100 <server> # per-hop loss/latency along the path
```

A timeout means packets are being silently dropped, unlike a refused (RST). The fastest way to localize is to capture on both ends: if the server never sees the client's SYN, the drop is in the path (firewall/SG/NACL/route); if the server replies but the client never receives it, the return path is broken.

● ROOT CAUSE

A firewall/security-group rule (or a stateless NACL missing the return-traffic allow) silently dropped the packets. The server never saw the SYNs, confirming the drop was in the path — not at the listener (which would have refused).

● FIX / MITIGATION

```
# Restore the path
```

```
# fix the firewall / SG / NACL rule that's dropping the traffic
```

```
# (remember NACLs are stateless: allow return/ephemeral ports too)
```

```
# verify with tcpdump that SYNs now reach the server and SYN-ACKs return
```

● PREVENTION

Manage firewall/SG/NACL rules as code with review so a silent drop can't be introduced unnoticed. Capturing on both ends is the canonical technique to localize a drop. Understand stateful (SG) vs stateless (NACL) so you allow return traffic correctly.

Interview tip: The method that impresses: **capture on both ends — if the server never sees the SYN, the drop is in the path (firewall/SG/NACL/route); if it replies but the client doesn't receive it, the return path is broken.** Combined with refused-vs-timeout, this pinpoints silent drops. The stateless-NACL return-traffic gotcha is a frequent culprit.

📖 Official docs

- [Amazon VPC: Security groups](#)

Spoken answer · ~83 s

When only one region sees intermittent latency, the path or a regional edge differs, since the code is the same. I run mtr from that region to the endpoint with many probes to see per-hop loss and latency, and I read it carefully: loss that appears at an intermediate hop but not at the final destination is just ICMP rate limiting, not real loss; only loss that persists to the end matters. Then I check the layers. DNS: are users in that region resolved to a distant or overloaded edge? CDN or edge: is that point of presence unhealthy, or cache hit ratio low there so requests go to origin across the world? Transit: a congested or rerouted ISP or peering path, which BGP changes can cause, and which looks intermittent. Origin: is the regional backend slow or its dependency, such as a database replica, far away? Real user monitoring broken down by region and ISP is the key data, and synthetic probes from that region help. Fixes include routing users to a closer edge, fixing cache configuration, adding a regional origin or replica, or working with the CDN or ISP. Alert on regional latency separately from the global average.

Likely follow-ups

- Why can intermediate-hop loss in mtr be misleading?
- What data would you want from real user monitoring?

SYMPTOM

Most users are happy, but users in one region (say, APAC) intermittently see high latency. Same service, same code — only certain geographies are affected, intermittently.

INVESTIGATION

```
# Region-specific intermittent latency -> the PATH or a regional edge differs
```

```
mtr -rwzbc 100 <endpoint> # FROM that region: per-hop loss/latency
```

```
# read mtr correctly: only loss that PERSISTS to the final hop is real
```

```
# (mid-path ICMP deprioritization looks like loss but isn't)
```

```
# check: is that region served by a distant origin? (no nearby edge/CDN PoP)
```

```
# routing change / peering issue / congested transit on that path?
```

```
# packet loss on a specific submarine/transit link -> retransmits -> latency
```

Geography-specific latency points to the network path or edge serving that region — a distant origin with no nearby CDN PoP, a routing/peering change, or a congested/lossy transit link. mtr from the affected region localizes it, with the caveat that only loss persisting to the final hop is real (mid-path ICMP deprioritization is a false signal).

● ROOT CAUSE

That region's traffic was routed to a far origin (no local CDN edge), and a congested transit link on the long path caused intermittent packet loss → TCP retransmits → latency spikes. Nearer regions hit a local edge and were unaffected.

● FIX / MITIGATION

Improve the regional path

add a CDN/edge presence (or origin) closer to that region

work with the provider on the routing/peering/transit issue

use anycast / latency-based routing to steer users to the nearest healthy PoP

● PREVENTION

Use a CDN with regional PoPs and latency-based/anycast routing so each region hits a nearby edge. Monitor per-region SLIs (RUM) so geographic degradation is visible. Separate latency (distance/path) from loss (congestion) from jitter when diagnosing.

Interview tip: Show region-aware diagnosis: **mtr from the affected region, read correctly (only loss to the final hop is real), and suspect path/edge differences — distant origin, peering/transit congestion.** The fix is edge presence + latency-based routing. Per-region RUM to even see the problem, and the mtr-misread caveat, are senior touches.

Official docs

- [Cloudflare Learning: What is a CDN?](#)

 Spoken answer · ~75 s

Cannot assign requested address, EADDRNOTAVAIL, under heavy outbound connections is ephemeral port exhaustion. Each outbound connection uses a unique tuple of source IP, source port, destination IP, and destination port, and the source ports come from a limited range, about twenty-eight thousand by default. If one client makes many connections to the same destination IP and port, it runs out of source ports, especially when closed connections sit in TIME_WAIT. I confirm with ss counting TIME_WAIT sockets and checking ip_local_port_range. Fixes, in order of preference: reuse connections with keep-alive and connection pooling, which attacks the cause rather than the symptom; widen the port range; enable tcp_tw_reuse for outbound connections; and add more source IPs or spread across more destination addresses, since the limit is per destination tuple. In a Kubernetes or NAT setup, a NAT gateway has its own port limit per destination, so add more NAT IPs. Also check for connection leaks, where the application doesn't close connections, and avoid lowering TIME_WAIT hacks like tcp_tw_recycle, which was removed because it breaks NAT. Monitor socket counts so I see it coming.

Likely follow-ups

- Why is connection pooling the best fix?
- Why does NAT make port exhaustion more likely?

 SYMPTOM

You scaled out a service that makes many outbound connections (e.g. to a database or API). Under the higher load it begins failing with 'cannot assign requested address' (EADDRNOTAVAIL).

 INVESTIGATION

```
# EADDRNOTAVAIL under high outbound connection churn = ephemeral port exhaustion
```

```
cat /proc/sys/net/ipv4/ip_local_port_range # ~28k ports by default
```

```
ss -tan state time-wait | wc -l # piles of TIME_WAIT eating source ports
```

```
# Each outbound conn to the SAME dst ip:port needs a free local source port.
```

```
# High churn -> many sockets stuck in TIME_WAIT (~60s) -> ports exhausted
```

```
# -> the kernel can't assign a source port for new connections
```

This error means the host ran out of ephemeral source ports for new outbound connections. High connection churn leaves thousands of sockets in TIME_WAIT (held ~60s by the side that closed), tying up the limited ephemeral range so new connects fail with EADDRNOTAVAIL.

 ROOT CAUSE

The service opened a fresh connection per request to the same backend instead of reusing connections. Under load, sockets churned into TIME_WAIT faster than they cleared, exhausting the ~28k ephemeral ports to that destination.

● FIX / MITIGATION

```
# Fix: reuse connections (the real fix), then tune
```

```
# use connection pooling / HTTP keep-alive -> far fewer, reused connections
```

```
# widen ip_local_port_range; enable net.ipv4.tcp_tw_reuse=1 (safe for outbound)
```

```
# do NOT use tcp_tw_recycle (removed; breaks NAT)
```

● PREVENTION

The real fix is connection reuse (pooling/keep-alive) so you don't churn ports at all; sysctl tuning (wider port range, tcp_tw_reuse) is secondary. Avoid tcp_tw_recycle (NAT-breaking, removed). This scales with the per-destination 4-tuple limit, so reuse is essential at high fan-out.

Interview tip: Trace the chain: **high connection churn** → **TIME_WAIT pileup** → **ephemeral port exhaustion** → **EADDRNOTAVAIL**. The correct primary fix is **connection pooling/keep-alive**, not sysctl tuning. Knowing tcp_tw_reuse is safe but tcp_tw_recycle is a NAT-breaking trap is the detail that signals real depth.

📖 Official docs

- [RFC 9293: TCP](#)
- [Terraform: Sensitive data in state](#)

Spoken answer · ~78 s

Small requests working while large transfers hang across a tunnel is the classic signature of an MTU and path MTU discovery black hole. A VPN or tunnel adds encapsulation overhead, so the effective MTU is smaller than the standard fifteen hundred bytes. Large packets with the don't-fragment bit set exceed the path MTU, and the router should send back an ICMP fragmentation needed message so the sender lowers its packet size. But many firewalls block ICMP, so the message never arrives and the sender keeps retransmitting oversized packets that get dropped silently; small packets and the TCP handshake work fine, which is why it looks so strange. I test with ping using the don't-fragment flag and increasing sizes to find the real MTU, and capture traffic to see retransmissions of large segments. Fixes: allow the necessary ICMP types through firewalls, lower the MTU on the tunnel interface, or clamp the TCP MSS on the tunnel endpoint, which is the most common practical fix because it makes both sides send smaller segments. Enabling PMTU blackhole detection helps too. Document the tunnel MTU so new links get configured correctly.

Likely follow-ups

- Why does blocking ICMP cause this problem?
- What is MSS clamping?

SYMPTOM

Small requests work fine everywhere. But large transfers (big uploads, bulk responses) succeed on the local network yet hang or fail when they traverse a VPN or certain WAN paths. Small packets fine, big ones stuck.

INVESTIGATION

```
# 'Small works, large hangs across a tunnel' = MTU / PMTUD black hole
```

```
# VPN/tunnel adds encapsulation overhead -> lower effective MTU
```

```
# Large packets exceed path MTU and need to fragment; if ICMP 'frag needed'
```

```
# is blocked, Path MTU Discovery fails silently -> big packets vanish
```

```
ping -M do -s 1472 <dst> # test if large (DF-set) packets get through
```

```
tracert <dst> # discover the actual path MTU
```

```
# small packets fit -> work; large ones drop -> hang on bulk transfer
```

This is the classic MTU/PMTUD black hole. A VPN/tunnel reduces the usable MTU via encapsulation overhead; large packets that exceed the path MTU must fragment, but if the ICMP 'fragmentation needed' messages are filtered, Path MTU Discovery breaks silently — small packets pass, large ones disappear.

● ROOT CAUSE

A firewall along the VPN path blocked ICMP, so PMTUD couldn't signal the sender to reduce packet size. The tunnel's lower MTU silently dropped full-size packets, hanging any transfer large enough to send them.

● FIX / MITIGATION

```
# Fix the MTU path
```

```
# allow ICMP 'fragmentation needed' (type 3 code 4) so PMTUD works
```

```
# or clamp MSS on the tunnel (MSS clamping) to fit the path MTU
```

```
# lower the interface MTU on the tunnel to a safe value
```

● PREVENTION

Enable MSS clamping on tunnels and don't blanket-block ICMP (PMTUD depends on it). Set tunnel MTUs correctly. The 'small works, large hangs' signature should immediately suggest MTU — it's a frequently-missed cause of VPN/WAN transfer failures.

Interview tip: The pattern recognition wins: **'small packets work, large ones hang across a tunnel' = MTU/PMTUD black hole, usually blocked ICMP 'frag needed.'** Fix with MSS clamping or allowing ICMP. Knowing that VPN encapsulation lowers effective MTU and that blocking all ICMP breaks PMTUD is a strong networking-depth signal.

Official docs

- [Cloudflare Learning: What is a VPN?](#)
- [Git LFS](#)

 Spoken answer · ~77 s

DNS is a shared dependency, so when it fails, it looks like everything failing. I diagnose the resolver path methodically. First, from an affected host, query the resolver directly with `dig` and look at the response time and status: `SERVFAIL`, `NXDOMAIN`, or timeouts. Check `resolv.conf`: which nameservers, the options like `timeout` and `attempts`, the search domains, and `ndots`, since a long search list multiplies queries. Compare resolvers: is one of several nameservers bad, so a fraction of lookups fail or wait for the timeout? Check the resolver itself: CPU and query load, cache hit rate, upstream forwarders, rate limits, and logs; in Kubernetes, check CoreDNS pods, their scaling, and the `contrack` UDP race. Check the upstream: authoritative servers, DNSSEC validation failures, or an expired record. Packet capture shows queries without answers or retries. Mitigations: add a local caching resolver on each node to absorb load and hide upstream blips, scale or fix the resolver, remove the bad nameserver, tune `ndots` and use fully qualified names, and make sure applications cache and use sensible timeouts. Monitor DNS latency and failure rate as its own SLI.

Likely follow-ups

- Why does a bad nameserver in the list cause intermittent slowness?
- How does a node-local cache help?

 SYMPTOM

Services intermittently fail to resolve hostnames, or resolution is slow, causing cascading timeouts and errors across many services at once. DNS is the common thread.

 INVESTIGATION

```
# DNS is a shared dependency -> its failure looks like everything failing
```

```
# Diagnose the resolver path:
```

```
dig @<resolver> <name> # is the resolver slow/failing/returning SERVFAIL?
```

```
cat /etc/resolv.conf # options ndots/timeout/attempts; multiple servers?
```

```
# high ndots -> many failed lookups (search-domain expansion) before success
```

```
# one unhealthy nameserver in the list -> per-query timeout then fallback
```

```
# resolver overloaded (e.g. CoreDNS under-provisioned) -> slow/dropped queries
```

Because DNS underlies nearly every connection, a flaky resolver makes unrelated services fail simultaneously. Diagnose the resolver path: a slow/overloaded nameserver, an unhealthy server in the list causing per-query timeouts, or high `ndots` causing many failed search-domain lookups before the real

query.

● ROOT CAUSE

The cluster's DNS (CoreDNS) was under-provisioned and overwhelmed under load, dropping/slowng queries. Combined with a high ndots value causing extra lookups per name, resolution became intermittently slow — and every service that resolves names was affected.

● FIX / MITIGATION

```
# Stabilize DNS
```

```
# scale/right-size the resolver (more CoreDNS replicas, autoscale)
```

```
# add a node-local DNS cache to absorb query load + cut latency
```

```
# tune ndots / use FQDNs to reduce lookups; remove unhealthy nameservers
```

```
# add DNS caching with sane TTLs
```

● PREVENTION

Treat DNS as a critical dependency: provision and monitor resolvers, run node-local DNS caches, tune ndots/use FQDNs, and cache with sane TTLs. Give DNS the same timeouts, redundancy, and observability as any other dependency — its failures fan out everywhere.

Interview tip: Recognize **DNS as a shared dependency whose failure makes everything look broken at once.**

Diagnose the resolver (overloaded CoreDNS, unhealthy nameserver, high ndots). Fixes: scale the resolver, node-local DNS cache, tune ndots/FQDNs. Treating DNS as a first-class dependency with caching, redundancy, and monitoring is the platform-level insight.

Official docs

- [RFC 1035: DNS implementation and specification](#)
- [Cloudflare Learning: What is DNS?](#)
- [Google Testing Blog: Flaky tests at Google](#)

 Spoken answer · ~82 s

Behind a layer-seven load balancer, the TCP source is the balancer, so the real client IP has to be carried another way. The common way is the X-Forwarded-For header, or the standard Forwarded header, where each proxy appends the address it saw. The fix is to configure the application and logging to read the client address from that header. But the critical caveat is trust: only trust the header from your own known proxies, otherwise a client can forge it and bypass rate limiting. So I configure the framework or web server to trust only the load balancer's address range, and take the right entry from the list, typically the rightmost untrusted one rather than the leftmost value the client controls. For layer-four load balancing, or where the header can't be used, the PROXY protocol passes the original source address at the start of the connection, and both sides must be configured for it. Some cloud load balancers can also preserve the source IP in network mode. After the fix, verify that logs, geolocation, and the rate limiter show real client IPs, and that spoofed headers are ignored. Also update access logs to include the header.

Likely follow-ups

- Why is trusting X-Forwarded-For blindly dangerous?
- When would you use the PROXY protocol?

 SYMPTOM

Your app sits behind a load balancer / reverse proxy. In the app's logs and rate limiter, every request appears to come from the LB's IP — so per-client rate limiting and geolocation are broken, and logs are useless for source IPs.

 INVESTIGATION

Behind an L7 proxy/LB the client IP is in a HEADER, not the TCP source

The real client IP is carried in X-Forwarded-For (or Forwarded / proxy protocol)

The app/rate-limiter must read XFF, not the TCP peer (which is the LB)

BUT: XFF is client-spoofable unless you trust only the LB's appended value

configure the app to trust the LB and parse the correct XFF entry

An L7 load balancer terminates the client's TCP connection and opens its own to the backend, so the TCP source IP is the LB. The real client IP is passed in X-Forwarded-For. The app and rate limiter must read XFF — but only the entry appended by your trusted LB, since clients can forge XFF.

 ROOT CAUSE

The app rate-limited on the TCP source address, which behind the LB is always the LB's IP — so all clients shared one bucket (and one client could exhaust everyone's limit). It needed to read X-Forwarded-For instead.

● FIX / MITIGATION

```
# Use the forwarded client IP, safely
```

```
# configure the app/framework to extract client IP from XFF (trusted proxy config)
```

```
# trust only the LB-appended entry; strip/ignore client-supplied XFF
```

```
# (for L4: use proxy protocol to preserve the source IP)
```

● PREVENTION

Configure trusted-proxy handling so the app derives client IP from XFF appended by your LB only (never trust client-supplied XFF — it's spoofable). For L4 LBs, use the proxy protocol to preserve source IPs. Then rate limiting, geo, and logging work per real client.

Interview tip: Explain that an **L7 LB makes the TCP source the LB's IP — the real client is in X-Forwarded-For**, so rate-limit/log on XFF. The security nuance that elevates the answer: **XFF is client-spoofable, so trust only the entry your LB appends**. Mentioning proxy protocol for L4 shows you know the layer distinction.

📖 Official docs

- [Cloudflare Learning: Load balancing](#)
- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)

Spoken answer · ~81 s

For intermittent TLS handshake errors with a partner, I debug the handshake directly. I run `openssl s_client` with the right server name and inspect the certificate chain, expiry, SNI, negotiated version and cipher, and whether the chain verifies against my trust store. Intermittent failures suggest the partner has multiple backends behind a load balancer that are not identical: one node may have an expired or different certificate, a missing intermediate, or different protocol and cipher settings. I run the test repeatedly, and against each resolved IP separately, to find the bad one. Other causes: the partner rotated certificates and my client has a stale trust store or pinned certificate; clock skew on my host making valid certificates look expired or not yet valid; SNI not being sent by an older client library, so the wrong certificate is served; a TLS version or cipher mismatch after the partner tightened policy; or a middlebox or proxy intercepting traffic. A packet capture shows the alert code in the handshake. Fix by updating the trust store, the client library, or time sync, or by asking the partner to fix the bad node, and monitor certificate expiry on dependencies.

Likely follow-ups

- How do you find which backend IP has the bad certificate?
- Why can clock skew cause TLS errors?

SYMPTOM

Calls from your service to a partner's HTTPS endpoint intermittently fail during the TLS handshake — not a timeout, a handshake/certificate error. Sometimes it works, sometimes not.

INVESTIGATION

```
# Debug the TLS handshake directly
```

```
openssl s_client -connect partner.com:443 -servername partner.com
```

```
# inspect: cert chain, expiry, SNI, negotiated version/cipher, CA trust
```

```
# Intermittent handshake failures – usual causes:
```

```
# - one endpoint behind the partner's LB has a bad/expired/mismatched cert
```

```
# - missing intermediate cert in the chain (some clients fail, some don't)
```

```
# - protocol/cipher mismatch (old TLS disabled on one side)
```

```
# - SNI not sent -> wrong cert served
```

TLS handshake failures (vs timeouts) point at certificate/protocol issues, and intermittency usually means a fleet inconsistency on the partner side — e.g. one server behind their LB has a bad or chain-incomplete cert. `openssl s_client` inspects the chain, version, cipher, and SNI to localize it.

● ROOT CAUSE

The partner had several servers behind a load balancer; one of them presented a cert with a missing intermediate (incomplete chain). Clients that didn't have that intermediate cached failed the handshake — but only when routed to that one server, hence the intermittency.

● FIX / MITIGATION

```
# Mitigate + escalate
```

```
# confirm which endpoint/cert fails (s_client repeatedly hits different backends)
```

```
# report the incomplete chain / bad cert to the partner to fix
```

```
# add retry (handshake failures may hit a good server next) + circuit breaker
```

```
# ensure your client trusts the proper CA + sends SNI
```

● PREVENTION

Monitor cert expiry/chain validity (yours and partners'), send SNI, and keep CA trust stores updated. Treat a flaky partner with retries + circuit breakers so intermittent handshake failures don't cascade. Inventory and alert on all certs including partner-facing ones.

Interview tip: Use `openssl s_client` to inspect the chain/version/cipher/SNI, and recognize that **intermittent handshake failures usually mean a fleet inconsistency on the other side (one server with a bad or incomplete-chain cert)**. The missing-intermediate-cert cause (works for some clients, not others) is a classic. Retry + circuit breaker contains a flaky partner.

Official docs

- [Kubernetes: Debug Services](#)
- [RFC 8446: TLS 1.3](#)
- [Cloudflare Learning: TLS handshake](#)

 Spoken answer · ~80 s

With many hops and poor tail latency, guessing is hopeless, so I use distributed tracing to break each request into spans per hop and sidecar and see which span owns the latency. I look at the slow requests specifically, the P99 traces, not the average, and compare them with fast ones. Then I separate causes. If time is spent inside a service, it's the application: slow code, a database call, or garbage collection. If time is between spans, in the gap between a caller's client span and the callee's server span, it's the network or the proxy: sidecar overhead, queuing, connection setup, or retransmissions. Mesh metrics help: the proxy's own request duration versus upstream duration. Common tail-latency contributors are retries and timeouts stacking up, head-of-line blocking, new connections without pooling, DNS or TLS handshakes, noisy neighbours, CPU throttling of the sidecar or app, and fan-out, where the slowest of many parallel calls determines overall latency. At the network layer, check retransmits and packet loss on specific nodes. Fixes: connection reuse, tuned timeouts and retry budgets, hedged requests, resource limits for sidecars, and reducing the number of hops. Measure again against the SLO.

Likely follow-ups

- Why does fan-out amplify tail latency?
- How do you tell sidecar overhead from application time?

 SYMPTOM

In a microservices architecture, P99 latency is poor end-to-end. You suspect network/mesh overhead but can't tell if it's the network, a proxy/sidecar, or an app hop. There are many services in the call chain.

 INVESTIGATION

```
# Many hops + bad tail latency -> use tracing to find WHERE time goes
```

```
# distributed tracing: break the request into spans per hop/sidecar
```

```
# -> which span owns the latency? app? network? the mesh sidecar (Envoy)?
```

```
# tail latency COMPOUNDS with fan-out: 1% slow per call x many calls
```

```
# = most end-to-end requests hit at least one slow hop
```

```
# check: sidecar/proxy overhead, mTLS handshake cost, connection pooling,
```

```
# retries adding hops, per-service P99
```

With many hops, only distributed tracing can localize tail latency to a specific span — app logic, the network, or the mesh sidecar. The key principle: tail latency compounds with fan-out, so a 1% slow rate per dependency means most multi-call requests hit at least one slow hop, making per-dependency P99

critical.

● ROOT CAUSE

A service-mesh sidecar (Envoy) was adding latency on each hop — partly mTLS handshakes without connection reuse, partly retry-on-each-hop amplifying calls. Across a deep call chain, these small per-hop costs compounded into a poor end-to-end P99.

● FIX / MITIGATION

```
# Reduce per-hop and tail cost
```

```
# enable connection reuse/keep-alive in the mesh (avoid per-call handshakes)
```

```
# tune/limit retries (retry budgets) so they don't multiply hops
```

```
# reduce call-chain depth / fan-out where possible; cache
```

```
# target the worst per-service P99 first (it dominates the chain)
```

● PREVENTION

Instrument everything with tracing so the slow hop is obvious. Understand fan-out math: improving one dependency's P99 helps every request that touches it. Use retry budgets and connection reuse in the mesh. Reduce unnecessary call-chain depth — each hop adds tail risk.

Interview tip: The standout concept: **tail latency compounds with fan-out — 1% slow per call means most fanned-out requests hit a slow hop, so per-dependency P99 matters enormously.** Use tracing to localize the span (app vs network vs sidecar), and know that mesh sidecars add mTLS/handshake/retry overhead. This separates architecture-level reasoning from 'the network is slow.'

Official docs

- [Kubernetes: Sidecar containers](#)
- [OpenTelemetry: Signals \(traces, metrics, logs\)](#)

Spoken answer · ~83 s

If a firewall or security group change broke connectivity, even locking me out, I recover through an out-of-band path that doesn't depend on the broken rule. In the cloud that means Session Manager through SSM, the EC2 serial console, or the provider's console access, which need no inbound SSH. For on-premise, use the out-of-band management interface like iLO or iDRAC, or the physical console. Then the fix is to revert the change, ideally through the same mechanism that applied it: the previous version in Terraform, or the CloudTrail record to see exactly what changed and who changed it. Revert rather than patch forward, as the quickest safe step. Be careful not to make it worse with further manual changes, and tell the team. Prevention: make firewall changes through code with peer review and plan output; test them in staging; apply with a safety net such as automatic rollback timers, for instance the iptables trick of scheduling a rule flush before applying changes so a lockout self-heals; keep a management path that isn't affected by application rules; always keep an allow rule for your own access; and roll out gradually with monitoring. A postmortem should capture the guardrails that were missing.

Likely follow-ups

- What is an out-of-band access path?
- How does a self-reverting timer protect you from lockout?

SYMPTOM

Someone applied a firewall or security-group change and it broke connectivity — possibly locking out SSH/management access to instances, or cutting traffic between tiers. You need to recover without making it worse.

INVESTIGATION

```
# Recover via an out-of-band path that doesn't depend on the broken rule
```

```
# - cloud serial console / Session Manager (SSM) – no network/SSH needed
```

```
# - revert the change at the control plane (it's likely IaC or a console edit)
```

```
# Then verify the rule logic:
```

```
# - SG (stateful): allow the needed inbound/outbound; return traffic auto-allowed
```

```
# - NACL (stateless): must allow BOTH directions incl. ephemeral return ports
```

```
# - first-match / ordering issues; an overly-broad deny above an allow
```

A connectivity-breaking firewall change is recovered through an out-of-band path that doesn't rely on the broken rule — a serial console or SSM Session Manager — then by reverting the change at the control plane. Understanding stateful (SG) vs stateless (NACL) is key to fixing the rule correctly.

● ROOT CAUSE

A new restrictive rule (or a NACL deny that broke stateless return traffic) cut access. Because SSH itself was blocked, recovery required the cloud serial console / SSM to get in and revert — you can't fix a network lockout over the network.

● FIX / MITIGATION

```
# Recover then prevent
```

```
# use serial console / SSM (out-of-band) to revert the rule
```

```
# correct SG vs NACL semantics; verify both directions for NACLs
```

```
# prevent: manage firewall rules as reviewed IaC; stage changes; keep an SSM path
```

● PREVENTION

Always keep an out-of-band access path (SSM/serial console) that doesn't depend on SSH or your network rules. Manage firewall/SG/NACL config as reviewed code, test changes in staging, and consider time-bound auto-revert for risky network changes so a lockout self-heals.

Interview tip: Two points: **recover via an out-of-band path (serial console/SSM) because you can't fix a network lockout over the network**, and correct **SG (stateful) vs NACL (stateless, needs explicit return traffic)** semantics. Prevention is firewall-rules-as-reviewed-IaC plus always maintaining an SSM/console escape hatch.

Official docs

- [Amazon VPC: Security groups](#)

Spoken answer · ~81 s

My systematic method is to walk the layers bottom-up, with each test isolating one layer. First DNS: does the name resolve, and to the right IP? I use `getent hosts` or `dig`, and compare results because the application may see something different from my terminal. Second routing: is there a path to that IP, with `ip route get`, and does the source interface make sense? Third reachability: ping or `mtr` for basic connectivity, understanding that ICMP can be blocked. Fourth the port: can I establish a TCP connection, using `nc` or `curl`, which separates refused from timeout. Refused means the host is reachable but nothing listens; timeout means something drops packets. Fifth TLS: does the handshake succeed, with `openssl s_client`, and is the certificate valid. Sixth the application: does an actual request return the right response, with `curl verbose`, and what do the logs say? At each step I note which layer fails first, and that is where I focus. If everything works from my shell but not from the service, I compare the environment: container network, proxy settings, DNS config, and security rules. Capture packets when ambiguity remains. This takes minutes and removes guesswork.

Likely follow-ups

- Why test from the same host or container the service runs in?
- How do you distinguish refused from timeout, and what does each mean?

SYMPTOM

A connection from your service to a dependency is failing and you don't yet know if it's name resolution, the network path, or the application/port. The interviewer wants your systematic method.

INVESTIGATION

Walk the layers bottom-up, each test isolating one layer:

`getent hosts <host>` # 1. DNS: does the name resolve? to the RIGHT IP?

`ip route get <ip>` # 2. Routing: is there a path? correct interface/source?

`ping <ip>` # 3. L3 reachability (ICMP may be blocked – not definitive)

`nc -vz <ip> <port>` # 4. L4: is the PORT open? (refused vs timeout!)

`curl -v <url>` # 5. L7: does the APP actually respond correctly?

refused = reached, nothing listening; timeout = dropped (firewall/route)

The disciplined method isolates one layer at a time: resolve the name (DNS), confirm a route exists, test L3 reachability, test the specific port (L4), then the application response (L7). Each step rules a layer in or out, and the refused-vs-timeout distinction at L4 immediately separates 'nothing listening' from 'traffic

blocked.'

● ROOT CAUSE

Working the layers showed DNS resolved to the wrong IP (a stale record after a migration), so every subsequent layer was being tested against the wrong host. Fixing DNS resolved it — a problem that 'feels like' a network or app issue but is actually name resolution.

● FIX / MITIGATION

```
# Fix at the layer that fails
```

```
# DNS wrong -> correct the record / cache / resolver
```

```
# no route -> fix routing; refused -> fix the listener; timeout -> fix firewall/SG
```

```
# app errors -> debug the application
```

● PREVENTION

This layered checklist (DNS → route → reach → port → app) is the universal connectivity-debugging method. Combined with refused-vs-timeout and capturing on both ends, it localizes nearly any connectivity issue quickly. Build it into runbooks so any responder can follow it.

Interview tip: Lay out the **layered checklist: DNS → route → L3 reach → L4 port (refused vs timeout) → L7 app**, each test isolating one layer. This systematic bottom-up approach — rather than guessing — is exactly what's being assessed. Noting that 'DNS resolves to the wrong IP' masquerades as a network/app problem shows the value of starting at layer one.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

 Spoken answer · ~78 s

First I decide: is this malicious or one buggy client, and at which layer? I look at the pattern: source IPs, user agents, targeted endpoints, request rates, and geographic spread. A few sources with huge volume suggests an abusive client or bot; a very distributed flood suggests a DDoS. The layer matters: volumetric network floods and SYN floods need upstream protection, while application-layer attacks target expensive endpoints. My response: protect the service first. Put the edge in front: a CDN and WAF with rate limiting, bot detection, and challenge pages; use DDoS protection from the provider, such as AWS Shield or Cloudflare. Block or throttle offending IPs, ASNs, or user agents, and rate limit per client on the expensive endpoints. Shed load gracefully: return cached content, degrade non-essential features, and protect the critical path with queueing and autoscaling, with caution because autoscaling into an attack is expensive. If it's a misbehaving legitimate client or partner, contact them and apply a quota. Communicate with stakeholders, preserve logs for analysis, and involve security. Afterwards: tighten rate limits, make expensive endpoints cacheable or authenticated, add capacity planning and runbooks, and rehearse.

Likely follow-ups

- How do you distinguish a DDoS from a buggy client?
- Why is autoscaling during an attack a risk?

 SYMPTOM

A sudden flood of traffic is overwhelming your service — it could be a DDoS attack or a single misbehaving client/bot hammering you. Legitimate users are being affected.

 INVESTIGATION

First: is it malicious (DDoS) or one buggy client? and what LAYER?

identify the pattern: source IPs, user-agents, target endpoint, geographic spread

- few sources, huge volume -> one abusive client/bot (rate-limit/block it)

- many distributed sources -> DDoS (needs upstream/edge mitigation)

- L7 (app-layer floods to an expensive endpoint) vs L3/4 (volumetric)

protect the service while you sort it out:

rate limiting, load shedding (429/503), WAF rules, edge/CDN DDoS protection

First classify: a few sources at huge volume is an abusive client (rate-limit/block), while many distributed sources is a DDoS needing edge/upstream mitigation; and L7 app-floods differ from L3/4 volumetric attacks. Meanwhile protect the service with rate limiting and load shedding so legitimate users still get served.

● ROOT CAUSE

A single misbehaving client (a buggy integration in a retry loop) was hammering an expensive endpoint — not a true DDoS. Per-client rate limiting and a temporary block of that source restored capacity for everyone else.

● FIX / MITIGATION

```
# Protect, then mitigate the source
```

```
# rate-limit per client/key; load-shed low-priority traffic (429 + Retry-After)
```

```
# block/throttle the abusive source(s); add WAF rules for the pattern
```

```
# for true DDoS: engage edge/CDN/cloud DDoS protection (absorb upstream)
```

● PREVENTION

Defend with rate limiting (token bucket per client), load shedding to protect the core, a WAF for L7 patterns, and edge/CDN DDoS protection for volumetric attacks. Doing some work well (shed 10%, serve 90%) beats collapsing under 100%. Return 429 with Retry-After so well-behaved clients back off.

Interview tip: Classify first (**single abusive client vs distributed DDoS, L7 vs L3/4**) because the mitigation differs — rate-limit/block one source, or push volumetric defense to the edge/CDN. Meanwhile **protect the service with rate limiting + load shedding (serve 90% well rather than collapse)**. Returning 429/Retry-After for graceful backoff is the cooperative-client touch.

Official docs

- [Cloudflare Learning: DDoS attacks](#)

S71 **Load average is 40 on an 8-core box, but top shows CPU mostly idle. What's going on?**

Linux Host

 Spoken answer · ~82 s

High load average with idle CPU means the load isn't CPU demand. On Linux, load average counts processes that are runnable and also those in uninterruptible sleep, state D, usually waiting on I/O. So lots of processes stuck waiting on disk or network storage inflate the load while the CPUs sit idle. I confirm with `vmstat`, watching the `b` column for blocked processes and the `wa` column for I/O wait, and I list processes in D state with `ps`. Then I find what they're waiting for: `iostat` shows disk utilisation, queue depth, and latency; `iostat` shows which processes do the I/O; and for a network filesystem like NFS, a hung or slow server causes this. Check `dmesg` for disk errors or timeouts, and a failing disk or saturated cloud volume, such as exhausted IOPS credits. Other causes include lock contention or many threads waiting on a mutex, though that usually shows as sleeping, not D state. The fix depends on the cause: reduce I/O, tune queries, add IOPS, fix the storage backend, or restore the NFS server. And I fix the alert: load average alone is a poor signal, so alert on saturation and latency instead.

Likely follow-ups

- Why does Linux count D-state processes in load average?
- What metrics would you alert on instead of load average?

 SYMPTOM

Alerts fire on load average >30. You log in expecting pegged CPUs, but `top` shows 85% idle and almost no user/sys time.

 INVESTIGATION

```
uptime
```

```
top # note %id high, but load high
```

```
vmstat 1 5 # watch the 'b' column (uninterruptible)
```

```
ps -eo state,pid,comm | awk '$1 ~ /D/'
```

Load average on Linux counts processes in runnable *and* uninterruptible (D) state. Idle CPU plus high load means processes are blocked in D-state — almost always I/O wait, often a slow or hung disk/NFS mount.

 ROOT CAUSE

Many processes sit in D-state waiting on a backing store. CPU is idle because they're blocked on the kernel, not computing. This is an I/O problem masquerading as a CPU alarm.

 FIX / MITIGATION

```
iostat -x 1 5 # %util ~100, await high on the culprit device
```

```
iotop -oPa
```

```
cat /proc/<pid>/stack # see where it's blocked in-kernel
```

● PREVENTION

Alert on what matters to users (latency, saturation) rather than raw load average. Track vmstat 'b' count and per-device %util/await so an I/O stall is named as such instead of looking like a CPU fire.

Interview tip: Knowing that Linux load includes D-state is a classic senior tell. Say it plainly: "high load + idle CPU = blocked on I/O, not compute," then go to iostat/iotop.

Official docs

- [Brendan Gregg: Linux performance](#)

S72 A process keeps getting killed and you find 'Out of memory: Killed process' in dmesg. How do you handle it?

Linux Host

Spoken answer · ~78 s

When the kernel's OOM killer terminates a process, I find out why memory ran out and whether the victim was a symptom. I read dmesg or journalctl for the kill record, which shows the victim, its memory use, and a table of all processes, so I can see who was really using memory. The killer picks victims using oom_score, roughly the biggest user, adjusted by oom_score_adj. Then diagnose: is it a leak, with memory climbing steadily until death, or a legitimate working set larger than the box, or a spike? Check available memory with free, trends in monitoring, and in containers whether the cgroup limit, not host memory, triggered it. Fixes: fix the leak using a profiler or heap dump, right-size the memory limit or instance, limit per-process memory or cache sizes, avoid overcommit surprises with sensible vm settings, and add swap only as a buffer, not a fix. Protect critical processes with a lower oom_score_adj, and for services let systemd restart them. Add alerting on memory pressure before the kill, using available memory and PSI, and make sure the service restarts automatically. Verify afterwards with load testing.

Likely follow-ups

- How does the OOM killer choose its victim?
- How is a cgroup OOM different from a host OOM?

● SYMPTOM

A service mysteriously dies every few hours with no application error. dmesg shows the OOM killer terminating it.

● INVESTIGATION

```
dmesg -T | grep -i 'killed process'
```

```
journalctl -k | grep -i oom
```

```
cat /proc/<pid>/oom_score # before it dies
```

```
free -m # available, not just free
```

The kernel ran out of allocatable memory and the OOM killer picked a victim by oom_score (roughly: biggest memory user, adjusted by oom_score_adj). Your process was the fattest target, not necessarily the leaker.

● ROOT CAUSE

Total committed memory exceeded physical + swap headroom. The killed process may be the victim, not the cause — you must find what actually grew. Check whether it's a real leak, a workload spike, or another co-tenant ballooning.

● FIX / MITIGATION

```
smem -tk # PSS per process, fairer than RSS
```

Reduce the workload / add memory / set sane cgroup limits

```
echo -1000 > /proc/<pid>/oom_score_adj # protect a critical PID (stopgap)
```

● PREVENTION

Run memory-critical services under cgroup memory limits (systemd MemoryMax= or container limits) so one process can't take down the host, and alert on MemAvailable trending down well before the killer fires.

Interview tip: Distinguish *victim* from *cause*. Mention MemAvailable vs free, oom_score_adj for protecting critical PIDs, and that swap thrash often precedes the kill.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

Spoken answer · ~78 s

A zombie is a child process that has finished, but whose parent hasn't read its exit status with `wait`. It holds only a process table entry and a PID, no memory or CPU, so a few zombies are harmless. The danger is volume: enough zombies exhaust the PID space, and then nothing can fork. You can't kill a zombie because it's already dead; the real fix is its parent. I find the common parent with `ps` looking at PPID and state Z. Then I either fix or restart the negligent parent so it reaps its children, since when the parent exits the zombies get re-parented to `init`, which reaps them. Sending `SIGCHLD` to the parent may prompt it to reap if it handles the signal correctly. The root cause is a bug in the parent: not calling `wait`, or ignoring `SIGCHLD` wrongly. In containers, a common cause is the main process not being a proper `init`, so orphaned children are never reaped; the fix is to use an `init` like `tini`, or Docker's `init` flag. Longer term, fix the code, and monitor the process count and zombie count.

Likely follow-ups

- Why can't you kill a zombie process directly?
- Why do containers commonly accumulate zombies?

SYMPTOM

Monitoring flags a climbing process count. `ps` shows hundreds of `<defunct>` entries.

INVESTIGATION

```
ps -el | grep -w Z | wc -l
```

```
ps -eo pid,ppid,state,comm | awk '$3=="Z"'
```

```
# identify the common PPID – that's the negligent parent
```

A zombie is a finished child whose exit status the parent never reaped via `wait()`. Zombies hold only a PID slot, not memory or CPU — but enough of them exhausts the PID table and blocks new forks.

ROOT CAUSE

The parent process has a bug: it isn't reaping children (missing `wait()`/`SIGCHLD` handling). The zombies themselves are already dead; the real defect is in the parent.

FIX / MITIGATION

```
kill -CHLD <ppid> # nudge the parent to reap
```

```
# if parent is broken, restart the parent – init then adopts & reaps orphans
```

```
# you cannot 'kill' a zombie; it's already dead
```

● PREVENTION

Fix the parent's child-reaping (proper SIGCHLD handler or a process supervisor). In containers, run a minimal init (tini, --init) as PID 1 so orphaned children get reaped instead of accumulating.

Interview tip: Make the PID-1-doesn't-reap point for containers — it's the modern version of this question. And be precise: zombies consume PID slots, not RAM.

📖 Official docs

- [Linux man-pages: fork\(2\)/wait\(2\)_\(zombies\)](#)

S74 Writes are failing with 'No space left on device' but df shows 40% free. What's the cause?

Linux Host

🗣 Spoken answer · ~80 s

No space left on device while df shows forty percent free means a different resource is exhausted. The most common is inodes. Every file uses one inode regardless of size, so a directory full of millions of tiny files, like session files, mail queues, or cache entries, can exhaust the inode table while bytes remain free. I confirm with df minus i, where IUse percent hits one hundred. To find the culprit, I count files per directory, for example looping over directories with find and counting, sorted. Then clean up safely, deleting old files, and with huge directories use find with delete or rsync with an empty directory, since rm with a glob can fail on argument length. Fix the root cause: add cleanup or rotation for whatever creates the files, store things differently, or recreate the filesystem with more inodes since ext4 fixes the count at creation; XFS allocates inodes dynamically. Other causes of ENOSPC with free space: a quota reached, a deleted-but-open file still holding space, a full reserved-block percentage for non-root users, or a small separate partition like the boot partition. Monitor inode usage as well as bytes.

Likely follow-ups

- Why can't you add inodes to an existing ext4 filesystem?
- What else besides inodes causes ENOSPC?

● SYMPTOM

Applications fail with ENOSPC / "No space left on device," yet df -h shows the filesystem only 40% full.

● INVESTIGATION

```
df -h /data # 40% used – misleading
```

```
df -i /data # IUse% 100% – the real story
```

```
for d in /data/*; do echo $(find "$d" | wc -l) $d; done | sort -rn | head
```

Bytes are free but *inodes* are exhausted. Each file consumes one inode regardless of size; a directory full of tiny files (sessions, cache, mail spool) burns inodes long before bytes.

● ROOT CAUSE

Inode exhaustion. Some directory is generating millions of small files — a runaway cache, unrotated session files, or a logging bug writing one file per event.

● FIX / MITIGATION

```
# delete/rotate the offending small-file directory
```

```
find /data/cache -type f -mtime +7 -delete
```

```
# inode count is fixed at mkfs time; to raise it you must reformat (-i bytes-per-inode)
```

● PREVENTION

Monitor `df -i` alongside `df -h`, set retention/rotation on anything that creates many small files, and size bytes-per-inode appropriately at format time for known small-file workloads.

Interview tip: The instant you say "ENOSPC but df shows free → check `df -i`" you signal real ops scars. Add that inode count is set at format time and can't be grown live.

📖 Official docs

- [Linux man-pages: inode\(7\)](#)

Spoken answer · ~81 s

When a host hits PID exhaustion, with errors like fork retry resource temporarily unavailable, even diagnostics fail since each command needs a fork. I check `pid_max` and the thread count, and who is spawning, but if I can't run commands, I use what's already running. If I still have a root shell, I use built-in commands only: the shell's own builtins, or `exec` to replace the current shell with a useful tool, then kill the offending process tree, for example with `kill` on the parent or `killall` by name. Sending `SIGSTOP` to the parent first stops it spawning more, then kill the group. If I can't get in, use the out-of-band console, or as a last resort reboot. After recovery, I find the cause in logs: a fork bomb, a runaway script, a bug looping and spawning processes, or a leak of threads or zombie processes. Prevention: set per-user process limits in `limits.conf` or `systemd`'s `TasksMax`, use `cgroup` pids limits for containers and services, raise `pid_max` if legitimately needed, monitor process and thread counts, and reserve a root-owned emergency shell. For Kubernetes, set pod pids limits so a single pod can't starve the node.

Likely follow-ups

- Why does a pids cgroup limit protect the node?
- How do you act on a system where you cannot fork?

SYMPTOM

The host stops accepting new SSH sessions; even local commands fail with "fork: retry: Resource temporarily unavailable."

INVESTIGATION

```
cat /proc/sys/kernel/pid_max

ps -elf | wc -l    # threads

ps -eo ppid,comm | sort | uniq -c | sort -rn | head    # who's spawning
```

The system hit `pid_max` or a per-user process limit. Once forks fail you often can't even run diagnostics — each tool needs a fork. A runaway loop or fork bomb is multiplying processes faster than they exit.

ROOT CAUSE

Process/PID exhaustion. A user or service is forking unboundedly (a bad loop, a crash-restart storm, or a deliberate fork bomb), starving the global PID space.

FIX / MITIGATION

```
# use an already-running shell / a reserved root session

pkill -STOP -u <baduser>    # freeze the spawner, then kill the tree
```

```
# systemd: systemctl kill --kill-who=all <unit>
```

● PREVENTION

Cap fork-happy services with cgroup TasksMax= (systemd) and set per-user nproc limits in /etc/security/limits.conf so one user or unit can never consume the whole PID table. Keep a reserved root login slot.

Interview tip: Emphasize prevention via TasksMax/nproc — reacting to a fork bomb is hard precisely because you can't fork. Mention freezing (SIGSTOP) the tree before killing.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S76 **A systemd service fails to start after a config change and just shows 'failed'. Walk me through debugging it.**

Linux Host

Spoken answer · ~80 s

When a systemd service shows just failed, I go to the detailed logs. `systemctl status` with full lines shows the exit code and last log lines, but the real error is in the journal: `journalctl` for that unit, the last hundred lines. I check the unit definition with `systemctl cat`, which shows the effective unit including drop-in overrides, and `systemd-analyze verify` to catch syntax and dependency errors. After editing a unit, I must run `daemon-reload`, a frequent miss. Common causes after a config change: a wrong path in `ExecStart`, a missing executable permission, a wrong User or permission on files, a syntax error in the application's own config file, an environment variable or `EnvironmentFile` problem, a port already in use, a missing dependency or ordering issue with `After` and `Requires`, a wrong Type such as simple versus forking so systemd thinks it failed, or a timeout because startup takes longer than `TimeoutStartSec`. Also SELinux or sandboxing options such as `ProtectSystem` blocking file access. I try running the `ExecStart` command manually as the service user to reproduce the error. Then fix, reload, restart, and confirm status and logs. Keep unit changes in version control.

Likely follow-ups

- Why do you need `daemon-reload` after editing a unit?
- How do you reproduce a failing `ExecStart` manually?

● SYMPTOM

After editing a unit, `systemctl start myapp` returns failed. The one-line status isn't enough.

● INVESTIGATION

```
systemctl status myapp -l --no-pager
```

```
journalctl -u myapp -n 100 --no-pager
```

```
systemd-analyze verify myapp.service # syntax/dependency check
```

```
systemctl cat myapp # see the effective unit incl. drop-ins
```

status shows the exit code and last log lines; the real error is in the journal. Common causes: bad ExecStart path, a missing dependency in After=/Requires=, a permission/SELinux denial, or an edit you forgot to daemon-reload.

● ROOT CAUSE

The unit's declared state doesn't match reality — wrong binary path, an unmet ordering dependency, a sandbox directive (ProtectSystem, ReadOnlyPaths) blocking a write, or a stale unit because daemon-reload wasn't run after editing.

● FIX / MITIGATION

```
systemctl daemon-reload # after ANY unit edit
```

```
# fix the ExecStart path / dependency / sandbox directive
```

```
systemctl reset-failed myapp # clear the failed latch before retry
```

● PREVENTION

Validate units in CI with `systemd-analyze verify`, manage them via config management so edits always trigger `daemon-reload`, and prefer drop-in files (`override.conf`) over editing vendor units so upgrades don't clobber changes.

Interview tip: The two things interviewers love here: always `daemon-reload` after edits, and read the journal not just status. Bonus: `systemctl cat` to reveal drop-ins masking your change.

Official docs

- [Kubernetes: Debug Services](#)
- [systemd.service\(5\)](#)
- [Google SRE Book: Effective Troubleshooting](#)

S77 The root filesystem filled up and journald logs are the culprit. How do you contain it without losing observability?

Linux Host

 Spoken answer · ~78 s

When journald fills the root filesystem, I contain it without losing observability. First, `df` and `journalctl disk-usage` confirm journal growth, and I look at what is flooding it, for instance a crash-restart loop emitting megabytes per second. Immediate relief: vacuum the journal by size or age, `journalctl vacuum-size`, which frees space safely, instead of deleting files by hand. But I also stop the source: fix or stop the chatty service, since otherwise the space returns. Root cause is typically no cap on persistent journal storage. The permanent fix is configuring journald limits: `SystemMaxUse` for the total size, `SystemKeepFree` to leave free space, `MaxRetentionSec`, and rate limiting with `RateLimitIntervalSec` and `RateLimitBurst` to throttle floods. To preserve observability, ship logs off the host to a central system like Loki or Elasticsearch, so local retention can be small without losing history, and the logs are not lost when the host dies. Put logs on a separate filesystem or volume so they can never fill the root. Also fix log levels in the noisy service, and alert on disk usage with a predicted fill time, as well as on log volume spikes.

Likely follow-ups

- Why vacuum with `journalctl` rather than delete files by hand?
- Why ship logs off-host?

SYMPTOM

`/` hits 100%; the host gets flaky. `du` points at `/var/log/journal` consuming tens of GB.

INVESTIGATION

```
df -h /
```

```
journalctl --disk-usage
```

```
du -sh /var/log/* | sort -rh | head
```

Persistent journald grew without a cap, or a chatty service is flooding logs (a crash-restart loop emitting megabytes/sec). Unbounded journal storage will happily consume the whole root partition.

ROOT CAUSE

No journal size limit (or one too high), amplified by a noisy unit. The fix has two halves: reclaim space now, and bound it permanently so it can't recur.

FIX / MITIGATION

```
journalctl --vacuum-size=2G # reclaim now
```

```
journalctl --vacuum-time=7d
```

```
# /etc/systemd/journald.conf: SystemMaxUse=2G RuntimeMaxUse=200M
```

```
systemctl restart systemd-journald
```

● PREVENTION

Set `SystemMaxUse` in `journald.conf`, put `/var/log` (or `/var`) on its own partition so a log flood can't take down `/`, and rate-limit or fix the service that's over-logging.

Interview tip: Show the both-halves instinct: reclaim with `--vacuum-*` now, then cap with `SystemMaxUse` forever. Mentioning a separate `/var` partition signals you've been burned before.

Official docs

- [journalctl\(1\)](#)

S78 An NFS-backed mount hangs and any command touching it freezes in D-state. How do you recover?

Linux Host

Spoken answer · ~84 s

A hard-mounted NFS share whose server became unreachable blocks I/O indefinitely by design, so any process touching it sinks into uninterruptible D state, and you can't even kill them. First I confirm: mount options showing hard versus soft, the server, and `dmesg` messages like NFS server not responding. I check if the server is reachable with `rpcinfo` or `showmount`, and look at the network between us. The best recovery is restoring the server or network, after which the processes resume. If the server is gone, options: unmount with a forced lazy unmount, `umount` minus `f` or minus `l`, which detaches the mount from the namespace so new commands stop hanging, though stuck processes may remain until they get their I/O error or the machine reboots. With the `intr` behaviour, in modern kernels, a `SIGKILL` can kill the process waiting on a hard mount in some cases. Prevention: choose mount options deliberately; soft mounts with timeouts avoid hangs but risk data corruption on writes, so use them only for read-mostly data; set `timeo` and `retrans`; use `autofs` so mounts are on demand; avoid putting NFS in critical paths such as monitoring agents; and make the NFS service highly available. Alert on NFS latency.

Likely follow-ups

- Why is a soft mount risky for writes?
- What does a lazy unmount do?

● SYMPTOM

Anything that touches `/mnt/shared` hangs forever — `ls`, `df`, even tab-completion. Processes pile up in D-state.

● INVESTIGATION

```
mount | grep nfs    # note hard vs soft, and the server
```

```
cat /proc/mounts | grep nfs
```

```
dmesg -T | grep -i 'nfs server.*not responding'
```

```
rpcinfo -p <nfs-server>    # is the export reachable?
```

A hard-mounted NFS share whose server became unreachable blocks I/O indefinitely (by design — hard mounts retry forever to avoid data loss). Every process touching it parks in uninterruptible sleep, which you can't even kill -9.

● ROOT CAUSE

The NFS server or the network path to it failed while the client used a hard mount. Hard mounts trade availability for correctness; the price is these hangs when the server disappears.

● FIX / MITIGATION

```
# restore the server / network path – the cleanest fix
```

```
umount -f -l /mnt/shared    # force + lazy detach
```

```
# mount with soft,timeo,retrans for non-critical data to fail fast instead
```

● PREVENTION

Use hard,intr-style resilient options and sane timeo/retrans; for non-critical reads consider soft mounts that error out instead of hanging. Monitor NFS server reachability so the cause is obvious before D-state spreads.

Interview tip: The hard-vs-soft tradeoff is the whole answer: hard = won't corrupt but hangs; soft = fails fast but can lose writes. Note you can't SIGKILL a true D-state process — only fixing the I/O or lazy-unmount frees it.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

 Spoken answer · ~86 s

When a user owns a file with seemingly correct mode bits but gets permission denied, the base mode isn't the whole story. First I check the path, not just the file: to reach a file, the user needs execute permission on every parent directory, so I list each directory with `ls` minus `ld` up the tree. Second, POSIX ACLs can override or restrict the base mode, which I see with `getfacl`; a mask entry can silently limit effective permissions. Third, SELinux or AppArmor: the security context may not allow the process to access the file, shown by `ls` minus `Z` and audit logs; this is common when files are moved instead of copied, or in containers, and the denial appears in the audit log. Fourth, the immutable attribute, set with `chattr` plus `i`, which blocks changes even for root, visible with `lsattr`. Fifth, filesystem-level restrictions: a read-only mount, `noexec` mount option, or an NFS root-squash or ID mapping issue. Also check which user the process actually runs as, group membership that needs a new login to take effect, and capabilities. I test by reproducing as that user with `sudo` minus `u`, and read the audit log. Then fix the specific layer, not by blanket `chmod 777`.

Likely follow-ups

- Why does a parent directory's permissions matter?
- How can an ACL mask restrict permissions?

 SYMPTOM

A user can't read a file. `ls -l` shows they own it and the mode looks right (e.g. `rw-r--r--`), yet access is denied.

 INVESTIGATION

```
ls -ld /path /path/.. /path/../../ # traversal needs +x on EVERY parent dir
```

```
getfacl /path/file # POSIX ACLs can override base mode
```

```
ls -Z /path/file # SELinux context
```

```
lsattr /path/file # immutable (+i) bit?
```

Base mode bits aren't the whole story. Access can be denied by: a parent directory missing the execute (traversal) bit, a POSIX ACL, an SELinux/AppArmor label mismatch, the immutable attribute, or the file living on a read-only/no-exec mount.

 ROOT CAUSE

Something beyond `rx` is governing access — most often a parent directory lacking `+x` (so you can't traverse into it) or an SELinux context denial. The visible mode bits are a red herring.

 FIX / MITIGATION

```
chmod +x <each parent dir> # restore traversal
```

```
setfacl -m u:alice:r /path/file # if ACL-driven
```

```
restorecon -v /path/file # fix SELinux label
```

```
ausearch -m avc -ts recent # confirm an SELinux AVC denial
```

● PREVENTION

Standardize directory traversal permissions in your provisioning, keep SELinux in enforcing with correct contexts (don't disable it), and use `getfacl/ls -Z` as routine checks so the "hidden" permission layers are visible.

Interview tip: List the layers fast: parent-dir +x, ACLs, SELinux label, immutable bit, mount options. The parent-directory traversal trap is the single most common gotcha — lead with it.

Official docs

- [RHEL 9 documentation](#)

S80 The system is constantly swapping and everything feels sluggish even though there's some free RAM. How do you diagnose?

Linux Host

Spoken answer · ~83 s

Heavy swap use with some free RAM and sluggishness means the working set exceeds memory, and hot pages are bouncing to disk. I check `vmstat`: nonzero `si` and `so` columns, swap in and swap out, mean active swapping, which is what hurts; swap that is merely used but not actively moving is usually fine. Disk is thousands of times slower than RAM, so latency collapses. Then I find who is using memory: `ps` sorted by resident size, `smem` for per-process swap, and check for a leak by trending. I look at `swappiness`, which controls how eagerly the kernel swaps; a high value on a database server can swap out hot data unnecessarily, and lowering it helps. Check for memory pressure from page cache versus application memory, and for cgroup limits causing local swapping even when the host has free memory. Also transparent huge pages and NUMA imbalance can cause odd behaviour. Fixes: reduce memory demand by fixing leaks, tuning caches and heap sizes, and limiting workloads; add RAM; lower `swappiness`; use `zram` or a fast swap device as a buffer; and set memory limits so one process can't push others into swap. Monitor PSI memory pressure and swap activity.

Likely follow-ups

- Why is swap used different from swap active?
- What does lowering `swappiness` do?

● SYMPTOM

Latency is awful and the disk light is always on. free shows swap heavily used and growing, though a little RAM appears free.

● INVESTIGATION

```
free -m
```

```
vmstat 1 5 # si/so columns = swap in/out per sec
```

```
cat /proc/sys/vm/swappiness
```

```
smem -s swap -r | head # who's swapped out
```

Active swap-in/swap-out (si/so nonzero) means the working set exceeds RAM, so the kernel pages hot memory to disk — each fault now costs a disk seek. High swappiness can also push file-backed pages out too aggressively.

● ROOT CAUSE

Memory pressure: the sum of active working sets is larger than physical RAM, so the kernel thrashes pages between RAM and swap. The “some free RAM” is misleading — it's reclaimable cache, not headroom for the working set.

● FIX / MITIGATION

```
# reduce memory demand: cap/limit the heaviest consumers
```

```
sysctl vm.swappiness=10 # favor keeping anon pages resident (tune, don't cargo-cult)
```

```
# add RAM or move a service off the host – the real fix if working set > RAM
```

● PREVENTION

Right-size memory to the working set, set per-service memory limits so one process can't force global thrash, and alert on si/so rates (not just swap-used) since steady swap usage can be benign while active swapping is not.

Interview tip: Distinguish *swap used* (often fine) from *swapping* (the si/so activity that kills latency). Tuning swappiness is a knob, not a cure — the cure is fitting the working set in RAM.

Official docs

- [Linux kernel docs: /proc/sys/vm](#)

 Spoken answer · ~84 s

If a server drops to an emergency shell after an fstab edit, a bad entry is failing the mount, and by default a failed required mount stops the boot. At the emergency shell, usually with root password or console access, I read the journal for failed units and dependencies, and check systemctl failed, then look at fstab and compare with blkid to see whether the UUIDs exist; typical causes are a wrong UUID, a removed device, a typo, or bad options. The root filesystem is often mounted read-only in emergency mode, so I remount it read-write, then edit fstab: correct the entry, or comment it out. Then I test with mount minus a, which validates the file without rebooting, and reboot. Prevention: always test fstab edits with mount minus a or findmnt verify before rebooting; use UUIDs or labels instead of device names that can change; add the nofail option, plus a short timeout, for non-critical mounts like data disks or network shares, so a missing one doesn't block boot; and keep a console or out-of-band access path. For cloud instances, if there's no console, detach the root volume and fix it from another instance. Managing mounts in configuration management helps.

Likely follow-ups

- What does the nofail option change?
- How do you test an fstab change safely before rebooting?

 SYMPTOM

After a storage change the host won't boot — it lands in an emergency/maintenance shell. The console mentions a failed mount.

 INVESTIGATION

```
journalctl -xb | grep -i 'failed\|dependency'
```

```
systemctl --failed
```

```
cat /etc/fstab # the suspect line
```

```
blkid # do the UUIDs in fstab actually exist?
```

A bad /etc/fstab entry (wrong UUID, removed device, or bad options) fails at boot. By default a failed required mount blocks local-fs.target, so systemd stops and drops to emergency mode rather than booting half-configured.

 ROOT CAUSE

An fstab line references a device/UUID that no longer matches reality. Because the mount is marked required, its failure halts the boot dependency chain.

 FIX / MITIGATION

```
mount -o remount,rw / # make root writable in the emergency shell
```

```
# fix or comment the bad fstab line; correct the UUID from blkid
```

```
# add 'nofail' (and optionally x-systemd.device-timeout=) to non-critical mounts
```

```
reboot
```

● PREVENTION

Use `nofail` on non-essential mounts so a missing disk doesn't block boot, reference filesystems by UUID/LABEL (stable across device renaming), and validate `fstab` changes with `mount -a` before rebooting.

Interview tip: The lifesaver line is `mount -o remount,rw /` to edit `fstab` from the emergency shell, plus `nofail` as the preventive option. Always test with `mount -a` before you reboot — say that and you sound battle-tested.

Official docs

- [Google SRE Book: Emergency Response](#)

S82 Under heavy connection load, new connections start failing and `dmesg` shows '`nf_conntrack: table full, dropping packet`'. What now?

Linux Host

Spoken answer · ~78 s

The message `nf_conntrack table full, dropping packet` means the kernel's connection-tracking table has reached its maximum, so new connections are dropped, which looks like random timeouts under load. I confirm by comparing the current count with the maximum in `sysctl` and listing entries with `conntrack`. Then I understand why it's full: genuinely many concurrent connections, a traffic spike or attack, long timeouts keeping dead connections around, especially the established TCP timeout, which is five days by default, or short-lived connection churn like DNS over UDP, or leaked connections. Fixes: raise `nf_conntrack_max` and the hash table size, remembering that each entry uses memory; lower the timeouts, such as the established and time-wait timeouts, so stale entries expire sooner; and reduce the load by reusing connections with `keepalive` and pooling. Where connection tracking isn't needed, exempt traffic using the raw table with a `NOTRACK` rule, such as for high-volume stateless traffic on a gateway. In Kubernetes, `kube-proxy` and DNS can stress `conntrack`, so node-local DNS caching helps. Monitor the count as a percentage of the max and alert at around eighty percent, and size nodes for the expected connection count.

Likely follow-ups

- Why does a five-day established timeout fill the table?
- When would you bypass `conntrack` with `NOTRACK`?

● SYMPTOM

A busy gateway/node starts dropping new connections under load. dmesg shows nf_conntrack: table full, dropping packet.

● INVESTIGATION

```
dmesg -T | grep conntrack
```

```
sysctl net.netfilter.nf_conntrack_count net.netfilter.nf_conntrack_max
```

```
cat /proc/sys/net/netfilter/nf_conntrack_max
```

```
conntrack -L | wc -l
```

The kernel's connection-tracking table is full. Once conntrack_count hits conntrack_max, new flows are dropped. High connection churn (many short-lived connections) plus a default-sized table, or entries lingering in TIME_WAIT/long timeouts, fills it.

● ROOT CAUSE

Conntrack table exhaustion. The host tracks more concurrent flows than the table allows — common on NAT gateways, busy load balancers, and Kubernetes nodes with high pod-to-service churn.

● FIX / MITIGATION

```
sysctl -w net.netfilter.nf_conntrack_max=262144 # raise table size (watch RAM)
```

```
sysctl -w net.netfilter.nf_conntrack_tcp_timeout_time_wait=30 # reclaim faster
```

```
# or bypass tracking for high-volume flows via iptables -t raw ... -j NOTRACK
```

● PREVENTION

Size conntrack_max to expected concurrency (and the hashsize accordingly), tune timeouts down for short-lived flows, and on Kubernetes watch this on nodes — it's a frequent silent cause of intermittent connection failures under load.

Interview tip: Naming nf_conntrack table-full as a cause of *intermittent* connection drops on NAT gateways and K8s nodes is a strong signal. Mention sizing max + hashsize together and NOTRACK for high-volume flows.

Official docs

- [Netfilter documentation](#)

S83 A test suite passes locally but fails intermittently in CI ('flaky tests'). How do you stabilize the pipeline?

CI/CD & Deploy

🗣 Spoken answer · ~83 s

Flaky tests erode trust in the pipeline, and retry-until-green hides real problems, so I treat flakiness as a defect to fix. First, measure: track which tests fail intermittently across runs, using CI analytics, so I know the real offenders instead of guessing. Re-run a failing job in isolation to confirm non-determinism, and run tests in randomised order and in parallel locally to expose order dependence. Then look for the usual causes: shared state between tests, such as fixed ports, temp files, or database rows not cleaned up; time dependence, with sleeps, real clocks, or time zones; race conditions and async waits without proper synchronisation; reliance on external services or the network; resource limits or slowness on CI runners; and non-deterministic data like random seeds or ordering of maps. Fixes are making tests hermetic, isolating state per test, injecting clocks, using polling with timeouts instead of sleeps, mocking external dependencies, and seeding randomness. Short term, quarantine known flaky tests into a separate non-blocking job, with an owner and deadline, so they don't block merges but don't get forgotten. Auto-retry can be used sparingly with reporting. Track the flake rate as a metric and treat a new flaky test as a bug.

Likely follow-ups

- Why is blindly retrying failed tests harmful?
- How does quarantining differ from deleting a flaky test?

SYMPTOM

The same commit sometimes passes, sometimes fails in CI. Engineers have learned to just hit "retry," and trust in the pipeline is eroding.

INVESTIGATION

- # re-run the failing job in isolation to confirm non-determinism
- # diff the failure set across runs – same tests or random?
- # check for shared state: ports, temp files, fixed timestamps, DB rows
- # run tests in randomized order locally to reproduce

Flaky tests usually stem from non-determinism: timing/sleep races, order-dependence via shared state, real network or clock dependencies, or unfixed random seeds. CI exposes them because it's slower, parallel, and clean-room compared to a warm laptop.

ROOT CAUSE

Tests are coupled to non-deterministic conditions — timing, ordering, external services, or shared fixtures — so they pass when conditions happen to line up and fail otherwise. The retry culture hides real signal.

● FIX / MITIGATION

```
# quarantine flaky tests into a separate, non-blocking lane
```

```
# fix root cause: inject the clock, mock the network, isolate state per test, fix seeds
```

```
# add retry-with-logging to surface the failure mode, not just paper over it
```

● PREVENTION

Track per-test flakiness rate and auto-quarantine offenders out of the blocking path, mandate hermetic tests (no real network/clock/shared DB), and treat a flaky test as a bug ticket — not a “retry and move on.” A pipeline you don't trust is worse than a slow one.

Interview tip: Senior framing: flakiness is a *trust* problem, not a test problem. Talk about quarantine-then-fix and hermeticity, and call out that retry culture masks real regressions.

Official docs

- [Testcontainers / test pyramid \(Martin Fowler\)](#)
- [Google Testing Blog: Flaky tests at Google](#)

S84 A deployment is stuck — the pipeline reports 'deploying' for 20 minutes with no progress. How do you triage?

CI/CD & Deploy

Spoken answer · ~82 s

When a deploy has said deploying for twenty minutes with no progress, I find out where it's stuck. First, ask the platform directly. In Kubernetes, rollout status with a timeout shows whether the rollout is progressing; get pods shows whether new pods are Pending, in CrashLoopBackOff, or running but not Ready, and describe deployment shows conditions like ProgressDeadlineExceeded. The cause is typically one of: new pods can't be scheduled because of insufficient resources or quota; image pull failures; the readiness probe never passing because of a bug, a missing config or secret, or a dependency unavailable; a PodDisruptionBudget or maxUnavailable setting blocking the rollout; or a pre-deploy hook or migration job that hangs. If the problem is the pipeline tool, I check the deploy tool's own logs, such as Argo, Spinnaker, or the CD runner, for a lost connection, expired credentials, or a lock. Users are still on the old version, so impact is limited; I decide whether to abort and roll back or fix forward. If unclear after a few minutes, I roll back to restore a known state, then investigate. Afterwards, add sensible timeouts so a stuck deploy fails fast and alerts, instead of hanging.

Likely follow-ups

- What does ProgressDeadlineExceeded mean?
- Why might a PodDisruptionBudget stall a rollout?

● SYMPTOM

A deploy job hangs in the “deploying” phase far past its usual time, neither succeeding nor failing. Users are still on the old version.

● INVESTIGATION

```
kubectl rollout status deploy/app --timeout=10s # if k8s
```

```
kubectl get pods -l app=app # new pods Pending/CrashLoop/not Ready?
```

```
kubectl describe deploy/app | sed -n '/Conditions/, $p'
```

```
# check the deploy tool's own logs (Argo/Spinnaker/CD runner)
```

A rolling deploy waits for new replicas to become Ready. If they never pass readiness (bad image, failing health check, unschedulable, or a missing dependency), the rollout stalls indefinitely rather than failing fast — by design it won't tear down healthy old pods.

● ROOT CAUSE

New instances aren't reaching a healthy/Ready state, so the orchestrator holds the rollout. The old version stays up (good for users) but the deploy never completes. The hang is a symptom; the new pods' failure is the cause.

● FIX / MITIGATION

```
# decide: roll forward (fix) or roll back
```

```
kubectl rollout undo deploy/app # fast mitigation
```

```
# fix the readiness probe / image / dependency, then redeploy
```

```
# set progressDeadlineSeconds so a stall fails the rollout instead of hanging
```

● PREVENTION

Set a progress deadline so stuck rollouts fail loudly instead of hanging, make readiness probes accurate (probe a real dependency, not just process-up), and gate deploys behind smoke checks so a bad image is caught before it stalls a rollout.

Interview tip: Lead with “new pods aren't Ready → rollout waits forever.” Mention `progressDeadlineSeconds` to convert a silent hang into a fast failure, and the roll-forward-vs-back decision.

Official docs

- [Kubernetes: Deployments](#)

S85 **You discover a plaintext secret (API key) was committed to the repo and is now in CI logs. What's your response?** CI/CD & Deploy

 Spoken answer · ~83 s

A plaintext API key in the repo and in CI logs is a compromised secret, and my first step is to revoke and rotate it, not to clean up history. Removing the commit doesn't help, because the key is already in git history, forks, clones, CI logs, caches, and artifacts. So step one: rotate the key at the provider, issue a new one, and revoke the old. Step two: determine scope. What can the key access, and how long was it exposed? I review the provider's audit logs and usage for unfamiliar activity, and if I see any, I escalate as a security incident. Step three: clean up. Search git history, CI logs, build caches, and artifacts for the value, delete or purge the logs where possible, and optionally rewrite history, accepting that it's secondary to rotation. Step four: put the new secret in a secrets manager or CI secret store and inject it at runtime, never in code. Step five: prevent a repeat. Add pre-commit and CI secret scanning with a tool like gitleaks, enable push protection, mask secrets in CI output, use short-lived credentials or workload identity, and restrict key scopes. Then a short blameless review.

Likely follow-ups

- Why is rotation more important than rewriting git history?
- How would you stop secrets appearing in CI logs?

SYMPTOM

A code review (or a scanner alert) reveals an API key hard-coded in a commit. It has already run through CI, so it may also sit in pipeline logs and artifacts.

INVESTIGATION

```
git log -p -S '<key-fragment>' # find when it entered history
```

```
# search CI logs/artifacts/caches for the same value
```

```
# determine the key's scope and what it can access
```

Once a secret is in git history (and CI logs), it must be treated as compromised — history is distributed and logs are cached. Deleting the line in a new commit does *not* remove it from history or from anyone's clone.

ROOT CAUSE

A secret leaked into version control and propagated to CI. The exposure window started at commit time, not discovery time, so the only safe assumption is that it's known to attackers.

FIX / MITIGATION

```
# 1. ROTATE/revoke the key first – this is the only thing that truly mitigates
```

2. purge from history (git filter-repo / BFG) and force-push, coordinate clones

3. scrub CI logs/artifact caches; check access logs for misuse

● PREVENTION

Rotation comes first, always. Then add pre-commit and CI secret scanning (gitleaks/trufflehog) to block leaks at the source, and move secrets to a manager (Vault, cloud secrets) injected at runtime so they never touch the repo.

Interview tip: The interviewer wants to hear **rotate first**. Scrubbing history matters but is secondary — a leaked key is compromised the moment it's pushed. Then talk prevention via scanning + secret managers.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S86 Terraform apply fails with a state lock error, and you suspect drift between state and real infra. How do you proceed safely?

CI/CD & Deploy

Spoken answer · ~83 s

A state lock error combined with suspected drift calls for caution, because forcing an apply could overwrite someone's work or real infrastructure. First I identify who holds the lock, from the backend's lock info such as the DynamoDB lock entry, and check whether a teammate's run is genuinely in progress or crashed and left a stale lock. I only force-unlock after confirming nobody is applying. Then I deal with drift: run a plan, reading it carefully, or a refresh-only plan to see what differs between state and reality, because a manual console change made last week will show up as a diff. For each difference, I decide: should the code win, in which case apply will revert the manual change, or should the manual change be kept, in which case I update the code to match, or import the resource? I never blindly apply a plan with unexpected destroys. I take a state backup first, and make sure everything goes through version control and review.

Prevention: restrict console write access, enable drift detection on a schedule, use remote state with locking, and make Terraform the only path for changes, with an emergency process that backports manual fixes to code.

Likely follow-ups

- When is it safe to force-unlock a state lock?
- What does a refresh-only plan show you?

● SYMPTOM

terraform apply errors with a state-lock message, and a teammate mentions someone changed a resource manually in the console last week.

● INVESTIGATION

```
terraform plan    # read carefully: what does it want to change & why?
```

```
# identify the lock holder (DynamoDB lock table / backend lock info)
```

```
terraform state list    # confirm what's tracked
```

```
terraform refresh / plan -refresh-only    # surface drift vs state
```

Two issues: a stale lock (a previous run died without releasing it) blocks apply, and manual console changes created *drift* — real infra no longer matches state. Blindly applying could revert the manual change or, worse, destroy/recreate resources.

● ROOT CAUSE

The lock is a safety mechanism that outlived its run; the drift is human out-of-band change. The danger is applying without understanding what the plan will do to the drifted resource.

● FIX / MITIGATION

```
terraform force-unlock <LOCK_ID>    # only after confirming no live run holds it
```

```
# reconcile drift: terraform import, or update code to match, or plan-revert intentionally
```

```
terraform apply -target=...    # narrow blast radius while reconciling
```

● PREVENTION

Enforce remote state with locking, ban manual console changes (or auto-detect drift on a schedule), require PR-reviewed plans before apply, and use `prevent_destroy` on stateful resources so a surprise plan can't nuke a database.

Interview tip: Two-part answer scores well: never force-unlock without confirming the holder is dead, and never apply over drift without reading the plan. Mention drift detection + `prevent_destroy` as prevention.

Official docs

- [Terraform: State](#)
- [Terraform: State locking](#)
- [Terraform CLI: plan](#)

S87 ArgoCD shows an app as 'OutOfSync' and someone keeps hotfixing the cluster directly. How do you handle GitOps drift?

CI/CD & Deploy

 Spoken answer · ~78 s

In GitOps, Git is the single source of truth, so a `kubectl` edit to hotfix production creates drift that Argo flags as `OutOfSync`. First, I look at exactly what differs with an `app diff`, and check what `auto-sync` and `self-heal` are set to, since that determines whether Argo will silently revert the hotfix, which can surprise the engineer mid-incident, or just report it. Handling it well has two parts. Short term, I don't punish the hotfix, because during an incident speed matters. I make sure the fix is captured: open a pull request that applies the same change in Git, so the source of truth matches reality, and then `sync`. Long term, I remove the reason people bypass Git: make the normal path fast, with an expedited pull request and review process, a well-practised rollback by reverting a commit, and a documented break-glass procedure that is audited and followed by an immediate backport. Restrict direct write access to production with RBAC, so manual edits are exceptional, and enable `self-heal` with alerts on drift. Track drift events as a signal. The cultural point is that Git is the interface.

Likely follow-ups

- What does `self-heal` do and why can it surprise people during an incident?
- What is a break-glass procedure?

SYMPTOM

Your GitOps tool reports an application `OutOfSync`. Investigation shows an engineer ran `kubectl edit` to hotfix prod during an incident.

INVESTIGATION

```
argocd app diff <app> # exactly what differs from Git
```

```
kubectl get <res> -o yaml | grep -A2 managedFields # who last changed it
```

```
# is auto-sync on? is self-heal on? – determines current behavior
```

GitOps treats Git as the single source of truth. A live `kubectl edit` makes the cluster diverge from the repo. With `self-heal` on, Argo will revert the hotfix (possibly mid-incident); with it off, the drift lingers and the next `sync` may clobber it unpredictably.

ROOT CAUSE

Out-of-band imperative change broke the declarative contract. The cluster and the repo now disagree, and which one “wins” depends on sync settings — a recipe for surprise reversions.

FIX / MITIGATION

```
# decide source of truth: codify the hotfix into Git (preferred), then sync
```

```
argocd app sync <app> # reconcile to Git once Git is correct
```

```
# if the live change was wrong, let self-heal revert it
```

● PREVENTION

Make the cluster read-only to humans (RBAC), so changes flow only through Git PRs; allow a documented break-glass path for emergencies that *requires* back-filling the change into Git afterward; and alert on OutOfSync so drift is caught in minutes.

Interview tip: The principle to state: in GitOps, fix forward by changing *Git*, not the cluster. Acknowledge the legitimate break-glass case during incidents — but with a mandatory codify-it-back step.

Official docs

- [Terraform CLI: plan](#)
- [OpenGitOps principles](#)
- [Argo CD documentation](#)

S88 A database migration baked into the deploy pipeline failed halfway and left the schema inconsistent. How do you recover?

CI/CD & Deploy

Spoken answer · ~84 s

A migration that failed halfway leaves the schema in an unknown state, so I first establish exactly what happened. I compare the migration tool's recorded state, the schema migrations table or Flyway history, with the actual schema, and work out which statements committed and which didn't. Whether the migration was transactional depends on the engine: Postgres supports transactional DDL, so a failure normally rolls the whole migration back, while MySQL commits DDL implicitly, so partial application is common. Meanwhile I stabilise the application: if the code can't work against the half-migrated schema, roll back the application version to one compatible with the old schema, or put the service into maintenance. Recovery options: if the changes are small and well understood, write a corrective script to finish or reverse the remaining steps; otherwise restore from a backup or snapshot taken before the migration, accepting data written since. Mark the migration tool state correctly so it doesn't retry blindly. Prevention: take a backup before migrations, split migrations into small steps, make them idempotent, follow expand and contract so schema changes are backward compatible with the previous app version, test them against production-sized data, and run them as a separate gated step before the deploy.

Likely follow-ups

- Why does transactional DDL change your recovery options?
- What is the expand and contract pattern?

● SYMPTOM

A deploy ran a schema migration that errored partway. Some statements applied, some didn't, and the app is now half-broken against an inconsistent schema.

● INVESTIGATION

```
# check migration tool state (schema_migrations / flyway history) vs actual schema
```

```
# was the migration transactional? (DDL transactionality is engine-dependent)
```

```
# determine: which statements committed, which rolled back
```

The migration wasn't atomic — either the engine doesn't wrap DDL in a transaction, or it spanned multiple statements/transactions. A partial apply leaves the schema in a state neither the old nor new code expects.

● ROOT CAUSE

Non-atomic migration plus a mid-flight failure. The migration tool's recorded state and the real schema have diverged, so a naive re-run may double-apply or skip steps.

● FIX / MITIGATION

```
# reconcile the migration ledger with reality before re-running
```

```
# apply the remaining steps manually if safe, or restore-and-replay from backup
```

```
# verify app works against the corrected schema in a lower env first
```

● PREVENTION

Make migrations atomic where possible and idempotent always; adopt expand/contract (backward-compatible) migrations decoupled from the deploy so a code rollback never strands the schema; test migrations against a production-like snapshot in CI.

Interview tip: The senior move is **expand/contract**: deploy schema changes that are backward-compatible and separate from app deploys, so neither can break the other. Mention DDL transactionality varies by engine (Postgres yes, MySQL largely no).

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S89 A deploy went out, errors spiked, and you need to roll back — but the previous artifact is gone from the registry. What do you do?

CI/CD & Deploy

Spoken answer · ~81 s

If the previous artifact has been pruned, rollback by redeploying the old image is impossible, so I find another route to a known-good state. First I check whether the old image is cached anywhere: nodes still holding it, which I check with `crictl images` or by describing pods, another registry or mirror, or a CI cache. If a node has it, I can retag and push it back. Second, rebuild the previous commit deterministically: with pinned dependencies, a lockfile, and a pinned base image, rebuilding gives an equivalent artifact; without them, the rebuild may differ, which is a risk to note. Third, revert the bad change in Git and run the normal pipeline, a roll-forward revert, which is often the cleanest. Fourth, use feature flags to switch off the bad behaviour without redeploying. Meanwhile mitigate the impact, for instance by shedding load or scaling. Afterwards, fix the root cause: retention policy for the registry should never delete anything that is currently deployed or recent releases, keep at least the last N production releases and anything tagged for production, pin images by digest, make builds reproducible, and test rollback regularly as a part of release readiness.

Likely follow-ups

- How do reproducible builds help in this situation?
- What retention policy should a registry have?

SYMPTOM

A bad release spikes errors. You reach for rollback and discover the prior image was pruned from the registry by an aggressive cleanup policy.

INVESTIGATION

```
# confirm current bad version & error signature
```

```
# can you rebuild the previous commit deterministically?
```

```
# is the prior image cached on any node? (kubectl describe / crictl images)
```

Rollback assumes the previous artifact still exists. An over-eager retention policy deleted it, so the fast path is gone. You're now choosing between rebuilding from the previous commit (slow, and only safe if builds are reproducible) and rolling forward with a fix.

ROOT CAUSE

Artifact retention was tuned for storage cost without protecting the rollback target. The incident is the deploy; the rollback gap is a process failure that just made the incident longer.

FIX / MITIGATION

```
# fastest safe path now: roll forward with a targeted fix or feature flag
```

```
# if a node still has the image cached, pin a pod to it as a stopgap
```

```
# rebuild prev commit only if builds are byte-reproducible & fast
```

● PREVENTION

Keep the last N known-good artifacts immutable and exempt from cleanup, make builds reproducible, and prefer feature flags so risky changes can be disabled without a redeploy at all — the fastest rollback is a config toggle.

Interview tip: Two lessons land here: protect known-good artifacts from retention pruning, and that feature flags often beat redeploy-rollback for speed. Show you'd roll *forward* when rolling back isn't available.

Official docs

- [AWS Builders' Library: Ensuring rollback safety during deployments](#)

S90 Builds suddenly produce broken binaries and you suspect a poisoned/stale build cache. How do you isolate it?

CI/CD & Deploy

Spoken answer · ~81 s

When builds suddenly ship broken output with no code change, I suspect something in the inputs, such as a stale or poisoned cache. The fastest test is to reproduce with the cache disabled: docker build with no-cache, or clearing the CI dependency cache. If the build goes green, the cache is the cause. Then I find which part: diff the resolved dependency tree against a known-good lockfile or a previous successful build's dependency list. Check for an upstream dependency that changed under a floating version range or a moving tag, such as latest, or a base image that was updated; a republished package is another possibility. Also check for cache key collisions, where a key doesn't include everything that affects the output, such as the lockfile hash, tool version, or build arguments, so a stale layer gets reused. In shared caches, one bad build can poison others. Remediation: invalidate the cache, fix the cache keys, pin dependencies with lockfiles and digests, and verify checksums. For supply-chain concerns, consider whether the dependency was tampered with, and compare hashes. Prevention: reproducible builds, periodic clean builds in CI, and recording build provenance so I can compare two builds.

Likely follow-ups

- Why should cache keys include the lockfile hash?
- How can you tell a cache problem from a dependency change?

● SYMPTOM

Builds that worked yesterday now ship subtly broken output, with no relevant code change. A shared layer/dependency cache is suspected.

● INVESTIGATION

```
# reproduce with cache disabled – does the build go green?
```

```
docker build --no-cache ... # or clear the CI dependency cache
```

```
# diff the resolved dependency tree vs a known-good lockfile
```

```
# check for an upstream dependency that changed under a floating version
```

A cache served a stale or corrupted layer, or a floating dependency version (no pinning) resolved to a new, broken release that then got cached. Because the cache is shared, every build inherits the poison until it's invalidated.

● ROOT CAUSE

Either cache corruption/staleness or unpinned dependencies pulling a bad upstream — both let a non-reproducible input slip into builds and persist via caching.

● FIX / MITIGATION

```
# bust the cache: invalidate keys / docker build --no-cache
```

```
# pin exact versions via lockfiles; commit them
```

```
# rebuild from the verified lockfile to confirm green
```

● PREVENTION

Pin every dependency with committed lockfiles, key caches on content hashes (so changed inputs auto-invalidate), and verify supply-chain integrity (checksums/signatures, SBOMs). Reproducible builds make “stale cache” diagnosable instead of mysterious.

Interview tip: Tie it to reproducibility: pinned lockfiles + content-addressed cache keys mean a cache can't silently poison you. Mention supply-chain verification (checksums/SBOM) for the senior-plus signal.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S91 A canary deploy was supposed to auto-rollback on errors but didn't, and the bad version reached 100%. Why, and how do you fix it?

CI/CD & Deploy

 Spoken answer · ~83 s

When a canary was supposed to auto-rollback but the bad version reached a hundred percent, the safety mechanism itself failed, and I debug the analysis. Auto-rollback depends entirely on the analysis, so the questions are: what metric and threshold was it watching, and was that the right one? A canary that watches CPU won't catch a rise in errors. Did the canary receive enough traffic and time to produce a statistically meaningful signal? A one percent slice for two minutes may see no errors at all. Did the metric query actually return data, or did it silently return no data, which some tools treat as success? That is a very common failure: a label change or query typo makes the check vacuously pass. Were the thresholds too lax, or only error rate checked and not latency or business metrics? Also, were there manual overrides or a skipped step in the pipeline? Fixes: make no-data a failure, use multiple SLO-based metrics, require minimum traffic before promotion, compare canary against baseline, not against an absolute number, and extend soak times for risky changes. Test the rollback path regularly by deliberately deploying a bad canary. Then roll back now and write the postmortem.

Likely follow-ups

- Why should missing metric data fail the canary rather than pass it?
- How do you compare canary to baseline?

SYMPTOM

A progressive/canary rollout should have halted on elevated errors but instead promoted the bad build to full traffic. Now everyone's affected.

INVESTIGATION

what metric/threshold was the canary analysis watching?

did the canary get enough traffic/time to produce signal?

did the metric query actually return data, or silently no-data?

Canary auto-rollback depends entirely on its analysis query. Common failures: the threshold was too lenient, the canary got too little traffic to reach statistical signal, the metric query returned *no data* (treated as "pass"), or the error type wasn't the one being measured.

ROOT CAUSE

The automated guardrail measured the wrong thing, lacked signal, or fail-opened on missing data — so it waved through a release it was supposed to stop. The deploy mechanism worked; the *decision* input was broken.

FIX / MITIGATION

```
kubectl rollout undo ... / promote last-good # mitigate now
```

```
# fix the analysis: correct metric, sane threshold, minimum-sample gate
```

```
# make no-data FAIL the canary, not pass it
```

● PREVENTION

Treat “no data” as failure (fail-closed), require a minimum traffic/sample before a pass verdict, validate analysis queries against a known-bad replay, and watch user-facing SLIs in the canary — not just process health.

Interview tip: The killer insight: most canary failures are **no-data treated as success**. Say you'd make the guardrail fail-closed and require minimum sample size — that's the senior reliability instinct.

Official docs

- [Kubernetes: Deployments](#)
- [AWS Builders' Library: Ensuring rollback safety during deployments](#)
- [Google SRE Workbook: Canarying Releases](#)

S92 A change works in staging but breaks in production, and the diff is 'just config'. How do you find the discrepancy?

CI/CD & Deploy

Spoken answer · ~84 s

When a change works in staging but breaks in production and the diff is just config, I treat it as an environment parity problem. First I diff the effective configuration, not just the files: environment variables, feature flags, secrets, scaling settings, and what the running process actually sees. Then I diff the environments themselves: software and dependency versions, data volume and shape, network egress rules, IAM permissions, TLS settings, and resource limits. Production often has far more data, so a query that is fast on staging data is slow in production. Does production hit a dependency that staging mocks or never exercises, such as a real payment provider, a real queue, or a third-party API with rate limits? Are there differences in scale, concurrency, or topology, like multiple replicas versus one? Config drift between environments is also common because someone changed production manually. I look at the logs and errors in production for the actual failure signature, which usually points to the difference. Then I reproduce it in staging by making it match production for that dimension. Long term, reduce differences: infrastructure as code for all environments, config in version control, production-like data volumes in a pre-production environment, and canary releases.

Likely follow-ups

- Which differences between staging and production most often cause surprises?
- How would you make staging more production-like?

● SYMPTOM

Identical code passed staging and failed in prod. The teams insist “nothing changed but config.”

● INVESTIGATION

```
# diff effective config prod vs staging (env vars, flags, secrets, scaling)
```

```
# diff the environments themselves: versions, data volume, network egress, IAM
```

```
# does prod hit a dependency staging mocks or never exercises?
```

“Works in staging” failures come from environment parity gaps: prod has real data volume, real traffic concurrency, stricter IAM/network policy, different feature-flag values, or a dependency staging never truly exercises. The code is innocent; the environment delta is the bug.

● ROOT CAUSE

Configuration and environment drift between staging and prod. The deployable is the same, but the surrounding context (data scale, permissions, dependencies, flags) differs in a way the change happened to depend on.

● FIX / MITIGATION

```
# reconcile the specific differing config/permission/flag
```

```
# reproduce in a prod-like env (real data scale, real IAM) to confirm
```

```
# add the missing dependency exercise to staging
```

● PREVENTION

Drive staging $\leftrightarrow$ prod parity (same topology, representative data, same policy model), manage config declaratively and diffable per environment, and shift dependencies left so staging actually exercises what prod does. Config is code — review and version it.

Interview tip: Reframe “just config” as a first-class change: config differences are the #1 staging-vs-prod gap. Name the usual suspects (data scale, IAM, flags, untested dependency) to show you’ve debugged this before.

Official docs

- [AKS: Best practices](#)

S93 An artifact was promoted from staging to prod but behaves differently — you suspect it's not actually the same artifact. How do you verify?

CI/CD & Deploy

 Spoken answer · ~80 s

To verify that production runs the same artifact staging validated, I compare image digests, not tags, because tags are mutable and can be overwritten or rebuilt. In staging and production I read the actual running digest, for example from the pod's `imageID`, and compare the sha256 values. If they match, the artifact is identical and the difference lies in configuration or environment. If they differ, I find out why: the tag was rebuilt or overwritten between staging and production, production pulls from a different registry or repository, or the pipeline rebuilt from source instead of promoting the same artifact. That last one is a classic anti-pattern, because a rebuild can pick up different dependencies or base images. The principle is build once, promote the same immutable artifact through environments, injecting environment-specific config at deploy time. To enforce it: deploy by digest, make tags immutable in the registry, sign images and verify signatures at admission, record provenance, and have the pipeline fail if the digest being deployed isn't the one that passed staging tests. Also confirm that nothing else in the deployment differs, such as sidecars or init containers that come from different images.

Likely follow-ups

- Why are image tags unreliable as identifiers?
- What is the build-once, promote-everywhere principle?

SYMPTOM

The promoted release misbehaves in prod. You want to rule out that prod is somehow running a *different* build than what staging validated.

INVESTIGATION

```
# compare image digests (sha256), not tags – tags are mutable
```

```
kubectl get pod -o jsonpath='{..imageID}' # actual running digest
```

```
# does prod pull from the same registry/repo as staging?
```

```
# was the tag rebuilt/overwritten between staging and prod pulls?
```

Tags like `:latest` or `:v1.2` are mutable pointers — a rebuild can move a tag so staging and prod, both “on v1.2,” run different bytes. Only the content digest (sha256) proves identity. If digests differ, you never actually promoted the validated artifact.

ROOT CAUSE

Promotion was done by mutable tag rather than immutable digest, so a rebuild silently substituted a different artifact. “Same version” was an illusion.

FIX / MITIGATION

```
# pin and promote by digest, not tag
```

```
# re-pull the exact staged digest in prod
```

```
# block tag overwrites in the registry (immutable tags)
```

● PREVENTION

Promote artifacts by immutable digest, enable tag immutability in the registry, and sign artifacts (cosign) so prod can verify it's running exactly the bytes that passed staging. "Build once, promote the same bytes" is the rule.

Interview tip: The crisp answer: **compare digests, not tags**, and promote by digest. "Build once, promote the same artifact across envs" is the principle interviewers want to hear.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S94 Your pipeline takes 45 minutes and developers are batching changes to avoid it, increasing risk. How do you approach this?

CI/CD & Deploy

Spoken answer · ~80 s

A forty-five minute pipeline changes behaviour for the worse: engineers batch changes to avoid the wait, so each deploy is larger and riskier, which is the opposite of what CI wants. I start by profiling: which stages dominate wall-clock time, what runs serially that could run in parallel, and what is rebuilt that could be cached? Then I apply the standard levers. Parallelise independent jobs and shard the test suite across runners. Cache dependencies and build layers, with correct keys. Run only what a change affects, using path filters or affected-test selection in a monorepo. Fix the test pyramid: if slow end-to-end tests gate every commit, move most checks to fast unit and integration tests, and run the slow suite after merge or on a schedule, with fast feedback up front. Use bigger or faster runners if it's cheap relative to developer time. Remove flaky tests that force reruns. Set a target, say under ten minutes for the commit stage, and track pipeline duration and queue time as metrics with an owner. Importantly, communicate and measure the result: smaller batches, higher deploy frequency, and lower change failure rate, which are the DORA outcomes.

Likely follow-ups

- How does a slow pipeline increase risk per deploy?
- What would you move out of the commit-gating stage?

● SYMPTOM

A slow pipeline (45+ min) is changing behavior: engineers batch many changes into one big merge to amortize the wait, so each deploy is larger and riskier.

● INVESTIGATION

```
# profile the pipeline: which stages dominate the wall-clock?
```

```
# what's serial that could be parallel? what's rebuilt that could be cached?
```

```
# test pyramid shape: are slow E2E tests gating every commit?
```

Slow pipelines create a perverse incentive: big batches mean bigger blast radius and harder rollbacks — the opposite of safe continuous delivery. The latency itself is a reliability risk, not just a productivity annoyance.

● ROOT CAUSE

The pipeline's cycle time is high enough to discourage small, frequent changes. Likely causes: serial stages, no caching, an inverted test pyramid (too many slow E2E tests on the critical path), or full rebuilds of unchanged components.

● FIX / MITIGATION

```
# parallelize independent stages; cache deps & build layers
```

```
# move slow E2E tests off the blocking path (post-merge / nightly)
```

```
# build/test only what changed (affected-targets)
```

```
# fail fast: cheap lint/unit first
```

● PREVENTION

Keep the blocking path fast (fail-fast cheap checks, parallelism, caching, change-scoped builds), push slow suites to non-blocking lanes, and track pipeline cycle time as an SLO — because batch size is downstream of pipeline speed. Small, frequent, safe.

Interview tip: Connect speed to *safety*: slow pipelines → big batches → risky deploys. That systems-thinking framing, plus concrete levers (parallelize, cache, change-scoped builds, demote E2E), is the senior answer.

Official docs

- [GitHub Actions: Caching dependencies](#)

S95 **Users report they save a change and it 'disappears' on the next page load. You're behind a read replica. What's happening?**

Data / Stateful

 Spoken answer · ~81 s

A user saving a change and not seeing it on the next load, then seeing it correct moments later, is classic read-after-write inconsistency from replication lag. The write goes to the primary, but the reload is load-balanced to a replica that hasn't yet applied the change. I confirm by checking replication lag, with `pg_stat_replication` or seconds behind, and by tracing one request to see that the write hit the primary and the read hit a lagging replica. I look at the lag distribution, not just the average, because spikes cause the user-visible issue. The fixes are about guaranteeing read-your-writes. Route a user's reads to the primary for a short window after they write, using a cookie or session marker. Or route reads that must be fresh, such as the page right after a save, to the primary always. Or wait for the replica to reach the write's position using the log sequence number or GTID. Sticky sessions to one replica can help partly. Also reduce the lag itself: fix slow replicas, long-running queries, or write-heavy bursts. Design the UI to show the saved value optimistically. Monitor lag and alert when it exceeds the tolerance.

Likely follow-ups

- How would you implement read-your-writes with replicas?
- Why look at the lag distribution rather than the average?

 SYMPTOM

A user updates their profile, reloads, and sees the old value — then it's correct moments later. Writes go to the primary; reads are load-balanced across replicas.

 INVESTIGATION

```
# check replication lag (Seconds_Behind_Master / pg_stat_replication)
```

```
# trace one request: did the write hit primary and the read hit a lagging replica?
```

```
# measure lag distribution, not just the average
```

This is a read-after-write consistency violation caused by replication lag. The write committed on the primary, but the immediate read landed on a replica that hadn't received it yet, so the user sees stale data. It "fixes itself" once the replica catches up.

 ROOT CAUSE

Asynchronous replication means replicas trail the primary. Routing a user's own just-written read to a lagging replica breaks their expectation of seeing their own change.

 FIX / MITIGATION

```
# route read-your-writes to the primary (sticky for a short window after a write)
```

```
# or use causal/consistent reads: wait for replica to reach the write's LSN/GTID
```

```
# alert on lag and pull replicas from rotation past a threshold
```

● PREVENTION

Implement read-your-writes (route a user's reads to the primary briefly after their write, or track the write position and only read from a caught-up replica), and treat replication lag as a first-class SLI with alerting and auto-eviction of badly-lagging replicas.

Interview tip: Name it precisely — “read-after-write consistency under async replication” — and offer the standard fixes (sticky-to-primary window, or LSN/GTID-aware reads). That vocabulary signals real database ops experience.

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)

S96 The app starts throwing 'too many connections' / connection pool timeouts under modest load. How do you diagnose?

Data / Stateful

Spoken answer · ~79 s

Too many connections under modest load means connections are being held rather than used, since they're a finite resource. On the database side, I look at the process list or `pg_stat_activity`: how many connections, and in which states? A lot of idle in transaction connections means the application opens a transaction and holds it while doing other work, which pins a connection. On the app side, I compare pool size multiplied by the number of app instances against the database's `max_connections`; scaling out the application multiplies the demand, which is why it breaks under modest growth. Other causes: connection leaks where the code doesn't return connections to the pool, slow queries that hold connections longer so the pool drains, and long transactions. Fixes: close or return connections reliably, with `try-finally`; shorten transactions and never do network calls inside them; size pools deliberately, with a smaller pool often performing better; put a pooler such as PgBouncer or RDS Proxy in front of the database to multiplex many clients over few connections; set statement and idle-in-transaction timeouts; and tune `max_connections` only with memory in mind. Monitor pool wait time and connection counts.

Likely follow-ups

- Why can a smaller connection pool perform better?
- What does PgBouncer do?

● SYMPTOM

Under only moderately higher traffic, requests fail with connection-pool timeouts or the DB rejects connections with “too many connections.”

● INVESTIGATION

DB side: SHOW PROCESSLIST / pg_stat_activity – how many, and in what state?

app side: pool size × number of app instances vs DB max_connections

are connections idle-in-transaction (held but not working)?

Connections are a finite resource. Either the pool is too small (requests queue and time out) or, more often, too large × too many app replicas, overwhelming the DB's max_connections. Idle-in-transaction connections (held open mid-transaction) silently consume slots without doing work.

● ROOT CAUSE

Connection demand exceeds supply somewhere in the chain: app pool → (optional proxy) → DB. Frequently the real culprit is leaked or idle-in-transaction connections, or simply pool_size × replicas exceeding the DB limit.

● FIX / MITIGATION

fix leaks: ensure connections are returned; set idle-in-transaction timeout

introduce a connection pooler (PgBouncer/ProxySQL) to multiplex

right-size: total app connections must be < DB max_connections

● PREVENTION

Put a pooler in front of the database to multiplex many app connections onto few DB connections, set idle and idle-in-transaction timeouts, and capacity-plan so (pool_size × replicas) stays comfortably under max_connections. Connections are capacity — budget them.

Interview tip: The trap to surface: people scale app replicas and forget each multiplies pool connections into a fixed DB limit. Mention a pooler (PgBouncer) and idle-in-transaction as the two things that fix most of these.

📖 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [PgBouncer](#)

 Spoken answer · ~82 s

A query that slowed over time with no code change points to data growth or changed statistics. I start with EXPLAIN ANALYZE to see the actual plan: is it doing a sequential scan on a table that has grown large, using the wrong index, or are the row estimates far from reality? I check table and index sizes and growth, and whether statistics are fresh, since stale stats make the planner choose a bad plan; running ANALYZE may fix it. Bloat from dead tuples, if autovacuum isn't keeping up, makes scans slower. Compare the current plan with how the query behaved earlier, since the planner may have switched plans as data crossed a threshold, or a cached plan became wrong for skewed parameters. Then the fixes: add or adjust an index that matches the filter and sort, perhaps a composite or partial index; rewrite the query to be selective, avoiding functions on indexed columns; update statistics or increase the statistics target for skewed columns; vacuum or reindex; or reduce the data with partitioning and archiving old rows. Verify with EXPLAIN again and with production timing. And add monitoring for slow queries and query regressions over time.

Likely follow-ups

- Why can stale statistics cause a bad plan?
- When does partitioning help a slow query?

 SYMPTOM

A query that was fast at launch now times out months later. No code change touched it — only the data grew.

 INVESTIGATION

```
EXPLAIN ANALYZE <query>    # seq scan? wrong index? bad row estimate?
```

```
# check table/index growth and stats freshness
```

```
# is the plan different now than before? (cached plan / stats drift)
```

As data grows, a query that once scanned a small table now scans a large one. Either an index is missing (so it's a sequential scan), the right index exists but stale statistics make the planner avoid it, or selectivity changed so the old plan is no longer optimal.

 ROOT CAUSE

Data volume crossed a threshold where the access path that was fine at small scale (e.g. a seq scan) became pathological. Often compounded by out-of-date planner statistics choosing a bad plan.

 FIX / MITIGATION

```
# add/adjust the index to support the query's predicates & sort
```

```
ANALYZE <table> # refresh statistics so the planner chooses well
```

```
# consider partitioning / archiving cold rows if the table is huge
```

● PREVENTION

Run `EXPLAIN ANALYZE` in code review for queries on growing tables, keep statistics fresh (autovacuum/auto-analyze tuned), monitor slow-query logs continuously, and design indexes for actual access patterns rather than waiting for scale to expose the gap.

Interview tip: Lead with `EXPLAIN ANALYZE` and the seq-scan-at-scale story. Bonus points for naming *stale statistics* — the subtle case where the index exists but the planner won't use it.

Official docs

- [GitHub Actions: Caching dependencies](#)

S98 Transactions intermittently fail with deadlock errors under concurrency. How do you find and fix the cause?

Data / Stateful

Spoken answer · ~78 s

A deadlock happens when two transactions each hold a lock the other needs, so the database aborts one. First I capture the details: in MySQL, show engine innodb status gives the latest detected deadlock; in Postgres, enabling `log_lock_waits` and reading the deadlock detail in the logs shows both queries. I identify the two transactions and the order in which each acquired locks. The classic cause is acquiring the same locks in opposite orders, for example one transaction updates row A then row B while another updates B then A. Fixes: make every code path acquire locks in a consistent order, such as sorting ids before updating; keep transactions short, and do slow work outside them; reduce the rows locked by using indexes so queries don't lock ranges unnecessarily; use appropriate isolation levels and select for update only when needed; and batch updates in deterministic order. Because deadlocks can still occur occasionally, the application must handle the error by retrying the transaction with backoff, idempotently. Monitoring the deadlock rate helps spot regressions after releases. For hot rows, consider redesigning, for example using atomic increments or queueing updates.

Likely follow-ups

- Why does consistent lock ordering prevent deadlocks?
- Why must the application still retry after a deadlock error?

● SYMPTOM

Under concurrent load some transactions abort with deadlock-detected errors. It's sporadic and load-dependent.

● INVESTIGATION

```
SHOW ENGINE INNODB STATUS  # 'LATEST DETECTED DEADLOCK' (MySQL)
```

```
# Postgres: log_lock_waits + the deadlock detail in logs
```

```
# identify the two transactions and the lock-acquisition order each took
```

A deadlock occurs when two transactions acquire the same locks in opposite order — each holds what the other needs. The DB detects the cycle and kills one as the victim. Inconsistent lock ordering across code paths is the usual root cause.

● ROOT CAUSE

Two transactions take locks in conflicting orders (e.g. row A then B vs B then A). Under low concurrency they rarely interleave badly; under load they collide and the engine breaks the tie by aborting a victim.

● FIX / MITIGATION

```
# enforce a consistent lock/access ordering across all code paths
```

```
# keep transactions short; touch rows in a deterministic order
```

```
# add application-level retry on deadlock (they're expected, recoverable)
```

● PREVENTION

Standardize lock acquisition order, keep transactions small and short-lived, use appropriate isolation levels, and build retry-on-deadlock into the data layer since deadlocks are a normal, recoverable outcome under concurrency — not a catastrophic failure.

Interview tip: Two ideas win: **consistent lock ordering** prevents most deadlocks, and **retry** handles the rest since they're expected. Read the engine's deadlock report to recover the exact ordering conflict.

Official docs

- [PostgreSQL: Explicit locking](#)
- [Google Testing Blog: Flaky tests at Google](#)

 Spoken answer · ~83 s

One hot partition with the others idle means the partition key distributes load unevenly. First I identify the hot key or range using per-shard QPS and latency, and decide whether it's a single hot key, like a celebrity account or a popular product, or a skewed range, like sequential keys or timestamps that all land on the newest shard. Then fix by cause. For a skewed key choice, change the partition key to one with higher cardinality and an even access distribution, or use hashing instead of range partitioning for sequential keys. For a single hot key, split the load: add a random suffix or salt to spread writes across several partitions and aggregate on read; cache the hot key in front of the database; replicate the hot item to multiple read replicas; or use write batching and counters sharded across rows. In the short term, increase capacity on that shard or enable adaptive capacity features if the database supports them, such as DynamoDB, while redesigning. Rebalancing and resharding are heavy, so plan them. Prevention: load test with realistic access patterns before choosing a key, monitor per-partition metrics, and avoid designs where a monotonic value is the partition key.

Likely follow-ups

- How does key salting spread load?
- Why are sequential keys a problem for range partitions?

 SYMPTOM

A sharded datastore shows one node at 95% utilization while peers idle. Latency spikes correlate with traffic to that one partition.

 INVESTIGATION

identify the hot key/partition (per-shard QPS & latency)

is it one key (celebrity/hot row) or a skewed key range?

check the partition key choice vs the actual access distribution

The partition key distributes data unevenly — either a single “celebrity” key concentrates traffic, or the key has low cardinality / monotonic ordering so writes pile onto one shard (the classic timestamp- or sequential-ID-as-key mistake).

 ROOT CAUSE

Poor key distribution. The shard key doesn't spread load proportional to access, so one partition absorbs disproportionate traffic — a hotspot the other shards can't relieve.

 FIX / MITIGATION

add a hot cache in front of celebrity keys to absorb reads

re-key with higher cardinality / a hashed or composite key

split the hot partition; for write hotspots, add a salt/prefix

● PREVENTION

Choose partition keys with high cardinality matched to access patterns, avoid monotonic keys for write-heavy tables (hash or salt them), cache celebrity keys, and monitor per-partition load so skew is visible before it becomes a latency cliff.

Interview tip: Name the two flavors — single hot key (cache it) vs skewed/monotonic key (re-key or salt). The “timestamp/auto-increment as shard key” antipattern is a great concrete example to cite.

Official docs

- [Microsoft: Sharding pattern](#)

S100 After a primary failover you see signs of split-brain — two nodes think they're primary. How do you handle it?

Data / Stateful

Spoken answer · ~82 s

Two nodes both accepting writes after a failover is split-brain, and the priority is to stop the divergence. First confirm both really accept writes and check the cluster manager and quorum state to understand what triggered the failover; typically a network partition let a new primary be promoted while the old one still believed it was primary. Step one: immediately fence one node. Pick the one to keep, normally the one with the most recent or most valuable data, or the one the quorum recognises, and isolate the other by stopping it, revoking its network or storage access, or setting it read-only; the technique is called STONITH or fencing. Step two: assess divergence: which writes landed on each side since the split, using binlogs or write-ahead logs and timestamps. Step three: reconcile. Replay or merge the missing writes from the fenced node into the surviving primary where possible, which may need manual conflict resolution, and then rebuild the fenced node as a replica from the surviving primary. Communicate impact to the business, especially for conflicting transactions. Prevention: quorum-based leader election with an odd number of nodes, fencing tokens, a witness node, and tested failover with proper timeouts.

Likely follow-ups

- What does fencing or STONITH do?
- How does quorum prevent split-brain?

● SYMPTOM

Following a network blip and failover, two database nodes both accept writes. Data is now diverging between them.

● INVESTIGATION

```
# confirm both are accepting writes (each thinks it's primary)
```

```
# check the cluster manager / quorum state and what triggered failover
```

```
# assess divergence: which writes landed where since the split?
```

A network partition let a failover promote a new primary while the old one was still alive and writable — classic split-brain. Without quorum/fencing, both sides accept conflicting writes, and merging divergent data afterward is painful or lossy.

● ROOT CAUSE

Failover happened without proper fencing or quorum, so the old primary wasn't reliably demoted/isolated before the new one took over. The partition created two writable masters.

● FIX / MITIGATION

```
# STOP the divergence: fence/demote one node to a single writable primary
```

```
# choose the authoritative copy; reconcile or replay the other's writes carefully
```

```
# this is often a data-loss event – escalate and decide RPO tradeoffs explicitly
```

● PREVENTION

Use quorum-based failover (odd member count, majority required to promote) and fencing/STONITH so a partitioned old primary is forcibly demoted before a new one is promoted. Design for it up front — split-brain recovery is far costlier than prevention.

Interview tip: Show you understand **quorum + fencing** are what prevent this, and that recovery is often lossy (a real RPO conversation). Connect it to CAP: during a partition you can't have both consistency and availability — pick deliberately.

Official docs

- [Google SRE Book: Managing Critical State \(consensus\)](#)

S101 **A disaster strikes and you go to restore from backups — only to find the restore process has never actually been tested. What now, and what's the lesson?**

Data / Stateful

🗣 Spoken answer · ~83 s

Finding out that restores were never tested is the moment the lesson becomes expensive. Right now, I focus on recovery. First verify the backups exist and are readable and complete, since a job exit code of success only means the job ran, not that the data is usable. I attempt a restore into an isolated environment, time it, and check the backup scope: does it include all tables, schema, and a consistent snapshot, or are pieces missing? I escalate and communicate realistic recovery time, because a restore of a large dataset can take many hours and the business needs honest expectations. If the restore works, validate the data with row counts and application-level checks before cutting over. If it partly fails, explore other sources: replicas, point-in-time recovery with logs, snapshots, and application-level exports. The lesson: a backup is not real until it's been restored. So I set up regular automated restore tests into a scratch environment with integrity checks and alerting on failure, measure the real RTO and RPO against the targets, keep backups off-site and immutable against ransomware and accidental deletion, monitor backup age and size anomalies, and run periodic disaster recovery game days with documented runbooks.

Likely follow-ups

- Why isn't a successful backup job proof of recoverability?
- What does an automated restore test check?

● SYMPTOM

Data loss occurs. You reach for backups and discover that while backups *ran*, a full restore was never once tested end-to-end.

● INVESTIGATION

```
# verify backups exist AND are readable/complete (not just that jobs 'succeeded')
```

```
# attempt restore to an isolated environment, time it
```

```
# check backup scope: schema only? missing tables? consistent snapshot?
```

"Backups succeeded" only means the job exited zero — not that the data is restorable. Untested backups commonly fail at restore time: incomplete scope, inconsistent (non-atomic) snapshots, unreadable files, or a restore that takes far longer than the business can tolerate (RTO miss).

● ROOT CAUSE

Backups were treated as a checkbox, not a tested capability. The real recoverability — completeness, consistency, and restore time — was never validated, so its true state is unknown precisely when it matters most.

● FIX / MITIGATION

```
# attempt the restore now in isolation; salvage whatever is recoverable
```

```
# if backups are bad, pivot to other sources (replicas, WAL/binlog, app logs)
```

```
# document actual RPO/RT0 achieved vs what was assumed
```

● PREVENTION

Test restores on a schedule (automated restore drills), verify backup *content* not just job exit codes, measure real RTO against the SLA, and store backups with independent durability (different account/region, immutable) so a single failure or ransomware can't take both prod and backups.

Interview tip: The line that lands: "a backup you haven't restored is a hope, not a backup." Talk about scheduled restore drills, verifying content, and measuring real RTO/RPO — that's the difference between checkbox and capability.

Official docs

- [Google SRE Book: Data Integrity](#)

S102 A schema migration on a large table took a long lock and stalled the whole application. How should it have been done?

Data / Stateful

Spoken answer · ~83 s

An ALTER TABLE on a multi-hundred-gigabyte table that locked it for minutes is an outage caused by the wrong method. Some DDL operations rewrite the whole table or take a lock that blocks reads and writes for the duration, and running it inside a long transaction holds the lock even longer. First I identify what lock the DDL took and what it blocked, which depends on the engine and operation. How it should have been done: use an online schema change approach. In MySQL, tools such as gh-ost or pt-online-schema-change build a shadow table, copy data in chunks, keep it in sync with triggers or the binlog, and swap it atomically at the end. In Postgres, many changes are cheap when done carefully: add a nullable column without a default, or with a default on modern versions, create indexes concurrently, and add constraints as not valid and validate them later. Set a lock timeout, so the DDL fails fast instead of queueing and blocking everything behind it. Follow expand and contract so the app works with both old and new schemas. Run during low traffic, test on a production-sized copy to estimate duration, and have a rollback plan.

Likely follow-ups

- Why set a lock timeout on DDL?
- How do online schema change tools work?

● SYMPTOM

An ALTER TABLE on a multi-hundred-GB table acquired a lock and blocked reads/writes for minutes, effectively an outage.

● INVESTIGATION

```
# which lock did the DDL take, and what did it block? (engine-dependent)
```

```
# table size & whether the operation rewrites the whole table
```

```
# was it run in a transaction holding the lock for the full duration?
```

Some DDL operations rewrite the entire table and/or take exclusive locks, scaling with table size. On a large table that's minutes of blocking — an outage. The migration was correct logically but operationally hostile because it wasn't done online.

● ROOT CAUSE

A blocking, table-rewriting schema change was run directly against a large hot table. The size made the lock duration intolerable.

● FIX / MITIGATION

```
# use online schema-change tooling (gh-ost, pt-online-schema-change) or native online DDL
```

```
# or expand/contract: add nullable column, backfill in batches, then enforce
```

```
# run heavy migrations in low-traffic windows with a timeout/kill-switch
```

● PREVENTION

Adopt online/non-blocking migration tooling, prefer additive expand/contract changes with batched backfills, set lock timeouts so a migration aborts rather than queuing the whole app behind it, and rehearse migrations against a production-sized snapshot.

Interview tip: Name the tools (gh-ost / pt-osc) and the expand-contract pattern, plus **lock_timeout** so a migration fails fast instead of stalling everything. That trio shows you've shipped schema changes safely at scale.

Official docs

- [gh-ost: GitHub's online schema migration](#)
- [Flyway documentation](#)

S103 A popular cache key expires and the database gets hammered by a stampede of identical queries. How do you prevent thundering herd?

Data / Stateful

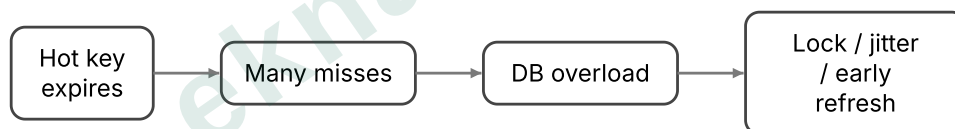
🗣 Spoken answer · ~80 s

A thundering herd on cache expiry happens when a hot key expires and thousands of concurrent requests all miss and hit the database with the same query at once. I confirm by correlating the spike with the TTL expiry of a hot key. The prevention techniques: request coalescing or a single-flight lock, so that only one request recomputes the value while the others wait for it or get the stale value; stale-while-revalidate, serving the slightly stale entry while one background refresh runs; jittered TTLs, adding random variation so many keys, and keys across instances, don't expire at the same instant; proactive refresh, where a background job renews hot keys before they expire; and a longer TTL with explicit invalidation on writes for data that rarely changes. Also consider a second-level in-process cache with short TTL to absorb bursts, negative caching for missing items, and protecting the database with rate limiting or a queue, plus circuit breakers, so a cache miss storm can't take it down. For cold starts after a cache flush or deploy, warm the cache before taking traffic. Monitor cache hit ratio and database load, and load test the miss scenario.

Likely follow-ups

- What is single-flight and how does it help?
- Why add jitter to TTLs?

🚩 Cache stampede



● SYMPTOM

At a predictable moment, latency spikes and the DB CPU pegs. A hot cache key just expired and thousands of concurrent requests all miss and hit the DB at once.

● INVESTIGATION

correlate the spike with a TTL expiry of a hot key

how many concurrent identical queries hit the DB on the miss?

is every replica/instance recomputing the same value independently?

When a heavily-read cache entry expires, every concurrent request misses simultaneously and stampedes the backend with identical work — a thundering herd / cache stampede. The DB, sized for cache-hit load, can't absorb the sudden synchronized miss.

● ROOT CAUSE

Synchronized cache expiry plus high read concurrency. Without coordination, N requests each independently regenerate the same value, multiplying load by N at the exact moment the cache can't help.

● FIX / MITIGATION

```
# single-flight / request coalescing: one request recomputes, others wait
```

```
# add jitter to TTLs so keys don't expire in lockstep
```

```
# early/probabilistic refresh before expiry; serve stale-while-revalidate
```

● PREVENTION

Coalesce concurrent regenerations (single-flight lock per key), jitter TTLs to avoid synchronized expiry, refresh hot keys proactively before they expire, and serve stale-while-revalidate so a miss never exposes the full herd to the backend.

Interview tip: Three named techniques to drop: single-flight (request coalescing), TTL jitter, and stale-while-revalidate / early refresh. Knowing the term “thundering herd” and its mitigations is a strong cache-design signal.

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Caching challenges and strategies](#)

S104 Monitoring is green but customers report wrong data — you suspect silent data corruption. How do you approach it?

Data / Stateful

 Spoken answer · ~83 s

Green dashboards with customers reporting wrong data suggest silent data corruption, which produces no exceptions, so monitoring built on errors can't see it. First I reproduce and scope it: pin down which records are wrong, how they're wrong, and since when. Then I trace the data's lineage to find where it could have been mutated: a recent migration, a bad deploy, a faulty transformation or ETL job, a race condition or lost update, a replication or serialisation bug, timezone or encoding issues, or a bad manual script. I compare against backups, the audit log, or event history to find when the values diverged, which narrows the cause to a deploy or job. Containment: stop the process that is writing bad data, perhaps by rolling back or pausing the job, before more is corrupted. Then repair: restore correct values from backups or event logs, with a reconciliation script, test on a copy, and notify affected customers where required. To detect this class in future: add data quality checks and invariants, such as reconciliation jobs comparing sources, checksums, row count checks and business rule assertions, plus monitoring of business metrics, not just infrastructure. Use constraints in the schema, and validation at boundaries.

Likely follow-ups

- How do you find the moment data started to diverge?
- What data quality checks catch silent corruption?

SYMPTOM

No errors, no alerts, dashboards green — yet customers report incorrect values. You suspect data corruption that no exception ever surfaced.

INVESTIGATION

```
# reproduce: pin down which records are wrong and how
```

```
# trace the data's lineage: where could it have been mutated?
```

```
# compare against backups/audit log to find when it diverged
```

Silent corruption produces no exceptions — a bad migration, a race condition double-writing, a serialization bug, or bit-rot at the storage layer wrote plausible-but-wrong values. Because nothing threw, monitoring built around errors and latency stayed green.

ROOT CAUSE

Data integrity was violated without any error signal. The corruption path wrote valid-looking but incorrect data, so error-based observability is blind to it — you need correctness checks, not just liveness checks.

FIX / MITIGATION

```
# stop further corruption: identify and disable the offending write path
```

```
# determine blast radius; restore/repair affected records from a clean source
```

```
# add an audit trail so future divergence is attributable
```

● PREVENTION

Add data-quality and invariant checks (constraints, checksums, reconciliation jobs that compare against source-of-truth), enable storage-level integrity (checksums), keep an audit log of mutations, and alert on *correctness* SLIs — not just availability and latency.

Interview tip: The senior insight: error-based monitoring is blind to silent corruption — you need **correctness** checks (constraints, reconciliation, checksums). Stopping the bad write path before repairing data is the right order of operations.

Official docs

- [Google SRE Book: Data Integrity](#)

S105 On-call is drowning in alerts and starting to ignore them. How do you fix alert fatigue without going blind?

Observability

Spoken answer · ~85 s

Alert fatigue is a signal quality problem, so I fix it with data and then restructure what pages. First I measure: pull alert volume per rule over thirty days, rank by frequency, and for each noisy rule ask whether anyone did something or just acknowledged it. The actionability rate, the share of alerts leading to real action, is my key metric. Then triage each rule. If not actionable, delete it or downgrade it to a ticket or dashboard. If it flaps, add a duration, hysteresis, or a longer window. If it measures a cause, such as CPU or a single pod being down, replace it with symptom-based alerts tied to user impact, like SLO burn rate alerts, with multi-window logic so brief blips don't page. Group and deduplicate related alerts, and route by severity: only urgent, user-impacting, actionable conditions page a human, the rest go to tickets in business hours. Each page links to a runbook with clear first steps and an owner. To avoid going blind, keep coverage of the important user journeys through SLOs, and review alerts regularly, including after every incident, asking whether they fired helpfully. Track pages per shift as a health metric, and treat sustained noise as a reliability bug.

Likely follow-ups

- What does an actionable alert mean?
- Why are SLO burn-rate alerts less noisy than cause-based alerts?

● SYMPTOM

The on-call channel fires dozens of alerts a shift. Most are non-actionable, so engineers mute or reflexively ack them — and a real incident recently slipped through unnoticed.

● INVESTIGATION

```
# pull alert volume per rule over 30 days; rank by frequency
```

```
# for each noisy rule: did anyone DO something, or just ack?
```

```
# measure actionability: % of alerts that led to a real action
```

Alert fatigue comes from paging on causes and thresholds rather than on user-facing symptoms. Every static-threshold or duplicate-cause alert that doesn't require human action erodes trust until the one that matters is lost in the noise.

● ROOT CAUSE

Too many non-actionable, cause-based alerts. The signal-to-noise ratio dropped so low that humans rationally tuned out — which is how the real incident was missed.

● FIX / MITIGATION

```
# delete or downgrade non-actionable rules to dashboards/tickets
```

```
# alert on symptoms (SLO burn) not every cause
```

```
# group/dedupe related alerts into one incident; add sane thresholds
```

● PREVENTION

Adopt symptom-based, SLO-driven alerting: page humans only for user-impacting problems that need action, route everything else to dashboards or tickets, and review alert actionability regularly. Every page should be something a human must act on now.

Interview tip: The principle interviewers want: **alert on symptoms, not causes**, and "every page must be actionable." Tie it to SLO burn-rate alerting and a regular noise-review ritual.

Official docs

- [Google SRE Book: Practical Alerting](#)
- [Google SRE Workbook: On-Call](#)

S106 An incident happened with zero alerts — a complete monitoring blind spot. How do you find and close such gaps?

Observability

Spoken answer · ~88 s

An outage with zero alerts is a monitoring blind spot, and the postmortem should classify why. The first question: what failed, and what signal should have caught it? Then determine the flavour. Was the metric never instrumented? Did it exist but have no alert? Was there an alert, but misconfigured with the wrong threshold, a wrong label, or a silently broken query? Or did the failure mode produce no telemetry at all, which is the hardest case, like a job that simply stopped running? Closing the gap: add the missing instrumentation and alert, preferably from the user's perspective, such as an SLI measured at the edge and synthetic checks that run the real journey, since those catch failures regardless of cause. For things that stop happening, use absence alerts and heartbeats or dead man's switches. Alert on symptoms such as error rate, latency, and traffic drops, using the four golden signals. Test the alerts: fire a deliberate failure in game days to confirm they trigger, and verify the monitoring pipeline itself is healthy. Review the dependency map for components with no coverage. Include a detection time target, such as time to detect, in postmortems, and track it as a metric. Finally, check who gets notified and whether the routing worked.

Likely follow-ups

- How do you alert on something that stopped happening?
- Why are synthetic checks good at catching unknown failure modes?

SYMPTOM

A user-impacting outage ran for 20 minutes with no alert at all. The team found out from a customer, not the monitoring.

INVESTIGATION

what failed, and what signal SHOULD have caught it?

was the metric missing, the alert missing, or the alert misconfigured?

check: did the failure mode produce any telemetry at all?

Blind spots come in flavors: the metric was never instrumented, it existed but had no alert, the alert existed but had a wrong threshold/grouping, or the failure killed the very thing that emits telemetry (e.g. the metrics agent went down with the host). You can't alert on what you don't measure.

ROOT CAUSE

The system lacked a signal for this failure mode — either uninstrumented, unalerted, or self-defeating (the outage suppressed its own telemetry). Coverage was assumed but never verified for this path.

FIX / MITIGATION

```
# instrument the missing signal; add a symptom-level SLO alert
```

```
# add a synthetic/black-box probe that exercises the user path externally
```

```
# add meta-monitoring: alert when telemetry itself stops arriving (dead-man's switch)
```

● PREVENTION

Combine white-box (internal metrics) with black-box synthetic probes so user-facing breakage is caught even when internals lie, add dead-man's-switch alerts that fire when expected telemetry goes silent, and run game-days to discover blind spots before incidents do.

Interview tip: Two ideas elevate this: **black-box synthetic monitoring** for the user path, and a **dead-man's switch** so silence itself pages you. Mention deriving alerts from postmortem action items so each blind spot closes permanently.

Official docs

- [Google SRE Book: Distributed tracing & logs \(Monitoring chapter\)](#)

S107 Your metrics bill exploded and queries got slow after a deploy added a high-cardinality label. How do you fix it?

Observability

Spoken answer · ~81 s

A new label with near-unlimited values, like user ID, request ID, full URL, or IP address, explodes cardinality, because a time-series database creates one series for every unique combination of label values. So the number of series multiplies, memory use and query cost rise, and the bill grows. First I identify the offending metric using the series count per metric, for example the top series counts from the Prometheus TSDB status, and see which label was added in the recent deploy. Quick fix: roll back or drop the label with a relabelling rule at ingestion, using `metric_relabel_configs` or the collector, so the series stop growing, and consider deleting the bad series. Long-term fix: labels should have bounded, small value sets, such as status code class, route template, and region, never unique identifiers. For per-user or per-request detail, use logs or traces instead, where high cardinality is acceptable, or exemplars linking metrics to traces. Prevention: cardinality limits and sample limits per scrape target, review of new metrics in code review, CI checks on label usage, cardinality dashboards and alerts on series growth, and budgets per team. Educate developers on the difference between metrics, logs, and traces.

Likely follow-ups

- Why does each unique label combination create a new series?
- Where should per-user detail go instead?

● SYMPTOM

Right after a release, the metrics backend slows down and the bill spikes. A new label was added that takes near-unlimited distinct values.

● INVESTIGATION

```
# identify the metric with exploding series count
```

```
# what label was added? (user_id, request_id, full URL, IP?)
```

```
# count distinct label values = number of time series created
```

A time-series database creates one series per unique label-set. A high-cardinality label (user ID, request ID, raw URL) multiplies series into the millions, blowing up storage, query time, and cost. Cardinality is the dominant cost driver in metrics systems.

● ROOT CAUSE

An unbounded-cardinality dimension was attached to a metric, exploding the series count. Metrics are for aggregatable, low-cardinality dimensions — per-request identifiers belong in traces/logs, not labels.

● FIX / MITIGATION

```
# drop/relabel the offending high-cardinality label at ingest
```

```
# move per-request detail to tracing or structured logs
```

```
# add cardinality limits/guardrails at the ingestion layer
```

● PREVENTION

Enforce cardinality budgets and relabeling rules at ingest, keep metrics dimensions low-cardinality and bounded, and put per-entity detail in traces and logs where it belongs. Treat “don't put unbounded IDs in metric labels” as a reviewable rule.

Interview tip: The crisp rule: **metrics labels must be low-cardinality**; per-request IDs go to traces/logs. Knowing that cardinality (series count) is the cost/performance driver — not data volume per se — is the senior signal.

Official docs

- [Prometheus: Metric and label naming](#)

S108 A dashboard shows healthy average latency, but customers complain it's slow. Why the disconnect?

Observability

Spoken answer · ~82 s

A healthy average with unhappy customers is the classic mean-versus-tail disconnect. Averages hide tails: a low mean can coexist with a terrible P99, since if one percent of requests take five seconds, the mean barely moves, but at scale that is many users, and each page load makes many requests so the chance of hitting a slow one is high. So I look at percentiles, P50, P95, P99, and P99.9, using histograms rather than averaged summaries, because you can't average percentiles across instances. I break latency down by route, endpoint, region, and customer cohort, because the slow slice may be a particular endpoint, a large tenant, or one region or device type that the global number masks. I check whether the tail is far above the median, and look at traces of the slow requests to see where time goes. The disconnect can also come from measuring in the wrong place: server-side latency misses queueing, network, and client time, so real user monitoring is needed. Then I set SLOs on percentiles, such as ninety-nine percent of requests under a threshold, and alert on those. Fix the slow tail causes: GC pauses, cold caches, retries, a slow dependency.

Likely follow-ups

- Why can't you average percentiles across instances?
- Why does the tail matter more when a page makes many requests?

SYMPTOM

The latency dashboard shows a comfortable ~80ms average and looks healthy, yet a meaningful slice of users report the app is slow.

INVESTIGATION

look at percentiles, not the mean: p50 vs p95 vs p99

break latency down by route/endpoint and by cohort

is the tail (p99) far above the average?

Averages hide tails. A low mean can coexist with a brutal p99 — if 1% of requests take 5 seconds, the average barely moves but those users have a terrible experience. Worse, a single user often makes many requests, so tail latency hits more sessions than the percentile suggests.

ROOT CAUSE

The dashboard measures the wrong statistic. The mean masks a high-latency tail; the unhappy users live in p95/p99, which the average smooths away.

FIX / MITIGATION

switch dashboards/SLOs to percentiles (p95/p99) and histograms

```
# slice by endpoint & cohort to localize the slow tail
```

```
# fix the tail source (slow dependency, GC pauses, cold cache, lock contention)
```

● PREVENTION

Define latency SLOs on percentiles (and track histograms/heatmaps, not averages), because users experience the tail, not the mean. Slice by route and cohort so a slow segment can't hide inside a healthy aggregate.

Interview tip: The one-liner: **averages lie; track p95/p99**. Add the multiplicative point — one slow page in a many-request session ruins the whole session — to show you understand user-perceived latency, not just metrics math.

Official docs

- [Grafana documentation](#)

S109 An error occurred in production but the trace for it was never captured because of sampling. How do you handle this?

Observability

Spoken answer · ~84 s

When a trace for a reported failure was dropped by sampling, I first handle the immediate investigation without it. I use the request ID or timestamp to correlate logs across services, and the error logs and metrics around that time often reveal the failing hop. Then I fix the sampling strategy, since head-based sampling decides at the start of the request, before it's known whether the request will fail or be slow, so the interesting traces are randomly dropped. The better approach is tail-based sampling: collect spans, wait until the trace completes, then decide to keep all traces with errors or high latency, plus a sample of normal ones. This is done in a collector, such as the OpenTelemetry collector's tail sampling processor, with the cost of buffering. Alternatively, force sampling for errors, specific customers, or when a debug header is set, and use adaptive sampling that keeps rare routes. Also store the trace ID in logs and expose it in error responses, so a customer report can be tied to a trace, and use exemplars linking metrics to traces. Balance the cost: set budgets, sample healthy traffic heavily, and keep all the bad. Review that critical paths are always captured.

Likely follow-ups

- Why is tail-based sampling better for debugging errors?
- What is the cost of tail-based sampling?

● SYMPTOM

A customer reports a failed request with a timestamp, but when you look for its distributed trace, it's missing — sampling dropped it.

● INVESTIGATION

```
# what's the sampling strategy & rate? (head-based at fixed %?)
```

```
# was this an error that should always be kept?
```

```
# can you correlate via logs/request-id even without the trace?
```

Head-based sampling decides whether to keep a trace at the start, before knowing the outcome — so it discards errors and slow requests at the same rate as boring successes. The interesting traces (the ones you actually need) get thrown away with everything else.

● ROOT CAUSE

Head-based (up-front) sampling is outcome-blind. It optimized for cost by dropping a fixed fraction, but that fraction included the error you now need, because the decision was made before the error existed.

● FIX / MITIGATION

```
# switch to tail-based sampling: keep ALL errors & slow traces, sample the rest
```

```
# always-keep policy for error spans and high-latency spans
```

```
# correlate via request-id across logs in the meantime
```

● PREVENTION

Use tail-based sampling so every error and slow request is retained while normal traffic is sampled for cost, propagate a request/correlation ID across logs and traces so you can stitch the story even when a trace is partial, and reserve full-fidelity capture for anomalies.

Interview tip: The key distinction: **head-based vs tail-based sampling**. "Keep all errors and slow traces, sample the happy path" is the answer. Correlation IDs as the cross-signal fallback shows depth.

Official docs

- [OpenTelemetry: Sampling](#)

S110 An SLO is burning error budget fast mid-incident. How do you use burn rate to drive the response?

Observability

Spoken answer · ~83 s

Burn rate tells me how fast I'm consuming error budget relative to the sustainable rate, so it translates an incident into urgency. A burn rate of one means I'd exactly use the budget over the SLO window; ten means I'm spending it ten times faster, so a thirty-day budget would be gone in three days. During an incident I compute the current burn rate, and from it how long until the budget is exhausted at this pace. I look at multiple windows: a fast burn on a one-hour window with a high multiplier means page now and mitigate immediately; a slow burn on a six or twenty-four hour window means the problem is real but can be handled as a priority ticket. Multi-window alerts, requiring both a long and a short window to exceed the threshold, reduce false alarms and also confirm the problem is still ongoing. The burn rate drives decisions: severity, whether to roll back instead of debug, who to wake, and what to tell leadership in business terms, like we've consumed forty percent of this month's budget. Afterwards, the remaining budget influences policy: if it's exhausted, freeze risky releases and prioritise reliability work until it recovers.

Likely follow-ups

- What does a burn rate of 10 mean?
- Why use two windows in a burn-rate alert?

SYMPTOM

During an incident, the error-budget burn-rate alert is firing. Leadership wants to know how serious it is and how to prioritize.

INVESTIGATION

current burn rate = how many x the sustainable rate are we spending budget?

at this rate, how long until the budget is exhausted?

multi-window check: is it a fast burn (1h) or slow burn (6h/24h)?

Burn rate expresses how fast you're consuming the error budget relative to the rate that would exactly exhaust it over the SLO window. A 14.4x burn over an hour means a month's budget gone in hours — page now. A slow 2x burn is a ticket, not a 3am page. Multi-window/multi-burn-rate alerts separate “act now” from “look soon.”

ROOT CAUSE

The SLO is being violated fast enough that the error budget will be exhausted well before the window ends. Burn rate quantifies urgency, turning a vague “it's bad” into a concrete “act in minutes vs hours.”

FIX / MITIGATION

fast/high burn → immediate mitigation: roll back, shed load, failover

protect the budget: stop risky deploys (freeze) while burning

slow burn → prioritized fix without heroics

● PREVENTION

Define SLOs with multi-window multi-burn-rate alerts (fast-burn pages, slow-burn tickets), use remaining error budget to gate release risk (freeze deploys when budget is low), and frame incident urgency in burn-rate terms so response effort matches actual user impact.

Interview tip: Show command of **multi-window, multi-burn-rate** alerting and using the budget to gate deploys. Translating burn rate into “time-to-exhaustion” is exactly how senior SREs prioritize during an incident.

📖 Official docs

- [Google SRE Book: Service Level Objectives](#)
- [Google SRE Book: Embracing Risk \(error budgets\)](#)
- [Google SRE Workbook: Alerting on SLOs](#)

S111 Log volume (and cost) is exploding, but you still can't find what you need during incidents. How do you fix logging?

Observability

🗣 Spoken answer · ~80 s

High log volume with poor usability means the logs are verbose, unstructured, and poorly organised. I start by measuring: volume by service and level, and how much is debug and info noise versus actionable. Then fix both cost and usefulness. Structure: emit structured JSON logs with consistent fields such as timestamp, level, service, request or trace ID, and error details, so they're queryable instead of grep-only. Reduce noise: set production log levels to info or warn, remove per-request chatter and debug logging, sample high-volume repetitive logs, deduplicate repeated errors, and avoid logging large payloads or secrets. Add the right context to important events, not more events. Route and tier: send logs through a pipeline that filters and drops low-value data, keep hot searchable storage for a short retention like seven days, move older logs to cheap object storage, and set retention by data class. Correlate: include trace and correlation IDs so logs link to traces and metrics, and use metrics for counting and alerting instead of counting log lines. Set budgets per team, and review log cost in showback. During incidents the result is fewer, richer entries that answer questions quickly.

Likely follow-ups

- Why should you use metrics rather than log counts for alerting?
- What fields belong in every structured log line?

● SYMPTOM

The logging bill keeps climbing, yet on-call still struggles to find relevant entries during incidents — too much noise, not enough signal.

● INVESTIGATION

what's the volume breakdown by service & log level?

how much is DEBUG/INFO noise vs actionable?

are logs structured (queryable) or free-text (grep-only)?

High volume and low usefulness coexist when logs are verbose, unstructured, and undifferentiated — chatty DEBUG/INFO at full retention, free-text that's hard to query, and no correlation IDs to tie a request's logs together. You pay to store noise and still can't search it.

● ROOT CAUSE

Logging is optimized for volume, not retrievability. Without structure, levels, and correlation, more logs just means more haystack — cost rises while findability falls.

● FIX / MITIGATION

structure logs (JSON) with consistent fields + a correlation/request ID

tier retention by level; sample high-volume INFO/DEBUG

move high-cardinality detail to traces; keep logs for events

● PREVENTION

Use structured logging with correlation IDs so a request's story is queryable, set log levels and tiered/sampled retention to control volume, and split responsibilities across the observability pillars — metrics for aggregates, traces for request flow, logs for discrete events — rather than dumping everything into logs.

Interview tip: Connect cost and usefulness: **structure + correlation IDs + tiered retention** fixes both at once. Mentioning the three-pillars division of labor (metrics/traces/logs) shows you think in observability architecture, not just "more logging."

Official docs

- [Prometheus: Storage \(TSDB\)](#)

S112 **During an incident you can't trace a request across services because there's no correlation ID. How do you fix this going forward?**

Observability

 Spoken answer · ~84 s

Without a correlation ID, each service's logs are islands, so tracing a request across services during an incident is slow and error-prone, reconstructing the path from timestamps. Going forward, I introduce a correlation ID, ideally adopting distributed tracing standards. At the entry point, the edge or API gateway, generate a unique request ID if the incoming request lacks one, and propagate it on every outgoing call: in HTTP headers, using the W3C Trace Context traceparent header, in message metadata for queues and events, and in gRPC metadata. Every service must read the ID, attach it to its logging context so every log line includes it automatically, using middleware, and pass it on to downstream calls. Context can be lost at async boundaries, thread pools, and queue consumers, so I check those explicitly. Better still, adopt OpenTelemetry for tracing, which gives IDs, spans, and timing across hops with auto-instrumentation libraries, and ties logs and metrics to traces. Return the ID to clients in a response header or error body so support can quote it. Add a test or lint check that verifies propagation, and an incident runbook entry for searching by ID. Roll out service by service, starting with the critical paths.

Likely follow-ups

- Where is correlation context most often lost?
- What is the W3C traceparent header?

 SYMPTOM

Mid-incident, a request fails somewhere across a chain of microservices, but each service's logs are islands — there's no shared ID to follow the request end-to-end.

 INVESTIGATION

can you reconstruct the path manually by timestamps? (slow, error-prone)

do any services propagate ANY shared identifier?

where does the request enter, and where is context first lost?

Without a correlation/trace ID generated at the edge and propagated through every hop (via headers/context), each service logs independently and there's no way to stitch one request's journey together. Debugging distributed flows by timestamp alone is unreliable, especially under concurrency.

 ROOT CAUSE

No distributed-context propagation. The system never threads a shared identifier through the call chain, so cross-service causality is unrecoverable during an incident — exactly when you need it most.

 FIX / MITIGATION

```
# generate a trace/correlation ID at the edge (or accept inbound one)
```

```
# propagate it via context/headers through every service & async hop
```

```
# include it in every log line; adopt OpenTelemetry context propagation
```

● PREVENTION

Standardize distributed tracing with context propagation (OpenTelemetry), inject a trace/correlation ID at the ingress and carry it through sync and async boundaries, and stamp it on every log line so traces and logs cross-reference. Make propagation a platform default, not a per-service afterthought.

Interview tip: The answer is **distributed context propagation** — a trace ID minted at the edge and carried everywhere, ideally via OpenTelemetry. Stress propagation across async boundaries (queues), where teams most often drop the context.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S113 A major traffic event (think Black Friday) is coming and autoscaling alone won't save you. How do you prepare?

Capacity & Scale

Spoken answer · ~80 s

Autoscaling reacts, so it adds capacity after load arrives, which isn't enough for a ten-times, known-time event. I prepare in advance. First, model the expected peak precisely: ten times which metric, requests per second, concurrent users, or writes? Then find the slowest-to-scale tier, which is usually the database or other stateful systems, cold caches, and quotas, not stateless app servers. Actions: load test to the target peak, finding the first bottleneck and fixing it iteratively; pre-scale ahead of the event instead of waiting for metrics, using scheduled scaling or a manual floor, and warm caches, connection pools, and JIT; raise cloud quotas and limits in advance, such as instance counts, load balancer capacity pre-warming, and API rate limits; scale the database with read replicas, bigger instances, or caching, since you can't autoscale it quickly. Add protective measures: rate limiting, load shedding, queueing for non-critical work, feature flags to turn off expensive features, and a virtual waiting room if needed. Freeze risky deploys before the event. Plan the war room: runbooks, on-call coverage, dashboards on key SLIs, clear decision rights, and a rollback plan. Rehearse with a game day, and review afterwards.

Likely follow-ups

- Why is the database usually the tier that fails first?
- What is pre-scaling and why use it?

● SYMPTOM

Marketing announces a flash sale expected to drive 10× normal traffic at a known time. The team's plan is "autoscaling will handle it."

● INVESTIGATION

model expected peak: 10x of WHICH metric? RPS, concurrent users, writes?

what's the slowest-to-scale tier? (DB, stateful, cold caches, quotas)

how long does a new node/pod take to become ready & warm?

Autoscaling reacts — it adds capacity *after* load arrives, with a lag (provision + boot + warm-up). For a sharp, scheduled spike that lag means you're under-provisioned exactly when it matters. And the hardest tiers (databases, stateful services, third-party quotas) don't scale on the same timeline as stateless pods.

● ROOT CAUSE

Reactive scaling can't absorb an instantaneous, predictable spike, and non-elastic tiers (DB, quotas, caches) become the bottleneck. "Autoscaling will handle it" ignores both the lag and the parts that don't autoscale.

● FIX / MITIGATION

PRE-SCALE / pre-warm before the event; don't wait for reactive scaling

load-test at 10x to find the real bottleneck (usually DB/quota, not app)

add shedding/queueing & a degraded-mode fallback for overflow

● PREVENTION

Combine pre-provisioning for known events with load testing to find the true bottleneck, raise cloud/quotas and DB capacity ahead of time, warm caches and connection pools, and design graceful degradation (load shedding, queues, read-only fallbacks) so overflow degrades instead of collapsing.

Interview tip: The headline: **pre-scale for predictable spikes; autoscaling is for the unpredictable.** Then name the non-elastic bottlenecks (DB, quotas, cold caches) and graceful degradation — that's the senior capacity-planning instinct.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S114 Your autoscaler is flapping — rapidly scaling up and down — causing instability. How do you stabilize it?

Capacity & Scale

 Spoken answer · ~82 s

Flapping autoscaling happens when the scaling signal is noisy and the scale-up and scale-down thresholds are too close together. I start with the metric that drives scaling and how noisy it is: a spiky instantaneous CPU reading is a poor input. Fixes: smooth the signal using a longer averaging window; add hysteresis, a clear gap between the scale-up and scale-down thresholds, so a small drop doesn't immediately trigger a scale-down; and configure stabilisation windows or cooldowns, for example in Kubernetes the HPA scale-down stabilisation window, which by default holds the highest recommendation over five minutes, so the system scales up quickly but down slowly. Control the rate of change with scaling policies limiting how many replicas can be removed per minute. Choose a better metric, such as requests per second or queue depth per replica, which reflects load more steadily than CPU. Check the interaction between autoscalers, like HPA and cluster autoscaler, and that readiness and warm-up time are accounted for, since new pods that take time to become ready cause overshoot. Also make sure requests and limits are sensible. Set minimum replicas to absorb typical load, and verify with a load test that reproduces the pattern.

Likely follow-ups

- Why scale up fast but scale down slowly?
- What does a stabilisation window do?

SYMPTOM

Replica count oscillates every few minutes — scale up, scale down, repeat — thrashing pods and disrupting in-flight work.

INVESTIGATION

what metric drives scaling, and how noisy is it?

what are the up/down thresholds – is there a gap (hysteresis) between them?

any stabilization window / cooldown configured?

Flapping happens when the scaling signal is noisy and the up/down thresholds are too close, so normal metric jitter crosses them repeatedly. Each scale action then changes per-pod load, which moves the metric again — a feedback loop. No cooldown or hysteresis lets it oscillate freely.

ROOT CAUSE

An unstable control loop: a noisy metric plus thresholds without a dead-band, and no stabilization window, so the autoscaler chases jitter and amplifies it.

FIX / MITIGATION

add a stabilization window / scale-down cooldown (slow down, esp. on the way down)

widen the gap between scale-up and scale-down thresholds (hysteresis)

smooth the input metric (averaging) and pick a stable signal

● PREVENTION

Tune the control loop like a control system: add hysteresis (separate up/down thresholds), stabilization windows (especially slower scale-down), and smoothed input metrics. Scale up fast for safety, down slowly for stability — and pick a metric that reflects sustained load, not momentary spikes.

Interview tip: Frame it as a control-systems problem: **hysteresis + stabilization windows + smoothed metric**. The asymmetry rule — scale up quickly, down slowly — is the practical heuristic interviewers like to hear.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S115 Capacity forecasting shows you'll run out of headroom in ~6 weeks at current growth. How do you respond?

Capacity & Scale

Spoken answer · ~81 s

Six weeks of headroom is a planning window, not a fire, so I respond deliberately. First, validate the forecast: is growth linear or accelerating, is it seasonal, and does a known launch or campaign change the slope? Is the growth organic, or a leak or inefficiency masquerading as growth, such as logs never rotated, data never archived, or a runaway job? Next, identify which resource saturates first and its lead time to expand, because a database storage expansion may be quick, while a quota increase, a hardware order, or a re-architecture may take longer than six weeks. Then choose among levers: reduce demand through cleanup, retention policies, compression, caching, and efficiency fixes; add capacity such as bigger nodes, more nodes, or more storage; or architect for scale, like sharding or tiering. I weigh cost and lead time. I put dates on it: a decision by a certain week, the work scheduled in the roadmap, and an alert threshold, for example at seventy percent, as a safety net. I communicate to stakeholders, including the cost impact. Afterwards, make capacity planning routine: track trends in dashboards, forecast regularly, set headroom targets, and include capacity in launch reviews.

Likely follow-ups

- How do you tell real growth from a leak?
- Why does lead time matter more than the time remaining?

● SYMPTOM

A trend analysis projects that a key resource (DB storage, node pool, a quota) hits 100% in roughly six weeks if growth continues.

● INVESTIGATION

validate the trend: linear or accelerating? seasonal?

which resource saturates first, and what's its lead time to expand?

is the growth organic, or a leak/inefficiency masquerading as growth?

Six weeks is a planning window, not a fire — if you act on it. The risk is ignoring the forecast until it's an incident. The other trap is assuming you must buy more capacity when the "growth" is actually waste (a leak, unindexed bloat, retention misconfig) that should be fixed instead.

● ROOT CAUSE

Demand is on track to exceed supply within the lead time required to add capacity. Whether the right answer is "add capacity" or "reclaim waste" depends on whether the growth is genuine.

● FIX / MITIGATION

first ask: is this real demand or reclaimable waste? (cleanup may buy months)

if real: provision ahead of the lead time, not at saturation

set a headroom threshold that triggers action with buffer to spare

● PREVENTION

Treat capacity as a continuous forecasting discipline: track utilization trends, define headroom thresholds that trigger action well before saturation (accounting for procurement/expansion lead time), and always check whether "growth" is demand or waste before spending. Plan capacity, don't react to it.

Interview tip: Two senior moves: **act within the lead time, with buffer**, and **distinguish real demand from reclaimable waste** before buying. Proactive headroom thresholds beat reacting at 100%.

Official docs

- [Google SRE Book: Software Engineering in SRE \(capacity planning\)](#)

S116 **You lose a node (or AZ) and the survivors get overloaded into a cascading failure. How should capacity have been planned?**

Capacity & Scale

 Spoken answer · ~82 s

If losing one node triggered a cascade, the cluster was running too close to capacity. When a node fails, its load redistributes to the survivors, and if they're already near their limits, they overload, fail in turn, and push more load onto fewer nodes. The core calculation: with N nodes each at X percent utilisation, after losing one the remaining N minus one nodes carry X times N over N minus one. Five nodes at eighty percent become one hundred percent after losing one, so there's no headroom. Capacity planning should be N plus one, or N plus two, and for AZ-level resilience, sizing so that losing a full AZ leaves the rest below a safe threshold; with three AZs each zone can run at no more than about sixty-six percent. Beyond sizing, add containment: load shedding and rate limiting so overload degrades gracefully, circuit breakers, timeouts, and bounded queues; autoscaling that reacts fast and has headroom to scale into; and avoiding retry storms with backoff and retry budgets. Test it by killing nodes in a game day or with chaos engineering and checking the survivors. Monitor utilisation against the failure-adjusted threshold and alert when headroom shrinks.

Likely follow-ups

- How much headroom is needed to survive losing one of N nodes?
- How do retries make a cascade worse?

 SYMPTOM

An instance fails and its load redistributes to the remaining nodes — which were already near capacity — pushing them over, causing more failures and a cascade.

 INVESTIGATION

`# what was per-node utilization BEFORE the failure?`

`# N nodes at X% — after losing one, can N-1 absorb the load?`

`# are there resource limits / load shedding to contain overload?`

Running every node near capacity leaves no slack to absorb a failure. When one dies, its share lands on survivors already at their limit, tipping them over — each failure increases load on the rest, a self-reinforcing cascade. The system had no $N+1$ (or $N+k$) redundancy headroom.

 ROOT CAUSE

Insufficient redundancy headroom. Capacity was planned for the happy path (all nodes healthy) rather than for graceful operation after losing one — so the first failure became many.

 FIX / MITIGATION

`# immediate: shed load / add capacity fast to break the cascade`

plan N+1 (or N+k): size so survivors absorb failures within healthy limits

add per-instance limits + load shedding so overload degrades, not cascades

● PREVENTION

Capacity-plan for failure: provision N+1/N+k so the loss of a node or AZ still leaves survivors within healthy utilization, spread across failure domains, and add backpressure/load shedding and circuit breakers so an overload sheds gracefully instead of cascading.

Interview tip: The phrase to use is **N+1 redundancy / failure headroom** — plan capacity for the post-failure state, not the all-healthy state. Pair it with load shedding and circuit breakers to stop cascades. AZ-level thinking earns extra credit.

Official docs

- [Google SRE Book: Handling Overload](#)
- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Using load shedding to avoid overload](#)

S117 Latency stays flat as load rises, then suddenly spikes off a cliff at a certain utilization. What's happening?

Capacity & Scale

Spoken answer · ~81 s

Flat latency and then a sudden cliff is the signature of queueing theory. As utilisation of a resource approaches one hundred percent, the waiting time grows non-linearly, roughly like one over one minus utilisation, so at fifty percent the queue is small, at ninety percent waiting is ten times service time, and at ninety-five percent it's twenty times, and once arrival rate exceeds service rate, the queue grows without bound. Small increases in load near the knee produce huge latency increases. To diagnose, I plot latency against utilisation to find the knee, and identify which resource is near saturation at that point: CPU, thread pool, connection pool, disk, or a lock, and whether a queue is growing unbounded past it. Practical consequences: don't run critical resources near saturation; keep target utilisation well below the knee, often fifty to seventy percent for latency-sensitive services, depending on variability. Add backpressure, bounded queues, and load shedding, so that overload produces fast errors instead of ever-growing latency. Scale before the knee using leading indicators such as queue depth, and not lagging ones. Load test beyond the expected peak to find the knee, and set alerts on saturation metrics.

Likely follow-ups

- Why is waiting time non-linear near full utilisation?
- What is load shedding and why does it help past the knee?

● SYMPTOM

Latency is steady and healthy as traffic grows — right up to a point, where it abruptly explodes with only a small further increase in load.

● INVESTIGATION

```
# plot latency vs utilization – where's the knee?
```

```
# what resource is near saturation at the cliff? (CPU, threads, connections, queue)
```

```
# is a queue growing unboundedly past that point?
```

Queueing theory: as utilization approaches 100%, wait time rises non-linearly — $\text{latency} \approx \text{service_time} / (1 - \text{utilization})$. Below the knee, added load barely moves latency; past it, small increases cause huge waits as queues build. The “cliff” is the system crossing into saturation.

● ROOT CAUSE

The system is operating too close to saturation. Latency vs utilization isn't linear — there's a knee near full utilization beyond which queues grow and latency blows up. Running near the cliff means a tiny load increase triggers the spike.

● FIX / MITIGATION

```
# back away from the cliff: add capacity to lower steady-state utilization
```

```
# cap concurrency / shed load to keep queues bounded
```

```
# target utilization with headroom (e.g. ~60–70%, not 95%)
```

● PREVENTION

Operate with utilization headroom below the latency knee, use admission control / concurrency limits so queues stay bounded under pressure, and monitor utilization against the cliff (not just current latency) — because a system that looks fine at 85% can fall off at 90%.

Interview tip: Cite the queueing-theory intuition — **$\text{latency} \approx \text{service_time} / (1 - \text{utilization})$** , with a knee near saturation. The takeaway: run with headroom (~60–70%), because the metric looks healthy right up until the cliff.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

 Spoken answer · ~79 s

Right-sizing a Kubernetes cluster safely means using measured data, applying changes gradually, and keeping a safety net, because past cuts caused OOMKills and throttling. First measure: compare requests and limits to actual usage per workload over a representative period, looking at P95 and P99 and peaks, not averages, including daily and weekly cycles. Tools like the Vertical Pod Autoscaler recommender in recommendation mode, Goldilocks, or metrics from Prometheus give suggestions without applying them. Memory is less forgiving than CPU: set memory requests close to peak usage plus headroom, since exceeding the limit kills the pod, while CPU is compressible and throttling can be handled with requests adjusted and limits relaxed. Then change in steps: start with the least critical, over-provisioned workloads, reduce by modest increments, say twenty to thirty percent, and watch OOMKills, throttling, restarts, and latency SLOs after each step, with a quick rollback. Use staged rollouts across environments. Protect critical services with Guaranteed or well-sized Burstable settings and PodDisruptionBudgets. Once requests are accurate, consolidate nodes with the cluster autoscaler, and consider spot capacity for tolerant workloads. Make it continuous with dashboards, regular reviews, and team-level cost visibility.

Likely follow-ups

- Why is memory riskier to cut than CPU?
- What does the VPA recommender give you?

 SYMPTOM

Cost review flags the cluster as heavily over-provisioned — nodes mostly idle — but past attempts to cut resources caused OOMKills and throttling, so the team padded everything “to be safe.”

 INVESTIGATION

```
kubectl top pods --all-namespaces # actual usage vs requests
```

```
# compare requests/limits to real p95/p99 usage per workload
```

```
# check the VPA recommender's suggestions if available
```

Requests reserve capacity (and drive scheduling/cost) whether or not pods use it; limits cap usage and cause OOMKill/CPU-throttle when hit. Padding requests “to be safe” strands capacity and inflates cost; setting limits too low causes the failures that scared the team. Right-sizing means matching reservations to real usage with a safety margin — not guessing high.

 ROOT CAUSE

Requests and limits were set by fear, not data — over-reserved for cost, with limits that bite when occasionally exercised. The gap between reserved and used is wasted spend; the failures came from limits set without measuring real peaks.

● FIX / MITIGATION

```
# set requests near observed p95 usage + margin; size limits above real peaks
```

```
# use VPA recommendations / historical usage to drive values
```

```
# roll out changes per-workload with monitoring, not cluster-wide at once
```

● PREVENTION

Right-size from observed usage (p95/p99 with headroom) rather than guesses, distinguish requests (scheduling/cost) from limits (the OOM/throttle ceiling), use the VPA recommender and bin-packing to reclaim waste, and change workloads incrementally with monitoring so you reclaim cost without reintroducing OOMKills.

Interview tip: Show you know the **requests vs limits** distinction cold — requests = reserved/cost/scheduling, limits = the OOMKill/throttle ceiling. “Size from real p95 usage + margin, roll out incrementally” is the safe, senior approach.

Official docs

- [Prometheus: Histograms and summaries](#)

S119 **Costs spiked and finance is asking why — you trace it to over-provisioning, not real demand. How do you bring it back down responsibly?**

Capacity & Scale

🗣 Spoken answer · ~82 s

When the bill rose from over-provisioning rather than demand, I bring it down responsibly: find the waste, remove it in order of safety, and prevent a rebound. First break the bill down by service, team, and tag to find where the delta came from. Then check utilisation of the expensive resources: are they used or idle? Look for zombie resources: unattached volumes, idle load balancers, old snapshots, unused IPs, forgotten dev and test environments, and oversized instances. Quick, low-risk wins first: delete clearly unused resources after confirming with owners, and schedule non-production environments to shut down out of hours. Then rightsize instances and databases based on measured utilisation, with gradual changes and monitoring, as in Kubernetes requests tuning. Then structural savings: reserved capacity or savings plans for steady baseline load, spot instances for tolerant workloads, storage tiering and lifecycle policies, and reducing data transfer costs. Responsibility means every cut is verified against SLOs, with a rollback, and done with the owning teams, not behind their backs. For prevention: enforce tagging, budgets and anomaly alerts, showback or chargeback so teams see their cost, and a regular FinOps review. Report savings to finance with before and after numbers.

Likely follow-ups

- Which cost savings are lowest risk to do first?
- How do showback and chargeback change behaviour?

● SYMPTOM

The monthly cloud bill jumped sharply. Investigation shows usage didn't grow proportionally — resources were over-provisioned, idle instances left running, and oversized everything.

● INVESTIGATION

break the bill down by service/team/tag – where did the delta come from?

utilization of the expensive resources: actually used, or idle?

any zombie resources (unattached disks, idle LBs, old environments)?

Cost spikes from over-provisioning come from idle/zombie resources (orphaned volumes, forgotten environments, idle load balancers), oversized instances, and on-demand pricing where committed or spot would do. The spend isn't buying reliability — it's pure waste, but cutting blindly risks an outage.

● ROOT CAUSE

Spend grew without corresponding demand: idle and oversized resources, plus no cost accountability (untagged, unattributed). The challenge is reclaiming it without trading a cost problem for a reliability problem.

● FIX / MITIGATION

kill zombie resources; right-size oversized instances from real usage

shift stable baseline to committed/reserved; bursty/fault-tolerant to spot

tag everything & attribute cost so each team sees & owns its spend

● PREVENTION

Build cost observability (tagging, per-team attribution, anomaly alerts) so spikes are caught and owned early, right-size and clean up zombies continuously, and match purchasing to workload shape (committed for baseline, spot for fault-tolerant, on-demand for spiky) — reclaiming waste while protecting reliability.

Interview tip: Frame it as FinOps: **visibility (tagging/attribution) + right-sizing + purchasing strategy**, all without sacrificing reliability. Naming zombie resources and committed-vs-spot-vs-on-demand shows practical cloud-cost fluency.

Official docs

- [Linux man-pages: fork\(2\)/wait\(2\)_\(zombies\)](#)

S120 **Capstone: a system handles load fine until concurrency climbs, then throughput collapses even though CPU isn't maxed. Diagnose using Little's Law.**

Capacity & Scale

🗣 Spoken answer · ~84 s

This is a Little's Law question: concurrency L equals throughput λ times latency W . When throughput collapses as concurrent requests rise while CPU, memory, and network look idle, something invisible is limiting concurrency or inflating latency. I measure the three terms: throughput, the number in flight, and latency. If throughput is capped, then latency grows in proportion to concurrency, meaning requests are queueing for a fixed-size resource rather than being processed. The usual hidden limiters: a connection pool or thread pool that is too small, so requests wait to get a connection or thread; lock contention or a single-threaded section serialising work; a database connection limit; downstream dependency limits; file descriptor limits; or rate limits. I find it by looking at pool wait times, thread dumps showing many threads blocked on the same lock or waiting for a connection, and queue depths. Beyond the limit, adding load only lengthens the queue and can cause timeouts and retries that make throughput fall, a collapse instead of a plateau. Fixes: size the pool using Little's Law, with required concurrency equal to target throughput times latency; remove the serialisation; add backpressure and load shedding; and scale the real bottleneck. Verify with a load test.

Likely follow-ups

- How do you use Little's Law to size a connection pool?
- Why does throughput fall rather than plateau past the limit?

● SYMPTOM

Throughput is healthy at low concurrency but *collapses* as concurrent requests rise — yet CPU, memory, and network all look under-utilized. Something invisible is the bottleneck.

● INVESTIGATION

measure the three terms: throughput (λ), concurrency (L), latency (W)

Little's Law: $L = \lambda \times W$ — which is fixed, which is exploding?

find the real limiter: connection pool, thread pool, locks, downstream cap

Little's Law ($L = \lambda \times W$) ties concurrency, throughput, and latency together. When throughput (λ) is capped by a hidden limited resource — a connection or thread pool, a lock, a downstream rate limit — adding concurrency (L) can't raise λ , so latency (W) must rise instead. Requests pile up waiting for the constrained resource while CPU sits idle, because the bottleneck isn't compute.

● ROOT CAUSE

A concurrency limit somewhere (pool size, lock, or downstream cap) bounds throughput. Past that point, more in-flight requests only inflate queue time — Little's Law says if λ is pinned and L grows, W must grow. The idle CPU proves the constraint is a logical resource, not hardware.

● FIX / MITIGATION

```
# find & raise the limiting resource (enlarge pool, reduce lock contention)
```

```
# OR cap admitted concurrency to keep the system below the collapse point
```

```
# add backpressure so excess load is shed, not queued into a meltdown
```

● PREVENTION

Reason about systems with Little's Law: identify the resource that actually caps throughput (pools, locks, downstream limits — rarely raw CPU), size it deliberately, and apply admission control/backpressure so concurrency stays in the stable regime instead of collapsing into unbounded queueing.

Interview tip: This is the capstone for a reason: invoking **Little's Law ($L = \lambda \times W$)** and reasoning that idle CPU + collapsing throughput means a *logical* bottleneck (pool/lock/downstream), not a hardware one, is exactly the structured systems-thinking senior interviewers are probing for. Close with backpressure/admission control as the stable-regime fix.

📖 Official docs

- [Little's law \(queueing theory reference\)](#)

S121 **Your checkout service latency doubles every day at exactly 14:00 and recovers by itself an hour later. How do you find out why?**

Incident Triage

🗣 Spoken answer · ~64 s

A problem that begins at exactly the same time every day and fixes itself points to something scheduled, not a code bug. I first overlay the latency window on everything that runs on a clock: cron jobs and Kubernetes cron jobs, backups, analytics exports, cache or token expiry, and partner traffic that is time-zone based. Then I look at which dependency's latency rises at the same moment, and at the database's top queries and lock waits during the window versus outside it. In this case I find a daily export running on the primary, competing for disk and locks with checkout transactions. The fix is moving the job to a read replica or off-peak, and throttling it with lower IO priority. To prevent recurrence, I keep an inventory of scheduled jobs against shared resources, enforce that heavy reads use replicas, and annotate dashboards with job schedules so a repeating pattern is recognised immediately.

Likely follow-ups

- What else repeats on a clock?
- Why use a read replica?

● SYMPTOM

p99 latency of checkout doubles daily from 14:00 to 15:00, then recovers with no human action. No deploys at that time.

● INVESTIGATION

```
# line up the window with scheduled work
```

```
crontab -l; kubectl get cronjob -A
```

```
# compare dependency latencies in that window
```

```
histogram_quantile(0.99, sum by (le, dep) (rate(dep_duration_bucket[5m])))
```

```
# check traffic mix and batch jobs on the DB
```

```
select query, calls from pg_stat_statements order by total_exec_time desc limit 5
```

● ROOT CAUSE

A daily analytics export job starts at 14:00 on the primary database and competes for IO and locks with checkout queries.

● FIX / MITIGATION

```
# move the export to a read replica
```

```
pg_dump --host replica ...
```

```
# or reschedule to off-peak and throttle IO
```

```
ionice -c3 nice -n19 ./export.sh
```

● PREVENTION

Inventory scheduled jobs per shared resource, run heavy reads on replicas, and add a latency SLO alert with a time-pattern annotation.

Interview tip: Anything that repeats on a clock is a schedule. Look for cron, batch, certificate or cache expiry before suspecting code.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S122 **One customer reports errors but your dashboards show a 99.99% success rate. How do you investigate a single-tenant problem?**

Incident Triage

 Spoken answer · ~64 s

Global success rate is dominated by the majority of traffic, so a single big tenant can be completely broken without moving it. I start by slicing metrics, logs and traces by tenant to confirm the pattern, then find their failed requests using the request or tenant ID. I compare what is different about them: data volume, features enabled, rate limits and quotas, region, client version and configuration. Often the answer is scale, such as a data set fifty times the median, so a query that is fine for others times out for them. I mitigate with a targeted fix, for example adding an index, increasing the timeout for that route or moving the tenant to bigger capacity, and keep the customer informed. For prevention, I add per-tenant SLIs for the largest customers, test with realistic whale-sized data, and set per-tenant query budgets and alerts for outliers so I see it before they call.

Likely follow-ups

- Why slice by tenant?
- What is a whale tenant?

 SYMPTOM

A single large customer sees failures. Global error rate is normal. Support has a request ID or tenant ID.

 INVESTIGATION

```
# slice by tenant, not globally
```

```
sum by (tenant) (rate(http_requests_total{code=~"5.."}[10m]))
```

```
# find their traces and logs
```

```
grep tenant_id=acme app.log | grep -i error | tail
```

```
# compare config, data size and quotas for that tenant
```

```
select count(*) from orders where tenant_id='acme'
```

 ROOT CAUSE

The tenant has a data volume 50x the median, so a query with no index on that tenant partition times out only for them.

 FIX / MITIGATION

```
# immediate: raise timeout or add index concurrently
```

```
create index concurrently on orders (tenant_id, created_at);
```

```
# route their traffic to a larger pool if needed
```

● PREVENTION

Track SLIs per tenant tier, alert on outliers, load test with whale-sized tenants, and add per-tenant query budgets.

Interview tip: Averages hide the biggest customers. Always be able to slice by tenant, region and version.

Official docs

- [Google SRE Book: Embracing Risk \(capacity & availability math\)](#)
- [Grafana documentation](#)

S123 **An incident is resolved, but the same alert returns three times the next day with a different apparent cause each time. What is going on?**

Incident Triage

Spoken answer · ~63 s

If the same symptom recurs with a new explanation every time, I suspect we are treating triggers and missing the underlying fragility. I build one timeline across all occurrences and look for what they share: a resource near its limit, a common dependency, a time pattern, or a recent change that altered a margin. For example, a database connection pool running at ninety-five percent has no slack, so a small traffic increase, a slow query or a deployment can each push it over, and each looks like a different cause. I confirm with saturation metrics at the time of each incident, then fix the constraint: resize the pool, introduce a pooler, find the leak that holds connections, and add an alert on saturation so I get warned before failure. I also record in the postmortem that the individual triggers were symptoms, and track recurrence so the problem is addressed systematically.

Likely follow-ups

- Why look for shared constraints?
- What alert catches this earlier?

● SYMPTOM

Repeat incidents, each blamed on a different cause, each 'fixed' individually. The alert keeps returning.

● INVESTIGATION

```
# build a combined timeline of all four occurrences
```

```
# look for a common precursor: change, load, resource
```

```
kubectl get events --sort-by=.lastTimestamp | head -50
```

```
# check shared resources: connection pool, node, queue
```

```
show max_connections; select count(*) from pg_stat_activity;
```

● ROOT CAUSE

A shared resource (connection pool) is near its limit, so any minor trigger tips it over. The 'different causes' were just different triggers.

● FIX / MITIGATION

```
# raise and right-size the pool, add pgbouncer
```

```
# add saturation alert at 80% pool use
```

```
# fix the leaking client that never returns connections
```

● PREVENTION

Alert on saturation, not just failures; after repeated incidents, look for a common underlying constraint instead of treating each trigger separately.

Interview tip: When causes keep changing, you are probably fixing triggers. Look for the shared constraint that makes the system fragile.

Official docs

- [PgBouncer](#)

S124 You are on call and two unrelated alerts fire at the same time on different services. How do you prioritise?

Incident Triage

Spoken answer · ~59 s

With two alerts, I rank them by user and business impact and urgency. Payments errors affect revenue and customers now, while a failing internal report can usually wait unless it feeds a regulatory deadline. I quickly check whether the two are related, looking at recent changes and shared dependencies, because a common cause changes the approach. I then act on the most severe one, declare an incident if needed, and bring in help: I page the secondary on-call or another team to handle the second alert so I am not context-switching. The less severe item is acknowledged, with a note of its status and an owner, so it is not lost. I communicate the priorities in the incident channel so stakeholders know what is happening. Later I check that severities and routing were right, and adjust alert metadata for faster triage.

Likely follow-ups

- How do you rank alerts?
- When do you declare an incident?

SYMPTOM

Alert A: payments error rate rising. Alert B: internal reporting job failing. Only one on-call.

INVESTIGATION

```
# impact first
```

```
check SLO burn and customer-facing impact for each
```

```
# are they related? shared dependency or change?
```

```
git log --since='30 min ago'; check shared DB/queue health
```

```
# can anyone else take one?
```

```
page secondary, declare incident for the highest-impact
```

ROOT CAUSE

They were unrelated: payments degradation from a provider issue, reporting failure from an expired credential.

FIX / MITIGATION

```
# payments first: failover to secondary provider
```

```
# ask secondary on-call to take reporting
```

```
# log reporting as ticket with owner and ETA
```

● PREVENTION

Define severities, keep a secondary on-call, and tag alerts with business impact so triage is quick under pressure.

Interview tip: Triage by user and business impact, not by who shouted first. Delegate early.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S125 **A new engineer deployed a hotfix directly to production, bypassing the pipeline, and now things are inconsistent. How do you recover and fix the process?**

Incident Triage

Spoken answer · ~62 s

My first step is to establish what is running where: compare versions or file hashes across hosts and check the audit log for who changed what, so I understand the drift. To recover, I redeploy a known version from the pipeline to every host so all are consistent, and then submit the hotfix as a proper change through code review and tests so it is tracked and repeatable. I talk to the engineer without blame to understand why they bypassed the pipeline: was it too slow in an emergency, did they lack access to a hotfix path, or did nobody explain? The systemic fixes are removing direct write access to production, creating an expedited hotfix lane with essential checks, using immutable deployments so manual edits are impossible or detected, and drift detection that alerts when running state differs from the declared one. Training and onboarding then reinforce the process.

Likely follow-ups

- Why avoid blaming the engineer?
- How do you detect drift?

● SYMPTOM

Manual change on some servers only. Behaviour differs per host. Git and prod no longer match.

● INVESTIGATION

```
# find drift
```

```
diff <(ssh h1 sha256sum /app/bin) <(ssh h2 sha256sum /app/bin)
```

```
kubectl get deploy api -o jsonpath='{.spec.template.spec.containers[*].image}'
```

```
# who and what changed
```

```
grep sudo /var/log/auth.log
```

● ROOT CAUSE

A manual binary swap on three of ten hosts left mixed versions, and the change never went through review or tests.

● FIX / MITIGATION

```
# bring all hosts back to the pipeline-built version
```

```
./deploy.sh --version 2026.10.3 --all
```

```
# add the hotfix properly via PR + CI
```

```
# document the change in the incident log
```

● PREVENTION

Remove direct prod write access, provide a fast hotfix lane in the pipeline, use immutable infrastructure, and treat the event as a systems problem, not a personal one.

Interview tip: People bypass the pipeline when it is slow or when they do not trust it. Make the right way the fast way.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S126 **A customer-facing feature breaks only when a specific third-party script or ad blocker is present. How do you diagnose a client-side issue from the server side?**

Incident Triage

🗣 Spoken answer · ~61 s

When server logs and dashboards look clean but users see a broken page, the problem is likely in the browser. I check real user monitoring and client error reports, filtering by browser, version and extension patterns, and look for the common factor among affected users. Then I reproduce with the same environment, for example a browser with the blocker enabled, and watch the console and network tab. Here the blocker prevents an analytics script loading, and the page's code assumed it exists and threw an unhandled exception, stopping rendering. The fix is defensive: wrap third-party initialisation in error handling, load such scripts asynchronously so they cannot block the page, and deploy the frontend update, verifying with the client error rate. For prevention, I include real user monitoring and error tracking in the standard stack, add synthetic checks with several browsers, and treat third-party scripts as unreliable dependencies.

Likely follow-ups

- Why is RUM needed?
- How should third-party scripts be loaded?

● SYMPTOM

Support reports a page blank for some users. Server logs are clean, API returns 200.

● INVESTIGATION

```
# check RUM / browser error reporting
```

```
filter JS errors by browser, extension, URL
```

```
# reproduce with the same browser profile
```

```
# compare requests with and without the extension
```

```
grep -c 'Failed to load resource' rum.log
```

● ROOT CAUSE

A blocking extension blocks a first-party analytics script whose failure throws an unhandled exception that halts rendering.

● FIX / MITIGATION

```
# wrap analytics init in try/catch, load async
```

```
# deploy frontend fix, verify via RUM error rate
```

● PREVENTION

Treat third-party scripts as untrusted, load them non-blocking, add RUM error tracking and synthetic checks across browsers.

Interview tip: Server-side 200s do not mean the user's page works. You need real user monitoring.

📖 Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S127 The database is healthy and the app is healthy, but a specific API endpoint returns 504 after exactly 30 seconds. Where do you look?

Incident Triage

🗣 Spoken answer · ~61 s

A failure at exactly thirty seconds is a signature of a configured timeout rather than random slowness. I list every layer between the client and the code, the CDN, load balancer, ingress, proxy, application server and client library, and find which has a thirty second limit. Then I trace the endpoint to see why it takes longer than that: here it generates a report that needs forty-five seconds. The quick mitigation is raising the timeout on that route at the ingress and load balancer, keeping the layers' timeouts ordered so that the inner ones expire before the outer ones. The proper fix is redesigning the operation to be asynchronous: the request starts a job and returns an identifier immediately, the client polls or receives a notification. Prevention includes documenting timeouts across layers, alerting on requests near the limit, and discouraging long synchronous operations in API design.

Likely follow-ups

- Why do round numbers matter?
- Why make long operations async?

● SYMPTOM

504 Gateway Timeout on one endpoint at exactly 30s. Other endpoints fine.

● INVESTIGATION

```
# a round number means a configured timeout
```

```
grep -R 'timeout' nginx.conf ingress.yaml
```

```
kubectl get ingress api -o yaml | grep -i timeout
```

```
# find the slow step inside that endpoint
```

```
trace the request; check outbound calls and locks
```

● ROOT CAUSE

The endpoint calls a slow report generator that takes 45s; the load balancer idle timeout of 30s cuts it before the app finishes.

● FIX / MITIGATION

```
# short term: raise LB/ingress timeout for that route
```

```
nginx.ingress.kubernetes.io/proxy-read-timeout: "60"
```

```
# real fix: make it async with a job id and polling
```

● PREVENTION

Set timeouts consistently across layers, keep synchronous requests short, and design long operations as asynchronous jobs.

Interview tip: Exact round-number durations almost always mean a timeout setting somewhere in the path.

Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Google SRE Book: Effective Troubleshooting](#)

S128 After a datacenter power event all servers came back, but some services are in a bad state. In what order do you bring things back and verify?

Incident Triage

Spoken answer · ~60 s

After a mass restart, I avoid starting everything at once. I work bottom-up through the dependency layers: first the network, DNS and time synchronisation, because many things fail strangely without them; then storage, databases and message queues, verifying file system checks, cluster quorum and replication status; then identity and secrets services; then the core application services; and finally edge components like load balancers and caches. At each layer I confirm health before moving on. Services that started too early and cached failures, such as an empty DNS answer, get restarted. I check data integrity carefully, particularly any replica that may be stale or corrupted, and resynchronise it from the primary if necessary. Afterwards I run synthetic end-to-end checks. To prevent chaos next time, I document and automate the startup order, build readiness checks and retries with backoff into services, and practise cold-start drills.

Likely follow-ups

- What starts first?
- Why restart services that cached failures?

● SYMPTOM

Mass restart after power loss. Services started in random order, some failing on missing dependencies.

● INVESTIGATION

```
# build the dependency order
```

```
1. network, DNS, NTP 2. storage/DB/queues 3. identity/secrets 4. core services 5. edge/LB
```

```
# verify each layer before the next
```

```
systemctl --failed; kubectl get pods -A | grep -v Running
```

```
# check data integrity
```

```
fsck results, replication status, WAL/replay
```

● ROOT CAUSE

Services started before DNS and the database quorum were ready and cached failures; the database had a stale replica.

● FIX / MITIGATION

```
# restart in dependency order, validate each tier
```

```
# resync stale replica from primary
```

```
# restart services that cached bad state
```

● PREVENTION

Document and automate startup ordering, add readiness gates and retries with backoff, test cold-start recovery in drills.

Interview tip: Bring up foundation services first and verify each layer. Starting everything at once creates cascading confusion.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S129 A Deployment shows all pods Running but the Service has no endpoints. How do you debug it?

Kubernetes

 Spoken answer · ~63 s

A Service routes to endpoints, which Kubernetes builds from pods that match the selector and are Ready. If the endpoint list is empty while pods are running, one of those two conditions fails. I compare the Service selector with the pod labels using `show-labels`; a typo or a changed label after a refactor is very common. If the labels match, I check readiness: describe the pod to see whether the readiness probe is failing, for instance because the path, port or required dependency is wrong, which keeps the pod out of the endpoints even though the container runs. I also check that the target port matches the container port and that the namespace is correct. The fix is correcting the selector or the probe, then verifying that the endpoints appear. To prevent this, I generate labels and selectors from shared templates and add a deployment check that endpoints are not empty.

Likely follow-ups

- What makes a pod appear in endpoints?
- How do you verify the fix?

SYMPTOM

`kubectl get pods` shows Running, but requests to the Service fail with connection refused. `kubectl get endpoints` shows none.

INVESTIGATION

```
kubectl get svc api -o yaml | grep -A3 selector
```

```
kubectl get pods --show-labels
```

```
kubectl get endpoints api
```

```
kubectl describe pod api-xyz | grep -i -A5 'ready\|conditions'
```

ROOT CAUSE

The Service selector uses `app=api` but pods are labelled `app=api-v2`, or the pods are Running but not Ready because the readiness probe fails.

FIX / MITIGATION

```
# fix labels or selector
```

```
kubectl patch svc api -p '{"spec":{"selector":{"app":"api-v2"}}}'
```

```
# or fix the readiness probe path/port
```

```
kubectl rollout status deploy/api
```

● PREVENTION

Use Helm templates that derive selectors and labels from one helper, and add a post-deploy check that endpoints are non-empty.

Interview tip: Endpoints only include pods that are Ready and match the selector. Check both conditions.

Official docs

- [Kubernetes: Debug Services](#)

S130 A StatefulSet pod won't start because its PersistentVolumeClaim is bound to a volume in a different availability zone. What happened and how do you fix it?

Kubernetes

Spoken answer · ~66 s

Block volumes such as EBS exist in a single availability zone and can only attach to nodes in that zone. The pod's claim is already bound to a volume in zone a, so the scheduler may only place the pod on nodes in zone a. If there are none with enough capacity, or they are cordoned or tainted, the pod stays pending with a node affinity conflict. The quick fix is to restore capacity in zone a: scale that node group, remove the blocking taint, or free resources. If zone a is unavailable, I can restore the data from a snapshot into a new volume in another zone and recreate the claim. Longer term, I configure the storage class with wait-for-first-consumer binding so volumes are created in the zone where the pod is scheduled, maintain node groups in every zone with enough headroom, and consider replication at the application layer so the loss of one zone is survivable.

Likely follow-ups

- Why can't volumes cross zones?
- What does WaitForFirstConsumer do?

● SYMPTOM

Pod Pending with 'volume node affinity conflict' or 'had volume node affinity conflict'.

● INVESTIGATION

```
kubectl describe pod db-0 | tail -20
```

```
kubectl get pv,pvc -o wide
```

```
kubectl get pv <name> -o yaml | grep -A8 nodeAffinity
```

```
kubectl get nodes -L topology.kubernetes.io/zone
```

● ROOT CAUSE

The EBS volume lives in zone a, but the only schedulable nodes (due to capacity or taints) are in zone b; block volumes cannot cross zones.

● FIX / MITIGATION

```
# add capacity in zone a
```

```
# or restore from snapshot into zone b
```

```
aws ec2 create-volume --snapshot-id snap-123 --availability-zone eu-west-1b
```

```
# use volumeBindingMode: WaitForFirstConsumer
```

● PREVENTION

Use WaitForFirstConsumer storage classes, spread node groups per zone, and keep topology spread constraints and capacity in every zone.

Interview tip: Block storage is zonal. Scheduling must follow the data, not the other way around.

Official docs

- [Kubernetes: Persistent Volumes](#)
- [Kubernetes: StatefulSets](#)

S131 After upgrading the cluster, many pods fail with 'no matches for kind' or deprecated API errors during deploy. How do you recover and prevent this?

Kubernetes

 Spoken answer · ~58 s

Kubernetes removes deprecated API versions on a published schedule, and manifests or stored Helm release data that still use them break after an upgrade. To recover, I identify which resources use removed versions, using the API server's deprecated request metrics and static scanners like pluto against manifests and Helm releases. I update manifests to the supported versions, for example the stable networking API for ingress, and for existing Helm releases I use the map-deprecated-APIs plugin to rewrite stored release manifests so that upgrades work. Then I redeploy and validate that workloads are healthy. To prevent it, my upgrade process includes scanning for deprecated and soon-to-be-removed APIs before upgrading, reading release notes, upgrading a staging cluster first with the real manifests, and keeping charts and operators updated. I also monitor the deprecated API usage metric continuously, not only at upgrade time.

Likely follow-ups

- How do you find deprecated APIs?
- Why test upgrades in staging first?

SYMPTOM

kubectl apply fails: 'no matches for kind Ingress in version extensions/v1beta1'. Helm releases cannot upgrade.

INVESTIGATION

```
kubectl api-resources | grep -i ingress
```

```
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis
```

```
pluto detect-files -d manifests/
```

```
helm mapkubeapis release --dry-run
```

ROOT CAUSE

Manifests and Helm release history reference API versions removed in the new Kubernetes version.

FIX / MITIGATION

```
# update manifests to supported API versions
```

```
apiVersion: networking.k8s.io/v1
```

```
helm mapkubeapis myrelease
```

```
# redeploy and verify
```

● PREVENTION

Scan for deprecated APIs before every upgrade, read release notes, test upgrades in staging and keep charts current.

Interview tip: Deprecated APIs are announced many releases ahead. Scanners such as pluto or kubent find them in minutes.

Official docs

- [Kubernetes: Upgrading kubeadm clusters](#)
- [Kubernetes: Deprecated API migration guide](#)

S132 Your cluster DNS (CoreDNS) is overloaded and pods see intermittent 'no such host' errors under load. What do you check and change?

Kubernetes

Spoken answer · ~60 s

DNS problems in Kubernetes under load often come from query amplification and insufficient capacity. By default pods have the ndots option set to five, so a name with fewer than five dots, such as api.example.com, is first tried with each search domain appended, producing several failed queries before the correct one, and in a busy cluster that multiplies traffic onto CoreDNS. I check CoreDNS CPU and logs, replica count, and the resolv.conf of the application pods. Mitigations are lowering ndots for workloads that mostly query external names, or using fully qualified names with a trailing dot, scaling the CoreDNS replicas, and deploying node-local DNS caching so most queries are answered on the node without crossing the network. I add autoscaling for CoreDNS proportional to cluster size, and monitoring for query rate, latency and SERVFAIL responses, so I see saturation before users see failures.

Likely follow-ups

- What does ndots do?
- What is NodeLocal DNSCache?

● SYMPTOM

Random resolution failures and slow lookups, especially during traffic peaks. CoreDNS pods at high CPU.

● INVESTIGATION

```
kubectl top pods -n kube-system -l k8s-app=kube-dns
```

```
kubectl logs -n kube-system -l k8s-app=kube-dns | tail
```

```
kubectl exec app -- cat /etc/resolv.conf # ndots:5?
```

```
kubectl get cm coredns -n kube-system -o yaml
```

● ROOT CAUSE

ndots:5 makes every external lookup try multiple search domains first, multiplying queries; CoreDNS has too few replicas and no node-local cache.

● FIX / MITIGATION

```
# lower ndots or use FQDNs with trailing dot
```

```
dnsConfig: options: [{name: ndots, value: "2"}]
```

```
kubectl scale deploy coredns -n kube-system --replicas=6
```

```
# deploy NodeLocal DNSCache
```

● PREVENTION

Run node-local DNS caching, autoscale CoreDNS with cluster-proportional-autoscaler, and monitor DNS latency and error rate.

Interview tip: With ndots 5, one lookup for api.example.com can generate five or more queries. Use trailing dots for external names.

Official docs

- [Kubernetes: Debugging DNS resolution](#)
- [Kubernetes: DNS for Services and Pods](#)
- [Google SRE Book: Handling Overload](#)

S133 A PodDisruptionBudget blocks a node drain during a cluster upgrade and the upgrade hangs. How do you handle it safely?

Kubernetes

🗣 Spoken answer · ~60 s

A PodDisruptionBudget protects availability by limiting voluntary evictions, but a badly set one can block maintenance. Describing the PDB shows how many disruptions are currently allowed. If minimum available equals the replica count, or one pod is already unhealthy, the allowed disruptions are zero and drain waits indefinitely. The safe response is not to force-delete pods, which would defeat the protection and might cause an outage. Instead I increase the replica count temporarily so that evicting one pod keeps the minimum, or fix the unhealthy pod so the budget recovers, then continue the drain. I never bypass the budget on a critical service without agreement. For the long term, I define PDBs using max unavailable of one, ensure at least two or three replicas for critical services, and include drain tests in the upgrade checklist, so these conflicts are discovered before the maintenance window.

Likely follow-ups

- What is allowedDisruptions?
- Why not delete the pods directly?

🚧 PDB blocks drain



● SYMPTOM

kubectl drain waits forever: 'Cannot evict pod as it would violate the pod's disruption budget.'

● INVESTIGATION

```
kubectl get pdb -A
```

```
kubectl describe pdb api -n prod
```

```
kubectl get pods -n prod -l app=api -o wide
```

```
kubectl get deploy api -n prod
```

● ROOT CAUSE

PDB requires minAvailable 2 but the deployment has only 2 replicas, or an unhealthy pod already uses up the budget, so no pod can be evicted.

● FIX / MITIGATION

```
# scale up temporarily so eviction is allowed
```

```
kubectl scale deploy/api --replicas=4
```

```
# fix the unhealthy replica
```

```
# long term: maxUnavailable: 1 with enough replicas
```

● PREVENTION

Set PDBs relative to replica counts, never minAvailable equal to replicas, and test drains in staging before production upgrades.

Interview tip: A PDB that allows zero disruptions makes the node undrainable. Check allowedDisruptions.

Official docs

- [Kubernetes: Safely drain a node](#)
- [Kubernetes: Upgrading kubeadm clusters](#)

S134 Ingress returns 502 for a few seconds after every deployment, even with rolling updates. What causes it and what do you fix?

Kubernetes

Spoken answer · ~66 s

The pattern is a short error burst at each rollout, which means that old pods are going away while traffic is still being sent to them. When Kubernetes terminates a pod, it removes it from endpoints and sends a termination signal at the same time, but the ingress controller and load balancers learn about the endpoint change a bit later. If the application exits immediately on the signal, requests in that window fail. The fix has several parts: add a pre-stop sleep of a few seconds so the pod keeps serving while the removal propagates, make the app handle the signal by stopping new connections and finishing in-flight requests, set the termination grace period to cover that, and configure the rolling update with no unavailable pods and some surge so capacity stays. I validate by running a load test during a rollout and checking that the error rate stays flat, and bake the settings into the base chart.

Likely follow-ups

- Why a preStop sleep?
- How do you test the fix?

● SYMPTOM

Short bursts of 502/503 on each rollout, then normal. Readiness probes pass.

● INVESTIGATION

```
kubectl get deploy api -o yaml | grep -A8 strategy
```

```
kubectl get deploy api -o yaml | grep -B2 -A10 lifecycle
```

```
kubectl logs ingress-nginx-controller | grep ' 502 '
```

```
# do old pods get SIGTERM before endpoints are removed?
```

● ROOT CAUSE

Pods receive SIGTERM and exit immediately while the ingress still routes to them, because endpoint removal propagates asynchronously.

● FIX / MITIGATION

```
lifecycle: preStop: exec: command: ["sleep","15"]
```

```
terminationGracePeriodSeconds: 45
```

```
# app: stop accepting new connections after SIGTERM, drain in-flight
```

```
maxUnavailable: 0, maxSurge: 25%
```

● PREVENTION

Standardise graceful shutdown in base charts, test rollouts under load in CI and monitor 5xx during deployments.

Interview tip: Endpoint removal and SIGTERM race each other. A preStop sleep gives the proxies time to update.

Official docs

- [Kubernetes: Ingress](#)
- [Kubernetes: Deployments](#)
- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)

S135 A namespace hits its ResourceQuota and new deployments fail with 'exceeded quota'. How do you resolve it without breaking other teams?

Kubernetes

🗣 Spoken answer · ~59 s

Resource quotas limit the total requests and limits a namespace can claim, and pods that would exceed it are rejected at creation. I look at quota consumption against actual usage. A common situation is that teams copy generous requests into every service, so the sum of requests hits the quota while real usage is far lower. The first fix is to right-size: use observed usage and vertical autoscaler recommendations to lower requests on over-provisioned workloads, delete unused deployments and old jobs. If the team really needs more capacity, they raise a request through the capacity planning process, so the cluster has enough nodes. I avoid simply increasing the quota without checking the cluster headroom, as that can starve other teams. To prevent recurrence, I add default requests through limit ranges, dashboards showing requests versus usage per namespace, and a periodic review.

Likely follow-ups

- Do quotas count usage or requests?
- What does a LimitRange do?

SYMPTOM

Pods not created: forbidden: exceeded quota: requests.cpu. ReplicaSet events show failures.

INVESTIGATION

```
kubectl describe quota -n team-a
```

```
kubectl top pods -n team-a --sort-by=cpu
```

```
kubectl get pods -n team-a -o custom-  
columns=NAME:.metadata.name,REQ:.spec.containers[*].resources.requests.cpu
```

```
kubectl get replicaset -n team-a
```

ROOT CAUSE

Over-requested CPU on idle services consumes the quota; real usage is a fraction of requests.

FIX / MITIGATION

```
# right-size requests using VPA recommendations
```

```
# remove unused workloads
```

```
# if justified, raise quota through the capacity process
```

● PREVENTION

Review requests regularly, use VPA in recommendation mode, set LimitRanges with sensible defaults and track quota utilisation.

Interview tip: Quota counts requests, not usage. Inflated requests waste shared cluster capacity.

📖 Official docs

- [Kubernetes: Resource Quotas](#)

S136 **A critical controller or operator pod keeps getting evicted and restarted because the node is under memory pressure. How do you protect system-critical pods?**

Kubernetes

🗣 Spoken answer · ~58 s

When a node runs low on memory, the kubelet evicts pods, choosing by quality-of-service class and then by usage relative to requests. A controller pod with no requests or limits is classified best effort and goes first, even though it is critical to the platform. I check the pod's QoS class and the node conditions. To protect it, I set resource requests equal to limits so it is in the guaranteed class, and assign a high priority class so the scheduler and eviction logic favour it over ordinary workloads. At the same time I address the cause of memory pressure: find workloads that run without limits and consume too much, enforce requests and limits through admission policy, and reserve memory for the operating system and kubelet using system-reserved settings. Cluster autoscaling may add nodes if overall capacity is short.

Likely follow-ups

- What is the QoS eviction order?
- What does a priority class do?

● SYMPTOM

Operator pod evicted repeatedly; custom resources stop reconciling; node condition MemoryPressure true.

● INVESTIGATION

```
kubectl describe node n1 | grep -A8 Conditions
```

```
kubectl get pod operator -o jsonpath='{.status.qosClass}'
```

```
kubectl get events --field-selector reason=Evicted
```

```
kubectl top node n1
```

● ROOT CAUSE

The operator runs as BestEffort with no requests, so it is the first to be evicted when memory is short.

● FIX / MITIGATION

```
# set requests=limits for Guaranteed QoS
```

```
resources: requests: {cpu: 200m, memory: 256Mi} limits: {cpu: 200m, memory: 256Mi}
```

```
priorityClassName: system-cluster-critical
```

```
# fix the noisy neighbours without limits
```

● PREVENTION

Require resource requests on all workloads, use priority classes for platform components and reserve resources for system daemons.

Interview tip: QoS class decides eviction order: BestEffort first, then Burstable, Guaranteed last.

Official docs

- [Kubernetes: Node-pressure eviction](#)

S137 A cronjob in Kubernetes silently stops running, no recent jobs created. How do you find out why?

Kubernetes

Spoken answer · ~58 s

A cronjob that stops producing jobs generates no logs or errors, so the first sign is usually a business process noticing missing output. I look at the cronjob status: is it suspended, when was the last schedule, what is the concurrency policy. A frequent culprit is concurrency policy forbid combined with a previous job that hangs forever, so the controller refuses to start another. Another is missing too many schedules during downtime, where the controller gives up after a number of missed starts unless a starting deadline is configured. I delete the stuck job, which allows new ones to run, and add an active deadline so jobs cannot hang indefinitely. For prevention, I add an alert on the time since the last successful completion, using metrics from kube-state-metrics, set history limits, and add timeouts, so silence becomes a visible signal.

Likely follow-ups

- Why does silence matter for jobs?
- What does startingDeadlineSeconds do?

● SYMPTOM

CronJob last schedule was days ago. No errors in app logs because no pods are created.

● INVESTIGATION

```
kubectl get cronjob report -o yaml | grep -E  
'suspend|lastScheduleTime|schedule|concurrencyPolicy|startingDeadlineSeconds'
```

```
kubectl get jobs --sort-by=.metadata.creationTimestamp
```

```
kubectl describe cronjob report
```

```
kubectl get events | grep -i report
```

● ROOT CAUSE

A previous job hung and concurrencyPolicy: Forbid prevents new runs; or more than 100 missed schedules triggered the controller to stop scheduling.

● FIX / MITIGATION

```
# delete the stuck job
```

```
kubectl delete job report-28345
```

```
# set activeDeadlineSeconds on the job
```

```
# set startingDeadlineSeconds
```

```
# alert on time since last success
```

● PREVENTION

Alert on time since the last successful run, set deadlines and history limits, and avoid Forbid without a timeout.

Interview tip: Monitor for absence. A job that no longer runs generates no errors, only silence.

Official docs

- [Kubernetes: CronJob](#)

 Spoken answer · ~59 s

The pattern of a slow first request after idle time is a cold start: Lambda must create an execution environment, download the code, start the runtime and run initialisation code before handling the request. I confirm with the init duration field in the logs. Contributing factors are a large deployment package, heavy imports, creating many SDK clients or fetching configuration during init, and in some cases VPC attachment. Improvements start with trimming the package and dependencies, lazy-loading rarely used modules, and moving expensive initialisation out of the critical path or caching it. For latency-sensitive endpoints, I configure provisioned concurrency to keep a number of environments warm, accepting the extra cost, or use SnapStart where supported. I monitor the init duration percentiles and cold start rate, and decide per function whether the cost of keeping it warm is justified by the user impact.

Likely follow-ups

- How do you confirm a cold start?
- What is provisioned concurrency?

 SYMPTOM

Occasional very slow first invocation (several seconds), then normal. Users notice sporadic delays.

 INVESTIGATION

```
aws logs filter-log-events --log-group-name /aws/lambda/fn --filter-pattern 'Init Duration'
```

```
aws lambda get-function-configuration --function-name fn
```

```
# look at package size, runtime, VPC config, init code
```

```
# CloudWatch: ConcurrentExecutions, Duration p99
```

 ROOT CAUSE

Cold starts: large package, heavy initialisation (SDK clients, config fetch) and VPC networking add several seconds when a new execution environment starts.

 FIX / MITIGATION

```
# reduce package, lazy-load, initialise clients outside handler
```

```
aws lambda put-provisioned-concurrency-config --function-name fn --qualifier prod --provisioned-concurrent-executions 5
```

```
# consider SnapStart for supported runtimes
```

● PREVENTION

Track init duration, set provisioned concurrency for latency-sensitive paths, keep functions small and avoid blocking init.

Interview tip: Provisioned concurrency trades cost for predictable latency. Use it only where cold starts hurt users.

📖 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [AWS Lambda Developer Guide](#)

S139 A cloud load balancer health check marks all targets unhealthy right after a security group change, but the app is fine. How do you recover?

Cloud / AWS

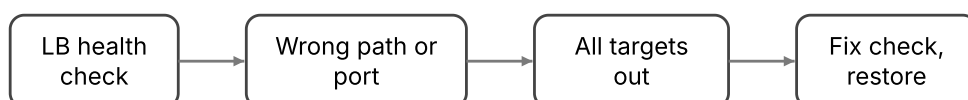
🗣 Spoken answer · ~62 s

If the application responds locally but the load balancer reports all targets unhealthy right after a network change, the health check traffic is probably blocked. I check the target health reason, then compare the instance security group rules to what the load balancer needs: the health check port must allow inbound traffic from the load balancer's security group or subnets. CloudTrail shows the recent security group modification, who made it and what was removed. The fix is re-adding the rule, preferably by referencing the load balancer's security group rather than IP ranges, then confirming the targets turn healthy and traffic resumes. I also check network ACLs and route tables in case they were involved. Preventing a repeat means managing security groups in infrastructure as code with peer review and plan output, and adding an automated connectivity test after changes, plus alerts on sudden drops in healthy host count.

Likely follow-ups

- Why reference security groups by ID?
- What does CloudTrail show?

🚩 All targets unhealthy



● SYMPTOM

All targets unhealthy, 503 from the load balancer. Instances respond locally. Change made 5 minutes ago.

● INVESTIGATION

```
aws elbv2 describe-target-health --target-group-arn $TG
```

```
aws ec2 describe-security-groups --group-ids sg-app
```

```
# can LB subnet reach the health check port?
```

```
aws cloudtrail lookup-events --lookup-attributes  
AttributeKey=EventName,AttributeValue=RevokeSecurityGroupIngress
```

● ROOT CAUSE

The change removed the inbound rule allowing the load balancer's security group on the health check port.

● FIX / MITIGATION

```
aws ec2 authorize-security-group-ingress --group-id sg-app --protocol tcp --port 8080 --  
source-group sg-alb
```

```
# verify targets return to healthy
```

```
aws elbv2 describe-target-health --target-group-arn $TG
```

● PREVENTION

Manage security groups as code with review, reference groups by ID instead of CIDR, and validate connectivity in a post-change check.

Interview tip: CloudTrail tells you exactly which rule changed and who changed it, which speeds up rollback.

Official docs

- [Cloudflare Learning: Load balancing](#)
- [Amazon VPC: Security groups](#)
- [Cloud Load Balancing](#)

S140 Your cloud storage bucket costs spiked because of millions of tiny requests and egress. How do you find the source and stop it?

Cloud / AWS

🗣 Spoken answer · ~58 s

I start with the billing breakdown by usage type to see which component grew: storage, request counts or data transfer out. Then I turn on or query the access logs and data events to see who is making the requests: the requester identity, IP addresses, user agents and the most requested keys. In this case, one internal client lists the bucket continuously, which generates millions of paid list requests, and a popular public file is being hot-linked by a third-party site, creating egress costs. I fix the client to rely on event notifications rather than polling, and I stop the leak by blocking public access, serving public content through a CDN with caching, or requiring signed URLs. To prevent surprises, I set up cost anomaly detection and budget alerts, tag buckets with owners, and review default access settings across accounts.

Likely follow-ups

- How do you find who makes the requests?
- Why use a CDN for public files?

SYMPTOM

Storage bill up 5x. Request and data transfer lines dominate. No new features shipped.

INVESTIGATION

```
aws s3api get-bucket-metrics-configuration ...
```

```
aws s3api list-bucket-metrics-configurations --bucket b
```

```
# enable and query S3 access logs / CloudTrail data events
```

```
athena: select requester, count(*) from s3_logs group by 1 order by 2 desc limit 10
```

```
aws ce get-cost-and-usage --group-by Type=DIMENSION,Key=USAGE_TYPE
```

ROOT CAUSE

A misconfigured client polls the bucket listing every second, and an unauthenticated public object is being hot-linked by a third party.

FIX / MITIGATION

```
# fix the polling client to use events/notifications
```

```
# block public access, restrict via CloudFront signed URLs
```

```
aws s3api put-public-access-block --bucket b --public-access-block-configuration
BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=
true
```

● PREVENTION

Enable cost anomaly detection, put CDN in front of public content, block public access by default and use event notifications instead of polling.

Interview tip: The requester and user agent in access logs usually identify the culprit within minutes.

Official docs

- [Cloud Storage](#)

S141 A cross-account IAM role assumption suddenly fails with AccessDenied after a policy cleanup. How do you debug it?

Cloud / AWS

Spoken answer · ~63 s

A cross-account assume-role requires two permissions to line up: the identity policy of the caller must allow the assume-role action on the target role, and the target role's trust policy must name the caller as a trusted principal, with any conditions satisfied, such as an external ID or multi-factor requirement. Organisation service control policies and permission boundaries can also deny it. I read the CloudTrail event for the denied call, which names the principal and often the reason, and use the policy simulator to check each layer. In this scenario the clean-up removed the allow statement from the caller, and the condition in the trust policy was no longer met. I restore the minimum permission and update the caller to pass the external ID. Afterwards, I improve how IAM changes are made: use last-accessed data before removing permissions, test with simulation, and keep policies in code with review and staged rollout.

Likely follow-ups

- What two sides must agree?
- What can silently deny besides IAM policies?

● SYMPTOM

sts:AssumeRole denied from account A to role in account B. Worked last week.

● INVESTIGATION

```
aws iam get-role --role-name target --query Role.AssumeRolePolicyDocument
```

```
aws iam simulate-principal-policy --policy-source-arn $CALLER --action-names
sts:AssumeRole --resource-arns $ROLE
```

```
aws cloudtrail lookup-events --lookup-attributes
AttributeKey=EventName,AttributeValue=AssumeRole
```

```
# check SCPs and permission boundaries
```

● ROOT CAUSE

The caller's identity policy no longer allows sts:AssumeRole on the target role after a cleanup, while the trust policy requires an ExternalId that was also removed from the caller's configuration.

● FIX / MITIGATION

```
# restore identity policy permitting sts:AssumeRole on that ARN
```

```
# pass ExternalId in the assume-role call
```

```
aws sts assume-role --role-arn $ROLE --role-session-name x --external-id abc123
```

● PREVENTION

Test IAM cleanups with policy simulation and last-accessed data, and manage roles as code with review.

Interview tip: Cross-account access needs both sides: the caller must be allowed to assume, and the target must trust the caller.

Official docs

- [AWS IAM: Roles](#)

S142 **A managed Kubernetes control plane upgrade finished, but nodes are on an old version and some workloads misbehave. How do you manage version skew?**

Cloud / AWS

🗣 Spoken answer · ~60 s

Kubernetes supports a limited version difference between the control plane and the kubelets, and the older the nodes, the more risk that APIs, features and behaviours mismatch. When I see nodes lagging, I list kubelet versions and compare them to the skew policy. If they are outside it, I treat it as a priority to realign. I upgrade node groups in sequence, one minor version at a time, using surge nodes or new node groups, cordon and drain old nodes while respecting disruption budgets, and verify workloads after each step. Meanwhile I limit risky changes and avoid deploying workloads that depend on new features. To prevent drift, I plan upgrades as a single process covering control plane, nodes and add-ons, automate node replacement on a regular cadence, and alert when node versions fall behind the control plane by more than a set margin.

Likely follow-ups

- What is version skew?
- How do you upgrade nodes safely?

● SYMPTOM

Control plane at 1.30, node groups still at 1.28. Some pods behave oddly, new features unavailable.

● INVESTIGATION

```
kubectl version; kubectl get nodes -o wide
```

```
kubectl get nodes -o custom-  
columns=NAME:.metadata.name,VER:.status.nodeInfo.kubeletVersion
```

```
# check supported skew policy
```

```
kubectl get pods -A -o wide | grep <old-node>
```

● ROOT CAUSE

Nodes lag beyond the supported skew (kubelet can be at most a few minor versions behind the API server), so behaviour is undefined.

● FIX / MITIGATION

```
# upgrade node groups one minor at a time, surge + drain
```

```
eksctl upgrade nodegroup --name ng1 --cluster c --kubernetes-version 1.29
```

```
# pause risky deploys until aligned
```

● PREVENTION

Upgrade control plane and nodes in one planned sequence, automate node rotation, and monitor version drift.

Interview tip: Skew limits exist for a reason. Treat out-of-policy skew as an incident-in-waiting.

📖 Official docs

- [Kubernetes: Version skew policy](#)

S143 Your multi-region failover test worked in staging but the real failover took 40 minutes because of DNS and stale caches. What do you improve?

Cloud / AWS

🗣 Spoken answer · ~63 s

Failover time is the sum of detection time, record time-to-live and how long clients cache answers. In the incident the health check needed several slow intervals to declare the region down, the record had a long time-to-live, and some clients, like a Java runtime with default caching, ignored the TTL. I shorten each contributor: quicker health checks with a low failure threshold, a TTL of thirty to sixty seconds on failover records, and client configuration for DNS caching in long-running processes. For faster and more reliable shifting I consider global load balancing or anycast, which moves traffic at the network layer without relying on client DNS behaviour. Importantly, I run realistic failover drills in production or a faithful replica, with real client types, measure the end-to-end recovery time against the target, and repeat until it is met. I also confirm that the secondary region has capacity for the full load.

Likely follow-ups

- What are the components of failover time?
- Why are drills with real clients important?

● SYMPTOM

Primary region failed; traffic took 40 minutes to shift. Some clients still hitting dead region.

● INVESTIGATION

```
dig +short app.example.com # TTL and answers
```

```
aws route53 get-health-check-status ...
```

```
# which clients cache DNS beyond TTL (JVM, mobile)
```

```
# check failover record config and health check interval
```

● ROOT CAUSE

High DNS TTL (1 hour), slow health check thresholds and clients that cache resolution kept routing to the failed region.

● FIX / MITIGATION

```
# lower TTL to 30-60s on failover records
```

```
# health check interval 10s, threshold 2
```

```
# use anycast / global load balancer for instant shift
```

```
# JVM: -Dnetworkaddress.cache.ttl=30
```

● PREVENTION

Run real failover game days with production-like clients, measure end-to-end RTO and fix each contributor.

Interview tip: DNS failover time = detection time + TTL + client caching. Measure all three.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S144 Intermittent connection resets (RST) occur between two services behind a NAT gateway on long-lived idle connections. What is happening?

Networking

Spoken answer · ~59 s

Firewalls, NAT gateways and load balancers keep a state table for each flow and expire entries after an idle timeout. If an application holds a pooled connection idle for longer than that, the middlebox forgets it, and when the client later sends data, the packet either gets a reset or disappears. The symptom is errors on the first request after a quiet period and healthy short-lived connections. I confirm by capturing packets to see the reset and by comparing the idle time with the gateway's documented timeout. The fix is to make the endpoints respect the shortest timeout in the path: enable TCP keepalives at an interval below it, or set the connection pool's maximum idle time lower, and add a safe retry on a stale connection for idempotent requests. Prevention is documenting each hop's timeout and standardising client libraries' pool settings.

Likely follow-ups

- Why does the first request after idle fail?
- What should keepalive be relative to the NAT timeout?

● SYMPTOM

Connections idle for several minutes then fail with connection reset on first use. Short connections fine.

● INVESTIGATION

```
tcpdump -i any host 10.0.1.5 and 'tcp[tcpflags] & tcp-rst != 0'
```

```
# NAT gateway idle timeout (e.g. 350s on AWS NAT)
```

```
sysctl net.ipv4.tcp_keepalive_time
```

```
# app connection pool max idle time
```

● ROOT CAUSE

The NAT gateway drops idle flows after its timeout; the client still believes the connection is open and sends on a dead flow, which gets an RST.

● FIX / MITIGATION

```
# enable TCP keepalive shorter than NAT timeout
```

```
sysctl -w net.ipv4.tcp_keepalive_time=120
```

```
# set pool max idle lifetime < NAT timeout
```

```
# add retry on stale connection for idempotent calls
```

● PREVENTION

Document infrastructure idle timeouts and configure pools and keepalives below the smallest one in the path.

Interview tip: Every stateful middlebox has an idle timeout. Keepalives and pool lifetimes must be shorter than the shortest.

Official docs

- [RFC 9293: TCP](#)
- [RFC 3022: Traditional NAT](#)
- [Amazon VPC: NAT gateways](#)

S145 **Packets are being dropped between two VPCs connected by peering, but only for large payloads. How do you diagnose MTU and fragmentation problems?**

Networking

🗣 Spoken answer · ~61 s

If handshakes and small messages succeed but larger transfers stall, I suspect a path MTU problem. Some link along the path, such as a VPN tunnel or overlay network, has a smaller maximum packet size than the endpoints assume. Normally a router that cannot forward an oversized packet with the do-not-fragment flag set sends back an ICMP fragmentation-needed message, and the sender reduces its packet size. If a firewall or security rule drops that ICMP message, the sender never learns and keeps retransmitting big packets that vanish. I confirm by pinging with the do-not-fragment flag at various sizes to find the largest that passes, and capturing traffic for the ICMP messages. The fix is to permit the required ICMP type, and clamp the TCP maximum segment size on the tunnel so the endpoints negotiate smaller segments, or set a consistent MTU. I then test with large transfers.

Likely follow-ups

- What is path MTU discovery?
- Why do small packets work?

● SYMPTOM

Small requests succeed, large responses hang. ICMP ping works. Path crosses a VPN or tunnel.

● INVESTIGATION

```
ping -M do -s 1472 target # find max unfragmented size
```

```
tracert target
```

```
ip link show eth0 | grep mtu
```

```
tcpdump -i eth0 'icmp[icmptype]==3 and icmp[icmpcode]==4'
```

● ROOT CAUSE

A tunnel with a smaller MTU drops oversized packets with DF set, and ICMP fragmentation-needed messages are blocked by a security rule, so path MTU discovery fails.

● FIX / MITIGATION

```
# allow ICMP type 3 code 4
```

```
# clamp TCP MSS on the tunnel
```

```
iptables -t mangle -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --clamp-mss-to-pmtu
```

```
# or lower interface MTU
```

● PREVENTION

Document MTU along each path, never block ICMP fragmentation-needed, and test large transfers after network changes.

Interview tip: The classic sign is: connect works, small data works, large data hangs. Think MTU.

Official docs

- [RFC 1191: Path MTU discovery](#)
- [Cloudflare Learning: What is MTU?](#)

S146 Users in one country can't reach your service while others can. How do you determine if it is routing, DNS, CDN or blocking?

Networking

Spoken answer · ~61 s

A regional problem has a limited set of causes, so I test from the affected region using external probes or a volunteer. I compare DNS answers from public resolvers and the local ISP resolver, check whether the TCP connection and TLS handshake succeed to the right address, trace the network path to see where packets stop, and look at CDN logs to see if the requests arrive at the edge. If the ISP resolver returns an outdated address, traffic goes to the wrong place; if the TLS certificate is wrong, something is intercepting; if the path ends at a particular network, it is a routing issue. In this case it is stale DNS. Mitigation includes keeping the old address serving or redirecting during transitions, lowering TTLs ahead of changes, advising users to use alternative resolvers, and contacting the ISP. Monitoring from several geographies catches regional breakage early.

Likely follow-ups

- How do you test from a specific region?
- How do you avoid stale DNS during migrations?

● SYMPTOM

Reports from users in one region/ISP: timeouts or wrong certificates. Others fine.

● INVESTIGATION

```
# test from multiple vantage points
```

```
curl -sv --resolve app.example.com:443:<ip> https://app.example.com
```

```
dig @8.8.8.8 app.example.com; dig @isp-resolver app.example.com
```

```
mtr -rw app.example.com # from affected region probe
```

```
# check CDN edge logs for that POP
```

● ROOT CAUSE

A regional ISP resolver returns an outdated IP for the domain, pointing to a decommissioned address range.

● FIX / MITIGATION

```
# lower TTL and wait/notify ISP, publish via alternative resolvers
```

```
# keep old IP responding with redirect during migration
```

```
# status page + support guidance (public DNS)
```

● PREVENTION

Keep old addresses alive through migrations for at least the max TTL, monitor from multiple geographies and test through major resolvers.

Interview tip: Compare answers from public resolvers and the ISP's resolver. Different answers point straight at DNS.

Official docs

- [Cloudflare Learning: What is a CDN?](#)

 Spoken answer · ~61 s

When security groups look correct, I check the other layers on the path. First, routing: does the new subnet's route table have a route to the database's subnet, or does the service run in a different network with no peering? Next, network ACLs, which are stateless, so they need explicit rules for the request and the response ephemeral port ranges; a missing return rule makes connections hang. Then I consider the database's own firewall settings, DNS resolution to a private address, and any endpoint policies. Reachability analysis tools can trace the path and point to the blocking component without guessing. Once the missing rule or route is added, I verify with a connection test and application logs. To avoid repeating this, I define networking with reusable modules, so new services inherit tested subnet, route and rule configurations, and I include automated reachability checks in the pipeline.

Likely follow-ups

- How do NACLs differ from security groups?
- What is reachability analysis?

 SYMPTOM

Connection timeout from new service to DB on port 5432. Existing services connect fine.

 INVESTIGATION

```
nc -zv db.internal 5432
```

```
# check route tables and subnets for the new service
```

```
aws ec2 describe-route-tables --filters Name=association.subnet-id,Values=$SUBNET
```

```
aws ec2 describe-network-acls --filters Name=association.subnet-id,Values=$SUBNET
```

```
# VPC reachability analyzer
```

 ROOT CAUSE

The new service sits in a subnet whose network ACL does not allow ephemeral return ports, or lacks a route to the database subnet.

 FIX / MITIGATION

```
# add NACL rules for return traffic (1024-65535) and DB port
```

```
# or place service in the correct subnet group
```

```
aws ec2 create-network-acl-entry ...
```

● PREVENTION

Define network layout in code with reusable modules and run reachability analysis in CI for new services.

Interview tip: Security groups are stateful, but network ACLs are stateless. They need explicit rules in both directions.

Official docs

- [Amazon VPC: Security groups](#)

S148 TLS handshakes to an internal service start failing across the fleet at midnight. Certificates look valid. What else could it be?

Networking

Spoken answer · ~56 s

Many teams monitor the expiry of the server certificate and forget everything above it. The client validates the entire chain, so an expired intermediate or root certificate breaks all connections that use it, typically at a precise time like midnight. I view the chain served by the endpoint and check the validity dates of each certificate, verify against the trust bundle, and also confirm clocks are synchronised, since skewed time causes similar errors. The remedy is installing the renewed intermediate in the chain the servers present, updating client trust stores if the root changed, and reloading or restarting services. For prevention, I monitor expiry dates for every certificate in the chain and in trust bundles, test validation using realistic clients, automate issuance and chain deployment, and keep a calendar for CA events and rotation procedures.

Likely follow-ups

- Why check intermediates?
- How can clock skew look like cert failure?

● SYMPTOM

x509 errors at 00:00 UTC across many clients. Server cert valid for 90 more days.

● INVESTIGATION

```
openssl s_client -connect svc:443 -showcerts </dev/null | openssl x509 -noout -dates - issuer
```

```
# check the intermediate and root in the chain
```

```
date -u; chronyc tracking
```

```
openssl verify -CAfile ca-bundle.pem chain.pem
```

● ROOT CAUSE

The intermediate CA certificate expired at midnight, though the leaf certificate was valid; servers did not send the new intermediate.

● FIX / MITIGATION

```
# install the renewed intermediate in the chain served
```

```
# update trust bundles in clients if root changed
```

```
# restart services to load the chain
```

● PREVENTION

Monitor expiry for every certificate in the chain, including intermediates and roots, and test chain validation from clients.

Interview tip: A valid leaf does not help if any certificate above it in the chain has expired.

Official docs

- [RFC 8446: TLS 1.3](#)
- [Cloudflare Learning: TLS handshake](#)
- [Argo CD: ApplicationSet](#)

S149 A server has plenty of free disk blocks but cannot create files because it has run out of inodes. How do you find and fix it?

Linux Host

Spoken answer · ~60 s

The error message is the same as a full disk, but the cause differs: every file needs an inode, and the filesystem has a fixed number of them. Checking inode usage with the inode flag on `df` confirms it. To find the offender, I count files per directory, looking for places that generate many tiny files such as session stores, cache directories, mail spools or temporary files from a runaway job. I delete the old files in batches using `find` with a delete action and an age filter, taking care not to remove active ones, and fix the root cause by adding a retention policy or cleanup job. If the workload legitimately needs enormous numbers of small files, I consider creating the filesystem with more inodes or switching to object storage or a database. I also add inode usage to monitoring and alerts.

Likely follow-ups

- How does `df -i` help?
- Why use `find -delete` instead of `rm *`?

● SYMPTOM

'No space left on device' while `df -h` shows 30% used. Application cannot write new files.

● INVESTIGATION

```
df -i
```

```
find /var -xdev -type d -size +1M
```

```
for d in /var/*; do echo $d $(find $d -xdev | wc -l); done | sort -k2 -n | tail
```

```
ls /var/spool/mail | wc -l
```

● ROOT CAUSE

Millions of tiny session or cache files accumulated in one directory without cleanup, exhausting the inode table.

● FIX / MITIGATION

```
# delete in batches to avoid blocking
```

```
find /var/lib/php/sessions -type f -mtime +1 -delete
```

```
# configure cleanup (tmppreaper/logrotate/app GC)
```

```
# longer term: recreate FS with more inodes or use object storage
```

● PREVENTION

Monitor inode usage as well as blocks, add retention to caches and sessions, and choose appropriate filesystem parameters.

Interview tip: Deleting millions of files with `rm *` may fail with argument list too long. Use `find -delete`.

Official docs

- [Linux man-pages: inode\(7\)](#).

S150 **A process is stuck in uninterruptible sleep (D state) and cannot be killed.**
What causes this and how do you recover?

Linux Host

 Spoken answer · ~63 s

The D state means the process is blocked in the kernel waiting for IO and cannot be interrupted, so signals, even kill nine, are not handled until the operation returns. It typically signals trouble below the application: an unreachable NFS server, a hung storage array or failing disk, or a driver problem. These blocked processes still count toward the load average, which is why load can be high while the CPU is idle. I look at the wait channel and kernel stack to see what they are waiting on, check mounts and storage latency, and the kernel log for errors. Recovery means fixing the underlying path, for example restoring network connectivity to the file server or failing over the storage. If that is impossible, a lazy unmount can detach the mount point, and as a last resort a reboot. Prevention includes careful mount options, monitoring and avoiding single points of failure.

Likely follow-ups

- Why does load rise with idle CPU?
- What is a lazy unmount?

SYMPTOM

ps shows D state processes, load average climbing, kill -9 has no effect.

INVESTIGATION

```
ps -eo pid,stat,wchan:30,cmd | awk '$2 ~ /D/'
```

```
cat /proc/<pid>/stack
```

```
dmesg -T | tail -50
```

```
mount | grep -E 'nfs|cifs'; iostat -x 1 5
```

ROOT CAUSE

Processes wait on IO that never completes because a network filesystem server is unreachable or a disk is failing.

FIX / MITIGATION

```
# restore the storage path (network, server, multipath)
```

```
# or lazy unmount as last resort
```

```
umount -l /mnt/data
```

```
# if hardware: fail over and replace the disk
```

```
# reboot only when nothing else works
```

● PREVENTION

Use soft/timeout mount options where safe, monitor storage latency and avoid putting critical paths on fragile network mounts.

Interview tip: D-state processes count toward load average even with idle CPU, which explains high load with a quiet CPU.

Official docs

- [Brendan Gregg: Linux performance](#)

S151 **A time-sensitive application fails authentication because server clocks drifted by minutes. How do you fix time sync and prevent it?**

Linux Host

Spoken answer · ~59 s

Clock drift causes confusing failures: tokens appear expired or not yet valid, Kerberos rejects tickets, certificates look invalid and distributed logs cannot be correlated. I check the synchronisation state on affected hosts, comparing time against a trusted source, and look at the time service's sources and offset. The usual causes are a stopped or misconfigured daemon, firewall rules blocking the protocol port, or virtual machines paused and resumed. I restore the service, ensure it can reach reliable servers, and correct the offset. Large jumps can disturb running applications, so I prefer gradual adjustment for small drift and a controlled step for large ones, possibly restarting sensitive services. For prevention I deploy redundant internal time servers, configure all hosts consistently through configuration management, export the clock offset as a metric, and alert when it exceeds a small threshold such as a hundred milliseconds.

Likely follow-ups

- Why are token failures a time symptom?
- Step vs slew?

● SYMPTOM

Kerberos / JWT / TOTP failures, log timestamps out of order across hosts, certificate 'not yet valid' errors.

● INVESTIGATION

```
timedatectl status
```

```
chronyc tracking; chronyc sources -v
```

```
date -u; ssh host2 date -u
```

```
systemctl status chronyd
```

● ROOT CAUSE

NTP daemon stopped after a config change and VMs drifted; firewall blocked UDP 123 to time servers.

● FIX / MITIGATION

```
systemctl enable --now chronyd
```

```
chronyc makestep
```

```
# allow UDP 123 to internal NTP servers
```

```
# verify offset < 100 ms
```

● PREVENTION

Run internal NTP sources, monitor clock offset on every host, and alert when drift exceeds a small threshold.

Interview tip: Prefer stepping slowly (slew) for running services; use makestep carefully since time jumps can break applications.

Official docs

- [Terraform CLI: plan](#)

S152 After a kernel update and reboot, a server's network interface names changed and services bound to the old interface fail. What happened?

Linux Host

🗣 Spoken answer · ~60 s

Modern systemd systems name network interfaces based on hardware topology, and a kernel, driver or virtual hardware change can alter that name, so the configuration that referred to the old name no longer matches and the interface stays unconfigured. With SSH down, I use the cloud console or out-of-band management to log in, list the interfaces and see what they are called now, and look at the boot log for rename messages. The quick fix is to adjust the configuration to the new name or, better, match the device by its MAC address and assign a stable name, then reapply the network configuration and update firewall and application bindings that referenced the name. To prevent the problem, I use persistent naming rules or MAC-based matching, test reboots after kernel upgrades in a staging environment, roll out updates gradually, and keep console access available.

Likely follow-ups

- Why can interface names change?
- How do you configure by MAC?

SYMPTOM

eth0 became ens5; static config and firewall rules reference the old name; no network after reboot.

INVESTIGATION

```
ip -br link
```

```
journalctl -b | grep -i 'renamed'
```

```
cat /etc/netplan/*.yaml; ls /etc/sysconfig/network-scripts
```

```
# console access needed if SSH is down
```

ROOT CAUSE

Predictable interface naming or a changed driver/PCI order renamed the NIC; configs matched by name.

FIX / MITIGATION

```
# match by MAC address in netplan
```

```
match: macaddress: 0a:1b:2c:3d:4e:5f    set-name: eth0
```

```
netplan apply
```

```
# update firewall rules to the new name
```

● PREVENTION

Match interfaces by MAC or use persistent names, test reboots after kernel updates and keep out-of-band console access.

Interview tip: Always have console or out-of-band access before rebooting a remote server after kernel or network changes.

📖 Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S153 A Java service is repeatedly killed by the OOM killer although its heap is capped well below the container memory limit. Why?

Linux Host

🗣 Spoken answer · ~62 s

The container limit applies to total resident memory of the process, not the Java heap. The maximum heap setting caps only one region; the JVM also uses metaspace for class metadata, a stack for every thread, direct buffers for network and file IO, the JIT code cache, garbage collector structures and native libraries. With a large number of threads or heavy direct buffer use, the total can exceed the limit even when the heap looks healthy, and the kernel kills the container. I diagnose using native memory tracking and the cgroup memory statistics. The fix is leaving headroom: set the heap as a percentage of the container memory, around sixty to seventy percent, cap direct memory and metaspace, and control thread counts. I size containers based on measured resident memory under realistic load and monitor the working set, not just the heap, with alerts before reaching the limit.

Likely follow-ups

- What is outside the heap?
- How do you size container memory for a JVM?

● SYMPTOM

Container OOMKilled, Xmx=2g, limit=3g. Heap graphs look fine.

● INVESTIGATION

```
kubectl describe pod | grep -A5 'Last State'
```

```
jcmd <pid> VM.native_memory summary
```

```
cat /sys/fs/cgroup/memory.stat
```

```
ps -o rss,vsz -p <pid>
```

● ROOT CAUSE

Non-heap memory (metaspace, thread stacks, direct buffers, code cache, native libraries) plus a high thread count pushes total RSS over the limit.

● FIX / MITIGATION

```
# size heap to ~60-70% of the limit
```

```
-XX:MaxRAMPercentage=65
```

```
# cap direct memory and metaspace
```

```
-XX:MaxDirectMemorySize=256m -XX:MaxMetaspaceSize=256m
```

```
# reduce thread count / stack size
```

● PREVENTION

Enable native memory tracking in testing, size containers using measured RSS under load and alert on container memory working set.

Interview tip: The heap is only part of the JVM's memory. Budget for metaspace, threads, buffers and the runtime itself.

Official docs

- [Kubernetes: Resource management for Pods & containers](#)
- [Linux kernel docs: /proc/sys/vm](#)
- [Docker: Resource constraints](#)

S154 **A pipeline suddenly fails to pull base images with 'toomanyrequests' rate-limit errors. How do you fix it right now and permanently?**

CI/CD & Deploy

🗣 Spoken answer · ~61 s

Public registries limit anonymous pulls per IP address, and CI runners often share a few outbound addresses, so a busy day can consume the allowance and block every build. The immediate fix is authenticating pulls, which gives a much higher quota per account, by adding a credential to the runners and the Docker daemon. The permanent solution is to stop depending on the public registry at build time: set up a pull-through cache or mirror inside our network, or copy the base images we use into our own registry, scanning and signing them at the same time. Builds then reference the internal registry, pin images by digest for reproducibility, and benefit from caching, speed and independence from outages or removed tags. I also reduce unnecessary pulls with layer caching on runners and monitor for registry errors in CI metrics so I see the problem before developers complain.

Likely follow-ups

- Why do shared runners hit limits?
- What does a pull-through cache give?

● SYMPTOM

docker pull fails: 'You have reached your pull rate limit'. Builds blocked across teams.

● INVESTIGATION

```
docker pull alpine:3.20 2>&1 | head
```

```
curl -s -H "Authorization: Bearer $TOKEN" -I https://registry-1.docker.io/v2/ratelimitpreview/test/manifests/latest
```

```
# which jobs pull most? shared runner IP?
```

● ROOT CAUSE

Many CI jobs behind a shared NAT IP make anonymous pulls and exhaust the public registry quota.

● FIX / MITIGATION

```
# authenticate pulls with a paid account / token
```

```
docker login -u ci -p $TOKEN
```

```
# configure a pull-through cache / mirror
```

```
# use internal registry for base images
```

● PREVENTION

Mirror public images into an internal registry, pin by digest and reduce pulls with layer caching.

Interview tip: A pull-through cache also protects you from registry outages and deleted upstream tags.

Official docs

- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)

S155 A deployment succeeds but the application serves the old version for some users for hours. How do you find the cause?

CI/CD & Deploy

Spoken answer · ~59 s

A green pipeline only says the deploy command succeeded, not that every user receives the new version. I first check which version each region, pod and CDN edge actually serves, using a version endpoint or response header. Two classic causes appear: mutable image tags such as latest combined with a pull policy that reuses a cached image on a node, so some pods run old code; and caching of static assets or API responses at a CDN or browser with long lifetimes, so users keep getting old bundles. The fixes are deploying immutable references by digest or unique tag, using content-hashed filenames for assets so a new build means a new URL, setting sensible cache headers, and invalidating the CDN when needed. I add a post-deployment verification step that queries the version in every region and fails the pipeline if they disagree.

Likely follow-ups

- Why avoid the latest tag?
- How do hashed filenames help?

● SYMPTOM

Pipeline green, new version live in some regions, others still show old behaviour.

● INVESTIGATION

```
curl -s https://app/version; for r in eu us ap; do curl -s https://$r.app/version; done
```

```
kubectl get deploy -A -o wide | grep app
```

```
# CDN cache headers
```

```
curl -sI https://app/static/main.js | grep -i 'cache-control\|age\|etag'
```

```
# check image tags: latest vs digest
```

● ROOT CAUSE

The deployment used the mutable latest tag and a node cached the old image with `imagePullPolicy IfNotPresent`; the CDN also cached the old bundle for 24 hours.

● FIX / MITIGATION

```
# deploy by immutable tag/digest
```

```
image: registry/app@sha256:abc...
```

```
# purge CDN and use content-hashed filenames
```

```
aws cloudfront create-invalidation --distribution-id D --paths '/*'
```

● PREVENTION

Never deploy mutable tags, use hashed asset names, and add a post-deploy check that verifies the running version in every region.

Interview tip: Expose the build version in a header or endpoint so you can verify what is actually serving traffic.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S156 A monorepo CI run takes two hours because it builds and tests everything on every commit. How do you cut it down safely?

CI/CD & Deploy

Spoken answer · ~62 s

First I measure: break the pipeline into stages to find the critical path and the slowest jobs. Then I apply layered improvements. Change-based selection uses the dependency graph to build and test only the projects affected by a commit, with full runs on the main branch or for shared changes. A remote cache stores build and test results keyed by inputs, so unchanged work is reused. Tests are sharded and run in parallel across multiple runners, and slow end-to-end suites move from every commit to a merge queue or nightly run, while a fast smoke and unit tier gives quick pre-merge feedback. I also fix flaky tests since retries add time. To avoid trading quality for speed, the complete suite still runs before release and on the main branch, and I track pipeline duration and failure escape rate as metrics, so I notice if the optimisation lets bugs through.

Likely follow-ups

- What is affected-only testing?
- What stays in the pre-merge tier?

● SYMPTOM

Developers wait hours for feedback, merge queue backed up, teams skip tests locally.

● INVESTIGATION

```
# profile pipeline stages and critical path
```

```
gh run view --log | grep -E 'took|duration'
```

```
# which tests fail rarely / are slowest
```

```
# what changed vs what is built
```

```
git diff --name-only origin/main...HEAD
```

● ROOT CAUSE

No change-based selection, no remote cache, serial test execution and heavy end-to-end tests run on every commit.

● FIX / MITIGATION

```
# affected-only builds with dependency graph
```

```
nx affected --target=test
```

```
# remote build cache
```

```
# shard tests across runners in parallel
```

```
# move slow e2e to merge queue / nightly
```

● PREVENTION

Track pipeline duration as an SLO, review the slowest jobs monthly and keep a fast pre-merge tier plus a thorough post-merge tier.

Interview tip: Fast feedback beats exhaustive feedback on every commit. Run the thorough suite where it counts, but still run it.

📖 Official docs

- [GNU coreutils manual](#)

design around it?

🗣 Spoken answer · ~59 s

Terraform's default behaviour when an attribute forces replacement is to destroy the old resource and then create the new one, which for something in the traffic path means downtime. The change was approved without anyone noticing the replacement marker in a long plan. To recover, I verify the new load balancer is healthy, update DNS or listeners as needed, and confirm that traffic is restored. To prevent a repeat, I use the lifecycle setting to create the replacement before destroying the old one, with names that avoid conflicts, or design blue-green load balancers so a switch is a controlled traffic move. Process fixes matter too: plan summaries that highlight destroys and replacements, policy-as-code that blocks or requires extra approval for replacing critical resources, and protecting production resources with prevent-destroy and deletion protection. Approvers are trained to read the replacement lines first.

Likely follow-ups

- What does `create_before_destroy` do?
- How do you make plans reviewable?

SYMPTOM

Brief production outage: LB deleted, new LB created after; DNS now points at nothing for several minutes.

INVESTIGATION

```
terraform plan | grep -B3 'must be replaced'
```

```
terraform state list | grep lb
```

```
# review the plan output that was approved
```

```
aws elbv2 describe-load-balancers
```

ROOT CAUSE

A change forced replacement of the load balancer, and the default order is destroy then create; the plan was approved without noticing the replacement.

FIX / MITIGATION

```
# apply with create_before_destroy where possible
```

```
lifecycle { create_before_destroy = true }
```

```
# use unique name_prefix to avoid conflicts
```

```
# manually switch DNS to new LB, verify
```

● PREVENTION

Review plans for replacement markers, add policy checks that block replacement of critical resources, and use blue/green for load balancers.

Interview tip: Search the plan for 'replace' and 'destroy'. Make approvers look at those lines first.

📖 Official docs

- [Cloudflare Learning: Load balancing](#)

S158 A dependency update bot merged a minor version bump that broke production because the library changed behaviour. How do you make automation safer?

CI/CD & Deploy

🗣 Spoken answer · ~57 s

Dependency automation is valuable for security and maintenance, but it moves risk into the pipeline. After reverting and restoring correct behaviour, I analyse why the tests did not catch the change. The pricing logic depended on library behaviour that no test pinned down, so a subtle rounding difference passed. I add characterisation tests with representative inputs and expected outputs for critical calculations. I also tighten the automation policy: auto-merge only low-risk updates, such as patch versions with high coverage, require human review for minor or major updates in critical libraries, group updates in smaller batches, and deploy through a canary stage with business metric monitoring so anomalies are caught on a small share of traffic. Reading release notes for libraries on critical paths and keeping lockfiles committed ensure the exact versions in production are known and reproducible.

Likely follow-ups

- Why is semver not a guarantee?
- What should be auto-merged?

● SYMPTOM

Auto-merged dependency PR passed unit tests but changed rounding in a pricing library; wrong prices shown.

● INVESTIGATION

```
git log -5 --oneline package-lock.json
```

```
diff old/new library release notes
```

```
# what tests would have caught it?
```

```
# check which services consume the lib
```

● ROOT CAUSE

A semantic change in a minor release was not covered by tests, and auto-merge had no canary or staged rollout.

● FIX / MITIGATION

```
# pin and revert to the previous version
```

```
git revert <bot-merge>
```

```
# add characterisation tests for pricing
```

```
# staged rollout of dependency updates
```

● PREVENTION

Auto-merge only patch updates with strong test coverage, require canary deployments, and add contract tests for critical calculations.

Interview tip: Semver is a promise, not a guarantee. Critical paths need tests that pin their expected behaviour.

Official docs

- [Kubernetes: Version skew policy](#)
- [Renovate documentation](#)

S159 Replication lag on a read replica grows steadily during a nightly batch job, causing stale reads. How do you handle it?

Data / Stateful

Spoken answer · ~61 s

Replication lag happens when the replica cannot apply changes as fast as the primary generates them, or when queries on the replica conflict with the replay of changes. A nightly batch job writes a large volume of data, and heavy queries on the replica may also hold back replay. I check the lag metric over time, the write volume on the primary, and long-running queries on the replica. Mitigations include splitting the batch into smaller transactions with pauses to smooth the write rate, dedicating a separate replica for long analytical queries, tuning the replay conflict settings with an understanding of the trade-offs, and giving the replica enough IO. On the application side, I route reads that require the latest data, such as immediately after a write, to the primary, and use lag-aware routing that avoids replicas that are too far behind. Alerts on lag provide early warning.

Likely follow-ups

- Why do queries conflict with replay?
- Which reads need the primary?

● SYMPTOM

Replica lag climbs to 20 minutes between 01:00 and 03:00; users see missing recent updates.

● INVESTIGATION

```
select now() - pg_last_xact_replay_timestamp() as lag;
```

```
show max_wal_senders; select * from pg_stat_replication;
```

```
# long-running transactions on the replica
```

```
select pid, now()-xact_start from pg_stat_activity where state<>'idle' order by 2 desc;
```

● ROOT CAUSE

The batch job produces a huge volume of WAL, and long analytical queries on the replica conflict with replay.

● FIX / MITIGATION

```
# batch in smaller transactions with sleeps
```

```
# route analytics to a dedicated replica
```

```
# tune max_standby_streaming_delay and hot_standby_feedback
```

```
# read-your-writes: send critical reads to primary
```

● PREVENTION

Alert on replication lag, separate workloads by replica, and design read paths to tolerate or avoid stale data.

Interview tip: Replicas are eventually consistent. Decide which reads need the primary.

Official docs

- [Kubernetes: Jobs](#)
- [PostgreSQL: High availability, load balancing, replication](#)

S160 A Kafka consumer group is stuck rebalancing constantly and throughput collapses. What do you check?

Data / Stateful

Spoken answer · ~58 s

In Kafka, consumers must call poll regularly. If processing a batch takes longer than the maximum poll interval, the group coordinator assumes the consumer has failed, removes it and triggers a rebalance, during which all consumers in the group pause. The consumer comes back, joins, and the cycle repeats, so throughput collapses and lag grows. I check the consumer logs for poll timeout messages, compare batch processing time with the configured interval, and look for external triggers like pods restarting or autoscaling flapping. I fix it by reducing the number of records per poll, speeding up or parallelising processing outside the polling thread, or increasing the interval within reason. Cooperative rebalancing and static membership reduce disruption from deployments and restarts. I add alerts on rebalance rate and lag, and load test with realistic message sizes and downstream latency.

Likely follow-ups

- Why does a rebalance pause the group?
- What is static membership?

SYMPTOM

Frequent rebalances, consumer lag grows, logs show 'Revoking previously assigned partitions' repeatedly.

INVESTIGATION

```
kafka-consumer-groups.sh --describe --group g --bootstrap-server b:9092
```

```
# max.poll.interval.ms vs processing time
```

```
grep -i 'rebalance\|poll timeout\|session timeout' consumer.log
```

```
# pod restarts / autoscaling of consumers
```

ROOT CAUSE

Message processing takes longer than max.poll.interval.ms, so the broker thinks the consumer is dead and triggers a rebalance, which repeats.

FIX / MITIGATION

```
# reduce max.poll.records or speed up processing
```

```
max.poll.interval.ms=600000 max.poll.records=100
```

```
# use cooperative sticky assignor
```

```
# stop autoscaling flapping consumers
```

```
# static membership group.instance.id
```

● PREVENTION

Alert on rebalance frequency and lag, decouple slow processing with worker pools, and use static membership for stable deployments.

Interview tip: Each rebalance pauses consumption for the group. Frequent rebalances turn into a throughput outage.

Official docs

- [Apache Kafka documentation](#)

S161 Elasticsearch cluster turns yellow then red after a node restart, and search results are incomplete. How do you recover?

Data / Stateful

Spoken answer · ~61 s

Red means at least one primary shard is unassigned, so some data is unavailable. I begin with the cluster health and list the unassigned shards, then use the allocation explanation to see why: not enough disk space under the watermark settings, no node with the right attributes, a node still missing, or corrupted data. In this case the remaining nodes crossed their disk watermark, so Elasticsearch refused to allocate shards to them, and an index with no replicas lost its only copy when the node went down. I free up disk or add capacity, adjust the watermarks temporarily if safe, and trigger a reroute to retry allocation. If a primary's data is truly lost, I restore that index from a snapshot. Afterwards I ensure important indices have replicas, monitor disk usage with margins, take regular snapshots and test restores, and use shard allocation awareness across zones.

Likely follow-ups

- What do watermarks do?
- Why have replicas?

● SYMPTOM

Cluster health red; unassigned shards; some indices unavailable.

● INVESTIGATION

```
curl -s localhost:9200/_cluster/health?pretty
```

```
curl -s localhost:9200/_cat/shards?v | grep UNASSIGNED
```

```
curl -s localhost:9200/_cluster/allocation/explain?pretty
```

```
curl -s localhost:9200/_cat/allocation?v
```

● ROOT CAUSE

Disk watermark exceeded on remaining nodes so shards cannot be allocated, and replica count was zero for one index so its data was only on the restarted node.

● FIX / MITIGATION

```
# free disk or add node; clear watermark block
```

```
curl -XPUT localhost:9200/_cluster/settings -d '{"transient":  
{"cluster.routing.allocation.disk.watermark.low":"90%"}}'
```

```
# retry allocation
```

```
curl -XPOST 'localhost:9200/_cluster/reroute?retry_failed=true'
```

```
# restore missing index from snapshot if needed
```

● PREVENTION

Keep at least one replica for important indices, maintain disk headroom, take snapshots and use allocation awareness.

Interview tip: The allocation explain API tells you exactly why a shard is not assigned. Start there.

Official docs

- [Elasticsearch reference](#)

 Spoken answer · ~62 s

When Redis reaches its memory limit, the eviction policy removes keys, and if the evicted entries are the hot ones, the cache hit ratio drops and requests fall through to the database, which can overload it. I check memory usage, the eviction counters and hit ratio trends, find large keys that consume the space, and see what changed recently. Here a new feature stores sizeable objects with no expiry. Immediate actions are increasing the memory limit or adding shards if possible, removing or shrinking the offending keys, and protecting the database with rate limiting, request coalescing so identical misses result in one query, and temporary load shedding. The permanent fix includes setting a default TTL, bounding object size, choosing an eviction policy that matches usage, and monitoring hit ratio, evictions and memory with alerts. I also load test the system with a cold cache to know its true capacity.

Likely follow-ups

- Why does eviction overload the DB?
- What is request coalescing?

 SYMPTOM

Hit ratio drops, evicted_keys increases, DB CPU spikes, latency rises.

 INVESTIGATION

```
redis-cli info memory | grep -E 'used_memory_human|maxmemory|evicted'
```

```
redis-cli info stats | grep -E 'keyspace_hits|keyspace_misses|evicted_keys'
```

```
redis-cli --bigkeys
```

```
redis-cli config get maxmemory-policy
```

 ROOT CAUSE

A new feature stores large objects without TTL, filling memory to maxmemory so the allkeys-lru policy evicts hot keys.

 FIX / MITIGATION

```
# add TTLs, cap object sizes
```

```
# increase memory or shard
```

```
redis-cli config set maxmemory 8gb
```

```
# protect DB with request coalescing and rate limiting
```

```
# remove the large keys
```

● PREVENTION

Alert on hit ratio and evictions, set TTLs by default, review key patterns, and protect the database with stampede controls.

Interview tip: A cache is a performance layer and often a hidden capacity dependency. Know what happens to the DB when hit ratio drops.

Official docs

- [Kubernetes: Node-pressure eviction](#)
- [Redis documentation](#)

S163 A data pipeline silently produced duplicate records after a retry, and downstream reports are inflated. How do you detect, fix and prevent it?

Data / Stateful

Spoken answer · ~58 s

To detect the problem, I compare record counts and checksums between source and destination per partition or day, and query for duplicate keys. The logs show that a batch load failed halfway and was retried. Because delivery is at least once and the insert has no uniqueness guarantee, rows loaded twice. To fix existing data, I identify the affected partitions, deduplicate using the business key, or reload those partitions from the source in an idempotent way, and re-run downstream reports, informing consumers about the corrected numbers. To prevent recurrence, I make loads idempotent using merge or upsert on a natural key or batch identifier, track processed batch IDs in an audit table, add uniqueness constraints where the store supports them, and run automated reconciliation checks after each load that alert on mismatches. Data quality tests become part of the pipeline.

Likely follow-ups

- What makes a load idempotent?
- How do you reconcile source and sink?

● SYMPTOM

Revenue report 12% too high. Row counts in the warehouse exceed source counts.

● INVESTIGATION

```
select id, count(*) from fact_orders group by id having count(*) > 1;
```

```
# compare source vs destination counts per day
```

```
# check job logs for retries / at-least-once delivery
```

```
select batch_id, count(*) from load_audit group by 1;
```

● ROOT CAUSE

The loader uses at-least-once delivery and inserts without a unique key, so a retried batch loaded twice.

● FIX / MITIGATION

```
# dedupe with window function and rebuild affected partitions
```

```
delete from fact_orders where ctid not in (select min(ctid) from fact_orders group by id);
```

```
# make loads idempotent: MERGE/upsert on natural key
```

```
# add unique constraint / dedup in ETL
```

● PREVENTION

Design pipelines to be idempotent, add reconciliation checks between source and sink and alert on count mismatch.

Interview tip: At-least-once plus idempotent writes gives effectively-once results. Retries without idempotency create duplicates.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Spoken answer · ~58 s

An alert that only says a service has a high error rate forces the responder to rediscover context under pressure. I improve alerts by making them answer four questions: what is broken from the user's point of view, how severe it is, where to look, and what to do first. Practically, each alert carries a human summary with the user impact, labels for the owning team and severity, links to the relevant dashboard and runbook, and where possible dynamic context like the top error message and the most recent deployment. I map services to user journeys, so alerts speak in terms such as card payments failing rather than internal names. Templates enforce required fields, and after every incident I review whether the alert helped or delayed diagnosis, then update it. Alert quality becomes a tracked item in on-call retrospectives.

Likely follow-ups

- What should every alert include?
- How do you map services to journeys?

SYMPTOM

Alert text: 'High error rate on service X'. Responder spends 15 minutes finding impact and owner.

INVESTIGATION

```
# review alert annotations and labels
```

```
kubectl get prometheusrule -o yaml | grep -A10 annotations
```

```
# do alerts link dashboards, runbooks, owners?
```

```
# can we map service to user journey?
```

ROOT CAUSE

Alerts have no context: no impact description, severity rationale, dashboard or runbook link, and no mapping to the user journey.

FIX / MITIGATION

```
annotations: summary: 'Checkout errors > 2% (card payments failing)'
```

```
dashboard: https://grafana/d/checkout
```

```
runbook: https://wiki/runbooks/checkout-errors
```

```
labels: team: payments, severity: page
```

```
# include top error and recent deploy
```

● PREVENTION

Alert templates with required fields, and a review of alerts after every incident for missing context.

Interview tip: A good alert answers: what is broken for whom, how bad, where to look, and what to do first.

Official docs

- [Google SRE Workbook: On-Call](#)

S165 Logs from one service stopped appearing in your central logging system an hour ago. How do you find where the pipeline is broken?

Observability

Spoken answer · ~59 s

I trace the path from source to storage. First I confirm the application still writes logs locally, using the container logs, so I know whether the problem is before or after collection. Then I check the collection agent on the node for errors, buffering and dropped records, and finally the backend's ingestion metrics for rejected or rate-limited streams. Here the deployment changed the log format and started emitting oversized lines that the agent or backend rejected. I fix the source by limiting message size and correcting the verbose log statement, adjust limits cautiously if the large lines are legitimate, and recover missing logs from node files if they are still available. To avoid a blind spot next time, I monitor the pipeline itself: ingest rate per service, drop counters and agent lag, and alert when a service that normally logs goes quiet.

Likely follow-ups

- How do you locate the break in the pipeline?
- Why alert on missing logs?

● SYMPTOM

Kibana/Loki shows no logs from service payments since 10:14. Service is running and healthy.

● INVESTIGATION

```
kubectl logs deploy/payments | tail
```

```
kubectl get pods -n logging; kubectl logs ds/fluent-bit -n logging | tail
```

```
# check agent buffer / backpressure / rejections
```

```
curl -s loki:3100/metrics | grep -E 'discarded|rate_limited'
```

```
# did log format or volume change?
```

● ROOT CAUSE

The service began emitting very large JSON lines after a deploy; the agent dropped them as oversized and the backend rate-limited the stream.

● FIX / MITIGATION

```
# cap line size in the app, fix the log message
```

```
# raise agent buffer / max line size cautiously
```

```
# re-ship from node files if still available
```

```
# add the stream to rate limit allowlist if justified
```

● PREVENTION

Monitor log pipeline health (ingest rate per service, drops, lag), alert on silent services, and set size limits at the source.

Interview tip: Alert on the absence of logs from services that should always log. Silence is a signal.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S166 Your trace sampling is set to 1% and the intermittent error you are chasing never shows up. How do you capture the interesting traces?

Observability

🗣 Spoken answer · ~61 s

Head-based sampling decides at the start of a request whether to record it, before knowing whether it will fail or be slow, so rare problems are kept only at the sampling rate. For a failure occurring in half a percent of requests and a one percent sample, I would see almost none. The better approach is tail-based sampling: collectors buffer spans for a request, and once the trace is complete, apply policies, keeping all traces with errors, all traces slower than a threshold, and a small random fraction for baseline comparison. That keeps costs under control while preserving the interesting data. It needs enough collector memory and routing so all spans of a trace reach the same collector. I also add exemplars so metric spikes link directly to example traces, and include the trace ID in log lines, so any log can be pivoted to its trace.

Likely follow-ups

- Why does head sampling miss rare errors?
- What does tail sampling need?

SYMPTOM

Rare failures (0.05%) are invisible in traces; logs lack context to link requests.

INVESTIGATION

```
# current sampling config

grep -R 'sampler\|sampling' otel-collector.yaml

# are errors and slow traces force-kept?

# error rate vs traces with error status
```

ROOT CAUSE

Head-based random sampling decides before the outcome is known, so rare errors are almost never kept.

FIX / MITIGATION

```
# tail-based sampling in the collector

processors: tail_sampling: policies: [{name: errors, type: status_code, status_code:
{status_codes: [ERROR]}}, {name: slow, type: latency, latency: {threshold_ms: 1000}},
{name: base, type: probabilistic, probabilistic: {sampling_percentage: 1}}]
```

PREVENTION

Use tail-based sampling to keep all errors and slow requests, add exemplars linking metrics to traces and keep trace IDs in logs.

Interview tip: Sample after you know the outcome. Keep every error and every slow trace, and a small random baseline.

Official docs

- [OpenTelemetry: Sampling](#)

S167 After enabling debug logging to chase a bug, the application becomes slower and the log bill jumps. How should you debug in production safely?

Observability

Spoken answer · ~55 s

Turning on debug level for everything is a blunt instrument: log volume explodes, hot code paths slow down because of formatting and IO, and ingestion costs rise. I revert to a normal level and use a more targeted approach. The application should support changing log levels at runtime for specific loggers and for a limited scope, such as one instance, one tenant or requests with a particular header, with an automatic expiry like fifteen minutes. Asynchronous appenders and sampling reduce overhead. Often traces, metrics and profiling answer the question without verbose logs, and a feature-flagged diagnostic mode can capture exactly the data needed. Everything gets recorded in the incident or investigation notes. Afterwards I ensure that logs are structured and include the identifiers needed to investigate without debug mode in the future.

Likely follow-ups

- Why should debug logging expire?
- What alternatives exist to verbose logs?

SYMPTOM

Latency up 20% and logging costs triple after switching the log level to debug fleet-wide.

INVESTIGATION

measure overhead: latency and CPU vs log volume

kubectl top pods; check log lines per second

is debug enabled everywhere or only needed paths?

ROOT CAUSE

Debug logging was enabled globally, producing high volume synchronously on hot paths.

FIX / MITIGATION

```
# revert to info; enable debug for one pod or one user via dynamic log level
```

```
curl -X POST app/admin/loglevel -d 'logger=payments&level=debug&ttr=15m'
```

```
# use sampling and async appenders
```

```
# use traces / feature-flagged diagnostics instead
```

● PREVENTION

Support dynamic, scoped, time-limited log levels, structured logs, and rely on traces and metrics before verbose logging.

Interview tip: Make debug logging expire automatically so it is never left on by accident.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)

S168 A queue consumer fleet autoscaled to 200 pods but throughput didn't improve, and the database is struggling. What went wrong?

Capacity & Scale

Spoken answer · ~57 s

Autoscaling the consumers assumed that consumer capacity was the limit. In reality the shared database is the bottleneck, so adding consumers only added more concurrent queries, lock contention and connections, which slowed everything, a classic case of congestion collapse. I confirm by looking at database CPU, waits and connection counts alongside per-message processing time, and by seeing that throughput stops rising or falls as concurrency increases. The fix is to cap the consumer count at what the database can sustain, then improve the actual bottleneck: batching writes, adding indexes, using a pooler, optimising queries, and scaling the database vertically or partitioning data. I add backpressure so consumers slow or pause when database latency rises, and set autoscaler limits based on downstream capacity. A full-path load test would have shown the real constraint before it hit production.

Likely follow-ups

- How do you find the true bottleneck?
- What is backpressure?

● SYMPTOM

Lag stays high despite 10x consumers; DB CPU at 100%, lock waits and connection errors.

● INVESTIGATION

```
kubectl get hpa; kafka-consumer-groups --describe
```

```
select count(*) from pg_stat_activity;
```

```
select wait_event_type, count(*) from pg_stat_activity group by 1;
```

```
# per-message processing time vs concurrency
```

● ROOT CAUSE

The bottleneck is the database, not the consumers. More consumers add contention and connection pressure, reducing total throughput.

● FIX / MITIGATION

```
# cap max replicas to what the DB can sustain
```

```
# batch writes, add indexes, use a connection pooler
```

```
# scale the DB or partition the workload
```

```
# apply backpressure: pause consumption when DB latency high
```

● PREVENTION

Load test the whole path to find the true bottleneck, set autoscaler maximums from downstream capacity, and implement backpressure.

Interview tip: Scaling a stage that isn't the bottleneck makes the real bottleneck worse.

Official docs

- [Argo CD: ApplicationSet](#)

S169 Your service handles peak traffic fine in load tests but fails in production at half that load. What differences do you look for?

Capacity & Scale

Spoken answer · ~60 s

A passing load test proves the system handles the test, not production. Typical differences are the request mix, because real users hit expensive endpoints and large tenants more than a uniform script does; data volume, since a small test database fits in memory and uses fast indexes; dependencies mocked in the test but real in production with latency and rate limits; cache state, such as warm caches in the test and cold or churning ones in production; and infrastructure differences like fewer replicas, shared noisy neighbours, connection limits and DNS behaviour. I compare telemetry from the failing production period against the test run to find which resource saturates first. Then I improve the test by replaying captured production traffic, using production-sized data, and including realistic downstream behaviour. After fixing the limiting factor, I rerun the test and validate its predictions against actual production load.

Likely follow-ups

- Why does data size matter in tests?
- How do you make load tests realistic?

SYMPTOM

Production degrades at 50% of tested peak. Staging passes with margin.

INVESTIGATION

```
# compare environments: data size, caches, topology, config
```

```
# request mix in prod vs test
```

```
# third-party calls mocked in test, real in prod
```

```
# noisy neighbours, shared limits, DNS, connection limits
```

ROOT CAUSE

The load test used a uniform request mix on a small dataset with mocked dependencies; production has a skewed mix, large data, real downstream rate limits and cold caches.

FIX / MITIGATION

```
# replay production traffic shape (shadow/replay)
```

```
# test with production-size data
```

```
# include real or realistic dependency latencies and limits
```

```
# fix the discovered limit (e.g. downstream rate limit)
```

● PREVENTION

Treat load test realism as a requirement: data volume, traffic mix, dependencies, cache state and concurrency.

Interview tip: A load test is only as good as its model of production. Validate the model against real traffic.

Official docs

- [Google SRE Book: Testing for Reliability](#)

S170 A latency-sensitive service shares nodes with batch jobs, and every batch run causes latency spikes. How do you isolate them?

Capacity & Scale

Spoken answer · ~58 s

Mixing latency-sensitive and batch workloads on the same machine invites interference. CPU limits and requests share the processor time, but not all resources are isolated: cache, memory bandwidth, disk and network IO are shared, and bursts from a batch job cause jitter in the service. I confirm by correlating spikes with the batch schedule and examining throttling metrics for the service. The most reliable fix is physical separation: a dedicated node pool for batch workloads with taints and tolerations, so they never land on the nodes used by the latency-critical service. For the service itself I use requests equal to limits for guaranteed quality of service, and consider CPU pinning. Priority classes ensure that the service wins under contention, and I set realistic requests on batch jobs and IO limits. Continued monitoring of tail latency and throttling verifies the result.

Likely follow-ups

- What resources aren't isolated by limits?
- Why use a dedicated node pool?

● SYMPTOM

p99 spikes coincide with batch jobs starting on the same node pool; CPU shares and cache contention.

● INVESTIGATION

```
kubectl get pods -o wide --all-namespaces | grep node-7
```

```
kubectl top node node-7
```

```
# CPU throttling for the latency service
```

```
container_cpu_cfs_throttled_periods_total
```

```
# check QoS classes and priorities
```

● ROOT CAUSE

Batch pods are Burstable with low requests, consuming bursty CPU, memory bandwidth and disk IO, while the latency service lacks guaranteed resources.

● FIX / MITIGATION

```
# dedicated node pool with taints for batch
```

```
tolerations + nodeSelector for batch jobs
```

```
# Guaranteed QoS and CPU pinning for latency-critical pods
```

```
# priority classes; set batch requests realistically
```

```
# IO limits for batch
```

● PREVENTION

Separate workload classes by node pool, set priorities and realistic requests and monitor throttling and noisy neighbour signals.

Interview tip: Shared nodes share caches, memory bandwidth and disk IO, which requests and limits do not fully isolate.

Official docs

- [Kubernetes: Jobs](#)

S171 **You must reduce infrastructure cost by 30% without hurting reliability.**
What is your approach?

Capacity & Scale

Spoken answer · ~58 s

I start with visibility: break down spending by service, team and environment using tags, and measure utilisation to find waste. The first wave of savings is usually safe: shut down idle or non-production environments outside working hours, delete orphaned volumes, old snapshots and unused load balancers, and apply storage lifecycle rules. The second wave is rightsizing, reducing instance sizes and Kubernetes requests based on observed usage, with headroom and SLO monitoring as guardrails, and applying changes gradually. The third is purchasing strategy: commit to savings plans or reservations for steady baseline load and use spot capacity for fault-tolerant batch work. I also look at architecture-driven costs like cross-zone traffic, egress and over-retained logs. Throughout, I track unit costs such as cost per request, keep reliability metrics visible, and make sure engineers see their own spend, so savings last.

Likely follow-ups

- What should you cut first?
- How do you protect reliability while cutting?

SYMPTOM

Finance target: cut cloud spend 30% in a quarter. SLOs must stay green.

INVESTIGATION

```
# find the big spenders and waste
```

```
aws ce get-cost-and-usage --group-by Type=DIMENSION,Key=SERVICE
```

```
# utilisation per resource
```

```
kubectl top nodes; compare requests vs usage
```

```
# idle resources, old snapshots, unattached volumes
```

ROOT CAUSE

Typical findings: over-provisioned instances, idle environments running 24/7, unattached storage, no commitments on baseline, and chatty cross-zone data transfer.

FIX / MITIGATION

```
# rightsize instances and requests with SLO guardrails
```

```
# schedule non-prod shutdown nights/weekends
```

```
# savings plans for baseline, spot for tolerant batch
```

delete orphaned volumes/snapshots, lifecycle storage

reduce cross-AZ and egress traffic

● PREVENTION

FinOps practice: tagging, showback, unit cost metrics, budgets, and regular reviews with engineers.

Interview tip: Remove waste first, then optimise. Cutting capacity that is actually used trades money for incidents.

Start here (official docs)

- [Google SRE Book: Effective Troubleshooting](#)