Cloud Troubleshooting & Support

A Complete 10-Chapter Support Engineering Course

Topics covered:

The support engineer's toolkit & diagnosing connectivity issues

IAM & permission troubleshooting · Reading logs & metrics under pressure

Common failure modes & root cause analysis · Incident response

Cost anomalies & billing support · Working with provider support

Multi-cloud & hybrid environments · Capstone: a full worked outage

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · the operational companion to Cloud Platform Fundamentals

Table of Contents

CHAPTER 1 OF 10

The Support Engineer's Cloud Toolkit

Cloud Troubleshooting & Support

Chapter 1 · The Support Engineer's Cloud Toolkit

Course 1 built the conceptual foundation — every major service category, framed cross-provider throughout. This course turns the troubleshooting instincts developed along the way into deliberate, practiced skill. This first chapter starts with the actual tools used day to day.

Console vs. CLI vs. API — Three Ways In

Every provider offers three ways to interact with resources: the **web console** (visual, good for exploration and one-off tasks — `cloud1-2`'s free-tier sandbox advice assumed exactly this), the **CLI** (`aws-cli`, `az`, `gcloud` — scriptable, faster for repetitive tasks, essential for automation), and the underlying **API** that both the console and CLI actually call underneath.

> **THE CONSOLE IS A UI, NOT A SEPARATE SYSTEM**
>
> Understanding that the console is just a visual layer over the same API the CLI calls directly explains why a change made in one shows up identically in the other — they're two interfaces to one underlying system, not two separate ones to reconcile. Worth keeping in mind if the CLI feels intimidating at first: it isn't doing anything fundamentally different from what clicking through the console already does.

The Three CLIs, Compared

	AWS	AZURE	GCP
CLI name	aws-cli	az	gcloud
Initial setup	`aws configure`	`az login`	`gcloud init`
Example command	`aws ec2 describe-instances`	`az vm list`	`gcloud compute instances list`

Exact syntax differs, but the underlying pattern is reassuringly consistent once recognized: `<cli> <service> <action>`, a noun-verb structure shared across all three.

Authentication for CLI Tools

Revisiting `cloud1-6`'s IAM material specifically for CLI access: access keys (AWS), service principals (Azure), and service accounts (GCP) are the CLI-specific credential types. The same hardcoded-credential warning from `cloud1-6` applies doubly here — a credentials file sitting on a support engineer's own laptop is a genuinely real risk if that laptop is ever compromised, or if the credentials file is accidentally synced or backed up somewhere it shouldn't be.

Why the CLI Matters for Support Work Specifically

- **Scriptability** — checking the same thing across many resources or accounts at once, something console clicking simply can't do efficiently.
- **Filterable, saveable output** — CLI output can be piped, filtered, and saved for documentation or escalation — this site's own `bash_intermediate_07` chapter on `jq` applies directly here for JSON output.
- **Reproducibility** — a documented CLI command in a support ticket is exact and unambiguous. "I clicked around in the console and fixed it" isn't reproducible or verifiable by anyone reviewing the ticket later.

Reading Architecture Diagrams

Each provider has its own standard icon set (AWS Architecture Icons, Azure Architecture Icons, GCP's own set) — not identical, but sharing the same visual language: boxes for resources, arrows for data/traffic flow, dotted boundaries for network boundaries (VPCs/subnets, `cloud1-5`). Being handed an unfamiliar customer's architecture diagram and quickly identifying which piece is likely failing, based on the reported symptoms, is a genuinely practical skill — directly drawing on `cloud1-5`'s networking material and `cloud1-8`'s monitoring material for deciding where to check first.

Common Support Workflows — A Practical Starting Toolkit

- **Resource inventory** — listing all resources of a given type or tag to confirm what actually exists versus what a customer believes exists (`cloud1-9`'s tagging material).
- **Point-in-time config snapshot** — capturing a resource's current configuration before making any change, as a rollback reference.
- **Cross-referencing a ticket's resource ID against current state** — a genuinely common first step, confirming the resource still exists and hasn't already been remediated by another automated process.

> **YOUR OWN LAPTOP'S CLI CREDENTIALS ARE A REAL SECURITY RESPONSIBILITY**
>
> Directly echoing `cloud1-6`'s and `pipelines1-5`'s hardcoded-credential warnings — now applied to a support engineer's own workstation rather than application code: credentials sitting in a CLI config file are a genuine target if that machine is compromised. Scoped-down, task-specific credentials (rather than broad admin access), MFA, and regular rotation all reduce this risk meaningfully.

Hands-On Exercises

EXERCISE 1

Explain why a CLI command documented in a support ticket is more useful to a colleague reviewing that ticket later than a note saying "I clicked through the console and fixed it."

View solution

EXERCISE 2

You're handed a diagram showing a load balancer, two web server icons behind it, and a database icon inside a separate dotted boundary. Users report "intermittent timeouts." Using Course 1's material, identify which piece you'd check first, and why.

View solution

EXERCISE 3

Explain the real security risk of CLI credentials stored on a support engineer's own laptop, and describe at least two practices that reduce that risk.

View solution

CHAPTER 1 QUICK REFERENCE

- **Console/CLI/API** — three interfaces to the same underlying system, not separate systems
- **aws-cli/az/gcloud** — different exact syntax, same noun-verb pattern (`<cli> <service> <action>`)
- CLI credentials (access keys/service principals/service accounts) carry the same hardcoded-credential risk as `cloud1-6`, now on a support engineer's own machine
- CLI advantages for support work: scriptability, filterable/saveable output, and — critically — reproducibility in tickets
- Reading architecture diagrams: boxes = resources, arrows = traffic flow, dotted boundaries = network boundaries (`cloud1-5`)
- Reusable workflows: resource inventory, point-in-time config snapshots, cross-referencing ticket resource IDs against current state
- **Next chapter:** Diagnosing Connectivity Issues — a real troubleshooting flowchart built on `cloud1-5`'s networking material

CHAPTER 2 OF 10

Diagnosing Connectivity Issues

Cloud Troubleshooting & Support

Chapter 2 · Diagnosing Connectivity Issues

"I can't connect to X" is probably the single most common category of real cloud support ticket. `cloud1-5` ended with a practical checklist — security group → NACL → route table → DNS. This chapter turns that checklist into a genuine, actionable flowchart.

Step 0 — Reproduce and Scope the Problem First

Before touching any configuration: confirm exactly what's failing — which specific connection, from where, to where, and what error is actually observed. "It doesn't work" isn't enough information to act on. Timeout, connection refused, and DNS resolution failure are three genuinely *different* categories of problem, and distinguishing between them immediately narrows the investigation before checking a single config setting.

Distinguishing Failure Types

FAILURE TYPE	WHAT IT MEANS
Connection refused	Something on the network path *is* reachable and actively rejecting the connection — often nothing is listening on that port, or an explicit reject rule exists
Connection timeout	No response at all — packets are being silently dropped, typically by a security group, NACL, or missing route (cloud firewalls usually drop rather than reject)
DNS resolution failure	The hostname never resolved to an IP at all — a completely different problem category, never even reaching the network layer

THIS DISTINCTION ALONE IS THE FASTEST TRIAGE STEP IN THIS ENTIRE CHAPTER

Refused vs. timeout vs. DNS failure narrows down which of `cloud1-5`'s four checklist layers is actually at fault before checking anything else — a DNS failure means the network layer was never even reached; a timeout points strongly at a silent drop (security group/NACL/route table); a refused connection points at something listening but rejecting, or nothing listening at all.

Working Through Security Groups

Check inbound rules on the destination for the correct port, protocol, and source. A genuinely common gotcha: fixating on the destination's security group while forgetting the *source* resource — if traffic originates from another VM or service inside the same VPC, that source resource's own outbound rules may also need to explicitly permit the connection.

Working Through NACLs

Revisiting `cloud1-5`'s stateless nuance in practice: check *both* inbound and outbound NACL rules explicitly, remembering the ephemeral port range needed for return traffic, since NACLs track nothing about the connection itself. A concrete, genuinely common mistake: allowing inbound port 443, but forgetting to allow the outbound ephemeral port range (typically 1024-65535) the response needs to actually leave on.

Working Through Route Tables

Confirm the subnet's route table actually has a route to wherever the traffic needs to go — an internet gateway for public traffic, a NAT gateway for private-subnet outbound, or a VPC peering connection/transit gateway for cross-VPC traffic (`cloud1-5`).

> **VPC PEERING NEEDS ROUTES ON BOTH SIDES**
>
> A route existing in one direction but not verified for the return path is a genuinely common real mistake, especially with VPC peering specifically — both VPCs need a route pointing at the peering connection, not just the one that initiated it. A route configured on only one side produces exactly the "traffic flows one way but not the other" symptom.

Working Through DNS

Confirm the hostname resolves to the IP actually expected, using tools like `nslookup` / `dig`. A genuinely common trap: DNS caching (`cloud1-8`'s TTL material) meaning a stale, outdated IP is still being returned locally, even though the DNS record itself was already correctly updated. Also worth checking: is this internal DNS (service discovery within a VPC) or external/public DNS (`cloud1-5`) — the two follow genuinely different resolution paths.

Putting It Together — The Full Flowchart

```
1. REPRODUCE & SCOPE   -- what, from where, to where, what error exactly
2. CLASSIFY THE FAILURE TYPE   -- refused / timeout / DNS failure
3a. If DNS failure  -> check resolution, TTL/caching, internal vs. external DNS
3b. If refused/timeout  -> check security groups (BOTH source and destination)
4. Check NACLs   -- both directions, remember ephemeral ports
5. Check route tables   -- both directions if cross-VPC/peered
6. If everything above passes  -> confirm the application itself is actually running and listening on th
```

That last step closes the loop deliberately: the network path can be entirely correct while nothing is actually listening on the other end — a genuinely common final culprit once every network-layer check has passed clean.

Hands-On Exercises

EXERCISE 1

A user reports "connection refused" (not a timeout) trying to reach a service on port 8080. Given this chapter's failure-type distinction, what does this specific error type suggest is *not* the problem, and what should be checked instead?

View solution

EXERCISE 2

Walk through the specific NACL gotcha of allowing inbound port 443 but forgetting the outbound ephemeral port range. Explain exactly what a user would observe as a result, and why.

View solution

EXERCISE 3

Two VPCs are peered, but traffic only flows in one direction. Using this chapter's route table material, explain the likely cause.

View solution

CHAPTER 2 QUICK REFERENCE

- **Refused** (something's listening, actively rejecting) vs. **timeout** (silent drop, likely SG/NACL/route) vs. **DNS failure** (never reached the network layer) — the fastest triage step in this chapter
- Security groups: check **both** the source's outbound and the destination's inbound rules

- NACLs: stateless — check both directions explicitly, including the ephemeral port range for return traffic
- Route tables: VPC peering needs a route configured on **both** sides, not just the initiator
- DNS: check for stale cached answers (TTL, `cloud1-8`) and confirm internal vs. external resolution path
- Full flowchart: reproduce/scope → classify failure type → security groups → NACLs → route tables → confirm the application is actually listening
- **Next chapter:** IAM & Permission Troubleshooting — reading and debugging policy errors, "access denied" causes

CHAPTER 3 OF 10

IAM & Permission Troubleshooting

Cloud Troubleshooting & Support

Chapter 3 · IAM & Permission Troubleshooting

Following Chapter 2's connectivity flowchart, this chapter tackles the second-most-common category of cloud support ticket: "access denied." It builds directly on `cloud1-6`'s IAM foundations, turning that chapter's concepts into a genuine, practiced investigation.

Restating the Foundation — Cloud1-6's AuthN vs. AuthZ Triage

Recall the split: is this a login problem, or a "logged in but blocked" problem? This chapter assumes authentication is already confirmed working, and focuses entirely on authorization — the much larger and more nuanced category of real IAM tickets in practice.

Reading a Policy Error Message

Every provider's "access denied" error carries real diagnostic information, often skipped past by a frustrated user or engineer. Worth reading in full, every time, before doing anything else:

- The specific **action** that was denied (e.g. `s3:GetObject`), not just "something failed."
- The specific **resource** it was attempted against — the exact identifier, not just "a bucket."
- Sometimes, which specific policy or organizational boundary actually caused the denial — modern provider tooling increasingly surfaces this directly.

> **READ THE FULL ERROR MESSAGE BEFORE GUESSING**
>
> One of the most common, easily avoidable time-wasters in real IAM troubleshooting is seeing "access denied," assuming a cause, and starting to make changes — rather than reading the specific action and resource the error already names. The error message is frequently the fastest path to the actual answer.

The Debugging Order

Revisiting `cloud1-6`'s explicit-deny-always-wins nuance, structured as an actual investigation sequence:

1. Confirm the identity is actually who you think it is — the role or account actually in use, not assumed.

2. List *all* policies attached to that identity — direct, via a group, via an assumed role — not just the obvious one.
3. Check for *any* explicit deny across all of them — one deny anywhere wins, regardless of allows granted elsewhere (`cloud1-6`).
4. Check resource-level restrictions and condition keys — is the allow scoped to a *different* specific resource than the one actually being accessed?
5. Check for a permission boundary or organization-level restriction that could override even a correctly-configured identity-level policy.

Permission Boundaries & Organization-Wide Policies

A layer `cloud1-6` didn't have room to cover properly: Service Control Policies (AWS), Azure Policy, and GCP Organization Policies apply at the **organization or account level** — above and entirely independent of any individual identity's own IAM policies. Even a user with a policy explicitly granting full administrator access can still be blocked by an org-level restriction, which is genuinely surprising to someone only checking the identity's own policies. This is a real, common source of confusing tickets where "the policy looks completely correct" and access is still denied — the missing piece is invisible at the identity level entirely.

A Common Real Scenario — Cross-Account/Cross-Resource Access

Revisiting `cloud1-6`'s cross-account role-assumption material in a troubleshooting context: a genuinely common real ticket pattern involves two *separate* policies that both need to be correct.

> **TRUST POLICY VS. PERMISSION POLICY — A COMMON, SUBTLE CONFUSION**
>
> A **trust policy** on the target role controls *who is allowed to assume it*. A **permission policy** controls *what they can do once assumed*. These are separate, independently configurable settings — a misconfigured trust policy prevents the role from ever being assumed at all (an error at the assumption step itself), while a misconfigured permission policy allows assumption to succeed but denies specific actions afterward. Confusing which one is broken wastes real troubleshooting time.

Practical Tools for IAM Debugging

Each provider offers policy simulation or testing tools — AWS's IAM Policy Simulator, Azure's "Check access" feature, GCP's Policy Troubleshooter — genuinely useful for confirming exactly why an action would or wouldn't be allowed, without needing to actually attempt the action for real against production.

A Support-Relevant Workflow — Putting It Together

```
1. CONFIRM IDENTITY        -- who is actually making this request
2. GATHER ALL POLICIES     -- direct, group, role — not just the obvious one
3. CHECK FOR EXPLICIT DENY -- one deny anywhere wins
4. CHECK RESOURCE/CONDITION SCOPING
5. CHECK ORG-LEVEL/BOUNDARY POLICIES
6. USE THE PROVIDER'S POLICY SIMULATOR TO CONFIRM
```

This is genuinely IAM troubleshooting's own version of Chapter 2's connectivity flowchart — the same disciplined, layer-by-layer structure, applied to a completely different kind of problem.

Hands-On Exercises

EXERCISE 1

A user has a policy that appears to grant full access to a service, and no explicit deny is found on any attached policy — yet access is still denied. What layer, not covered by checking the identity's own policies, should be checked next?

View solution

EXERCISE 2

Explain the difference between a trust policy and a permission policy in the context of assuming a role in another account, and describe what a misconfiguration in each would actually look like as a symptom.

View solution

EXERCISE 3

Explain why reading the full "access denied" error message — rather than just seeing "denied" and starting to guess — saves real time, using the specific pieces of information this chapter says such an error typically contains.

View solution

CHAPTER 3 QUICK REFERENCE

- This chapter assumes AuthN already works — it's entirely about AuthZ (`cloud1-6`)
- Read the full error: specific action + specific resource + (sometimes) which policy caused the denial
- Debugging order: confirm identity → gather all attached policies → check for explicit deny → check resource/condition scoping → check org-level/boundary policies

- **SCPs/Azure Policy/GCP Org Policies** sit above identity-level policies entirely — a correct-looking identity policy can still be overridden
- **Trust policy** (who can assume a role) ≠ **permission policy** (what they can do once assumed) — two separate, both-required settings
- Use the provider's own policy simulator (IAM Policy Simulator/Check access/Policy Troubleshooter) to confirm without testing against production
- **Next chapter:** Reading Logs & Metrics Under Pressure — building on `cloud1-8`'s metrics-then-logs workflow

CHAPTER 4 OF 10

Reading Logs & Metrics Under Pressure

Cloud Troubleshooting & Support

Chapter 4 · Reading Logs & Metrics Under Pressure

Connectivity (Chapter 2) and IAM (Chapter 3) covered specific problem categories. This chapter is about a skill that cuts across all of them: reading monitoring data effectively *while* an incident is actively happening, when time pressure genuinely changes how the investigation should proceed. It builds directly on `cloud1-8`'s metrics-then-logs workflow — the theory covered there, practiced properly here.

Restating the Foundation — Cloud1-8's Metrics-Then-Logs Workflow

Recall the sequence: metrics narrow down *when* and roughly *where*; logs explain *what* actually happened. This chapter is about doing that efficiently and correctly under real time pressure, not just knowing the theory.

The First 5 Minutes of an Incident

Resist the urge to start reading logs immediately. First, establish the metrics-level picture: what changed, when did it start, and how widespread is it — one instance or all of them, one region or all regions (`cloud1-1`'s regions/AZs). This initial triage determines the **scope** of the investigation before diving into detail. Skipping straight to logs risks getting lost in fine-grained detail for a problem that's actually much narrower — or much broader — than initially assumed.

Correlating Logs Across Multiple Services

A single user-facing failure often spans multiple services — a load balancer, web servers, a database (`cloud2-1`'s architecture-diagram material) — each potentially logging to a different aggregation location (`cloud1-8`). The practical technique: if the application logs a shared **correlation ID** (a request or trace ID) with each request, use it to filter every service's logs down to just the entries relevant to one specific failing request, rather than manually scanning unrelated traffic across each source separately. Without a correlation ID, fall back to narrowing by time window plus the likely source IP or resource identifier as the next-best approach.

Effective Log Query Techniques Under Time Pressure

- **Start broad, then narrow** — time range plus service first, then a specific error pattern or status code — rather than trying to write the "perfect" query immediately.
- **Absence can be diagnostic too** — a service that should log "request received" but doesn't suggests the request never actually reached it at all, a genuinely useful reframing that's easy to overlook.
- **Keep a small personal library of go-to queries** for common scenarios, rather than rebuilding syntax from memory during a live incident — a genuinely practical time-saving habit.

Distinguishing Correlation From Causation in Metrics

A real, common mistake under pressure: two things happening around the same time doesn't mean one caused the other. CPU spiking exactly when errors started could be the *errors* causing retries and extra load — a symptom, not the cause. The useful discipline: look at what changed *first* in the timeline, not just what looks abnormal right now, and stay willing to revise an initial hypothesis if later evidence doesn't fit it.

> **DON'T GRAB THE FIRST ABNORMAL-LOOKING METRIC AS "THE CAUSE"**
> Under real time pressure, it's genuinely tempting to seize on whatever metric looks the most dramatically wrong and call it the root cause. Confirming actual sequence — what changed first, not just what's most visually striking right now — is what actually separates a cause from a downstream symptom.

When Alert Fatigue Becomes a Real Problem Mid-Incident

Revisiting `cloud1-8`'s alert fatigue material specifically during an active incident: a flood of secondary, downstream alerts — services failing *because of* the root cause, not the root cause itself — can bury the one alert that actually points at the real problem.

> **LOOK FOR THE EARLIEST ALERT, NOT THE LOUDEST**
> Root causes typically alert before their downstream symptoms do. During a real incident with many simultaneous alerts firing, sorting by timestamp and starting from the earliest one is usually a faster path to the actual root cause than chasing whichever alert seems the most urgent or frequent right now.

Documenting What You Found, As You Go

A genuinely practical habit: jot down timestamps, findings, and ruled-out theories *as* the investigation happens, not reconstructed afterward from memory. This is directly useful for Chapter 6's incident response and postmortem material, and it also simply prevents re-checking the same thing twice during a long, stressful investigation.

Hands-On Exercises

EXERCISE 1

An incident starts. Before opening any logs, what metrics-level questions should be answered first, and why does skipping this step risk wasting time?

View solution

EXERCISE 2

A user-facing request fails, touching a load balancer, two application servers, and a database. Explain the correlation-ID technique for investigating this efficiently, and what to fall back on if no correlation ID exists.

View solution

EXERCISE 3

During an incident, CPU usage spikes at the same moment error rates spike. Explain why this alone doesn't tell you which one caused the other, and what additional evidence would help distinguish cause from symptom.

View solution

CHAPTER 4 QUICK REFERENCE

- First establish scope via metrics (what/when/how widespread) before diving into logs
- Use a correlation/trace ID to filter multiple services' logs down to one failing request; fall back to time window + resource identifier without one
- Start broad, narrow progressively; absence of an expected log entry is diagnostic too; keep a go-to query library
- Correlation ≠ causation — look at what changed **first** in the timeline, not just what looks most abnormal right now
- During an incident, find the **earliest** alert, not the loudest — root causes alert before their downstream symptoms
- Document timestamps/findings/ruled-out theories as you go — feeds directly into Chapter 6's postmortem material
- **Next chapter:** Common Failure Modes & Root Cause Analysis

CHAPTER 5 OF 10

Common Failure Modes & Root Cause Analysis

Cloud Troubleshooting & Support

Chapter 5 · Common Failure Modes & Root Cause Analysis

Chapter 4 covered investigation *technique*. This chapter catalogs the actual failure *patterns* a support engineer will run into repeatedly — recognizing a familiar pattern is genuinely faster than diagnosing every incident entirely from first principles.

Why a Catalog of Common Failures Is Worth Having

This isn't an exhaustive list — it's the failure modes common enough to be worth having pre-loaded, directly echoing Chapter 4's "keep a go-to query library" habit: this is the analogous library, but for recognizing failure *patterns* rather than writing queries.

Instance Failures

- **Hardware failure** — rare but real; the underlying host has an issue, sometimes surfaced in advance via provider-initiated retirement notices.
- **OS-level crashes/kernel panics** — often visible in system logs beforehand, if caught in time.
- **Out-of-memory (OOM) kills** — the OS killing a process to free memory, a genuinely common and diagnosable pattern via specific log signatures.
- **Health-check-triggered replacement** (`cloud1-3`/`cloud1-5`) — the instance may *look* like it failed when the health check itself was actually misconfigured (checking the wrong port or path) — a real, common false positive worth naming explicitly.

Storage Throttling

Revisiting `cloud1-4`: managed storage volumes typically have IOPS/throughput limits tied to volume size or type. A workload exceeding those limits gets **throttled**, not failed outright — performance degrades rather than producing an obvious error, which is genuinely confusing since nothing "broke" in the usual sense. The symptom pattern: increased latency correlating with high disk I/O metrics specifically, not CPU. The fix is usually resizing the volume or moving to a higher-performance storage tier — not debugging application code.

Database Connection Exhaustion

Revisiting `cloud1-7` as a diagnosable pattern: "too many connections" errors appearing suddenly under load, or right after a deployment that introduced a connection leak (connections not properly closed or returned to the pool). To confirm: check the database's current connection count against its configured maximum — a direct diagnostic. A genuinely common root cause worth naming specifically: a recent code deployment changing connection pooling behavior — directly tying back to Chapter 4's "what changed and when" discipline.

Auto-Scaling Misconfigurations

Revisiting `cloud1-3`'s auto-scaling masking material, now cataloged as specific, diagnosable patterns:

- **Scaling too slowly** — an overly conservative trigger or cooldown period means new instances come online after the spike has already caused user-visible failures.
- **Scaling on the wrong metric** — scaling on CPU when the real bottleneck is memory or database connections, so adding instances never actually relieves the real constraint.
- **Hitting the max instance limit** — `cloud1-3`'s own "eventually you hit the max" gotcha, now a directly checkable diagnostic: is the current instance count sitting right at its configured maximum?

A Root Cause Analysis Framework

The **5 Whys** technique: repeatedly ask "why did that happen" past the first, surface-level answer, to avoid stopping at a symptom rather than the actual root cause.

```
"The site was slow."
  Why? -- The database was slow.
  Why? -- The connection pool was exhausted.
  Why? -- A recent deployment introduced a connection leak.
  Why? -- Code review didn't catch it, because there's no automated test for connection cleanup.
```

That last answer is a genuine root cause worth fixing — versus stopping at "the database was slow" and just restarting it, which fixes nothing long-term.

Distinguishing a True Root Cause From a Contributing Factor

A genuinely useful nuance: sometimes there are *multiple* contributing factors, not one single clean root cause. In the example above, both the connection leak (a code bug) and an undersized connection pool (a config choice) could plausibly have contributed — fixing only one might reduce frequency without eliminating the

risk entirely. Worth being honest about this in a postmortem rather than declaring an artificially clean single cause.

PATTERN RECOGNITION IS A SKILL WORTH BUILDING DELIBERATELY

Recognizing a familiar failure pattern doesn't have to be something that "just comes with experience" passively — this chapter's catalog is a deliberate shortcut to that recognition, worth reviewing periodically rather than relying purely on accumulated exposure over time.

STOPPING AT THE FIRST "WHY" FIXES NOTHING LONG-TERM

Restarting the database or manually killing idle connections resolves the immediate symptom but leaves the actual root cause (per the 5 Whys example above) completely untouched — directly echoing `cloud1-3`'s auto-scaling-masking material: the underlying problem will simply recur, likely at a worse scale, until the real root cause is found and fixed.

Hands-On Exercises

EXERCISE 1

A database starts returning "too many connections" errors during a traffic spike, but the same traffic level was handled fine yesterday. Using this chapter's material, what's the most likely first thing to check, and why?

View solution

EXERCISE 2

Explain the difference between storage throttling and an outright storage failure — what each would look like in metrics, and why the correct fix differs between the two.

View solution

EXERCISE 3

Apply the "5 Whys" technique to this starting symptom: "users report the website is loading images slowly." Write out a plausible chain of at least 3 "whys" leading to a genuine root cause, not just a first-level symptom.

View solution

CHAPTER 5 QUICK REFERENCE

- **Instance failures** — hardware, OS crashes, OOM kills, and misconfigured health checks producing false-positive "failures"
- **Storage throttling** — degraded performance, not an outright error; latency correlates with disk I/O, not CPU; fix is resizing/tiering, not app debugging
- **Connection exhaustion** — check current vs. max connections directly; a recent deployment is a common root cause (tie to Ch.4's "what changed")
- **Auto-scaling misconfigurations** — too slow to react, scaling on the wrong metric, or hitting the configured max
- **5 Whys** — keep asking past the first symptom to find a genuine, fixable root cause
- Multiple contributing factors can coexist — don't force an artificially clean single cause in a postmortem
- **Next chapter:** Incident Response in the Cloud — severity triage, communication, escalation, postmortems

CHAPTER 6 OF 10

Incident Response in the Cloud

Cloud Troubleshooting & Support

Chapter 6 · Incident Response in the Cloud

Chapters 2-5 built the technical investigation skills. This chapter is about the *process* wrapped around an incident — managing it as an event with stakeholders, communication needs, and a defined lifecycle, not just diagnosing it technically.

Severity Triage — Not Every Incident Is the Same

Severity levels (commonly Sev1-Sev4 or P1-P4) are based on **impact**, not raw technical severity — a completely broken production system affecting every customer is a different category from a minor cosmetic bug, even if the underlying code change behind each was similarly small. Dimensions that matter: how many users are affected, whether a workaround exists, whether data is at risk (`cloud1-10`'s compliance material), and whether revenue or an SLA is at risk. Triage happens fast — often within minutes — and gets revised as more information comes in, not decided once and left fixed.

The Incident Commander Role

In a significant incident, one person is designated to **coordinate** the response — not necessarily the person actually fixing the technical problem. This role tracks status, makes communication decisions, and decides when to escalate further. Separating "who's fixing it" from "who's coordinating it" is genuinely valuable at scale, since someone deep in a technical fix often can't simultaneously manage communication well too.

Customer Communication During an Incident

- **Communicate early**, even with incomplete information — customers already know something is wrong; silence reads as worse than an honest "we're investigating."
- **Avoid speculating about root cause publicly** before it's actually confirmed — a genuinely common mistake; stating an unconfirmed cause and having to walk it back later damages trust more than simply saying "still investigating."
- **Maintain a regular update cadence**, even with nothing new to report ("still investigating, next update in 30 minutes"), rather than long silences.

"STILL INVESTIGATING" IS A COMPLETELY LEGITIMATE, PROFESSIONAL UPDATE

Genuinely reassuring permission for anyone newer to customer-facing incident communication: there's no obligation to have a definitive answer immediately. A clear, timely "we're still investigating, next update in 30 minutes" is a real, professional update — not an admission of failure.

Escalation Paths

When to escalate beyond the initial responder: hitting the limits of your own access or knowledge (echoing `cloud1-6`/`cloud2-3`'s least-privilege material — genuinely not having the access needed to fix something is fine, and escalating is the correct response, not a failure); the incident exceeding a severity/time threshold the organization has already defined; or needing a specialist (database, security) for something outside general troubleshooting scope. Escalating early when in doubt is usually right — the cost of an unnecessary escalation is much lower than the cost of a prolonged, unresolved incident.

When to Involve Cloud Provider Support

Sometimes the issue genuinely sits on the provider's side of Chapter 1's shared responsibility model — "of the cloud," not "in the cloud." Chapter 8 covers working with provider support in full; worth knowing here as one specific, legitimate escalation path.

Postmortems — Learning From the Incident

A postmortem happens after resolution, using the timeline and findings already documented during the investigation (Chapter 4's "document as you go" habit). The goal is genuinely **blameless** — focus on what allowed the failure to happen (process or system gaps), not who made a mistake. A good postmortem includes: the timeline, the root cause (Chapter 5's 5 Whys), contributing factors (Chapter 5's honesty about multiple factors), and concrete **action items with owners** — a postmortem without action items is just a story, not a learning process.

This connects directly to this site's own `owasp1-9` logging-failures category: good incident logging and documentation (Chapter 4) is what makes a genuinely useful postmortem possible at all — insufficient logging undermines incident forensics generally, exactly the point `owasp1-9` makes.

BLAME-FOCUSED POSTMORTEMS BACKFIRE

Focusing a postmortem on who made a mistake, rather than what process or system gap allowed the mistake to cause real impact, is a natural but genuinely counterproductive instinct — it makes people less likely to report issues honestly in the future, a well-documented pattern in incident response culture. A blameless approach produces more honest, more useful incident data over time.

A Support-Relevant Incident Lifecycle

```
1. DETECT           -- alerting (Ch.1-8 / cloud1-8)
2. TRIAGE SEVERITY   -- impact-based, revised as info arrives
3. DECLARE INCIDENT COMMANDER  -- if significant enough
4. COMMUNICATE       -- early and regularly
5. INVESTIGATE       -- Ch.2-5's techniques
6. MITIGATE / RESOLVE
7. POSTMORTEM        -- blameless, with action items
8. TRACK ACTION ITEMS TO COMPLETION
```

A full loop — not "fix it and move on."

Hands-On Exercises

EXERCISE 1

Two incidents occur at the same time: a typo on a rarely-visited help page, and checkout failing for 100% of customers. Explain how severity triage would differ between the two, and specifically which dimensions drove that difference — not just "one is obviously worse."

View solution

EXERCISE 2

Explain why communicating "we're still investigating, next update in 30 minutes" early in an incident is better practice than staying silent until a root cause is confirmed.

View solution

EXERCISE 3

Explain why a blameless postmortem culture tends to produce better incident data over time than a blame-focused one, even though it feels counterintuitive not to explicitly identify who made the mistake.

View solution

CHAPTER 6 QUICK REFERENCE

- **Severity triage** is impact-based (users affected, workaround exists, data/revenue/SLA at risk), decided fast, revised as info arrives

- **Incident commander** — coordinates the response, separate from whoever is actively fixing the technical problem
- Communicate early and regularly; never speculate publicly about an unconfirmed root cause
- Escalate when you hit the limits of your access/knowledge, a defined threshold is exceeded, or a specialist is needed — escalating early is cheap, a prolonged incident is expensive
- Provider support is a legitimate escalation path when the issue is genuinely "of the cloud" (Ch.1) — full coverage in Chapter 8
- **Blameless postmortems** — timeline, root cause (5 Whys), contributing factors, and action items with owners; blame-focused postmortems produce worse data over time
- Full lifecycle: detect → triage → (commander) → communicate → investigate → resolve → postmortem → track action items
- **Next chapter:** Cost Anomalies & Billing Support — diagnosing unexpected charges, runaway usage, common billing scenarios

CHAPTER 7 OF 10

Cost Anomalies & Billing Support

Cloud Troubleshooting & Support

Chapter 7 · Cost Anomalies & Billing Support

A different but genuinely common category of ticket: unexpected cost. This chapter builds directly on `cloud1-9`'s cost management foundations, turning that chapter's "usual suspects" list into a real, dedicated support investigation.

Restating the Foundation — Cloud1-9's Usual Suspects

Recall the list: idle/oversized compute, orphaned storage, data egress, unused load balancers/NAT gateways, forgotten dev/test environments. This chapter goes deeper into *how* to actually investigate each one as a real ticket.

The Billing Support Investigation Order

Echoing Chapter 2's connectivity flowchart and Chapter 3's IAM debugging order:

1. Confirm the **time window** of the increase precisely — which billing period, did it start suddenly or ramp gradually (Chapter 4's "what changed and when").
2. Break down spend by **service** first, then by **resource/tag** (`cloud1-9`'s tagging material).
3. Cross-reference the identified resource(s) against recent provisioning/deployment history — was something new created around when the increase started.
4. Check the identified resource(s) against the specific usual-suspect patterns below.

Investigating Idle/Oversized Compute

Compare instance size/type against actual CPU/memory utilization metrics (`cloud1-8`) over a representative period. A genuinely common finding: an instance sized for an anticipated peak load that never materialized, running oversized 24/7 unnecessarily. The fix is **rightsizing**, not simply "turn it off," if the resource is still genuinely needed — just wrong-sized.

Investigating Orphaned Storage

List storage volumes not currently attached to any running instance, and cross-reference against recently terminated instances (`cloud1-3`'s own stopped-VM-billing gotcha, taken further). A genuinely common real finding: a batch of disks left behind after an environment teardown that removed the compute but never the associated storage.

Investigating Data Egress Costs

Break down data transfer costs specifically by **destination** — cross-region, cross-AZ, or out to the internet — since these are priced very differently. A genuinely common real finding: a misconfigured backup/replication job sending far more data cross-region than intended, or a CDN/caching misconfiguration causing repeated re-fetching of the same data from origin rather than being served from cache — directly the scenario `cloud1-9`'s own multi-cloud/hybrid egress warning anticipated.

Investigating Forgotten Dev/Test Environments

Cross-reference resource creation date and tags (or the *lack* of tags — `cloud1-9`'s own tip-box) against any currently active project. A genuinely common real finding: an environment spun up for a now-completed or abandoned project, still running because nobody remembered to tear it down — often discoverable specifically *because* it lacks proper tagging in the first place, closing the loop on `cloud1-9`'s own tagging argument in a concrete investigative context.

When the Cause Isn't a "Usual Suspect" — Pricing/Rate Changes

A genuinely distinct category worth naming separately: sometimes cost increases with *no* change in actual resource usage at all, because the provider changed pricing for a service, or a discount/committed-use agreement (`cloud1-9`'s pricing-model material) expired and usage reverted to on-demand rates. Important to rule in or out early, since no amount of resource-level investigation explains a cost increase that's actually a pure pricing change.

> **CHECK RESERVED/COMMITTED-USE EXPIRATION FIRST**
> A genuinely common, easy-to-miss real scenario: a reserved instance or committed-use discount quietly expiring produces exactly the same-shaped cost jump as a "usual suspect" resource problem, but with literally zero change in actual resource usage — checking this specifically, and early, can save an entire investigation from heading down the wrong path.

Communicating Findings to a Customer

Tying back to Chapter 6's communication material: present findings clearly — *what* resource, *why* it's costing more, and the specific recommended action (delete/resize/re-tag), rather than a raw cost breakdown dump. Genuinely useful practice: distinguish clearly between "this is a legitimate cost from real, needed usage" and "this is something that can likely be safely reduced or eliminated," since customers need a different follow-up action for each.

NEVER RECOMMEND DELETION WITHOUT CONFIRMING IT'S ACTUALLY SAFE

Unlike most cost-investigation actions, deleting the wrong resource has real, hard-to-reverse consequences. A storage volume that looks orphaned might be a deliberate backup; an idle-looking instance might be a rarely-used but genuinely needed system. Always confirm with the customer before recommending removal — don't jump straight to "just delete it."

Hands-On Exercises

EXERCISE 1

A customer's bill jumped 40% this month with literally zero change reported in their actual application usage or traffic. Using this chapter's material, what's a genuinely important category to rule out first, and why does it not require any resource-level investigation to check?

View solution

EXERCISE 2

A customer has several storage volumes not attached to any running instance. Explain how you'd confirm whether these are genuinely orphaned versus intentionally kept (e.g. a deliberate backup strategy) before recommending they be removed.

View solution

EXERCISE 3

Explain why data egress costs specifically need to be broken down by destination (cross-region vs. cross-AZ vs. internet) rather than just looking at total data transfer volume alone.

View solution

CHAPTER 7 QUICK REFERENCE

- Investigation order: confirm time window → break down by service, then resource/tag → cross-reference provisioning history → check against usual-suspect patterns
- **Idle/oversized compute** — rightsize, don't just delete if it's still needed
- **Orphaned storage** — cross-reference against recently terminated instances (Ch.3's billing gotcha, extended)
- **Data egress** — break down by destination; misconfigured cross-region replication or cache misses are common real causes
- **Forgotten dev/test environments** — often discoverable specifically because they lack tags
- **Pricing/rate changes** (reserved/committed-use expiration) — check this first; produces the exact same symptom with zero usage change
- Always confirm a resource is genuinely safe to remove before recommending deletion
- **Next chapter:** Working With Cloud Provider Support — understanding support tiers/SLAs, escalating from the provider side

CHAPTER 8 OF 10

Working With Cloud Provider Support

Cloud Troubleshooting & Support

Chapter 8 · Working With Cloud Provider Support

Chapter 6 covered internal escalation. This chapter covers the other side of it: engaging the cloud provider's own support organization directly, when an issue genuinely sits on their side of Chapter 1's shared responsibility model — "of the cloud," not "in the cloud."

When to Actually Engage Provider Support

Genuinely provider-side issues: a documented service outage, a hardware failure affecting a specific resource (`cloud2-5`'s instance-failure material), or behavior that contradicts documented service guarantees. **Not** provider support's job: your own IAM misconfiguration (`cloud2-3`), your own application bugs, or architecture-optimization advice for your own choices (though some higher support tiers do offer this as a paid add-on — worth distinguishing from break/fix support). A practical gut check: is this something only the provider can fix, or something fixable internally with the right access or knowledge? If the latter, Chapter 6's internal escalation is the right path, not a provider case.

Support Tiers, Compared

PROVIDER	TIERS
AWS	Basic (free, no technical support) → Developer → Business → Enterprise
Azure	Basic → Developer → Standard → Professional Direct → Premier
GCP	Basic → Standard → Enhanced → Premium

The general pattern: higher tiers mean faster guaranteed response times, broader scope (architecture guidance, not just "is this broken"), and often a named technical account manager at the top tier. Which tier an organization has directly determines what's actually possible to get from a support case — worth knowing before setting expectations with a customer.

Severity Levels in Provider Support Cases

Revisiting Chapter 6's own internal severity triage, now applied to the provider's side: providers have their own severity definitions (e.g. "production system down" vs. "general guidance") that determine *their* response-time commitment, entirely separate from an organization's internal severity classification. Accurately classifying severity in the case itself genuinely matters — an inflated severity may simply get reclassified by the provider rather than fast-tracked, while an understated one produces a slower response than the actual situation needs.

What Makes a Good Support Case

- The **specific resource ID/ARN**, not just "my service."
- The **exact error message**, not a paraphrase.
- A precise **timeline** of when it started and what, if anything, changed around that time — Chapter 4's own documentation habit pays off directly here.
- What's **already been tried or ruled out** — saves the support engineer from suggesting things already tested.

> **VAGUE CASES GET VAGUE, SLOW RESPONSES**
>
> "It's slow" or "something's wrong" forces the provider's own support engineer to do the exact same scoping work Chapter 4 covers before they can even begin helping — gathering that information up front, rather than in a slow back-and-forth, directly speeds up resolution.

Understanding SLA Credits

If a provider fails to meet its own published SLA (a documented uptime guarantee, for instance), customers are typically entitled to service credits — but this is **not** usually automatic. It typically requires actively filing a claim with specific supporting evidence, directly rewarding the documentation habit from Chapters 4 and 6. Assuming credits happen automatically is a genuinely common misconception worth correcting directly.

Reading Provider Status Pages & Health Dashboards

> **CHECK THE ACCOUNT-SPECIFIC HEALTH DASHBOARD BEFORE OPENING A CASE**
>
> Many providers offer an account-specific health view, distinct from the generic public status page, showing issues actually affecting your own resources specifically. Checking this before opening a case can immediately confirm a known, already-being-addressed issue — saving time on both sides, and a genuinely good habit to build into the very start of any investigation, echoing Chapter 1's own shared-responsibility framing.

Working a Case Collaboratively

- Respond promptly to support's follow-up questions — cases can get deprioritized or closed for inactivity.
- Provide the specific additional diagnostic information requested, rather than re-explaining the original problem from scratch.
- If the case involves genuinely time-sensitive production impact, say so explicitly and make sure the case's severity actually reflects that.

Hands-On Exercises

EXERCISE 1

A customer reports "the cloud provider's service is broken." Using this chapter's material, what specific questions need answers before this is even ready to become a support case, and why does a vague case get a slower response?

View solution

EXERCISE 2

Explain the difference between an organization's own internal severity classification (Chapter 6) and the severity classification within a provider support case. Why are these two separate things, and why does accurately setting the provider-side severity matter?

View solution

EXERCISE 3

Explain why checking a provider's account-specific health dashboard before opening a support case is a genuinely useful habit, even when you're confident there's a real problem.

View solution

CHAPTER 8 QUICK REFERENCE

- Engage provider support for genuinely "of the cloud" issues; internal escalation (Ch.6) covers everything else
- Support tiers determine what's actually possible — response time, scope, and named account management
- Provider-side severity is separate from internal severity — inflating it risks reclassification, understating it slows response
- Good cases: exact resource ID, exact error message, precise timeline, what's already been ruled out

- SLA credits require an actively-filed claim with evidence — not automatic
- Check the account-specific health dashboard before opening a case — it can immediately confirm a known issue
- Respond promptly, provide exactly what's requested, and make sure case severity reflects real production impact
- **Next chapter:** Multi-Cloud & Hybrid Environments — the support challenges these architectures create

CHAPTER 9 OF 10

Multi-Cloud & Hybrid Environments

Cloud Troubleshooting & Support

Chapter 9 · Multi-Cloud & Hybrid Environments

Chapters 2-8 largely assumed a single provider. This chapter addresses what happens when a support engineer has to work across multiple providers, or a hybrid on-prem/cloud environment, simultaneously — genuinely common in real support work, building on the honest framing `cloud1-1` and `cloud1-12` already established.

Restating the Foundation — Why Multi-Cloud/Hybrid Happens

Per `cloud1-1`/`cloud1-12`: deliberate reasons (compliance, latency, legacy systems, risk diversification) versus accidental reasons (acquisitions, uncoordinated team choices over time). Either way, the support burden that results is identical — `cloud1-12`'s own point, restated here because it's about to become directly practical rather than theoretical.

Hybrid Connectivity — How On-Prem and Cloud Actually Connect

- **Site-to-site VPN** — an encrypted tunnel over the public internet; cheaper, but with variable performance and latency.
- **Dedicated connections** — AWS Direct Connect, Azure ExpressRoute, GCP Dedicated/Partner Interconnect: a private, dedicated physical connection to the provider's network, bypassing the public internet entirely — more expensive, but with consistent, reliable performance, often used for latency-sensitive or high-volume workloads.

A genuinely important distinction for troubleshooting: which connectivity type is in use directly determines what's actually diagnosable from the cloud side alone versus what genuinely requires on-prem network team involvement.

The Terminology-Mapping Problem, Compounded

Revisiting `cloud1-2`'s terminology map — now as a genuine support-time cost, not just a learning curve. A support engineer fluent in AWS-specific troubleshooting (Chapters 2-7's techniques) has to consciously re-map every one of those techniques to Azure or GCP's equivalent concepts and tools mid-investigation, when

working a genuine multi-cloud incident. This genuinely slows investigation compared to a single-provider environment — worth being honest about, rather than pretending cross-provider skill transfer is seamless.

Correlating Issues Across Providers

Revisiting Chapter 4's correlation-ID technique, now across provider boundaries: if a workflow spans AWS and Azure (or on-prem and cloud), each side almost certainly has completely separate logging and monitoring systems (`cloud1-8`'s own point about log query languages differing). A shared correlation ID has to be **deliberately propagated** across that boundary by the application itself — the providers' own systems won't do this automatically the way they might within a single provider's own service mesh. Without that, falling back to time-window plus resource identifier across two completely separate monitoring systems is genuinely more manual and error-prone than Chapter 4's single-provider version.

Multi-Cloud Cost & Networking Overlap

Revisiting `cloud1-9`'s egress warning directly, now as a common incident *trigger*: data moving between cloud providers, or between on-prem and cloud, incurs egress charges on the sending side — a genuinely common real finding when investigating a cost anomaly (Chapter 7) in a hybrid or multi-cloud environment specifically. Chapter 7's techniques apply directly here, with this added wrinkle worth checking explicitly.

Whose Responsibility Is It — A Genuinely Harder Version of Chapter 1

In a single-provider environment, Chapter 1's "of the cloud" vs. "in the cloud" line is relatively clean. In a hybrid or multi-cloud environment, a **third zone** emerges — the connectivity and integration layer *between* environments — which may not clearly belong to either provider's own responsibility model, or to on-prem IT, at all.

> **THE CONNECTIVITY LAYER'S OWNERSHIP AMBIGUITY IS A REAL, COMMON SOURCE OF DELAY**
>
> During an incident spanning a VPN or dedicated connection, it's genuinely easy for both the cloud team and the on-prem network team to each assume the other owns investigating that specific link — producing real finger-pointing and delay precisely because Chapter 1's clean two-way responsibility split doesn't automatically extend to cover this third, in-between zone without someone explicitly deciding it does.

A Practical Approach to Multi-Cloud/Hybrid Support

1. Identify early which environment(s) are actually involved, before assuming techniques from one provider apply directly.

2. Check the connectivity type (VPN vs. dedicated connection) — it determines what's diagnosable from where.
3. Expect to consciously re-map terminology and tools, rather than assuming muscle memory transfers.
4. Look for a correlation ID; failing that, fall back to time-window correlation across each separate system.
5. Remember egress costs specifically when investigating cost anomalies in these environments.
6. Be explicit about who owns the connectivity/integration layer — don't assume it's automatically covered by an existing responsibility split.

THIS IS GENUINELY HARDER SUPPORT WORK, NOT JUST CONCEPTUALLY MORE COMPLEX TO LEARN ABOUT

`cloud1-12` made the point that multi-cloud's operational cost applies regardless of how an organization ended up there. Worth restating directly here: every technique this course has built — connectivity troubleshooting, IAM debugging, log correlation, cost investigation — still works in a multi-cloud/hybrid environment, it just takes real, deliberate extra effort to apply consistently across boundaries that don't share tooling, terminology, or a single clean responsibility model.

Hands-On Exercises

EXERCISE 1

An organization's request workflow spans an on-prem system and an AWS-hosted service, with no correlation ID built into the application. Using this chapter's material, what's the practical fallback approach for investigating a failure in this workflow, and what limitation does it have compared to Chapter 4's single-provider version?

View solution

EXERCISE 2

Explain why a cost anomaly investigation (Chapter 7) in a hybrid environment specifically should include checking egress costs, even if nothing about compute, storage, or tagging looks unusual.

View solution

EXERCISE 3

A production incident spans a VPN connection between on-prem infrastructure and a cloud VPC, and neither the on-prem network team nor the cloud team is certain who should own investigating the connection itself. Using this chapter's material, explain why this ambiguity happens, and what should be done about it.

View solution

CHAPTER 9 QUICK REFERENCE

- Deliberate or accidental, multi-cloud/hybrid's support burden is the same either way (`cloud1-12`)
- **VPN** (cheaper, variable) vs. **dedicated connections** (Direct Connect/ExpressRoute/Interconnect — pricier, consistent) — determines what's diagnosable from the cloud side alone
- Terminology re-mapping mid-investigation genuinely slows things down — a real cost, not just a learning-curve inconvenience
- Correlation IDs must be deliberately propagated across provider/on-prem boundaries — nothing does this automatically
- Cross-environment egress costs are a common, easily-missed cost anomaly trigger
- A third "connectivity layer" responsibility zone exists between environments — ownership must be explicitly assigned, not assumed
- **Next chapter:** Capstone — Diagnosing a Real Multi-Service Outage, combining every technique from this entire course

CHAPTER 10 OF 10

Capstone: Diagnosing a Real Multi-Service Outage

Cloud Troubleshooting & Support

Chapter 10 · Capstone — Diagnosing a Real Multi-Service Outage

One full incident, worked end to end, combining every technique from this course's nine prior chapters — and, throughout, the Course 1 material each of those techniques was originally built on.

The Scenario

TechShop, a mid-size e-commerce company, launches a major flash sale. Twenty minutes in, checkout starts failing intermittently for a growing share of customers.

Step 1 — Detect & Triage

An error-rate alarm fires on the checkout service (`cloud1-8`'s alerting). Severity triage (Chapter 6) is immediate: checkout is directly revenue-generating, a large and growing share of customers are affected, there's no workaround, and it's happening during a promotional event — this is a Sev1 without hesitation. An incident commander is declared.

Step 2 — Scope Before Diving Into Logs

Following Chapter 4's first-five-minutes discipline: what changed, when did it start, how widespread is it. The picture that emerges — errors are *intermittent*, not total, and started almost exactly when flash-sale traffic began climbing. This intermittent pattern is the same diagnostic signal `cloud2-1`'s own exercise walked through: a symptom pointing at something failing under load, not a total, clean outage.

Step 3 — Diagnosing Connectivity? Ruled Out Quickly

Applying Chapter 2's flowchart briefly: security groups, NACLs, and route tables are all checked and confirmed unchanged recently — no networking deployment in the relevant window. This step returns clean fast, which is itself useful information: it's genuinely fine, and expected, for a flowchart step to rule something out quickly rather than assuming every incident must be a networking problem.

Step 4 — Correlating Logs Across Services

Using Chapter 4's correlation-ID technique: request IDs are traced from the load balancer, through the application servers, to the database layer. The pattern that emerges — every failing request shows a **connection timeout** from the application server to the database, specifically.

Step 5 — Recognizing the Failure Pattern

This matches Chapter 5's catalog directly: **database connection exhaustion**. Confirmed by checking the database's current connection count against its configured maximum — sitting at 100%. Applying "what changed": a deployment two days earlier changed the application's connection pooling library. Normal, pre-sale traffic never came close to the new pool's limit, so the change went unnoticed until flash-sale traffic pushed it over.

Step 6 — Applying the 5 Whys

```
"Checkout is failing."
  Why? -- Requests are timing out connecting to the database.
  Why? -- The database connection pool is exhausted.
  Why? -- The pool size was left at the new library's small default value.
  Why? -- Nobody noticed, because normal traffic never approached that limit.
  Why? -- The deployment two days ago was never load-tested at flash-sale-level traffic before release.
```

The genuine root cause: a missing load-testing step in the deployment process — not simply "the pool was too small," which is only a symptom of that deeper gap.

Step 7 — Immediate Mitigation vs. Long-Term Fix

Per Chapter 5's own distinction: the **immediate mitigation** is increasing the connection pool size and restarting affected instances to clear stuck connections, restoring service quickly. The **long-term fix** — adding load-testing to the deployment process — is the actual root cause from Step 6. Both matter; stopping at the mitigation alone, per Chapter 5's warn-box, leaves the underlying gap ready to cause the same failure again.

Step 8 — Communicating Throughout

Per Chapter 6: an early "we're investigating" update goes out within minutes of the Sev1 declaration. Regular updates follow on a set cadence. No public speculation about cause happens until Step 5-6 actually confirm it. A clear resolution update follows once the mitigation is applied.

Step 9 — A Related Cost Question Surfaces

During the investigation, someone notices the auto-scaling group scaled out significantly. Is this legitimate cost from genuine load, or `cloud1-3`'s own auto-scaling-masking-a-root-cause gotcha? Here, it's honestly both — the scaling was a reasonable reaction to real load, but it was also masking the connection pool issue underneath the whole time, exactly the pattern `cloud1-3` warned about. Per `cloud1-9`'s idle-compute material, the extra instances should be scaled back down once the real fix is in place, not left running unnecessarily.

Step 10 — Was This Us, or the Provider?

Applying Chapter 1's shared responsibility model, checked early per Chapter 8: the provider's account-specific health dashboard showed nothing. This was entirely a customer-side application and configuration issue — not a provider outage — closing off the provider-support escalation path from Chapter 8 as unnecessary here.

The Postmortem

A blameless postmortem (Chapter 6) is written: timeline, the 5 Whys root cause chain from Step 6, and an honest note on contributing factors (Chapter 5) — both the small default pool size *and* the missing load-testing process contributed, not one clean single cause. Action items with owners:

1. Immediately fix the connection pool size — done, during the incident.
2. Add load-testing to the deployment pipeline — owned by the platform team, with a due date.
3. Add a dedicated connection-count alert (`cloud1-8`), so this shows up as its own early signal next time, rather than only manifesting as checkout errors after the fact.

What This Capstone Demonstrated

STEP	CHAPTER APPLIED
1, 8	Chapter 6 — severity triage, communication, postmortem
2	Chapter 4 — scope before logs
3	Chapter 2 — connectivity flowchart
4	Chapter 4 — correlation IDs
5, 6, 7	Chapter 5 — failure catalog, 5 Whys, mitigation vs. fix
9	Chapter 7 — cost investigation

STEP	CHAPTER APPLIED
10	Chapter 8 — provider vs. customer triage

And underneath every one of those: Course 1's own material — `cloud1-1`'s shared responsibility, `cloud1-3`'s auto-scaling and connection concepts, `cloud1-7`'s connection management, `cloud1-8`'s monitoring, `cloud1-9`'s cost management — the foundation this entire course was built to turn into practiced, applied skill.

WHY A FULL WORKED SCENARIO, NOT JUST ISOLATED EXERCISES

Real incidents rarely respect chapter boundaries — this capstone deliberately shows every technique from this course being reached for in sequence, in the order a genuine investigation would actually use them, rather than practiced in isolation.

Hands-On Exercises

EXERCISE 1

Identify which specific chapter of this course each of the following investigation actions came from: (a) checking metrics before opening logs, (b) using a request ID to trace across services, (c) confirming this wasn't a provider-side issue via the health dashboard.

View solution

EXERCISE 2

In this capstone's own root cause chain, identify the difference between the immediate mitigation and the long-term fix, and explain why a postmortem needs to include both rather than being satisfied with just the mitigation.

View solution

EXERCISE 3

A colleague suggests: "since the auto-scaling actually kept the site technically online during the incident, maybe we don't need to change anything." Using this chapter's and `cloud1-3`'s material, explain what's wrong with this reasoning.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Full incident lifecycle applied end to end: detect → triage → scope → investigate (connectivity, logs, failure patterns) → 5 Whys → mitigate + fix → communicate → cost check → provider-vs-customer triage → postmortem
- A mitigation restores service; a fix addresses the actual root cause — postmortems need both named separately
- Auto-scaling "keeping the site online" can still be masking an unaddressed root cause underneath (`cloud1-3`)
- **Course complete** — Cloud Troubleshooting & Support, 10 chapters, completing the full Cloud Platforms track (Course 1 + Course 2, 22 chapters)