Web & Application Troubleshooting

A Complete 10-Chapter Technical Support Course

Topics covered:

Reading 5xx errors as a diagnostic language · connection pool exhaustion

Caching bugs & cache stampede · sessions in load-balanced setups

Slow queries & N+1 · deployment version skew & migrations

Health checks, graceful shutdown & rate limiting

Capstone: three real application tickets, triaged end to end

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

From Layer to Symptom: What This Course Adds

Web & Application Troubleshooting

Chapter 1 · From Layer to Symptom: What This Course Adds

"The checkout API is throwing 500 errors, intermittently, during busy periods." One engineer's first instinct is to check the network and the server's own resources — reasonable instincts, and exactly what **Network Troubleshooting** and **System Monitoring & Performance Diagnosis** teach. The other engineer checks those too, finds them clean, and knows there's a whole category of genuine causes that live one layer higher — specific to running an actual application, not the machine or network underneath it. This course is entirely about that layer.

What the Other Three Courses Already Cover

COURSE	WHAT IT DIAGNOSES
Logging & Log Analysis (`log1`)	Reading logs correctly — levels, locations, correlation, and two full troubleshooting walkthroughs
Network Troubleshooting (`netdiag1`)	DNS, reachability, ports, firewalls, proxies/VPN/NAT, and reading HTTP/TLS results at the network boundary
System Monitoring & Performance Diagnosis (`perfdiag1`)	CPU, memory, disk, and recognizing when a resource — not the application itself — is the bottleneck

This course assumes those three are either already ruled out or being checked in parallel — it doesn't re-teach any of them. What it covers instead is the set of things that can genuinely go wrong even when the network is clean, resources are healthy, and the logs don't show an obvious smoking gun.

The Genuine Gap: What Lives Above the Resource Layer

A short preview of what the rest of this course actually covers — all of it specific to running a web application, none of it explained by network or raw resource exhaustion: database connection pool exhaustion, caching bugs (stale data and cache stampedes), session and state issues in load-balanced environments, slow queries and N+1 patterns, deployment-related version skew, health checks and graceful shutdown, and rate limiting.

```
ERROR: could not obtain connection from pool: timeout after 30000ms
       (pool size: 20, active: 20, waiting: 14)
```

Nothing about this error involves the network, the CPU, or the disk — the machine is fine. You'll learn to recognize and resolve exactly this pattern in Chapter 3.

A Concrete Example: The Same "500 Error," Six Different Layers

"The API returned a 500" is one symptom that genuinely could be caused by something in any of this subject's four courses — worth seeing side by side, so this course's own boundary is clear from the start:

ACTUAL CAUSE	DIAGNOSED BY
A firewall silently blocking the app's connection to its database	`netdiag1`
The server genuinely out of memory, the process OOM-killed mid-request	`perfdiag1`
An unhandled exception, with the real cause visible only in the application's own stack trace	`log1`
A slow query holding a connection too long, exhausting the pool for everyone else	**This course** — Chapters 3, 6
A stale cache entry serving corrupted data after an update	**This course** — Chapter 4
A half-rolled-out deployment, old and new code running side by side	**This course** — Chapter 7

Six genuinely different root causes, the exact same reported symptom. Knowing which of the four courses actually owns a given cause is most of the battle — this table is worth remembering as later chapters build out the three "this course" rows in full.

DON'T SKIP STRAIGHT TO "IT MUST BE A CODE BUG" EITHER

It's just as easy to over-correct the other direction — assuming every 500 error is automatically an application-layer problem and diving straight into connection pools or caching, skipping the network and resource checks entirely. The discipline this whole subject shares — check first, in the right order, rather than guessing which layer feels most likely — still applies here. This course's own content is real and specific, but it isn't the first place to look on every ticket.

"ONLY HAPPENS UNDER LOAD" IS ITSELF A CLUE

A problem that appears only during busy periods and disappears when traffic is light is a strong hint toward something with a fixed capacity — a connection pool, a cache, a rate limiter — rather than a straightforward code bug that would fail the same way regardless of load. Chapters 3 and 4 build directly on this instinct.

The Shared Discipline Still Applies

Even though the content here is genuinely new, the method carrying it isn't: scope the complaint before naming a cause, gather real evidence rather than guessing, and don't take an action (a restart, a redeploy) that destroys the evidence needed to actually understand what happened — the same principles `log1`, `netdiag1`, and `perfdiag1` each opened with in their own first chapters, still doing the same work here.

What This Course Covers

Reading the 5xx family as a genuine diagnostic language, database connection pool exhaustion, caching bugs, session/state issues in load-balanced setups, slow queries and N+1 patterns, deployment-related version skew, health checks and graceful shutdown, and rate limiting. The capstone applies all of it to three realistic application tickets.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own six-cause table, why "the API returned a 500" isn't enough on its own to say which of this subject's four courses actually diagnoses the real cause.

View solution

EXERCISE 2

Explain why this chapter warns against assuming every 500 error is automatically an application-layer problem, even though this course's own content is real and specific.

View solution

EXERCISE 3

A problem only appears under heavy load and disappears when traffic is light. Explain what this chapter says that pattern suggests, and why.

View solution

CHAPTER 1 QUICK REFERENCE

- This course assumes **network (`netdiag1`), resources (`perfdiag1`), and general log-reading (`log1`)** are ruled out or checked in parallel — it doesn't re-teach any of them

- The genuine gap this course fills: **connection pools, caching, sessions, deployments, health checks, rate limiting** — none covered by the other three
- The same symptom ("500 error") can genuinely belong to any of the four courses — know which one before diving in
- Don't over-correct either direction: don't skip application-layer checks, and don't skip network/resource checks either
- A problem that only appears **under load** often points at something with a fixed capacity, not a straightforward code bug
- **Next chapter:** Reading 5xx Errors as a Diagnostic Language

CHAPTER 2 OF 10

Reading 5xx Errors as a Diagnostic Language

Web & Application Troubleshooting

Chapter 2 · Reading 5xx Errors as a Diagnostic Language

Network Troubleshooting's own Chapter 9 already covered what 500, 502, 503, and 504 mean from a reverse proxy's point of view — a genuine, valuable finding: any HTTP response at all, even an error, proves DNS, the network path, the port, and TLS all worked. This chapter starts exactly where that one stopped: once you know it's a genuine application-layer 500 (not a proxy failing to reach its backend at all), what does the application's *own* response actually tell you?

Recap: The Proxy-Level View

CODE	`NETDIAG1`'S OWN MEANING
500	A generic failure inside the application itself
502	The proxy got an invalid response from its backend
503	Deliberately unavailable — often overload or maintenance mode
504	The proxy's upstream never responded in time

That table answers "did the request reach the application, and did something come back." This chapter is about the 500s that *do* reach the application — and specifically, what the application chose to say about it.

500 Isn't One Thing: Reading the Error Body

A well-built API rarely returns an empty 500 — it returns a structured error body, and that body is often far more informative than the bare status code. Worth being honest about a genuine, common wrinkle first:

APPLICATIONS FREQUENTLY MISUSE STATUS CODES

A "500" doesn't always mean the server genuinely failed — plenty of applications return 500 for things that are really the client's fault (a validation error that should have been a 400) or an ordinary business-logic outcome (like "insufficient inventory") that shouldn't be an error status at all. Don't assume every 500 automatically implicates the server; reading the actual error body is what tells you whether this is a real server failure or a status code that was simply chosen carelessly.

The Correlation ID: A Direct Line to the Right Log Entry

Well-designed APIs return a request ID or correlation ID with every response — in the body, a header like `X-Request-Id`, or both — specifically so a user-facing error can be matched to the exact server-side log line that explains it. This is a genuine upgrade over **Logging & Log Analysis**'s own timestamp-correlation technique: instead of narrowing down logs by "roughly when this happened," a correlation ID lets you grep for the *exact* request, no ambiguity at all.

```
$ grep "a1b2c3d4-e5f6" /var/log/app/service.log
[ERROR] request_id=a1b2c3d4-e5f6 Could not obtain a database connection within 30000ms
```

All Requests Failing vs. Some Requests Failing

A genuinely useful distinguishing question before reading a single error body: what fraction of requests to this endpoint are actually failing?

PATTERN	WHAT IT USUALLY POINTS TO
100% of requests fail, suddenly	Something broke universally — often a deployment (Chapter 7), a config error, or a missing dependency hit by every request
A small, intermittent percentage fails	A resource-contention or timing-sensitive cause — connection pool exhaustion (Chapter 3), a cache stampede (Chapter 4), or a specific edge-case input

This single distinction alone points you toward roughly half of this course's own remaining chapters before you've read a single log line.

CHECK THE RESPONSE HEADERS, NOT JUST THE BODY

A `Retry-After` header (mentioned in `netdiag1`'s own 503 coverage) is a genuine signal the application is deliberately telling clients to back off, not silently breaking. A response body that explicitly marks itself `"retryable": true` or `false` is even more direct — the application is telling you, in plain terms, whether this specific failure is expected to resolve on its own.

AN EMPTY, GENERIC ERROR BODY IS ITSELF A FINDING

A bare "Internal Server Error" with no correlation ID and no structured detail doesn't just fail to help with this particular ticket — it reveals that the application's own error handling is genuinely weak. That's worth flagging as its own improvement, separate from whatever actually caused this incident, echoing `log1`'s own point that a silent failure isn't the same as no failure — it just means nobody chose to record useful detail about it.

Working Example: Reading Chapter 1's Own Ticket

Chapter 1 left an open ticket: the checkout API returning intermittent 500s under load, with network and resources already ruled out. This chapter's own techniques resolve it immediately — the error body itself:

```
{
  "error": "pool_timeout",
  "message": "Could not obtain a database connection within 30000ms",
  "request_id": "a1b2c3d4-e5f6-...",
  "retryable": true
}
```

No log-grepping needed — the application is telling us directly: this is a connection pool timeout, it's expected to be transient (`"retryable": true`), and a correlation ID is available if deeper investigation is still needed. Combined with the "some requests, not all" failure pattern from earlier in this chapter, this confirms the exact category Chapter 1 previewed. Chapter 3 covers connection pool exhaustion in full — reading this exact ticket through to its actual fix.

Hands-On Exercises

EXERCISE 1

Explain why this chapter warns against assuming every 500 status code automatically means the server genuinely failed.

View solution

EXERCISE 2

Explain why a correlation ID is described as a genuine upgrade over `log1`'s own timestamp-based correlation technique.

View solution

EXERCISE 3

In this chapter's worked example, explain what the error body directly revealed, and what two other pieces of evidence in this chapter both agreed with that finding.

View solution

CHAPTER 2 QUICK REFERENCE

- This chapter picks up where `netdiag1`'s proxy-level 500/502/503/504 breakdown leaves off — reading what the application itself says
- Applications frequently misuse status codes — a 500 isn't automatically a genuine server failure
- A **correlation ID** lets you grep for the exact log entry, no timestamp ambiguity
- **100% failure, sudden** = usually a deployment; **a small, intermittent %** = usually a resource-contention or timing cause
- Check headers too — `Retry-After` and an explicit `"retryable"` field are genuine signals, not noise
- An empty, generic error body is itself a finding — weak error handling, worth flagging separately from the incident
- **Next chapter:** Database Connection Pool Exhaustion

CHAPTER 3 OF 10

Database Connection Pool Exhaustion

Web & Application Troubleshooting

Chapter 3 · Database Connection Pool Exhaustion

Chapter 2 fully identified the checkout ticket's error: `pool_timeout`, retryable, correlation ID in hand. This chapter is about actually resolving it — connection pools, why they exist, the two genuinely different reasons one runs dry, and why the tempting quick fix can make the real cause worse.

What a Connection Pool Is and Why It Exists

Opening a new database connection is expensive — a TCP handshake, authentication, session setup — expensive enough that doing it fresh for every single request would be far too slow. A connection pool solves this by keeping a fixed set of connections already open, which application code borrows for the duration of a query and returns immediately afterward, ready for the next request to reuse.

Reading Pool Metrics

```
pool size: 20
active:    20
idle:      0
waiting:   14
```

Active connections are currently checked out and in use; **idle** ones are open but available; **waiting** is the number of requests currently blocked, unable to get a connection at all — exactly this course's own opening example from Chapter 1. All 20 connections in use, none idle, 14 more requests queued behind them: the pool is genuinely exhausted. The interesting question is why.

Two Genuinely Different Root Causes

CAUSE	WHAT'S ACTUALLY HAPPENING
Genuine capacity shortage	Traffic has grown, and the pool size was never increased to match — under peak load, demand simply exceeds the pool's supply, spread evenly across many ordinary, short-lived connections

CAUSE	WHAT'S ACTUALLY HAPPENING
A small number of connections held too long	The pool isn't undersized at all — a small number of misbehaving requests (a slow query, a forgotten commit/rollback, a leaked connection) are holding onto connections far longer than normal, starving everyone else

This is genuinely the same shape of question **System Monitoring & Performance Diagnosis**'s own Chapter 9 asked about a rising resource trend — is this real growth, or a leak — just applied to a connection pool instead of memory.

Telling the Two Apart: Connection Hold Time

The active/idle/waiting counts alone can't distinguish the two — both look identical from that view alone. What actually separates them is how long connections are being held. If most connections check out and return quickly, and the pool is simply saturated by genuinely high concurrent traffic, that's a capacity problem. If a small handful of connections are held for seconds or minutes while everything else churns normally in milliseconds, that's a small number of culprits starving the whole pool — not a sizing problem at all.

Finding the Specific Culprit

Most databases expose exactly this — which queries are currently running, and for how long:

```
-- PostgreSQL
SELECT pid, now() - query_start AS duration, state, query
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY duration DESC
LIMIT 5;

  pid  | duration |  state  |                  query
-------+----------+---------+-----------------------------------------
18832 | 00:04:12 | active  | SELECT * FROM order_items WHERE ...
  9021 | 00:00:03 | active  | SELECT * FROM users WHERE id = $1
  9033 | 00:00:02 | active  | UPDATE inventory SET qty = qty - 1 ...
```

One query has been running for over four minutes — dramatically longer than everything else on the list, which is finishing in a couple of seconds. This isn't a capacity problem; it's one specific query holding a connection hostage while everything else is genuinely fine.

> **"JUST INCREASE THE POOL SIZE" CAN MAKE A LEAK WORSE**
>
> Bumping the pool size is a tempting quick fix, and it genuinely helps if the real cause is capacity. If the real cause is a leak or a slow query instead, a bigger pool doesn't fix anything — the same misbehaving query eventually exhausts the larger pool too, just after a slightly longer delay. Worse, a larger pool means more simultaneous connections the

database itself has to serve, adding real load to a server that might already be struggling with the slow query in the first place.

MYSQL'S OWN EQUIVALENT

`SHOW PROCESSLIST;` (or `SELECT * FROM information_schema.PROCESSLIST;` for a queryable form) shows the same information on MySQL — each connection's current query and how long it's been running, sorted the same way.

Working Example: Fully Resolving the Checkout Ticket

Chapter 1's ticket, finally closed out: `pg_stat_activity` confirms exactly one query — a full scan of `order_items` with no supporting index on the column it filters by — has been running for over four minutes, while every other query on the system finishes normally. That single stuck query is holding a connection the whole time, and under peak checkout traffic, enough concurrent requests hit the same slow code path to exhaust the pool entirely, producing the "some requests fail, not all" pattern Chapter 2 identified. The actual fix is adding the missing index (or rewriting the query) — not increasing the pool size, which would only mask the same problem a little longer while adding more load to an already-struggling database.

Hands-On Exercises

EXERCISE 1

Explain why "active: 20, waiting: 14" alone can't tell you whether a connection pool is genuinely undersized or being starved by a small number of misbehaving connections.

View solution

EXERCISE 2

Explain why simply increasing the pool size can make things worse if the real cause is a leak or a slow query, rather than genuine capacity shortage.

View solution

EXERCISE 3

In this chapter's worked example, explain what the `pg_stat_activity` query actually revealed, and why that specifically confirms this was a "held too long" problem, not a capacity problem.

View solution

CHAPTER 3 QUICK REFERENCE

- A connection pool exists because opening a new database connection per request is too expensive to do every time
- Two genuinely different causes of exhaustion: **genuine capacity shortage** vs. **a small number of connections held too long**
- The active/idle/waiting counts alone can't distinguish them — **connection hold time** is what actually separates the two
- `pg_stat_activity` (PostgreSQL) / `SHOW PROCESSLIST` (MySQL) find the specific stuck query directly
- **Increasing pool size doesn't fix a leak** — it delays the same symptom and adds real load to the database
- **Next chapter:** Caching Layer Problems: Stale Data & Cache Stampede

CHAPTER 4 OF 10

Caching Layer Problems: Stale Data & Cache Stampede

Web & Application Troubleshooting

Chapter 4 · Caching Layer Problems: Stale Data & Cache Stampede

A cache trades correctness for speed — it serves a stored copy of data instead of recomputing or refetching it, on the bet that the copy is still good enough to use. That bet is usually right. This chapter is about the two genuinely different ways it goes wrong: a cache quietly serving something that's no longer true, and a cache emptying itself all at once and taking the backend down with it for a few seconds.

Cache Invalidation: The Genuinely Hard Problem

Two common strategies, each with a real, honest tradeoff:

STRATEGY	THE TRADEOFF
TTL-based expiry	Simple and reliable, but a fixed staleness window is built in by design — too short and the cache barely helps, too long and real staleness is guaranteed for that whole window
Explicit invalidation on write	No inherent staleness window in theory, but fails completely the moment any code path updates the underlying data without also triggering the invalidation — a direct admin script, a different service writing the same table, or a bug that only invalidates on the "happy path"

Neither approach is simply "better" — TTL-based caching accepts a known, bounded staleness window up front; explicit invalidation aims for zero staleness but is only as reliable as every single write path that's supposed to trigger it, which in a real system with multiple services and admin tooling is genuinely easy to miss.

Reading a Stale-Data Complaint

Before assuming a "wrong data" complaint is a cache bug, confirm it directly — compare what the cache is actually returning against what the real source of truth currently holds for the same key:

```
$ redis-cli GET product:4821:price
"29.99"

$ psql -c "SELECT price FROM products WHERE id = 4821;"
 price
-------
 24.99
```

A genuine mismatch confirms a stale cache. If the two actually agree, the cache isn't the problem at all — the "wrong data" complaint is something else entirely (a display bug, a genuine data error, or user confusion), and continuing to chase caching as the cause would be a wasted detour.

Cache Stampede: When Many Requests Miss at Once

A stampede (also called a thundering herd) happens when a popular cache key expires, and many concurrent requests all miss the cache at the exact same instant — all of them then hit the backend simultaneously to regenerate the same value, producing a sudden, sharp spike in backend load precisely at the moment of expiry.

A STAMPEDE IS EASY FOR A DASHBOARD AVERAGE TO HIDE

A stampede is typically brief — seconds, not minutes. Per **System Monitoring & Performance Diagnosis**'s own Chapter 7, a brief, severe spike can vanish almost entirely into a 5-minute averaged dashboard graph, exactly the kind of event that looks unremarkable at a glance but is genuinely causing real, user-visible pain in the moment it happens.

Real mitigations worth knowing: **jittered TTLs** (randomizing expiry slightly per key, so not every instance expires at the exact same moment), a **single-flight** pattern (only one request regenerates the value while everyone else waits for that result instead of duplicating the work), and **stale-while-revalidate** (serving the slightly-stale value immediately while quietly regenerating it in the background).

Recognizing a Stampede From the Outside

A stampede caused by a fixed TTL has a genuinely distinctive signature: it recurs at a suspiciously regular, clock-aligned interval — every hour, on the hour, if the TTL is exactly 3600 seconds. That periodicity itself is real diagnostic evidence, not a coincidence worth ignoring.

Working Example: The Hourly Database Spike

A fresh ticket: every hour, right on the hour, the database briefly spikes to near-100% CPU for about 10 seconds, then returns to normal — otherwise, everything is healthy. The clock-aligned periodicity is the immediate tell. Checking a suspected key's own remaining time-to-live confirms it:

```
$ redis-cli TTL homepage_featured_products
(integer) 3542
```

A TTL close to a full hour, on a genuinely popular key — the homepage's own featured-products list, requested by nearly every visitor. Every instance of the application shares the same cache key, so every one of them experiences the exact same expiry moment simultaneously, and every request that arrives in that brief window misses the cache and hits the database at once. Adding a small amount of random jitter to the TTL (so different instances' copies expire at slightly different times) spreads that load out instead of concentrating it into one sharp spike every hour.

Hands-On Exercises

EXERCISE 1

Explain the real tradeoff between TTL-based expiry and explicit invalidation-on-write, using this chapter's own description of when each one fails.

View solution

EXERCISE 2

Explain why comparing the cached value directly against the database is the right first step for a "wrong data" complaint, rather than assuming it's a cache bug.

View solution

EXERCISE 3

Explain why the hourly-on-the-hour timing of the database spike in this chapter's worked example was itself important diagnostic evidence, not just a detail.

View solution

CHAPTER 4 QUICK REFERENCE

- **TTL-based expiry** accepts a known staleness window; **explicit invalidation** aims for zero staleness but fails if any write path skips it
- Confirm a stale-data complaint by comparing the cache directly against the source of truth — don't assume
- A **cache stampede**: a popular key expires, many requests miss simultaneously, all hit the backend at once

- A stampede is often brief enough to **hide inside an averaged dashboard graph** — the same gotcha as `perfdiag1` Chapter 7
- **Clock-aligned periodicity** (every hour, on the hour) is itself real evidence of a fixed-TTL stampede
- Mitigations: **jittered TTLs**, **single-flight regeneration**, **stale-while-revalidate**
- **Next chapter:** Session & State Issues in Load-Balanced Environments

CHAPTER 5 OF 10

Session & State Issues in Load-Balanced Environments

Web & Application Troubleshooting

Chapter 5 · Session & State Issues in Load-Balanced Environments

Spread traffic across several application servers behind a load balancer, and a real question appears that a single-server setup never has to ask: where does a logged-in user's session actually live? If each server only keeps sessions in its own local memory, a session exists on exactly one server — whichever one happened to handle the login. This chapter is about what happens when a later request lands somewhere else.

Sticky Sessions (Session Affinity)

One fix: configure the load balancer to route every request from a given user consistently to the same backend server, usually via a cookie identifying which server they were first assigned to. It solves the local-memory problem without needing any shared infrastructure — but it has a genuine, honest failure mode of its own.

STICKY SESSIONS CONCENTRATE RISK ON ONE SERVER

Every user stuck to a given server has all of their session state living only there. If that one server goes down, gets restarted, or is removed during a deployment or a scaling event, every session assigned to it is lost simultaneously — not gradually, not one user at a time, but all at once, for whichever users happened to be routed there.

Shared Session Stores

The more resilient alternative: store session data in a shared, external store — commonly Redis — that every application server can read from and write to, regardless of which one a given request lands on. Any server can now correctly serve any user, removing the single-server session-loss risk sticky sessions carry.

NOT A STRICTLY BETTER, NO-DOWNSIDES FIX

A shared session store introduces its own dependency and its own failure mode: if the shared store itself becomes unavailable, every session everywhere fails at once — a different, and arguably more severe, single point of failure than losing sessions on one backend server. It also adds a real network round-trip to every session read and write, where a local-memory lookup had none. Neither approach is simply "better" — each accepts a different kind of risk.

Recognizing the Symptom Pattern

"Logged out randomly," "my cart emptied," or "had to log in again mid-session" — reported by some users but not others, especially clustering around a recent deployment or scaling event — is the classic tell. Worth being precise about how this differs from Chapters 3 and 4's own "some requests fail" patterns: there, the "some" correlated with timing and resource contention. Here, the "some" correlates with *which server a user happens to be routed to* — a genuinely different kind of "intermittent," worth telling apart before assuming the same category of cause applies again.

A Practical Diagnostic Check

Two things worth checking directly: whether the timing lines up with a deployment or scaling event (the same "does it correlate with a known event" question from Chapter 1, applied here), and whether the sticky-session cookie is actually present and being honored.

```
$ curl -v https://app.example.com/login
...
< Set-Cookie: AWSALB=xyz123...; Path=/
< Set-Cookie: session_id=abc987...; Path=/; HttpOnly
```

A missing or unexpectedly absent affinity cookie — stripped by a misconfigured proxy or CDN somewhere in the path, or a load balancer configuration change — would explain session loss just as directly as a server restart would.

Working Example: The Scaling Event That Reshuffled Sessions

A fresh ticket: since yesterday's autoscaling event added two new server instances, roughly 15% of users report being logged out mid-session, seemingly at random. Investigating confirms sticky sessions are configured — but the application was never using a shared session store, only local, per-server memory. Users who happened to remain assigned to the original server instances continued fine. But many load balancers redistribute existing connections across the new, larger pool of backends when the pool itself changes — exactly what happened here, reshuffling a portion of users onto instances that had never seen their session before. Their local-only session simply didn't exist on the new server, and they were silently logged out.

The fix isn't reverting the scaling event — it's migrating to a shared, Redis-backed session store, so a scaling event (or any future one) no longer has the power to erase sessions just by changing which server happens to answer a given request.

Hands-On Exercises

EXERCISE 1

Explain why sticky sessions solve the local-session problem without needing shared infrastructure, and what genuine failure mode they introduce in exchange.

View solution

EXERCISE 2

Explain why this chapter says a shared Redis-backed session store isn't simply a strictly better fix than sticky sessions, even though it solves the single-server session-loss problem.

View solution

EXERCISE 3

In this chapter's worked example, explain why the autoscaling event specifically caused session loss for some users, even though sticky sessions were correctly configured.

View solution

CHAPTER 5 QUICK REFERENCE

- Local, per-server sessions only work if every request from a user lands on the same server — **sticky sessions** enforce that via a routing cookie
- Sticky sessions concentrate risk: **losing one server loses every session assigned to it, all at once**
- A **shared session store** (e.g. Redis) removes that risk but adds its own dependency and a network round-trip per session access
- "Logged out randomly" correlating with a **deployment or scaling event** is the classic session-affinity symptom — a genuinely different kind of "intermittent" than Chapters 3–4's own resource-contention patterns
- Check for the sticky-session cookie directly (`curl -v`) — a missing one explains session loss just as directly as a server going down
- **Next chapter:** Slow Query & N+1 Diagnosis

CHAPTER 6 OF 10

Slow Query & N+1 Diagnosis

Web & Application Troubleshooting

Chapter 6 · Slow Query & N+1 Diagnosis

Logging & Log Analysis's own capstone found an N+1 pattern by comparing query counts before and after a deployment. This chapter goes deeper: how to recognize an N+1 pattern directly, how to tell it apart from a single genuinely slow query (a different problem with a completely different fix), and how to read just enough of an `EXPLAIN` plan to spot the single most common cause of a slow query without needing a full database-tuning background.

What N+1 Actually Is

Instead of one query that fetches everything needed at once — a `JOIN`, or a single query with an `IN` clause — the code runs one query to get a list of N items, then loops through them and runs N additional individual queries, one per item, to fetch each one's related data. N+1 total queries where 1 or 2 would have done the job. A very common ORM pitfall, usually from a relationship being lazily loaded inside a loop without anyone noticing.

Recognizing N+1 vs. a Single Slow Query

The fix for each is completely different, so telling them apart matters — and the distinguishing evidence is straightforward: count the queries a single request triggers, and look at each one's own duration.

```
# A single slow query
[query] SELECT * FROM order_items WHERE order_id = 9001;  -- 812ms
Total queries: 1   Total time: 812ms

# N+1
[query] SELECT * FROM orders WHERE user_id = 4821;         -- 3ms
[query] SELECT * FROM order_items WHERE order_id = 9001;   -- 15ms
[query] SELECT * FROM order_items WHERE order_id = 9002;   -- 14ms
[query] SELECT * FROM order_items WHERE order_id = 9003;   -- 16ms
... 47 more nearly-identical lines ...
Total queries: 51   Total time: 780ms
```

One long query and dozens of individually-fast ones can add up to almost the same total time — but the fix couldn't be more different: optimizing or indexing one query, versus restructuring the code so it stops looping

and issuing a query per item.

Reading EXPLAIN for a Genuinely Slow Single Query

A full database-tuning background isn't needed to catch the single most common cause of a slow query — a full table scan on a large table:

```
EXPLAIN ANALYZE SELECT * FROM order_items WHERE order_id = 9001;

Seq Scan on order_items  (cost=0.00..48213.00 rows=1 width=64) (actual time=810.442..810.443 rows=1 loops=1)
  Filter: (order_id = 9001)
  Rows Removed by Filter: 2499998
Planning Time: 0.112 ms
Execution Time: 810.501 ms
```

`Seq Scan` means the database read through the entire table — 2.5 million rows, to find one matching row — because no index exists on the column being filtered. Contrast with a healthy result on an indexed column:

```
Index Scan using idx_order_items_order_id on order_items  (cost=0.42..8.44 rows=1 width=64) (actual time=0.015..0
```

Same query shape, over 50,000 times faster — the entire difference is having (or not having) the right index. Recognizing `Seq Scan` on a large table in an `EXPLAIN` output is, by itself, one of the highest-value diagnostic skills in this chapter.

The N+1 Fix, at a Practical Level

Two standard fix shapes worth being able to describe clearly, even if the actual code change belongs to a developer: **batch loading** (one additional query using an `IN` clause to fetch every needed related record at once, instead of one query per item) or a **JOIN** that retrieves everything in a single query from the start.

> **"IT'S FAST IN TESTING" IS EXACTLY WHAT YOU'D EXPECT**
>
> N+1 patterns scale with N — with a small test dataset (5 items = 6 queries total), the extra overhead is barely noticeable. In production, with realistic data volumes (500 items = 501 queries), the exact same code becomes genuinely slow. This is precisely why these bugs so often pass testing unnoticed and only surface under real load — the same "only happens under load" signal Chapter 1 opened with.

Working Example: The Slow Order History Page

A fresh ticket: the order history page takes 8+ seconds to load for users with a long order history, but loads instantly for new users with few orders. That user-count correlation is itself a strong early hint — a single slow query wouldn't care how many orders a particular user has; an N+1 pattern would scale exactly this way. Query logging confirms it directly: one query fetching the order list, followed by one additional query per order to fetch that order's line items — 51 total queries for a user with 50 orders, each individually fast, together adding up to the full 8 seconds plus per-query round-trip overhead. Replacing the per-order loop with a single batched query (an `IN` clause covering every order ID from the first query) cuts the page load from 8 seconds to under 200ms — the same total data, retrieved in two queries instead of fifty-one.

Hands-On Exercises

EXERCISE 1

Explain why a single 812ms query and 51 queries totaling 780ms can both make a page feel equally slow, but need completely different fixes.

View solution

EXERCISE 2

Explain what a `Seq Scan` in an `EXPLAIN` output actually means, and why it's a genuine red flag on a large table specifically.

View solution

EXERCISE 3

Explain why the order-history page's slowness scaling with a user's own order count was itself a meaningful clue, before any query log was even checked.

View solution

CHAPTER 6 QUICK REFERENCE

- **N+1**: one query for a list, then N more — one per item — instead of a single batched fetch
- Distinguish a single slow query from N+1 by **counting queries per request**, not just total time
- `Seq Scan` on a large table in `EXPLAIN ANALYZE` = the single most common, easiest-to-spot cause of a slow single query
- Fixes: **batch loading (`IN` clause)** or a **JOIN** — one query instead of many

- N+1 bugs routinely pass testing because they scale with N — small test data hides them; production-scale data reveals them
- **Next chapter:** Deployment-Related Symptoms: Version Skew & Migration Failures

CHAPTER 7 OF 10

Deployment-Related Symptoms: Version Skew & Migration Failures

Web & Application Troubleshooting

Chapter 7 · Deployment-Related Symptoms: Version Skew & Migration Failures

Chapter 2 named "100% of requests fail, suddenly" as the pattern most likely to point at a deployment. This chapter gives that heuristic its fullest treatment — what actually happens during the window a rolling deployment is in progress, why a database migration can fail independently of the code deploy that depends on it, and a concrete technique for confirming a deployment is the cause rather than just suspecting it.

Version Skew: Old and New Code, Running at the Same Time

A rolling or blue-green deployment inevitably passes through a window where old and new code run simultaneously, across different server instances. If the new version changes an API contract in a way that isn't backward- or forward-compatible — a renamed field, a newly required field, a changed response shape — requests can fail unpredictably depending on which specific instance happens to handle them during that window. This applies to service-to-service communication too, not just client requests: a frontend expecting the new shape can just as easily hit an old backend instance, or the reverse.

This directly explains a genuinely confusing symptom: "it works sometimes and fails other times, right after a deploy, with no code changes since" — because which instance answers a given request is effectively random during the rollout, and each instance is running one version or the other, never a blend.

Recognizing Version Skew From the Outside

Directly checking each instance's own reported version settles this immediately:

```
$ for host in app-1 app-2 app-3 app-4 app-5 app-6 app-7 app-8 app-9 app-10; do
    echo -n "$host: "; curl -s https://$host.internal/version
done

app-1: {"version":"2.14.0"}
app-2: {"version":"2.14.0"}
app-3: {"version":"2.15.0"}
app-4: {"version":"2.14.0"}
app-5: {"version":"2.14.0"}
app-6: {"version":"2.15.0"}
app-7: {"version":"2.14.0"}
app-8: {"version":"2.14.0"}
app-9: {"version":"2.14.0"}
app-10: {"version":"2.15.0"}
```

Three of ten instances already on the new version, seven still on the old — a rollout genuinely mid-flight. In a container-orchestrated environment, checking each running pod's own image tag serves the same purpose.

Database Migration Failures

A schema migration can fail partway through, or succeed on the database while the application code deployed alongside it doesn't actually match what the migration has finished doing yet. A genuinely common real mistake: deploying code that expects a new field *before* the migration that backfills it has actually completed everywhere — or the reverse, old code still running against a schema that's already been changed underneath it.

THE EXPAND-CONTRACT PATTERN EXISTS SPECIFICALLY TO AVOID THIS

The safe sequencing most teams aim for: **expand** first (add the new column/field, deploy code that can handle both the old and new shapes at once), let that settle completely, *then* **contract** (remove the old field in a later, separate release). The whole point is avoiding any single moment where a schema change and a code deploy have to land in perfect lockstep. A sudden burst of "field not found" or "column does not exist" errors immediately following a deploy is the classic symptom of that discipline having been skipped.

A Genuine Technique: Bisecting by Time

When an error rate shows a sudden, sharp step-change at one specific timestamp — not a gradual climb — checking exactly what deployed or migrated at that precise moment is almost always faster than examining application code in the abstract. The deployment history and migration log are themselves evidence, in exactly the same spirit as this subject's own "check first, don't guess" theme.

Working Example: The 30% "Field Not Found" Ticket

A fresh ticket: right after this afternoon's deploy, about 30% of requests to `/profile` started returning "field not found" errors; the rest work fine. Checking each instance's own version confirms exactly the mid-rollout pattern above — 3 of 10 instances on the new version. 3 out of 10 is 30%, matching the failure rate almost exactly.

```
[ERROR] KeyError: 'display_name' -- user_id=88213, field missing
```

The new version's code expects a `display_name` field that a migration was meant to backfill onto every existing user record — but the backfill is still running and only partially complete. Requests landing on new-version instances fail specifically for the subset of accounts the backfill hasn't reached yet. The fix: pause the rollout, let the backfill finish fully, confirm the field is populated everywhere, then resume — the expand-contract discipline this chapter names, applied correctly this time by fixing the sequencing rather than rolling forward blind.

Hands-On Exercises

EXERCISE 1

Explain why "it works sometimes and fails other times, right after a deploy, with no code changes since" is a classic version-skew symptom, and what makes the failures effectively random from a user's point of view.

View solution

EXERCISE 2

Explain what the expand-contract pattern is for, and specifically what kind of failure it's designed to prevent.

View solution

EXERCISE 3

In this chapter's worked example, explain why the 30% failure rate matching "3 of 10 instances on the new version" was significant, and what the actual root cause turned out to be.

View solution

CHAPTER 7 QUICK REFERENCE

- A rolling/blue-green deployment always passes through a window where **old and new code run simultaneously** — a genuine, real source of unpredictable, instance-dependent failures
- Check each instance's own `/version` endpoint (or pod image tag) to confirm a mid-rollout mix directly
- The **expand-contract pattern**: add first, deploy code that handles both shapes, remove later — avoids needing a schema change and a code deploy to land in perfect lockstep
- A sudden error-rate **step-change at one exact timestamp** — check deployment/migration history first, before diving into code
- **Next chapter:** Health Checks, Readiness Probes & Graceful Shutdown

CHAPTER 8 OF 10

Health Checks, Readiness Probes & Graceful Shutdown

Web & Application Troubleshooting

Chapter 8 · Health Checks, Readiness Probes & Graceful Shutdown

Chapter 7 covered what goes wrong *during* a rollout. This chapter covers the machinery that's supposed to make a rollout — or any routine restart — safe in the first place: liveness and readiness checks, and what happens when an instance is shut down without giving it a chance to finish what it was doing.

Liveness vs. Readiness: A Real, Important Distinction

Two genuinely different questions, with genuinely different consequences when the answer is no:

CHECK	QUESTION IT ANSWERS	WHAT HAPPENS ON FAILURE
Liveness	Is this process alive at all?	The orchestrator restarts the container
Readiness	Can this instance currently serve traffic correctly?	The instance is pulled from the load balancer's rotation — *not* restarted

USING THE SAME CHECK FOR BOTH IS A REAL, COMMON MISTAKE

A single shared check that's too broad — for example, one that verifies a downstream third-party dependency is reachable — can turn an unrelated outage into needless restarts. A temporary blip in a payment provider's API should mean "temporarily stop sending this instance new requests" (a readiness concern), not "kill and restart this perfectly healthy process" (a liveness consequence) — but with a shared check, the orchestrator can't tell the two apart.

```
# Separate, correctly-scoped checks
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  periodSeconds: 10
  failureThreshold: 3

readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  periodSeconds: 5
  failureThreshold: 2
```

What a Readiness Probe Should (and Shouldn't) Check

Reasonable readiness checks: can this instance reach its own database connection pool, is its cache connection alive, has startup initialization finished. Unreasonable: checking something unrelated to whether *this specific instance* can serve traffic — a third-party API's own health, for instance, can take an entire fleet out of rotation over someone else's outage, even though the application itself is otherwise perfectly capable of handling most requests.

The Symptom of a Readiness Probe Gone Wrong

A misconfigured readiness check produces a distinctive pattern: instances repeatedly pulled from rotation and put back — "flapping" — visible as capacity periodically dropping even though nothing actually crashed. Checking the orchestrator's own event history (or a load balancer's own health-check log) directly shows readiness failures, rather than requiring you to assume instances are genuinely down.

Graceful Shutdown: The Other Half of the Lifecycle

When an instance is being replaced — during a deployment, a scale-down, or a routine restart — simply killing the process immediately can cut off requests that were still mid-processing. A well-behaved shutdown sequence does three things in order: mark the instance not-ready (so the load balancer stops sending new requests), wait for in-flight requests to actually finish (a **drain period**), and only then terminate.

```
  lifecycle:
    preStop:
      exec:
        command: ["sh", "-c", "sleep 15"]
  terminationGracePeriodSeconds: 30
```

Skipping the drain step produces its own recognizable symptom: a small, brief burst of connection-reset or aborted-request errors, correlating precisely with deployment or scale-down events.

DISTINCT FROM CHAPTER 7'S OWN DEPLOY-CORRELATED SYMPTOM

Both are deploy-correlated, but they're not the same problem: version skew (Chapter 7) produces wrong data or contract mismatches, because two genuinely different code versions are both answering requests. A missing drain period produces dropped or reset connections specifically at the exact moment of termination, because a request was still in flight when its server was killed out from under it. Telling them apart matters — the fixes are completely different.

A Concrete Symptom-to-Cause Table

SYMPTOM	LIKELY CAUSE
Capacity flaps up and down, no actual crashes	Readiness check misconfigured — too strict, or checking the wrong thing
Brief burst of connection resets, exactly at deploy/scale-down moments	No graceful shutdown/drain period configured
Unnecessary restarts correlating with an unrelated dependency's own outage	Liveness check too broad — checking something readiness should own instead

Working Example: The 20-Request Deploy Blip

A fresh ticket: every deployment causes a brief spike of roughly 20 failed "connection reset" requests, lasting just a few seconds — the deployment itself completes successfully, and the new version works fine immediately afterward. Checking the deployment process confirms old instances are terminated the instant the new version becomes ready, with no drain period configured at all. Requests still in flight on an old instance at that exact moment get abruptly cut off mid-response. Adding a `preStop` hook that pauses before actual termination — giving the load balancer time to stop routing new traffic and letting existing requests finish — resolves the blip entirely without changing anything about the deployment's own speed or correctness.

Hands-On Exercises

EXERCISE 1

Explain why using the same check for both liveness and readiness is a real mistake, using this chapter's own payment-provider example.

View solution

EXERCISE 2

Explain the difference between the symptom caused by version skew (Chapter 7) and the symptom caused by a missing drain period, and why they need different fixes.

View solution

EXERCISE 3

In this chapter's worked example, explain exactly why in-flight requests were being cut off, and how a `preStop` hook fixes it without changing the deployment's own logic.

View solution

CHAPTER 8 QUICK REFERENCE

- **Liveness** = is the process alive (failure → restart); **readiness** = can it serve traffic right now (failure → pulled from rotation, not restarted)
- A shared check for both is a real mistake — an unrelated dependency blip can trigger needless restarts
- Readiness should check *this instance's own* ability to serve — not a third-party dependency's own health
- A flapping capacity count with no real crashes = a misconfigured readiness check
- **Graceful shutdown**: mark not-ready, drain in-flight requests, then terminate — skipping it causes connection resets exactly at deploy/scale-down moments
- Distinct from version skew: wrong data/contract mismatch (Ch7) vs. dropped connections at the moment of termination (this chapter)
- **Next chapter:** Rate Limiting & Throttling Symptoms

CHAPTER 9 OF 10

Rate Limiting & Throttling Symptoms

Web & Application Troubleshooting

Chapter 9 · Rate Limiting & Throttling Symptoms

A 429 looks alarming in a dashboard full of otherwise-green metrics, but it's a genuinely different kind of signal from everything else this course has covered so far — a rate limit rejecting a request isn't a sign anything is broken. This chapter closes out the course's content chapters by reading that signal correctly, and by recognizing one more version of a pattern that's now appeared several times: aggregate exhaustion caused either by genuine broad demand, or by one specific culprit.

"Throttled" Is Not "Down"

A **429 Too Many Requests** is the server explicitly saying: I'm healthy, I received your request, and I'm deliberately declining to process it right now. That's a fundamentally different situation from a 5xx (the server struggling) or a timeout (the network or server genuinely unresponsive) — recognizing a 429 immediately for what it is avoids wasting time chasing a "the service is down" theory when the service is, in fact, working exactly as designed.

Reading the Rate Limit Headers

```
$ curl -v https://api.example.com/orders
< HTTP/1.1 429 Too Many Requests
< X-RateLimit-Limit: 1000
< X-RateLimit-Remaining: 0
< X-RateLimit-Reset: 1723190400
< Retry-After: 42
```

These headers turn a vague "we got throttled" into a precise statement: this client's limit is 1,000 requests per window, 0 remain, and it resets in 42 seconds. No guessing needed about when the client will be unblocked.

Client-Side vs. Server-Side: Who Actually Applied the Limit?

A rate limit might be the application's own internal per-user or per-key limit, protecting its own resources — or it might come from somewhere entirely outside the application's control, like a third-party API the app calls

being rate-limited, or a CDN/WAF applying its own limit before a request even reaches the application at all. This is genuinely the same ambiguity **Network Troubleshooting**'s own Chapter 9 raised about a 403 — the response code alone doesn't always say which layer actually produced it.

Two Genuinely Different Symptom Patterns

A pattern this course keeps returning to, in a new shape one more time: aggregate exhaustion caused either by genuine, broad demand exceeding a limit that's simply set too low (Chapter 3's own connection-pool "capacity" case, again) — or by one small number of culprits consuming a disproportionate share of a shared pool (Chapter 3's own "held too long" case, again, this time a client hammering a rate limit rather than a query holding a connection).

CAUSE	THE FIX
The limit is genuinely too low for legitimate usage	Adjust the limit itself
One misbehaving client is consuming a disproportionate share	Identify and fix (or block) that specific client — raising the limit for everyone else doesn't address the actual cause

Finding the Actual Culprit

Most rate limiters can report consumption broken down by client, API key, or IP — not just the aggregate. A single key consuming a wildly disproportionate share points directly at the second cause:

```
API Key              Requests (last hour)   % of total
partner_x_7a92        42,850                  87.3%
mobile_app_ios         3,120                   6.4%
mobile_app_android     2,890                   5.9%
... dozens more keys, each under 1% ...
```

RETRYING WITHOUT BACKOFF MAKES THROTTLING WORSE, NOT BETTER

A client that immediately retries the instant it receives a 429, without waiting, adds more load exactly when the server just asked it to slow down — which can turn a brief, legitimate throttle into a sustained, self-inflicted problem. A well-behaved client respects the `Retry-After` header directly (introduced back in Chapter 2) rather than guessing at its own retry timing.

Working Example: The Partner Integration's Retry Bug

A fresh ticket: since this morning, a significant fraction of API requests from mobile app users are returning 429s — even though overall traffic hasn't meaningfully grown. The per-key breakdown above tells the real story immediately: one specific API key, belonging to a single integration partner rather than typical mobile traffic, accounts for over 87% of requests against the shared limit. A recent bug in that partner's own retry logic started triggering far more often this morning, and — ignoring `Retry-After` entirely — it retries immediately on every 429, compounding the problem with every failed attempt.

This is genuinely not a capacity problem — the limit was perfectly adequate for real traffic before this partner's bug started firing — and it's not something this application's own team can fix directly, since the broken retry logic lives in someone else's system. The honest resolution here is external: reaching out to the partner to fix their retry behavior, and, in the meantime, applying a tighter limit specifically scoped to that one key so it stops degrading service for every other legitimate client sharing the same overall pool.

Hands-On Exercises

EXERCISE 1

Explain why this chapter treats a 429 as a fundamentally different signal from a 5xx or a timeout, and why confusing the two wastes troubleshooting time.

View solution

EXERCISE 2

Explain why a client retrying immediately after a 429, without respecting `Retry-After`, can turn a brief throttle into a sustained problem.

View solution

EXERCISE 3

In this chapter's worked example, explain why raising the shared rate limit for everyone would have been the wrong fix, and what the actual resolution was instead.

View solution

CHAPTER 9 QUICK REFERENCE

- A **429** means the server is healthy and deliberately declining the request — a different signal from a 5xx or a timeout entirely

- `X-RateLimit-Limit/Remaining/Reset` turn "we got throttled" into an exact, precise statement of when it clears
- A rate limit might come from the app itself, or from something upstream (a WAF, a third-party API) — the same "who actually produced this" ambiguity as a 403
- Aggregate exhaustion: **genuinely too-low limit** vs. **one misbehaving client** — the same shape this course has returned to since Chapter 3
- Check **per-key/IP consumption**, not just the aggregate, to tell them apart
- **Retrying without backoff makes throttling worse** — respect `Retry-After`
- Not every fix is internal — sometimes the real resolution is external, organizational, not a code change
- **Next chapter:** Capstone: Triaging Three Real Application Tickets

CHAPTER 10 OF 10

Capstone: Triaging Three Real Application Tickets

Web & Application Troubleshooting

Chapter 10 · Capstone — Triaging Three Real Application Tickets

Nine chapters built the pieces — knowing this course's own boundary against its three siblings, reading error bodies, connection pools, caching, sessions, slow queries, deployments, health checks, and rate limits. This capstone applies all of it to three fresh tickets, worked more briskly than earlier chapters' own dedicated walkthroughs, since the underlying discipline should already feel familiar by now.

Ticket 1: "The admin dashboard is unusably slow for our biggest customers"

The admin dashboard's user list page loads instantly for small organizations, but takes 12+ seconds for organizations with hundreds of users.

APPLYING CHAPTER 1'S SCOPING INSTINCT

Load time scales with a specific count (organization size), not with overall traffic or time of day — exactly the signature Chapter 6 taught to recognize before even opening a query log.

APPLYING CHAPTER 6

Query logging confirms it directly: one query for the organization's user list, followed by one additional query per user to fetch that user's role and permission set — 340 total queries for an organization with 339 users. Replacing the per-user loop with a single batched query fetching all roles at once cuts the page load from 12 seconds to well under a second.

Ticket 2: "Since this morning's deploy, we're intermittently getting connection pool exhaustion"

Roughly a third of requests to a specific endpoint have started failing with pool timeouts since this morning's release; the rest work fine.

APPLYING CHAPTERS 1 AND 2

Sudden onset, tied to today's deploy, some requests failing rather than all — the error body confirms a familiar shape: `{"error": "pool_timeout", "request_id": "..."}`, with a correlation ID ready for deeper investigation.

RULING OUT CHAPTER 9 FIRST

An early guess — a partner integration retrying aggressively and exhausting shared resources — is checked directly via per-key rate-limit consumption. Every key shows normal, unremarkable usage. Ruled out; the real cause lies elsewhere.

APPLYING CHAPTER 7

Checking each instance's own version confirms the failures cluster specifically on instances already running this morning's release — a genuine version-skew signature, narrowing the search to what actually changed in the new code.

APPLYING CHAPTER 3, ONE LEVEL DEEPER

```
SELECT pid, now() - state_change AS duration, state
FROM pg_stat_activity
WHERE state = 'idle in transaction'
ORDER BY duration DESC;

  pid   | duration |        state
-------+----------+---------------------
18921 | 00:52:10 | idle in transaction
19042 | 00:41:03 | idle in transaction
```

Not simply a slow query this time — connections stuck in PostgreSQL's own `idle in transaction` state, meaning a transaction was opened and never committed or rolled back. This morning's new code path opens a transaction, but under one specific error condition throws before ever reaching the commit — silently leaking the connection forever, on every instance running the new version. The fix: wrap the transaction in a proper try/finally so it's always closed regardless of how the request ends, shipped as an immediate hotfix.

Ticket 3: "Since today's scale-up, users are getting logged out, and our real capacity seems lower than it should be"

Two complaints arriving together right after scaling up for a traffic event: random session loss, and capacity that doesn't match the number of instances actually running.

RULING OUT CHAPTER 4 FIRST

An early guess — new instances stampeding a cold cache — is checked directly: TTLs are properly jittered, and there's no clock-aligned periodicity in the load pattern. Ruled out.

APPLYING CHAPTER 5

The session-loss complaint matches this course's own established pattern exactly: the load balancer redistributed sticky-session assignments when the instance pool changed size, and the application still relies on local, per-server session storage rather than a shared store.

APPLYING CHAPTER 8

Separately, the orchestrator's own event log shows the new instances repeatedly cycling in and out of ready state — flapping, with no actual crashes. Their readiness check includes a call to a third-party shipping-rates API that's been running unusually slowly today, and a slow (not down) dependency is enough to fail the check. The application itself is otherwise perfectly capable of serving most requests; readiness was checking something it shouldn't have owned.

Two genuinely separate fixes for two genuinely separate causes, both surfaced by the same scaling event: migrating to a shared, Redis-backed session store, and narrowing the readiness check to only cover this instance's own true dependencies.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Scoping instincts — what a symptom's own shape suggests before checking anything (all three tickets)	Chapter 1
Reading the structured error body and correlation ID (Ticket 2)	Chapter 2
`pg_stat_activity`, extended to "idle in transaction" (Ticket 2)	Chapter 3

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Ruling out a cache stampede via TTL/jitter check (Ticket 3) — not the primary finding, but the same technique applies directly to a genuine stampede ticket	Chapter 4
Sticky-session reshuffling on a scaling event (Ticket 3)	Chapter 5
N+1 recognized via load time scaling with a count, confirmed by query logging (Ticket 1)	Chapter 6
Checking each instance's own version to confirm version-skew (Ticket 2)	Chapter 7
Reading the orchestrator's own event log for readiness flapping (Ticket 3)	Chapter 8
Ruling out rate limiting via per-key consumption (Ticket 2) — not the primary finding, but the same technique applies directly to a genuine throttling ticket	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No deep, ORM-specific syntax walkthroughs or commercial APM tool tutorials — the underlying patterns transfer, but specific tool interfaces vary too much to cover here
- No service mesh or sidecar-specific troubleshooting (Istio, Linkerd, and similar) — a real, separate discipline on top of what's covered here
- No distributed tracing systems in depth (Jaeger, Zipkin) — correlation IDs get you far; a full tracing setup is a bigger, separate topic
- No chaos engineering or fault-injection testing methodology — this course diagnoses real incidents, not designing tests that manufacture them
- No application-security-specific vulnerabilities — XSS, CSRF, SQL injection, and similar are covered in this site's own dedicated Security courses, not here

Each is a legitimate, separate topic — not silently assumed solved by what this course actually covers.

Hands-On Exercises

EXERCISE 1

Explain what made Ticket 1's symptom recognizable as N+1 before a single query log was checked, and how that matched Chapter 6's own reasoning.

View solution

EXERCISE 2

Explain what "idle in transaction" specifically means in Ticket 2, and why it's a genuinely different finding than the slow single query Chapter 3 originally taught with.

View solution

EXERCISE 3

Explain why Ticket 3 needed two separate fixes rather than one, even though both complaints started at the exact same moment.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Ticket 1: N+1 on the admin dashboard, recognized by load time scaling with organization size — confirmed and fixed with batch loading
- Ticket 2: a version-skew-scoped connection leak, found by ruling out rate limiting, confirming the affected instances' version, then reading "idle in transaction" state directly
- Ticket 3: one scaling event, two genuinely separate causes — sticky-session reshuffling and an overly broad readiness check — needing two separate fixes
- The recurring theme across all ten chapters: **know which of the four Technical Support courses actually owns a symptom, gather real evidence, and don't assume a fix is needed before confirming the cause**
- This closes *Web & Application Troubleshooting*, 10/10 chapters — the fourth complete course under the Technical Support subject, alongside *Logging & Log Analysis*, *Network Troubleshooting*, and *System Monitoring & Performance Diagnosis*