ROOT CAUSE CASEBOOK — FREE TEMPLATE

Root Cause Analysis Template

A fill-in deck for incident postmortems and RCA reviews

Problem statement · Timeline · Evidence log · 5 Whys · Contributing factors · Corrective actions · Follow-up verification

rootcausecasebook.com/root-cause-analysis-template/

START HERE

How to use this template

- **Restore service first** — the analysis starts when impact ends. Keep a scratch timeline during the incident: clock time, action, result.
- **Duplicate the blank slides (3–9)** — one set per incident. Delete the worked example (slides 10–12) or keep it as a reference.
- **Write impact before cause** — starting from the cause drops the misleading symptoms — and those symptoms are what the next on-call will see.
- **Keep observations separate from explanations** — a chart is an observation. The sentence that says what it means has to name the evidence behind it.
- **State three different things** — the trigger, the root-cause mechanism, and the contributing conditions. They are not interchangeable.
- **Keep the hypotheses that failed** — record what ruled them out, so the next investigation can skip the dead end.
- **Stay blameless** — describe the signals people had and the system conditions around their actions. Name failed controls, not character flaws.

Problem statement

- **Incident** — [system / service] — [date], [start–end, timezone]
- **What failed** — [the user-visible failure, in one or two sentences]
- **Who was affected** — [customers, regions, products, internal teams]
- **Impact** — [duration, scope, error rates or counts — only measurements you actually have]
- **Detected by** — [alert, dashboard, customer report — and what stayed quiet]
- **Severity** — [your severity scale and why this incident met it]

Write this slide so a reader who was not in the incident can trust it before reading the rest.
Where a number was never recorded, leave it out. An honest gap beats an invented figure.

TEMPLATE · 2 OF 7

Timeline

Time (UTC)	Event / action	Result / evidence

Rebuild the timeline from alerts, deploys, chat, and logs — not from memory. Mark a gap as a gap.

Evidence log

Source	What was checked	Finding
Telemetry / logs		
Change & rollout history		
Dependency health		
External observation		

Evidence is what separates an explanation from a guess. Log the checks that ruled things OUT as well as the ones that ruled things in.

5 Whys worksheet

Why 1? [answer — each answer must be supported by the evidence log]

Why 2? [answer — each answer must be supported by the evidence log]

Why 3? [answer — each answer must be supported by the evidence log]

Why 4? [answer — each answer must be supported by the evidence log]

Why 5? [answer — each answer must be supported by the evidence log]

Stop when you reach a mechanism you can change. “Human error” is never the last why — ask what made that action reasonable at the time.

Contributing factors

Change

What shipped, when, and how was it rolled out?

[your factor]

Code

Unhandled errors, limits, latent assumptions?

[your factor]

Configuration

Defaults, flags, permissions, generated files?

[your factor]

Infrastructure

Capacity, shared fate, single points of failure?

[your factor]

Process

Reviews, tests, guardrails that were missing?

[your factor]

Detection

Alerts or dashboards that stayed quiet?

[your factor]

The trigger started this occurrence. Contributing factors are the conditions that let a local fault become an outage — they are where most fixes live.

Corrective actions

Action	Owner	Due	Status

Each action must change a system, a limit, an alert, a test, or a review — and have a named owner.
“Be more careful” does not survive contact with the next incident.

Follow-up verification

- **Verification method** — [for each corrective action: how will you prove it works — a test, a game day, a re-run of the failing input?]
- **What would have caught it** — [the check, alert, or review that would have stopped this incident — and whether it now exists]
- **Review date** — [when this analysis gets re-read, and by whom]
- **Open questions** — [what is still unproven, stated plainly]
- **Shared with** — [teams who received this analysis and where the record lives]

Example: Cloudflare, November 18, 2025

Problem statement (filled in)

A routine ClickHouse permissions change caused a metadata query to return duplicate rows, doubling the Bot Management ML feature file. When the oversized file propagated to the global edge, the core proxy's Bot Management module hit its 200-feature preallocated limit and panicked, returning HTTP 5xx network-wide. The fluctuating failure pattern initially led engineers to suspect a hyper-scale DDoS attack.

Time (UTC)	Event	Result
11:05	ClickHouse permissions change (implicit → explicit access)	Generator query begins returning duplicate rows
11:20	Global HTTP 5xx spike — fluctuating, recovering, failing again	Pattern read as a possible attack
—	Crash signatures correlated across PoPs	Common Bot Management panic; not routing/BGP
—	Feature file inspected	Over 200 features vs ~60 normal — past the module limit
13:04	Workers KV patched to bypass the core proxy	Downstream impact mitigated
14:30	Bad file propagation halted; known-good file inserted; proxy restarted	Recovery under way
17:06	Full recovery	Incident closed

Example: the 5 Whys, filled in

Why did the network return HTTP 5xx? The core proxy panicked when the Bot Management module loaded its feature file.

Why did loading the file panic the proxy? The file held over 200 features, past the module's preallocated limit of 200, and the FL2 Rust code called Result::unwrap() on the error instead of handling it.

Why was the file oversized? The generator's system.columns query returned duplicate metadata rows, more than doubling the file.

Why did the query return duplicates? The 11:05 UTC permissions change made table access explicit; the query had no database-name filter, so it read rows from both the default and r0 databases.

Why did this reach production? The query silently depended on the old implicit-access semantics, and the consumer never validated the file's size — a latent assumption, not a reviewed decision.

Example: factors, lessons, next steps

- **Contributing factor — fault isolation** — a failure in a non-critical subsystem (Bot Management config loading) was able to panic the critical request path.
- **Contributing factor — latent assumption** — a query written without a database filter depended on implicit access semantics that a permissions change altered.
- **Contributing factor — rollout masking** — the 5-minute regeneration cadence plus a gradual rollout made the failure look intermittent and adversarial.
- **Lesson — validate consumers** — configuration-file consumers must be validated against oversized or malformed inputs: fail open or degrade gracefully instead of panicking.
- **Dead end worth keeping** — the attack hypothesis and the coincidental status-page outage consumed early attention — recorded in the case, so the next reader can skip it faster.
- **Read the full case** — rootcausecasebook.com — “Cloudflare outage on November 18, 2025” — with the complete timeline, evidence, and source postmortem.