2024-05-22T14:35:10Z [info] system_monitor: memory_usage=45.2%
2024-05-22T14:40:12Z [info] cursor_service: active_threads=128
2024-05-22T15:15:30Z [warn] kernel: high memory pressure detected
2024-05-22T15:15:30Z [warn] kernel: high memory pressure detected
2024-05-22T16:00:05Z [error] cursor: memory_allocation_failed
2024-05-22T17:12:45Z [info] system: oom-killer invoked, process='cursor'

STATUS: OFFLINE

My Server Died Again (This Time It Was Cursor)

2024-05-22T14:35:10Z [info] system_monitor: memory_usage=45.2%
2024-05-22T14:40:12Z [info] cursor_service: active_threads=128

A DevOps Post-Mortem: Tracking a 15-hour memory leak on a 32GB GCP devbox.

2024-05-22T14:40:12Z [info] system_monitor: memory_usage=45.2%
2024-05-22T15:15:30Z [warn] kernel: high memory pressure detected
2024-05-22T16:00:05Z [error] cursor: memory_allocation_failed
2024-05-22T17:12:45Z [alert] system: oom-killer invoked, process='cursor'
2024-05-22T18:30:00Z [info] system_monitor: uptime=15h45m

The System Froze Before the Banner Exchanged

```
ssh devbox
                    ⋮

ssh: connect to host devbox
port 22: Connection timed out
```

System Vitals Panel

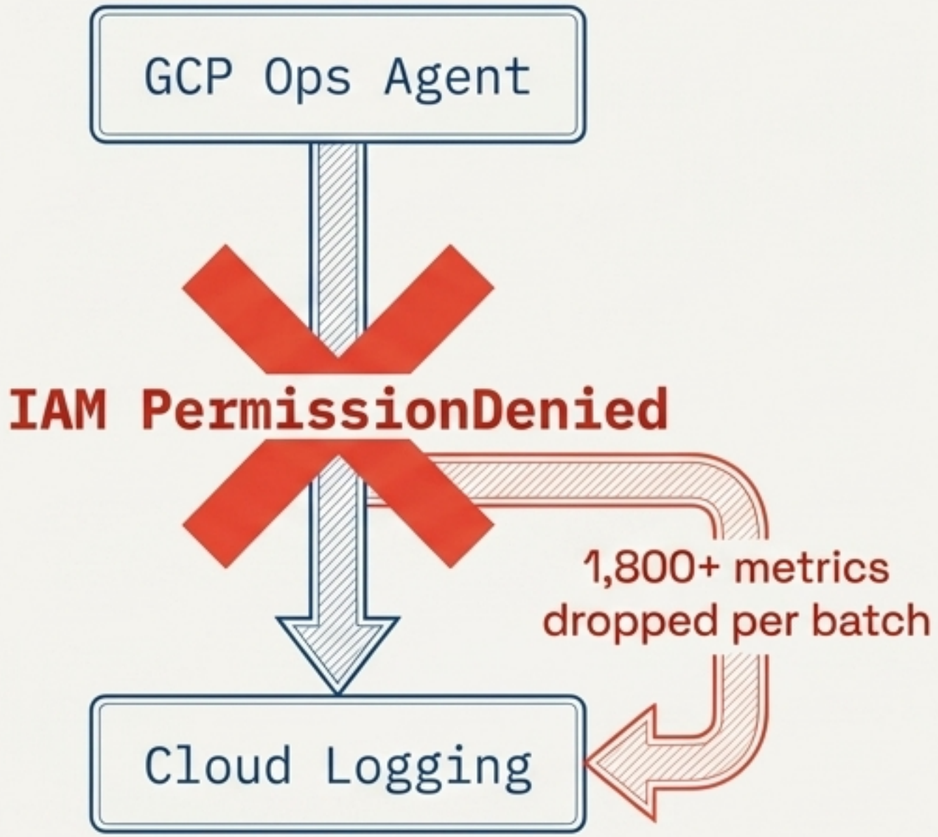
Kernel: ALIVE

Userspace: FROZEN

The Machine	The Context	The Symptom
e2-highmem-4 GCP devbox with 32GB RAM.	Upgraded and hardened after a previous Chrome fork-bomb crash.	Status showed RUNNING, but SSH connections timed out instantly. The kernel was responsive; nobody was home.

First Clue: The Monitoring Agent Compounded the Crash

The Irony of otelopscol: The tool installed to prevent crashes accelerated this one through unchecked I/O spam.

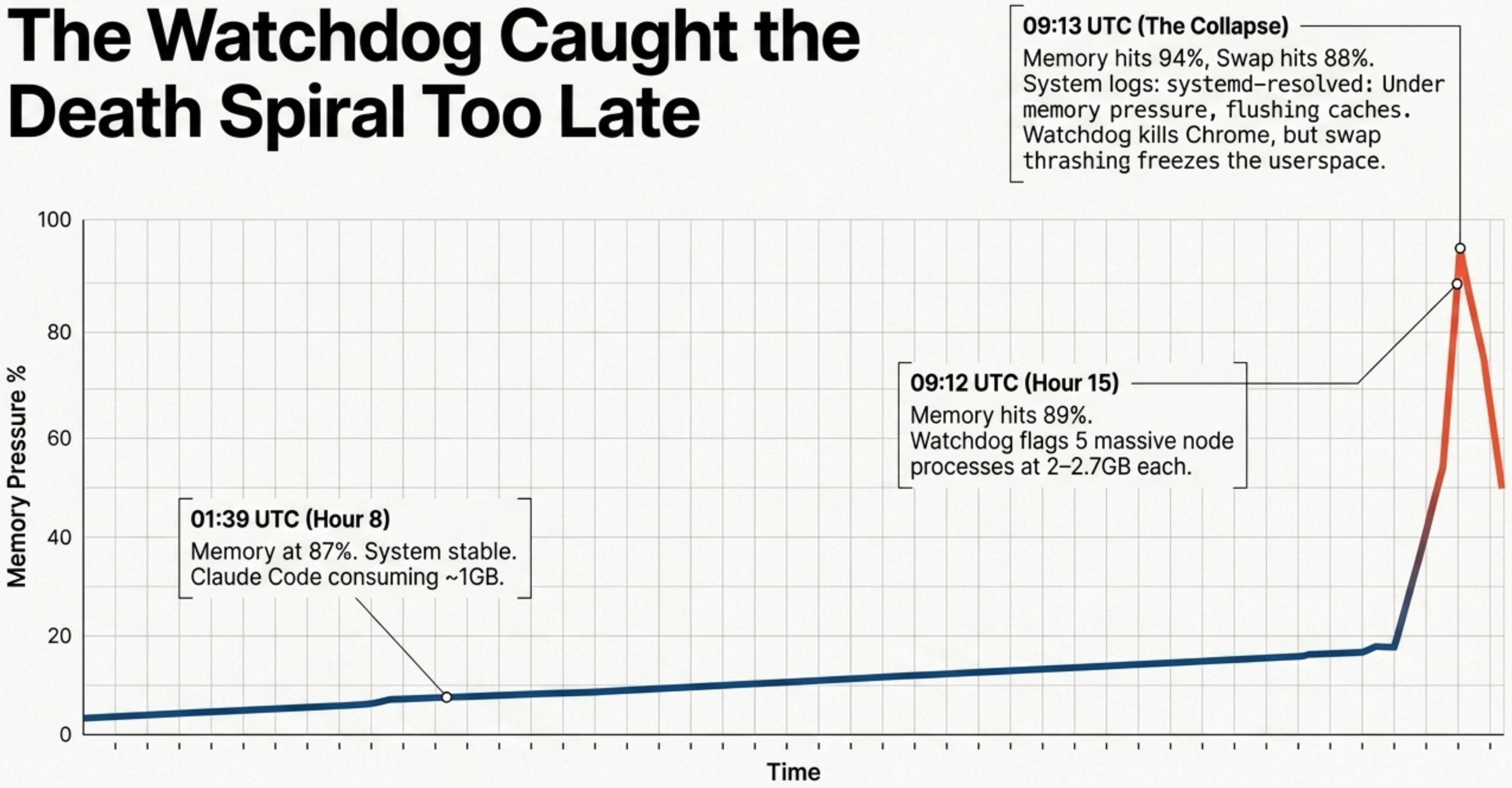


Upon pulling serial console logs, the first red flag wasn't memory—it was disk I/O.

The GCP Ops Agent lacked proper IAM permissions, spam-looping PermissionDenied errors.

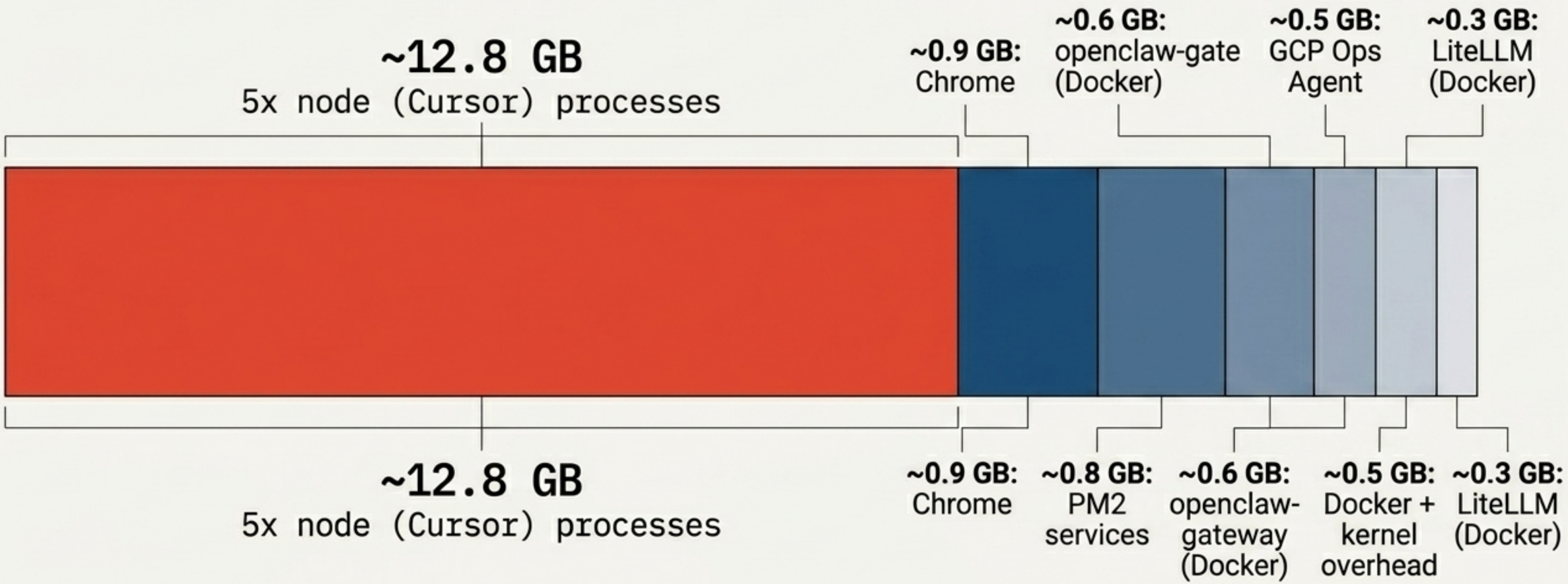
Every failed metric push generated massive error logs, eating disk I/O and memory right when the system was most vulnerable.

The Watchdog Caught the Death Spiral Too Late



Autopsy of a 20GB Footprint

Nearly half of the 32GB machine's RAM was consumed by just five rogue node processes, alongside 3.5GB of exhausted swap space.



The Twist: Blaming the Wrong AI

The Assumption

process: claude

~~5 sessions at 2.7GB~~

The False Suspect: Claude Code is known to grow to 1-2GB per session. Five heavy sessions seemed like a perfectly plausible culprit for the 12GB spike.

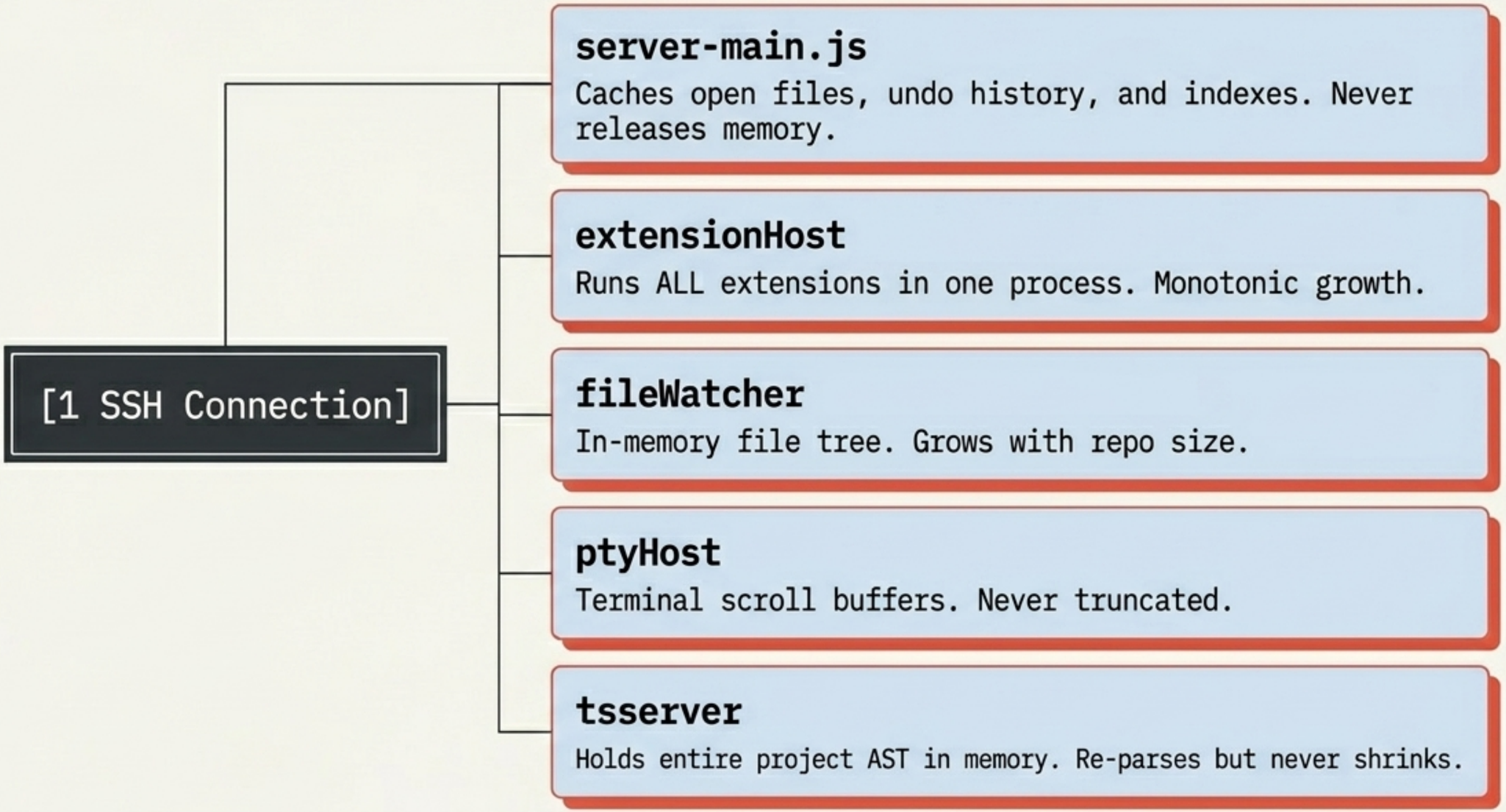
The Reality

process: node

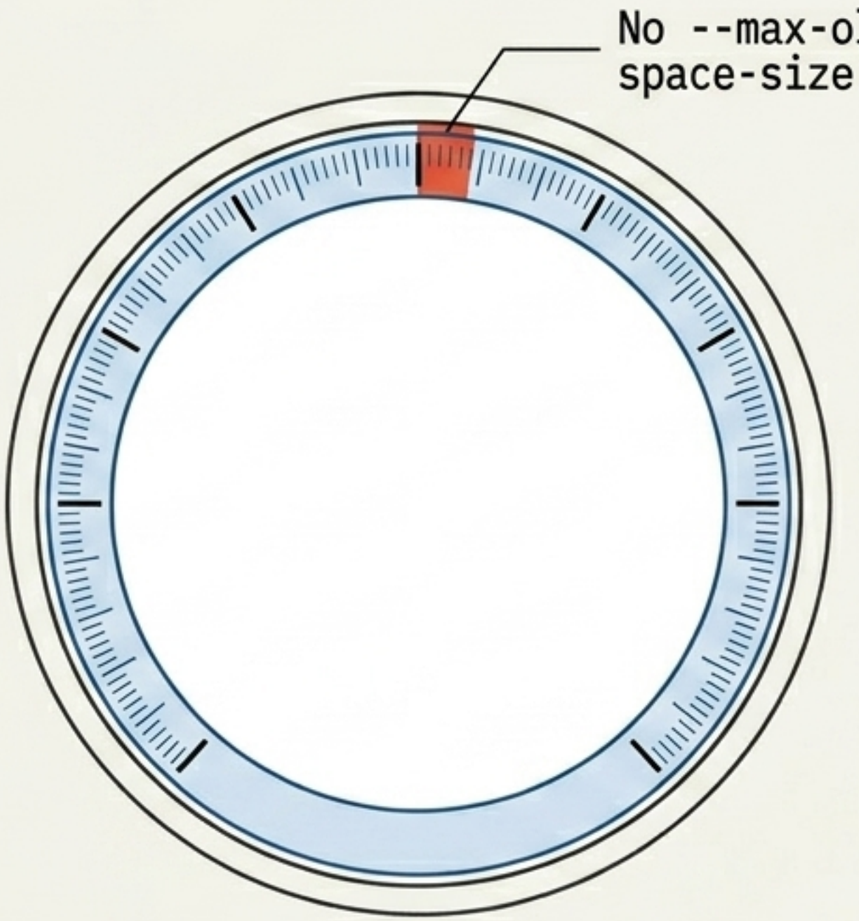
The Verdict: The watchdog explicitly logged 'node'. It wasn't Claude. It was Cursor's remote SSH server silently leaking for 15 hours straight.

The Crucial Distinction:
Process names matter. Claude registers as 'claude' in the comm field. Relying on assumptions leads to the wrong fix.

Why Cursor Leaks: A 5-Pronged Architecture



V8's Default Trap: Lazy Garbage Collection



V8 Heap Utilization
(Pressure Cooker)

01

No Ceilings

Cursor relies on V8's default heap limit (~4GB per process).

02

Lazy Cleanup

V8's garbage collector only triggers a major sweep under extreme memory pressure (close to the absolute limit).

03

The Math

Over 15 hours, each of the 5 processes grew from ~400MB to 2.7GB (a 6-7x expansion). With no forced collection, 12.8GB accumulated silently.

The Three-Part Remediation Strategy

Fix the Ops Agent

Action: Granted **monitoring.metricWriter** and **logging.logWriter** IAM roles to the VM.

Impact: Stopped the catastrophic I/O and memory drain caused by error-log spam error-log spam loops.

Watchdog v2

Action: Lowered Lowered intervention thresholds drastically.

Rules:

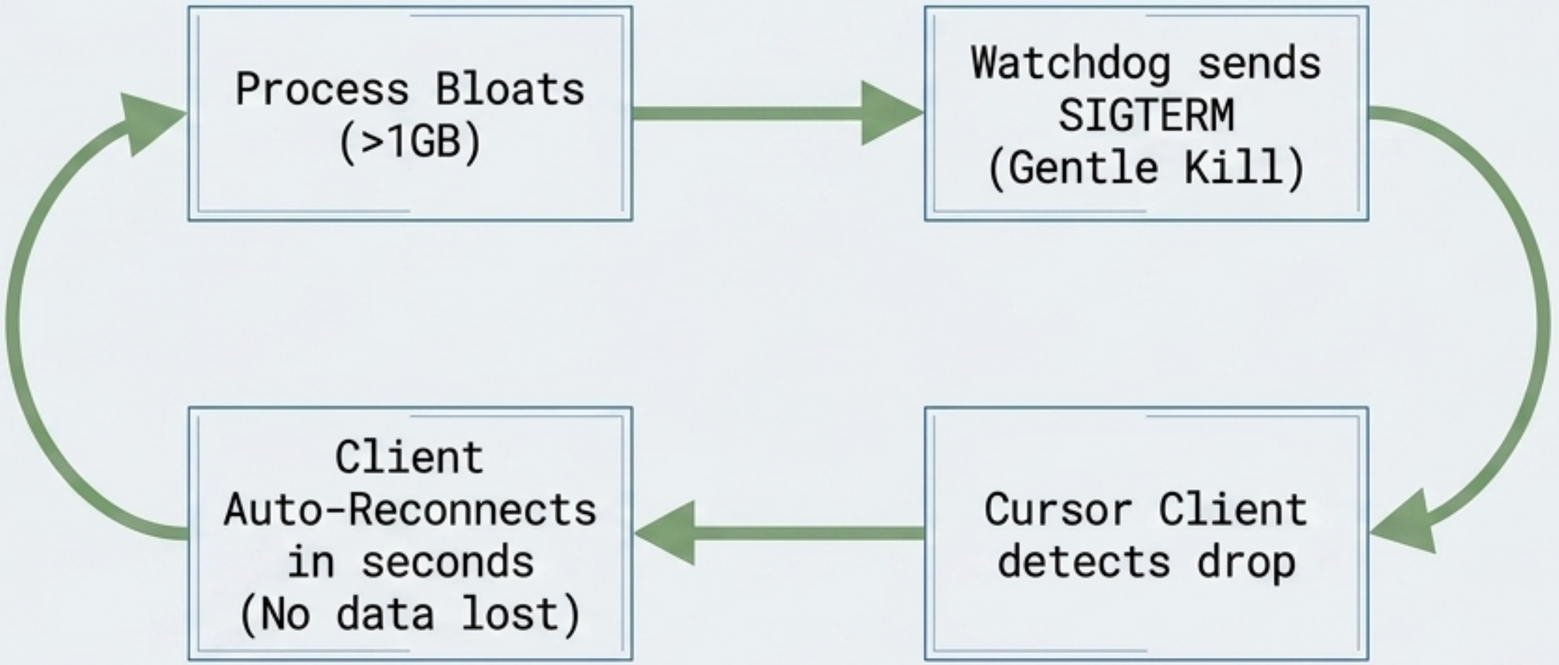
75%: Warn/Log
80%: Kill Chrome as sacrificial lamb
85%: Kill any node process > 1GB RSS

Cursor Memory Guard

Action: Deployed a 5-minute cron job to ruthlessly SIGTERM bloated Cursor processes.

Impact: Forced garbage collection via intentional process termination.

Reconnectable Architectures Change the Rules



Killing a bloated server process isn't destructive—it's just forced garbage collection that V8 refuses to do on its own.

Cursor's remote server is highly stateless. The client handles drops gracefully with a brief "Reconnecting..." banner, making aggressive memory guards a viable feature rather than a bug.

The Underlying Pattern: Unbounded Memory Always Fails

Incident	Specs	Root Cause
Part 3 Crash	16GB RAM	Chrome Snap Fork-Bomb
Part 7 Crash	32GB RAM	Cursor Node Leak

> Upgrading from 16GB to 32GB bought headroom, but it didn't change the fundamental dynamic. Without strict, per-process memory limits, any long-running process with lazy garbage collection will eventually consume every byte of RAM available to it. More RAM just means a slower crash.

Three Lessons for the Incident Dossier

01	**Monitor the Monitors** Tools installed to catch failures can cause them. Monitoring agents require correct permissions and hard resource constraints.
02	**Read the Logs, Question the Assumptions** Assuming “`node`” meant Claude would have led to the wrong fix. The process name in the `comm` field is the ultimate source of truth.
03	**Exploit Statelessness** When an architecture is designed to auto-reconnect gracefully, you can—and should—use aggressive process termination as a legitimate memory management strategy.