System Monitoring & Performance Diagnosis

A Complete 10-Chapter Technical Support Course

Topics covered:

A resource-first "it's slow" mindset · CPU load, utilization & run queues

Memory, cache & the OOM killer · disk I/O and disk space

Network as a performance symptom · reading monitoring dashboards

Process-level diagnosis · sustained trends vs. normal spikes

Capstone: three real performance 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

The "It's Slow" Starting Point: A Resource-First Diagnostic Mindset

System Monitoring & Performance Diagnosis

Chapter 1 · The "It's Slow" Starting Point: A Resource-First Diagnostic Mindset

A ticket comes in: "the app is slow." One engineer restarts the service, waits to see if the complaints stop, and moves on. The other opens a resource monitor first, sees exactly what the system was doing in the moments before the restart would have wiped it away, and only then decides what to do. This course is entirely about becoming the second engineer — not because a restart never helps, but because checking first turns "it's slow" from a vague complaint into a specific, evidence-backed finding.

Four Things "Slow" Could Mean

"Slow" isn't one problem — it's a category covering at least four genuinely different resource bottlenecks, each with its own evidence and its own fix:

RESOURCE	WHAT IT LOOKS LIKE WHEN IT'S THE BOTTLENECK	COVERED IN
CPU	High load, processes competing for processor time, everything feels uniformly sluggish	Chapter 2
Memory	The system starts swapping to disk, or a process is killed outright for using too much	Chapter 3
Disk I/O	Reads and writes queue up and take far longer than usual, even though CPU and memory look fine	Chapter 4
Network	The local system itself is healthy, but something between it and the user is slow or lossy	This site's own *Network Troubleshooting* (`netdiag1`)

Notice the last row doesn't point at a chapter in this course at all — Chapter 6 covers exactly how to recognize that pattern and hand off cleanly, rather than re-teaching a course this site already has.

Scoping the Complaint First

Before touching any specific resource, a handful of questions narrow "it's slow" down dramatically — the same discipline this site's own `netdiag1` course applies to connectivity complaints, adapted here for performance:

QUESTION	WHY IT MATTERS
One machine, or many?	One machine points at something local to it; many machines at once points at something shared — a network dependency, a database, or a common upstream cause
One resource elevated, or several at once?	Several resources moving together is itself a clue — for example, heavy disk I/O can starve the CPU of work to do while it waits, which shows up as elevated CPU "wait" time rather than a genuine CPU bottleneck
Sudden onset, or a gradual drift?	A sudden change points toward a specific triggering event — a deployment, a cron job, a traffic spike; a slow drift over days or weeks points toward a genuine capacity or leak problem
Does it line up with a known event?	A deployment, a scheduled job, or a marketing campaign happening at the same time is often the actual explanation, hiding in plain sight

Why "Just Restart It" Is a Guess, Not a Diagnosis

A restart is tempting precisely because it often works — many resource problems do clear up, at least temporarily, once a process starts fresh. But "it worked" and "I know why it was happening" are different things entirely, and a restart that isn't preceded by a quick look at what the system was actually doing throws away the one chance to find out.

> **A RESTART DESTROYS THE EXACT EVIDENCE YOU NEED**
>
> Once a struggling process restarts, its accumulated memory usage resets, its open file handles close, and its current CPU activity vanishes — the live picture of what was actually going wrong is gone the moment the process exits. This is the same underlying caution as this site's own *Logging & Log Analysis* course's warning against deleting a log file mid-incident: whatever's about to disappear might be the only copy of the evidence that explains what happened. If the same problem returns tomorrow, you'll be starting from zero again.

A Concrete Example: One Symptom, Several Different Causes

Take "the app is slow" and see how differently it resolves depending on which resource is actually behind it:

- **A runaway process pegging every CPU core:** a genuine CPU bottleneck — see Chapter 2
- **The system swapping heavily to disk:** a memory shortage severe enough that the OS is paging active memory out — see Chapter 3
- **A nightly backup job saturating the disk:** a disk I/O bottleneck that happens to overlap with business hours today — see Chapter 4
- **Everything local looks perfectly healthy:** the actual cause is very likely on the network path, not this machine at all — see Chapter 6, which hands off to `netdiag1`

Four genuinely different root causes, all hiding behind the same two words. Scoping the complaint first, then checking the right resource, is what tells them apart.

A Quick First Look

Before diving deep into any single resource, a quick overall check is worth running first — `top` on Linux, Task Manager or Resource Monitor on Windows. You'll learn to properly interpret every line of this in Chapters 2 and 3, but even without that depth yet, notice how much is visible in one glance:

```
$ top

top - 14:32:07 up 12 days,  3:41,  2 users,  load average: 8.42, 6.15, 3.02
Tasks: 214 total,   3 running, 211 sleeping,   0 stopped,   0 zombie
%Cpu(s): 87.3 us,  9.1 sy,  0.0 ni,  2.1 id,  0.0 wa,  0.0 hi,  1.5 si,  0.0 st
MiB Mem :  16034.2 total,    412.6 free,  11203.8 used,   4417.8 buff/cache
MiB Swap:   2048.0 total,  1890.3 used,    157.7 free
```

A load average of `8.42`, very little "free" memory, and heavy swap usage are all visible in this one snapshot — but they don't carry equal weight, and this chapter deliberately isn't explaining why yet. Chapter 2 covers what actually makes a load average high or normal for a given machine, and Chapter 3 explains why the low "free" figure here is very likely a red herring while the swap usage is a genuine warning sign. This exact snapshot is worth remembering — both of the next two chapters come back to it directly.

> **A NUMBER ALONE ISN'T DIAGNOSTIC WITHOUT A BASELINE**
>
> Seeing "CPU at 80%" and immediately treating it as the problem skips a real question: is 80% actually unusual for this machine, at this time of day, under its normal workload? A batch-processing server that regularly runs at 80% CPU during its nightly job isn't showing a problem at all — it's showing normal, expected behavior. A number only becomes evidence once it's compared against what's typical for that specific system.

What This Course Covers

Reading CPU load and utilization correctly, understanding what "used" memory actually means, telling a busy disk apart from a genuinely slow one, tracking down where disk space actually went, recognizing when "slow" is really a network problem in disguise, reading a monitoring dashboard's own summarized view, drilling from system-wide symptoms down to the one specific process responsible, and telling a real trend apart from a normal, temporary spike. The capstone applies all of it to three realistic performance tickets.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning, why restarting a slow service before checking any resource metrics can make a recurring problem harder to solve rather than easier.

View solution

EXERCISE 2

A ticket says "the server is slow." List this chapter's four scoping questions, and explain what a specific answer to each one would point toward.

View solution

EXERCISE 3

Explain why this chapter says "CPU at 80%" isn't automatically evidence of a problem, using the batch-processing server example, and what would actually be needed to know whether it's a real issue.

View solution

CHAPTER 1 QUICK REFERENCE

- "Slow" covers (at least) four different bottlenecks: **CPU, memory, disk I/O, and network** — each with its own evidence and fix
- Scope first: **one machine or many? one resource or several? sudden or gradual? tied to a known event?**
- A restart may relieve the symptom, but it also **destroys the evidence** needed to know why it happened — echoing this site's own "never delete logs mid-incident" caution
- The same complaint can hide genuinely different causes — a runaway process, memory exhaustion, a disk-saturating job, or a problem that isn't local at all
- A resource number alone isn't diagnostic — it needs a **baseline** for what's normal on that specific system
- **Next chapter:** CPU: Reading Load, Utilization & Run Queues

CHAPTER 2 OF 10

CPU: Reading Load, Utilization & Run Queues

System Monitoring & Performance Diagnosis

Chapter 2 · CPU: Reading Load, Utilization & Run Queues

Chapter 1 left one number deliberately unexplained: a load average of `8.42` in that chapter's `top` snapshot. This chapter explains exactly what load average measures (and what it surprisingly includes), why it means nothing at all without knowing the machine's core count, and how to read the rest of the CPU line — including two fields, `wa` and `st`, that can mean "slow" isn't really a CPU problem at all.

Load Average vs. CPU Utilization: Two Different Numbers

CPU utilization (the percentage figures in `top`'s `%Cpu(s)` line) describes how busy the CPU is right now, over a very short interval. Load average is a different measurement entirely: a running average, over 1, 5, and 15 minutes, of the number of processes that were either using the CPU or waiting for something.

LOAD AVERAGE INCLUDES MORE THAN JUST CPU-HUNGRY PROCESSES

On Linux, a process waiting on disk I/O — not just one waiting for a CPU core to become free — counts toward load average too. This is a genuinely common source of confusion: a high load average doesn't necessarily mean the CPU is the bottleneck at all. Chapter 4 covers disk I/O in depth, but it's worth knowing now that this chapter's own load-average number can be telling a disk story as easily as a CPU one — which is exactly why the rest of this chapter's fields matter.

What "Normal" Load Actually Depends On: Core Count

A load average of `8` means something completely different on a 4-core machine than on a 16-core one — the raw number is meaningless without knowing how many cores are actually available to share the work.

```
$ nproc
4
```

LOAD AVERAGE ÷ CORE COUNT	WHAT IT SUGGESTS
Well under 1.0	Comfortably idle — plenty of spare capacity
Around 1.0	Fully utilized, but not yet queueing — every core is busy, nothing is waiting

LOAD AVERAGE ÷ CORE COUNT	WHAT IT SUGGESTS
Well over 1.0	Genuinely overloaded — more work wants to run than the machine can currently handle, and some of it is waiting

Chapter 1's own example machine has **4 cores**. A load average of `8.42` divided by 4 is just over **2.1 per core** — meaning, on average, roughly twice as much work wants to run as the machine can actually handle at once. That's a genuine, meaningful overload signal, not a false alarm.

The %Cpu(s) Line In Depth

Chapter 1's snapshot also showed: `87.3 us, 9.1 sy, 0.0 ni, 2.1 id, 0.0 wa, 0.0 hi, 1.5 si, 0.0 st`. Each field means something specific:

FIELD	WHAT IT MEASURES
us	Time spent running normal application ("user-space") code
sy	Time spent in the kernel — system calls, scheduling, and similar overhead
ni	User-space time specifically from processes running at a lowered ("niced") priority
id	Genuinely idle — nothing to do
wa	**iowait** — the CPU is idle, but specifically because it's waiting on a disk (or other I/O) operation to finish, not because there's no work
hi / si	Hardware and software interrupt handling
st	**Steal time** — on a virtual machine, time this VM wanted to run but the hypervisor gave the physical CPU to a different tenant instead

In Chapter 1's snapshot, `wa` and `st` are both `0.0`, and `us` alone accounts for 87.3% — confirming this genuinely is a CPU-bound problem, not a disk bottleneck wearing a CPU-shaped disguise, and not a cloud tenant losing CPU time to a noisy neighbor.

HIGH WA MEANS GO CHECK CHAPTER 4, NOT THIS CHAPTER

If a system shows a high load average alongside a high `wa` figure, the CPU itself usually isn't the actual bottleneck — it's sitting idle, waiting on disk. Treating that as a CPU problem and looking for a runaway process will come up empty; the real evidence lives in disk I/O metrics instead.

STEAL TIME IS A REAL CAUSE YOU CAN'T FIX LOCALLY

On a cloud or virtualized server, a nonzero `st` figure means the physical hardware underneath this machine is genuinely oversubscribed by the hosting provider — no amount of tuning inside this particular system will fix it,

because the CPU time is being taken away one layer below where this system has any control. Recognizing steal time early avoids a long, fruitless search through this machine's own processes for a cause that isn't there.

Run Queue: Processes Actually Waiting for a Turn

`vmstat` shows a more direct number: the `r` column, the count of processes literally waiting for a CPU core right now — a narrower, more specific signal than load average's broader, I/O-inclusive definition.

```
$ vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 7  0 1935872 422528 145200 4523000    0    0     2     8  980 2100 85  9  4  2  0
 8  0 1935872 415200 145200 4519200    0    0     0     4  975 2050 88  8  3  1  0
 6  0 1935872 409600 145200 4517800    0    0     0     0  960 1980 86  9  4  1  0
```

An `r` value consistently well above the core count (6–8, on a 4-core machine) across several samples confirms the same story load average already suggested: genuine, sustained CPU contention, not a brief one-off spike.

WINDOWS DOESN'T HAVE A DIRECT EQUIVALENT TO LOAD AVERAGE

Windows relies primarily on CPU utilization percentage, viewed in Task Manager or Resource Monitor, rather than a load-average-style rolling number. Performance Monitor's **Processor Queue Length** counter is the closest analogue to `vmstat`'s `r` column — the number of threads actually waiting for CPU time — but there's no built-in figure that mirrors Linux's 1/5/15-minute averaged load number.

Working Example: Fully Reading Chapter 1's Snapshot

Put together: a load average of `8.42` on a 4-core machine is roughly **2.1 per core** — genuinely overloaded. The `%Cpu(s)` line's `0.0 wa` and `0.0 st` rule out both disk I/O and hypervisor steal time as explanations. `87.3% us` confirms the time is being spent running actual application code, not the kernel or interrupt handling. A follow-up `vmstat` check showing `r` consistently at 6–8 confirms this isn't a passing blip. Every piece of evidence agrees: this machine has a genuine CPU bottleneck, and the next step is identifying which specific process is responsible — Chapter 8's own territory.

Hands-On Exercises

EXERCISE 1

Explain why a load average of 8 could be either a genuine problem or completely normal, depending on one specific piece of information this chapter says is required to interpret it.

View solution

EXERCISE 2

A machine shows a high load average alongside a high `wa` figure in `top`. Explain why this chapter says that's misleading to treat as a CPU problem, and where the real evidence would actually be found.

View solution

EXERCISE 3

Explain what "steal time" specifically measures, and why this chapter says it can't be fixed by anything done inside the affected virtual machine itself.

View solution

CHAPTER 2 QUICK REFERENCE

- **Load average** includes processes waiting on I/O, not just CPU-hungry ones — it can reflect a disk problem, not just a CPU one
- Always divide load average by **core count** (`nproc`) before judging it — the raw number alone is meaningless
- `%Cpu(s)` breakdown: **us** (app code), **sy** (kernel), **wa** (iowait — a disk story, not CPU), **st** (steal time — a hypervisor-level cause you can't fix locally)
- `vmstat`'s **r** column is a more direct, narrower measure of processes actually waiting for a CPU core right now
- Windows has no direct load-average equivalent — CPU utilization % and Processor Queue Length are the closest counterparts
- **Next chapter:** Memory: Used, Free, Cached & the "Available" Metric

CHAPTER 3 OF 10

Memory: Used, Free, Cached & the "Available" Metric

System Monitoring & Performance Diagnosis

Chapter 3 · Memory: Used, Free, Cached & the "Available" Metric

Chapter 1's snapshot showed only `412.6` MiB "free" out of 16 GB total, alongside heavy swap usage — and deliberately left both unexplained. This chapter resolves both: why the low "free" figure is very likely a false alarm, and why the swap figure deserves real attention, though — as the closing worked example shows — not always for the reason it first appears to.

The Classic "No Free Memory" False Alarm

Linux treats unused RAM as wasted RAM. Rather than leaving memory sitting empty, the kernel fills it with a cache of recently-read disk data — the **page cache** — so a repeated read can be served instantly from memory instead of hitting the disk again. That cache shows up in `buff/cache`, and on any machine that's been running a while, it tends to consume most of whatever isn't already allocated to running processes.

CACHED MEMORY IS RECLAIMABLE, ESSENTIALLY FOR FREE

Unlike memory a process is actively using, page cache can be dropped almost instantly the moment an application actually needs that RAM — there's no cost to reclaiming it, since the same data can simply be re-read from disk later if needed again. A low "free" number with a large "buff/cache" number isn't a shortage at all; it's the kernel using spare capacity productively, exactly as designed.

Reading free -h Properly

```
$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi        11Gi       402Mi       210Mi       4.3Gi       4.1Gi
Swap:          2.0Gi       1.8Gi       154Mi
```

COLUMN	WHAT IT ACTUALLY MEANS
used	Memory genuinely allocated to running processes — the number closest to "real" usage
free	Memory touched by nothing at all — usually small on a healthy, active machine, and not a cause for concern by itself
buff/cache	Reclaimable page cache — available the instant something else needs it

COLUMN	WHAT IT ACTUALLY MEANS
available	The kernel's own estimate of how much memory a new application could get right now without needing to swap — the single most useful number here

LOOK AT "AVAILABLE" FIRST, NOT "FREE"

"Free" answers a narrow, mostly uninteresting question — how much memory is completely untouched. "Available" answers the question that actually matters for diagnosing memory pressure: how much memory could this system hand to a new process right now, accounting for cache it could reclaim instantly. A low "free" figure next to a healthy "available" figure is not a problem.

When Memory Pressure Is Real: Swap

Swap is fundamentally different from cache. When physical RAM is genuinely insufficient, the kernel moves active, in-use memory pages out to disk to free up space — a process that's orders of magnitude slower than RAM itself, since it involves real disk I/O for data a process actually needs. Unlike reclaiming cache, swapping has a real, measurable performance cost.

TOTAL SWAP USED ISN'T THE SAME AS SWAPPING ACTIVELY HAPPENING RIGHT NOW

A high total swap-used figure can be historical — memory that was pushed out during an earlier period of pressure, or proactively swapped out because it was cold and unused, and simply hasn't been swapped back in since nothing has needed it. What actually indicates an ongoing, active problem is `vmstat`'s `si` and `so` columns (swap in/out per second) showing sustained, nonzero movement — not the total figure alone.

The OOM Killer: When Memory Pressure Gets Severe Enough

If physical RAM and swap are both genuinely exhausted, Linux's Out-Of-Memory (OOM) killer steps in and picks a process to terminate outright, freeing its memory by force. A process that simply vanishes, with no crash log of its own and no obvious cause, is a classic OOM-kill symptom — confirmed by checking the kernel's own log:

```
$ dmesg | grep -i "killed process"
[124560.221033] Out of memory: Killed process 18832 (java) total-vm:4194304kB, anon-rss:3801216kB

# or, on a systemd-based system
$ journalctl -k | grep -i oom
```

Reading these entries correctly is exactly the log-reading discipline this site's own **Logging & Log Analysis** (`log1`) course covers in depth — this is one specific, high-value example of it, applied to a memory-pressure

investigation specifically.

Windows' Own Memory Model

Windows doesn't expose the exact same free-vs-cache split. Task Manager shows figures like "In use," "Available," "Committed," and "Cached," alongside a "Standby" list — memory holding recently-used file data that can be reclaimed, conceptually similar to Linux's page cache. Task Manager's own "Available" figure already accounts for reclaimable standby memory, in the same spirit as Linux's "available" column, though the two aren't a precise one-to-one match.

Working Example: Fully Reading Chapter 1's Snapshot

Chapter 1 showed `412.6` MiB free, `4417.8` MiB buff/cache, and swap at `1890.3` of `2048.0` MiB used. The low "free" figure is exactly this chapter's false alarm — a healthy 4.3 GiB of reclaimable cache sits right next to it. The swap figure, at first glance, looks like a serious active problem. But checking whether it's actually active right now means returning to Chapter 2's own `vmstat` sample from this same machine:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 7  0 1935872 422528 145200 4523000    0    0     2     8  980 2100 85  9  4  2  0
 8  0 1935872 415200 145200 4519200    0    0     0     4  975 2050 88  8  3  1  0
 6  0 1935872 409600 145200 4517800    0    0     0     0  960 1980 86  9  4  1  0
```

`si` and `so` are both `0` across every sample — nothing is actively swapping in or out right now. Combined with Chapter 2's own finding of `0.0 wa`, the picture is consistent and honest: this machine's high swap-used figure is a leftover from an earlier period of pressure, not an active crisis, and the actual, currently-active bottleneck is exactly what Chapter 2 already found — genuine CPU contention. Memory isn't this machine's real problem right now; it's a secondary artifact worth noting, not the headline finding.

Hands-On Exercises

EXERCISE 1

Explain why a low "free" memory figure next to a large "buff/cache" figure isn't evidence of a memory shortage, using this chapter's own reasoning about page cache.

View solution

EXERCISE 2

Explain the difference between a high total swap-used figure and actively-occurring swapping, and which specific metric this chapter says distinguishes the two.

View solution

EXERCISE 3

In this chapter's worked example, explain why the machine's high swap-used figure was ultimately judged not to be the active problem, and what the real, currently-active bottleneck turned out to be.

View solution

CHAPTER 3 QUICK REFERENCE

- A low **"free"** figure is usually a false alarm — Linux deliberately fills spare RAM with reclaimable **page cache** (`buff/cache`)
- **"Available"** (in `free -h`) is the number that actually matters — an estimate of usable memory accounting for reclaimable cache
- Swap usage is the genuine warning sign, but a **high total** isn't the same as **actively swapping right now** — check `vmstat`'s `si` / `so` columns to tell them apart
- When memory and swap are both exhausted, Linux's **OOM killer** terminates a process outright — confirm with `dmesg` / `journalctl -k`
- Windows' "Standby" list plays a similar role to Linux's page cache, though the two aren't an exact match
- **Next chapter:** Disk I/O: IOPS, Throughput & Latency

CHAPTER 4 OF 10

Disk I/O: IOPS, Throughput & Latency

System Monitoring & Performance Diagnosis

Chapter 4 · Disk I/O: IOPS, Throughput & Latency

The machine from Chapters 1–3 turned out to be CPU-bound, with no meaningful disk involvement — its `wa` figure was `0.0` the whole time. This chapter is about the case Chapters 2 and 3 both mentioned but never demonstrated: a system where disk I/O genuinely is the bottleneck, using a fresh example — the nightly backup job Chapter 1 named as one of its four possible causes for "the app is slow."

Three Different Numbers, Three Different Questions

METRIC	THE QUESTION IT ANSWERS
IOPS	How many individual read/write operations happen per second
Throughput (MB/s)	How much total data moves per second
Latency	How long each individual operation takes to actually complete

These three don't move together the way it might seem — a workload of many small, random operations (a busy database) is IOPS-bound and can look fine on throughput while genuinely struggling on IOPS and latency; a workload of a few large, sequential transfers (a backup job copying big files) is throughput-bound and can post huge MB/s numbers while barely touching IOPS. Knowing which kind of workload you're looking at changes which number actually matters.

iostat: Reading Linux's Own Disk Tool

```
$ iostat -x 1 3
Device     r/s     w/s   rkB/s    wkB/s   await  %util
sda       12.00  340.00   480.00  42500.00  85.40  98.70
```

`r/s` and `w/s` are IOPS (reads and writes per second); `rkB/s` and `wkB/s` are throughput; `await` is average latency in milliseconds, including time spent queued, not just time spent actually transferring data.

%UTIL DOESN'T MEAN WHAT IT SOUNDS LIKE ON MODERN STORAGE

`%util` measures how much of the time the device had at least one request outstanding — not literally "how full" the disk is in any simple sense. A modern SSD can handle many requests in parallel through deep internal queuing, so it's entirely possible to see `%util` near 100% while the device is still comfortably keeping up with additional load. Treating `%util` alone as "the disk is maxed out" is the same category of mistake as reading a high load average without checking core count in Chapter 2.

The Metric That Actually Matters Most: Latency

`await` is usually the number closest to what a user actually experiences — a slow individual operation feels slow regardless of how busy the device looks overall or how much total throughput it's pushing. Rough reference points worth having in mind, since (per Chapter 1) a number alone means nothing without a baseline:

STORAGE TYPE	TYPICAL LATENCY
NVMe SSD	Well under 1 ms
SATA SSD	Low single-digit ms
Spinning HDD	5–15 ms typical, often worse under random access due to physical seek time

Busy vs. Slow: A Genuine Distinction

%UTIL	AWAIT	WHAT IT MEANS
High	Low	Busy, but healthy — heavily utilized and keeping up comfortably
Low/moderate	High	Genuinely struggling — each request takes a long time even though the device isn't constantly occupied; often a queue-depth or hardware problem
High	High	Genuinely saturated — busy and slow at the same time, the clearest sign of a real bottleneck

iotop: Finding Which Process Is Responsible

```
$ sudo iotop -o
  PID  PRIO  USER     DISK READ   DISK WRITE  COMMAND
 4821  be/4  root       2.10 M/s    38.20 M/s  rsync -a /data /backup/
```

`-o` shows only processes actually doing I/O right now — a fast way to skip straight to the culprit rather than scanning every process on the system.

WINDOWS' EQUIVALENT VIEW

Resource Monitor's Disk tab shows **Active Time %** (roughly `%util`'s counterpart), **Avg. Disk sec/Read** and **Avg. Disk sec/Write** (latency), and **Disk Queue Length** — plus a live, per-process breakdown of exactly which process is generating the read/write activity, the same job `iotop` does on Linux.

Working Example: Confirming Chapter 1's Backup-Job Scenario

Chapter 1 named "a nightly backup job saturating the disk" as one possible cause behind "the app is slow." A different server than Chapters 1–3's own machine shows exactly this: `iostat -x 1` reports `await` at **85.40 ms** — dramatically above any of this chapter's reference points for either SSD or spinning disk — alongside `%util` at **98.70%**. High and high: genuinely saturated, not just busy. `iotop` immediately confirms the responsible process: an `rsync` backup job writing at over 38 MB/s, which today happens to be running well past its usual overnight window and directly overlapping with business-hours traffic. The fix isn't a code change or a resource upgrade — it's rescheduling or throttling the backup job so it no longer competes with live traffic for the same disk.

Hands-On Exercises

EXERCISE 1

Explain why a busy database server and a backup job copying large files can both show heavy disk activity, yet be bottlenecked on two genuinely different metrics.

View solution

EXERCISE 2

Explain why this chapter says `%util` at 100% doesn't automatically mean a disk is overloaded, particularly for modern SSDs.

View solution

EXERCISE 3

In this chapter's worked example, explain what specifically confirmed the disk was genuinely saturated (not just busy), and how `iotop` identified the actual cause.

View solution

CHAPTER 4 QUICK REFERENCE

- **IOPS, throughput, and latency** answer three different questions — a workload can be bound by one while looking fine on another
- `iostat -x` shows all three at once; `await` is usually closest to what a user actually feels
- **%util isn't literally "how full"** — modern SSDs with deep queuing can show near-100% while still comfortably keeping up
- Reference latencies: NVMe well under 1ms, SATA SSD low single-digit ms, HDD 5–15ms+
- **High %util + low await** = busy but healthy; **low %util + high await** = genuinely struggling; **both high** = genuinely saturated
- `iotop -o` finds the specific process responsible; Windows' Resource Monitor Disk tab does the same job
- **Next chapter:** Disk Space: Filling Up, and Where It Actually Went

CHAPTER 5 OF 10

Disk Space: Filling Up, and Where It Actually Went

System Monitoring & Performance Diagnosis

Chapter 5 · Disk Space: Filling Up, and Where It Actually Went

Chapter 4 covered disk performance — how fast a disk can service requests. This chapter covers a different, equally common problem: a disk running out of space entirely. The two headline tools, `df` and `du`, usually agree — until they don't, and the gap between them turns out to be this chapter's own most useful diagnostic finding.

df: The Quick Overview

```
$ df -h /var
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        50G   49G  1.2G  98% /var
```

`df` reports space usage at the filesystem level — fast, and a good first check, but it says nothing about which files or directories are actually responsible.

du: Finding Where the Space Actually Went

```
$ du -sh /var/* 2>/dev/null | sort -rh | head -5
18G     /var/lib
9.2G    /var/log
2.1G    /var/cache
890M    /var/spool
210M    /var/tmp
```

`du` walks the actual directory tree, adding up real file sizes, and is the natural next step once `df` has confirmed a filesystem is genuinely full.

The Classic df/du Mismatch

Adding up the `du` figures above comes to roughly 30 GB. `df` reported 49 GB used on the same filesystem. That's not a rounding error — it's a real, specific, and genuinely common phenomenon.

A DELETED FILE CAN STILL OCCUPY REAL DISK SPACE

When a file is deleted while a running process still has it open, Linux doesn't actually reclaim the space — the space is only freed once every process holding it closed has released it. `du` walks the visible directory tree by name, and a deleted file has no name left to walk to, so it's invisible to `du` entirely. `df`, which reports actual block-level usage, still counts it. The gap between the two totals is real, allocated disk space, held open by a process, with no visible trace in the filesystem tree at all.

Finding the Phantom Space

```
$ sudo lsof +L1
COMMAND   PID   USER   FD   TYPE DEVICE     SIZE/OFF NLINK    NODE NAME
java     2214   app    5w   REG    8,1  19327352832     0  524291 /var/log/app/service.log (deleted)
```

`lsof +L1` lists open files with a link count of 0 — files that have been deleted but are still held open by a running process, exactly this chapter's own missing gap. The `SIZE/OFF` column here, roughly 18 GB, accounts for almost the entire shortfall between `df` and `du`'s totals.

THIS CONNECTS DIRECTLY TO LOG ROTATION

A classic cause of exactly this scenario: a naive cleanup script deletes a large log file directly, without signaling the application that's still writing to it to reopen a fresh file. This site's own *Logging & Log Analysis* (`log1`) course covers the proper, proactive fix — `logrotate`, which handles this correctly — in its own Chapter 9. This chapter is about diagnosing the symptom once it's already happened, not preventing it in the first place.

Windows' Own Disk-Usage Story

Windows' built-in tooling for "where did my space actually go" is comparatively weak — Storage settings gives only a rough, high-level breakdown by category, not a real directory-by-directory view. Third-party tools like WinDirStat fill that gap with a genuine visual breakdown, similar in spirit to `du`. A related situation to the deleted-but-open-file gotcha can occur on Windows too — a file with an open handle can resist being fully reclaimed even after deletion — though NTFS's own file-locking model differs enough from Linux's link-count semantics that the two aren't a precise match.

When Disk Fills Completely

A filesystem that reaches 100% doesn't just stop growing gracefully — a database can refuse further writes, a web server can fail to log or even serve requests, and in severe cases the operating system itself can struggle, since some system services genuinely need a small amount of free space to operate at all. This is worth treating with real urgency once a filesystem is closing in on full, not just noted and revisited later.

Working Example: The Missing 18 GB

A ticket: `/var` is at 98% and climbing, but a directory walk only accounts for about 30 GB of the 49 GB `df` reports as used. `lsof +L1` finds the answer directly: a Java application still holding open a deleted 18 GB log file — deleted by an overnight cleanup script that never told the running process to reopen a fresh one. Restarting the application (or, less disruptively, sending it a signal to reopen its log handles, if it supports one) releases the space immediately, confirmed by a fresh `df -h` showing usage drop back in line with what `du` was already reporting.

Hands-On Exercises

EXERCISE 1

Explain why `du`'s totals can add up to noticeably less than what `df` reports as used on the same filesystem, using this chapter's own explanation.

View solution

EXERCISE 2

Explain what `lsof +L1` specifically looks for, and why that's exactly the right tool for the df/du mismatch this chapter describes.

View solution

EXERCISE 3

Explain what caused the missing 18 GB in this chapter's worked example, and why simply deleting a large log file directly (instead of using something like logrotate) can lead to exactly this situation.

View solution

CHAPTER 5 QUICK REFERENCE

- `df -h` shows overall filesystem usage; `du -sh` shows where it lives in the directory tree
- A **deleted file still held open by a running process** occupies real space invisible to `du` but still counted by `df` — the classic mismatch
- `lsof +L1` finds exactly these deleted-but-open files directly
- The proactive fix (`logrotate`) is covered in *Logging & Log Analysis* (`log1`) Chapter 9 — this chapter diagnoses the symptom after the fact

- Windows' built-in disk-usage tooling is comparatively weak — third-party tools like WinDirStat fill the gap
- A completely full disk has real downstream consequences — database write failures, logging failures, even OS-level instability
- **Next chapter:** Network as a Performance Symptom

CHAPTER 6 OF 10

Network as a Performance Symptom

System Monitoring & Performance Diagnosis

Chapter 6 · Network as a Performance Symptom

Chapters 2 through 5 covered CPU, memory, and disk — all local to the machine you're standing on. This chapter is deliberately short, and deliberately doesn't try to re-teach a topic this site already covers in full: it's about recognizing when "slow" isn't a local resource problem at all, running one quick local-side check before committing to that conclusion, and handing off cleanly to this site's own **Network Troubleshooting** (`netdiag1`) course once that's confirmed.

The Tell: Everything Local Looks Healthy

The single clearest signal that a symptom is network-related rather than a local resource problem: CPU, memory, and disk all check out clean using Chapters 2 through 5's own tools, and the complaint persists anyway. A machine with plenty of spare CPU, healthy memory, and fast disk I/O that's still "slow" from a user's point of view is very likely waiting on something outside itself — the network path to it, or the network path from it to something it depends on.

Two Genuinely Different Network Symptoms

DIRECTION	WHAT IT LOOKS LIKE
Slow to receive requests	Something upstream of this machine — a user's own connection, a CDN, a load balancer — is the bottleneck, not this server
Slow to call something else	This machine is healthy, but it's waiting on a downstream dependency — a database on another host, a third-party API — over a slow or lossy network path

A PROCESS WAITING ON THE NETWORK CAN LOOK DECEPTIVELY IDLE

A process stuck waiting on a slow remote call — a database query over the network, an outbound API request — often shows up as low CPU usage, since it genuinely isn't doing any computation while it waits. That can look like "the app isn't under any resource pressure" when what's actually happening is the app is stuck, doing nothing, purely because something it depends on hasn't answered yet. Low CPU alongside a slow response is itself a real clue pointing toward network or dependency latency, not away from it.

A Quick Local Check Before Handing Off

Before fully escalating, one more local-machine check is worth running: TCP retransmissions, a rough but genuinely useful local-side signal that packets are actually being lost somewhere on the network.

```
# First check
$ netstat -s | grep -i retrans
    2841 segments retransmitted

# Same check, 60 seconds later
$ netstat -s | grep -i retrans
    3960 segments retransmitted
```

The absolute number matters less than the trend — over 1,100 additional retransmitted segments in one minute, on a machine that isn't under heavy load per Chapters 2–5, is a real, active signal that packets are being lost somewhere on the path, not just background noise.

When to Hand Off to Network Troubleshooting

Once local resources are confirmed clean and retransmissions confirm real packet loss is actually occurring, the right next step is this site's own `netdiag1` course, not a deeper local investigation — its own ladder (from that course's Chapter 2 onward) is built exactly for this:

- Confirming DNS resolution and basic reachability — `netdiag1` Chapters 4–5
- Checking whether the specific port/service is actually reachable — `netdiag1` Chapter 6
- Firewalls, proxies, VPNs, and NAT possibly altering the path — `netdiag1` Chapters 7–8
- Reading the actual HTTP/TLS-level result once a connection is made — `netdiag1` Chapter 9

Repeating any of that material here would just be a shallower copy of a course this site already has — this chapter's own job is recognizing the handoff point, not replacing the course on the other side of it.

Working Example: The Slow Checkout Page

A ticket: the checkout page is slow, intermittently, for some users. CPU (Chapter 2) sits comfortably under load; memory (Chapter 3) shows healthy "available" figures with no active swapping; disk I/O (Chapter 4) is unremarkable. Everything local genuinely looks fine. A quick `netstat -s` check, repeated a minute apart, shows retransmissions climbing steadily — real, active packet loss. Handing off to `netdiag1`'s own techniques (a traceroute to the specific dependency the checkout flow calls) finds the actual cause: intermittent loss on the path to a third-party payment provider's own API, not anything on this application server at all. The checkout page isn't slow because of anything this machine is doing — it's slow because it's waiting on a lossy network path to someone else's service.

Hands-On Exercises

EXERCISE 1

Explain what this chapter says is the clearest signal that a "slow" complaint is network-related rather than a local resource problem.

View solution

EXERCISE 2

A colleague sees low CPU usage on a slow server and concludes there's no resource problem at all, so the app itself must be broken. Explain why this chapter says that conclusion can be wrong.

View solution

EXERCISE 3

Explain why this chapter checks TCP retransmissions twice, a minute apart, rather than just once, and what the difference between the two readings tells you that a single reading wouldn't.

View solution

CHAPTER 6 QUICK REFERENCE

- The clearest network tell: **CPU/memory/disk all check out clean, but the complaint persists anyway**
- Two directions: **slow to receive requests** (something upstream) vs. **slow to call something else** (a downstream dependency)
- A process waiting on a slow network call can show **deceptively low CPU** — that's a clue toward network latency, not away from it
- `netstat -s`'s retransmission count, checked twice and compared, is a quick local-side signal of real packet loss
- Once local resources are clean and retransmits confirm loss, hand off to **Network Troubleshooting (`netdiag1`)** rather than re-diagnosing locally
- **Next chapter:** Reading Monitoring Dashboards: Grafana & Cloud Console Metrics at a Glance

CHAPTER 7 OF 10

Reading Monitoring Dashboards: Grafana & Cloud Console Metrics at a Glance

System Monitoring & Performance Diagnosis

Chapter 7 · Reading Monitoring Dashboards: Grafana & Cloud Console Metrics at a Glance

Chapters 2 through 6 all assumed direct access — SSHing in and running a live command. Plenty of real environments don't work that way: the first (and sometimes only) view you get is a pre-built dashboard — Grafana, CloudWatch, Azure Monitor, or similar. Reading one correctly is a genuinely different skill from reading a live terminal, with its own real gotchas.

The Genuine Gotcha: Averaging Hides Spikes

Dashboards commonly store and display metrics aggregated over an interval — a 1-minute or 5-minute average is typical, both for storage efficiency and for keeping graphs readable. That averaging can genuinely hide the exact spike that caused real, user-visible pain.

A BRIEF, SEVERE SPIKE CAN VANISH INTO AN AVERAGE

Imagine CPU usage spikes to 100% for 15 seconds — long enough to cause real request timeouts — then drops back to normal. Averaged over a 5-minute window, that 15-second spike might only nudge the displayed line up to a mild-looking 60%. The dashboard isn't lying, but the number it's showing genuinely doesn't represent what actually happened at the moment users felt it.

Most monitoring tools let you switch the aggregation function applied to the same underlying data — from **avg** to **max** — over the same time window. Re-checking a suspicious-looking flat graph with `max` instead of `avg` is often the single fastest way to confirm whether a real spike is hiding underneath a smoothed-over average.

Reading a Typical Dashboard Layout

Most performance dashboards follow a similar shape: separate panels for CPU, memory, disk, and network, each a time-series line graph across a selectable time range, usually with a hover tooltip showing the exact value at a specific point.

CHECK THE TIME RANGE BEFORE TRUSTING A HEALTHY-LOOKING GRAPH

A dashboard defaulting to "last 6 hours" (or whatever the tool's own default happens to be) can make an incident that happened yesterday invisible — not because nothing happened, but because the currently-selected window simply

doesn't include it. A perfectly flat, healthy-looking graph is only meaningful once you've confirmed it's actually showing the period the complaint is about.

Percentiles: A Better Summary Than Average

For response-time and latency panels specifically, an average can look perfectly fine while a meaningful fraction of requests are genuinely suffering — the same underlying "summarization hides the extreme" problem as averaging over time, applied instead to averaging *across requests*.

METRIC	WHAT IT TELLS YOU
Average	A single blended figure — easily dominated by the bulk of fast, ordinary requests, hiding a smaller group of genuinely slow ones
p50 (median)	The typical request's experience — half of all requests were faster than this, half slower
p95 / p99	What the slowest 5% or 1% of requests actually experienced — often dramatically worse than the average, and exactly the group most likely to generate complaints

An average response time of 200ms sounds healthy. A p99 of 8 seconds, on the very same data, reveals that 1% of users — potentially a meaningful number of real people, depending on traffic — are having a genuinely bad experience the average alone never showed.

The Real Value: Correlating Multiple Panels at Once

Worth stating plainly, not just as a list of gotchas: a dashboard's genuine strength over one-command-at-a-time CLI tools is viewing CPU, memory, disk, and network side by side on the same time axis, and visually spotting that several of them moved together at the exact same moment — a correlation that's much harder to notice checking each metric separately, at different times, with separate commands.

Working Example: Chapter 4's Backup Job, Viewed on a Dashboard

Chapter 4's backup-job disk saturation, viewed as a dashboard instead of a live `iostat` session: the disk I/O panel shows a sustained plateau across the overnight window — visually obvious, no averaging trick needed, since sustained saturation doesn't get smoothed away the way a brief spike does. But the CPU panel for the same window, shown as an average, looks unremarkable. Switching that one panel's aggregation from `avg` to `max` reveals brief CPU spikes coinciding with the busiest moments of the backup — moments an averaged view alone would have hidden completely, exactly this chapter's own opening warning, now demonstrated against a scenario this course has already fully diagnosed by other means.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own 15-second CPU spike example, why a dashboard graph showing a "mild" 60% CPU average isn't necessarily telling the whole story.

View solution

EXERCISE 2

Explain why an average response time of 200ms and a p99 of 8 seconds can both be accurate descriptions of the exact same underlying data, and why the p99 figure matters for diagnosing complaints.

View solution

EXERCISE 3

A dashboard shows a completely flat, healthy CPU graph for a server a user says was slow yesterday. Explain what this chapter says to check before concluding CPU wasn't the cause.

View solution

CHAPTER 7 QUICK REFERENCE

- Dashboards typically show **averaged** data — a brief, severe spike can vanish into an unremarkable-looking average
- Switch the aggregation from **avg to max** on the same panel/window to check for a hidden spike
- Always confirm the **time range** actually covers the incident before trusting a flat, healthy-looking graph
- **Percentiles (p95/p99) beat averages** for latency — an average can look fine while a real, painful minority of requests suffer
- A dashboard's genuine strength: **correlating multiple panels on the same time axis** at once — harder to do one CLI command at a time
- **Next chapter:** Process-Level Diagnosis: Finding the Specific Culprit

CHAPTER 8 OF 10

Process-Level Diagnosis: Finding the Specific Culprit

System Monitoring & Performance Diagnosis

Chapter 8 · Process-Level Diagnosis: Finding the Specific Culprit

Chapters 2 through 7 established *which resource* is under pressure. This chapter is about the natural next question: *which specific process* is actually responsible. Usually straightforward — until the culprit has already finished running by the time anyone looks.

top and htop's Per-Process View

Sorting `top` by CPU (`Shift+P`) or memory (`Shift+M`) surfaces the current heaviest consumers directly. `htop` offers the same information with a friendlier, scrollable, color-coded view, including per-core usage bars.

```
$ top
top - 09:14:02 up 5 days,  2:10,  1 user,  load average: 1.20, 1.05, 0.98
Tasks: 198 total,   2 running, 196 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8.1 us,  2.0 sy,  0.0 ni, 89.5 id,  0.2 wa,  0.0 hi,  0.2 si,  0.0 st
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3312 postgres  20   0  842104  91200  18320 R   4.3   0.6   0:12.44 postgres
```

A perfectly calm-looking morning — nothing here explains a slowdown a colleague swears happened overnight.

A LIVE SNAPSHOT CAN COMPLETELY MISS A SHORT-LIVED CULPRIT

`top`'s default view shows a single moment. A process that spiked CPU heavily for a short burst and finished (or dropped back down) before anyone happened to look won't appear as a top consumer in a snapshot taken afterward — even though it was the actual cause of a preceding slowdown. This is the exact same underlying problem as Chapter 7's own averaging gotcha, applied to process attribution instead of a dashboard graph: a brief, severe event can be invisible to a check that only looks at "right now."

Sampling Over Time Instead of Trusting One Look

Two ways to avoid needing to have been watching at the exact right second:

```
# Capture a rolling snapshot log every second for a minute
$ top -b -d 1 -n 60 > top_log.txt

# sar logs historical per-interval CPU data automatically on many systems
$ sar -u -f /var/log/sysstat/sa08
02:00:01 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
02:10:01 AM     all      92.10      0.00      5.40      0.10      0.00      2.40
02:20:01 AM     all       6.20      0.00      2.10      0.05      0.00     91.65
```

`sar`'s historical logging confirms a genuine 92% CPU spike at 2:10 AM — evidence that already existed before anyone thought to go looking for it, since the system had been logging it automatically the whole time.

WINDOWS HAS BOTH EQUIVALENTS TOO

Resource Monitor's per-process view extends Task Manager's basic CPU/memory columns with granular per-process disk and network activity — the closest Windows counterpart to `iotop`'s per-process disk breakdown from Chapter 4. For historical logging over time, Performance Monitor's **Data Collector Sets** can log counters continuously to a file for later review, the same role `sar` plays on Linux.

Finding Who, Not Just When

`sar` confirms *when* the spike happened, but not by itself *which process* caused it. A `top -b` log, if one was already running, or a targeted historical process-accounting tool, fills that gap:

```
$ grep -A4 "02:10:0" top_log.txt
top - 02:10:03 up 5 days,  2:10,  0 users,  load average: 6.80, 3.20, 1.50
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 9981 postgres  20   0 1042104 210200  28320 R  88.2   1.4   0:41.02 postgres: reindex_job
```

A nightly database reindex job, running at exactly 02:10 — the same timestamp `sar` already flagged. Two independent sources of evidence, checked separately, agreeing on the same answer.

Closing the Loop on Chapter 1's Own "0 zombie" Field

Every `top` snapshot in this course, starting with Chapter 1's very first example, has quietly shown a `Tasks:` line ending in `0 zombie` — never explained until now. A **zombie** (shown as `<defunct>` in `ps`) is a process that has already finished running, but whose exit status hasn't yet been collected by its parent process. It isn't consuming meaningful CPU or memory — its actual resources are already released — but a large accumulation of zombies can exhaust the system's process table, preventing new processes from starting at all. A single zombie is rarely worth investigating on its own; a steadily growing zombie count over time points at a parent process with a genuine bug in how it manages its children.

```
$ ps aux | grep defunct
web      4021  0.0  0.0      0     0 ?        Z    03:14   0:00 [node] <defunct>
```

Working Example: The 2 AM Spike Nobody Was Watching

A dashboard (Chapter 7) shows a nightly CPU spike around 2 AM, but by the time anyone checks the following morning, a live `top` looks completely calm — exactly this chapter's own opening scenario. `sar -u` confirms the exact timestamp: 02:10 AM, 92% user CPU. Cross-referencing a `top -b` batch log already being captured by a scheduled job identifies the responsible process precisely: a nightly `postgres` reindex job. Rescheduling it to a genuinely quiet window, or spreading the work out instead of running it all at once, resolves the recurring nightly spike — found entirely through historical evidence, without anyone needing to be awake and watching at 2 AM.

Hands-On Exercises

EXERCISE 1

Explain why a colleague's live `top` check the morning after an overnight slowdown might show nothing unusual, even if a real, severe spike genuinely happened.

View solution

EXERCISE 2

Explain the difference between what `sar -u` tells you and what a `top -b` log tells you, and why this chapter uses both together in its worked example rather than just one.

View solution

EXERCISE 3

Explain what a zombie process actually is, why a single zombie usually isn't worth investigating, and what a steadily growing zombie count would suggest instead.

View solution

CHAPTER 8 QUICK REFERENCE

- `top` / `htop`, sorted by CPU or memory, surface the current heaviest consumers directly
- A **live snapshot can miss a short-lived culprit** entirely — the same underlying problem as Chapter 7's averaging gotcha, applied to processes instead of graphs
- `top -b` batch logging and `sar`'s automatic historical logging both let you look back after the fact, rather than needing to catch the moment live
- Windows equivalents: Resource Monitor's per-process view, and Performance Monitor's Data Collector Sets for historical logging
- A **zombie** process (`<defunct>`) has finished but hasn't been reaped by its parent — harmless alone, but a growing count can exhaust the process table
- **Next chapter:** Sustained vs. Transient Load: When a Spike Isn't a Problem

CHAPTER 9 OF 10

Sustained vs. Transient Load: When a Spike Isn't a Problem

System Monitoring & Performance Diagnosis

Chapter 9 · Sustained vs. Transient Load: When a Spike Isn't a Problem

Every earlier chapter has been about confirming a genuine problem. This one is the deliberate flip side — recognizing elevated resource usage that's actually normal, bursty, expected behavior, and telling it apart from a real, sustained trend that genuinely needs action. Chapter 1's own baseline theme (an 80% CPU batch server being perfectly normal) gets its fullest treatment here.

Transient Spikes: Normal, Bursty Behavior

Plenty of legitimate causes produce a real, visible spike that isn't a problem at all: application startup (populating caches, JIT warmup), a scheduled batch job, a genuine burst of real traffic (a marketing email going out, a product launch), and — in managed-memory languages — garbage collection pauses, a normal, expected part of how the runtime reclaims memory, not inherently a bug.

"NORMAL" GARBAGE COLLECTION HAS ITS OWN LIMIT

A brief GC pause is routine. A GC pause that's unusually long, or happening far more frequently than normal, can itself be a genuine symptom — often of memory pressure serious enough that the runtime is working hard to reclaim space. The line between "normal GC" and "a real problem showing up as GC activity" is exactly the same kind of baseline comparison this whole course keeps returning to.

Sustained Trends: The Kind That Actually Needs Action

A genuine sustained trend looks different from a transient spike in one specific way: it doesn't fully return to where it started. Real, sustained growth usually means one of two things — organic growth (more real users, more real data, a genuine capacity question) or a resource leak (a bug causing gradual, unbounded consumption that's never actually released, even during quiet periods when load is low).

The "Floor" Technique: Comparing Quiet Periods, Not Peaks

The single most useful technique in this chapter: instead of watching the peaks, watch the **low points** — the quiet periods, night after night, week after week. A transient spike, no matter how dramatic, returns fully to the same floor each time. A real leak or genuine growth trend leaves a slightly higher floor after every cycle, even if the peaks themselves look similar day to day.

THE FLOOR IS THE TELL, NOT THE PEAK

Two servers can show visually similar-looking daily graphs — a climb during the day, a drop overnight — and still be in completely different states. One returns to exactly the same overnight floor every single night: healthy, bursty, normal. The other's overnight floor creeps very slightly higher each night: a real, sustained problem hiding underneath what looks, at a glance, like the same repeating pattern.

A Genuine Judgment Call: When to Escalate a Trend

Recognizing a real, sustained trend is necessary, but it isn't the same as knowing what to do about it. Whether a confirmed trend needs urgent action, a scheduled fix, or genuine capacity planning depends on context this course can't fully supply on its own — how close the trend is to a hard limit, how fast it's actually climbing, and whether it represents real, wanted growth or a bug that needs fixing. Recognizing the trend is this chapter's job; deciding the response is a separate judgment call, often above what a single support engineer decides alone.

Working Example: The Nightly Restart That Was Never a Fix

A dashboard shows memory climbing to a concerning 85% by the end of every day — but a nightly scheduled restart drops it back to a comfortable-looking 40% each time, and the pattern has repeated, unremarked-on, for weeks. This is worth pausing on directly: Chapter 1 warned that a restart destroys the evidence needed to diagnose a problem. Here's the longer-term version of that same warning — a nightly restart didn't just destroy evidence once, it's been quietly masking a real problem for so long that nobody remembers it's a problem at all.

Applying the floor technique — checking available memory at the exact same point each day, right after the restart, over several weeks — tells a very different story than the daily graph alone:

WEEK	AVAILABLE MEMORY, RIGHT AFTER RESTART
3 weeks ago	9.2 GB
2 weeks ago	8.1 GB
Last week	6.9 GB
This week	5.8 GB

Even the "reset" state is degrading, week over week — a real, slow leak, one that the nightly restart schedule has been tolerating rather than fixing. Left alone, the same trend that's been climbing for weeks eventually reaches the point where a nightly restart is no longer enough to buy back the space, and the actual leak — never diagnosed, only postponed — finally causes a real incident.

Hands-On Exercises

EXERCISE 1

Explain the difference between a transient spike and a sustained trend, using this chapter's own definition of what distinguishes the two.

View solution

EXERCISE 2

Explain the "floor" technique this chapter describes, and why comparing quiet-period low points reveals a real leak that looking only at daily peaks might miss.

View solution

EXERCISE 3

Explain how this chapter's worked example connects back to Chapter 1's own warning about restarts destroying evidence, and why weeks of nightly restarts made the underlying problem harder to notice, not easier.

View solution

CHAPTER 9 QUICK REFERENCE

- Transient spikes (startup, batch jobs, real traffic bursts, GC pauses) are **normal** and resolve fully on their own
- A genuine sustained trend **doesn't fully return to where it started** — organic growth or a real leak
- The **"floor" technique**: compare quiet-period low points over time, not peaks — a healthy system returns to the same floor every time; a real problem leaves a slightly higher floor each cycle
- Recognizing a trend is this course's job; **deciding the response** (capacity planning vs. a bug fix) is a separate, context-dependent judgment call
- A recurring "fix" (like a nightly restart) can **mask** a real problem for weeks — echoing Chapter 1's own restart-destroys-evidence warning, at a longer timescale

- **Next chapter:** Capstone: Triaging Three Real Performance Tickets

CHAPTER 10 OF 10

Capstone: Triaging Three Real Performance Tickets

System Monitoring & Performance Diagnosis

Chapter 10 · Capstone — Triaging Three Real Performance Tickets

Nine chapters built the pieces — a resource-first mindset, CPU, memory, disk, when it's not local at all, dashboards, process-level attribution, and telling a real trend apart from normal noise. 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: "One API server in the pool is randomly slow"

One server out of an eight-machine pool serving the same API is intermittently slow; the other seven are fine. Started right after yesterday's deploy.

APPLYING CHAPTER 1'S SCOPING QUESTIONS

One machine (not the whole pool), sudden onset, correlating with a known event — yesterday's deploy. All four answers point toward something specific to this one machine's own current state, not a shared cause.

APPLYING CHAPTER 2'S LOAD-AVERAGE MATH

```
$ nproc
8
$ uptime
 14:02:11 up 3 days,  1:20,  1 user,  load average: 14.80, 12.10, 9.40
```

14.80 ÷ 8 cores ≈ **1.85 per core** — genuinely overloaded, not a false alarm.

APPLYING CHAPTER 8'S PROCESS-LEVEL VIEW

```
$ top -o %CPU
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 8823 app       20   0  612000  88000  15200 R  96.4   0.7  41:02.18 api-worker (v2.4.1)
```

The process from yesterday's deploy — a new retry-handling code path with a bug that spins in a tight loop under a specific error condition, pegging a full core. Rolling back the deploy restores this server to the same behavior as the other seven.

Ticket 2: "The reporting service keeps crashing without warning"

A reporting service has been crashing sporadically for weeks, with no obvious error in its own application logs. Restarting it has become routine.

APPLYING CHAPTER 3'S OOM CHECK

```
$ journalctl -k | grep -i oom
Aug 08 03:14:02 host kernel: Out of memory: Killed process 5521 (reporting-svc) total-vm:8391104kB, anon-
```

Not an unexplained crash at all — the kernel's own OOM killer, confirming memory exhaustion is the real cause.

APPLYING CHAPTERS 7 AND 9'S FLOOR TECHNIQUE

Reviewing the dashboard's own historical data, checking available memory right after each restart, week over week:

WEEK	AVAILABLE MEMORY, RIGHT AFTER RESTART
3 weeks ago	2.1 GB
2 weeks ago	1.4 GB
Last week	0.6 GB
This week	OOM-killed before ever settling

The post-restart floor has been climbing steadily worse for weeks — a genuine, slow memory leak, not a series of unrelated crashes. (A quick `df -h` also turns up several weeks of accumulated core dump files

under `/var/crash` from the repeated crashes — Chapter 5's own territory, worth cleaning up, though not the actual root cause here.)

The routine restarts were never a fix — they were exactly Chapter 1's own restart-destroys-evidence warning, repeated nightly for weeks, hiding a real leak that now needs actual profiling and a code fix.

Ticket 3: "Checkout is intermittently slow, but nothing local looks wrong"

The checkout page is slow for some users at unpredictable times. CPU, memory, and disk on the app server all look completely healthy every time anyone checks.

APPLYING CHAPTERS 2, 3, AND 4 TO RULE OUT LOCAL RESOURCES

Load average sits comfortably under 1 per core. Memory shows a healthy "available" figure with no active swapping. `iostat` shows low `%util` and low `await` — disk isn't it either.

APPLYING CHAPTER 6'S LOCAL NETWORK CHECK — A CLEAN RESULT, THIS TIME

```
$ netstat -s | grep -i retrans
    412 segments retransmitted
# ...60 seconds later...
$ netstat -s | grep -i retrans
    414 segments retransmitted
```

Almost no change — unlike Chapter 6's own worked example, packet loss isn't the answer this time. Ruling this out is itself real progress, not a dead end.

A NETWORK CHECK CHAPTER 6 DIDN'T COVER: INTERFACE THROUGHPUT

```
$ sar -n DEV 1 3
       IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s
14:02:01   eth0   9200.00   8850.00  118000.00  114500.00
```

This server's network interface is rated for roughly 1 Gbps (about 125,000 kB/s) — `118,000 kB/s` is right at the edge of what the link can physically carry. Not packet loss, but genuine bandwidth saturation on this machine's own interface. A large scheduled data-sync job, meant to run overnight, has been drifting

later each day and now overlaps with peak checkout traffic — competing for the same finite link capacity, exactly Chapter 1's own "does it correlate with a known event" question, answered by a job's schedule quietly drifting rather than a one-off deploy.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Scoping questions; the restart-destroys-evidence warning (Tickets 1, 2, 3)	Chapter 1
Load average ÷ core count (Ticket 1)	Chapter 2
The OOM killer, `journalctl -k` (Ticket 2)	Chapter 3
`iostat`'s `%util`/`await` ruling disk out (Ticket 3)	Chapter 4
`df -h` surfacing accumulated core dumps (Ticket 2) — not the primary finding here, but the same technique applies directly to a disk-space-specific ticket	Chapter 5
Local `netstat -s` check, ruled out this time; interface throughput as an extension (Ticket 3)	Chapter 6
Reviewing dashboard historical data (Ticket 2)	Chapter 7
`top -o %CPU` finding the specific responsible process (Ticket 1)	Chapter 8
The floor technique across successive weeks (Ticket 2)	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No deep dive into application-level profiling tools (flame graphs, language-specific profilers) — this course stops at "which process," not "which line of code"
- No container/Kubernetes-specific resource metrics (cgroups limits, pod-level throttling) — the underlying principles carry over, but the concrete tooling genuinely differs
- No capacity-planning methodology (forecasting, load testing) — Chapter 9 recognizes a genuine trend, but deciding how much capacity to add is a separate discipline
- No database-specific performance tuning (query plans, index design) — a real, deep topic in its own right, out of scope here
- No setting up monitoring/alerting infrastructure itself — this course reads dashboards someone else built, not building the monitoring stack

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

Hands-On Exercises

EXERCISE 1

Explain how Ticket 1's four scoping answers (one machine, sudden onset, tied to a deploy) narrowed the investigation before `top` was ever run.

View solution

EXERCISE 2

Explain why Ticket 2's routine restarts were "never a fix," and what specifically the floor technique revealed that a single dashboard glance wouldn't have.

View solution

EXERCISE 3

Explain why ruling out retransmissions in Ticket 3 was still useful progress, even though it wasn't the actual cause, and what genuinely new check found the real answer.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Ticket 1: a single overloaded server, traced to a specific buggy process from yesterday's deploy — CPU, load average, and process-level tools working together
- Ticket 2: routine restarts masking a real memory leak, only visible by comparing the post-restart floor across several weeks
- Ticket 3: local resources genuinely clean, retransmissions genuinely clean too — the real cause was network interface bandwidth saturation from a drifting scheduled job
- The recurring theme across all ten chapters: **check against a baseline, don't destroy the evidence, and let elimination — not a guess — point you at the answer**
- This closes *System Monitoring & Performance Diagnosis*, 10/10 chapters — the third complete course under the Technical Support subject, alongside *Logging & Log Analysis* and *Network Troubleshooting*