Linux Server Troubleshooting Course

Flavio Copes

Updated August 3, 2026

Contents

Diagnose Linux server failures with a repeatable method across services, logs, CPU, memory, storage, processes, networking, and recent changes.

What you'll learn: Collect evidence, isolate the first failing boundary, restore service safely, and leave a useful incident record.

Method and evidence

Define the symptom, preserve facts, build a timeline, and test one hypothesis at a time.

Define the symptom and scope

Turn a vague report into an observable failure with affected users, operations, hosts, and time boundaries.

The symptom is where every investigation starts, and most people skip it. Someone says ”the server is slow” and you start typing commands. Ten minutes later you're deep in `top` output with no idea what you're looking for.

Start by writing what fails, for whom, from where, and since when. One sentence, testable, with numbers in it.

Separate unavailable, slow, incorrect, and intermittent behavior. Each one points at different tools. ”Slow” needs timing. ”Unavailable” needs a connection test. ”Incorrect” needs a request and its wrong response. ”Intermittent” needs a counter, because you'll have to catch it in the act.

Turn the report into a measurement

”The server is slow” becomes measurable with one command:

```
curl -sS -o /dev/null -w 'status=%{http_code} total=%{time_total}s\n' https://app.example.com
```

```
status=200 total=4.812s
```

Now you have a fact. This endpoint answers with 200 but takes 4.8 seconds. Run it a few times, and run it against a healthy path too:

```
curl -sS -o /dev/null -w 'status=%{http_code} total=%{time_total}s\n' https://app.example.com
# status=200 total=0.031s
```

Comparing one affected path with one healthy path narrows the scope immediately. Here the host, TLS, and web server are fine. Something specific to `/api/orders` is slow.

Write the statement

Record exact errors and request identifiers while they're fresh. Then write the symptom in this shape:

```
Operation: GET /api/orders from the office network
Expected: 200 in under 300ms
Measured: 200 in 4.8s, every request since 09:42 UTC
Scope: all users; /health unaffected
First seen: 09:42 UTC (first Slack report 09:55)
```

Everything after this point tests that statement. If a command you're about to run can't confirm or narrow it, don't run it yet.

One trap to avoid: taking the report's timestamp as the start time. Users report incidents late. Check your logs or metrics for when the numbers actually changed, because ”what changed at 09:42” is the question that usually solves the case, and asking it about 09:55 sends you to the wrong deploy.

Preserve state before changing it

Capture volatile process, service, resource, network, and log evidence before a restart destroys useful clues.

A restart can restore service and erase the state that explained the failure. The stuck process, its open files, the socket backlog, the memory numbers: all gone the moment you type `systemctl restart`.

So before you change anything, spend two minutes capturing what's there. You're allowed to restart afterwards. You're just not allowed to restart first.

The capture

Save everything into a timestamped directory so it doesn't get mixed up with the next incident:

```
d=/var/tmp/incident-$(date -u +%Y%m%dT%H%M%S)
mkdir "$d" && cd "$d"

date -u > time.txt
uptime > uptime.txt
systemctl --failed > failed-units.txt
ps auxww --sort=-%cpu > processes.txt
free -h > memory.txt
df -h > disk.txt
ss -tnp state established > sockets.txt
journalctl -u app.service --since "-30 min" --no-pager > app-log.txt
dmesg --ctime | tail -100 > kernel.txt
```

Each file answers one later question. `processes.txt` tells you what was running and in which state. `sockets.txt` tells you who was connected. `failed-units.txt` tells you whether the problem was wider than one service.

Record time, uptime, recent changes, failed units, processes, load, memory, storage, sockets, and focused logs. That list covers almost every ”I wish I knew what it looked like” moment I've had.

Verify the capture before acting

Check the directory actually has content:

```
wc -l *.txt
# 42 processes.txt
# 118 sockets.txt
# ...
```

An empty file here means a command failed silently, and you want to know now, not during the review.

Avoid giant uncontrolled dumps containing secrets or unrelated user data. A full `journalctl` export or a core dump of a process that handles passwords is a liability. Capture focused slices, and note which files might contain sensitive values before you share them in a ticket.

The trap: capturing evidence *about the wrong thing*. If the symptom is network timeouts and you only save CPU and memory output, the capture was theater. Match the capture to the symptom you defined in the previous lesson.

Create a ten-command evidence checklist for your own server. Mark which outputs are volatile (lost on restart) and which might contain sensitive information. When an incident hits, you paste it, not compose it.

Build a timeline

Correlate deployment, configuration, kernel, service, monitoring, and user events using one time reference.

Most incidents become easier when events are placed in order. "The service crashed" is a mystery. "The deploy finished at 14:02, memory climbed from 14:05, the OOM killer fired at 14:31" is almost a conclusion.

Confirm the clock first

A timeline built on a drifting clock is worse than no timeline. Confirm the server clock and timezone before you trust a single timestamp:

```
timedatectl
```

```
               Local time: Mon 2026-08-03 14:40:12 UTC
           Universal time: Mon 2026-08-03 14:40:12 UTC
                 Time zone: Etc/UTC (UTC, +0000)
System clock synchronized: yes
              NTP service: active
```

The two lines that matter: `System clock synchronized: yes` and the time zone. If synchronization is `no`, note it in the timeline itself, because every local timestamp is now suspect.

Pull events with absolute timestamps

Use absolute timestamps everywhere. "5 minutes ago" is useless in a document you'll read tomorrow.

```
journalctl -u app.service --since "2026-08-03 13:50" --until "2026-08-03 14:35" -o short-iso
```

```
2026-08-03T14:02:11+0000 web1 systemd[1]: Started app.service.
2026-08-03T14:31:04+0000 web1 kernel: Out of memory: Killed process 3721 (node)
```

The `-o short-iso` flag gives you sortable timestamps you can paste next to events from other systems. If the machine rebooted during the incident, add `-b -1` to read the previous boot's journal — the messages right before a crash are usually the interesting ones.

Deployment records come from your CI system, provider events from the cloud status page or maintenance emails, and user reports from wherever they landed. Each source gets its own line in the timeline, with its origin noted, so you can weigh it later.

Now align journal entries, deployment records, provider events, and client reports in one list:

```
13:58 UTC  deploy pipeline started (CI log)          - evidence
14:02 UTC  app.service restarted (journal)           - evidence
14:05 UTC  RSS starts climbing (metrics)             - evidence
14:31 UTC  OOM kill of node PID 3721 (kernel log)    - evidence
14:33 UTC  first user report (Slack)                 - evidence
           new release leaks memory                  - inference, needs verification
```

Mark which line is evidence and which is an inference. Correlation suggests a hypothesis but does

not prove causation: the deploy is the obvious suspect, but a traffic spike at 14:00 would fit the same timeline.

The trap is mixing time zones. CI logs in local time, journals in UTC, a teammate quoting their own wall clock. One event lands an hour off, and the timeline "proves" the crash happened before its cause. Convert everything to UTC as you write each line down.

Test one hypothesis at a time

Choose the smallest test that could disprove an explanation before changing several layers together.

A hypothesis should predict an observable result. If it cannot be tested, it does not yet guide the investigation. "Maybe it's the network" predicts nothing. "The app can't reach the database, so a TCP connect to port 5432 from this host will fail" predicts something you can check in five seconds.

Make the prediction, then run the smallest test

Say the symptom is API requests timing out. Hypothesis one: the database is unreachable from the app host. The prediction: a plain TCP connection will hang or be refused.

```
timeout 3 bash -c 'cat < /dev/null > /dev/tcp/10.0.2.15/5432' && echo open || echo closed-or-
```

```
open
```

The connection opens. Hypothesis rejected, in one read-only command, without touching the app. That's the shape you want: cheap, reversible, and capable of *disproving* the idea.

Hypothesis two: the database accepts connections but answers slowly. Prediction: a trivial query will take much longer than usual.

```
time psql -h 10.0.2.15 -U app -d shop -c 'SELECT 1;'
# real    0m2.984s
```

Three seconds for `SELECT 1` is a real finding. Now you know where to dig, and you got there by rejecting one explanation at a time.

Rank before you test

You'll usually have more hypotheses than time. Rank explanations by evidence, impact, and test cost. Test the ones that are cheap to check and would explain the most, even if they feel less likely. A ten-second `df -h` is worth running before a two-hour packet capture, whatever your gut says.

Prefer reversible diagnostics first. Reading logs, timing queries, and probing ports change nothing. Restarting services and editing configs change the system, and belong later.

When a test *does* require a change, change one variable, record the result, and return to the previous state when the test fails. If you bump a timeout, disable a cron job, and restart the service together, and things improve, you've learned almost nothing and you can't cleanly undo it.

That's the trap: the shotgun fix. Three changes at once that "solve" the incident tonight and leave you unable to say which one mattered, which one was harmless, and which one is a new bug waiting for traffic. Take one symptom from your own system, write three hypotheses, and for each define one read-only test and the result that would make you reject it.

Check your understanding: method and evidence

Check symptom definition, scope, timelines, evidence preservation, hypotheses, baselines, and safe changes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Services and processes

Diagnose failed units, process state, signals, file descriptors, and application logs.

Diagnose a failed unit

Read systemd result, exit status, effective configuration, and journal before repeatedly restarting a service.

`systemctl status` gives a useful summary, but the full explanation usually needs more evidence.

Inspect `Result`, `ExecMainStatus`, the unit journal, and `systemctl cat`. Check referenced users, paths, files, and ports. Validate application configuration with its own check command.

Break a disposable service through one invalid path. Capture the manager result and journal, repair it, and prove the service stays healthy after restart.

Capture state before restarting the service:

```
systemctl --failed
systemctl status app.service --no-pager
journalctl -u app.service -b -n 100 --no-pager
```

Find when the failure began and what changed just before it. Check the process exit status, user, paths, ports, and dependencies. Restart only after you have enough evidence to test one explanation, then verify the user-facing endpoint.

Read process state and relationships

Inspect PID, parent, user, state, start time, CPU time, memory, and command without trusting a process name alone.

A process list is a snapshot. Read identity and relationships before sending signals or attaching tools. Half of all "I killed it and it came back" and "I killed it and something else died" stories start with someone acting on a process name alone.

Ask ps for exactly what you need

Use `ps` with explicit columns instead of scanning the default output:

```
ps -o pid,ppid,user,stat,etime,nlwp,rss,cmd -C nginx
```

```
    PID    PPID USER     STAT     ELAPSED NLWP   RSS CMD
   1298       1 root     Ss      12-03:11:42    1  2140 nginx: master process /usr/sbin/nginx
   1299    1298 www-data S       12-03:11:42    1  4820 nginx: worker process
   1300    1298 www-data S       12-03:11:42    1  4716 nginx: worker process
```

The fields that matter: `PPID` tells you who owns this process (PPID 1 usually means systemd started it). `STAT` is the state. `ELAPSED` tells you if it restarted recently — a service that's been "up" for 40 seconds after a 12-day incident has been crash-looping. `NLWP` is the thread count, and `RSS` is resident memory.

Read the state letter

The first character of `STAT` suggests your next question. `R` is running or runnable. `S` is sleeping, waiting for an event — normal for most of a server's life. `D` is **uninterruptible sleep**, almost always waiting on I/O; a pile of `D` processes points at storage or NFS, not at the application. `Z` is a **zombie**: already dead, waiting for its parent to collect the exit status. `T` is stopped, often by a forgotten `Ctrl-Z` or a debugger.

See the family tree

Use `pstree` for parent-child structure:

```
pstree -p 1298
```

```
nginx(1298)  nginx(1299)
             nginx(1300)
```

And confirm which systemd unit owns the process, because that's who will react when you touch it:

```
systemctl status 1298 --no-pager | head -2
#   nginx.service - A high performance web server
```

The trap: killing a zombie. A `Z` process is already dead; the PID in your `ps` output holds no memory and runs no code. Signaling it does nothing. The bug is in its *parent*, which never called `wait()`. Fix or restart the parent, and the zombie disappears.

Choose one service process on your machine and record its parent, user, state, elapsed time, threads, children, and systemd unit. That's the identity card you want before the next lesson, where we start sending signals.

Use signals deliberately

Request reload or termination through the service manager and reserve SIGKILL for processes that cannot clean up.

Signals ask a process to perform an action. They do not all mean "stop now." Sending the wrong one, or sending the right one to the wrong PID, turns a small problem into a bigger one.

The three you'll actually use:

SIGTERM (`kill PID`, the default) allows normal shutdown. The process can finish requests, flush buffers, and remove its lock files. This is always your first choice.

SIGHUP often requests reload — re-read config without dropping connections — but behavior is application-specific. nginx reloads on it. A process with no handler just dies. Check the application docs before assuming.

SIGKILL (`kill -9`) cannot be handled and prevents cleanup. The kernel removes the process immediately. No flush, no lock removal, no goodbye to clients. It's the last resort for a process that ignored SIGTERM, not the first move.

Prefer the service manager

Use `systemctl` when systemd owns the process. It signals the right PIDs, in the right order, and tracks the result:

```
sudo systemctl reload nginx     # sends the unit's configured reload signal
sudo systemctl stop myapp       # SIGTERM, waits, then escalates for you
```

To see what a stop will do before you do it:

```
systemctl show myapp -p ExecMainPID -p KillSignal -p TimeoutStopUSec
```

```
ExecMainPID=2143
KillSignal=15
TimeoutStopUSec=1min 30s
```

That reads: SIGTERM (signal 15) goes to the service, and if it's still alive after 90 seconds, systemd escalates to SIGKILL itself. You rarely need to run `kill -9` by hand on a managed service.

If you must signal directly, verify the PID first:

```
ps -o pid,ppid,user,cmd -p 2143
kill 2143          # SIGTERM
kill -0 2143 && echo still-running || echo gone
```

`kill -0` sends no signal at all — it just checks the process still exists, which makes it a clean way to verify the stop worked.

The trap: killing a PID you copied from an old terminal or a stale log. PIDs get reused. The worker you meant to kill exited an hour ago, and 2143 now belongs to a backup job mid-write. Always re-check what a PID is *right now* before signaling it. And on systemd services, remember that killing the main process by hand often just triggers `Restart=always` — you didn't stop the service, you restarted it and confused the journal.

Find file descriptor problems

Inspect open files, sockets, limits, leaks, and deleted files when a process reports too many open files.

Every process uses file descriptors for files, sockets, pipes, and other kernel objects. When a process runs out, you see this in the logs:

```
Error: EMFILE: too many open files, open '/var/lib/app/cache/items.json'
```

The symptom looks like a file problem. It's really an accounting problem: the process hit its descriptor limit.

Count and compare

Get the service's main PID and count its open descriptors:

```
systemctl show myapp -p MainPID
# MainPID=2143

ls /proc/2143/fd | wc -l
# 3987
```

Then check the limit that count is racing toward:

```
grep 'Max open files' /proc/2143/limits
# Max open files    4096    4096    files
```

3987 of 4096. This process is about to fail, and now you know why.

A single count doesn't tell you whether it's a leak or just a busy service. Compare descriptor counts over time: measure twice, minutes apart, under known activity. A steady 3900 on a server that handles thousands of connections may be legitimate sizing. A count that climbs and never comes down is a leak.

Group by type to find the leak

Inspect `/proc/PID/fd` or use `lsof` and group what you find:

```
sudo lsof -p 2143 | awk '{print $5}' | sort | uniq -c | sort -rn | head
```

```
   3418 sock
    521 REG
     14 CHR
      8 FIFO
```

3418 sockets tells a very different story than 3418 regular files. Sockets piling up usually means connections opened and never closed — often to a database or an upstream API. Regular files piling up means file handles are opened without being closed, and `lsof` will show you which paths.

Fix the right thing

If it's a genuine capacity issue, raise the limit where systemd will actually apply it:

```
# systemctl edit myapp
[Service]
LimitNOFILE=16384
```

Then `systemctl daemon-reload && systemctl restart myapp` and re-check `/proc/PID/limits`.

The trap: raising a limit can delay a leak without fixing it. If descriptors grow without bound, 16384 just moves the crash from Tuesday to Friday, with four times more open sockets to clean up. Identify the descriptor type and owner first, fix the code path or connection pool that never closes, and treat the limit bump as breathing room, not the repair.

Check your understanding: services and processes

Check failed units, exit status, process states, signals, file descriptors, logs, and process ownership.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

CPU, memory, and storage

Interpret load, CPU pressure, memory, swap, I/O, filesystem, and kernel evidence.

Interpret load and CPU

Separate runnable demand, blocked work, CPU utilization, per-core saturation, and one busy process.

Load average is not a CPU percentage. It counts runnable tasks and tasks in uninterruptible sleep, averaged over 1, 5, and 15 minutes. That last part is why Linux load numbers confuse people: a machine doing zero CPU work can show a load of 20 if twenty processes are stuck waiting on a dead disk.

Read load against the core count

```
uptime
# 15:04:33 up 12 days,  3:12,  1 user,  load average: 6.42, 5.90, 2.15
nproc
# 4
```

Compare load with CPU count. Load 6.4 on 4 cores means demand exceeds capacity: on average, 2+ tasks are waiting at any moment. The three numbers also give you direction — here the 15-minute average is 2.15, so this started recently and is getting worse.

Find out what kind of demand it is

`top` splits CPU time into categories on its `%Cpu(s)` line:

```
%Cpu(s):  8.1 us,  2.3 sy,  0.0 ni, 41.2 id, 48.0 wa,  0.0 hi,  0.4 si
```

The fields that matter: `us` is user code, `sy` is kernel work, `id` is idle, and `wa` is **I/O wait** — CPU sitting idle while tasks wait for storage. This sample is the classic pattern: load is high, but 41% of CPU is idle and 48% is `wa`. The bottleneck is the disk, not the processor. Chasing a "CPU problem" here wastes the afternoon.

Then check per-core numbers, because averages hide saturation:

```
mpstat -P ALL 5 1
```

```
CPU    %usr   %sys  %iowait   %idle
all   26.10   3.02     0.51   70.37
  0   99.20   0.60     0.00    0.20
  1    2.10   3.80     0.70   93.40
```

One saturated core can limit a single-threaded application while total CPU still looks moderate — 26% overall, but core 0 is pinned at 99% and that's exactly where your Node process lives. `pidstat 5` (or pressing `1` inside `top`) tells you which process owns the busy core.

Capture five timed samples during a controlled load rather than staring at one refresh. A single snapshot catches noise; five samples show whether demand is user CPU, system CPU, I/O wait, or one constrained process.

The trap is the one this lesson opened with: high load with idle CPUs. Load counts `D`-state tasks too, so slow storage or a hung NFS mount inflates it. When load looks scary, check `wa` and process states before you blame the CPU.

Diagnose memory pressure

Read available memory, cache, swap activity, per-process use, and OOM evidence without treating used RAM as failure.

Linux uses unused memory for cache and can reclaim much of it. That's why almost every "the server is out of RAM" report from someone reading the `used` column is a false alarm. Focus on pressure and behavior, not the used column alone.

Read free correctly

```
free -h
```

```
               total        used        free      shared  buff/cache   available
Mem:           7.8Gi       5.1Gi       210Mi       120Mi       2.5Gi       2.3Gi
Swap:          2.0Gi       1.4Gi       600Mi
```

The column that matters is **available**: the kernel's estimate of memory that can be handed to new work, including reclaimable cache. Here `free` is a scary 210Mi, but `available` is 2.3Gi. Healthy. The number that *should* worry you in this output is swap: 1.4Gi used means the system was under real pressure at some point.

Check whether pressure is happening now

Swap *used* is history. Swap *activity* is the present. Watch `vmstat`:

```
vmstat 5 3
```

```
procs -----------memory---------- ---swap-- ...
 r  b   swpd   free   buff  cache   si   so
 2  0 1468k 21540  84120 2519k     0    0
 3  1 1471k  98212  84120 2380k   840  1204
```

The `si` and `so` columns are pages swapped in and out per second. Zeros mean the old swap usage is stale. Repeated swap-in and swap-out — sustained nonzero `si`/`so` — means the working set doesn't fit and the system is thrashing right now.

Then find who's holding the memory, by process RSS:

```
ps -eo pid,user,rss,cmd --sort=-rss | head -4
# 3721 app   4183212 node /srv/app/server.js
```

If the service runs in a cgroup (any systemd unit with `MemoryMax`), check its limit too — a container can be OOM-killed while the host has gigabytes free.

When the OOM killer fires

The OOM killer records which process it selected and why, in the kernel log:

```
journalctl -k | grep -i 'out of memory'
# kernel: Out of memory: Killed process 3721 (node) total-vm:5124404kB, anon-rss:4183212kB
```

The trap: assuming the killed process was the guilty one. The OOM killer picks a victim by score, which favors big processes — but a small, fast leaker can push the system over while the database, being the largest RSS, takes the bullet. Collect memory evidence before and during a controlled workload, and decide from the trend whether the system is healthy, reclaiming cache, swapping, or killing work.

Diagnose storage latency

Separate filesystem capacity from block-device latency, queueing, throughput, and application sync behavior.

A filesystem can have free space while storage latency makes every request slow. `df -h` says 60% free, the app times out anyway, and everyone is confused. Capacity and speed are different problems.

The symptom pattern: requests slow down across the board, load average climbs, but CPU sits mostly idle with high I/O wait. That combination says "storage," and the next job is finding which device and which process.

Measure the device

```
iostat -xz 5 3
```

```
Device   r/s    w/s   rkB/s   wkB/s  r_await  w_await  aqu-sz  %util
sda     2.1   312.4    84.2  48120.5     3.1    187.4    18.2   99.6
```

Skip the first sample — it's the average since boot, not the current state. Then read three fields. `r_await` and `w_await` are the average milliseconds each read or write waits, queue time included: single-digit values are fine for SSDs, and 187ms per write is a disaster. `aqu-sz` is the average queue depth — 18 requests waiting means work arrives much faster than the device completes it. `%util` near 100 says the device was busy the whole interval.

Map the busy device to what's mounted on it:

```
findmnt -o TARGET,SOURCE,FSTYPE /dev/sda1
# /var  /dev/sda1  ext4
```

Find the process creating the work

I/O wait needs context: it can coexist with idle CPUs and does not identify the responsible process alone. `pidstat -d` does:

```
pidstat -d 5 2
```

```
      UID       PID   kB_rd/s   kB_wr/s  Command
      998      4110      0.00  46210.30  pg_dump
     1001      2143     12.40     88.10  node
```

There's the answer: a backup job writing 46 MB/s to the same disk the application lives on. The fix isn't faster hardware, it's moving the backup window, throttling it with `ionice`, or pointing it at a different volume.

Confirm from the application side too — database slow-query logs or request timing should improve the moment the writer stops. Capture device samples during a known read or write, and match the busy device to its mount and the process creating work.

The trap is trusting `%util` on SSDs and NVMe. Those devices serve many requests in parallel, so "100% busy" can still mean plenty of spare capacity. On modern flash, judge saturation by `await` climbing and `aqu-sz` growing, not by the utilization column.

Read filesystem and kernel evidence

Check space, inodes, mounts, read-only remounts, device errors, and previous-boot messages after storage failures.

Write failures can come from full blocks, full inodes, permissions, quotas, read-only filesystems, or device errors. They often share one misleading error message, so the job is walking the possibilities in order instead of guessing.

The symptom: an application logs `No space left on device` or `Read-only file system` and stops writing.

Blocks, then inodes

```
df -h /var/lib/app
# Filesystem      Size  Used Avail Use% Mounted on
# /dev/sda1        50G   31G   17G  65% /var
```

65% used, plenty of space — and yet the write fails. Check inodes, because `No space left on device` also fires when the filesystem runs out of them:

```
df -i /var/lib/app
# Filesystem      Inodes  IUsed  IFree IUse% Mounted on
# /dev/sda1      3276800 3276800     0  100% /var
```

There it is. Millions of tiny files — often a cache, session store, or mail queue — consumed every inode while using a third of the blocks. The fix is deleting or consolidating the small files, not adding disk.

Confirm you're looking at the right filesystem, too. `findmnt /var/lib/app` shows which mount actually receives the write; a path that *looks* like it lives on the big data volume may sit on the small root filesystem because a mount failed at boot.

Space that won't come back

If `df` says full but `du` can't find the data, a process is holding deleted files open. The kernel frees those blocks only when the last descriptor closes:

```
sudo lsof +L1 /var
# COMMAND  PID  USER  FD  SIZE/OFF NLINK NAME
# app     2143  app   4w  18734211072    0 /var/log/app/debug.log (deleted)
```

18GB in a deleted log. Restart the writer (or make it reopen its logs) and the space returns.

That's also the trap in this lesson: ”clearing” a live log with `rm`. The file vanishes from the directory, the process keeps writing to it, and the disk stays full while you can no longer see why. Truncate instead: `truncate -s 0 /var/log/app/debug.log` frees the space and keeps the descriptor valid.

When the kernel took the filesystem away

`Read-only file system` on a mount that should be writable usually means the kernel saw device errors and remounted it read-only to protect the data. Check the kernel log with `journalctl -k` or `dmesg` for lines like `I/O error, dev sda` or `EXT4-fs error`. If the kernel remounted a filesystem read-only, preserve data and investigate the storage path before forcing it writable — remounting `rw` over a failing disk converts a warning into corruption.

Check your understanding: cpu, memory, and storage

Check load, CPU saturation, memory pressure, OOM evidence, I/O wait, storage latency, capacity, and kernel messages.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Network and time

Test names, routes, sockets, firewalls, packets, TLS, and clock synchronization.

Test the network in layers

Check name resolution, selected address, route, transport port, TLS, and application response as separate boundaries.

"The network is down" combines several systems that can fail independently.

Resolve with `dig`, inspect `ip route get`, test the port with `nc`, and inspect the application with `curl -v`. Stop at the first failing boundary instead of reconfiguring every layer.

Trace one HTTPS request from the server. Write a single sentence naming the first failure or confirming every tested boundary.

Use one command for each boundary:

```
ip route get 1.1.1.1
dig +short app.example.com
nc -vz app.example.com 443
curl -v https://app.example.com/health
```

Stop at the first failing layer. A correct DNS answer does not prove the port accepts connections, and an open port does not prove TLS or HTTP works. Save timestamps and exact errors before changing firewall, DNS, certificates, or application configuration.

Inspect listeners and firewalls

Connect a server process and bind address to host and cloud firewall policy for one exact flow.

A running service is reachable only if it listens on the intended address and every policy boundary allows the flow. "The service is up but I can't connect" almost always breaks at one of three points: the bind address, the host firewall, or the provider firewall. Check them in that order.

What is actually listening

Use `ss -lntup` to identify the process, protocol, address, and port:

```
sudo ss -lntup
```

```
Netid State  Local Address:Port  Process
tcp   LISTEN     127.0.0.1:3000  users:(("node",pid=2143,fd=18))
tcp   LISTEN       0.0.0.0:22    users:(("sshd",pid=812,fd=3))
tcp   LISTEN          [::]:443   users:(("nginx",pid=1298,fd=6))
```

The **Local Address** column decides everything. `0.0.0.0` (or `[::]` for IPv6) means all interfaces — reachable from outside if firewalls allow it. `127.0.0.1` means loopback only. That `node` process on port 3000 will never answer a remote client, no matter what you do to the firewall. Loopback

listeners are intentionally unavailable from remote clients; that's often correct (an app behind nginx *should* bind to loopback), but it's the first thing to rule out.

If the port doesn't appear at all, the service isn't the network's problem — it failed to bind, and the reason is in its journal.

Walk the policy boundaries

Check host firewall and provider firewall rules separately, because either can silently drop the flow:

```
sudo nft list ruleset | grep -A2 'dport 443'   # or: sudo ufw status
```

Then the cloud layer — security group, VPC firewall, or provider "networking" tab. Nothing on the host will show you a packet the provider dropped before it arrived. A capture proves it: run `sudo tcpdump -ni any port 443` while a remote client connects. No SYN arriving means the block is upstream of the machine.

Prove it from a real client

Finish with a test from where users actually connect:

```
nc -vz -w 3 203.0.113.40 443
# Connection to 203.0.113.40 443 port [tcp/https] succeeded!
```

A timeout points at a filter along the path. An immediate `refused` means the packet arrived and nothing was listening on that address.

The trap: testing from the server itself. `curl localhost:3000` succeeds over loopback and proves nothing about remote reachability — this is exactly how a loopback-only bind hides for hours. Choose one server port and record its listener, bind address, host rule, cloud rule, and a remote connection test.

Capture a focused packet trace

Use an authorized narrow capture to distinguish missing traffic, retransmission, reset, handshake, and application behavior.

A packet capture is the ground truth for one question: did the bytes actually arrive? Logs can lie by omission and metrics average things away, but a packet either crossed the interface or it didn't. Reach for a capture when logs on both ends disagree, or when connections fail with no error on either side.

One rule before anything else: only capture traffic on systems you're authorized to inspect, and keep the capture narrow. Traces can contain credentials and user data.

Capture with a filter, always

Filter by host, port, and protocol so you record one conversation, not the whole interface:

```
sudo tcpdump -ni any host 203.0.113.40 and port 443 -c 200
```

`-n` skips DNS lookups (they're slow and add noise), `any` covers all interfaces, and `-c 200` stops after 200 packets so a forgotten capture can't fill the disk.

Read the handshake

A healthy TCP connection opens with three packets:

```
10.0.2.5.51844 > 203.0.113.40.443: Flags [S], seq 1290481, ...
203.0.113.40.443 > 10.0.2.5.51844: Flags [S.], seq 884211, ack 1290482, ...
10.0.2.5.51844 > 203.0.113.40.443: Flags [.], ack 1, ...
```

The `Flags` field carries the diagnosis. `[S]` is the client's SYN. `[S.]` is SYN-ACK — the server answered. `[R]` is a reset: something actively refused or tore down the connection, which points at the application or an in-path device, not a silent drop. Repeated `[S]` with no reply means packets leave and nothing comes back — a filter or routing problem. Lines marked `retransmission` mean packets are being lost mid-conversation. After the handshake, look for TLS alerts and actual application bytes; `length 0` keepalives aren't data.

Remember what a capture can't tell you

A packet capture shows what crossed one interface, not what happened everywhere. A missing SYN here means it didn't *arrive here* — it may have left the client fine and died at a firewall in between. Capture on both sides when a middle boundary is uncertain, and save to files you can compare:

```
sudo tcpdump -ni any host 203.0.113.40 and port 443 -w /var/tmp/server-side.pcap -c 500
```

The trap: concluding ”the client never sent it” from a server-side capture alone. Absence of evidence on one interface is not absence of the packet. Capture one connection you own with `tcpdump -ni any host ADDRESS and port PORT`, and match packets to the client and server logs by timestamp — that three-way comparison settles most arguments about whose side is broken.

Check time, TLS, and certificates

Recognize how clock drift, certificate names, chains, validity, and renewal failures break otherwise healthy connections.

TLS validation depends on the requested name, a trusted chain, and current time. Break any of the three and connections fail with errors like `certificate verify failed` or `certificate has expired` — while ping, DNS, and TCP all look perfectly healthy. A working TCP handshake does not prove TLS or HTTP is correct.

Check the clock before the certificate

Certificate validity is checked against the local clock, so start with `timedatectl`:

```
timedatectl
# System clock synchronized: yes
#               NTP service: active
```

If the clock is hours or days off, perfectly valid certificates look ”expired” or ”not yet valid” on this machine only. That's the classic signature of clock drift: TLS fails *here* while every other client is fine. Fix time synchronization, not the certificate.

Inspect what the server presents

```
openssl s_client -connect app.example.com:443 -servername app.example.com </dev/null 2>/dev/n
```

```
subject=CN=app.example.com
notAfter=Oct 12 09:31:00 2026 GMT
X509v3 Subject Alternative Name:
    DNS:app.example.com, DNS:www.example.com
```

Three things to verify in that output. The **subjectAltName** list must contain the exact name the client requested — clients match against SAN entries, not the CN. The `notAfter` date must be in the future. And the chain must verify: run the same command without the pipe and look for `Verify return code: 0 (ok)` near the end. Code 21 (`unable to verify the first certificate`) usually means the server sends its own certificate but not the intermediate — browsers may paper over it, strict clients won't.

The `-servername` flag matters: it sets SNI, and a server hosting several sites returns different certificates for different names. Testing without it can show you a certificate no real client ever sees.

Separate the layers in one pass

`curl -v https://app.example.com/` shows DNS resolution, TCP connect, the negotiated TLS version, certificate acceptance, and the HTTP status in order. Record DNS address, TCP success, negotiated TLS, certificate name, expiry, and HTTP status separately — knowing which line failed tells you which team or config file to open.

The trap: "fixing" a verification error with `curl -k` or by disabling verification in the app. That doesn't fix anything, it removes the protection that was correctly telling you something is wrong, and it has a habit of getting committed. Diagnose with it if you must, never ship it. Renewal failures are the usual real cause — check the expiry date first, and check whether the renewal job (certbot timer or similar) has been failing quietly for weeks.

Check your understanding: network and time

Check DNS, addresses, routes, listeners, firewalls, packet capture, TLS, HTTP, and clock synchronization.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Recover and prevent

Roll back safely, verify recovery, improve observability, and write a useful review.

Choose mitigation before root cause

Restore safe service quickly when impact is high while preserving enough evidence for later analysis.

The fastest safe mitigation and the complete root cause are often different tasks. When users are down, your first job is stopping the bleeding, not explaining it. Understanding why the new release leaks memory can happen tomorrow; rolling back to the release that didn't takes five minutes tonight.

The standard mitigations

You have four moves, roughly in order of preference. **Rollback**: return to the last known-good version or config. **Fail over**: move traffic to a healthy replica or region. **Shed load**: rate-limit or turn away some traffic so the rest succeeds. **Disable one feature**: flag off the code path that's hurting, keep everything else running.

Pick based on evidence, not reflex. If the timeline says the incident started with the 14:02 deploy, rollback is justified. If nothing was deployed and one host is sick, failover fits better. Rollback, fail over, shed load, or disable one feature when evidence supports it — a mitigation chosen at random is just another uncontrolled change.

Preserve, then mitigate

Record the state first. The mitigation will destroy the evidence, so spend the two minutes from the earlier lesson before you pull the trigger:

```
d=/var/tmp/incident-$(date -u +%Y%m%dT%H%M%S); mkdir "$d"
ps auxww > "$d/ps.txt"; free -h > "$d/mem.txt"; ss -tnp > "$d/ss.txt"
journalctl -u app.service --since "-20 min" --no-pager > "$d/journal.txt"
```

Then mitigate, and define the exit check *before* you act:

```
systemctl revert app.service && systemctl restart app.service
watch -n 10 'curl -sS -o /dev/null -w "%{http_code} %{time_total}s\n" https://app.example.com
```

The user-visible check proves impact ended. Internal green lights don't count; the request that was failing has to succeed. Decide in advance what result means "the mitigation worked" and what result triggers the next option, so you're not inventing criteria under pressure.

The trap: the risky workaround. Under pressure, "just open the port to everyone", "disable auth temporarily", or "chmod 777 the directory" all feel like mitigations. They're not — they trade an availability incident for a security or data-integrity incident, which is a much worse trade. Avoid a risky workaround that creates a larger security or data-integrity problem; a legitimate mitigation reduces total risk, and if your idea only moves the risk somewhere darker, keep looking.

For a simulated bad deployment, write down your mitigation, the rollback trigger, the evidence to preserve, and the user-visible check that proves impact ended.

Verify recovery from the outside

Confirm the original user operation, dependencies, data integrity, error rate, and recurrence window after a repair.

A green process status proves only that a process is running. `systemctl status` says `active (running)` while the app returns 500 on every request — a process can be alive and broken at the same time. Declaring victory from the inside is how incidents get "resolved" twice in one evening.

Repeat the operation that failed

Repeat the exact failed operation from the relevant client path. Not a simpler one, not the health endpoint — the request users were making when things broke, from where they make it:

```
curl -sS -w '\nstatus=%{http_code} total=%{time_total}s\n' https://app.example.com/api/orders
```

```
[{"id":18234,"status":"shipped"},...]
status=200 total=0.214s
```

Both numbers matter. A 200 in 4 seconds means the outage became a slowdown, and users still feel it. If the incident affected external users, run the check from outside your network — an internal test can't see a broken load balancer, CDN, or DNS record.

Check what a request can't show

One good response is a start, not proof. Check logs, latency, errors, queues, and data:

```
journalctl -u app.service --since "-5 min" --no-pager | grep -ci error
# 0
```

Then look at whatever accumulated during the outage. A message queue that grew to 40,000 jobs will keep the system degraded for an hour after the "fix". Writes that failed silently leave data gaps that no status page reveals — spot-check records created during the incident window.

Watch past the trigger

Watch long enough to cover the trigger that caused the incident. If the crash came from the nightly backup, a service that survives ten quiet minutes at 3pm proves nothing; recovery is confirmed after tonight's backup runs clean. If load triggered it, watch through the next peak:

```
watch -n 30 'curl -sS -o /dev/null -w "%{http_code} %{time_total}s\n" https://app.example.com
```

The trap: verifying with a different, easier operation. The health check hits `/health`, which does almost nothing, so it recovered instantly — while `/api/orders`, which touches the database that's still rebuilding, stays broken. You close the incident, users reopen it. Always verify with the operation from the original symptom definition.

Write a recovery checklist for one service you run. Include an external request, one data check, one dependency check, and a monitoring window tied to the trigger.

Add useful observability

Turn missing evidence into bounded logs, metrics, traces, health checks, and alerts tied to user impact.

Observability should help answer a concrete operational question, not collect every possible value. The test for any new log line, metric, or alert is: which question, during which future incident, does this answer? If you can't name the question, you're collecting noise you'll pay to store.

The best source of questions is your last incident. Every time you thought "I wish I knew what the memory looked like before the restart" or "which request started this?", that's a missing signal.

Make logs answerable

Add request identifiers and structured errors where they shorten diagnosis. The difference in practice:

```
error: query failed
```

versus:

```
level=error msg="query failed" request_id=req_8f3a21 route=/api/orders user_id=1842 db_wait_m
```

The first line tells you something broke. The second tells you *which* request, on *which* route, and that the database took 2.9 seconds — the diagnosis is half done. With systemd, structured fields also make the journal searchable:

```
journalctl -u app.service -o json --since "-1 hour" | grep req_8f3a21
```

Measure the service, not just the host

CPU and memory graphs describe the machine. Users experience request rate, error rate, and latency — measure those per service, plus dependency timing (how long *your* calls to the database or upstream API take). When latency spikes, one graph comparison tells you whether the slowness is yours or inherited.

A health check is the cheapest signal of all, as long as it checks something real:

```
curl -fsS https://app.example.com/health
# {"status":"ok","db":"ok","queue_depth":12}
```

A health endpoint that returns `ok` without touching its dependencies verifies only that the process can print `ok`.

Alerts that get acted on

Alerts need an owner, urgency, and action. "Disk 80% on web1" paging nobody in particular at 3am, with no hint of what to do, trains people to ignore it. A useful alert names who gets it, why it can't wait, and the first command to run. If an alert fires repeatedly and the response is always "ignore it", delete it or fix the threshold — every tolerated false alarm buries the real one.

Two things to keep out: secrets and unbounded labels. Passwords and tokens in logs become an incident of their own, and a metric labeled by user ID or request ID explodes into millions of series that cost real money.

The trap is adding observability *during* the incident and trusting it immediately. New instrumentation has its own bugs. Take one difficult diagnostic question from this course and add the smallest signal that would answer it — before the next incident, while you can verify the signal is telling the truth.

Write the incident review

Document impact, timeline, contributing conditions, response, recovery, and owned follow-up without blaming individuals.

A useful review explains how the system allowed the incident and how the response unfolded. Not who typed the bad command — how the system made that command possible, easy, and invisible until it hurt. Reviews that hunt for a culprit produce quiet engineers and repeat incidents; reviews that hunt for conditions produce fixes.

Write it within a few days, while the timeline you built and the evidence you preserved still make sense to you.

The structure

One page covers it. Include detection, impact, mitigation, root cause, contributing factors, what worked, and specific follow-up owners:

```
# Incident review: API outage 2026-08-03

**Impact**: /api/orders returned 500 for all users, 14:05-14:38 UTC (33 min).
**Detection**: user report in Slack at 14:33. No alert fired.

## Timeline (UTC)
```

```
- 14:02 deploy of release 2026.31 finished
- 14:05 error rate reached 100% (found later in logs)
- 14:33 first user report
- 14:38 rollback completed, external check green

## Root cause
Release 2026.31 read DB_POOL_SIZE as a string; the pool
initialized with 0 connections. [confirmed in staging]

## Contributing factors
- config values not validated at startup [fact]
- no alert on error rate; detection relied on users [fact]
- deploy finished at peak hours [inference: worsened impact]

## What worked
Rollback took 5 minutes; evidence capture made root cause quick.

## Follow-ups
- [ ] validate config at startup, fail fast - Ana, by Aug 14
- [ ] alert: 5xx rate > 5% for 5 min - Marco, by Aug 10
- [ ] deploy checklist: verify one real request - Flavio, by Aug 7
```

Separate confirmed facts from inference, exactly like in the timeline lesson. "The pool initialized with 0 connections" was reproduced in staging: fact. "Peak hours worsened impact": plausible inference, marked as such. A review that presents guesses as findings sends the next responder down the wrong path with extra confidence.

Follow-ups that actually change something

Prefer changes that remove failure modes over changes that ask humans to try harder. "Be more careful with config" fixes nothing — the same person under the same pressure makes the same mistake. "The app refuses to start with invalid config" makes the mistake impossible. And notice the detection gap deserves a follow-up of its own: 28 of the 33 minutes were spent not knowing.

Every follow-up needs an owner and a date, or it's decoration.

The trap: writing the review and never checking the follow-ups landed. A pile of reviews with open checkboxes is how the same incident happens twice, with a better-documented second act. Write a one-page review for a simulated incident, and end with one prevention change, one detection change, and one response improvement.

Check your understanding: recover and prevent

Check mitigation, rollback, recovery verification, observability, alerts, runbooks, incident review, and final diagnosis.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/linux-troubleshooting/.