macOS Internals and Troubleshooting Course

Flavio Copes

Updated August 3, 2026

Contents

Understand macOS processes, filesystems, metadata, services, logs, memory, networking, and recovery tools through evidence-based troubleshooting labs.

What you'll learn: Trace a Mac problem to its first failing boundary, collect useful evidence, repair the cause safely, and leave a repeatable diagnostic record.

Map the system

Connect Darwin, the kernel, processes, launchd, hardware architecture, and APFS volumes into one system model.

Separate macOS, Darwin, and the kernel

Connect the macOS product version to the Darwin kernel version without treating their version numbers as interchangeable.

macOS is the complete operating system: the kernel, the UNIX layer, the frameworks, and everything you see on screen. Darwin is its open-source UNIX foundation, and XNU is the kernel that manages processes, memory, devices, and system calls.

You care about the split because different tools report different version numbers, and mixing them up sends a bug report or a web search in the wrong direction.

Two version schemes

Compare the product and kernel views:

```
sw_vers
```

```
ProductName:        macOS
ProductVersion:     26.1
BuildVersion:       25B78
```

Read three fields. `ProductVersion` is the marketing version users talk about. `BuildVersion` identifies the exact build, which matters when a bug exists in one build and not the next.

Now ask the kernel:

```
uname -a
# Darwin studio.local 25.1.0 Darwin Kernel Version 25.1.0 ... RELEASE_ARM64_T6031 arm64
```

The first number after the hostname is the Darwin release. The macOS and Darwin version numbers follow different schemes: macOS 26 ships Darwin 25, and macOS 15 shipped Darwin 24. They move together but never match.

Why the layers matter when troubleshooting

When you read the unified log, kernel events come from the XNU layer. When you run `ls` or `ps`, you are using Darwin's BSD userland. When an app misbehaves in System Settings, you are in the macOS product layer. Knowing which layer produced a message tells you which documentation to read.

One more kernel-adjacent fact worth knowing: modern macOS pushes most third-party drivers out of the kernel. System extensions run in user space, while legacy kernel extensions are being phased out. A misbehaving ”driver” today is usually a user-space process you can inspect like any other.

The mistake to avoid

Record both versions when diagnosing low-level software, because a report that says only “version 25” is ambiguous. Is that Darwin 25 or a macOS release? You cannot tell, and neither can the person reading your report. Paste the output of `sw_vers` and `uname -a` verbatim and the ambiguity disappears.

Read the process tree

Use process identity, parentage, user, start time, and command to understand who launched a program and which context owns it.

Every process has an identifier and a parent, except the first system process. On macOS that first process is `launchd`, always PID 1:

```
ps -p 1 -o pid,ppid,command
#  PID  PPID COMMAND
#    1     0 /sbin/launchd
```

Process ancestry often explains unexpected environment, permissions, and lifecycle. The same program behaves differently depending on who started it, so before you debug a process, find out where it came from.

Inspect a compact process table

```
ps -axo pid,ppid,user,lstart,command
```

Read the columns like this: `PID` is the process, `PPID` is its parent, `USER` tells you which account it runs as, `lstart` gives the exact start time, and `COMMAND` shows the full executable path. The path matters. Two processes can both be called `node`, but `/opt/homebrew/bin/node` and a copy bundled inside an app are different stories.

Find the target PID, then follow PPID values upward:

```
ps -p 4821 -o pid,ppid,user,command
ps -p 812 -o pid,ppid,user,command   # 812 was the PPID of 4821
```

Repeat until you reach PID 1. A typical chain for a dev server looks like `node` ← `zsh` ← `login` ← `Terminal`. A background job started by `launchd` has PPID 1 directly.

Why the ancestry matters

A process started by Terminal, Finder, an application helper, and launchd can receive different environment and restart behavior. Terminal children inherit your shell's `PATH` and variables. launchd jobs get a minimal environment and come back automatically if configured to. Privacy permissions are attributed differently too, so a script may read a protected folder from Terminal but fail when launchd runs it.

The classic failure this explains: ”it works when I run it by hand, but fails as a background job”. Same binary, different parent, different environment. Compare the two ancestries before blaming the program itself.

Understand launchd domains

Distinguish system, user, and graphical-session services before inspecting or changing a background job.

`launchd` starts and supervises many system and user processes. Jobs live in **domains**, so the same label can have meaning only within its system or user context.

There are three domains you will actually query:

- `system`: jobs that run as root, from boot, with no user logged in. Defined in `/System/Library/LaunchDaemons` and `/Library/LaunchDaemons`.

- `user/<uid>`: background jobs for one user, even without a graphical session.
- `gui/<uid>`: jobs tied to a user's logged-in graphical session. Third-party agents install into `~/Library/LaunchAgents` or `/Library/LaunchAgents`.

Inspect a domain

Inspect the current graphical user domain:

```
launchctl print gui/$(id -u)
```

The output is long. It lists the domain's services with their labels and current state. Search for a specific label instead of reading the entire output blindly:

```
launchctl print gui/$(id -u)/com.example.sync
```

Read these fields in the output: `state` (running or not), `path` (the property list that defines the job), `program` or `arguments` (the executable it launches), and `last exit code`. Those four answer most "what is this thing?" questions.

Query the right domain

A daemon defined in `/Library/LaunchDaemons` does not appear in your `gui` domain. If you ask the wrong domain, `launchctl` reports the service as unknown and you may wrongly conclude it is not installed:

```
sudo launchctl print system/com.example.backupd
```

Note the `sudo`: the system domain belongs to root.

What not to do

Do not unload unknown Apple services; first identify the job owner, property list, executable, and recent exit state. Many Apple jobs restart on demand anyway, so a `bootout` teaches you nothing and can destabilize the session. Inspection is cheap and reversible. Removal is neither, so save it for jobs you have positively identified as yours or as third-party leftovers.

Map APFS volumes and mounts

Relate the startup disk container, system and data volumes, snapshots, and mounted paths without deleting storage metadata.

Modern macOS uses APFS volume groups and mounts to present one familiar startup hierarchy. What looks like a single "Macintosh HD" is really several volumes inside one **APFS container**, all sharing the same pool of free space.

The split that matters for troubleshooting: the sealed system volume and writable data volume have different roles. The system volume holds macOS itself, is cryptographically sealed, and mounts read-only at `/`. Your files, apps, and settings live on the data volume, mounted at `/System/Volumes/Data`. Firmlinks stitch the two together so `/Applications` and `/Users` feel like one filesystem.

Inspect rather than modify

```
diskutil apfs list
```

Read the tree: the container (something like `disk3`), then each APFS volume with its role. You will see `Macintosh HD` marked as the system volume and `Macintosh HD - Data` in the same volume

group. Note the volume identifiers (`disk3s1`, `disk3s5`) so you can match them elsewhere.

Now connect volumes to paths:

```
mount | head -5
df -h / /System/Volumes/Data
```

`mount` shows which device is mounted where, and that `/` is `read-only`. `df -h` reports capacity and free space. Both lines show the same available space because the volumes share the container.

Snapshots

APFS also keeps local snapshots, point-in-time views of the data volume that Time Machine and the update process create:

```
tmutil listlocalsnapshots /
# com.apple.TimeMachine.2026-08-02-221304.local
```

Snapshots explain the classic mystery of "I deleted 40 GB and got no space back". The space is held by a snapshot and is released when the snapshot expires.

The mistake to avoid

Do not delete a volume or snapshot because its name looks duplicated; verify its role and backup state first. `Macintosh HD - Data` is not a duplicate of `Macintosh HD`. It is your entire user data. Deleting storage metadata is the one troubleshooting move you usually cannot undo.

Check your understanding: map the system

Check the system model, evidence, safety boundaries, and diagnostic decisions from map the system.

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

Take this interactive quiz online

Inspect files and applications

Read permissions, ACLs, extended attributes, app bundles, property lists, Library folders, and quarantine state.

Read mode, ACLs, and flags

Inspect every macOS file access layer before changing permissions or recursively rewriting a directory.

Traditional owner, group, and mode bits are only part of file access. macOS also uses **access control lists** and **file flags**, and any of the three layers can be the one that blocks you. If you only look at `rwx` bits, you will fix the wrong layer.

Inspect one path and its parent

```
ls -led0@ project project/config.json
stat -x project/config.json
```

`ls -e` shows ACL entries, `-O` shows flags, and `-@` shows extended attributes. A typical result:

```
drwxr-xr-x@  8 flavio  staff  uchg 256 Aug  3 11:02 project
 0: group:everyone deny delete
-rw-r--r--   1 flavio  staff  -     412 Aug  3 11:02 project/config.json
```

Read it layer by layer. The `+` or `@` after the mode string signals there is more than mode bits. Numbered lines like `0: group:everyone deny delete` are ACL entries, evaluated before the classic bits — a single `deny` entry wins even when the mode looks permissive. The flags column (here `uchg`, the user immutable flag) can make a file read-only regardless of everything else.

`stat -x` confirms the numeric mode and exact ownership when the `ls` output is ambiguous.

Change only the proven layer

If an ACL denies the operation, edit the ACL (`chmod -a` removes an entry). If a flag locks the file, clear the flag (`chflags nouchg project`). If the mode is wrong, fix the mode. Change only the layer proven to block the intended process.

One flag deserves special respect: `restricted` marks paths protected by System Integrity Protection. You cannot clear it while SIP is on, and disabling SIP to "fix permissions" is never the answer. If a file is SIP-protected, macOS owns it, and the fix belongs elsewhere.

The mistake to avoid

Never start with recursive `chmod 777`. It rarely fixes ACL or flag problems, it destroys the permission record you needed as evidence, and it leaves world-writable files a later security review will flag. Diagnose first, then make the smallest change that unblocks the process.

Inspect extended attributes

Read file metadata such as quarantine state without removing every attribute as a generic installation fix.

Extended attributes store metadata outside normal file contents: named key-value pairs attached to a file. macOS uses them heavily. Downloaded items may carry quarantine information used by macOS security checks, Finder stores tags and comments in them, and apps stash their own bookkeeping there.

Inspect names and values

```
xattr file.zip
xattr -l file.zip
```

The first form lists attribute names only. The second prints values too. On a fresh download you typically see:

```
com.apple.quarantine: 0083;68b2c41a;Safari;9F2A1C44-73D2-4E8B-9E01-2B7C55D10E42
com.apple.metadata:kMDItemWhereFroms: ...
```

The quarantine value is semicolon-separated: flags, a hex timestamp, the application that downloaded the file, and an identifier. `kMDItemWhereFroms` records the download URL. Together they answer a genuinely useful question during troubleshooting: where did this file come from, and has Gatekeeper evaluated it yet?

When you first open a quarantined app, macOS runs its security checks. If the app is blocked, the quarantine attribute is part of why, and that is your cue to check the publisher, not to strip

metadata.

Treat attributes as evidence

Treat the attribute as evidence about origin and handling. Verify the publisher and download source. If a legitimately obtained file from a verified developer still trips a check, removing quarantine from that one file is a deliberate, informed decision:

```
xattr -d com.apple.quarantine Example.app
```

That is one attribute, one path, after verification.

The mistake to avoid

Do not normalize `xattr -cr` as an installation step because it removes metadata recursively without solving trust. It strips every attribute from every file in the tree — quarantine, tags, resource metadata, all of it. Any tutorial whose install instructions start with `xattr -cr` is asking you to blind the security system instead of understanding why it objected. Read the attributes first. They usually tell you exactly what happened.

Open an application bundle

Inspect the executable, resources, frameworks, helpers, and Info.plist inside a Mac app without changing its signed contents.

A `.app` is a directory with a defined bundle structure. Finder normally presents it as one application, but the shell sees a plain folder you can walk through. Knowing the layout lets you answer questions like ”which binary actually runs?” and ”does this app ship its own helpers?”.

Walk the structure

Inspect a copy or a trusted app:

```
find "/Applications/TextEdit.app/Contents" -maxdepth 2 -print
```

The layout is predictable. `Contents/MacOS` holds the main executable. `Contents/Resources` holds assets and localizations. `Contents/Frameworks` holds bundled libraries, and `Contents/PlugIns` or embedded `.app` bundles hold helper processes. When Activity Monitor shows a helper with a strange name, this is where it lives.

Read Info.plist

```
plutil -p "/Applications/TextEdit.app/Contents/Info.plist"
```

`Info.plist` describes identifiers, executable names, document roles, and capabilities. The fields worth reading first: `CFBundleIdentifier` (the bundle ID, `com.apple.TextEdit` here, which names the app's preferences and containers), `CFBundleExecutable` (which file in `MacOS` starts), and `CFBundleShortVersionString` (the version to record in any bug report).

The bundle ID is the key that links a running process back to its on-disk state, so note it down whenever you investigate an app.

Check the signature without touching anything

```
codesign -dv --verbose=2 /Applications/TextEdit.app
```

This prints the signing `Identifier` and `TeamIdentifier`, telling you who shipped the binary. It reads; it changes nothing.

The mistake to avoid

Changing files inside a signed app can invalidate its signature, so inspect first and repair through a trusted reinstall. Even ”harmless” edits like swapping an icon or deleting a localization break the seal, and macOS may then refuse to launch the app at all. If a bundle's contents look wrong or modified, the fix is a fresh copy from the developer, not surgery on the broken one.

Trace Library state by scope

Identify whether application state belongs to one user, every user, the system, a sandbox container, or a shared group.

The same application may use `~/Library`, `/Library`, and sandbox container directories for different state. Scope matters before deletion or migration: `~/Library` is one user's state, `/Library` applies to every user on the Mac, and `/System/Library` belongs to macOS itself and is off limits.

Sandboxed apps add a twist. Their writable world lives in `~/Library/Containers/<bundle-id>`, which mirrors a private Library inside, and app families share data through `~/Library/Group Containers`.

Search narrowly by bundle identifier

Get the app's bundle ID first, then look for it:

```
osascript -e 'id of app "Notes"'
# com.apple.Notes
```

```
ls ~/Library/Preferences | grep -i com.apple.Notes
ls ~/Library/Containers | grep -i com.apple.Notes
ls ~/Library/Application\ Support | grep -i Notes
```

Search narrowly by bundle identifier in the current user Library. Inspect Preferences, Application Support, Caches, Logs, Containers, and Group Containers without assuming each is present. A non-sandboxed tool may only have a single plist in Preferences; a sandboxed app may keep everything under its container.

What each location tells you: Preferences holds settings, Application Support holds documents and databases the app depends on, Caches holds regenerable data, Logs holds the app's own log files.

Change state carefully

Back up valuable data and quit the app before changing state. Quitting matters more than it looks: preferences are cached by the `cfprefsd` daemon, so if you delete a plist while the app runs, the old values can be written straight back and you will conclude, wrongly, that the file was not the problem.

Then reset one proven file or container, not the entire Library folder. Move the candidate aside instead of deleting it:

```
mv ~/Library/Preferences/com.example.editor.plist ~/Desktop/
```

Relaunch and retest. If the symptom survives, move the file back. That single-file discipline keeps the experiment reversible and tells you precisely which piece of state carried the fault.

Check your understanding: inspect files and applications

Check the system model, evidence, safety boundaries, and diagnostic decisions from inspect files and applications.

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

Take this interactive quiz online

Diagnose processes and resources

Inspect process ancestry, open files, CPU, memory pressure, disk activity, hangs, crashes, and resource limits.

Identify the real process

Match an application symptom to the responsible process, executable, user, architecture, parent, and open resources.

An application can contain several helpers. The visible app name may not match the process consuming CPU or holding a file. A browser is the obvious example: one window, but a main process plus a renderer helper per tab group, a GPU helper, and a networking helper. When ”the browser is slow”, the question is which of those processes is slow.

Find the candidate, then verify it

Start with Activity Monitor or `ps`. Activity Monitor's CPU tab sorted by `% CPU` gives you a name and a PID fast. In Activity Monitor you can also double-click the process and check the Open Files and Ports tab.

Then inspect one PID from the shell:

```
ps -p 1234 -o pid,ppid,user,%cpu,%mem,etime,command
```

Replace `1234` with the observed PID. Read each field with intent: `USER` tells you whose session owns it, `ETIME` says how long it has been alive (a process born seconds ago is a different suspect than one running for days), and `COMMAND` shows the full executable path. The path is the identity check — a helper inside `/Applications/Slack.app/Contents/Frameworks` belongs to Slack no matter what its short name says.

Check what it holds open

```
lsof -p 1234 | head
```

The first lines show the working directory (`cwd`), the executable itself (`txt`), and then open files, sockets, and devices. If a process ”won't let go” of a file or a port, this is where you see it, with the exact path in the `NAME` column.

Confirm before acting

Confirm ownership and purpose before quitting or signaling it. Killing a helper often just makes the parent respawn it, and killing the wrong `node` takes down someone else's work.

One trap: PIDs are reused, so recheck immediately before acting. A PID you wrote down ten minutes ago can now belong to an entirely different process. Re-run the `ps` line, confirm the same command path, then act.

Measure CPU and a hang

Distinguish busy computation from waiting, then capture a short process sample before force quitting an unresponsive app.

High CPU and an unresponsive window are symptoms, not causes. They are also independent. A process can spin at 100% CPU doing real work, and a process can also hang while waiting on a lock, file, network request, or another process while using almost no CPU at all.

The split tells you where to look. High CPU plus no progress suggests a busy loop or runaway work. A beachball with near-zero CPU means the app is blocked on something, and the interesting question is on what.

Activity Monitor gives you the first read: the `% CPU` column, and the process shown in red as "Not Responding" when its main thread stops servicing events.

Capture a short stack sample

Before you force quit, take evidence. `sample` records where a process's threads spend time:

```
sample 1234 5 -file sample.txt
```

That samples PID 1234 for 5 seconds and writes a text report. Open it and find the call graph: an indented tree of function names with counts next to them.

Read repeated stacks to find where threads spend time. Follow the largest counts downward. If the main thread shows the same deep stack in nearly every sample — say, sitting inside a file-read call or waiting on a lock function — that stack names the wait. Function names from the app's own code, visible in the tree, tell the developer exactly where it stuck.

For a hang that spans several processes, `sudo spindump` captures the whole system for a few seconds, which helps when app A is blocked waiting on app B.

Preserve context, then quit gently

Save the app version and reproduction with the sample. A stack trace without the version it came from is much less useful in a bug report.

Then, and only then, unstick the app. Use normal Quit before Force Quit when data may be unsaved. A hung app sometimes recovers when whatever it waited on completes, and normal Quit gives it a chance to save. Force Quit is the last step, not the first.

Interpret memory pressure

Use system memory pressure, swap, and process growth instead of treating low free memory as proof of a leak.

macOS uses available memory for caches. Low free memory alone is normal — unused RAM is wasted RAM, so the system fills it with recently used files and gives it back the moment an app needs it. That is why "only 200 MB free" is not, by itself, a problem.

Memory Pressure summarizes whether the system can satisfy demand efficiently. Activity Monitor's Memory tab shows it as a graph: green means comfortable, yellow means the system is compressing memory to keep up, red means it is struggling and swapping heavily. The `Swap Used` figure next to it tells you how much has been pushed to disk.

Inspect system and process evidence

`memory_pressure`

Skip to the last line: `System-wide memory free percentage: 68%`. That single number is the CLI equivalent of the pressure graph.

`vm_stat`

Read three counters: `Pages free`, `Pageouts` (memory written to swap — a steadily climbing value under load is real pressure), and `Pages occupied by compressor` (how much the system is compressing to avoid swapping).

Then rank processes by resident memory:

```
ps -axo pid,rss,%mem,etime,command | sort -nrk2 | head
```

`rss` is resident memory in kilobytes. Note the top entries together with `etime`, because size only means something relative to how long the process has run and what it was doing.

A leak is a trend, not a reading

Compare the same workload over time. Run the suspect app through the same task today and an hour from now, and record `rss` each time. A leak is sustained unexplained growth connected to a process and reproduction, not one large reading. A browser using 6 GB with forty tabs open is doing its job.

The classic mistake is the opposite reflex: seeing yellow pressure once and force-quitting everything, or installing a "memory cleaner". You lose the evidence and the trend, which were the only things that could have told you whether a real leak existed.

Trace disk activity, files, and crashes

Connect disk pressure or an application crash to the process, path, report, and time window without running broad destructive cleanup.

When the disk is grinding or filling up, you want two facts: which process is doing the writing, and to which paths. When an app dies, you want the report it left behind. Both are answerable without deleting anything.

Who is hitting the disk

Use Activity Monitor's Disk view and `lsof` to identify the process and open paths. The Disk view's `Bytes Written` and `Bytes Read` columns, sorted descending, name the busy process. Then list what it has open:

```
lsof -p 1234 | grep -v '\.dylib' | head -20
```

The `NAME` column gives you the actual paths, and filtering out libraries keeps the list readable.

For a short authorized trace, `fs_usage` can show live filesystem calls but produces sensitive, noisy output and may require elevated access:

```
sudo fs_usage -w -f filesys 1234
```

Each line shows the syscall, the path, and the elapsed time. Watch for a few seconds, then stop with Ctrl-C. That is usually enough to catch a process rewriting the same cache file in a loop.

Where crash reports live

Crash reports live in the user or system DiagnosticReports directories and can also be viewed through Console:

```
ls -lt ~/Library/Logs/DiagnosticReports | head
ls -lt /Library/Logs/DiagnosticReports | head
```

User processes report to the first, system daemons to the second. Recent reports are `.ips` files named after the process and timestamp. Open one in Console (Crash Reports section) or read it directly.

Match process name, timestamp, version, exception, and crashed thread. Concretely: confirm the timestamp matches your incident, record the app version, then read `Exception Type` (for example `EXC_BAD_ACCESS (SIGSEGV)`) and jump to the thread marked `Crashed`. Its stack is the heart of the report and what a developer needs.

Keep the evidence

Do not delete caches or reports before preserving the evidence that explains the failure. "Cleanup" utilities that purge DiagnosticReports destroy the only record of why the app died. Copy the relevant `.ips` file somewhere safe first; free the disk space second.

Check your understanding: diagnose processes and resources

Check the system model, evidence, safety boundaries, and diagnostic decisions from diagnose processes and resources.

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

Take this interactive quiz online

Trace services, logs, and network

Query launchd, unified logs, listeners, routes, DNS state, and connection paths without destroying evidence.

Query unified logs narrowly

Filter the macOS unified log by time, process, subsystem, category, and message instead of exporting unrelated system history.

The **unified logging system** stores structured events viewed through Console or the `log` command. Every process on the Mac writes into it, which is both the strength and the trap: everything is there, and an unfiltered query drowns you in it.

The discipline: begin with the failure time and one process or subsystem. You know roughly when the problem happened. Anchor every query to that window.

Filter by process and time

```
log show --last 10m --style compact \
  --predicate 'process == "ExampleApp"'
```

The compact style prints one event per line: timestamp, event type, then `process[pid:thread]` and the message. Scan the timestamps around the failure moment and read the messages near it.

Predicates compose, so you can tighten further:

```
log show --last 10m --style compact \
  --predicate 'subsystem == "com.example.sync" AND eventMessage CONTAINS[c] "timeout"'
```

`subsystem` and `category` are the labels developers give their own log lines, and `eventMessage CONTAINS[c]` does a case-insensitive text match. Those four keys — `process`, `subsystem`, `category`, `eventMessage` — cover most troubleshooting queries.

Watch live instead of digging

When you can reproduce the problem on demand, stream instead:

```
log stream --predicate 'process == "ExampleApp"'
```

Start the stream, trigger the failure, and read what appears at that instant. This beats archaeology every time the bug is reproducible.

Scope and privacy

Expand the time window only when evidence requires it. `--last 10m` returns in seconds; `--last 24h` without a predicate can take minutes and produce gigabytes of noise that hides the signal.

Logs can include private paths and user data, while some values are deliberately redacted. You will see `<private>` where an app chose not to expose a value — that is by design, not corruption. And treat exported logs as sensitive: they can reveal filenames, server names, and account hints, so scrub them before attaching to a public bug report.

Connect launchd state to logs

Use job state, last exit information, executable paths, and matching log events to diagnose a background service.

A background process that disappears may have exited normally, crashed, been throttled, or failed before launch. Each of those leaves a different trail, and launchd plus the unified log together tell you which one happened.

Inspect the owning launchd domain and exact label first. Guessing at the label wastes time; find it in the job's property list or with `launchctl print gui/$(id -u)` and a search.

Read the job state

Capture `launchctl print` output for the job:

```
launchctl print gui/$(id -u)/com.example.sync
```

Four fields carry most of the diagnosis. `state` tells you whether it is running now. `path` points at the property list that defines it. `program` (or the `arguments` array) is the executable launchd tries

to run. `last exit code` is the punchline: `0` means the last run ended normally, a non-zero value means the program reported failure, and a line mentioning a signal means it was killed or crashed.

Compare the executable path with the expected installation. A job whose `program` points at `/usr/local/bin/sync-agent` after you moved the tool to `/opt/homebrew/bin` will fail before launch, forever, with no output of its own.

Match the log events

Then query recent logs for its process or subsystem:

```
log show --last 30m --style compact \
  --predicate 'process == "sync-agent"'
```

Line up log timestamps with the exit state you just captured. A crash shows the process starting and dying repeatedly. A throttled job shows launchd complaining about respawning too fast. A path problem shows launchd erroring at spawn time with no process events at all.

Preserve before restarting

Do not repeatedly kickstart the job before saving exit state and logs. Restarts can erase timing and create a new symptom. Every `launchctl kickstart -k` resets `last exit code` and muddies the timeline, so capture both outputs to a file first, then experiment.

Inspect listeners and connections

Identify the process, address family, local address, port, and connection state behind a network symptom.

"The port is open" is incomplete. A listener can bind only to loopback, one interface, IPv4, IPv6, or every address. Two services can even share a port number on different addresses. Until you know the exact binding, "it works here but not from the other machine" stays a mystery.

Inspect TCP port 3000

```
lsof -nP -iTCP:3000 -sTCP:LISTEN
```

`-n` and `-P` skip DNS and service-name lookups so you see raw addresses and ports. Typical output:

```
COMMAND  PID    USER   FD  TYPE  DEVICE  SIZE/OFF  NODE  NAME
node     4821   flavio 23u IPv4  0x1a2b  0t0       TCP   127.0.0.1:3000 (LISTEN)
```

Read four things. `COMMAND` and `PID` identify the process — verify the PID with `ps -p 4821 -o command` to see the full path. `USER` tells you whose process it is. `TYPE` says IPv4 or IPv6, which matters because a server listening only on IPv6 is invisible to IPv4 clients. And `NAME` shows the bound address: `127.0.0.1:3000` accepts loopback connections only, while `*:3000` accepts from any interface.

That last distinction solves the most common case: your dev server answers on `localhost` but not from your phone because it bound to `127.0.0.1`, not `0.0.0.0`.

See active connections too

Drop the state filter to include established connections:

```
lsof -nP -iTCP:3000
```

Now each line's `NAME` shows `local->remote` pairs with a state like `(ESTABLISHED)` or `(CLOSE_WAIT)`. A pile of `CLOSE_WAIT` entries points at an app not closing sockets it is done with.

The mistake to avoid

Match the PID to the expected executable and user before doing anything else. Do not kill an unknown listener just to free the port; identify who owns it and what depends on it. The "mystery" process on your port is often a legitimate system service or another project of yours, and killing it trades one symptom for two.

Trace route, DNS, and path

Separate name resolution, route selection, reachability, transport connection, and application response during a Mac network failure.

"The internet is broken" is five different failures wearing the same coat. Test the path in layers. First inspect DNS configuration and answers, then the selected route, then the destination port, then the application protocol. Each command below answers exactly one question, and the first one that fails names your problem.

Walk the layers

Which resolver is the Mac actually using?

```
scutil --dns | head -15
```

Read `resolver #1` and its `nameserver[0]` entries. This is the ground truth, whatever you think you configured.

Does the name resolve?

```
dig example.org
```

Check `status: NOERROR` in the header and the `ANSWER SECTION` for the returned address. `NXDOMAIN` means the name itself fails; an unexpected IP means you are being answered by the wrong resolver.

Where does traffic leave the machine?

```
route -n get default
```

Read `gateway` and `interface`. Traffic going out `en0` when you expected the VPN's `utun` interface explains a lot of "works off VPN, fails on VPN" cases.

Can we reach the port?

```
nc -vz example.org 443
# Connection to example.org port 443 [tcp/https] succeeded!
```

That `succeeded!` line proves DNS, routing, and the TCP handshake all work. A timeout here with working DNS points at a firewall or a dead host.

Does the application layer answer?

```
curl -v https://example.org/
```

`-v` shows the TLS handshake, the certificate, and the HTTP status. A certificate name mismatch or an HTTP `503` lives here, above everything you tested before.

Stop at the first failing boundary

Fix where the failure is, not where habit points. Changing DNS cannot fix a local listener, and flushing caches cannot repair a TLS name mismatch. If `dig` succeeds and `nc` succeeds, stop touching DNS.

One more distinction: `networkQuality` measures capacity and responsiveness of the connection. Useful when everything works but feels slow — a different investigation than the layered reachability walk above.

Check your understanding: trace services, logs, and network

Check the system model, evidence, safety boundaries, and diagnostic decisions from trace services, logs, and network.

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

Take this interactive quiz online

Recover from failures

Isolate scope, compare user and system state, use safe startup and recovery tools, verify repairs, and document the incident.

Define scope before changing state

Determine whether a failure affects one file, application, user, network, peripheral, or the whole Mac before applying a fix.

The most expensive troubleshooting mistake happens before any command runs: fixing the wrong-sized problem. Reinstalling macOS for a broken preference file, or nuking one app's cache when the whole disk is failing. **Scope** is what stops that.

Start with observable scope. Ask when the failure began, how often it occurs, which users and apps are affected, and what still works. Does the same document fail in a different app? Does the same app fail on a different file? Does anything fail for another user? Each answer shrinks the search space by half.

Write facts before touching anything

Write facts separately from hypotheses. ”Save fails with error -36 in Pages since Thursday” is a fact. ”The disk is probably dying” is a hypothesis. Keep them in different sections so you never mistake a guess for evidence.

```
FACTS
- 2026-08-03 09:12: Pages "Save" fails, error -36, file on external SSD
- Same file saves fine to ~/Documents
- TextEdit save to the SSD also fails
- macOS 26.1 (25B78), Pages 14.2

HYPOTHESES
- external SSD or its cable/filesystem, not Pages
```

Preserve versions, timestamps, errors, logs, and a small reproduction before updating, reinstalling, or deleting state. Every state change you make afterward overwrites part of the crime scene, so the record has to come first.

Match the experiment to the scope

A narrow symptom deserves a narrow experiment. In the example above, the facts already point at the SSD, so the next test is copying one file to it from Finder — not reinstalling Pages, and not rebooting.

Restarting the whole Mac may restore service, but it also removes process and timing evidence. Sometimes a restart is the right call, especially when others are blocked. Make it a decision, not a reflex, and capture the evidence you will want afterward — open process list, recent logs, the exact error text — before you press Restart.

Compare another user and safe mode

Use a clean account and Safe Mode as controlled comparisons without treating either result as the final diagnosis.

macOS gives you two built-in comparison environments, and each one removes a different set of variables.

A fresh local account removes the original user's login items, preferences, and application state from the comparison. Create one in System Settings → Users & Groups; it costs nothing and you can delete it after the investigation.

Safe Mode changes startup behavior and performs limited checks. It loads only required system components, blocks login items and third-party startup software, and verifies the startup disk. On Apple silicon: shut down, hold the power button until "Loading startup options" appears, select the startup disk while holding Shift, and choose "Continue in Safe Mode". On Intel Macs, hold Shift right after power-on.

Confirm you are really in Safe Mode before trusting the test:

```
system_profiler SPSoftwareDataType | grep "Boot Mode"
# Boot Mode: Safe
```

If that says `Normal`, the Shift key did not register and your comparison is worthless.

Run the identical reproduction

Run the same exact reproduction in the original account, a clean account, and Safe Mode where appropriate. "Same exact" is the whole method: same file, same steps, same network. Record what changes and what remains:

```
Original account:  crash on export, every time
Clean account:     no crash, same file, same steps
Safe Mode:         not tested (symptom already isolated to user scope)
```

Interpret without over-claiming

If a clean account works, investigate user-scoped state. The suspects are the original account's preferences, containers, caches, and login items — which you already know how to trace by scope.

If Safe Mode works, inspect startup software and extensions. Something that loads at login or startup is interfering, and the next step is disabling those items one at a time in the normal boot.

Neither result names the cause by itself. These are scope-narrowing experiments, not verdicts. Safe Mode also clears some caches as a side effect, so ”fixed after one Safe Mode boot” may just mean a cache was rebuilt — note it, reboot normally, and confirm the symptom is actually gone.

Use recovery tools with a purpose

Choose Apple Diagnostics, Disk Utility First Aid, Safe Mode, or macOS Recovery only when the evidence matches the tool’s boundary.

Each recovery tool answers one question, and using a tool outside its boundary produces answers that mean nothing.

Apple Diagnostics checks supported hardware: memory, power, sensors. It ends with a reference code — `ADP000` means no issue found, anything else is a code to record verbatim and look up. It says nothing about software.

Disk Utility First Aid checks filesystem structures. It verifies and repairs APFS metadata. It cannot fix a misbehaving app, and a clean First Aid pass does not rule out a failing drive — it only rules out structural filesystem damage.

macOS Recovery provides repair, restore, and reinstallation paths outside the normal startup system. From there you can run Disk Utility on the startup disk, restore from Time Machine, or reinstall macOS.

Pick the tool the evidence points at

Match symptom to boundary. Kernel panics or shutdowns under load: run Apple Diagnostics first. Files failing with I/O errors, a volume that will not mount: First Aid, from Recovery so the startup disk is not in use. System files damaged but user data intact: a reinstall over the existing system.

Before any of them: back up recoverable data before repair when possible. Repairs can fail halfway, and a disk healthy enough to copy files from today may not be tomorrow. Record error codes and exact disk targets — ”First Aid failed” is useless, ”First Aid on disk3s5 exited with error 8 at 14:32” is evidence:

```
2026-08-03 14:32  First Aid on disk3s5 (Macintosh HD - Data)
result: exit code 8, "The volume could not be verified completely"
next: back up now, then retry from Recovery
```

Reinstalls and the SIP temptation

Do not erase or reinstall because a generic checklist places it first. A reinstall can replace system files while preserving the user problem that caused the symptom, so define the expected outcome before starting. If the fault lives in `~/Library`, a reinstall changes nothing except your afternoon.

Recovery's Terminal also offers `csrutil`, which can disable System Integrity Protection. Never do that as a troubleshooting step. No ordinary diagnosis requires it, and any guide that starts there is a guide to distrust.

Write and verify the diagnostic record

Finish a complete troubleshooting case from symptom and evidence through cause, repair, verification, rollback, and prevention.

This is the capstone exercise. Everything in this course — scoping, process trees, launchd state, logs, file layers — converges into one written artifact: a diagnostic record another person can follow.

Choose one controlled failure: a shadowed executable, broken LaunchAgent path, occupied port, denied file permission, or user-scoped preference problem. Set it up deliberately on your own Mac so you know the true cause, then investigate as if you did not. Practicing on a failure with a known answer is the point — you can grade your own investigation against the truth.

The record's structure

Record the starting state, reproduction, scope, evidence, hypotheses, one-variable experiments, first failing boundary, repair, and post-repair checks. A working skeleton:

```
CASE: LaunchAgent com.example.sync never runs
STARTING STATE: macOS 26.1 (25B78), agent installed 2026-08-01
REPRODUCTION: log out/in; expected sync marker file absent
SCOPE: one user, one job; other agents load fine
EVIDENCE:
  launchctl print gui/501/com.example.sync -> last exit code = 78
  log show --last 30m --predicate 'process == "sync-agent"' -> no events
  plist ProgramArguments -> /usr/local/bin/sync-agent (missing)
  which sync-agent -> /opt/homebrew/bin/sync-agent
HYPOTHESIS: plist points at the old install path
EXPERIMENT (one variable): fix path in plist, reload the job
RESULT: state = running, marker file appears
ROLLBACK: restore plist from ~/Desktop/com.example.sync.plist.bak
PREVENTION: install script should write the resolved path
```

Notice the shape: every claim in EVIDENCE is a command plus its observed output, and the experiment changes exactly one variable. Include commands but remove private identifiers — usernames, hostnames, internal server names — before sharing.

Verify the record, not just the fix

The fix working once is weak evidence. Run the reproduction again after the repair, and once more after a restart, before writing RESULT. Then have another person follow the evidence without your explanation. If they reach the same cause from your record alone, the record is complete. If they ask you a question, the answer belongs in the record.

A useful record proves why the repair fits the cause and how to reverse it. The rollback line is not decoration: a repair you cannot undo is a bet, and the record is what turns bets back into experiments.

Check your understanding: recover from failures

Check the system model, evidence, safety boundaries, and diagnostic decisions from recover from failures.

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/macos-internals-troubleshooting/.