LITPATH LABS / FIELD MANUAL 01

GB10 WITHOUT LYING TO YOURSELF

OPERATING A DGX SPARK

A MADMAN'S FIELD MANUAL

THE INSTRUMENT

[N / A]



UNKNOWN IS NOT EMPTY.

TWELVE PLATES / THREE PARTS / ONE SHARED POOL

J.C. PEREGOY

FIRST EDITION · FREE TO READ · NOVEMBER 2026

READ STRAIGHT THROUGH / FIND THE FAILURE

CONTENTS

The machine reports. It does not confess. 3

GB10 in ninety seconds 5

Find the symptom 7

PART 1 / THE INSTRUMENTS LIE

01 The N/A Card 9

02 The Stuck-Busy Mismatch 13

03 Ping Is Not Health 17

04 Silent Success 21

PART 2 / ONE POOL, ONE PRESSURE

05 Host Pressure Is Gpu Pressure 26

06 The Wedge That Needs A Power Cycle 29

07 The Idle Squatter 32

08 The Compile Is Load Too 36

PART 3 / STANDING ORDERS

09 Free First, Then Load 41

10 The Watcher Goes Away 44

11 Don'T Believe The Quant'S Brochure 48

12 The Do-Not-Touch List 51

Source notes & incident register 55

THE MACHINE REPORTS.

IT DOES NOT CONFESS.

You bought a machine with enough memory to make an ambitious idea look practical. Then a render slowed to a crawl while its queue said idle. A model answered ping through an outage. A successful request returned an empty string. Every instrument supplied a fact. None supplied the answer you needed.

This is a field manual about the distance between those two things.

The twelve plates come from a working lab: two ASUS Ascent GX10 systems built around NVIDIA GB10, with a Windows workstation alongside them. The incidents are small enough to recognise and expensive enough to remember. They are not a benchmark suite. They are the occasions on which a plausible interpretation survived until somebody opened the output, timed the job or read the next instrument.

The useful habit is simple: name what a reading proves, then name what it leaves untested. An empty queue proves that an engine has no queued work. It does not prove that the engine released its model. A finished HTTP request proves that a request finished. It does not prove that the result contains anything worth keeping.

The chapter device is **THE LIE / THE TRUTH**. Usually the instrument is not literally lying. The left column is the conclusion an operator is tempted to draw; the right is the observation that defeats it. Sometimes the bad conclusion came from our own runbook. Those are worth keeping in the book too.

Read a plate when the symptom appears. Read the three standing-order plates before it does. Keep a small known workload and a record of what it normally costs. They are more useful than a beautifully coloured dashboard with nothing calibrated behind it.

Who this is for

You operate a local AI workstation: model serving, image or video jobs, CUDA extensions, a few services sharing more memory than their separate dashboards admit. You can read a shell command and identify the service it addresses. You want fewer mystery outages and fewer confident explanations that fall apart at the next measurement.

This is an operator's incident book, not a setup guide or a replacement for the hardware vendor's recovery procedure. The GB10 examples are the centre of it; the same discipline often travels to other systems. The exact numbers do not.

How the evidence is marked

Observed: a value or outcome retained in the lab's dated incident records. The original records are curated operational notes, not a complete set of raw traces.

Reconstructed readout: a compact representation of those observations, with placeholder hostnames or omitted fields. Unless explicitly identified otherwise, terminal blocks in this book are reconstructed examples. A `$` or `PS>` prefix identifies a shell, not text to paste.

Diagnostic recipe: a proposed check, with comments describing how to interpret it. No output is invented to make a recipe look tested.

Inference: an explanation consistent with the record but not isolated by a controlled experiment. An untested local prohibition stays a policy; it does not become a product defect.

Edition note

First edition, free to read · November 2, 2026. J.C. Peregoy · LitPath Labs.

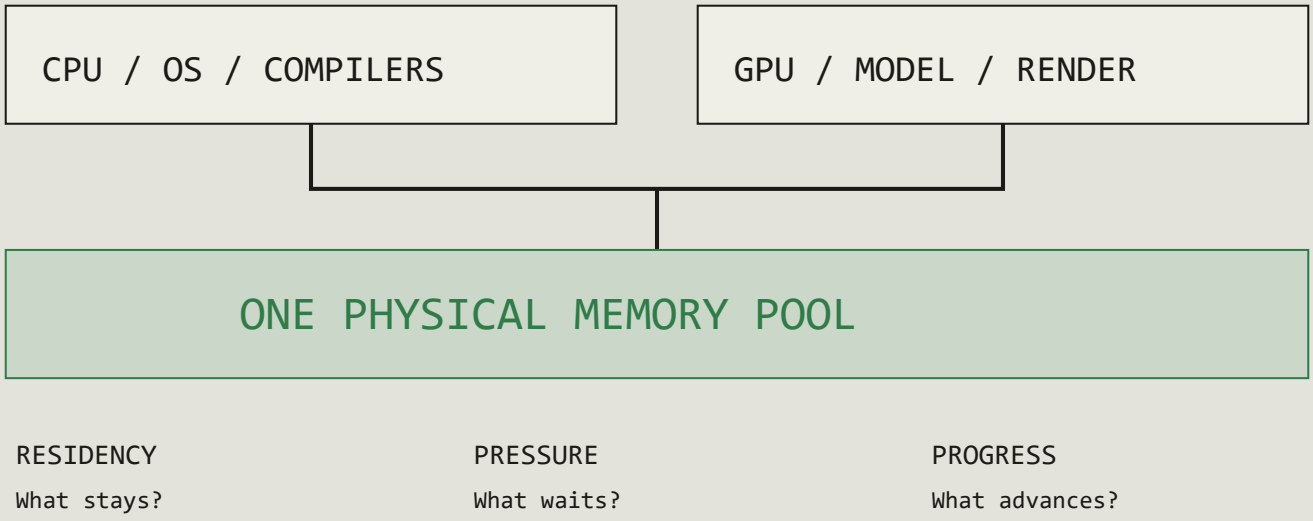
The hardware incidents date mainly from July and August 2026. Model, driver and runtime behaviour can change. NVIDIA's current release notes describe changes to memory handling, and expressly distinguish Founders Edition updates from partner-system schedules. The old incidents remain evidence of those runs; they are not a claim that every current installation reproduces them. See NVIDIA's release notes.

The manuscript was developed with AI assistance and checked against the available incident records and primary documentation. It is published free to read. It is not a vendor publication or an independently replicated performance study.

All original text and diagrams © 2026 J.C. Peregoy / LitPath Labs. Product names identify the systems discussed and do not imply endorsement.

GB10 IN NINETY SECONDS

One physical memory pool is the organising fact. CPU work and GPU work can compete for that capacity. An idle model server, a compiler and a render engine therefore belong in the same admission decision. A process does not become cheap merely because it has no CUDA context.



The CPU and GPU draw from one physical pool; residency, pressure and progress answer different questions.

Capacity is not bandwidth. A large pool may hold a model whose single-user generation remains slow. A small job may fit easily while saturating compute. Name the workload before deciding which resource limits it.

Unknown is not empty. On the observed GB10 Linux stack, aggregate framebuffer totals were unsupported while per-process GPU memory was available. Use the tools that report meaningful values on your own version. A blank field must remain unknown in your dashboard.

Occupancy is not pressure. Used RAM and occupied swap describe residency. Stall time, I/O and workload progress describe what that residency is costing. Both views matter.

File size is not peak runtime memory. Add context storage, temporary buffers, concurrency and the loader's behaviour. Keep a budget attached to a tested configuration.

Three measurements beat one verdict. What is resident? What is waiting? Is the requested work advancing? A useful health check answers the question you actually depend on.

Units you will meet

MARK	MEANING IN THIS BOOK
GB / GB/s	Decimal bytes or bytes per second, unless an incident note explicitly lacks precision
GiB / MiB	Binary units: powers of 1,024
`free -g`	Integer GiB-scale readings; rounded rows are not exact accounting identities
tok/s	Generated tokens per second; state batch/concurrency and whether it is per-user or aggregate
s/step	Seconds per measured work step; compare the same job and settings

Historical notes sometimes use “GB” loosely. We preserve approximate observations as approximate and state units explicitly for new calculations. Numbers printed by different instruments are not automatically additive.

QUICK ENTRY

FIND THE SYMPTOM

WHAT YOU SEE	FIRST USEFUL CHECK	PLATE
Memory total reads N/A	Read the supported host and process views.	01
GPU looks stuck, but work may be advancing	Time the same workload; do not convict it by one counter.	02
Ping works; SSH or the app does not	Test the layer you actually need.	03
Success returned an empty description	Validate content and completion before filing.	04
Swap is full; the render is slow	Measure pressure and progress, not occupancy alone.	05
A new model load took useful access away	Stop adding work; use the independent recovery path.	06
Queue is empty; memory is still occupied	Identify and unload the idle tenant.	07
A CUDA build destabilised the host	Inspect the build tree and limit parallelism.	08
Two large models need the same machine	Release, verify, then admit the replacement.	09
UI says failed; engine may still be working	Preserve job identity and inspect the engine record.	10
New quant, disappointing chat speed	Separate first-token latency and steady decode.	11
Maintenance may remove your way back in	Establish ownership and independent access first.	12

PART 1 / PLATES 01–04

THE INSTRUMENTS LIE

A reading can be accurate while the conclusion drawn from it is wrong. Start with what the instrument actually proves.

PLATE 01 · PART I

THE N/A CARD

Failure class: blind instrument · Escape hatch: read `free -g`, attribute from the services

You open `nvidia-smi` because the box feels full and you want to know what is holding it. The header prints. Where the memory gauge should be — the one number you came for — there are two brackets and three letters: `[N/A] / [N/A]`. The tool that exists to tell you what is on the card has just told you it doesn't know. Meanwhile the render that made you look is crawling, and somewhere in the unified pool there is 75 GB of something you cannot name.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`Memory-Usage: [N/A] / [N/A]` The driver can't read the framebuffer, so there is nothing to worry about here. No number means no problem. Look somewhere else for your stall.	`free -g → used 75 · available 45 · swap 15/15` `available` reads 45 in the recorded `free -g` output while swap is full and the job crawls. That combination calls for investigation; swap occupancy alone does not prove current thrashing. The card is not empty; the *counter* is empty. The memory you want to see is host memory, and job progress and memory-pressure measurements establish whether the host is struggling (Plate 05).

The gauge reads blank not because nothing is resident but because, on the GB10, the classic framebuffer totals are unsupported on the observed build. Grace-Blackwell shares one physical pool between CPU and GPU. `nvidia-smi` was written for a discrete card with its own VRAM, so `memory.used` and `memory.total` — the fields the whole ecosystem of monitoring scripts scrapes — come back `[N/A]`. The instrument is not broken. It is answering a question the architecture doesn't have: *how full is the card's dedicated memory?* There is no dedicated memory. There is one pool, and this tool cannot see it.

The incident, from the log

A render that normally finishes warm in about eight minutes was crawling at 265 seconds per step — a single six-step pass stretched to roughly 27 minutes, and still going. The first instinct is the right instrument, the wrong reading:

```
$ nvidia-smi --query-gpu=utilization.gpu,memory.used,memory.total --format=csv
utilization.gpu [%], memory.used [MiB], memory.total [MiB]
96 %, [N/A], [N/A]
# 96% util looks like hard work. The two [N/A]s look like "nothing to see."
# Neither reading establishes useful progress; the missing total is unsupported, not zero.
```

The utilisation field says busy; the memory field says unknown. Reading unknown as empty is the mistake. So you stop asking the card and start asking the architecture:

```
$ free -g
              total        used        free      shared  buff/cache   available
Mem:            121          75           2           0          44          45
Swap:            15          15           0
# available reads 45, NOT free's 2 — and swap is full in this slow run.
# free reads 2 and looks fatal; available reads 45 (buff/cache 44 is the gap).
# Neither supplies a diagnosis by itself. Compare pressure and actual progress (Plate 05).
```

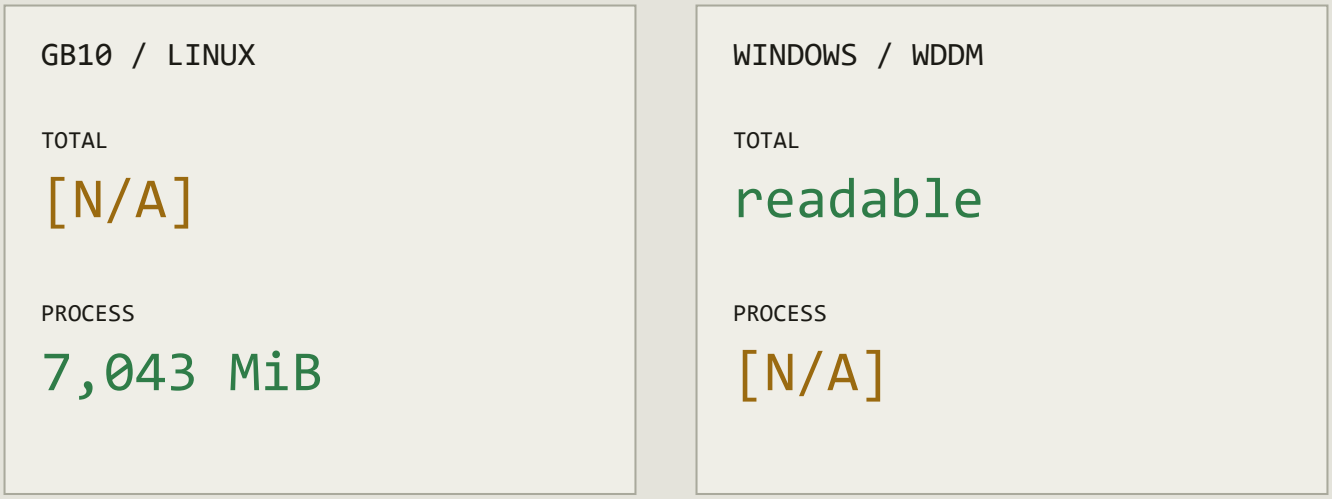
There it is — not on the GPU tool, on the memory tool. Now the second question: *what* is holding 75 GB. the aggregate gauge will not itemise it, but on this Linux box the per-process query does work:

```
$ nvidia-smi --query-compute-apps=pid,used_memory --format=csv
pid, used_gpu_memory [MiB]
<pid>, 7043 MiB        # a second render engine, idle — queue running:0 — squatting 7 GB
# The totals were [N/A]. The per-process figures are real.
# On Linux, ask per-process, not per-card.
```

The 7,043 MiB was a second render service sitting on the card doing nothing (Plate 07). Freeing it without stopping it — `curl -X POST localhost:8190/free -d '{"unload_models":true,"free_memory":true}'` — dropped its reported footprint to 355 MiB. Over the recovery interval, the host's `available` rose 45 → 79 and swap occupancy fell 15/15 → 8/15. The stalled render finished. Those host deltas include changing system state; they are not all attributable to the one process (Plate 05). The per-process change was visible in `nvidia-smi`; the host-wide recovery needed a second instrument.

The lesson is procedural, not lucky. When the GPU tool goes blank, you do not stare at it harder. You change instruments: `free -g` for the pressure, `--query-compute-apps` for the culprits, and — where those still leave a gap — the services themselves. A render UI knows its own allocation (`/sdapi/v1/memory` → `cuda.allocated.current`); an inference server lists its resident models with sizes (`/api/ps` → `size_vram` per model); a second render engine reports its footprint (`/system_stats`; inspect its reported free and total values together). Those three endpoint/field names are the ones the render-UI and inference-server builds we run actually expose; a reader on different software needs the equivalent endpoint, not these literal paths. These accounting views can overlap and use different definitions. Do not add them into a supposedly exact ledger. Label unexplained memory **unattributed** until process and allocator measurements identify it; a desktop is a candidate, not an answer.

> **WHY THIS IS WORSE THAN IT LOOKS**
>
> A blank reading is more dangerous than a wrong one. A wrong number at least makes you argue with it. `[N/A]` makes you *skip the memory question entirely* and go hunting for a compute problem, a driver bug, a bad cable — anywhere but the one pool that is actually full. Every monitoring dashboard built on scraping `memory.used` inherits this blindness silently: the column is `None` on every row, not because the box is idle but because the field was never populated, and no amount of polishing that table will ever make it answer *what is eating my card*. The instrument that should raise the alarm is the one instrument guaranteed to stay quiet.



Each observed platform hides a different half of the memory report.

The Windows / Linux asymmetry — a callout worth memorising

The same tool lies in *opposite directions* on the two platforms, and knowing which way yours lies is the whole skill:

- **On the GB10 under Linux:** the per-card totals (`memory.used` / `memory.total`) are `[N/A]`, but `nvidia-smi --query-compute-apps=pid,used_memory` returns **real per-process figures**. Ask per-process, and back it with `free -g`.
- **On our Windows box (WDDM):** the per-card total *works*, but `--query-compute-apps` returns `[N/A]` for `used_gpu_memory` on **every** process — the whole column is empty by construction. There is no `free -g` to fall back on either; you attribute entirely from the services (the render UI's memory endpoint, the inference server's model list) plus the "everything else" gap row. This WDDM per-process `[N/A]` behaviour is what we observe on our current driver/OS build; treat it as driver-version-specific — a GPU-driver update could quietly change it, so re-check before relying on it long-term.

NVIDIA documents unsupported aggregate framebuffer memory on this integrated-GPU platform, with per-process memory still listed. Our driver printed `[N/A]`; the exact display can change. A driver/library version mismatch is a different failure and must be diagnosed separately. See NVIDIA's known issues.

Recovery & prevention

1. **When the gauge is blank, change instruments — don't re-run it.** `free -g` first: read `available`, not `free` — but read `swap` in the same glance. On this die a comfortable-looking `available` does not explain a slow job. Here it read 45 while swap read 15/15 and the render crawled; investigate those observations together (Plate 05).
2. **Name every tenant before you blame the card.** Compare the services that report their own footprint with host-wide measurements. Avoid double-counting shared allocations. Leave a gap unattributed until you measure its owner.
3. `free` **beats** `kill`**.** A squatter usually has a `/free` endpoint that releases its memory without a restart — recover the GB before you kill the process and lose its warm state.
4. **Never wire an alarm to a field that reads** `[N/A]`**.** Any dashboard column fed by `memory.used` is structurally blind on this box. Build the memory alarm on `free -g available` (Linux) or OS and service measurements (Windows), and delete the column that was always going to be `None`.

CROSS-REF → Plate 02 (The Stuck-Busy Mismatch — why a busy counter does not prove progress) · Plate 05 (Host Pressure Is GPU Pressure — why `free -g` is the VRAM gauge) · Plate 07 (The Idle Squatter — the 7 GB tenant doing nothing)

PLATE 02 · PART I

THE STUCK-BUSY MISMATCH

Failure class: misread wedge signature · Escape hatch: diagnose by step rate, never by the idle triad

A render is running and you glance at the card to check on it. `nvidia-smi` gives you three numbers that, on any discrete GPU you have ever owned, mean exactly one thing: utilisation pinned at 96%, the SM clock floored near idle, power draw down in the single watts. Our old runbook calls that triad *stuck-busy*. It is a suspicion, not a diagnosis. You are about to declare a wedge and reboot. The numbers are real. The verdict is wrong, and the reason it is wrong is that these fields did not distinguish the two states in our observations.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`utilization.gpu: 96 %` `clocks.current.sm: 721 MHz (max 3003)` `power.draw: ~6 W` Pinned at 96% but clocked at idle and sipping single-digit watts — the classic stuck-busy mismatch. High utilisation *paired with* abnormally low power is a card paging, not computing. This render is wedged.	That exact reading — 96% util, 721 MHz, ~6 W — is what a **perfectly healthy** render produced, ticking along at **56 s/step**, nothing wrong with it at all. These observations do not establish what the clock and power fields were measuring, or why. They only show that our chosen thresholds were not diagnostic. And the clock barely moves between the two states: a healthy render read **721 MHz**, a genuine post-OOM wedge sat at **728 MHz** — a hair apart, both at 96% util. Power stays uselessly low in every figure we have. Measured job progress is the stronger evidence here.

Our discrete-GPU runbook had promoted a heuristic into a verdict. High utilisation, a low reported clock and low power can justify investigation, but they do not establish paging or a stuck kernel on any architecture. A clock may really be low; a sensor may report only one power domain. The unsupported memory-total field in Plate 01 does not prove that these different counters are unsupported too. Keep the observations. Discard the unsupported explanation.

The incident, from the log

A render is running and something about the card feels wrong, so you check the counters — and the triad screams wedge. You go to the card ready to reboot it:

```
$ nvidia-smi --query-gpu=utilization.gpu,clocks.current.sm,clocks.max.sm,power.draw --format=csv
utilization.gpu [%], clocks.current.sm [MHz], clocks.max.sm [MHz], power.draw [W]
96 %, 721 MHz, 3003 MHz, 6 W
# Rounded values from the incident note. Our runbook treated this as a wedge signature.
# A healthy run produced this reading. It does not establish a wedge.
```

Believe that triad and you reboot a working machine. So you stop asking the card how it *feels* and ask the job how far it has *got* — the step rate from the progress bar:

```
$ docker logs --tail 2000 <engine> | tr '\r' '\n' | grep '%|'
 42%|████      |  ...  56s/it
# tqdm writes each update with a CARRIAGE RETURN, not a newline. `docker logs --tail 5` shows
# an unhelpful view of progress. Split \r to \n and check timestamps and new steps.
# 56 s/it is well inside normal for this pass. Not wedged. Just quiet, on a card that can't
# tell you it's busy.
```

Fifty-six seconds per step is healthy. The clock said idle; the bar said working; the bar was right. That is the whole plate in two readings.

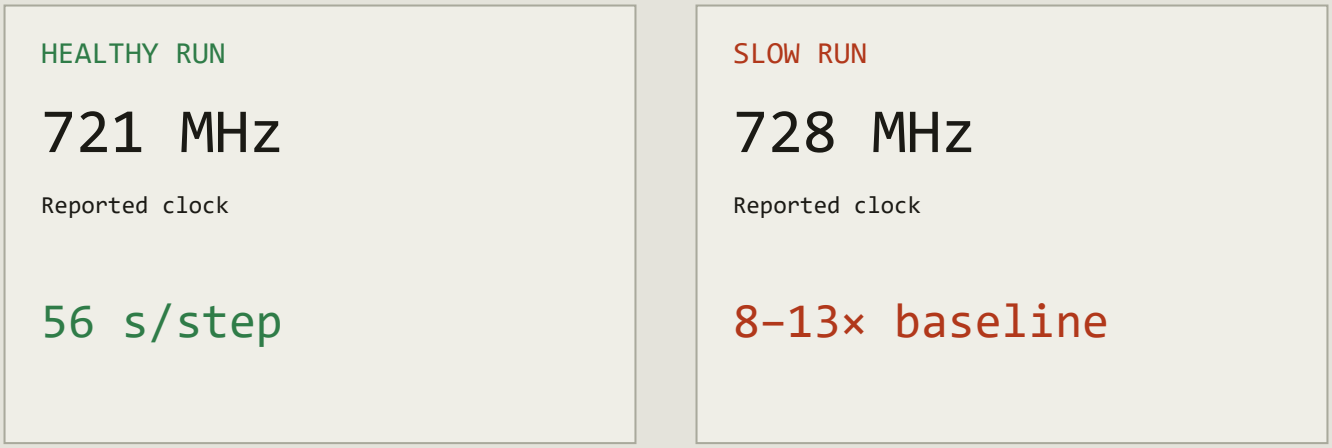
Now the other half — because the wedge is also *real*, it just doesn't announce itself in the triad. On the day it genuinely happened, an inference process and a render collided in the one unified pool and the kernel logged an honest out-of-memory:

```
$ dmesg -T | grep -i nvrm
... NVRM: nvCheckOkFailedNoLog: Check failed: Out of memory [NV_ERR_NO_MEMORY]
# ~6 seconds after the inference process logged "Device 0: NVIDIA GB10 ...". Two tenants,
# one pool, no MIG to wall them off. This is the actual failure (Plate 05).
```

The damage was measured off the render engine's own execution timestamps, not guessed: one clip took **102.9 minutes**, the next **65.3 minutes**, against the ~8-minute warm baseline — an 8–13× penalty, not the "twice as long" it felt like. And the clock? Afterward it sat stuck at **728 MHz** at idle, long after the render finished — a hair off the 721 MHz the *healthy* render had shown. Similar clock readings, different measured performance.

> **WHY THE INSTRUMENT CANNOT TELL THEM APART**
>
> Our records show 721 MHz during a healthy render and 728 MHz after a severe slowdown. They also record about 6 W during healthy work and 21 W on a separately troubled worker. These are different observations, not a controlled sensor calibration. We cannot derive a universal threshold or declare the sensors meaningless. We can reject the threshold that would have condemned the healthy run. Compare the same workload's elapsed time, step progress and output before deciding what to repair.

The trap has a second edge, and it bit on recovery. After applying the first-aid free (`curl -X POST localhost:PORT/free -d '{"unload_models":true,"free_memory":true}'`), the idle clock *still* read 728 MHz — so by the clock, the wedge had not cleared. It had. Proof came from behaviour, not the counter: a fresh known render completed in **7.2 minutes** against its ~9-minute baseline, output verified real end to end. The low clock did not confirm the fix. **Behaviour under a fresh load is the test; the idle reading is not.**



Similar clocks. Different measured work. Separate observations.

Similar reported clocks accompanied very different job progress.

Recovery & prevention

1. **Diagnose a wedge by step rate, never by the idle triad.** Read the progress bar, splitting carriage returns first: `docker logs --tail 2000 <engine> | tr '\r' '\n' | grep '%|'`. A short `--tail 5` can conceal progress; compare timestamps and the engine state.
2. **If no bar has flushed yet, prove liveness another way** — the engine's queue endpoint showing the job *running*, plus a hand-timed wall-clock delta between two steps. Do not read "no new line" as "hung"; buffering and carriage-return rendering can hide updates; flushing behaviour depends on the process and its output stream.

3. **The real test that a wedge cleared is behaviour under a FRESH load, not the idle reading.** The SM clock stayed at 728 MHz after the fix that worked. Run a small known job and time it; trust the stopwatch, not the counter.
4. **Know the actual cause so you fix that, not the symptom.** In this incident, the evidence pointed to memory pressure — two tenants sharing one pool with no MIG (Plate 05), confirmed by `NV_ERR_NO_MEMORY` in `dmesg`. The cure is *free first, then load* (Plate 09), not a reboot on the strength of a clock reading.

There is no GB10 wall-power baseline in this record, and no experiment establishing the physical meaning of the reported power values. We therefore do not convert six watts into a claim about the whole machine. A future driver or firmware update may change both counters and performance. Record that software level with your baseline and repeat the small reference job after an update. A useful instrument is one you have calibrated against work. The name of the counter is not its calibration.

CROSS-REF → Plate 01 (The N/A Card — the unsupported memory-total field) · Plate 05 (Host Pressure Is GPU Pressure — the unified-pool overcommit that is the real wedge) · Plate 07 (The Idle Squatter — the 265 s/step crawl that reads as a wedge but is contention) · Plate 09 (Free First, Then Load — the ordering rule that prevents the collision)

PLATE 03 · PART I

PING IS NOT HEALTH

Failure class: false liveness · Escape hatch: health-check the layer that fails — a real fork, a real device-touching endpoint

You loaded a model that didn't fit and the box went quiet. You ping it out of reflex, and it answers — instantly, 0% loss, sub-millisecond. So the box is *up*. You ssh in to fix it, and the connection dies before it prints a prompt. You ping again: still perfect. The two facts sit on your screen refusing to agree, and the wrong one is the one that feels most like proof.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`5 packets transmitted, 5 received, 0% packet loss` `rtt ... time=0.26 ms` The box is up. It is on the network, it is answering, latency is perfect. Whatever is wrong, it isn't the machine — go check the cable, the bridge, the service config.	`kex_exchange_identification: Connection closed by remote host` The host's network stack is responsive. ICMP does not test whether the required application can work. The required remote shell and services did not respond in this incident. It is reachable and completely unavailable at the same time. Ping tested a layer that never went down.

An echo reply exercises the network stack. It does not require a successful application request or a new login shell. Do not turn that narrow success into a statement about every process on the machine. Conversely, a failed SSH handshake does not by itself prove memory exhaustion. In this incident it arrived immediately after an overcommitting load, alongside failed service requests and heavy disk activity. That combination, rather than a single error string, supported the pressure diagnosis.

The incident, from the log

A large inference host had a resident model holding **109 GB of a 121 GB** unified pool. `free -g` read **12 GB available**. A second model — **17.4 GB** — was loaded into that 12 GB to test a service slot. Both numbers had been on-screen two tool calls earlier (that ordering failure is its own plate — see Plate 09). The box went into swap thrash and did not come out.

The first instrument said everything was fine:

```
$ ping -c 5 <box>          # over the private cluster link, bridge bypassed
64 bytes from <box>: icmp_seq=1 ttl=64 time=0.26 ms
... # further replies omitted
--- <box> ping statistics ---
5 packets transmitted, 5 received, 0% packet loss
# 0.26 ms, zero loss. The kernel is answering from softirq.
# This tells you NOTHING about userspace.
```

The second instrument told the truth, and it was the opposite truth:

```
$ ssh <box>
kex_exchange_identification: Connection closed by remote host
# The record describes a banner-exchange failure and attributes it to pressure.
# This error alone does not establish a failed fork or its cause.
# (This is OpenSSH's standard pre-banner error — what the client prints when the remote
#  closes during key exchange. Our own log recorded this failure as "sshd dead at banner
#  exchange, cannot fork"; the line above is the canonical form of that same failure,
#  not a captured transcript.)
```

The recorded service probes failed too; the commands below show the equivalent check with a bounded timeout:

```
# inference server
$ curl --max-time 10 -sS -o /dev/null -w '%{http_code}\n' http://<box>:11434/api/tags
000
# second server
$ curl --max-time 10 -sS -o /dev/null -w '%{http_code}\n' http://<box>:8001/v1/models
000
# 000 is curl’s no-HTTP-status report, not an HTTP code. Inspect its error and exit status.
# DNS, TLS, connection failure and timeout can all produce it. Direct probes bypassed
# the bridge; they did not by themselves rule out every network fault.
```

Three instruments now said *dead* and one said *perfect*. The one that said perfect was the one that tested a layer which had never failed. Thirty minutes of waiting produced no recovery (Plate 06). Our remote escape hatches were ssh, a bridge daemon, and a repair deck, all of which terminate in a shell *on the far side*. When userspace is dead there is no software path in. It ended in a hard power cycle.

> **WHY THIS IS WORSE THAN IT LOOKS**
>
> A host that stops answering everything at least makes the loss of service obvious. A box that answers ping at 0.26 ms while unable to fork a shell steals your time in a specific way — it sends you looking everywhere the machine *isn't*. You check the bridge, the switch, the service config, the firewall, DNS, the port. Ping keeps telling you "the box is fine," so you keep believing the fault is between you and it. The false liveness is not a harmless extra reading; it is an active pointer aimed away from the answer. In our incident the bad state persisted for thirty minutes. File-backed reclaim and refaulting were the recorded explanation, not a proved rule that all memory-mapped loads evade the OOM killer. The practical lesson is to set a recovery deadline and use independent evidence; neither a ping reply nor an absent OOM line is a promise of recovery.

Health-check the layer that fails

The rule the incident teaches: a health check is only worth anything if it *touches the layer that breaks*. Ping touches the kernel network stack — the layer least likely to fail on an overloaded box, which is exactly why it lies. A real check forces the machine to do the thing you need it to do.

- **To prove userspace can run: force a fork.** `ssh -o BatchMode=yes -o ConnectTimeout=5 <box> true` returning 0 demonstrates a remote command completed. A failure has several possible causes; inspect the error and correlate it with other probes. A successful remote command tests more than a bare ping.
- **To prove a service is alive, hit an endpoint that reaches the device, not the router.** A web server's `/` can be served by the HTTP front-end while the worker behind it is wedged — we have watched a render backend accept a submission and hand back a real job id in 82 ms while its GPU worker was dead. The probe that caught it was the endpoint that touches the torch/device layer (`/system_stats`), which returned HTTP 500 the whole time `/` returned 200. **Health-check the deepest layer you depend on, not the shallowest one that answers.** That `/system_stats` path is specific to the class of render backend we run; on your own software, find the endpoint that forces the device layer to answer — not `/` — rather than assuming this literal path exists.

Recovery & prevention

1. **When ping and ssh disagree, believe ssh.** Ping proves the kernel schedules interrupts and nothing more. If `ssh <box> true` cannot return an exit code, the box is unavailable regardless of what ICMP says. A banner-exchange failure is not an OOM diagnosis by itself; authentication policy, connection limits and network failures also need consideration.

2. **Know your out-of-band hatch before you need it.** Every remote path — ssh, a bridge daemon, a repair dashboard — ends in a shell on the far side and is useless once userspace dies. Our available remote paths failed together. Prepare an independent recovery path: a smart plug, a PDU, IPMI/BMC, or hands on the power button. Wire one up *before* you run a risky load, not after the box goes silent.
3. **Build liveness alarms on a fork, not a ping.** Any monitor that pages green on ICMP will call this box healthy at the exact moment it is most broken. Make the health check run a command and read an exit code, and make the service check hit a device-touching endpoint.
4. **Prevent the state upstream.** This limbo is what an over-commit *produces*; the fix is to not over-commit. Read `available` (not `free`), compare it to a runtime budget including model and context, and free the resident giant first (Plate 09). Ping being green afterward is not confirmation the load worked — it never went red to begin with.

CROSS-REF → Plate 06 (The Wedge That Needs a Power Cycle — the mmap thrash that produces this ping-alive/fork-dead state and refuses to self-heal) · Plate 09 (Free First, Then Load — the ordering rule that prevents the over-commit) · Plate 10 (The Watcher Goes Away — its mirror image: there the poller dies while the work *succeeds*; here the box answers while the work cannot even start)

PLATE 04 · PART I

SILENT SUCCESS

Failure class: the empty result that files itself as done · Escape hatch: assert non-empty, budget for the hidden reasoning, fail honestly

You point a vision-analysis call at a folder of six images and walk away. It comes back green: every call returned `success: true`, no error in the log, the catalogue written and closed. Then you open the catalogue. The first image has a description. The second has a description. The others are blank — not "error," not "skipped," just empty strings sitting in the `analysis` field where a paragraph should be. Nothing failed. Nothing warned you. Several results are simply not there, and the tool is certain the job is done.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`{"success": true, "analysis": ""}` The call completed. The status field says `success` — no exception, no timeout, nothing thrown. The description just happens to be an empty string, and an empty string is a valid value. Write it to the record and move on.	The model never emitted an answer at all. The output-token budget is shared with the model's own hidden reasoning, and the reasoning ate the entire budget before a single content token was written. The call really did run — that part isn't a lie — but "ran without error" and "produced a description" are two different facts, and only the first is true. The artifact is empty because the answer was never born, not because the answer was short.

On the reasoning-enabled builds in this incident, the output budget had to cover reasoning as well as the visible answer. Parameter names and accounting differ across APIs and model templates; inspect the returned usage and stopping fields. A clean transport response does not guarantee that a model supplied usable content. Our wrapper needed a separate artifact check, not another interpretation of HTTP success.

The incident, from the log

The report was precise: a cataloguing run "will properly identify the first or second picture, but then just fill in the rest as blank." That shape — the first one or two succeed, the rest go empty — is the tell. The shape prompted a budget test; it did not establish a cache-related cause.

The vision slot on our box was a Qwen3-family vision build, and this class of model *thinks* before it answers. The behaviour — reasoning consuming a shared output budget — applies where a runtime counts reasoning and answer tokens together, but the character counts below are specific to *this build*, not a universal constant. Measured on our box, it spends **1,500–7,000 characters** deliberating before the first character of the actual description. The output caps in force at the time were 300 / 500 / 800 tokens. For most images the thinking alone was longer than the whole budget:

```
# Same six-image folder, only num_predict changed. Count of blank descriptions:
num_predict = 800  -> 3 of 6 blank   # thinking overran the budget on half the images
num_predict = 2000 -> 0 of 6 blank   # budget finally clears the reasoning; every answer lands
# The images did not change. The prompt did not change. Only the ceiling on generated tokens.
```

The obvious escape — turn the thinking off — did not work:

```
# Request sent with think=False. The model thought anyway:
reasoning length = 3474 chars   # with the flag explicitly set
content          = ""           # still empty
# This abliterated build ignores the flag. The budget is the only lever that moves.
```

Version-fragile. The flag being ignored is specific to *this abliterated build*; a stock build may honour `think=False`, in which case disabling the reasoning is a second lever. On this one the budget was the only lever that moved. Re-check after any model or runtime update.

The same mechanism reaches all the way up to the largest model on the property. Health-probing a 122B-class reasoning server through its OpenAI-compatible endpoint with a deliberately tiny `max_tokens: 20` returned this:

```
$ curl .../v1/chat/completions -d '{"messages":[...],"max_tokens":20}'
{"choices":[{"message":{"content":""},"finish_reason":"stop"}], ...}
# content empty, finish_reason "stop" — the CLEAN ending, not "length". That is the surprise:
# a 20-token cap should truncate, yet the server reports a normal stop with nothing produced.
# It is not dead: 196 completion tokens went to the reasoning field before any content.
# Re-probed at max_tokens:200 -> it answered correctly.
```

Here `finish_reason: stop` — no truncation flag — does the same damage the empty `analysis` string did above, and it is more convincing because it looks so clean. This tiny probe returned no answer; it was inadequate evidence of a dead server. The recorded 196 completion tokens also exceed the requested cap of 20; that discrepancy means this response is not a clean demonstration of budget accounting. The larger probe succeeded. Keep the empty-result observation without pretending the runtime semantics are fully explained.

The stop reason is runtime-dependent. The clean `finish_reason: stop` on an empty completion is what the vLLM-served server returned here. An Ollama call that hits `num_predict` instead surfaces `done_reason: "length"` — a real truncation flag in a differently named field. Which signal you get, and whether it even admits truncation, depends on your runtime, so read your own server's stop reason rather than trusting either value as universal.

> **WHY IT NEVER SELF-HEALS**
>
> A blank counts as done. That is the whole trap. The cataloguing tool wrote each empty string into its record as a completed entry, and *a completed entry is never retried* — so on the next run the blank entries are skipped as already-processed, and the gap becomes permanent. It is not a transient that clears when you re-run; re-running is precisely what preserves it. This is the same family as every silent-artifact failure in this book: the status field is green, the log is clean, and the thing you actually wanted was never produced. No error will ever lead you here. The only evidence is the artifact itself — which is why you find it weeks later, by opening the file, not by reading the log.

Recovery & prevention

1. **Never return empty as success.** A call that produces no content is a *failure*, whatever the status field says. Wrap the call so an empty result raises, retries, or is filed as an explicit error — never as a done record. On our box the fix was a wrapper that retries once at 2.5× the budget and hard-fails if content is still empty, plus a catalogue writer that refuses to file a blank so the next run retries it.
2. **Budget for the reasoning, not just the answer.** Where the runtime shares one generated-token budget, it must cover reasoning and the answer. Measure representative prompts and leave room for both reasoning and the requested answer. On our vision build, answers only became reliable at `num_predict = 2000`; the 300-token budget was unreliable for the observed reasoning lengths. That 2000 is *this build's* working floor, not a universal constant — measure your own model rather than inheriting ours.
3. **Assert on the content, never the status.** Read the returned string and check it is non-empty, complete and fit for its intended use before recording the call as done. `success: true` and `finish_reason: stop` are both true and both useless as proof of an artifact.
4. **When a whole class of output is mysteriously empty, count the artifacts before theorising.** Counting blank entries made the failure measurable. The evidence is on disk, not in the log.
5. **On a health probe, give a reasoning model room to think — and read the reasoning field.** A probe with `max_tokens` below the model's thinking length reports a healthy model as dead. Give any reasoning-model probe enough budget to clear the deliberation (≥ ~200 tokens on the model here) and inspect the reasoning field, not just `content`, before concluding anything is wrong.

Scope. The "generated-token budget includes the hidden reasoning" behaviour is confirmed here on two runtimes — Ollama (`num_predict`) and our OpenAI-compatible server (`max_tokens`) — but *only with reasoning-emitting models*. A non-reasoning model can also return empty or incomplete content for other reasons. It does not have this particular reasoning-budget mechanism. Confirm which kind of model you are on before quoting this as universal.

CROSS-REF → Plate 10 (The Watcher Goes Away — the other way a real absence-of-result reads as done; check the engine's own artifact, not the status) · Plate 09 (Free First, Then Load — the same discipline of verifying the real thing came back before trusting the report)

PART 2 / PLATES 05–08

II

ONE POOL, ONE PRESSURE

The compiler, the model and the render all need room. Measure residency, pressure and work before assigning a cause.

PLATE 05 · PART II

HOST PRESSURE IS GPU PRESSURE

Failure class: two-budget thinking on a shared-memory machine · Escape hatch: measure headroom, pressure and progress together

The render is crawling at 265 seconds per step. You check GPU utilisation, then check host memory, as though the two instruments describe separate budgets. One says busy. The other says there is room. On this machine the allocations draw from the same physical memory, but neither number explains how much useful work is getting done. You need a third question: what is the machine spending its time waiting for?

THE LIE / THE TRUTH

THE LIE — THE CONVENIENT READING	THE TRUTH — THE MEASURED LIMITS
`GPU: 96% · available: 45 · swap: 15/15` The GPU is busy and the host has headroom. Alternatively, swap is full, so the machine must be thrashing. Pick whichever number agrees with your first guess.	`render: 265 s/step` The job is slow. Those occupancy counters are context, not a complete diagnosis. Host and GPU workloads share capacity; current reclaim, I/O and job progress tell you whether that shared capacity is under harmful pressure.

The architecture matters because a host allocation can reduce room for a GPU workload. It does not make every GPU failure a memory failure. A kernel bug, poor batching, a slow disk or an expensive computation can still be the bottleneck. Shared memory changes the budget; it does not abolish diagnosis.

The incident, from the log

The recorded host readout and the render's pace looked like this. The table is a reconstruction from rounded incident values, not a byte-exact transcript:

```
$ free -g
              total  used  free  shared  buff/cache  available
Mem:            121    75     2       0          44         45
Swap:            15    15     0
# free -g displays GiB-scale integer values. Keep the original units.
# available is an estimate; full swap is occupancy, not a rate.

# Job progress recorded separately:
265 s/step
```

A second render engine had an empty queue while retaining **7,043 MiB** in the process view. A graphical session was also open. During recovery, the engine's footprint fell to **355 MiB**, available memory rose from **45 to 79**, and swap occupancy fell from **15/15 to 8/15**. The stalled render finished.

That is useful evidence for relieving contention. It is not a balanced accounting sheet. A roughly 6.5 GiB process reduction cannot by itself explain a 34 GiB rise in the host estimate. Other allocations, page-cache state and the timing of the readings matter. The record supports the intervention and the recovery; it does not isolate every byte returned by every action.

Read an estimate as an estimate

`MemAvailable` estimates memory that can be supplied without swapping. It includes some reclaimable memory as well as completely unused memory. Clean file-cache pages can be discarded and read again from their backing files; they do not require an empty swap slot. Full swap therefore does **not** make all reclaim impossible or invalidate `available` automatically. Linux memory concepts distinguish file-backed and anonymous memory.

The opposite mistake is spending the entire estimate. A future model load needs weights, context storage, runtime buffers and transient allocations. Some memory is pinned or otherwise constrained, and device allocators have their own reporting limits. A host estimate is a planning input, not a guarantee that a particular CUDA allocation will succeed.

> **WHY OCCUPANCY IS NOT PRESSURE**
>
> A swap area can remain occupied after the original pressure has passed. A snapshot tells you that pages are there; it does not tell you whether the machine is currently shuttling them in and out. Likewise, high memory use may be a productive cache or an oversized working set. Diagnose from movement and delay: swap-in and swap-out rates, memory-stall time, I/O latency and the job's progress. Then compare the same signals after a controlled release.

The following is a diagnostic recipe, not additional evidence from our historical run. It asks questions the old record should have answered:

```
$ vmstat 1 5
# Read interval samples, not only the first since-boot summary.
# si/so show swap traffic; blocked work and CPU wait add context.

$ cat /proc/pressure/memory
# Memory stall time where the kernel exposes PSI.
# Compare before/after and alongside job progress.

$ swapon --show
# Which swap areas exist, and how much of each is occupied?
```

Reclaiming clean model pages can cause heavy refault I/O without large swap-in/out counters. Zero `si` and `so` is therefore not a certificate of healthy memory behaviour either. Record major faults, storage activity and the workload's own timing when the simpler view leaves a gap. PSI measures stalls, not their entire causal story. See the kernel PSI documentation.

Recovery & prevention

1. **Name the workload and its baseline.** A 265-second step is meaningful only against comparable work. Hold model, resolution, settings and warm/cold state steady.
2. **Read the shared pool.** Start with `free -g`, then add interval pressure and I/O measurements. Treat full swap as a reason to investigate, not a verdict.
3. **Identify an idle tenant before releasing it.** Check its queue, model residency and process identity. Prefer its documented unload operation when available. Confirm the intended service changed.
4. **Measure recovery at both layers.** More available memory demonstrates headroom. Faster steps and a valid output demonstrate a recovered job. One does not substitute for the other.
5. **Budget the whole workload.** Leave room for context, temporary buffers, the OS and the other services that must remain live. An on-disk weight file is only part of that budget.

The pool is shared. The counters are partial. Read enough of them to explain the work.

CROSS-REF → Plate 01 (unsupported memory totals) · Plate 06 (the load that did not recover) · Plate 07 (an idle engine retaining memory) · Plate 09 (release, verify, load) · Plate 11 (bandwidth and decode)

PLATE 06 · PART II

THE WEDGE THAT NEEDS A POWER CYCLE

Failure class: severe pressure with no usable remote recovery · Escape hatch: independent access, then power recovery if necessary

You had the numbers on your screen. Available memory read 12. The model you were about to load was listed at 17.4 GB on disk. A large model was already resident. You submitted the test anyway. The box kept answering ping while every useful remote path stopped working, and thirty minutes later you were still looking at the same reassuring network reply.

THE LIE / THE TRUTH

THE LIE — THE REASSURANCE	THE TRUTH — THE RECORDED OUTCOME
`ping: 0% loss · 0.26 ms` No reported OOM rescue. The machine is alive; wait for the load to sort itself out.	`30 minutes · no recovery` SSH failed during banner exchange, direct service probes supplied no HTTP response, and disk activity continued. Our remote recovery tools depended on those same broken paths. The incident ended in a hard power cycle.

That outcome is the evidence. The explanation needs more care than the original runbook gave it. The source note attributed the persistent state to reclaim and refaulting of memory-mapped model weights. It did not preserve a kernel trace that proves every step of that mechanism. A faithful account keeps the explanation as an inference and the thirty-minute failure as an observation.

The incident, from the log

On August 7, a large inference process had been resident for nine days on a host whose memory tool reported a total near **121 GiB**. The note records roughly **109 GB** held by that serving workload and **12** in `free -g`'s available column. The container had no configured Docker memory limit. A second model, listed at **17.4 GB**, was requested through another server.

The mixed units do not rescue the decision. Converting decimal 17.4 GB to GiB still leaves the weight file larger than the reported available memory, before context storage and load-time overhead. Disk size is not a runtime allocation guarantee; here it was already enough to show that the load had not been provisioned responsibly.

```
# Reconstructed ledger; not a simultaneous terminal transcript.
host memory tool total:       121 GiB-scale units
available before request:      12 GiB-scale units
challenger weight file:        17.4 GB, as recorded
resident age:                   9 days
direct ping:                    0% loss, about 0.26 ms
remote login:                   failed at banner exchange
direct HTTP probes:             no HTTP response
recovery during observation:    none, over 30 minutes
eventual recovery:              hard power cycle
```

The recorded SSH symptom was paraphrased as an inability to fork. That is the operator's diagnosis, not a captured allocation error. Similarly, curl's `000` is a local report that no HTTP status was received; it is not a status the server sent. A single failed handshake can have other causes. The simultaneous loss of useful service after a badly provisioned load is what makes the pressure explanation compelling here.

What reclaim can do to a working set

A clean file-backed page can be evicted and later read again. If a workload repeatedly needs more resident data than the system can keep, eviction and refaulting can consume much of the available time. The machine may remain technically active while useful work makes little progress.

That is **not** a rule that `mmap` disables the OOM killer. File-backed workloads can encounter OOM handling, and `mmap` can create anonymous mappings too. Nor does an anonymous allocation guarantee that the kernel will kill one process promptly and return a healthy machine. Reclaim, swap, cgroup limits, pinned allocations and driver behaviour all affect the outcome. The Linux memory-management overview supplies the mechanism; our record supplies one failure, not a universal prognosis.

> **WHY WAITING IS NOT A RECOVERY PLAN**
>
> An absent OOM message is not proof of spare memory. It is also not proof of an infinite, unkillable state. In our incident no rescue was observed during the window we waited. The error was treating the possibility of eventual recovery as an operational plan, with no deadline and no independent way to intervene. Decide in advance how long the work may stall and which access path remains available if ordinary logins fail.

The compile incident in Plate 08 is a useful counterexample: access returned and uptime proved there had been no reboot. It prevents this plate from becoming another misleading rule. Similar symptoms can have different outcomes. Record the outcome you actually observe instead of assigning it from the loader's file format.

Recovery & prevention

1. **Stop adding work.** Repeated submissions can deepen pressure and obscure which request caused it. Preserve the job IDs and the last known good memory reading.
2. **Check an existing session or local console.** A working shell may still allow the offending workload to be stopped cleanly. Do not assume that every path failed because one new SSH connection did.
3. **Use a bounded recovery window.** Correlate service checks, workload progress and pressure. If useful access remains unavailable, use the independent management or physical recovery path prepared for this host. A hard power cut risks unsaved state; it was our last available recovery, not the first diagnostic command.
4. **Release before the next experiment.** Stop or unload the conflicting resident, verify recovered headroom, then admit the replacement workload with allowance for context and transient buffers. Plate 09 makes this order explicit.
5. **Validate limits instead of assuming them.** Docker memory limits can constrain charged container memory; they are not a demonstrated reservation for every CUDA allocation on every driver. A limit may cause allocation failure or a container OOM. Test the actual stack under a controlled load. See Docker resource constraints.

The missing OOM message did not make the host healthy. The power cycle did not prove a general law about mapped files. Both facts belong in the record.

CROSS-REF → Plate 03 (network reachability versus useful service) · Plate 05 (occupancy and pressure) · Plate 08 (a different observed recovery) · Plate 09 (admission order)

PLATE 07 · PART II

THE IDLE SQUATTER

Failure class: silent VRAM squat · Escape hatch: the engine's own `/free`, before the render, not after

A render that finishes warm in about ten minutes is forty minutes in and still crawling. You go to the render engine and ask it what it's doing: nothing. Its queue is empty — no job running, none pending. Every instrument agrees the box is free. So why is the render dying? Because a second service that stopped working hours ago never gave its memory back, and on this die the memory it is squatting is the same memory your render needs.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`queue running:0 pending:0` The engine has nothing queued and nothing running. It is idle. The card is free. There is no reason a fresh render should be slow — go ahead and submit.	`<pid>, 7043 MiB # that same "idle" engine` An empty queue is not an empty card. The service finished its last job and kept **7,043 MiB** resident — model weights it will not touch again until you queue something. On a unified die that 7 GB is not idling harmlessly on a spare card; it is holding down the one pool your render is trying to allocate from, and it has driven the box into swap.

"Queue empty" and "card empty" are two different facts, and the gap between them is this whole plate. A render engine reports on its *work* — jobs running, jobs waiting. It does not report on its *footprint*, and most of them hold their loaded weights resident between jobs on purpose, so the next submission starts warm. That is a reasonable optimisation on a machine with a dedicated card doing one thing. On a GB10, where every service draws from the same pool as every other service and as the host itself (Plate 05), an idle engine holding several GB is not warm — it is a squatter, and it is charging rent in render time.

The incident, from the log

The law was first measured on a Windows workstation with a discrete 16 GB card, as a separate comparison; discrete GPUs can also page or offload under pressure. The same clip was rendered twice, back to back, the only difference being whether a second render UI was left resident and idle:

```
# clean — the idle render UI unloaded first:
10.8 min
# resident — the same UI sitting idle, holding a checkpoint:
40.6 min
# ~4x. Identical job, identical settings. The only variable was 5 GB of idle squat.
```

The idle UI was holding roughly 5 GB, and with it resident the card sat pinned at **15,485 / 16,303 MiB** at 100% the whole render — no headroom, the render fighting for every allocation. Recovering it did not require stopping the service:

```
> curl -X POST http://127.0.0.1:7860/sdapi/v1/unload-checkpoint
# drops the resident checkpoint (~5 GB) WITHOUT stopping the service.
# It reloads on its next generation, so you lose nothing but the warm start.
```

A ~4× penalty on a discrete card is bad. On the GB10 it is worse, because the squat does not stop at the card — it reaches into the one pool and hits swap. A render on the Linux box looked outright locked up, crawling at **265 s/step**. It was not wedged (that misread is Plate 02's job); it was starving. A second render engine had finished its work and gone quiet:

```
$ nvidia-smi --query-compute-apps=pid,used_memory --format=csv
pid, used_gpu_memory [MiB]
<pid>, 7043 MiB     # a second render engine, queue running:0 pending:0 — idle, holding 7 GB
# On the GB10 under Linux the PER-PROCESS query works
# (the per-card totals are [N/A] — Plate 01).
# This one line names the squatter. On Windows this column is [N/A]
# and you attribute by service.
```

```
$ free -g
              total        used        free      shared  buff/cache   available
Mem:            121          75           2           0          44          45
Swap:            15          15           0
# full swap in the recorded slow run.
# Occupancy alone is not a thrashing diagnosis. Compare pressure and render progress.
# The idle engine is one measured tenant, not a complete accounting of the host.
```

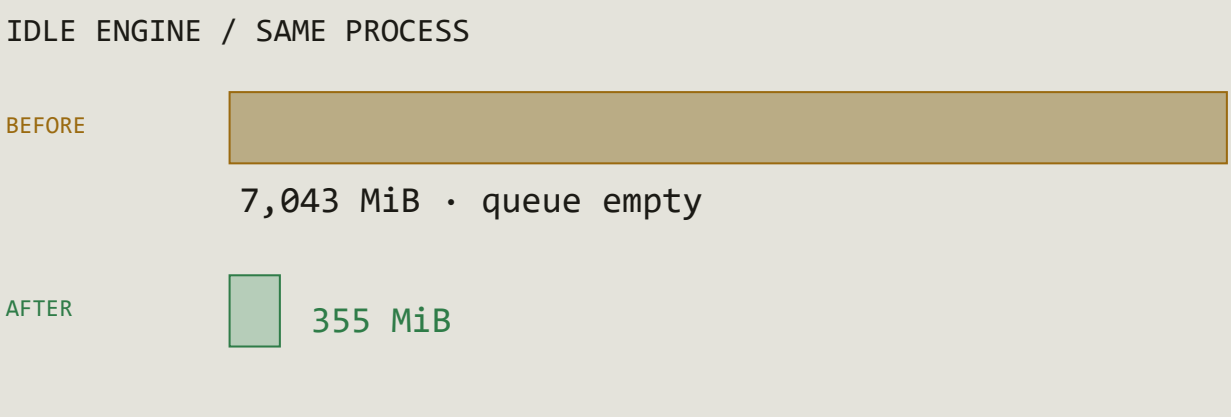
The host was also running a desktop session. During the recovery, the record reports the idle engine falling from 7,043 MiB to 355 MiB, `available` rising from 45 to 79, and swap occupancy falling from 15/15 to 8/15. The stalled render completed. Those deltas do not balance to a single 7 GB release; they are host observations over the recovery interval, not proof that one endpoint returned 34 GB.

```
# Equivalent request; PORT is the measured engine endpoint, not a literal port number.
$ curl -X POST http://localhost:PORT/free \
    -H 'Content-Type: application/json' \
    -d '{"unload_models":true,"free_memory":true}'
# Verify the target engine is idle before unloading.
# Check its footprint and the running job afterward.
```

Both endpoint paths above are what our specific render-UI and render-engine builds expose — a reader on different software needs the equivalent "unload / free memory" call for their engine, not these literal paths. The *shape* is what generalises: some resident engines have a way to drop weights without a restart, and finding it is the difference between recovering the GB and killing the process (and its warm state) to get them back.

WHY THIS NEVER SHOWS UP WHERE YOU'RE LOOKING

The squatter is invisible to every instrument you'd naturally check. Its own queue says idle — truthfully, it *is* idle. `nvidia-smi` 's per-card memory gauge reads `[N/A]` on this die (Plate 01), so it can't show you the 7 GB. `utilization.gpu` reads ~96% off the *victim* render and tells you nothing about *who* is holding memory. The only instruments that name the culprit are the ones you have to think to run: `free -g` for the pressure, `--query-compute-apps` for the per-process footprint (Linux only), and the idle engine's own memory endpoint. A squat produces no error, no log line, no alarm — just a render that is inexplicably four times too slow, on a box that swears it's free.



The idle engine retained memory after its queue emptied.

Recovery & prevention

1. **Find the squatter, don't guess it.** On the Linux GB10: `nvidia-smi --query-compute-apps=pid,used_memory --format=csv` plus `free -g` — an engine holding big VRAM with an **empty queue** is the signature. On Windows the per-process column reads `[N/A]` on our current WDDM driver build — attribute from each service's own memory endpoint instead (Plate 01). Treat that `[N/A]` as driver-version-specific: an update could change it, so re-check before relying on it.
2. `free` **beats** `kill`**.** Call the engine's unload/free endpoint to drop its weights without stopping it (`.../unload-checkpoint` on the render UI; `POST /free` with `{"unload_models":true,"free_memory":true}` on the render engine). You recover the GB and keep the service alive; it reloads on its next real job. Both paths are our specific builds' — find the equivalent unload/free call for your own engine and version.
3. **Then read the pool, not the gauge.** Confirm the recovery on `free -g` **available** and **swap** — a rise in `available` is evidence of released headroom; restored job progress demonstrates recovery (Plate 05). The per-card memory gauge will still read `[N/A]` and confirm nothing.
4. **The ordering rule: free the other engines BEFORE a big render, not after it stalls.** Generate all your stills first, unload the render UI, *then* start the video render; free every other resident engine before a heavy job on the shared die. This is the same discipline as *free first, then load* for models (Plate 09) — an idle render engine is a resident model by another name. These figures are incidents, not constants: the ~4× (10.8 vs 40.6 min) is from our 16 GB discrete card, the swap-starvation from our GB10 — expect the shape, not the exact numbers, on your own box.

An idle engine holding memory is the most polite failure in this book. It breaks nothing, logs nothing, and answers every question you ask it honestly — *"am I running a job?"* "No." The question it never volunteers, and the only one that matters, is *"am I still holding the card?"* On a machine where the card and the memory and the host are one pool, that is the question you have to learn to ask before every render.

CROSS-REF → Plate 01 (The N/A Card — why per-card memory is blank, and per-process works only on Linux) · Plate 02 (The Stuck-Busy Mismatch — the 265 s/step crawl reads like a wedge; it's contention) · Plate 05 (Host Pressure Is GPU Pressure — one pool, occupancy and pressure are different measurements) · Plate 09 (Free First, Then Load — the same ordering rule, for models)

PLATE 08 · PART II

THE COMPILE IS LOAD TOO

Failure class: a build treated as "not a GPU job" · Escape hatch: cap `MAX_JOBS`, free residents first, and kill the tree correctly

You needed to build a CUDA extension. There was a large model server resident, two render engines up, a desktop session logged in, and the disk at 92% — but a compile is just the CPU chewing on source, so you set `MAX_JOBS=8` to get it over with and walked away. Within minutes the box was gone. Not slow: gone, in the exact way a bad model load takes it — ping perfect, `sshd` dead at the banner, every service port accepting the connection and then answering nothing.

THE LIE / THE TRUTH

THE LIE — WHAT THE INSTRUMENT REPORTS	THE TRUTH — WHAT IS PHYSICALLY HAPPENING
`nvidia-smi --query-compute-apps=pid,used_memory` → (build absent) The compilers never appear on the GPU tool — `nvcc`, `cicc`, `ptxas` hold no CUDA context while they compile. By every GPU instrument the build is invisible. It is *"only a build, not a GPU job,"* so it is safe to run alongside the resident model. The card isn't even involved.	`ps --sort=-rss -eo pid,rss,comm` → 19 × cicc, ~6 GB each The compile is a memory load like any other. Each parallel compiler process is several GB of **host** RAM, and on a unified die host RAM *is* the pool the model lives in (Plate 05). `MAX_JOBS=8` — and the deeper tree it spawns — is a tens-of-GB spike arriving all at once into the same pool. It drove the box into the same swap wedge a too-big model load does.

The GPU tool is telling the truth about its own question and it is the wrong question. "Is this process holding GPU memory?" — no, a compiler holds none. "Is this process about to exhaust the one pool the GPU draws from?" — that is the question, and no GPU counter is watching it. The whole error is the word *only*: even a discrete-GPU workload needs host memory and CPU resources, so a compile can contend there too. On the GB10 they are the same bytes. A build is not a lighter thing than a model load. It is a model load with a different name.

The incident, from the log

One box, unified memory, already carrying a resident model server (~21 GB), two render engines, and a graphical session, at 92% disk. A CUDA extension was kicked off with `MAX_JOBS=8`. Within minutes the box wedged. The probes read exactly like Plate 06:

```
$ ping -c 4 <box>
64 bytes from <box>: icmp_seq=1 ttl=64 time=<1 ms
4 packets transmitted, 4 received, 0% packet loss        # kernel alive, on the wire
# Our log for THIS incident recorded the ping only qualitatively ("perfect"); the shape
# shown here reproduces the same wedge signature measured exactly in Plate 06.
# It is not a captured transcript.

$ ssh <box>
kex_exchange_identification: Connection closed by remote host   # socket accepted, then no fork
# OpenSSH's canonical pre-banner error — the client's form of "sshd could not fork a session."
# Our log recorded it as "sshd dying at banner exchange"; this is that
# failure's standard string, not a captured transcript.

$ curl -s http://<box>:<svc>/            # a resident service port
# connection accepted, request sent, no bytes returned — it hangs. "up" on the TCP handshake,
# dead on the application. Reachability is not availability (Plate 03).
```

Five minutes of patient retries bought nothing, and the on-call reflex — the one Plate 06 trains — was *this is the mmap wedge, walk to the wall.* That reflex was **wrong here**, and the difference is the most useful thing in this plate.

After the outage, two service ports had disappeared and the source note attributed their loss to OOM kills. The decisive recovery fact was `uptime`: **12 days**. The box had recovered without a reboot. We do not retain kernel kill records sufficient to reconstruct victim selection here. Anonymous heaps can be swapped, reclaimed indirectly or encounter OOM handling; file-backed workloads can encounter OOM too. The allocation type is useful context, not a prognosis. This incident recovered; the model-loading incident did not. That is the comparison the evidence supports.

> **WHY RECOVERY IS NOT RESOLUTION**
>
> The box came back — and the cause was still running. `ps --sort=-rss` after recovery showed **19** `cicc` **processes, with the source note reporting roughly 6 GB apiece**: the extension was *still compiling*, a large aggregate resident footprint, potentially including shared pages, about to wedge the machine a second time the moment those jobs allocated together. The OOM kill had culled a couple of victims and relieved the instant, not the source. A box that recovers from a load-wedge and leaves the load running has not recovered; it has rearmed. After any self-heal, find the cause with `ps --sort=-rss` and kill it before it fires again.

Killing a runaway build tree (this took three attempts)

Reaping a ninja/`nvcc` tree is its own trap, because the obvious kills silently miss:

```
$ kill -TERM -<SID>
# nothing dies. kill -N targets a process GROUP (PGID), not a session id — passing an SID fails
# if it is not also the intended PGID. Our process tree crossed groups; inspect yours first.

# Historical recovery on our dedicated build host; not a shared-host cleanup recipe.
# What worked: stop the DRIVER first, then its verified workers.
# 1. find the build script in /proc/*/cmdline (skip your own shell, $$) and kill it, so no new
#    workers are spawned.
# 2. then reap the workers:
$ for w in ninja cicc cc1plus nvcc ptxas; do pkill -x "$w"; done   # -x = EXACT name match
# available 36 GB → 111 GB, load falling.
```

Exact name is still not ownership. The historical loop above can stop unrelated builds with the same worker names. For routine recovery use the build service/cgroup or verified descendant PIDs, then re-check the process tree. The relevant distinction in the incident was `-x` versus `-f`. `pkill -x cicc` matches processes literally named `cicc`. `pkill -f cicc` matches the whole command *line* — including the `ssh <box> "… cicc …"` string you are typing the kill through, so `-f` reaps your own session mid-command and you lose the shell you were using to fix the box. Prefer verified workload ownership to either kind of name match (Plate 12).

What actually built — and the one to skip

On aarch64 against torch 2.12 / cu130, the extensions that were expected to be the ARM64 wall compiled fine — `flexgemm`, `cumesh`, `nvdiffrast`, `nvdiffrec` all built clean. ARM64 was never the problem. One dependency failed only for a missing header (`Eigen`, header-only — fetch it to a local dir and set `CPLUS_INCLUDE_PATH`, no root needed). The genuinely dangerous one was `flash-attn`: it is the build whose `cicc` swarm was still resident hours later, and it is the one to skip in favour of `xformers` where the specific project supports that alternative. That pairing is what we saw on this stack — `xformers` is a broadly accepted memory-efficient-attention substitute (it backs the same attention path in many diffusion and LLM projects), but not every project accepts the swap, so confirm yours does before you make it rather than treating it as a universal rule.

Both the clean-build list and the flash-attn verdict are pinned to **torch 2.12 / cu130**. A newer torch or CUDA, or a different revision of any of these extensions, can move a dependency from one column to the other — re-measure against your own toolchain, don't inherit ours.

Recovery & prevention

1. **Diagnose before you pull the plug.** Use a bounded recovery window and an independent access path. Check `uptime`, kernel logs and process identities after access returns. A vanished port alone does not prove the OOM killer acted; a surviving socket alone does not establish boot history.

2. **Recovery ≠ resolution.** After it comes back, `ps --sort=-rss` and kill the still-running cause — driver process first (so it spawns no more workers), then stop verified descendants or the owning cgroup. The historical exact-name cleanup is not safe on a host running unrelated builds.
3. **Apply the headroom law to builds.** Everything Plate 09 says about model loads applies to a compile. `free -g` first — read `available`. Free the resident render/model engines with their `/free` endpoints exactly as you would before a heavy render. Prefer to build when *no* render lane is resident.
4. **Start with low build parallelism.** Our working convention was `MAX_JOBS=2`. The build is slower; the box lives. The default parallelism is tuned for a machine whose compilers don't compete with its accelerator for the same memory. This one does. The ~6 GB-per-`cicc` figure is this one build's measurement and per-job footprint varies by extension, so treat `2` as a starting point, not a safety guarantee, that you can raise deliberately while watching `free -g` — not a universal constant.

The honest one-liner: on a unified-memory box, **a build is not exempt from the memory budget just because it isn't a GPU job** — the accelerator and the compiler spend the same pool, and `MAX_JOBS` is a memory dial wearing a CPU label.

CROSS-REF → Plate 05 (Host Pressure Is GPU Pressure — the one pool a compile also draws from) · Plate 06 (The Wedge That Needs a Power Cycle — a similar symptom with a different observed outcome) · Plate 09 (Free First, Then Load — the headroom law, now applied to builds) · Plate 12 (The Do-Not-Touch List — `pkill -f` matches your own command, stated as a standing order)

PART 3 / PLATES 09–12

III

STANDING ORDERS

Make the next action depend on the fact you just checked. A warning in a notebook cannot reserve memory or preserve a job.

PLATE 09 · PART III

FREE FIRST, THEN LOAD

Failure class: admission without a runtime budget · Escape hatch: release, verify headroom, then load

You compare a 17.4 GB model file with the capacity printed on a 128 GB machine and decide it fits. The arithmetic is tidy because you left out the model already living there. By the time the new load tells you about the omission, the shell you would use to correct it no longer answers. The useful subtraction belonged before the request.

THE LIE / THE TRUTH

THE LIE — CAPACITY MISTAKEN FOR VACANCY	THE TRUTH — THE ADMISSION QUESTION
`total: 121 · challenger file: 17.4 GB` The file is much smaller than the machine. Start it.	`available: 12` The host is already carrying a large resident. The candidate's weights exceed the reported headroom even before its runtime overhead. Free the conflicting workload and measure again.

`total` describes capacity exposed to the operating system. `available` estimates immediate headroom. Neither tells you the exact peak requirement of a future job. For that you need the serving configuration: weight representation, context length, concurrency, KV cache, temporary buffers and the load path. A weight file is evidence about one component of the requirement, not a certificate of fit.

The incident, from the log

The August 7 note records a large resident running for nine days, roughly 109 GB attributed to it, and only 12 in the `free -g` available column. The challenger was listed at 17.4 GB. Both numbers had been visible before the request. Afterward the machine answered ping while failing useful remote checks, did not recover during thirty minutes of observation, and required a power cycle.

The failure was not that a reader lacked an obscure command. It was that a known constraint had no force over the next action. The system allowed a model test to begin without confirming that the model could be provisioned alongside the resident. A warning in prose did not reserve any memory.

```
# Diagnostic recipe; replace placeholders for your own service.
$ free -g
# Record available headroom and swap occupancy before any change.

$ curl --max-time 10 -s http://<host>:11434/api/tags |
    jq '.models[] | select(.name=="<candidate>") | .size'
# This Ollama endpoint lists file size in bytes, NOT peak runtime RAM.
# Convert units before comparing it with free -g's GiB-scale output.
```

The observed shortfall justified refusing that combination. It does not prove that every larger-than-RAM file must fail identically: loading strategy, offload, caching and reclaim can change what is resident and how slowly it runs. It certainly does not prove that a file slightly smaller than `available` is safe. The admission decision should be made from a tested workload budget, with headroom, rather than from the narrowest comparison that permits the request.




```
File size + context + buffers + other residents + headroom

A stop request is not a memory measurement.
```

Release, verify, then load.

The order, and the fact each step must establish

1. **Release the conflicting resident.** Prefer the system's existing service manager or documented unload operation. Identify the exact tenant and confirm it is not doing work that must complete. A generic process-name kill is not a model switch.
2. **Verify returned headroom.** Read the host again and inspect the old service's residency. The accepted stop request is not the measurement. Some allocators retain memory; another service may also be consuming it.

3. **Load under a measured budget.** Allow for the candidate's peak requirement and the services that must remain. Observe startup and readiness. An open TCP port while weights are loading does not prove that the model can answer.

These steps reduce one demonstrated risk. They do not guarantee that a driver, model or storage system cannot fail afterward. A passed memory check is a passed memory check, not a proof that the rest of the system works.

> THE CHECK HAS TO OWN THE DECISION
>
> A dashboard can display twelve available gigabytes and still let the next caller spend twenty. Put the admission check in the path that starts the work. If multiple callers can submit, the decision also needs ownership: one caller cannot reserve the same headroom another has just committed. On a small shared workstation, serialising heavy jobs is often easier to verify than predicting every overlap. A refusal with a clear reason is useful behaviour.

Recovery & prevention

1. **Keep a workload ledger.** Record warm and cold peak memory for the configurations you actually use. Include context length, batch size, image resolution and relevant runtime versions. A different configuration is a new budget.
2. **Make release observable.** Check the intended service's residency and host headroom after unloading. If the numbers disagree with the expected change, resolve that disagreement before admitting another large job.
3. **Use limits as tested controls.** A missing Docker memory limit does not force a process to fill the machine; the application determines its demand. Conversely, adding a limit does not prove safe GPU co-residency. Verify what is charged, what fails at the limit, and whether the OS retains room to recover.
4. **Define the failure path.** If the candidate cannot become ready in its expected window, retain the loader logs and use the agreed stop/recovery route. Do not send a second request just because the first caller timed out.
5. **Recalibrate after material changes.** A new model, context setting, allocator or driver can change the peak. Keep the last known good budget attached to its configuration, not as a folklore number for the entire machine.

The number on the box is not permission to load. Release the room, measure the room, then spend it.

CROSS-REF → Plate 05 (headroom and pressure) · Plate 06 (the consequence of a bad admission) · Plate 07 (releasing idle tenants) · Plate 10 (a timeout is an unknown outcome)

PLATE 10 · PART III

THE WATCHER GOES AWAY

Failure class: false failure (the poller dies while the work succeeds) · Escape hatch: read the engine's own record — its job history, its output directory — never the poller

This is a standing order, not a war story. A long job on this box runs somewhere you are not watching from: another process, a container, another machine. You submit it and you *poll* — a spinner in a browser tab, a request handler holding a socket open, a daemon thread ticking a status. The engine can outlive the watcher. When the watcher dies, times out, or navigates away before the job finishes, a real, completed, on-disk result reports to you as *it didn't work* — and you go hunting a failure that never happened.

THE LIE / THE TRUTH

THE LIE — WHAT THE POLLER REPORTS	THE TRUTH — WHAT IS PHYSICALLY ON DISK
The gallery is empty. The reply finished with no artifact attached. The poll returned an error, or a timeout, or simply nothing — and the log the service writes shows no trace of the request at all. *The render failed. It produced nothing. Submit it again.*	The job **completed.** The engine's own job history has it marked done, and the output file is sitting in the engine's output directory with a real byte count and a real mtime. Nothing failed. The only thing that broke was the thread, tab, or timer that was supposed to carry the result back — and it broke *after* the work was already safe.

The asymmetry is the whole lesson: **the poller and the work have separate lifetimes.** The work runs in a process built to run for forty minutes; the poller lives in a browser tab you might close, a request handler with a timeout, or a thread that dies the instant its parent process restarts. Either layer can fail. Establish which one failed before submitting the work again. Treat a missing result as a *reporting* failure first and a *work* failure only after the engine's own record has told you the job is genuinely gone.

The incident, from the log

The archetype: a long render — a job whose engine reports a warm baseline near **28 minutes** — was submitted, and the process holding the filing thread was restarted about **three minutes in**. The thread died with its process. The engine kept running on the other machine, finished the clip, and wrote it to its output directory exactly as instructed. But the thread that was supposed to pull it back and file it was gone, so the artifact existed *everywhere except the one place anything looked for it* — absent from every downstream index, invisible to the UI. Every surface a human checks said the render had produced nothing. It had produced a finished file the whole time.

The engine's own record told the truth, and it was one query away:

```
$ curl -s http://<engine>/history/<job_id> | jq '.[].status'
{ "status_str": "success", "completed": true }   # a ComfyUI-class backend records the job here
# The ENGINE says done. The poller that submitted it was already dead. Believe the engine.

$ ls -l <engine-output-dir>/
-rw-r--r-- 1 ...  <bytes>  <job-output>.mp4
# A real file, real size, real mtime — it was never missing. The artifact was on disk
# before the UI ever finished reporting "nothing."
# The gap was in the CARRY, not the WORK.
```

The same shape wore three other faces the same day, and none of them looked alike on the surface:

- **A poll timeout was read as failure.** A watcher gave up after its **40-minute** limit and reported `error`. The render had not stopped — it was still going, and finished later. A timeout is the *watcher* quitting, not the *work* quitting. When the engine confirms it, report "still rendering" and retain the job ID. Without that confirmation, report an unknown outcome rather than promising success or failure.
- **A browser tab aborted its own request.** An `onsubmit="return handler(...)"` whose handler was `async` returned a Promise — always truthy — so the browser never cancelled the native form submit. The page did a real GET navigation that **killed its own in-flight** `fetch` **POST** mid-request. Racy, so it looked intermittent. The tell-tale is in the address bar: the form fields appear in the URL (`?prompt=...&motion=...`) after you submit. The watcher didn't just miss the result — it could interrupt submission before the server accepted the work; server acceptance must be checked before retrying. This is standard HTML form behaviour on every browser, nothing to do with this box: `onsubmit="return handler()"` cancels the native submit on a literal `false` return, and an `async` function always returns a Promise, which is always truthy — so the submit proceeds and its `return false` resolves far too late to matter.

- **A finished file 404'd on retrieval** because the engine records an output as `{filename, subfolder, type}` and the retrieval URL was built from the filename alone, dropping the subfolder. The work was done and readable; the *address* the watcher constructed for it was wrong. Read the real output entry from the engine's history; do not reconstruct the path. (A ComfyUI-class backend does record each output as exactly those three fields — but the lesson is *read the record*, whatever your engine names them, not build the address yourself.)

> **SILENT LOGS ARE NOT EVIDENCE**
>
> The trap that turns a five-minute lookup into an afternoon: you check the service log, see no line for your request, and conclude the request never arrived. It proves nothing. A service started at `uvicorn.run(..., log_level="warning")` writes **no access log at all** — its console holds the startup banner and tracebacks and nothing else, so "no POST in the log" is the expected state whether the request succeeded, failed, or was never sent. Worse, buffering and carriage-return progress output can make a short `docker logs` tail misleading, so a live job's log tail sits frozen on an old line and reads exactly like a hang. **Before you reason from a log's silence, know what that service actually logs.** The authoritative record is the engine's structured job state — its `/history`, its in-memory jobs map, its output directory — not the absence of a line in a stream that was never going to print one. And note the second-order trap: an in-memory jobs map is itself a fragile watcher — a restart wipes it, so a genuinely-finished job can vanish from it while the file on disk is untouched. A valid file matched to the job is stronger evidence than a missing log line. Check the actual content.

The order of investigation — when something "produced nothing"

Do these in order. The artifact may exist even when the UI says otherwise.

1. **Did it actually run? Ask the engine, not yourself.** Query the render engine's own job history (a ComfyUI-class backend exposes this at `/history/<id>`; your engine has an equivalent) and list its output directory. Match the job ID to its recorded output, then open or decode the file. A plausible mtime or a nonzero size alone does not prove the requested artifact completed.
2. **Was anything still watching when it finished?** Compare the job's end time against your service restarts — timestamped console logs (`ls .../logs/*_console_*.log`) make this fast. If the process that submitted the job restarted between submit and completion, the filing thread died with it and the result is orphaned, not lost.
3. **Only now suspect the job.** An explicit engine error is evidence of failure. An absent record may instead mean history was lost; check the queue and durable submission record before deciding whether to retry. That is the last hypothesis, not the first.

Recovery & prevention

1. **Recover an orphan by re-filing it, not re-rendering it.** When the engine's record shows a completed job whose output never got carried back, pull the file from the engine's output directory and file it. Build this as a reconciler that runs at process startup and on demand: sweep every render host for completed jobs whose outputs aren't yet indexed, and adopt them. Re-rendering a clip that already exists wastes a full render slot to reproduce a file you already own. **Verify the reconciler on a disposable copied output with its index entry omitted** — a sweep that never rescues anything is a sweep you cannot trust.
2. **Make the watcher outlive the work, or make the work self-report.** A poller in a tab or a thread is doomed by design. Either move the carry into something as durable as the job (the engine writes the result to its final home itself; the watcher only *notices*), or accept that the watcher will sometimes die and lean on the startup reconciler to catch what it drops. Do not build the only record of a forty-minute job into a thing that lives thirty seconds.
3. **A timeout reports an unknown outcome until checked.** Say "still running" only when the engine confirms it. Otherwise preserve the job ID, report the lost observation, and check before retrying. Reporting `error` on a timeout is the poller confessing its own impatience as the job's death, and it sends the next operator to re-run a render that was about to finish.
4. **Health-check the record, not the reporter.** Before you believe a job failed, hit the layer that holds the truth — the engine's history and its output directory — the same discipline as every plate in this book: read the fact from the layer that actually knows it, not the shallow surface that answers first.

The honest one-liner: **a finished job and a living watcher are two different facts, and only one of them is on disk. When they disagree, believe the disk.**

CROSS-REF → Plate 03 (Ping Is Not Health — the mirror image: there the box answers while the work cannot even start; here the work succeeds while the reporter has already gone) · Plate 04 (Silent Success — the opposite direction of the same sin: there an empty result files itself as done; here a done result files itself as nothing) · Plate 06 (The Wedge That Needs a Power Cycle — a real hang, so you can tell it apart from this false one) · Plate 09 (Free First, Then Load — the ordering discipline this Part is built on)

PLATE 11 · PART III

DON'T BELIEVE THE QUANT'S BROCHURE

Failure class: a format label mistaken for delivered speed · Escape hatch: measure first-token latency and steady decode separately

You rebuild a 70B-class model in fp8 and expect the reply to arrive twice as fast. Eight is half of sixteen. The chat window proceeds at about four tokens per second, much as it did before. The conversion completed; the expectation did not. You compared the new format with an imagined fp16 baseline, while the actual previous deployment was already a quantised GGUF model.

THE LIE / THE TRUTH

THE LIE — THE EXPECTATION	THE TRUTH — THE OBSERVATION
`fp16 → fp8 = half the width = twice the chat speed` Treat a representation change as a throughput promise. Assume every benefit arrives on the axis you care about.	`observed long-reply decode: about 4 tok/s` The replacement delivered roughly the old deployment's rate. The serving runtime, parser and quantisation had changed together. That comparison cannot isolate which change caused each benefit.

The “lie” here is our expectation, not a quotation from a vendor. No identified brochure promised that this exact model, runtime and context would double in speed. The chapter title names the habit: importing a headline or a format label into a workload it never measured.

The incident, from the log

The existing model had a useful voice but an unreliable tool-call parser in its serving path. The replacement used the model's fp16 weights to produce an fp8 checkpoint, then served it with a different runtime. The source note records four shards, a checkpoint in the roughly 68 GB class, cleaner parsing and useful prompt/prefix-cache behaviour. Long replies remained around four tokens per second.

Later loader notes identify a checkpoint size of **67.68 GiB**. That distinction matters when doing arithmetic against a bandwidth specification expressed in decimal GB/s. “About 68 GB” is adequate for a rough incident description; it is not adequate for claiming a precise ceiling to two decimal places.

```
# Reconstructed deployment scorecard, not a controlled benchmark.
previous:    quantised GGUF, previous serving runtime
replacement: fp8 checkpoint, different serving runtime
decode:      about 4 tokens/s in the replacement
comparison:  reported as about the same rate as before
benefits:    parser behaviour and prompt/prefix-cache latency
```

The tempting explanation is memory bandwidth. For a large dense model decoding one sequence, much of the weight set may be read for each generated token. If weight traffic dominates, a first-order estimate is bandwidth divided by bytes read per token. That is a useful sizing model. It is not a measurement of this runtime's actual memory traffic.

```
# Illustrative weight-streaming estimate with explicit units.
$ python3 -c 'print(273e9 / (67.68 * 2**30))'
3.756...
# Decimal bandwidth / binary checkpoint bytes: roughly 3.8 tokens/s.
# Idealised estimate, not an exact bound or measured bandwidth.
```

NVIDIA lists **273 GB/s** memory bandwidth for DGX Spark in its hardware specification. The approximation above omits KV-cache traffic, activations, kernel overhead, caching effects and the difference between a checkpoint file and the bytes read by a running kernel. Its proximity to four tokens per second is consistent with a bandwidth-sensitive workload; it is not proof of a fully explained bottleneck.

What a format name leaves out

GGUF is a container format that can hold several quantisation types. Saying “GGUF versus fp8” does not specify two comparable bit widths. Nor does similar measured speed prove that two implementations move the same number of bytes. One may spend more time in dequantisation, different kernels or cache management. Our deployment change was useful, but it was not a controlled quantisation study.

Lowering weight precision **can** improve decode throughput when bandwidth is the bottleneck. It can also change kernel support, quality and memory overhead. A four-bit model is not automatically twice as fast as an eight-bit model, and fp16 to fp8 is not automatically irrelevant to speed. Both claims would outrun this incident.

> **KEEP THE SCORECARD IN SEPARATE COLUMNS**
>
> First-token latency, sustained tokens per second, aggregate throughput, tool-call validity and model quality are different outcomes. A runtime may improve one while leaving another unchanged. Prefix reuse can make a familiar prompt cheap without improving long-form generation. A batching benchmark may serve many users efficiently while telling you little about one person's waiting time. Keep the prompt, context, concurrency and cache state beside the number.

The parser improvement belongs to the serving path, not to fp8 as a property. The retained voice is an assessment of this model in use, not a guarantee that a precision will preserve every finetune. The prefill improvement was reported with a runtime and caching change; attributing all of it to quantisation would conceal the part that actually did the work.

Recovery & prevention

1. **Name the baseline exactly.** Model revision, quant type, runtime, kernel configuration and hardware. A format family is not a baseline.
2. **Measure the wait and the stream.** Record time to first token and sustained decode separately. Use the same prompt, context and output length; report cold and warm conditions.
3. **Use the roofline as a plausibility check.** Bandwidth divided by estimated bytes per token helps set expectations. Do not invert the ratio and call seconds per token “tokens per second,” or mix GB and GiB silently.
4. **Change one major factor when making a causal claim.** If you replace the runtime and the quant together, report a deployment comparison. It may be exactly the practical comparison you need, but it cannot identify every mechanism.
5. **Keep the benefits that arrived.** A stable parser can justify a migration even without faster prose. Report that benefit directly, alongside the unchanged decode rate.

The quant label describes a representation. The stopwatch describes your workload. Keep both.

CROSS-REF → Plate 05 (shared capacity and pressure) · Plate 09 (weight size is not peak runtime memory) · Plate 04 (a valid response must contain usable work)

PLATE 12 · PART III

THE DO-NOT-TOUCH LIST

Failure class: maintenance that removes the recovery path · Escape hatch: verify ownership and preserve independent access

The link looks redundant. The process name looks specific. The container is running, so surely it will run after the next reboot. None of these impressions tells you what depends on the thing you are about to change. The last plate is a list of dependencies to establish before ordinary housekeeping becomes an outage.

THE LIE / THE TRUTH

THE LIE — THE APPARENT REDUNDANCY	THE TRUTH — THE ACTUAL DEPENDENCY
`wired network + Wi-Fi + VPN + private link` Several labels on the diagram. Several independent ways back in. The private cable can be changed casually.	`one reliable route to the second host` On our pair, Wi-Fi had already failed for two days. The private link was the dependable path for the bridge and administrative hop. A VPN also needs an underlying network path; its name does not create another cable.

Our incident notes record a private pair of addresses ending in `.10` and `.11`; they do not establish a subnet mask for a reusable reader recipe. Read the deployed interface and route configuration. A precise-looking address example copied without its actual mask is worse than a schematic: it can become a broken instruction.

The orders

1. Do not improvise firmware maintenance. Our internal operating notes prohibit `mst` and `mstflint` on our GX10s. They do not contain a reproduced failure or a vendor bulletin establishing that merely running either utility bricks every GX10. The book cannot turn that local prohibition into a hardware fact. Our machines retain their local rule. For your machine, use the exact vendor-supported update procedure, matching hardware identity and firmware, with a recovery plan. An inspection command and a firmware-write command are different operations. We have not tested the alleged bricking mechanism.

2. Protect the path you are using to recover. Before changing a NIC, address, route or firewall, establish a second path that actually works. An independent console can be more valuable than three services that all depend on one SSH connection. On our pair, the private ConnectX link was load-bearing. That is our topology, not a property that makes every private cluster cable untouchable. NVIDIA's networking guide describes supported Spark networking; partner hardware still needs its own applicable guidance.

3. Use one owner for model switching. A switch should stop the old resident, verify headroom, start the new workload and wait for readiness. Our lab has a tool for that sequence. There is no universal GB10 “brain-switch” command. Preserve the sequence and its checks, whether your owner is a service manager, script or human runbook. Ask the server's model registry what loaded; a model's prose about its identity is not evidence.

4. Choose restart policy deliberately. Our source notes found a serving container configured with `restart=no`. That would not return automatically solely through Docker's restart policy. For a mutually exclusive model that must remain stopped after an intentional switch, `unless-stopped` is often appropriate:

```
# Configuration example, not a command run for this book.
$ docker inspect -f '{{.HostConfig.RestartPolicy.Name}}' <container>
no

$ docker update --restart unless-stopped <container>
# Verify policy and the deployment's ownership convention.
```

Both `always` and `unless-stopped` respect a manual stop while the current daemon keeps running. The important difference is across daemon restart: `always` can bring a manually stopped container back; `unless-stopped` preserves that stopped state. See Docker's restart-policy documentation. Do not configure every mutually exclusive large model to return together after a reboot.

5. Prove boot behaviour. Being a systemd unit does not by itself establish automatic startup or restart-on-failure. Enablement, dependencies, conditions and restart settings matter. A user unit may need a persistent user manager for startup without login. In our incident a render service launched as a detached process did not return as the supervised services did. Inventory the service owner and verify a controlled restart window. See systemd's login manager documentation for user lingering.

6. Stop the exact workload you own. A broad match for `server.py` once killed our supervisor while targeting a render service. A full-command match can also catch a remote shell containing the pattern. Exact process names reduce one kind of over-match, but `pkill -x cicc` still targets every matching compiler, including another build. Prefer the owning service, container or cgroup. Otherwise inspect executable, owner and ancestry before acting on verified PIDs.

```
# Read-only identification examples; inspect before stopping anything.
$ ps -eo pid,ppid,pgid,comm,args

PS> Get-NetTCPConnection -LocalPort <port> -State Listen |
      Select-Object LocalAddress,LocalPort,OwningProcess -Unique
# Multiple listeners can exist across addresses/configurations.
# Verify the process and its service owner before any stop.
```

The incident, from the record

These orders join separate observations: the two-day Wi-Fi loss, the unconfigured container restart policy, the detached render service, the overbroad process kill and the shell-matching trap. They are not one escalating disaster. Keeping them separate matters. The firmware prohibition is an untested local policy; the access loss and mistaken process termination were observed. A book about evidence should not make the former sound more certain than the latter.

> **THE RECOVERY PATH IS ALSO A DEPENDENCY**
>
> A repair dashboard that calls SSH cannot repair a host whose only SSH route you just removed. A supervisor you killed cannot restart the service you meant to target. Before a change, draw the path from your hand to the action and back to its confirmation. If both directions depend on the component being changed, prepare another way in first.

Recovery & prevention

1. **Record actual topology.** Interface names, addresses, routes and a tested independent console. Several boxes on a diagram do not prove several paths.
2. **Keep ownership explicit.** One authoritative switch path, an intentional boot policy, and identified service boundaries make cleanup predictable.
3. **Separate inspection from mutation.** Read-only identification precedes a process stop or network change. Firmware writes require the applicable hardware procedure.
4. **Verify the effect.** Confirm the intended workload stopped, unrelated services survived, and the promised access path still works. The command exiting successfully is only the beginning of that check.

Maintenance is work on the dependencies you usually take for granted. Name them before touching them.

CROSS-REF → Plate 03 (what a reachable host proves) · Plate 06 (independent recovery) · Plate 08 (build ownership) · Plate 09 (switching with headroom)

SOURCE NOTES

The lab records support the incidents; the external references support the narrower technical explanations. These are different kinds of evidence. Neither makes an unrecorded experiment happen retroactively.

Incident register

PLATES	RETAINED RECORD	WHAT IT SUPPORTS
01, 05, 07	GPU accounting notes and the two-host operations log, July–August 2026	Unsupported total, idle engine footprint, host readings and recovery interval
02	Two-host log and GPU accounting notes, July 2026	Similar clocks across different performance states; measured job durations
03, 06, 09	August 7 headroom incident	Resident load, candidate file size, ping, failed useful access, thirty-minute wait and power recovery
04	Vision-budget investigation, July 29; model-probe notes	Six-image budget comparison, ignored thinking flag on one build, empty completions
08	August 10 build incident and its later correction	Parallel compiler load, uptime proving no reboot, remaining worker tree, cleanup
10	Watcher and render-retrieval investigations, July–September	Engine work surviving a dead poller; aborted form request; wrong retrieval path
11	fp8 deployment scorecard, August; loader record, September 16	Approximate decode rate, parser/runtime changes, 67.68 GiB checkpoint
12	Network, supervision and process-identity notes	Failed Wi-Fi path, detached service, restart policy and process-match errors

The private source paths and editorial change list are retained in the author's private records. They are omitted here because they contain unrelated household projects and operational details. No reader is asked to accept them as public raw benchmark data.

Primary references

Checked for the September 20, 2026 editorial pass. Links open the supporting documentation, not a search page.

1. NVIDIA DGX Spark known issues — aggregate framebuffer reporting and unified-memory reporting limits. Plates 01 and 05.
2. NVIDIA nvidia-smi reference — supported metrics, unavailable data and platform-specific reporting. Plates 01 and 02.
3. NVIDIA DGX Spark hardware overview — published memory specification. Plate 11's approximation is the author's calculation, not a published benchmark.
4. NVIDIA DGX Spark release notes — version changes and partner-system scope. Edition note.
5. Linux memory-management concepts — file-backed pages, anonymous memory, reclaim and OOM. Plates 05, 06 and 08.
6. Linux pressure-stall information — interpretation of memory pressure. Plate 05.
7. Docker resource constraints — container limits and OOM considerations. Plates 06 and 09.
8. Docker restart policies — manual stop and daemon-restart behaviour. Plate 12.
9. systemd loginctl reference — persistent user managers through lingering. Plate 12.
10. NVIDIA ConnectX networking on DGX Spark — platform networking reference. Plate 12.
11. Ollama model listing — model-list response and file-size field. Plate 09.
12. GGUF format specification — container format and tensor types. Plate 11.

Before relying on a command

Examples containing `<host>`, `<port>`, `<container>` or `<engine>` require substitution and inspection. Engine-specific endpoints are identified as examples. Read-only diagnostics and state-changing recovery actions are labelled differently; a stop or unload must target the workload you intend to change.

The desktop and server examples are separate incidents. They do not establish a cross-platform performance comparison. Job durations, memory footprints and model behaviour are observations of particular configurations, not product guarantees.