// GPU OBSERVABILITY AGENT

Automated Root-Cause Diagnosis.

ApexData Grogh reads the telemetry of a GPU training run and produces a grounded root-cause verdict — what failed, why, and what to do about it — without a human poring over logs and dashboards.

- Supports 20+ known failure modes
- No access to training code required
- Instant, defensible root-cause report
- Slurm-scheduled GPU clusters

Contents

// Capabilities Overview

Overview

01 Who it is for — 3

02 How it reports — 4

03 Frameworks & training stacks — 4

Failure Cases

A Startup and configuration failures — 5

B GPU memory and hardware faults — 6

C Distributed coordination and hangs — 8

D Throughput and utilization — 10

E Training correctness — 12

F Cluster and scheduler events — 13

G Healthy baseline — 15

Observability

04 What we collect for observability — 16

05 Contact & next steps — 17

// What it does

Who it is for

It reasons over telemetry **only** `metrics`, `logs`, and `scheduler records` collected by an on-node observability agent. It never needs to read the customer's training code.

For teams running large ML trainings

A multi-node training job that crashes, hangs, or silently produces a bad model can burn days of expensive GPU time before anyone notices. The agent watches every run and, the moment one ends badly, tells you which of roughly twenty known failure modes occurred, cites the exact metric and log evidence behind the call, and separates a code or config problem you must fix from an infrastructure problem you should escalate.

⚠ **Catches "done-but-bad" cases: exits cleanly but learned nothing**

For GPU neoclouds and cluster operators

When a customer reports "my training failed on your cluster," the operator faces a support burden and an attribution problem: was it the customer's code, or the cluster's hardware and network? The agent answers that question in minutes, from telemetry alone, with no access to the customer's source. It cleanly attributes each failure to the job (code or config) or the infrastructure (ECC error, node down, slow interconnect) — with the evidence attached.

// instant, defensible report — lower support cost, faster TTR

// Output format

How it reports

01 Shows	02 Implies	03 Cannot Decide
Each claim carries the exact query and numbers behind it. No black-box assertions — every data point is cited.	The agent corroborates a cause across stores before concluding — confirming OOM in both log traceback and GPU memory curve.	States its limits honestly rather than guessing. The result is a verdict an engineer can trust and act on directly.

Every investigation ends in a three-part verdict — what the data shows, what it implies, and what it cannot decide — plus a single failure family and a confidence score.

// Framework support

Frameworks & training stacks we recognize

The agent reads training progress — `loss`, `gradient norm`, `learning rate`, `checkpoints` — directly from the job's standard output using a registry of framework-specific parsers.

Hugging Face Transformers Trainer

Full ecosystem support including the surrounding fine-tuning and alignment stack.

TRL · LLaMA-Factory · Axolotl · Unsloth

Megatron-LM

Megatron-Core support for large-scale tensor and pipeline parallelism workloads.

Megatron-Core

PyTorch Lightning

Full trainer lifecycle tracking with step and epoch-level telemetry parsing.

Keras / TensorFlow

Callback-level and fit-loop telemetry for TF2 and Keras training runs.

It recognizes the PyTorch distributed launchers (`torchrun` / `torchelastic`) and their failure summaries, and the standard exception shapes from the surrounding stack: the `datasets / PyArrow` data pipeline, **PEFT** (LoRA and adapters), the model-hub client for model, tokenizer and checkpoint access, **bitsandbytes**, and the **CUDA / NCCL** runtime. Runs are scheduled by **Slurm**, and the agent reads the Slurm job and step lifecycle to anchor every investigation. When a run uses a fully custom training loop that prints no recognizable progress, the agent says so — it reports training correctness as "unverifiable" rather than guessing.

SUPPORTED FAILURE CASES

The agent rules each of the following families in or out on every run, with counterfactuals, before concluding. Cases are grouped by where in the run's life the problem appears.

- A · Startup & config
- B · GPU memory & hardware
- C · Distributed & hangs
- D · Throughput & utilization
- E · Training correctness
- F · Cluster & scheduler
- G · Healthy baseline

Startup and configuration failures

The run never really begins — it dies at setup, before training.

A1 Startup crash: bad data, config, environment, or asset

The job fails within seconds, before the training loop starts, because of an application-level error. The agent names the exact exception and classifies a coarse fault class: data (a malformed or unparseable dataset), config (a typo'd hyperparameter or an invalid adapter target), env (a missing or incompatible package, or an unsupported model architecture), or asset (a model, tokenizer, or checkpoint that cannot be loaded — wrong path, gated repository, or missing token). No GPU work has begun, and the same misconfiguration hits every rank.

Example verdict

What the data shows

All 16 ranks raised ValueError: Target modules {'query_key_value'} not found in the base model before any kernel launched; the GPU committed under two percent of its memory (a CUDA context only). The fault class is config .

What it implies

This is a startup crash, not an out-of-memory or a hang — the run died at model setup. "query_key_value" is a Falcon/BLOOM attention name; this model family uses q_proj/k_proj/v_proj. Fix the adapter target modules in the training config and resubmit. The nodes were healthy, so this is the job's own configuration error, not a cluster fault.

A2 No-traceback signal kill (OOM-kill / native crash)

A training process is killed by a fatal OS signal before it can print a Python stack — most often SIGKILL from a host or cgroup out-of-memory condition (dataloader workers or pinned buffers exhausting host RAM), sometimes SIGSEGV / SIGABRT / SIGBUS from a native crash (a C-extension, driver, or kernel fault). Because the launcher wraps the killed child, the only exceptions in the log are the generic launcher wrappers, and the job's exit code looks like an ordinary failure — the signal is visible only in the launcher's failure summary.

Example verdict

What the data shows

Ranks 1 and 4 were killed by SIGKILL (signal 9) with no Python traceback; the only captured exceptions are the launcher wrappers. The kill fired at the startup phase — zero kernels launched, zero GPU memory committed. The nodes stayed healthy and the job was not requeued.

What it implies

A SIGKILL with no traceback in a zero-GPU-memory startup context is most consistent with a host-RAM out-of-memory kill (dataloader workers forking dataset copies, or a large model loaded into host memory before GPU transfer). Reduce host-RAM pressure — fewer dataloader workers, smaller prefetch — and rerun.

What it cannot decide

Whether the kill came from the host OOM-killer, a cgroup memory limit, or an external watchdog cannot be confirmed: no host-memory metric is available to pin the exact trigger.

B GPU memory and hardware faults

The GPU itself runs out of memory or goes bad under the job.

B1 CUDA out-of-memory

A rank requests more GPU memory than the card has — at model load, at the first forward/backward pass, or in the optimizer step — and PyTorch raises an out-of-memory error with the requested size. The discriminator is a single allocation request that exceeds the card's capacity (a workload memory-plan problem), not a hardware fault.

Example verdict

What the data shows

All 16 ranks raised torch. OutOfMemoryError at the first training step; one rank tried to allocate 91.6 GiB on a card with 79.2 GiB total. The memory curve climbed to near capacity on the run's GPUs, and the hardware counters were clean.

What it implies

This is a workload memory-plan limit, not a hardware fault — a single oversized allocation fails immediately, so the sampled peak can sit below the cap even though the request exceeds it. Reduce per-GPU memory pressure: smaller micro-batch, activation checkpointing, sharding, or shorter sequence length.

What it cannot decide

The exact knob to turn is not in the telemetry — the scheduler stores the run's resource request and standard output, but not the training configuration.

B2 GPU hardware fault: XID / ECC / HBM row-remap

A GPU degrades or fails under the job — an XID error, an uncorrectable (double-bit) ECC error, or an HBM row-remap failure. This is the hardware's fault, not the code's. A separate, quieter case is latent degradation : a serious counter that accrued in the past and now sits flat on an otherwise-healthy run — a GPU to flag for replacement before it fails outright.

Example verdict

What the data shows

On the run's GPUs, the uncorrectable-ECC counter rose co-incident with the failure window (a true delta above zero) and the run ended abnormally at that moment. (Latent variant: a GPU carries three uncorrectable HBM row-remaps that did not rise during this run — flat and pre-existing.)

What it implies

A fault counter that rose under the run makes the hardware the cause: drain or replace the GPU, escalate to operations, and rerun on healthy hardware — this is not a code fix. A flat pre-existing counter is latent degradation: a secondary "flag this GPU for ops" warning that names the GPU but does not change the run's verdict. A bare health-status flag with clean concrete counters is treated as incidental, never as the cause.

B3 Clock throttling: thermal or power

A GPU drops its clocks because it hit a thermal or power limit, cutting throughput. A severe throttle condition (hardware slowdown, thermal, or power-brake) with clocks dropping and progress slowing is the cause; benign idle or soft-power-cap states at low temperature are normal and are not flagged.

Example verdict

What the data shows

A severe thermal/slowdown throttle condition was set on the run's GPU while clocks dropped and step time roughly doubled; the loss kept decreasing, so the run was otherwise healthy.

What it implies

The slowdown is a hardware thermal or power condition, not a code or data problem — check cooling and power on the affected node and rerun; the work itself is fine. Benign throttle states at low temperature would not be flagged.

What it cannot decide

The root environmental cause (cooling, power delivery) is outside the telemetry.

Distributed coordination and hangs

The run holds its GPUs but makes no progress.

Distributed init / rendezvous hang

A multi-node job hangs before training starts because the ranks cannot form their process group — typically an unreachable or misconfigured coordinator address. All ranks sit cold across every node, no collectives ever start, and the job eventually times out (or fails late when a rendezvous socket times out).

Example verdict

What the data shows

All 16 ranks across both nodes are cold — engine activity near zero, zero kernel launches, and no NCCL communicator or collective series ever appeared . The launcher printed a distributed-init banner and went silent; the run ended on a rendezvous timeout.

What it implies

The ranks never got past process-group rendezvous — a coordination failure before NCCL (an unreachable or wrong coordinator, a firewall, or DNS), not a per-host stall and not a training fault. The blast radius is the whole multi-node allocation.

What it cannot decide

Which of unreachable-coordinator, firewall, or DNS is the trigger — the telemetry shows all ranks stuck before NCCL, not why.

NCCL collective hang (mid-training freeze)

A multi-rank job trains for a while, then forward progress freezes while it still holds its GPUs — often because one rank died (for example, a dataloader worker was killed) and the survivors block forever in a collective when the NCCL watchdog is disabled. A stuck collective pins engine activity high (a busy-spin) even though no real compute happens.

Example verdict

What the data shows

In the recent window, NCCL collective completions and kernel launches both fell to zero while the allocation stayed held; whole-run totals were large (it trained earlier). Fifteen ranks are spinning at engine activity near 1.0 with SM activity near zero, one rank went cold, and a dataloader-worker kill appears in that rank's log.

What it implies

This is a mid-training collective freeze triggered by a lost peer — the survivors block on the dead rank's collective indefinitely. High engine activity here is a busy- wait , not health, and a large whole-run collective count does not prove current progress. Report both the trigger (the rank loss) and the systemic hang; rerun with a NCCL watchdog so a lost peer fails fast instead of hanging.

C3 Idle-GPU hang (single GPU)

A single-GPU job holds its allocation but does no compute — engine and SM activity near zero, zero kernel launches — while it stays running. The process is alive but stuck before or between GPU work (a host-side deadlock, a blocking wait, a stuck dataloader). Logs are often empty, so this failure is invisible without metrics.

Example verdict

What the data shows

The GPU is held (memory resident) but idle — engine and SM activity near zero with zero kernel launches sustained — while the job is still running; no process ever attached to the GPU. Standard output is empty: no traceback, no progress.

What it implies

This is an idle-GPU hang — the allocation is wasted and making no progress. The absence of exceptions in an empty log is not health; this failure lives only in the metrics. Cancel and rerun, ideally with a wall-clock limit so a stuck run is reaped automatically.

What it cannot decide

The host-side cause (a deadlock, a blocking wait, a stuck dataloader or launcher) — the telemetry sees the idle GPU, not the reason.

Throughput and utilization

The run is slow, not broken — it completes but wastes the hardware. These "done-but-bad" cases are the easiest to miss.

D1 GPU under-utilization (input-pipeline starvation)

The GPU launches kernels and makes progress, but its duty cycle is low — idle a large fraction of wall-time between bursts, classically because the input pipeline cannot keep it fed (a slow or streamed dataloader, on-the-fly tokenization on the critical path). The run usually completes cleanly, slow rather than crashed, so nothing else flags it.

Example verdict

What the data shows

The GPU is launching kernels and the loss is decreasing, but the engine duty cycle is low with recurring step-scale launch gaps — the GPU sits idle a large fraction of wall-time. The hardware is clean.

What it implies

The run is under-utilizing the GPU, most likely input-pipeline starvation. Baseline-free, we can confirm a severe under-utilization (near-zero engine while launching, or multi-second hard stalls) but only suspect a mild one — a small or CPU-bound model can legitimately under-fill a large GPU.

What it cannot decide

Separating data-starvation from inherently-low utilization needs a throughput target for this workload; baseline-free, the cause and the magnitude are not decidable.

D2 Throughput-bound (I/O save-storm / interconnect / launch-blocking)

A run completes and learns, but runs at a fraction of the hardware's worth because of a nameable host or cluster bottleneck. Three sub-modes: an I/O save-storm (checkpointing nearly every step blocks on storage); an interconnect bottleneck (GPUs busy-wait on a slow collective — for example NCCL over TCP sockets instead of the fast fabric); or launch-blocking (a debug flag serializes every kernel launch).

Example verdict

What the data shows

The run completed and the loss learned, but throughput was low for a nameable reason. In the interconnect sub-mode the collectives keep advancing , yet engine activity is pinned near 1.0 while SM activity is near 0.1 and the CUDA-API time is roughly forty times its normal floor — the GPUs are spinning on a slow asynchronous collective. In the I/O sub-mode the log shows a checkpoint save at nearly every training step with a collapsed SM duty cycle.

What it implies

This is throughput-bound, not a hang (it keeps advancing) and not a straggler (it is uniform across ranks). High engine activity here is a busy- wait , not health — a utilization-only view would mislabel it "well-utilized." The remedy is sub-mode-specific: fix the NCCL transport/fabric; reduce save cadence or use faster storage; or unset the launch-blocking debug flag. It is silent waste — the run completes, so nothing else catches it.

D3 Straggler (one slow rank)

A multi-rank job runs and usually completes, the model learns, the hardware is clean — yet throughput is well below the allocation's worth because one (or a few) ranks are persistently slower. Every synchronized step waits on the slowest, so the faster ranks idle in the all-reduce. Nothing crashes or hangs.

Example verdict

What the data shows

The run is advancing (collectives keep completing) but one rank shows a sustained SM duty cycle far above the cross-rank median (about 0.55 versus 0.16, a 3.6x gap) while the peers spin-wait on it each step. Per-rank kernel and collective counts are identical (synchronous training is lock-step), so the asymmetry lives only in the duty cycle.

What it implies

This is a straggler — one lagging rank drags the whole multi-GPU allocation to its pace while every aggregate signal looks fine. We judge on SM activity, not engine activity (the idle peers spin inside the all-reduce, pinning their engine high), and we name the lagging rank by node and GPU.

What it cannot decide

The host-side cause of the slow rank — a sick GPU, a slow data shard, a NUMA/PCIe imbalance, or CPU contention — and the exact throughput loss without a baseline.

Training correctness

The run completes and exits cleanly, but the model is wrong. Infrastructure health does not establish training correctness.

E1 Loss divergence / not-learning

The most dangerous false-negative: the infrastructure is perfect — GPUs busy, no OOM or hang, hardware clean — and the run completes successfully, yet the model learned nothing. The loss went non-finite, spiked far above its start, collapsed to a flat high plateau, or stayed flat with gradient norm and learning rate near zero. Reading the loss trajectory is the only way to catch it.

Example verdict

What the data shows

The run completed cleanly and the GPUs were healthy, but the loss trajectory is wrong — it rose and dead-plateaued near 11.9 (or hit NaN), with the gradient norm trending to zero. Every infrastructure tool read "healthy" until the loss was read.

What it implies

This is loss-divergence — a training- correctness failure, not an infrastructure one. The whole allocation produced a model that learned nothing, and because it completed cleanly nothing else flagged it. The tell is the shape of the loss, not the presence of NaN (a divergence can leave zero NaN lines).

What it cannot decide

The root cause — learning-rate value or schedule, mixed-precision instability, or dataset/label integrity — the configuration behind the trajectory is not in the telemetry.

E2 Silent resume-from-scratch (checkpoint not loaded)

A run meant to resume from a checkpoint instead trained from scratch: it never loaded the checkpoint (no skip-marker, the loss restarted at the initial value), wasting the entire parent run's compute. Because it completes and the loss trends down, every infrastructure and loss check calls it healthy. A related case is a resume that crashes at the checkpoint load — wrong base, size mismatch,
or an empty/corrupt checkpoint directory.

Example verdict

What the data shows

This job is a resume (submitted with a dependency on a parent run), but the log shows no "loading from checkpoint" or "continuing from step N" marker, and the loss restarted at the initial value (head loss far above the parent's tail) — the same shape as a fresh run. It completed cleanly.

What it implies

The run silently trained from scratch — it did not honor the checkpoint and wasted the parent run's entire compute, despite completing and looking healthy. This is not healthy, and not loss-divergence (the loss is fine — the fault is the un-loaded checkpoint). Fix the checkpoint reference and rerun. In the crash variant, the verdict names the checkpoint cause (size-mismatch, no-valid-checkpoint, or corrupt) and points at the matching base, not a generic code fix.

E3 Undertrained (step budget too low)

A run completes but logged only a handful of training steps — the step budget (max steps or epochs) was set far too low. It did negligible training, wasted the allocation, and its correctness cannot even be assessed from so few steps.

Example verdict

What the data shows

The run completed but logged only one training step; any save-cadence or utilization ratio computed over a single step is degenerate, not a real signal.

What it implies

The run is undertrained — the step budget was set too low, so it did negligible training and wasted the allocation, and correctness cannot be assessed. This is a configuration issue, not a throughput problem and not a healthy run; we surface the step count and the wasted allocation as the headline and recommend fixing the step budget.

F Cluster and scheduler events

The run died for a reason that is not its own.
The action is the opposite of a code fix: rerun, not debug.

F1 Node-down mid-run

A run dies because a node it was using failed or was drained under it — not because of anything in the code. The training was healthy until the node vanished (loss down, collectives flowing, GPUs busy), then it just stopped, usually with no traceback (the controller kills the job before any exception fires).

Example verdict

What the data shows

A node in the run's node list entered a DOWN/DRAIN state co-incident with the run's end, while the other nodes stayed healthy; the run ended abnormally at that moment with no traceback. Up to the kill, it was progressing normally.

What it implies

This is a cluster failure — the blast radius is a node , not the job: the run died because the cluster lost a node under it, not because of a code fault. Rerun on healthy nodes and have the node investigated. We confirm and localize the node from the scheduler's node-state signal, never from an incidental GPU health flag.

What it cannot decide

Why the node failed or was drained — that hardware cause is outside the telemetry.

F2 Requeue / preemption

The scheduler sent the job back to the queue (a requeue) or evicted it for a higher-priority job (a preemption) — a scheduler or operator action, not a job fault. The job reappears and reruns; the right read is the final run's outcome plus whether the rerun recovered.

Example verdict

What the data shows

The job carries a requeue event — it reappeared under the same job id (a new accounting row) and reran, ultimately completing. The mid-run teardown that ended the first attempt was the requeue kill, not a crash.

What it implies

This is a scheduler action, not a job fault — and it holds even though the job ultimately completed (a requeue is an event in the run's history). We report the final disposition and whether the rerun recovered: a requeue that completed is low-severity wasted compute; one still pending or lost needs a rerun. A no-context reader would misread the two accounting rows as two unrelated runs.

F3 Timeout

A run was reaped at its wall-clock limit. Timeout is a disposition , not a root cause — the real question is whether the run was healthy-but-slow and simply needed more time, or was hung.

Example verdict

What the data shows

The run hit its wall-clock limit with the nodes healthy (no node or scheduler event). The other tools show whether it was making progress up to the wall or had stalled.

What it implies

Timeout is a disposition, not a cause. If the run was healthy-but-slow, raise the wall or speed it up (see the throughput cases); if it was hung, the timeout is a consequence and the hang is the real cause to name. We never let "timeout" mask an underlying hang, and never call a timeout a crash.

Healthy baseline

Not every run is broken — and the agent confirms health on positive evidence, never on the mere absence of errors.

G1 Healthy / healthy-but-slow

The agent positively confirms a healthy run — the GPUs did real work, the loss decreased, the hardware was clean, the run completed — rather than defaulting to "healthy" on the absence of exceptions. A slow-but-learning run is healthy-but-slow, not diverged.

Example verdict

What the data shows

The run completed cleanly; the GPUs launched kernels and did real compute (SM activity above floor), the loss trended down throughout, and the hardware counters were clean. A latent hardware marker on one GPU, if present, is reported as a secondary "flag for ops" warning, not as a fault.

What it implies

This is a genuinely healthy run — established on positive evidence that the GPU did work and the model learned, not on the mere absence of exceptions. If the duty cycle were low we would record under-utilization as a secondary warning without flipping the verdict, and a run that is slow but still learning is healthy-but-slow, not diverged.

// Observability agent

What we collect for observability

All of the above is powered by grogh, a lightweight on-node observability agent deployed across the GPU nodes of a Slurm-scheduled training cluster. It collects the following with **no instrumentation of the customer's training code** — no SDK, no code changes, no special logging sink:

GPU telemetry (per physical GPU)

- **Memory — used and total framebuffer bytes, and the allocation curve over the run.**
- **Compute — engine-active and SM-active duty cycles (the difference distinguishes a busy-waiting GPU from a productive one).**
- **Health — XID errors, correctable and uncorrectable ECC counts, HBM row-remap counts and the row-remap-failure flag, NVLink errors, clock-throttle reasons (thermal/power), temperature, performance state, and a device health roll-up.**

CUDA and NCCL runtime tracing (kernel-level, via eBPF)

- **CUDA kernel-launch counts and whether a process ever attached to the GPU — the signal that separates an idle hang from real work.**
- **CUDA API-call durations and host-to-device / device-to-host memory-copy volumes.**
- **NCCL communicator and collective state — in-flight collectives, completion counts, and enqueue durations — used to tell an advancing run from a frozen one.**

Job logs (standard output and standard error)

- **Captured at the kernel level and attributed by training rank, node, and process — the loss trajectory, framework progress lines, Python tracebacks, and the distributed launcher's failure summary (including signal-kill summaries) — all without the application writing anywhere special.**

Topology and attribution

- **A continuously-maintained map from container to pod to Slurm job to rank, so every metric, log line, and kernel event ties back to the specific training run and GPU that produced it.**

Slurm scheduler records

- The full job and step lifecycle as events — submitted, started, blocked (with the pending reason), suspended/resumed, requeued, updated, and the terminal disposition (completed, failed, cancelled, timeout, node-fail, preempted) — with exit codes, signals, elapsed time, the node list, restart count, and the submit line.
- Per-node state (idle, allocated, down, drain) and down/drain seconds with reasons; per-node CPU load, memory, and GPU allocation.
- Controller and scheduler health — queue depths, job throughput and failure counters, and RPC stats — plus per-job time limit, priority, and I/O paths.

This breadth is what lets the agent corroborate a cause across stores before concluding — confirming an out-of-memory in both the log traceback and the GPU memory curve, distinguishing a busy-waiting GPU from a productive one by reading SM activity rather than engine activity, or separating a job fault from a node failure by lining the run's end up against the scheduler's node-state record. No single signal is trusted alone; the verdict is only as strong as the evidence that agrees across metrics, logs, and scheduler records.

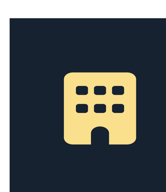
// Get in touch

Contact & next steps

General inquiries

evgeny@apexdata.ai

For product questions, demo requests, and partnership discussions.



Site

apexdata.ai

// Ready to talk?

Start with a 30-minute demo

We'll run the agent live on a sample cluster and walk through a real failure report.

Book a Demo