What Stormlog Is, How It Works, and Why We're Building It

> *A visual guide to GPU memory profiling, telemetry, diagnostics, artifacts, distributed workloads, and inference profiling.*

AUTHOR
Prince Agyei Tuffour

PUBLISHED
October 3, 2026

READING TIME
30 min

COVERS
Stormlog v0.4.0

IN THIS PIECE

01 A job that was fine until it wasn't
02 The memory numbers that matter
03 What Stormlog is
04 Who this is for
05 Profiler or tracker
06 The Stormlog mental model
07 Architecture
08 What Stormlog can see
09 From a running job to evidence
10 Sessions and canonical telemetry
11 Artifacts that outlive the process
12 A leak that looks harmless
13 The rank an average hides
14 Inference: from endpoint numbers to engine evidence
15 Native tools, and where Stormlog stops
16 Shipped, building, researching
17 Try it
18 Keep reading

01

A job that was fine until it wasn't

A training job can look healthy for twenty minutes. Loss goes down, throughput holds, the GPU shows as busy. Meanwhile the memory footprint is quietly changing. Maybe a list somewhere keeps a reference to every step's output. Maybe one rank in an eight-GPU job starts drifting while the other seven stay flat. Then the job dies with an out-of-memory error, and the only evidence you have is the stack trace from the moment of death.

Inference has its own version of this. An endpoint can hit its latency target with one request in flight and behave very differently with thirty-two. The quick test you ran before deploying doesn't tell you where the curve bends, or why.

In both cases you're debugging from a symptom. The cause happened earlier, in a place nobody was recording. Stormlog is our attempt to make that earlier behavior inspectable and reproducible: measure it while it happens, keep the evidence after the process exits, and make it possible to look at the same evidence twice.

I work on Stormlog with Silas Asamoah and Derrick Dwamena. It's open source (MIT), written in Python, and it covers PyTorch, TensorFlow, JAX, and OpenAI-compatible inference endpoints. This page is the explanation I wanted to be able to link to: what it is, how the pieces fit, and where its edges are.

The page is long, and it's built so you can stop early. The first sections explain the problem and the mental model with very little code. The middle covers how Stormlog is put together. The end has runnable examples, an honest list of what Stormlog isn't, and the roadmap. If you'd rather start with commands, go to Try it.

> ***Written against v0.4.0***
>
> *Stormlog v0.4.0[1] was released on 2 October 2026. When this page says something is shipped, it's in that release. Work that exists only on the development branch, or only as a GitHub issue, is labeled that way.*

02

The memory numbers that matter

You don't need to know CUDA to follow the rest of this page. A handful of ideas cover most of it.

- **Device memory**. The GPU's own memory. Weights, activations, gradients, optimizer state, and everything else your job puts on the card live here. It's finite. When a request can't be satisfied, you get an out-of-memory error (OOM).
- **Allocated and reserved**. Frameworks like PyTorch don't ask the driver for memory every time a tensor is created. Their allocator grabs larger blocks and hands out pieces. Allocated is what live tensors use right now. Reserved is what the allocator is holding, including pieces nobody is using at the moment. Reserved is always at least as large as allocated.
- **Whole-device usage**. What the driver reports as used. It includes the allocator's reserve, the runtime's own overhead, and other processes sharing the GPU. This is the number nvidia-smi shows, and it isn't the same as allocated.
- **Peak**. The highest value in a window. An average can look calm while the peak is what runs you out of memory.
- **Growth and retention**. Memory that rises step after step and doesn't come back. Usually something still holds a reference to tensors you're done with. That's retention.
- **Fragmentation**. Reserved memory that exists but is split into pieces too small or too scattered for the next large request. You can hit an OOM with idle memory sitting in the cache.

None of these has to look wrong until the last step. Reserved memory is allowed to stay high. Allocated memory can creep for hours. A job fails when one request for one block can't be satisfied, and that depends on the peak and on how the free space is arranged, not on how the run looked a minute earlier.

Step through five situations below. The percentages are illustrative. They're there to show the idea, not to describe a particular GPU.

FIGURE 1

One GPU, five situations

A horizontal bar represents all device memory, split into tensors in use, idle allocator reserve, other device usage, and free space. Five states show the free space shrinking as tensor memory grows through healthy, growing, retention, near out-of-memory, and out-of-memory.

Illustrative percentages. Select a segment to read what it means.

Drawn as an interactive diagram on the web: see it live at [2].

What Stormlog records here, for the runtimes that expose it, is the shape of the climb instead of only the final error. What each backend exposes differs, and the section on what Stormlog can see gets specific.

03

What Stormlog is

Stormlog is an open-source toolkit for profiling GPU memory and inference behavior. It measures a workload, writes what it saw to local files, and gives you several ways to read those files again: Python APIs, command-line tools, and a terminal interface.

That sentence covers a lot of surface, so here is what the v0.4.0 release contains, with the caveat that matters for each part.

Area	What's there	Caveat
PyTorch	`GPUMemoryProfiler` for bounded profiling, `MemoryTracker` for tracking over time.	Bounded profiling targets torch.cuda runtimes (CUDA and ROCm). MPS and CPU use the tracker or the CPU classes.
TensorFlow	`stormlog.tensorflow`: `TFMemoryProfiler`, `TensorFlowMemoryTracker`, analyzer, visualizer.	Counters can depend on the runtime, especially on Metal.
JAX	`stormlog.jax`: `JAXMemoryProfiler`-, tracker, analyzer, pprof-based graph view.	Device memory appears only when the runtime exposes it. Otherwise it's marked unavailable and process memory is reported separately.
CPU	`CPUMemoryProfiler`, `CPUMemoryTracker`, and a CPU fallback in the CLI.	Useful for checking a workflow without a GPU. It isn't a substitute for device counters.

Command line	`gpumemprof`, `tfmemprof`, `jaxmemprof` (each with info, monitor, track, analyze, diagnose), plus `stormlog query` and `stormlog infer`.	Same command names across frameworks, but the options differ.
Terminal UI	A Textual app with Overview, PyTorch, TensorFlow, Monitoring, Visualizations, Diagnostics, and CLI & Actions tabs.	Startup currently imports PyTorch, so install `stormlog[tui,torch]`.
Telemetry and artifacts	Session-aware events, append-only sinks, diagnose bundles, OOM bundles, rollups.	Inference runs use their own record format.
Analysis	Growth and leak heuristics, hidden-memory gap analysis, cross-rank first-cause suspects.	Heuristics. They point somewhere. They don't prove a cause.
Visual exports	PNG and HTML timelines, heatmaps, dashboards.	Needs the `viz` extra.
Inference	`stormlog infer profile-`, `analyze`, and `collect-server` for OpenAI-compatible Chat Completions endpoints.	Client-observed metrics. Server memory only with the optional host collector.

> ***Backends aren't identical***
>
> *CUDA, ROCm, MPS, TensorFlow, JAX, and CPU expose different counters. Stormlog records what each runtime exposes and marks the rest unavailable instead of filling it in. The table in the signals section shows who exposes what.*

04

Who this is for

People open Stormlog for different reasons. These are the ones we design for. Each card ends with a fragment of the workflow that person would run.

- ML engineer

  Your training or evaluation job uses more memory than you expected, and you want to know where it goes.

  profiler.profile_function(train_step)

- Researcher

  You're comparing experiments, and you need enough evidence kept to reproduce a failure later.

  stormlog query sessions ./artifacts --table

- ML systems and infrastructure engineer

  Long-running jobs, multi-GPU behavior, regressions after an upgrade.

  gpumemprof track --job-id train-42 --rank 1 --world-size 8

- Inference engineer

  Latency, throughput, failures, token counts, workload shape, and memory under controlled load.

  stormlog infer profile --arrival poisson --rate 4,8 ...

- CI and release engineer

  Deterministic diagnostics and machine-readable reports you can archive with a build.

  gpumemprof diagnose --duration 0 --output ./diag # exit 3 on risk

- Open-source contributor

  Collectors, telemetry schemas, analyzers, visualization, runtime support, inference observability.

  stormlog/device_collectors.py # DeviceMemoryCollector

05

Profiler or tracker

Stormlog gives you two kinds of instrument, and choosing between them is mostly a matter of which question you're asking. A profiler answers a bounded question: what happened inside this operation? A tracker answers a time-based one: what happened over time?

FIGURE 2

Two instruments for two questions

A profiler wraps one bounded operation, such as a function or a context block, and returns snapshots and a summary. A tracker samples a whole process on a timer and produces a stream of telemetry events, with alerts when thresholds are crossed.

The profiler's bracket has a start and an end. The tracker's ticks keep going until you stop it.

Drawn as an interactive diagram on the web: see it live at [2].

In practice you often start with a tracker, because you don't yet know where to look. Once the timeline shows when something changes, a profiler around the suspect step tells you more about that step. The two are different code paths in the package, not one tool with two modes.

profile_one_step.py

```
import torch
from stormlog import GPUMemoryProfiler

profiler = GPUMemoryProfiler()
device = profiler.device
model = torch.nn.Linear(1024, 128).to(device)

def train_step() -> torch.Tensor:
x = torch.randn(64, 1024, device=device)
return model(x).sum()

profile = profiler.profile_function(train_step)
summary = profiler.get_summary()

print(profile.function_name)
print(f"Peak memory: {summary['peak_memory_usage'] / (1024**3):.2f} GB")
```

A bounded question. profile_function runs the function once and keeps before and after snapshots. get_summary() aggregates what the profiler has seen.

track_a_run.py

```python
from stormlog import MemoryTracker

tracker = MemoryTracker(sampling_interval=0.5, enable_alerts=True)

tracker.start_tracking()
with tracker.phase("train"):
run_training()  # your workload
tracker.stop_tracking()

stats = tracker.get_statistics()
peak = stats.get("peak_memory")  # None where the backend has no allocator counters
print("Peak allocated:", "unavailable" if peak is None else f"{peak / 1024**3:.2f} GB")
print(f"Events: {stats.get('total_events', 0)}")
```

A time-based question. sampling_interval is in seconds. phase() writes enter and exit events, so later analysis can say which phase a spike belongs to. peak_memory comes from allocator counters, so it is None on a backend without them, and the check keeps the example honest. A clean stop_tracking() marks the session completed.

On a machine without a supported GPU, `GPUMemoryProfiler` and `MemoryTracker` raise a `RuntimeError`. Swap in `CPUMemoryProfiler` or `CPUMemoryTracker` to check your instrumentation on a laptop. They offer the same methods (`profile_function`, `get_summary`, `phase`), but the CPU profile result names the function `name` rather than `function_name`.

Both examples follow the usage guide in the Stormlog documentation[3]. See also the Stormlog project page[4].

06

The Stormlog mental model

Every Stormlog workflow follows one path. A workload produces measurements. A profiler, tracker, or collector reads them. They're normalized into events that carry an identity. Those events are written as artifacts. Analysis, plots, and the terminal interface read the artifacts.

FIGURE 3

How a measurement becomes evidence

Workloads in PyTorch, TensorFlow, JAX, or behind an OpenAI-compatible endpoint are observed by profilers, trackers, and collectors. Memory and runtime measurements become canonical telemetry events with a session identity. Inference measurements become inference JSONL records with their own run identity. Both are written as local artifacts and read by the command line, the terminal interface, reports, visualizations, CI, and later investigation. The terminal interface reads the memory telemetry lane.

Two lanes share the same ideas (local files, session identity, later analysis) but not the same record format. The terminal interface reads the memory lane.

Drawn as an interactive diagram on the web: see it live at [2].

There are two lanes because inference measures different things. A training job reports memory counters from inside the process. An inference run mostly observes a server from the outside, as a client, and records requests. The lanes share the ideas that matter (local files, identity, analysis later) but not the record format. We keep that distinction visible on purpose: inference artifacts are `infer.*` JSONL records, not `TelemetryEventV4`.

Two things the diagram can't show. The TUI isn't a separate analysis engine: it reuses tracker sessions for live data and the same event model for artifacts. And nothing in the path needs a hosted service.

07

Architecture

The code is organized as four packages, and the boundaries follow what you'd guess from the names.

- `stormlog` holds the PyTorch profiler and tracker, the CPU fallbacks, telemetry normalization and the session contract, the analyzers, the visualizer, the device collectors, local query, the TUI, and inference.
- `stormlog.tensorflow` holds the TensorFlow profiler, tracker, analyzer, visualizer, and runtime diagnostics. It has no TUI of its own.
- `stormlog.jax` holds the JAX profiler, tracker, diagnostics, analyzer, visualizer, and runtime helpers.

- `stormlog.infer` profiles OpenAI-compatible endpoints. It's separate from the framework tools because the endpoint might be backed by PyTorch, vLLM, SGLang, TensorRT-LLM, MLX-LM, or a hosted gateway.

FIGURE 4

Stormlog in five layers

From top to bottom: user surfaces, profiling and tracking, analysis and sessions, canonical telemetry and artifacts, and backend collectors. Each layer reads from the one below it. Open a layer for more detail.

A layer talks to the one below it. The TUI sits in the top layer and reads the same telemetry and artifacts as the CLI.

Drawn as an interactive diagram on the web: see it live at [2].

The TUI reads the same model

The `stormlog` command opens a Textual app. It adapts live tracker data through a tracker session and loads saved artifacts as `TelemetryEventV4` records, the same records the CLI reads. Plot export reuses the visualizer. That's why a diagnose bundle written by a CI job can be opened in the TUI Diagnostics tab by someone else later, and why a bug fix in an analyzer shows up in both places.

Collectors declare what they can measure

On the PyTorch side, device memory comes from a collector, and every collector answers three questions: what is the current sample, what can you measure, and what are you called. CUDA, ROCm, and MPS each have one. The tracker checks every sample against the collector's declaration. A populated field the collector said it can't provide counts as a collector failure.

Declaring capabilities lets Stormlog leave a gap as a gap. The easy move for a profiler is to fill missing counters with zeros so charts keep drawing, but a zero is a claim: it says the allocator holds nothing. Stormlog has moved away from that more than once. Always-on tracking exports health events instead of synthetic zero samples (0.3.0). JAX statistics the runtime can't provide are marked unavailable instead of reported as zero (0.3.8). Device-only tracking never fabricates allocator, fragmentation, history, or attribution findings (0.3.10).

SEE THE COLLECTOR CONTRACT

- `sample()` returns a normalized `DeviceMemorySample`. Allocator and device counters may be `None`.

- `capabilities()` returns a frozen `DeviceMemoryCapabilities` describing each supported counter and allocator-native feature.
- `name()` identifies the backend: `cuda`, `rocm`, or `mps`.
- A supported field may be missing from a sample only when collector diagnostics name it as partial.
- Third-party runtimes can pass a collector to `MemoryTracker(collector=...)` without a torch device. No global registry is involved. Device-only collectors keep sessions, distributed identity, sinks, query, and TUI behavior. Allocator events, fragmentation, attribution, native history, and bounded profiling stay unavailable.

Source of truth: the architecture guide[5], which describes the code as it is, not a roadmap.

08

What Stormlog can see

"Memory" is several measurements at different layers, and a backend may expose some of them and not others. It helps to keep the layers apart.

FIGURE 5

Five layers of signal

From the application at the top to the system at the bottom: application, framework, allocator, device, and system. Each layer exposes different measurements, and not every backend exposes every layer.

Stormlog records the layers a backend exposes and labels the rest unavailable.

Drawn as an interactive diagram on the web: see it live at [2].

Runtime	Allocator counters	Whole-device used, free, total	Native allocator history	Bounded profiler
CUDA	Yes	Yes	Yes, opt-in	GPUMemoryProfiler
ROCm	Yes	Yes	No, CUDA only today	GPUMemoryProfiler
Apple MPS	Allocated and reserved	Used; free and total when the runtime reports a maximum	No	No, use MemoryTracker

CPU only	Not applicable	Process and system memory	No	CPUMemoryProfiler
Injected device-only collector	Null, with a declared reason	As the collector declares	No	No

That table is why "supports CUDA, ROCm, MPS, TensorFlow, and JAX" is an incomplete sentence. Support means different things on each. TensorFlow and JAX sit outside the PyTorch collector contract and read their own runtimes:

- TensorFlow runtimes can be CUDA, ROCm, Metal, or CPU. Counters can depend on the runtime on Metal. `tfmemprof info` prints build and runtime diagnostics.
- JAX device memory is read through `jax.Device.memory_stats()` after an XLA sync. It appears only when the runtime reports `bytes_in_use`. Process memory is reported separately.

The point of the layering isn't to apologize for gaps. It's that an investigation can use the layers that exist. A device-only runtime still gives you a device-used timeline, sessions, and query. It just can't give you fragmentation, and it says so.

09

From a running job to evidence

A Stormlog investigation tends to have five stages. They aren't rigid, and you'll often loop between the middle ones.

1. 1Instrument

   Start a tracker in your script, or capture with `gpumemprof track`. Mark phases if you want spikes attributed to them.

2. 2Observe

   Alerts fire when a warning or critical threshold is crossed (`--warning-threshold-`, `--critical-threshold`). The TUI's Monitoring tab shows the same tracker data live.

3. 3Diagnose

   `gpumemprof analyze` reads the saved telemetry. `gpumemprof diagnose` writes a bundle with a verdict and exits 3 when it raises a risk finding.

4. 4Preserve

   Everything above already wrote files. For crashes, `--oom-flight-recorder` keeps a rolling buffer and dumps it when an OOM is recognized.

5. 5Compare or fix

   Query sessions side by side with `stormlog query`, or change the code and capture again. The first capture is still there to compare against.

The thing to notice is the line between stages two and three. Until the process exits, the evidence is a set of numbers in memory. After, it's files.

FIGURE 6

Where the evidence lives before and after the process ends

A training script runs with a MemoryTracker. The tracker turns samples into canonical telemetry events that each carry a session identity. When the run ends, those events exist as files on disk: exports, sink segments, or a diagnose bundle. An analyzer can reload those files later, and the terminal interface, a report, a CI job, or a teammate can use them.

The dashed rule is the point of the whole design. Before it, evidence exists only while the process runs. After it, the evidence is a file.

Drawn as an interactive diagram on the web: see it live at [2].

```terminal
gpumemprof track --duration 30 --interval 0.5 --output track.json --format json
gpumemprof analyze track.json --format txt --output analysis.txt
gpumemprof diagnose --duration 5 --interval 0.5 --output ./diag_bundle
echo $?
```

Capture, analyze, then write a portable bundle. The last line prints diagnose's exit code: 0 when no risk finding was raised, 3 when one was. diag_bundle/report.json carries the same verdict with its findings.

Those commands run on a CPU-only host too, using the CPU fallback, which makes them a reasonable first check of your setup before you point them at a GPU.

10

Sessions and canonical telemetry

Telemetry is only useful across tools if they agree on what an event is. Stormlog has one canonical event and one session contract, and the tracker, the CLI, diagnose bundles, OOM bundles, and the TUI all use them.

- **session_id.** A unique ID for one capture. Every event, diagnose bundle, and OOM bundle from that capture carries or references it.
- **Lifecycle state.** running, completed, interrupted, or incomplete. A clean stop marks a session completed. A process that died while running is recovered as interrupted on the next start.
- **Backend identity.** Which collector produced the event and which runtime it came from, such as cuda, rocm, or mps.
- **Distributed identity.** job_id, rank, local_rank, and world_size, inferred from common launcher environment variables or set explicitly.
- **Normalized counters.** Allocator and device memory in bytes, with null where a backend can't provide a counter.
- **Capability metadata.** A typed object saying which counters and analyses this backend supports. It travels with the data.

The current event is `TelemetryEventV4`. Version 2, version 3, and recognized legacy records are upgraded to it on load, so older artifacts still open.

If you reuse one sink directory across runs, captures stay separate by `session_id`. Analysis defaults to the newest completed session, then the newest interrupted one, then the newest incomplete one, and you can pick another with `--session-id`.

SEE THE EVENT MODEL

one TelemetryEventV4 record (trimmed, illustrative values)


```
{
"schema_version": 4,
"session_id": "2b30f4a4-7d2d-48f7-a9f6-7d40c14eb95e",
"timestamp_ns": 1800000000000000000,
"event_type": "sample",
"collector": "stormlog.cuda_tracker",
"sampling_interval_ms": 500,
"pid": 41873,
```

```
"host": "gpu-node-03",
"job_id": "train-42",
"rank": 2,
"local_rank": 2,
"world_size": 4,
"device_id": 2,
"allocator_allocated_bytes": 6442450944,
"allocator_reserved_bytes": 8589934592,
"device_used_bytes": 10737418240,
"device_free_bytes": 14495514624,
"device_total_bytes": 25769803776,
"context": null,
"metadata": {
"memory_capabilities": {
"backend": "cuda",
"supports_allocator_reserved": true,
"supports_device_used": true,
"supports_native_allocator_history": true
}
}
}
```


Trimmed for reading. The published schema also requires allocator_active_bytes, allocator_inactive_bytes, allocator_change_bytes, and the full set of supports_* booleans. The values here are made up.

On a device-only runtime the allocator fields are `null` and the capability object says why. The published JSON Schema rejects a counter that's declared unsupported but filled in. The loader also checks what a schema can't say, such as `rank` being below `world_size` and used plus free not exceeding total.

11

Artifacts that outlive the process

Artifacts are what make the rest of this useful after the job is gone. In plain terms, an artifact is a file or directory holding what Stormlog saw, in a form something else can read.

Artifact	What it holds	Produced by
Telemetry exports and sink segments	Canonical events as JSON or CSV, or append-only JSONL segments with a manifest, rollover, and retention limits.	`track`, trackers, TUI exports

Diagnose bundle	`environment.json`, `telemetry_timeline.json`, `diagnostic_summary.json`, `manifest.json`, and `report.json`.	`diagnose`
OOM flight-recorder bundle	Recent events from before an out-of-memory error, with a manifest, metadata, and environment. On CUDA, optional native allocator snapshots.	`--oom-flight-recorder`, `capture_oom()`
Visual exports	PNG and HTML timelines, heatmaps, and dashboards.	The visualizer, the TUI, analyze with --visualization
Inference JSONL	A session record, the workload record, request traces, phase windows, cache-state records, optional system samples.	`stormlog infer profile`
Reports	Text or JSON analysis, and a versioned report envelope with a verdict and findings.	`analyze`, `diagnose`
Session metadata	A session ledger in sink manifests. Bundles record the session that owns them.	Trackers and diagnose

That gives a failure five properties it usually lacks:

- **Reloadable.** `gpumemprof analyze ./live_sink` reads a whole directory, not just the last run.
- **Shareable.** Send the directory. The other person sees the same events.
- **Comparable.** `stormlog query summary` groups the same metric by session or by rank.
- **Suitable for CI.** A fixed exit-code table and a `report.json` verdict, so a job can branch without parsing text. See the report contract[6].
- **Reviewable after the process exits.** An OOM bundle holds the events that led to the failure, not only the failure.

Session identity is what holds this together. Without it, a directory with five runs in it is one pile of events. With it, a diagnose bundle's manifest names its session, an OOM bundle points back at the run that produced it, and the TUI can switch between captures instead of merging them. The artifacts article[7] goes through the file layouts in detail, so I won't repeat them here.

Local-first

Everything in the core workflow works on files you choose, in directories you choose. `stormlog query` reads them with no database. The CLI and the TUI run locally. Nothing in the path above sends profiling data to a hosted service.

Some things do leave the machine, and they're opt-in. Weights & Biases and MLflow exports exist as optional extras. Inference profiling sends requests to the endpoint you point it at, which is the point of it. Stormlog doesn't record the API key for an inference run, and it stores a cache-reset URL without credentials or query string.

> ***Look before you share***
>
> *Local files aren't scrubbed files. A policy for scrubbing and suppressing fields in shareable artifacts is still an open investigation (#111[8]), so open a bundle and read it before you send it to someone outside your team.*

12

A leak that looks harmless

Here's a loop where the bug is one line that looks like bookkeeping.

```
train.py

saved_outputs = []

for step, (x, y) in enumerate(loader):
output = model(x)
loss = criterion(output, y)
loss.backward()
optimizer.step()
optimizer.zero_grad()
saved_outputs.append(output)  # looks harmless
```

Nothing errors and the loss goes down. The appended output is still attached to its autograd graph, so every step's activations stay alive.

The fix is usually small, such as appending `output.detach().cpu()`, or not keeping the output at all. The hard part is noticing. A tracker running next to this loop records allocated memory on every interval, and the timeline shows the difference between the two runs immediately.

FIGURE 7

Allocated memory per training step, healthy run versus retention run

Illustrative data. The healthy run climbs for the first few steps and then stays flat near 3 gigabytes. The retention run climbs steadily every step and reaches the 10 gigabyte device capacity at step 40, where an out-of-memory error would occur.

Illustrative, not measured. The curves are generated from formulas to show the shapes. A measured comparison, with the hardware and numbers, is in the leak walkthrough linked at the end.

Drawn as an interactive diagram on the web: see it live at [2].

`gpumemprof analyze` looks at that timeline for growth. Its leak heuristic reports only consistently positive growth, which is why a healthy run that climbs and then levels off isn't flagged.

Stormlog can tell you that memory is climbing and when it started. It can't tell you which line holds the reference. For allocator-level evidence on CUDA there's an opt-in native history mode that records allocator history and attaches best-effort pointer-to-tensor attribution to a diagnose or OOM bundle. The leak walkthrough[9] runs a measured investigation end to end, with the hardware and numbers.

13

The rank an average hides

One GPU number for a whole job is a sum or an average, and both can hide the interesting rank. If one of four ranks drifts, the average moves by a quarter of what that rank did. The chart below makes the point visible: toggle the average and compare it with rank 2.

FIGURE 8

Allocated memory on four ranks of one job

Illustrative data. Ranks 0, 1 and 3 stay near 5.8 gigabytes. Rank 2 begins to climb around sample 22 and ends near 11.6 gigabytes. The average across all four ranks ends near 7.5 gigabytes, so the average shows a much smaller rise than rank 2 does.

Illustrative, not measured. With four ranks, one drifting rank moves the average by a quarter of its own change.

Drawn as an interactive diagram on the web: see it live at [2].

Four pieces make this investigation possible.

- **Rank identity.** Telemetry carries `job_id`, `rank`, `local_rank`, and `world_size`. They're inferred from common launcher environment variables, or set with options such as `--job-id` and `--rank` on `gpumemprof track`.
- **Aligned telemetry.** `gpumemprof analyze` merges per-rank timelines. With `--visualization` it writes a cross-rank timeline plot.
- **Loading all ranks at once.** The TUI Diagnostics tab loads rank artifacts, keeps ranks separate, and renders per-rank timelines and a rank table.
- **First-cause suspects.** The analyzer ranks which rank and phase spiked first. These are ranked heuristics. When phases overlap across threads, Stormlog marks the attribution ambiguous instead of guessing.

The distributed diagnostics article[10] walks through the rank-aware workflow in detail.

14

Inference: from endpoint numbers to engine evidence

Stormlog started with training memory, and it isn't only that anymore. Inference raises a different class of questions, and most of them are about behavior under load rather than a single peak.

- Time to first token (TTFT) and end-to-end latency
- Throughput, and how it changes with concurrency
- Token counts, and whether they came from the server or an estimate
- Failures, timeouts, and rejected requests
- Workload shape: arrival pattern, prompt lengths, shared prefixes
- Cache state, and whether the cache was cold
- Device memory while all of that happens
- What the serving engine itself was doing

The tool for it is `stormlog infer`. It sends controlled traffic to any endpoint that accepts an OpenAI-style Chat Completions request, which covers PyTorch servers, vLLM, SGLang, TensorRT-LLM, MLX-LM, and hosted gateways, and it records what a client can observe.

FIGURE 9

The request path, and what Stormlog sees at each hop

A request travels from the Stormlog client through an OpenAI-compatible endpoint to a serving engine, a server process, and finally the GPU and runtime. Stormlog v0.4.0 observes the client-visible hops and, optionally, host-side process and device memory. Serving-engine scheduler and cache signals exist for vLLM on the development branch and are not released. Other engines are planned.

Each badge is a text label for the state of that signal, so status never depends on color.

Drawn as an interactive diagram on the web: see it live at [2].

Available today

These are in v0.4.0 and documented in the inference guide[11]:

- A workload matrix over concurrency, input length, and output length, streaming or not, with warmup requests recorded but excluded from analysis.
- Controlled arrivals: closed loop, fixed rate, seeded Poisson, bursts, and replay of a recorded trace. Open-loop runs also report latency from each request's intended arrival, so time spent waiting for a slot stays visible.
- Prompt control: repeat, unique, or shared-prefix prompts, and a way to ask for a cold prefix cache by calling a reset URL before each case.
- A workload record with a seed and a digest, so the same workload sent to two engine configurations can be recognized as the same.
- Client-observed latency percentiles, TTFT for streaming responses, throughput, failure rate, and token counts with their source recorded on every request.
- An optional collector, `stormlog infer collect-server`, that runs on the serving host and records server process memory and whole-device or MIG memory. The analyzer joins it to a case only when server identity and clocks line up, and it never sums values from different collectors.
- A fixed exit-code table, so a CI job can tell "no request succeeded" apart from a usage error.

> ***Where the limits are***
>
> *TTFT is measured on the client from request start to the first non-empty streamed content, and non-streaming runs have none. Chunk timing is chunk-level, not token-level. A requested cold cache is recorded as `unverified`, because no engine adapter can read the cache yet. Whole-device memory*

> *isn't a per-request cost.*

```
terminal

stormlog infer profile \
--base-url http://localhost:8000/v1 \
--model Qwen/Qwen2.5-7B-Instruct \
--arrival poisson --rate 2,4,8 --duration 60 \
--prompt-mode unique --seed 7 \
--input-tokens 512 --output-tokens 128 \
--output artifacts/infer_poisson.jsonl

stormlog infer analyze artifacts/infer_poisson.jsonl
```

Three arrival rates, each sent for 60 seconds on a schedule fixed before the run starts. unique prompts avoid accidental prefix-cache hits, and the seed makes the run repeatable.

From endpoint measurements to serving-engine evidence

A client can see that latency rose. It can't see why. The active roadmap (#210[12]) is about closing that gap by joining five kinds of evidence:

1. 1a request
   what the client sent and saw
   released
2. 2server, process, and GPU identity
   which machine and device the numbers belong to
   released, optional collector
3. 3scheduler and cache behavior
   queueing, KV-cache pressure, prefix-cache hits
   vLLM on the development branch
4. 4bounded GPU activity
   a trace window around the incident
   in development
5. 5an evidence-backed explanation
   findings with their evidence and limits
   planned

Two caveats keep this honest. A batch is shared by many requests, so Stormlog's correlation records represent shared execution through membership. They keep measured batch duration separate from any estimated per-request cost, and don't assign a shared kernel to one request. And an aggregate metrics scrape stays aggregate evidence. Exact request-to-iteration attribution needs worker instrumentation, which is a separate piece of work.

What exists beyond v0.4.0 is on the development branch and listed under "Unreleased" in the changelog[13]: scraping vLLM's Prometheus metrics during a run, receiving vLLM request spans, and importing a vLLM execution hook's log into request, iteration, and membership records. That's vLLM only, and it isn't in a release yet.

15

Native tools, and where Stormlog stops

Native tools are good, and you should keep using them. Different tools answer different layers of the problem.

Tool	Strong at	How Stormlog relates
`nvidia-smi`	Whole-device memory and utilization, and which processes use the GPU, right now.	Stormlog reads device counters through collectors too, but samples them over time next to allocator counters and attaches session identity.
PyTorch memory APIs and snapshots	Exact allocator counters and, on CUDA, allocator history.	Stormlog builds on those counters and adds sampling, alerts, artifacts, and analysis. Its native-history mode writes PyTorch's allocator snapshots into a bundle.
TensorFlow and JAX profilers	Deep framework-specific inspection: ops, compilation, device memory profiles.	Stormlog gives one workflow and artifact shape across frameworks. On JAX, the OOM recorder attaches the runtime's own device memory profile.
Nsight and ROCm tools	Kernel timelines and GPU-level detail.	A different layer. Stormlog doesn't replace it. The inference roadmap looks at linking bounded traces to incidents.
System monitors and exporters	Host and fleet-wide metrics.	Stormlog is per-workload evidence. Ingesting DCGM readings as optional server telemetry is an open issue (-#247[14]).

What Stormlog adds is mostly about everything around the measurement: a common workflow across frameworks, normalized telemetry, artifacts that survive the process, session-aware investigation, Python, CLI, and TUI paths to the same data, automated diagnostics, repeatable runs, CI-friendly output, and inference workload profiling. That's a different job from reading a kernel timeline.

What Stormlog is not

- A replacement for every vendor GPU profiler.
- A hosted observability platform.
- A root-cause oracle. Its analyzers are heuristics that point at evidence, and findings carry confidence and limits.
- A guarantee that every backend exposes the same counters.
- A reason to skip the native tools for your framework or runtime.

- An LLM that guesses what went wrong. The measurements and labels don't depend on a model.

16

Shipped, building, researching

It's easy to blur "works today", "being built", and "we'd like to find out". Here they're kept apart. The three columns are the same split the project uses, and the inference roadmap itself says its milestones describe outcomes, not release versions or dates. I'm following that: nothing below has a ship date.

Shipped

In v0.4.0 or an earlier release.

- PyTorch, TensorFlow, and JAX profilers and trackersJAX since 0.3.5.
- Sessions, TelemetryEventV4, append-only sinks, rollups
- Diagnose bundles with report.json and a fixed exit-code tableNew in 0.4.0.
- OOM flight recorder, and opt-in CUDA allocator history
- Rank-aware analysis and TUI Diagnostics
- Local query layer, including correlate and run catalogs
- Controlled arrivals, prompt modes, and cold-cache requests for inference#[15] 212[15]New in 0.4.0.
- Host server telemetry with clock alignment and tensor-parallel groups#[16] 214[16]0.3.10.

Building

Concrete items on the inference roadmap (#210). Some are merged but unreleased.

- vLLM metrics and request spans during a run#[17] 215[17]On the development branch. Not in a release.
- GPU execution capture, and linking requests to scheduler iterations and GPU activity#[18] 216[18], #[19] 217[19]The vLLM importer is listed as Unreleased. Capture work is on feature branches.

- SLO goodput and repeatable baseline comparisons#[20]
  213[20]
- Evidence-backed explanations of incidents#[21]
  218[21]
- Incident capture with bounded history#[22]
  219[22], #[23]
  220[23], #[24]
  221[24]
- SGLang, TensorRT-LLM, and TensorRT, each with its own capability matrix#[25]
  222[25], #[26]
  223[26], #[27]
  224[27]
- A local Web UI for inference profiling#[28]
  227[28]Design and a prototype with synthetic fixtures.

Researching

Experiments. No implementation is promised, and "reject" is an acceptable result.

- Optional native probes: CUPTI, USDT or eBPF, programmable GPU probes#[29]
  118[29], #[30]
  235[30]
- Workload-aware incident detection against simpler baselines#[31]
  225[31]
- Per-request GPU cost estimates under shared batching#[32]
  226[32]
- A scrubbing policy for shareable artifacts#[8]
  111[8]
- Native allocator debugging on MPS and ROCm#[33]
  97[33]
- TorchTPU support, once its public runtime exists#[34]
  125[34]

Execution correlation, without double counting

The contract for relating a request to shared server iterations and GPU activity shipped in 0.3.10 (#211[35]). It reports iteration elapsed time, summed activity duration, and merged GPU interval time as three different numbers. The rule behind it: shared execution isn't falsely assigned to one request. Only the collector side is still being built, and the first target is vLLM.

A local Web UI

#227[28] describes a local UI for investigating a slow or failed inference run through its raw requests and metrics, with every displayed value traceable to a field or a calculation. It's designed to sit on Stormlog's existing artifacts and report contracts behind a small service boundary, not to reimplement analysis in the frontend. A prototype exists (PR #228[36]) with explicitly synthetic fixtures. It's a design and data-contract exercise, not something you can install.

Native probes are research

#118[29] asks whether an optional native collector can provide evidence, or lower collection cost, that the existing PyTorch and Nsight route can't. Candidates include CUPTI activity collection, USDT or eBPF probes, and programmable GPU probes. None is a dependency of Stormlog, none is committed, and a negative result is a valid way for that issue to end.

17

Try it

Install the package, then add extras for the runtime you use.

```
terminal

pip install stormlog

# Pick the extras you need:
pip install "stormlog[torch]"            # PyTorch
pip install "stormlog[tf]"               # TensorFlow
pip install "stormlog[jax]"              # JAX
pip install "stormlog[viz]"              # PNG and HTML plots
pip install "stormlog[tui,torch]"        # terminal UI
pip install "stormlog[infer-tokenizers]" # better token counts for inference
pip install "stormlog[all]"              # everything
```

Then a ten-second capture, an analysis, a diagnose bundle, and a look at what was saved. This

runs on a CPU-only laptop too.

```
terminal

pip install "stormlog[torch]"

gpumemprof info

gpumemprof track \
--duration 10 \
--interval 0.5 \
--output run.json \
--format json

gpumemprof analyze run.json --format txt --output analysis.txt
gpumemprof diagnose --duration 0 --output ./diag
stormlog query sessions ./diag --table
```

info shows which backend Stormlog detected. diagnose --duration 0 writes a bundle without a new tracking window. query lists the session that bundle owns.

For inference, point the profiler at any OpenAI-compatible endpoint you're allowed to send traffic to:

```
terminal

stormlog infer profile \
--base-url http://localhost:8000/v1 \
--model Qwen/Qwen2.5-7B-Instruct \
--concurrency 1,4,8 \
--requests 20 \
--output infer.jsonl

stormlog infer analyze infer.jsonl
```

A closed-loop run at three concurrency levels, 20 measured requests per case.

To see everything in one place, `pip install "stormlog[tui,torch]"` and run `stormlog`. Docs, source, and the package page:

- Documentation[3], including the production cookbook[37] and the CPU compatibility guide[38]
- GitHub[39], where issues and the roadmap live
- stormlog on PyPI[40]
- Getting started[41], the step-by-step version of this section

HOW I CHECKED THESE COMMANDS

I ran the CPU-only sequence above, and the `query` command, against a fresh install of stormlog 0.4.0 from PyPI on a machine without a GPU. The inference command ran against a local stub server, so I checked its options and artifact output, not its behavior against a deployed model. I ran the tracker example through `MemoryTracker` with an injected device-only collector, which is how I found that `peak_memory` is `None` when a backend has no allocator counters. `GPUMemoryProfiler` refuses to start without a supported accelerator, so I checked its calls against the 0.4.0 source instead of running it, and I haven't run either class on a GPU.

18

Keep reading

This page is the map. These are the places it points to, and what each one teaches.

- Introducing StormlogThe original case for keeping evidence after the first crash.[42]
- Getting startedInstall, instrument, and run a first profile with the CLI, the API, and the TUI.[41]
- Memory leak walkthroughA leak investigation with measured numbers, hardware, and a fix.[9]
- Artifacts explainedWhat each exported file contains and why.[7]
- Distributed diagnosticsRank-aware analysis for multi-GPU runs.[10]
- JAX and inference profilingThe release article for the JAX tracker and the inference endpoint profiler.[43]
- DocumentationArchitecture, API, CLI, TUI, inference, cookbook, and compatibility pages.[3]
- GitHubSource, issues, the inference roadmap, and the changelog.[39]

More from me on Stormlog: what I learned walking through the tutorial[44], the OOM flight recorder[45], and a leak that only showed on Apple MPS[46].

• • •

That's the piece. If it earned a margin note, I'd love to hear it.

prince.agyei.tuffour@gmail.com

LINKS, FOR WHEN YOU'RE READING ON PAPER

[1] https://github.com/Silas-Asamoah/stormlog/releases/tag/v0.4.0

[2] https://princeagyeituffour.com/writing/what-is-stormlog

[3] https://stormlog.readthedocs.io/en/latest/

[4] https://princeagyeituffour.com/projects/stormlog

[5] https://stormlog.readthedocs.io/en/latest/architecture.html

[6] https://stormlog.readthedocs.io/en/latest/report_contract.html

[7] https://www.stormlog.dev/blogs/artifacts-explained

[8] https://github.com/Silas-Asamoah/stormlog/issues/111

[9] https://www.stormlog.dev/blogs/memory-leak-walkthrough

[10] https://www.stormlog.dev/blogs/distributed-diagnostics

[11] https://stormlog.readthedocs.io/en/latest/inference.html

[12] https://github.com/Silas-Asamoah/stormlog/issues/210

[13] https://github.com/Silas-Asamoah/stormlog/blob/main/CHANGELOG.md

[14] https://github.com/Silas-Asamoah/stormlog/issues/247

[15] https://github.com/Silas-Asamoah/stormlog/issues/212

[16] https://github.com/Silas-Asamoah/stormlog/issues/214

[17] https://github.com/Silas-Asamoah/stormlog/issues/215

[18] https://github.com/Silas-Asamoah/stormlog/issues/216

[19] https://github.com/Silas-Asamoah/stormlog/issues/217

[20] https://github.com/Silas-Asamoah/stormlog/issues/213

[21] https://github.com/Silas-Asamoah/stormlog/issues/218

[22] https://github.com/Silas-Asamoah/stormlog/issues/219

[23] https://github.com/Silas-Asamoah/stormlog/issues/220

[24] https://github.com/Silas-Asamoah/stormlog/issues/221

[25] https://github.com/Silas-Asamoah/stormlog/issues/222

[26] https://github.com/Silas-Asamoah/stormlog/issues/223

[27] https://github.com/Silas-Asamoah/stormlog/issues/224

[28] https://github.com/Silas-Asamoah/stormlog/issues/227

[29] https://github.com/Silas-Asamoah/stormlog/issues/118

[30] https://github.com/Silas-Asamoah/stormlog/issues/235

[31] https://github.com/Silas-Asamoah/stormlog/issues/225

[32] https://github.com/Silas-Asamoah/stormlog/issues/226

[33] https://github.com/Silas-Asamoah/stormlog/issues/97

[34] https://github.com/Silas-Asamoah/stormlog/issues/125

[35] https://github.com/Silas-Asamoah/stormlog/issues/211

[36] https://github.com/Silas-Asamoah/stormlog/pull/228

[37] https://stormlog.readthedocs.io/en/latest/cookbook/index.html

[38] https://stormlog.readthedocs.io/en/latest/cpu_compatibility.html

[39] https://github.com/Silas-Asamoah/stormlog

[40] https://pypi.org/project/stormlog/

[41] https://www.stormlog.dev/blogs/getting-started

[42] https://www.stormlog.dev/blogs/introducing-stormlog

[43] https://www.stormlog.dev/blogs/jax-and-inference-profiling

[44] https://princeagyeituffour.com/writing/stormlog-tutorial-walkthrough

[45] https://princeagyeituffour.com/writing/oom-flight-recorder-deep-dive

[46] https://princeagyeituffour.com/writing/mps-memory-leak

Typeset from the live article at princeagyeituffour.com/writing/what-is-stormlog. The web version may be newer and carries the interactive parts.