Serving a 27B model to seven concurrent agents on one 32 GB GPU

Engine choice, GPU sharing and the incidents that shaped it.

Bogdan-Stefan Boghea · Site Reliability Engineer · linkedin.com/in/bstefan93 · November 2026

I designed and built a self-hosted LLM inference and agent platform on a single workstation. It serves a 27-billion-parameter model to up to seven AI agents at once, shares the same GPU with image, video and voice jobs, and makes every engine pass a qualification gate before it may serve. This page is the engineering record: what I measured, what I decided and why, how the system evolved over eleven months, and what went wrong.

> A personal project; the code is private. I am the sole architect and made every decision. AI coding agents wrote most of the code, under review rules I designed (section 10). Every number states the conditions it was measured under.

~575 tokens/s
seven agents generating at once
peak rate; whole runs averaged ~360-370; zero preemptions in 2 of 2 runs

~130 tokens/s
a single request
27B dense model, 234K-token context window

~17 s
to wake the model after a media job
against 2-6 minutes for a cold start

~25% sooner
vLLM against llama.cpp with three agents at once
one repetition, synthetic workload; llama.cpp won for one agent

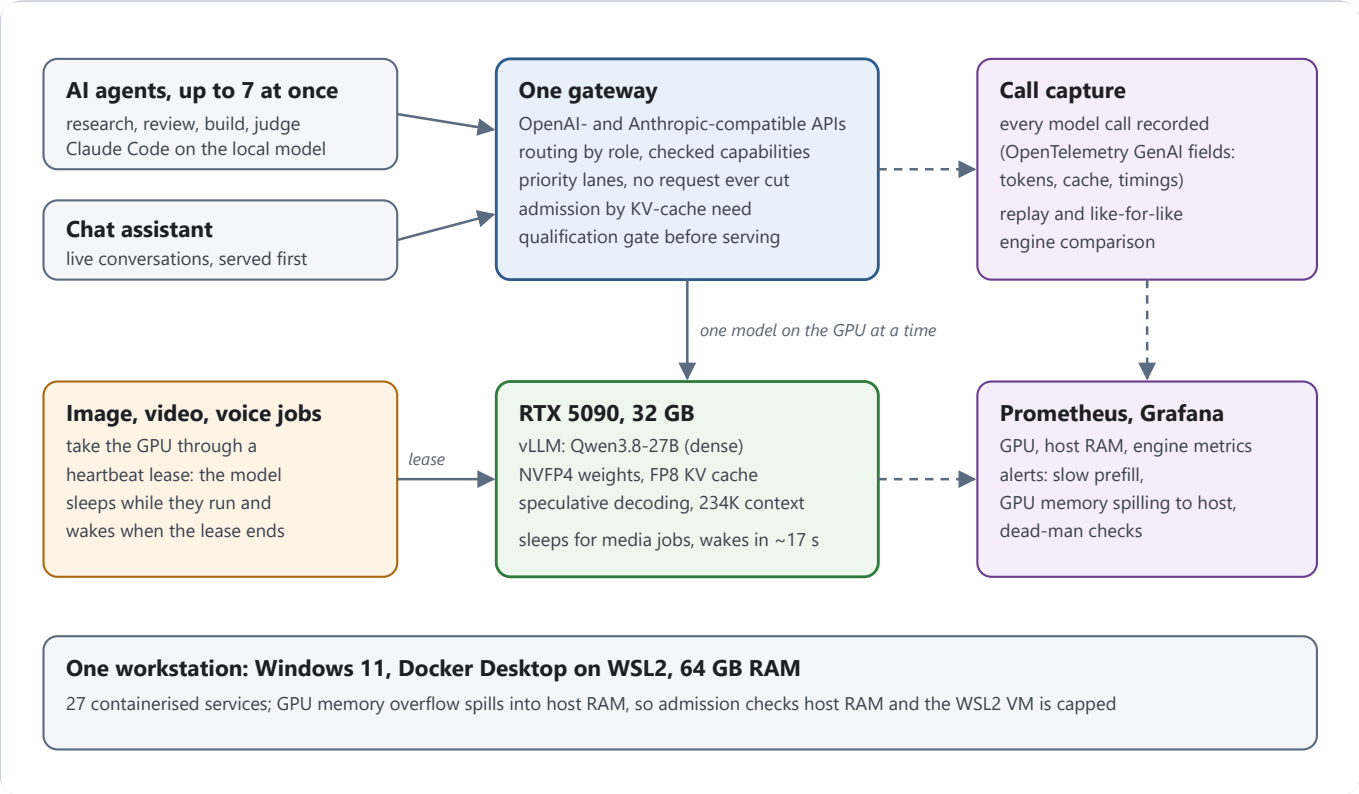
12 checks + a golden set
before any engine or model may serve
each pass bound to the exact configuration

CONTENTS

1. The system
2. How it evolved
3. The decisions
4. Choosing the engine
5. Capacity
6. Sharing the GPU
7. Trust before traffic
8. Observability and operations
9. What went wrong
10. How it was built
11. Limitations
12. What I would do next

1. The system

GPU	NVIDIA RTX 5090, 32 GB
Host	Windows 11, Docker Desktop on WSL2, 64 GB RAM
Model	Qwen3.8-27B (dense), NVFP4 weights, FP8 KV cache, MTP speculative decoding
Engine	vLLM, pinned image, 234K-token context window
Clients	AI agents (Claude Code sessions pointed at the local model), a chat assistant, media pipelines
Scale	27 containerised services (19 always on, 8 started on demand); about 17,000 automated tests; 360 written decision records



Architecture: agents and media jobs share one GPU through one gateway.

The question. Can one consumer GPU serve several agents at once, with predictable latency, while it also runs image, video and voice jobs, and without anyone babysitting it? The agents are the demanding client: they send long prompts, tens of thousands of tokens of context, and several run at once.

Why one workstation? Cost and control. It is also a hard constraint worth engineering against: one GPU, one model resident at a time, and a host whose RAM is shared with the GPU's overflow.

How to read the numbers. Everything is measured on this one machine, so no latency includes a network hop. Load tests use synthetic, agent-shaped conversations; the prompt-reading rates come from recorded live calls. Each chart and table says which, with the date and the configuration.

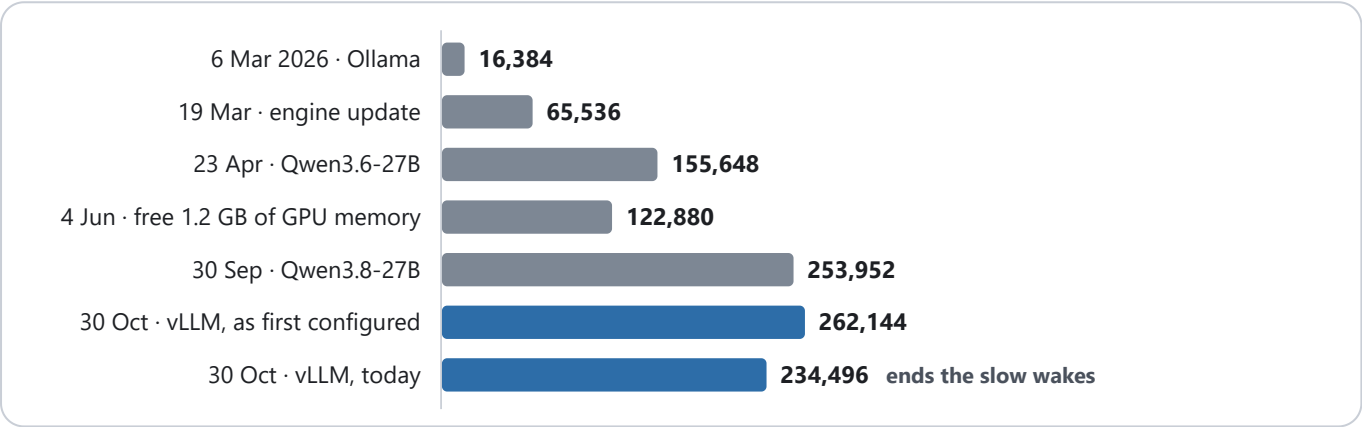
2. How it evolved

The platform did not start as a platform. It started in March 2026 as a streaming proxy in front of one local model, and grew one measured decision at a time: four serving engines, three main models, three designs for sharing the GPU.

Dec 2025	**The project starts** The repository is created; the first code follows in March.
Mar 2026	**First working stack** An OpenAI-compatible streaming proxy in front of a local model; the backend moves to Ollama within the first week. Context window: 16K tokens, 65K by the end of the month.
Apr 2026	**A 27B dense model and the first GPU sharing** Qwen3.6-27B replaces a 32B model on 23 April, with a 156K window. The model is unloaded whenever an image or video job needs the card.
Jun 2026	**"GPU out of memory" traced to host RAM** Two large models were stacked; the overflow spilled into host memory and took down the VM that runs every container. Admission moved to host RAM (section 9).
Aug 2026	**Hosted CI replaced by a local release gate** The checks that decide a merge run on the workstation itself; the hosted pipeline is removed.
30 Sep 2026	**The model is chosen by bench: Qwen3.8-27B** Three candidates compared on the largest window that fits, speed, a 290-step tool chain and a golden set. Recording every model call is decided the same day.
3-27 Oct 2026	**Engine change 1: Ollama to llama.cpp** Decided on 3 October on a measurement (NVFP4 weights on llama.cpp: 127 against 85.5 tokens/s on long answers, same model); llama.cpp serves the main model from 27 October.
9-14 Oct 2026	**Monitoring, a GPU lease and one gateway** Prometheus, Grafana and host exporters go live; one lease decides who holds the card; roles, engine adapters and admission move behind one API. SGLang joins as a third engine for the reviewer role.
21-23 Oct 2026	**Local agents** An Anthropic-compatible route lets Claude Code sessions run on the local model; the first review agent runs.
27-30 Oct 2026	**Engine change 2: llama.cpp to vLLM** Seven measured cells on a 102-request workload: vLLM wins with three agents (section 4). Sleep and wake replaces stop and start; the window goes from 262K to 234K; seven agents pass with zero preemptions.
2-3 Nov 2026	**Admission by KV-cache need** A queue with priority lanes replaces a plain cap on concurrent requests.

3-7 Nov 2026	**The 21-hour slow-prefill incident** Prompt reading ran 16x slower while everything else looked normal (section 9). llama.cpp and SGLang are retired on 4 November; alerts for slow prefill and for GPU memory spilling into host RAM follow on 7 November.
8-9 Nov 2026	**The SSD cache tier is dropped** A waiting agent will resume from its saved conversation instead (section 5).
16 Nov 2026	**Prefill re-measured from 5,987 live calls** My own older figure had gone stale.

The context window is the simplest trace of that evolution. It moved with every model, every engine and every failure, because it is a memory budget and not a setting:



Context window the main model served with, in tokens, at each change. Grey: Ollama. Blue: vLLM. The three days on llama.cpp (27-30 October) are not shown.

3. The decisions

Each row is a decision I made on a measurement. The sections after it show the evidence.

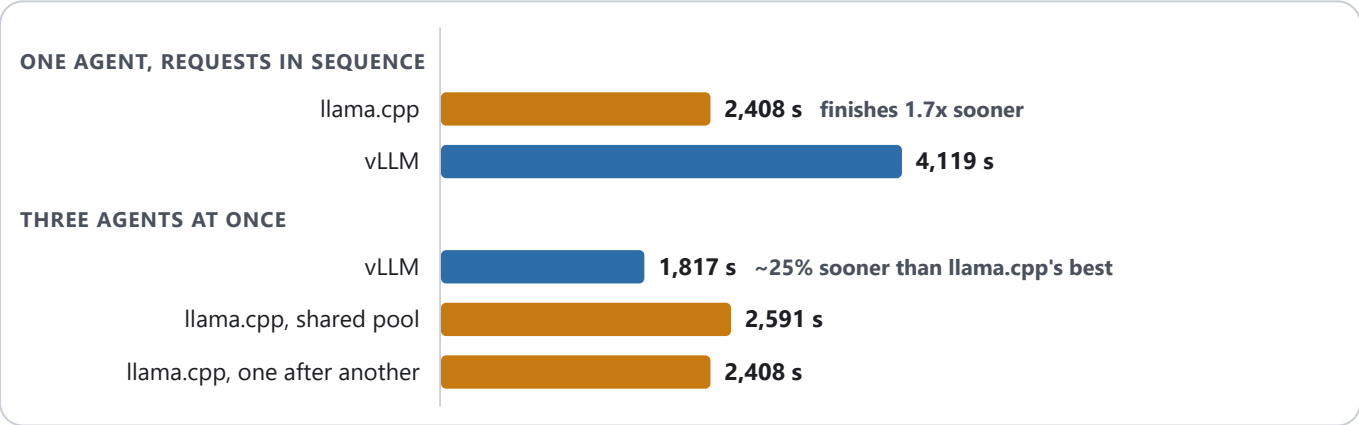
Decision	Alternatives	What was measured	Outcome
Model: Qwen3.8-27B	Two other candidates, one a faster 35B mixture-of-experts model	A 290-step tool chain: 290 of 290 steps with 0 errors; the faster model stopped at step 64	Correctness on long tool chains over raw speed
Weights: NVFP4, FP8 KV cache, MTP speculative decoding	Q4_K_M weights; n-gram speculation	NVFP4 on llama.cpp against Q4_K_M on Ollama: 127 vs 85.5 tokens/s. N-gram speculation ran the same job 2.4x slower (346 s vs 145 s)	Kept since
Engine: vLLM	llama.cpp, SGLang, Ollama	One agent: llama.cpp finishes 1.7x sooner. Three agents: vLLM ~25% sooner, and llama.cpp's prompt reading fell 63%	vLLM serves the main model; llama.cpp and SGLang were retired on 4 November
Concurrency cap: seven	Eight	Eight agents: 168 preemptions with the KV cache at 100%. Seven: zero preemptions in 2 of 2 runs	An eighth request waits in the queue

Decision	Alternatives	What was measured	Outcome
Context window: 234K, down from 262K	Keep 262K and retry slow wakes; a 500 MiB smaller cache; a settle delay	Slow wakes: 1 in 3 before; 0 in 12 after a 1 GiB cut; 2 in 12 after the 500 MiB cut; 4 in 8 with the delay	10.5% less window; the largest call ever recorded is 141K tokens
GPU sharing: sleep and wake	Stop and restart the engine	Restart: 163-170 s. Wake: ~17 s	Sleep and wake; a stop is for recovery only
Long sessions: resume from the saved conversation	KV cache on SSD; KV cache in host RAM	Re-reading a 150K-token conversation takes ~40 s	Disk tier dropped; a waiting agent will hold no memory
Local code reviewer: advisory only	Let it block merges	~70% of its high-severity findings held on re-check, fully or in part (14 of 20), against ~99% for Claude Sonnet (175 of 176)	It never blocks a merge

---

4. Choosing the engine

I compared llama.cpp and vLLM on a **synthetic, agent-shaped workload**: 102 requests in three conversations of 34 turns each, with prompts of 5-60K tokens. It is not a recording of real traffic.



Time to finish the 102-request workload, in seconds (shorter is better).

Conditions: RTX 5090, NVFP4-class weights on both engines, the 102-request synthetic workload, one repetition per cell, 27-28 October 2026; vLLM on an earlier configuration than today's.

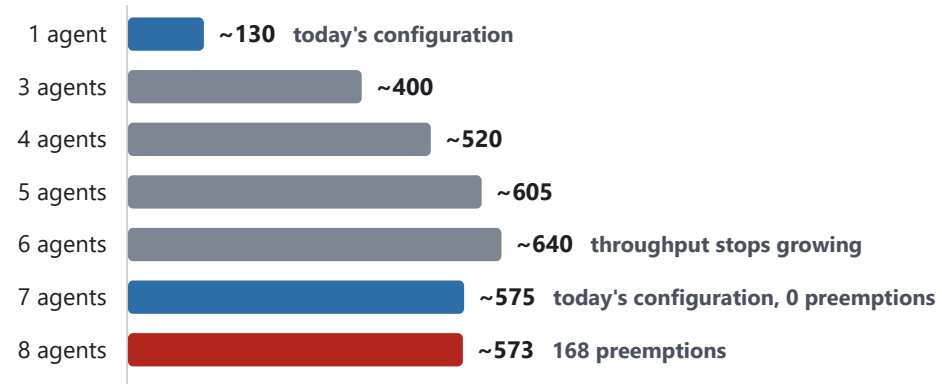
For a single user, llama.cpp is clearly faster. With several agents at once, vLLM's continuous batching and paged KV cache win, and llama.cpp's shared pool was even slower than running the three conversations one after another. The reason showed in prompt reading: llama.cpp read 2,314 tokens/s with one agent and 863 tokens/s in total with three.

The rule written in advance would have waited for a replay of real traffic; I decided on the measured result instead, since concurrent agent work is what the platform is for. **vLLM became the primary engine.**

The serving layer is engine-agnostic: llama.cpp, SGLang and Ollama have each served through the same adapter interface, so changing the engine was a configuration change. Ollama still serves vision (section 7).

5. Capacity

Generation. Aggregate throughput grows with each agent up to six and then flattens. At eight, the KV cache is full and the engine starts preempting requests.



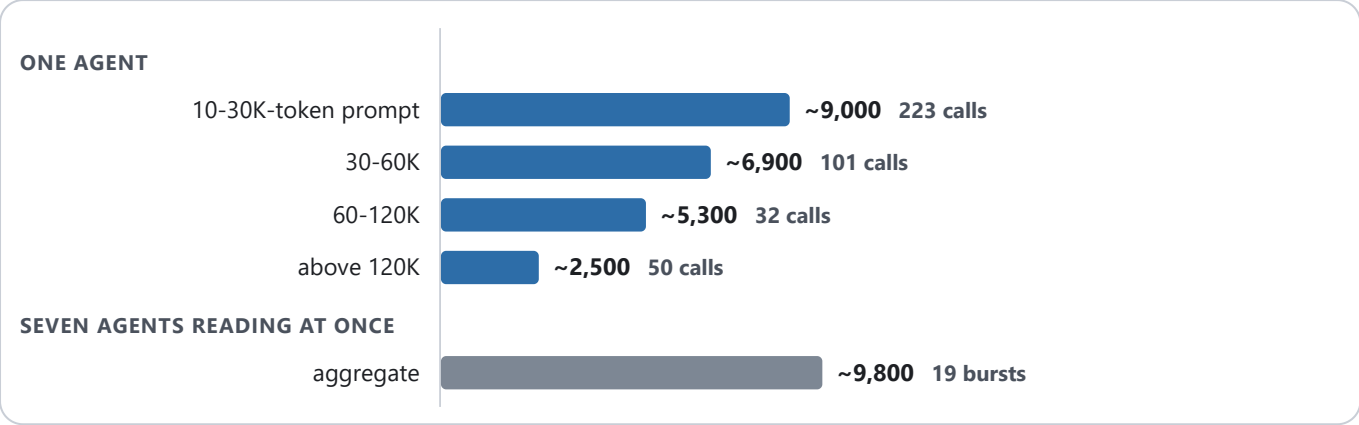
Aggregate generation rate by number of agents generating at once (tokens/s). Blue: today's configuration. Grey and red: the earlier configuration (see the conditions under the table).

Agents	Per agent, tokens/s	Time to first token, median / p90	KV-cache peak	Preemptions	Configuration
1	~130	0.56 s / 0.76 s	17%	0	today's
3	127-135	0.57 s / 0.74 s	42%	0	earlier
4	129	0.63 s / 1.57 s	56%	0	earlier
5	121	0.60 s / 0.97 s	64%	0	earlier
6	107	0.64 s / 2.04 s	75%	0	earlier
7	82	2.15 s / 4.44 s	94%	0	today's
8	72	1.16 s / 5.57 s	100%	168	earlier

Conditions: synthetic multi-turn conversations, 10 turns per agent, 5-20K-token prompts, prefix cache reset before each run, 29-30 October 2026. "Today's" is the 234K-window configuration that serves now; "earlier" is the 262K-window configuration with a larger KV pool that was measured the day before. Two agents were never measured, and five and six were not re-run on today's configuration. The seven-agent row is one of two runs; the repeat had 1.38 s / 4.28 s, a 97.5% peak and zero preemptions. ~575 is the rate while all seven decode at once; the whole runs averaged ~360-370, because agents do not all decode all the time.

Seven is the last clean point. On today's configuration the margin is thin (a 94-97.5% KV-cache peak), so the eighth request waits instead of forcing preemptions. Seven agents ran slower on today's configuration than on the earlier one, but the two runs differ from each other as much as from the earlier configuration, so I do not attribute the drop to the smaller cache.

Prompt reading (prefill). Reading the prompt, not generating the answer, is what a long agent conversation spends its time on.



Prompt-reading rate by prompt length (median tokens/s, uncached prompt tokens only).

Conditions: 406 single-agent calls and 19 multi-agent bursts drawn from 5,987 live calls, 2026-11-06 to 11-16; cache hits excluded; rate = uncached prompt tokens / time to first token, so it is conservative.

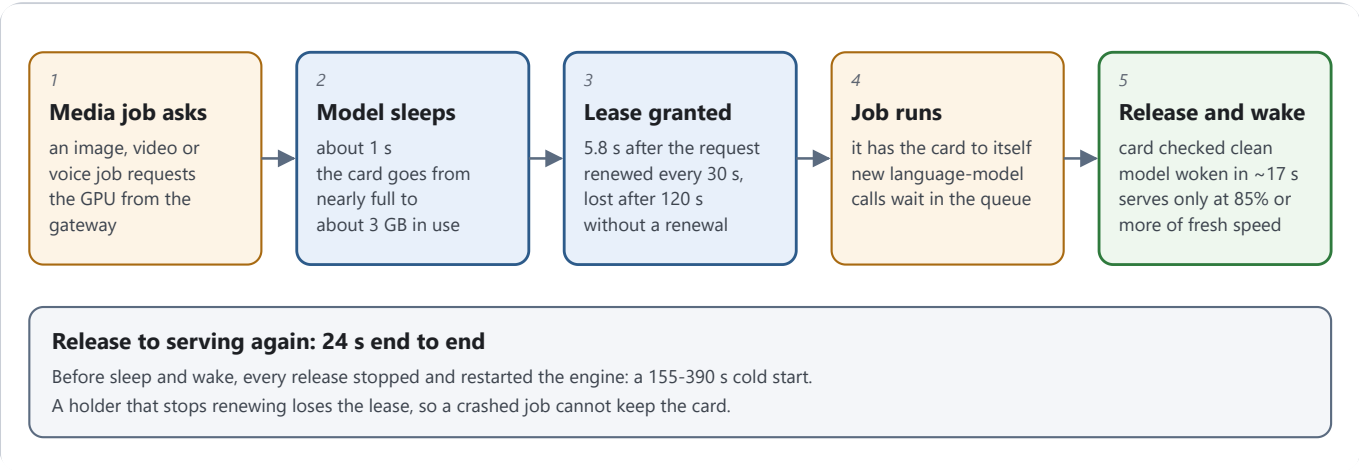
Prefill slows with prompt length because attention cost grows with it, which is why a long conversation is the expensive one.

The admission queue admits requests by priority and by their estimated KV-cache need, measured against the engine's live free-cache gauge. Live conversations go first, a waiting request ages upward, and a running request is never cut.

Long sessions without holding memory. Agents keep long conversations, so I weighed keeping their KV cache on SSD against simply re-reading the saved conversation when an agent resumes (~40 s for a 150K-token context, measured). I dropped the disk tier: it would save only minutes of GPU time a week against heavy daily SSD writes. The design, now being built, is that a waiting agent holds no GPU or host memory at all: the simpler design won on measurement.

6. Sharing the GPU

Image, video and voice models need most of the card, so they cannot share it with a language-model engine that fills nearly all of the 32 GB. Sharing went through three designs. First the model was unloaded for every media job (April). Then a lease stopped the engine and proved it down (October), which was cheap while an engine loaded in seconds but cost a 155-390 s cold start per release once vLLM served. Today's design is sleep and wake (30 October).



How a media job borrows the GPU and gives it back. Times are from one live job on 30 October 2026.

Media jobs take the GPU through a **heartbeat lease**. The gateway puts vLLM into sleep mode, grants the lease, holds new language-model calls while it lasts, and wakes the model when it ends. A holder that stops renewing loses the lease.

- **Wake:** ~17 s (16.8-18.5 s measured), against 2-6 minutes for a cold start.

Two silent regressions: both were found by measuring, not by errors.

1. **Slow first wakes:** about 1 wake in 3 decoded at 42-85 tokens/s instead of ~130. After I shrank the KV cache by 1 GiB (context window 262K to 234K tokens), it fell to 0 in 12. That was one afternoon of tests, and the link to free GPU memory after the wake is a correlation, not a proven cause.
2. **Speculative decoding silently lost after each wake.** Zero draft tokens were accepted out of 1,198 proposed, so generation fell from ~130 to ~62 tokens/s. Nothing failed; the model was simply half as fast. An engine flag that preserves the draft model's weights across sleep fixed it.

7. Trust before traffic

The qualification gate. No engine or model serves until it passes a suite of 12 checks and a golden set:

- chat, streaming and tool calls;
- reasoning on and off, vision and structured output;
- the Anthropic-compatible route;
- three concurrent requests;
- the context window read back from the engine, and a near-window request of 211K tokens;
- cancellation;
- a golden set of known answers.

Each pass is bound to the configuration's fingerprint, and a drift check flags anything serving without a current proof. A check that cannot run counts as not measured, never as a pass. A test that becomes a release gate is the difference between "it worked on my machine" and "it is allowed to serve".

Evaluation against a baseline. The local agents are measured against Claude:

What	Result	Conditions
Research agent	0.82 against 0.79 for Claude Sonnet, on a fixed answer key	3 questions × 3 runs: a small sample, so "matched", not "beat"
Code reviewer, seeded defects	Caught 31-36 of 40	A sealed set, measured on the earlier engines
Code reviewer, real reviews	~70% of its high-severity findings held up on re-check, fully or in part (14 of 20), against ~99% for Sonnet (175 of 176)	One month of real reviews
Long tool chain	290 of 290 steps, 0 errors	Measured on two earlier engines, not re-run on vLLM
Vision	Six golden items pass on vLLM; one pair has never run	So vision stays on the previous engine

The reviewer result changed the design: **it stays an advisory second reviewer and never gates a merge.** A negative result that changes the design is worth more than a positive one that doesn't. The research agent is in daily use: 227 of its 242 runs since 30 October completed.

---

8. Observability and operations

- **Every model call is recorded.** A capture proxy inside each engine image stores the request, the answer, reasoning, tool calls, token counts and timings, using OpenTelemetry GenAI field names. Credentials are never written. Recorded calls can be replayed against another engine for a like-for-like comparison.
- **Metrics and alerts.** Prometheus and Grafana cover the GPU, host RAM, containers and the engine. There are 10 alert rules, including slow prefill and GPU memory spilling into host RAM; a separate dead-man check watches the nightly jobs.
- **Backups with a rehearsed restore.** Five categories of data are backed up on a schedule. A restore is rehearsed monthly: two rehearsals so far, both passed, covering three of the five categories. The other two have never been rehearsed.
- **An engine version rolls back by image tag.** The latest engine upgrade was gated by a measured test run before it served.

---

9. What went wrong

The 21-hour slow-prefill incident (3-4 November 2026).

- **Impact:** for 21 hours, prompt reading ran ~16x slower than normal (about 211 vs 3,344 tokens/s), while generation looked healthy. Agents were slow, but nothing was down. (That "normal" is lower than section 5's figures: an earlier engine version, and a median over all prompt lengths including cached calls.)
- **Detection gap:** no alert watched prefill speed. The capture proxy, which records every call with token counts and timings, made it visible afterwards.

- **Resolution:** an engine replacement ended it. **The root cause is not known**, and I would rather say so than invent one.
- **Prevention:** two LLM-specific alerts, one for slow prefill and one for GPU memory spilling into host memory. The prefill threshold is provisional, since it comes from one episode.

Out-of-memory crashes that were not in VRAM (June 2026).

- **Impact:** every container restarted at once.
- **Cause:** under WSL2, GPU memory overflow spills into host RAM, so "GPU out of memory" was really host RAM running out after two large models were stacked.
- **Fix:** admission checks moved to host RAM; the WSL2 VM is capped; one model sits on the GPU at a time; the page cache is cleared after each model load; heavy jobs are refused under memory pressure.

Data lost during a folder move (May 2026).

- **Impact:** a move skipped folders that were locked by running programs, and the forced delete that followed removed them.
- **Recovery:** three rule files were rebuilt byte-exact from session transcripts.
- **Prevention:** scheduled backups, the monthly restore rehearsal, a written checklist before any destructive command, and a guard on destructive commands.

A stale number of my own. My records said prefill ran at ~4,000 tokens/s. That came from an earlier configuration with smaller prefill chunks. Re-measured from 5,987 live calls, it is ~9,000 for one agent at 10-30K tokens. The lesson: a measurement has a configuration and a date, and a number without them goes stale silently.

Beliefs that measurement overturned.

- "*While one agent reads a long prompt, the others stall.*" Measured on the three-agent test workload: their generation slows by 15% (133.5 to 113 tokens/s). A cold 60K-token read is still untested.
- "*A shared pool lets llama.cpp serve three agents faster.*" Measured: 7.6% slower than serving them one after another.
- "*A sleeping model leaves the whole card free.*" Measured: it still holds about 3 GB.

---

10. How it was built

I am the sole architect and made every decision. Most of the code was written by AI coding agents (Claude Code and the platform's own local agents) working in fixed roles: one plans, one builds, others review and evaluate. The rules they work under are mine:

- **one writer per repository** at a time;
- **an independent review of every change before merge**, with every finding re-measured before it is accepted or rejected;
- **merge checks proven by planting the defect** each one must catch: a check that cannot fail proves nothing;
- **no claim without a measurement** from the same session.

222	independent reviews recorded in one month (15 October to 15 November 2026)
1,352	findings; 260 of them must-fix, and a merge is refused while one is open
94%	of high-severity findings confirmed when re-measured (241 of 257)
424	planted-defect specifications behind the merge checks
123	blocking checks in the local release gate
3,900+	commits since March 2026

The rules catch real errors, including mine. The independent review of this page traced every new figure to its original record and sent ten claims back for correction before publication. A five-round red team of a new internal tool took its confirmed findings from 40 in the first round to 20 in the fifth, each one re-measured before it was closed.

I verify by measurement, which is how most of the numbers above exist.

---

11. Limitations

- **Single node, single GPU:** nothing here proves multi-GPU or multi-node behaviour.
- **The engine comparison is one repetition per cell.** The ~25% gap is large, but it is not a distribution.
- **The capacity table mixes two configurations.** Only the one-agent and seven-agent rows are on today's configuration.
- **The research evaluation is n = 9** (3 questions × 3 runs), and two evaluations were measured on earlier engines only.
- **The KV margin at seven agents is thin** (94-97.5% peak); a heavier mix of long prompts would likely preempt.
- **The incident's root cause is unknown.**
- **No SLOs yet:** the platform has alerts and dashboards, but no formal SLOs or error budgets.
- **Cost per token is not measured.**

---

12. What I would do next

- **SLOs with error budgets:** availability of the gateway, time to first token bucketed by prompt length, and wake success, with burn-rate alerts.
- **Kubernetes:** the same serving stack on Kubernetes with the GPU Operator, to bring the single-node lessons to where production inference runs.
- **Cost:** a measured cost per million tokens, local against API.
- **More than one GPU:** tensor parallelism, or a second model resident instead of sleeping one.

Contact: linkedin.com/in/bstefan93