

Local LLM Inference Engineering

Architecture, benchmarking methodology, and hardware-aware serving analysis

Prepared by	Amine Hamouchi
Role	LLM Inference Engineer
Organisation	AI Crafters
Date	14 August 2026
Classification	Internal · Technical review
Version	1.0
Hardware of record	NVIDIA GeForce GTX 1660 Ti, 6 GB, Turing (SM 7.5)

Self-hosted inference stack · multi-engine evaluation · first-principles performance model · production reporting

This document is written as a professional engineering report, not as an academic internship diary. Implementation artefacts live in the repository; this text captures the engineering argument.

Document control

Field	Content
Title	Local LLM Inference Engineering: architecture, methodology, and serving analysis
Author	Amine Hamouchi, LLM Inference Engineer
Audience	Technical leadership and internship supervisor
Purpose	Record the system as an inference-engineering deliverable: what was built, why the architecture looks this way, how performance is measured, and which deployment decisions the evidence supports.
Scope	Serving stack, engine adapters, benchmark harness, hardware-calibrated simulator, observability, experiment tracking, desktop packaging, and reporting.
Companion artefacts	LLM Engine Benchmark web application, Windows/Linux desktop installer, Grafana dashboard, MLflow tracking store.

Abstract

Selecting a local inference engine is not a model-card decision. Throughput, time-to-first-token, VRAM occupancy, and behaviour under concurrency are properties of the engine × kernel × GPU generation × quantization combination. This report describes a production-oriented platform that makes that combination measurable.

The system has two layers. The first is a self-hosted OpenAI-compatible serving stack: vLLM behind an nginx bearer-token gateway, with Prometheus, Grafana, and MLflow. The second is an evaluation product — FastAPI orchestration, a Next.js console, and a desktop installer — that launches vLLM, Ollama, or llama.cpp in Docker, drives a domain workload through a concurrency sweep, samples GPU state via NVML, and persists every run with its full configuration.

On the hardware of record (GTX 1660 Ti, 6 GB, Turing), a like-for-like 3B-parameter comparison produced a result that folklore would have gotten backwards: Ollama delivered approximately 6× the single-stream decode rate of vLLM (55.1 vs. 8.8 tok/s) at 42% of the VRAM. The cause is kernel availability, not a defect in vLLM. Marlin, vLLM’s fast AWQ path, requires compute capability ≥ 8.0. Turing falls back to a slow dequant loop, saturating the GPU at ~9 tok/s. The same architectural differences imply the ranking flips on Ampere-class 24 GB cards under concurrent load. A first-principles simulator, calibrated to the measured runs, encodes that hardware dependence so projections are not invented from noise.

The engineering contribution is a repeatable inference-evaluation loop: orchestrate, measure serving SLIs, inspect generations, record experiments, and emit a decision-oriented report — with an explicit distinction between measured GPU data and simulated estimates.

1. Engineering mandate

The deployment question this work is built to answer is operational, not academic:

Which engine, model, and quantization should we serve, on this hardware, for this workload — and what evidence supports that choice?

Public blog benchmarks do not answer it. They almost never hold engine, model, quantization, prompt mix, and GPU generation constant at once. They rarely report time-to-first-token under load, aggregate tokens per second, or VRAM headroom. They never capture the actual generation for the tasks the system will run in production.

Three constraints defined the mandate.

Hardware is a first-class input

The development GPU is a GTX 1660 Ti: 6 GB, Turing, no bfloat16, no Marlin. That is a hostile environment for several “default” serving stacks, and therefore a useful one. An evaluation platform that only looks correct on an H100 is not an evaluation platform; it is a demonstration. The system had to tell the truth on constrained silicon and remain correct when the card is upgraded.

The workload is the product

The target domain is banking operations: transaction categorisation, fraud triage, KYC extraction, DTI arithmetic, dispute replies, statement Q&A, interest calculation, and compliance notes. Latency and throughput numbers are only meaningful if the prompt mix resembles production. Generic chat prompts were rejected as the default.

Measurement must survive contact with operations

A one-off notebook is not a serving decision. The platform needed engine lifecycle (pull, launch, health, teardown), live GPU sampling, experiment tracking, authentication, a shareable report, and a desktop path for real GPU runs on Windows — the environment where the GPU actually lives.

The resulting system is therefore not “a vLLM docker-compose file with a UI on top”. It is an inference-engineering workstation: a serving stack for day-to-day use, and a benchmark product for deciding what that stack should run.

2. System architecture

2.1 Two complementary surfaces

The repository exposes two operational surfaces that share the same mental model — OpenAI-compatible chat completions — but serve different jobs.

1. Serving stack (docker-compose.yml). A pinned vLLM server behind nginx, with Prometheus scraping /metrics, a provisioned Grafana dashboard, and MLflow. This is the always-on inference endpoint.
2. Evaluation platform (webapp/). A Next.js console and FastAPI control plane that can launch any of three engines as an ephemeral Docker container, sweep concurrency, and write a structured experiment to MLflow. A desktop Electron shell bundles this path for Windows and Linux so real GPU runs do not require a manual Python/Node toolchain.

Keeping those surfaces separate is deliberate. Serving wants stability, a fixed model, and an auth gateway. Evaluation wants isolation, teardown after every run, and the freedom to swap engines. Conflating them produces either an unsafe production box or a benchmark that cannot be reproduced.

2.2 Serving topology

Clients never address vLLM directly. Traffic hits nginx on port 8000, which enforces Authorization: Bearer <VLLM_API_KEY> before proxying to the engine. Prometheus scrapes vLLM's native /metrics endpoint; Grafana loads a provisioned "vLLM Inference Monitoring" dashboard; MLflow provides experiment tracking.

Data path

```
Client (chat.py / HTTP) → nginx :8000 (bearer gateway) → vLLM (OpenAI API)
                                                                ├── Prometheus :9090
                                                                ├── Grafana :3000
                                                                └── MLflow :5000
```

Configuration is env-driven. Model id, GPU memory utilisation, max model length, image tag, and extra vLLM flags live in .env. The compose file is not edited to swap a model. That is a production habit, not a convenience: an inference box that requires a YAML edit to change weights will drift.

2.3 Evaluation topology

Control plane

```
Next.js UI :3947 → FastAPI :8947 → Docker SDK (GPU passthrough) → Engine
container :8100 /v1
                        ├── NVML sampler (1 Hz GPU / memory)
                        └── MLflow store (history and artefacts)
```

The adapter contract is the architectural hinge. Each engine implements image, command, port, volumes, health URL, and optional post-launch hooks (Ollama must pull a model after the daemon is up). The runner then speaks one protocol: streaming /v1/chat/completions with stream_options.include_usage. That single decision — normalise engines at the OpenAI boundary — is what makes a fair concurrency sweep possible.

2.4 Lifecycle and failure handling

Engine containers are named llm-bench-<engine>-<epoch> and torn down after the sweep. Before launch, stale containers with that prefix are force-removed so a crashed Docker Desktop session cannot hold port 8100 indefinitely. Image pulls report layer progress to the UI; a multi-gigabyte first pull must not look like a hung bar at 2%.

Readiness is not "container running". The adapter polls the engine health endpoint until it returns a non-5xx, with a separate path for a confirmed exited status. Transient Docker daemon timeouts — common while a large GGUF is loading — increment a failure streak rather than aborting the run. Only a sustained unreachable daemon, or a container that has actually exited, fails the experiment. Exit logs are reduced to the last error-bearing line so the operator sees the cause, not a 60-line CUDA dump.

This is serving-operations work. A benchmark that cannot start, recover, or explain a failed engine is not usable by anyone except its author.

3. Inference engines and kernel reality

3.1 The three engines, as serving systems

The platform treats vLLM, Ollama, and llama.cpp as three different answers to the same job: turn weights into a streaming token API.

	vLLM	Ollama	llama.cpp
Unit of serving	Continuous batch, paged KV	Default single slot	Default single slot
Weight format	HF / AWQ / GPTQ	GGUF via tags	Explicit GGUF file
Fast quant path	Marlin / Machete, SM $\geq$ 8.0	llama.cpp CUDA kernels	llama.cpp CUDA kernels
VRAM policy	Reserves gpu-memory-utilization up front	Grows with load	Grows with load
Image	vllm/vllm-openai:v0.7.3	ollama/ollama	llama.cpp:server-cuda
Role in the matrix	Throughput under concurrency on modern NVIDIA	Operational simplicity on mixed hardware	Deterministic GGUF, maximum kernel control

vLLM is configured with `ipc=host`, `--dtype float16`, `--swap-space 0`, and a Hugging Face cache volume. Float16 is forced because Turing has no bfloat16; swap space is disabled because paging weights to host RAM on a 6 GB card produces a result that looks like a hang. The image is pinned to v0.7.3: newer builds were observed to crash on import under WSL2, and the V0 engine in this line is the more reliable path on that host.

llama.cpp models are addressed as `repo::file.gguf`. The combined `-hf repo:tag` form treats the suffix as a quant tag, not a filename; passing a GGUF filename there fails resolution. Validation happens before the container starts so the operator gets a precise error instead of a dead container.

Ollama starts empty. After `/api/tags` is healthy, the adapter streams `/api/pull` and maps download percentage onto the progress bar, then proceeds to GPU load. First-run UX is part of inference engineering; a five-minute silent pull is a product defect.

3.2 Why Turing changes the ranking

Decode is memory-bandwidth bound: each generated token requires reading the entire weight tensor once. The practical ceiling is:

$$\text{tok/s} = (B_{\text{mem}} \times \text{MBU}) / (\text{bytes_per_param} \times N + \text{KV traffic})$$

where B_{mem} is device memory bandwidth, MBU is achieved memory-bandwidth utilisation, N is parameter count, and KV traffic is cache read volume at the current context.

Prefill (prompt processing) is compute-bound:

$$\text{TTFT} \approx (2 \times N \times T_{\text{prompt}}) / (\eta \times \text{FP16 FLOPs}) + T_{\text{overhead}}$$

MBU is not a property of the GPU alone. It is a property of the kernel. vLLM's fast AWQ path (Marlin) is compiled for Ampere and newer. On Turing it is absent. The engine still runs; it dequantises along a slow path, the SM fills up, and decode collapses to single-digit tok/s. Ollama and llama.cpp use GGUF CUDA kernels that are tuned for this generation, so they sit on the memory-bandwidth roofline instead of the dequant ALU roofline.

That is the entire Turing story. It is also why a simulator that ignores compute capability is a toy. The performance model encodes SM 8.0 as a hard gate on vLLM's MBU.

3.3 Continuous batching versus a single slot

vLLM reads weights once per batch and amortises that cost across concurrent requests. Aggregate throughput can therefore rise with concurrency until compute saturates. Ollama and llama.cpp, in the default serving configuration used here, process one request at a time. Aggregate throughput stays close to the single-stream rate; additional concurrency becomes queueing delay. Both behaviours are correct. They are different products. The benchmark's job is to make the difference visible on a throughput-versus-concurrency chart rather than hidden inside a single "tokens/sec" number.

4. Benchmarking methodology

4.1 What is measured

Every completed run records the serving SLIs that decide whether an endpoint can be deployed.

Metric	Definition	Why it is a serving metric
TTFT p50 / p95	Wall time to the first SSE data: chunk on a streaming completion	Interactive latency. Users feel this, not decode speed.
Tokens / sec	Output tokens divided by generation time after the first token; peak and per-level mean	Single-user generation speed.
Aggregate throughput	Sum of output tokens across in-flight requests, divided by level wall-clock	Capacity of the server, not of one client.
Max concurrency	Highest sweep level with $\geq 90\%$ successful requests	Crude capacity number; SLO-based capacity is the intended successor.
Degradation	Drop in per-request tok/s from the single-stream best to peak concurrency	Stability under load. Retention = $100\% - \text{degradation}$.
GPU memory / util.	NVML samples at 1 Hz, plus Docker container memory	Fit, headroom, and whether the engine is compute-bound or waiting.
Success / error	HTTP failures, empty generations, timeouts	Reliability is a first-class result, not a footnote.

TTFT is taken from the client side of the stream, at the first server-sent event, using `time.perf_counter()`. Token counts prefer the engine's usage field when `include_usage` is honoured, and fall back to

incremental chunk counting. Percentiles use linear interpolation; there is no NumPy dependency in the hot path.

4.2 Workload

The default prompt set, `banking_core`, is eight tasks written as a banking operations assistant: structured JSON where the task is extraction or categorisation, short prose where the task is explanation or customer reply, and explicit arithmetic where the task is DTI or interest. A shared system prompt forbids invented account data.

This is not a quality benchmark in the HELM sense. It is a serving benchmark with a quality spot-check. Every prompt's generation is captured so a fast engine cannot be declared the winner if it emits unusable text. On the 3B Qwen2.5-Instruct runs, output quality was comparable across engines; the difference was speed and footprint. That statement is only licensed because the generations were stored.

Prompt sets are editable. A domain that is not banking should not be evaluated on banking prompts. The harness is the product; the prompt file is configuration.

4.3 Concurrency sweep

A run is a sequence of levels (default 1, 2, 4, 8), each issuing a fixed number of requests against the live endpoint with a bounded number of in-flight calls equal to the level. Levels are serial so GPU samples can be attributed. A "Quick (1, 4)" preset exists for perf-only iteration: fewer requests, sample-output capture off, so a kernel or flag change can be checked in minutes rather than a full campaign.

4.4 Real versus simulated

A run is real when Docker and an NVIDIA device are present: the engine is launched, the API is driven, NVML is sampled. If either is missing, the platform falls back to simulation rather than crashing the UI. That fallback is a product requirement (demos, CI, machines without a GPU). It is also a scientific hazard. Simulated numbers are estimates. The report generator is required to brand them as such; they must never be visually interchangeable with measured GPU data.

4.5 Fairness

A comparison is like-for-like only when engine is the experimental factor and model, quantization, hardware, dtype, prompt set, context length, and sweep are held. The 3B vLLM-versus-Ollama comparison in Section 5 meets that bar. The llama.cpp 1.5B data point does not, and is reported separately. The reporting layer flags partial comparisons rather than silently ranking unlike configurations — a requirement of any evaluation product that will be shown to a decision-maker.

5. First-principles performance model

Simulation in this system is not a random number generator with a GPU skin. It is a reduced-order model of transformer inference, calibrated against the same hardware the real runner measured.

5.1 Weight bytes, KV cache, and fit

Effective bytes per parameter include scale/zero overhead. Q4_K_M is treated as ≈ 4.5 bits/weight (0.58 B/param), not 4. AWQ/GPTQ sit near 0.55 B/param. Weights, KV, and a small activation/CUDA overhead are summed and compared to usable device memory (published size $\times$ a vendor-specific usable fraction; 0.90 on the 1660 Ti to leave room for the driver and display).

KV bytes per token are estimated as $22,000 \times \text{vN_billions}$ in FP16, which reproduces published Qwen2.5 1.5B/3B/7B figures to within a few percent without shipping a full architecture table. If the working set exceeds usable memory, the overflow fraction is served across PCIe at ≈ 16 GB/s. The harmonic combination of on-device and off-device time dominates the result: a model that “almost fits” is often an order of magnitude slower, and the model says so explicitly.

5.2 Calibrated MBU

MBU constants are not taken from marketing decks. They are inverted from measured single-stream decode on the 1660 Ti ($B_{\text{mem}} = 288$ GB/s):

Engine	Model / quant	Measured tok/s	Implied MBU	Simulator tok/s
llama.cpp	Qwen2.5-1.5B Q4_K_M	127.3	0.39	118.3 (−7%)
Ollama	Qwen2.5-3B Q4_K_M	55.1	0.34	60.2 (+9%)
vLLM	Qwen2.5-3B AWQ	8.8	0.05	8.3 (−5%)

The vLLM MBU of 0.05 is the Turing fallback. On $SM \geq 8.0$ the same engine is assigned MBU 0.62, reflecting Marlin/Machete plus PagedAttention. Prefill efficiency follows the same split: $\eta \approx 0.07$ on pre-Ampere vLLM (matching a measured 778 ms TTFT on a ~ 93 -token prompt), $\eta \approx 0.45$ on Ampere+ vLLM, $\eta \approx 0.17$ for the GGUF CUDA path.

Hardware presets cover NVIDIA discrete GPUs from GTX 1660 Ti through H100, Apple Silicon M1–M4 Max, and CPU/DDR, using published bandwidth and FP16 TFLOPS. Custom profiles accept memory, bandwidth, TFLOPS, and compute capability directly. vLLM is refused on Metal and CPU: it has no such backend, and inventing one would be a lie.

5.3 Known limits, stated

TTFT models a cold prefill. Engines that cache a shared system-prompt prefix (llama.cpp, Ollama) will beat the estimate after the first request, sometimes by an order of magnitude. The simulator reports the assumptions it used. A reader can reject them. That is the point of a model, as opposed to a fitted black box.

6. Experimental results

6.1 Setup of record

- **GPU.** NVIDIA GeForce GTX 1660 Ti, 6 GB, Turing, compute 7.5.

- **Runtime.** Docker on WSL2, NVIDIA Container Toolkit, dtype=float16.
- **Workload.** banking_core, 8 prompts.
- **Head-to-head.** Qwen2.5-3B-Instruct, concurrency {1, 2, 4, 8}, 8 requests per level, 256 max tokens.
- **Formats.** vLLM AWQ 4-bit; Ollama Q4_K_M.
- **Supplementary.** llama.cpp on Qwen2.5-1.5B Q4_K_M, single-stream only — not in the ranking.

6.2 Head-to-head, identical 3B model

Metric	vLLM AWQ	Ollama Q4_K_M	Reading
Single-stream tok/s	8.8	55.1	6.3×
TTFT p50 at c=1	778 ms	343 ms	first token
TTFT p50 at c=8	8.8 s	10.4 s	both queue
Peak aggregate tok/s	25.7	50.8	server output
GPU memory peak	5,792 MB	2,440 MB	vLLM reserves ~90%
GPU utilisation peak	100%	63%	vLLM compute-bound
Degradation under load	9.8%	14.6%	per-request tok/s
Max concurrency (≥ 90% ok)	8	8	card, not engine

Ollama is the correct default on this card for interactive and lightly concurrent serving: faster, lighter, and leaving VRAM for context. vLLM is not “slow” in general. It is compute-bound on a kernel path that this GPU does not have. The 100% utilisation at 8.8 tok/s is the diagnostic: the SM is busy dequantising, not waiting on GDDR.

Aggregate throughput still reveals engine strategy. vLLM climbs from ~9 to 25.7 tok/s as concurrency goes 1→8: continuous batching packing a queue. Ollama starts near 55 tok/s and plateaus: one slot, requests wait. At c=8, TTFT on both engines is seconds, not milliseconds. Six gigabytes cannot hold many concurrent KV caches. That is a capacity limit of the device.

6.3 Supplementary: llama.cpp 1.5B

Single-stream on Qwen2.5-1.5B Q4_K_M: 127 tok/s peak, 39 ms TTFT, 1,515 MB VRAM. This is the leanest measured path on the card. It is not a 3B result and is not ranked against the table above.

6.4 What Ampere-class hardware is expected to change

The model assigns vLLM MBU 0.62 once Marlin is available. Combined with PagedAttention and continuous batching, the ranking is expected to reverse under multi-user load on a 24 GB Ampere+ GPU (RTX 3090/4090, A10, L4). That statement is a projection, labelled as such. The purpose of the platform is that the projection can be confirmed in one run on the new card, with the same prompts and sweep, rather than re-litigated in a meeting.

7. From measurements to a deployment decision

Raw SLIs are necessary and insufficient. A decision-maker needs a recommendation, a validity statement, and a workload-shaped score — not a dashboard dump.

The reporting pipeline already computes:

- **Workload profiles with explicit weights:** interactive (TTFT and stability), API serving (aggregate and concurrency), batch (throughput), cost (efficiency).
- **GPU efficiency:** tok/s per GB VRAM and tok/s per utilisation point, when NVML samples exist.
- **SLO-shaped capacity** from per-level TTFT, latency, and error rate, rather than “max concurrency = 8” alone.
- **Fairness / confidence:** like-for-like versus partial; real versus simulated; sample-size caution.
- **Regression view:** same engine/model/quant/hardware across dates, with signed deltas.

The recommended reading order of an exported report is: recommendation and KPIs, validity banner, throughput and TTFT versus concurrency, comparison matrix, then run detail. Methodology and raw logs belong in the appendix. That hierarchy is how inference evaluation is presented in a serious organisation: decision first, evidence second, reproduction third.

On the hardware of record, the decision is unambiguous:

- **Interactive / single-user, Turing 6 GB.** Ollama or llama.cpp, Q4_K_M, smallest model that meets quality.
- **Multi-user serving, Ampere+ 24 GB.** vLLM, AWQ, with a measured concurrency sweep before committing an SLO.
- **Never:** quote simulated GPU memory as a physical measurement; never rank a 1.5B GGUF against a 3B AWQ without a fairness warning.

8. Productionisation

Inference engineering does not stop at a good kernel. The surrounding system is what makes the measurement repeatable by someone who did not write it.

8.1 Authentication and tenancy

The serving gateway checks Authorization: Bearer <VLLM_API_KEY> at nginx. The evaluation app uses Clerk when BENCH_CLERK_JWKS_URL is set: the Next.js session token is sent as a bearer to FastAPI, which verifies against JWKS. History and generated reports are scoped to the signed-in user. Local development can disable auth; production must not. Direct access to the FastAPI port is protected under the same policy.

8.2 Experiment tracking

Every run is an MLflow experiment: config, metrics, GPU timeline, prompt outputs, engine logs. The default tracking URI can be a hosted store (mlflow.comptek.store) so desktop and web clients share history without bundling a tracking server into the installer. Reproducibility is a stored artefact, not a screenshot.

8.3 Observability of the serving stack

Prometheus scrapes vLLM every 5 s. Grafana loads a provisioned dashboard. This is the live counterpart of the benchmark: once a configuration is chosen, the same class of metrics (throughput, latency, cache, GPU) is watched in production rather than only in an experiment.

8.4 Desktop distribution

Real GPU runs on Windows require the NVIDIA stack and Docker Desktop on the host, not in a remote Linux CI runner. The Electron application bundles Python, FastAPI, and a standalone Next.js UI into a single installer (~120–180 MB after slimming). Docker and GPU drivers remain a host dependency — they cannot be honestly hidden. Clerk is disabled locally. Orchestration is on. This is the path that produces measured numbers rather than simulated ones.

Linux is shipped as Appliance and .deb, built on Ubuntu; it is not cross-compiled from Windows because the bundle contains native binaries.

8.5 Deployment of the web console

The control plane splits cleanly: FastAPI on Railway (Docker), Next.js on Vercel, MLflow already hosted. That split matches how the process boundaries were drawn. Simulation mode keeps the cloud UI demonstrable without attaching a GPU to a serverless function — another reason simulated versus real must stay visually distinct.

9. Engineering decisions worth recording

- **Normalise at the OpenAI API, not inside each engine.** The alternative is three runners and three metric definitions. One streaming client, three adapters, one result schema.
- **Pin the vLLM image and force float16.** Floating to :latest on WSL2 was a reliability regression. Turing has no bf16; leaving dtype to the engine's default is a silent footgun.
- **Refuse impossible combinations in validation, not at runtime.** llama.cpp without a ::file.gguf, vLLM on Metal, empty model ids — all fail before containers.run. Operator time is part of system quality.
- **Calibrate the simulator to this project's measurements.** A generic "7B Q4 is ~40 tok/s" constant would have hidden the Turing/Marlin cliff. Inverting MBU from real runs made the cliff a parameter.
- **Treat first-run downloads as a progress state.** Multi-GB image and weight pulls are the common case the first time an engine is selected. Silent blocking is indistinguishable from a hang.
- **Do not mix serving and evaluation process models.** A long-lived vLLM behind nginx is the production shape. Ephemeral llm-bench-* containers are the experiment shape.
- **Ship the desktop installer for the GPU that exists.** The card is on a Windows workstation. Pretending that "just run it on Linux" is the only professional path would have made real measurements optional.

10. Recommendations

3. On the current 6 GB Turing workstation, serve with Ollama or llama.cpp. vLLM remains in the matrix for when the GPU generation changes; it is not the daily driver here.

4. Re-run the identical 3B sweep on the first Ampere+ 24 GB card acquired. Treat that run as the go-live evidence for vLLM, not this report's projection.
5. Promote SLO-based capacity (TTFT p95, latency p95, error rate) as the headline capacity number in operator-facing reports, with max concurrency retained as a supporting figure.
6. Keep the fairness and simulation banners non-optional. A polished PDF that hides unlike models or simulated GPU data is worse than an ugly one that does not.
7. Use `banking_core` as the default, not the only, prompt set. When the served product changes, the evaluation set must change with it — in the same harness.

11. Roadmap

The measurement core is in place. The next engineering increments are decision-quality, not additional charts.

- **Capacity at SLO as a first-class metric**, with configurable thresholds and a vertical marker on the concurrency charts.
- **Transparent decision matrix** per workload profile, with weights visible, never a mysterious 94/100.
- **Energy and cost layer** (GPU watts, Wh/1k tokens, tokens per GB) once NVML power samples are wired — connecting serving performance to infrastructure spend.
- **Automatic regression gates** on matching engine/model/quant/hardware tuples, so a flag change that costs 12% throughput is visible the same day.

None of these require a new benchmark philosophy. They consume the per-level traces the runner already stores.

12. Conclusion

This work delivers an inference-engineering system: a guarded local serving stack, a multi-engine evaluation product, a hardware-calibrated performance model, and a reporting path that turns serving SLIs into a deployment recommendation.

The headline empirical result is uncomfortable for anyone who treats “vLLM” as a synonym for “fast”: on Turing 6 GB, Ollama was ~6× faster on the same 3B model, at well under half the VRAM. The explanation is kernel availability. The implication is that engine choice is a function of GPU generation and concurrency shape, and must be measured on the hardware that will actually serve.

That is the standard this platform enforces. Folklore is cheap. A concurrency sweep against a domain workload, with GPU traces and stored generations, is the job.

Appendix A. Metric and configuration notes

Serving stack knobs

Variable	Role
HF_TOKEN	Hugging Face access for gated weights and rate limits
VLLM_API_KEY	nginx bearer token on :8000; default must be changed off-localhost
VLLM_IMAGE_TAG	Pinned vLLM image (default v0.7.3)
VLLM_MODEL	Served model id
VLLM_GPU_MEM_UTIL	Fraction of VRAM reserved by vLLM
VLLM_MAX_MODEL_LEN	Context window
VLLM_EXTRA_ARGS	Escape hatch for additional engine flags

Evaluation ports

UI :3947, FastAPI :8947, ephemeral engine :8100, Grafana :3000, Prometheus :9090, serving nginx :8000, MLflow :5000 (local) or hosted.

Success criterion for a sweep level

A level counts toward max concurrency when at least 90% of its requests complete without HTTP error or runner-side timeout. This is a reliability floor, not an SLO.

Degradation

Let r_1 be best single-stream tok/s and r_c the mean per-request tok/s at the highest successful level. Degradation = $(r_1 - r_c) / r_1$. Retention = $1 - \text{degradation}$.

Appendix B. Selected references

- **Kwon et al.**, Efficient Memory Management for Large Language Model Serving with PagedAttention (vLLM), SOSP 2023.
- **Lin et al.**, AWQ: Activation-aware Weight Quantization.
- **Frantar & Alistarh**, GPTQ; Marlin kernel path in vLLM for Ampere+.
- **Gerganov et al.**, llama.cpp / GGUF.
- **NVIDIA**, Turing (SM 7.5) vs. Ampere (SM 8.x) programming guides — bf16 and tensor-core availability.

End of report. Source: Local LLM Setup repository. Author: Amine Hamouchi, LLM Inference Engineer, AI Crafters. 14 August 2026.