



LLM Load Testing: Open vs Closed Loop and Goodput

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-29 · genai-perf · llm load testing methodology · vllm bench serve request rate · guidellm benchmark · aiperf benchmark llm · open loop vs closed loop load testing llm · llm benchmark poisson arrival

Original article: <https://gpusmith.com/articles/en/llm-load-testing-methodology-open-vs-closed-loop>



In this article

1. Why most LLM benchmarks mislead
 2. Closed loop vs open loop
 3. Little's law: the arithmetic that links the two
 4. Coordinated omission, with numbers
 5. Arrival processes: constant, Poisson, gamma
 6. Concurrency caps and what they do
 7. Input and output length distributions
 8. Warm-up, duration and steady state
 9. Goodput and SLO definition
 10. Sweeping to find the knee
 11. Tool comparison
 12. Reporting template and pitfalls
-

Why most LLM benchmarks mislead

The most common LLM serving benchmark is a concurrency sweep: N simulated clients each send a request, wait for the full response, and send the next one, with N stepped through 1, 8, 32, 128. It answers a narrow question: how much throughput the server delivers when the load generator politely waits for it. It does not answer what a capacity plan or SLO sign-off asks, which is what happens to time to first token (TTFT) and per-token latency when users arrive at a given rate regardless of how busy the server is. The gap between the two answers is widest near saturation, exactly where it matters.

Three design errors account for most misleading results. The arrival model: a fixed-concurrency generator slows down whenever the server slows down, so it never reproduces the queueing collapse real traffic causes. Measurement bias known as coordinated omission: the generator stops sending during a stall and records almost no slow samples. Workload shape: one fixed prompt length and one fixed output length, which removes the variance that drives batching and KV-cache pressure. This guide treats the benchmark as experiment design first, then maps each choice onto `vllm bench serve`, `GuideLLM` and `AIPerf`, as documented on 2026-10-28 (`vLLM` "latest" docs and v0.32.0 source, `GuideLLM` v0.8.x, `AIPerf` v0.13.0). These tools change monthly; flag names should be rechecked against `--help` before a run.

Closed loop vs open loop

A **closed-loop** generator holds a fixed population of N clients, each issuing a request only after its previous response completes (optionally after a "think time"). In-flight requests never exceed N , and the arrival rate is an output of the experiment, not an input. An **open-loop** generator issues requests on a schedule independent of completions, so the number in flight is unbounded and is itself a measured outcome.

Schroeder, Wierman and Harchol-Balter made this distinction the subject of an NSDI 2006 paper ([Schroeder et al., NSDI 2006](#)). Their case studies were web servers, databases and an auction site, not LLM servers, but the principles transfer. Principle (i): "For a given load, mean response times are significantly lower in closed

systems than in open systems." Principle (ii): as the multiprogramming level (MPL, the client count N) grows, closed systems approach open behaviour, "but convergence is slow for practical purposes." Principle (iii): variability has a large effect in open systems and a much smaller one in closed systems. They also describe a partly-open model, in which users arrive openly and then issue a session of requests in closed fashion, and give a rule of thumb that it behaves like an open system when sessions average 5 or fewer requests and like a closed system at 10 or more ([Schroeder et al.](#)).

The two models answer different questions, and both are legitimate:

#	Question	Model	Typical tool setting
1	Peak throughput the engine can sustain with a bounded batch	Closed loop, high N	AIPerf --concurrency, GuideLLM throughput / concurrent
2	Single-request latency floor	Closed loop, $N = 1$	GuideLLM synchronous
3	Latency at a forecast request rate (capacity plan, SLO)	Open loop	--request-rate with Poisson or gamma arrivals
4	Behaviour behind a gateway that caps in-flight requests	Open loop plus concurrency cap	--request-rate + --max-concurrency / -concurrency
5	Agentic or chat sessions with multi-turn reuse	Partly open or trace replay	AIPerf --fixed-schedule, GuideLLM trace replay

A public chat endpoint or multi-tenant API is closer to row 3; a batch pipeline with a fixed worker pool is genuinely closed (row 1), which is why closed-loop numbers fit offline jobs (see the [batch vs real-time sizing analysis](#)).

Little's law: the arithmetic that links the two

Little's law states that the average number of items in a queueing system, L , equals the average arrival rate, λ , times the average time an item spends in the system, W : **$L = \lambda W$** ([Little, Operations Research 2011](#)). It holds regardless of arrival distribution, which makes it the conversion factor between concurrency and rate.

Worked example 1, sizing the concurrency a forecast implies (illustrative). Assume a forecast peak of $\lambda = 6$ requests/s and a mean end-to-end latency of $W = 15$ s (for example about 0.5 s TTFT plus 500 output tokens at about 29 ms per token). Then $L = 6 \times 15 = 90$ requests in flight on average. A closed-loop test that stops at $N = 32$ never observes the operating point the forecast implies.

Worked example 2, how closed loop self-throttles (illustrative). A closed-loop test with $N = 64$ and zero think time measures mean latency $W = 20$ s. By Little's law its throughput is $X = N / W = 64 / 20 = 3.2$ requests/s. Suppose a configuration change makes the server slower and W rises to 32 s. The generator now offers $X = 64 / 32 = 2.0$ requests/s, a 37.5% load reduction it applied to itself. Real users would not have slowed down; production would have seen a queue.

Worked example 3, open-loop overload (illustrative). An open-loop test offers $\lambda = 4$ requests/s to a deployment whose sustainable completion rate is $\mu = 3.5$ requests/s. The backlog grows by $4 - 3.5 = 0.5$ requests every second. After a 600 s run, about $0.5 \times 600 = 300$ requests are queued beyond what the engine

can work on, and a request arriving at the end waits on the order of $300 / 3.5 \approx 86$ s before its share of service begins. The run has no steady state, which gives a practical diagnostic: if p99 TTFT grows when the same rate runs twice as long, the offered rate exceeds capacity and the number is not a latency measurement.

Coordinated omission, with numbers

Gil Tene's wrk2 README describes the effect precisely: in a wait-for-response design, "high latency responses result in the load generator coordinating with the server to avoid measurement during high latency periods." wrk2's remedy is a constant-throughput schedule that "measures response latency from the time the transmission should have occurred" ([wrk2 README](#)).

Worked example 4 (illustrative, simplified). One closed-loop client, normal latency 0.1 s, 200 s run, a single 50 s server stall starting at $t = 100$ s.

- Closed loop: about 1,000 samples at 0.1 s before the stall, one sample at about 50 s, and about 500 samples at 0.1 s after. Of about 1,501 samples, one is slow. The p99 is 0.1 s and the mean is $(100 + 50 + 50) / 1,501 \approx 0.13$ s. Only the maximum reveals the stall.
- Open loop at an intended 10 requests/s: 2,000 requests are scheduled, 500 of them during the stall. Measured from their intended send time, those 500 see latencies spread roughly uniformly from about 50 s down to 0.1 s (assuming, as a simplification, that the backlog drains immediately when the stall ends). Now 25% of samples are slow, the p99 (the 20 slowest samples) is about 48 s, and the mean is $(1,500 \times 0.1 + 500 \times 25) / 2,000 \approx 6.3$ s.

Same server, same stall: p99 of 0.1 s versus about 48 s. LLM engines rarely stall for 50 s, but KV-cache preemption and long prefills produce the same pattern at smaller scale. Since vLLM v0.30.0 (2026-09-22), when `--max-concurrency` is set the client records `client_queue_time = start_time - request_arrival_time`, and with a finite request rate it also reports `e2e_including_client_queue`, the latency measured from the scheduled arrival rather than from the actual send ([vLLM serve.py, v0.32.0](#)). That metric is the coordinated-omission-corrected view and should be requested explicitly via `--percentile-metrics` ([vLLM bench serve CLI](#)).

Arrival processes: constant, Poisson, gamma

A **constant** process spaces requests exactly $1/\lambda$ apart. A **Poisson** process draws exponential inter-arrival times with mean $1/\lambda$, giving a coefficient of variation (CV, standard deviation over mean) of 1; it is the standard model for many independent users. A **gamma** process generalises both. vLLM's implementation samples each interval as `np.random.gamma(shape=burstiness, scale=theta)` with `theta = 1.0 / (current_request_rate * burstiness)` ([vLLM serve.py, v0.32.0](#)). With shape k and scale $1/(\lambda k)$, the mean interval is $k \times 1/(\lambda k) = 1/\lambda$ for every k , the variance is $k \times (1/(\lambda k))^2 = 1/(\lambda^2 k)$, and the CV is $1/\sqrt{k}$.

#	Burstiness k	Process	Mean interval at $\lambda = 4$ req/s	Std. dev.	CV
1	0.25	Bursty gamma	250 ms	500 ms	2.0
2	1.0	Poisson (exponential)	250 ms	250 ms	1.0

#	Burstiness k	Process	Mean interval at $\lambda = 4$ req/s	Std. dev.	CV
3	4.0	Smooth gamma	250 ms	125 ms	0.5
4	∞	Constant	250 ms	0	0

Derived from the gamma formulas above; the rate is illustrative.

All four offer the same average load; principle (iii) predicts that row 1 will show worse tail latency than row 4. vLLM documents `--burstiness` as defaulting to 1.0, which "follows Poisson process", with values below 1 more bursty and above 1 more uniform ([vLLM bench serve CLI](#)). AIPerf exposes `--arrival-pattern` with constant, poisson (the default) and gamma, plus `--arrival-smoothness`, which its reference describes as "Compatible with vLLM's `--burstiness` parameter (same value = same distribution)" ([AIPerf CLI options](#)). GuideLLM offers constant and poisson profiles with the rate set inside the profile spec, for example `--profile kind=constant,rate=10` ([GuideLLM README](#)); v0.8.1 (2026-10-08) has no gamma strategy.

One default deserves attention: vLLM's `--request-rate` defaults to `inf`, and "If this is `inf`, then all the requests are sent at time 0" ([vLLM bench serve CLI](#)). Combined with the default `--num-prompts` of 1000, an unconfigured run is a single 1,000-request burst. That measures drain throughput, not latency at any meaningful offered load.

Concurrency caps and what they do

Adding a concurrency cap to a rate-driven test produces a hybrid: arrivals are scheduled openly, but a client-side semaphore admits only K at a time. This models a gateway or a fixed connection pool, and it is valid when production has such a component. The cost is that the cap reintroduces closed-loop throttling at the client. vLLM's help text states that with both flags set, "the actual request rate may be lower than specified ... if the server is not processing requests fast enough to keep up"; `--max-concurrency` has no default, so concurrency is otherwise unlimited ([vLLM bench serve CLI](#); [vLLM benchmarking CLI guide](#)). In AIPerf, `--concurrency` alone is closed loop ("AIPerf issues a new request immediately when one completes"), while with `--request-rate` it "acts as a ceiling; requests scheduled by rate are blocked if at limit" ([AIPerf load generator reference](#); [AIPerf CLI options](#)). The AIPerf README calls this combination "Dual request control" ([AIPerf README](#)).

Whenever a cap is active, report achieved rate next to target rate, and latency including client-side queue time: if achieved rate falls below target, the cap set the result, and the client-side wait is user-visible latency that server-side TTFT omits. AIPerf v0.13.0 latencies exclude that wait: it issues a request only after a concurrency slot is free and times latency from the actual send (`credit/issuer.py`), so ceiling wait is invisible.

Input and output length distributions

Prefill cost scales with input sequence length (ISL), decode cost with output sequence length (OSL), and KV-cache occupancy with both, so fixed lengths remove the variance that makes batching hard. Out of the box, vLLM and AIPerf produce fixed lengths. vLLM's random dataset defaults to `--random-input-len 1024`, `--random-output-len 128` and `--random-range-ratio 0.0`, which samples lengths from $[\text{input_len} * (1 -$

`range_ratio)`, `input_len * (1 + range_ratio)]`, a zero-width range at the default ([vLLM bench serve CLI](#)). AIPerf synthesises ISL from a normal distribution with `--isl` defaulting to 550 and `--isl-stddev` defaulting to 0.0, and only caps output via `max_completion_tokens` when `--osl` is set ([AIPerf CLI options](#)).

Three levels of realism are available:

#	Workload source	Strength	Weakness
1	Synthetic, fixed or normal ISL/OSL	Reproducible, isolates prefill vs decode	No real stopping behaviour, no prefix reuse
2	Public conversation datasets (vLLM sharegpt, burstgpt, hf)	Realistic length spread	Not the operator's traffic; prompt mix may not match
3	Production trace replay (AIPerf mooncake_trace, bailian_trace, baseten_trace; GuideLLM WEKA/OTEL replay; vLLM timed_trace)	Real arrivals, lengths and prefix sharing	Requires capture and sanitisation

Dataset names are from the [vLLM bench serve CLI](#), [AIPerf CLI options](#) and [GuideLLM v0.8.0 release notes](#). Trace datasets with timestamps auto-promote AIPerf to `--fixed-schedule` mode, which "replays the exact timing pattern from the dataset"; `--no-fixed-schedule` keeps the user's chosen timing mode instead ([AIPerf CLI options](#)).

Prefix caching is the most frequent hidden variable: random prompts share nothing and understate a cache-enabled engine, while a shared long system prompt overstates it. vLLM's `--random-prefix-len` adds a fixed shared prefix of chosen length, and AIPerf can synthesise prompts from trace `hash_ids` to reproduce KV-cache sharing patterns ([vLLM bench serve CLI](#); [AIPerf CLI options](#)). Forcing output length with `--ignore-eos` makes OSL deterministic, which suits regression tests but hides natural stopping variance; the run report should state which was used.

Warm-up, duration and steady state

The first requests of a run pay for cache population and connection setup; the last run against a draining queue. Both transients should be excluded. vLLM's `--num-warmups` defaults to 0 ([vLLM bench serve CLI](#)). AIPerf runs a separate warmup phase whose metrics "are discarded", configured with stop conditions such as `--warmup-request-count` or `--warmup-duration` ([AIPerf warmup tutorial](#)). GuideLLM accepts warmup and cooldown as fractions or absolute values inside the profile spec ([GuideLLM README](#)). GuideLLM v0.8.0 also added a `min_requests` constraint because a `max_requests` stop "tails off" at the end of the run, while `min_requests` keeps the full rate until the counted requests finish ([GuideLLM v0.8.0 release notes](#)).

Run length should be chosen from the percentile being reported. A p99 from 1,000 requests rests on the 10 slowest samples; a p99.9 from the same run rests on one. As illustrative arithmetic, 10,000 measured requests at 4 requests/s take 2,500 s, about 42 minutes. Repeat runs quantify the rest: AIPerf's `--num-profile-runs` (1 to 10) computes mean, standard deviation and confidence intervals across runs, at a default confidence

level of 0.95 ([AIPerf CLI options](#)). Report percentiles (p50, p90, p99), not means, because the mean of a latency distribution with a queueing tail describes no actual request. vLLM's `--metric-percentiles` defaults to "99" only, so 50, 90, 99 should be passed explicitly ([vLLM bench serve CLI](#)).

Goodput and SLO definition

Throughput counts every completed request; goodput counts only those that met their latency targets. DistServe framed serving performance "in terms of the maximum rate that can be served within both TTFT and TPOT constraints on each GPU", and reported serving "7.4x more requests or 12.6x tighter SLO" than prior systems "while staying within latency constraints for > 90% of requests" ([DistServe](#), [arXiv 2401.09670](#)). Those gains are the paper's results for its disaggregated design, not a general expectation (see the [disaggregated prefill/decode break-even](#) analysis).

Current tools implement goodput as follows (metric definitions are in the [TTFT, ITL and goodput definitions](#) guide):

- **vLLM:** `--goodput` takes KEY:VALUE pairs in milliseconds with keys `ttft`, `tpot`, `e2e1` ([vLLM bench serve CLI](#)).
- **AIPerf:** `--goodput "time_to_first_token:100 inter_token_latency:3.40"`, defined as "the number of completed requests per second that meet specified metric constraints" ([AIPerf goodput tutorial](#)); values are in each metric's display unit ([AIPerf CLI options](#)).
- **GuideLLM (v0.8.0+):** an SLO set through `--metrics '{"kind":"generative","slo":{"ttft_ms":200,"tpot_ms":50}}'` produces a "Goodput Attainment" figure, and a goodput profile "will search for the highest load that can meet the SLO ... at some target attainment level" ([GuideLLM v0.8.0 release notes](#)).

An SLO for capacity purposes needs three parts: per-request thresholds (for example $TTFT \leq 500$ ms and $TPOT \leq 50$ ms), an attainment level (for example 99% of requests), and the workload it applies to (arrival process, ISL/OSL distribution). **Worked example 5 (illustrative):** a run at 5 requests/s for 1,200 s completes 6,000 requests; 5,820 meet both thresholds. Goodput is $5,820 / 1,200 = 4.85$ requests/s and attainment is $5,820 / 6,000 = 97.0\%$. Against a 99% attainment target this rate fails, even though raw throughput (5.0 requests/s) matched the offered load.

Sweeping to find the knee

Capacity is the highest offered rate that still meets the SLO at the required attainment, found by sweeping rate in an open-loop test and plotting p99 TTFT, p99 TPOT and attainment against offered load. Below the knee, latency is nearly flat; beyond it, Worked example 3 applies and latency grows with run length. GuideLLM's sweep profile automates the bracket: it runs synchronous and throughput strategies first, then interpolates rates between them for asynchronous constant or poisson runs ([GuideLLM sweep profile reference](#)). Its knee profile instead fits a two-segment line to throughput versus concurrency and reports where throughput stops rising, which is a closed-loop saturation estimate rather than an SLO capacity ([GuideLLM knee guide](#)). GuideLLM's over-saturation constraint can stop a run when concurrency and TTFT both trend significantly upward ([GuideLLM over-saturation guide](#)). vLLM supports `--ramp-up-strategy linear|exponential` with -

--ramp-up-start-rps and --ramp-up-end-rps, and AIPerf accepts comma-separated lists such as --request-rate 10,20,50 to run a sweep ([vLLM bench serve CLI](#); [AIPerf CLI options](#)). A ramp mixes load levels in one run, so discrete steady-state steps give cleaner numbers; report the last passing step and show the next one failing.

Tool comparison

#	Capability	vllm bench serve	GuideLLM	AIPerf
1	Closed loop	--max-concurrency with --request-rate inf	synchronous, concurrent (streams=), throughput	--concurrency
2	Open-loop arrivals	--request-rate + --burstiness (gamma, 1.0 = Poisson)	constant, poisson profiles (no gamma)	--request-rate + --arrival-pattern constant/poisson/gamma
3	Rate plus cap	Yes; client queue time recorded	Yes: max_concurrency= on constant/poisson profiles	Yes ("Dual request control")
4	Goodput / SLO	--goodput ttft/tpot/e2e1	--metrics SLO, goodput profile (v0.8.0+)	--goodput
5	Sweep / search	Ramp-up; external scripting	sweep, knee, goodput profiles	List-valued flags, multi-run CIs
6	Trace replay	timed_trace dataset (trace timestamps)	WEKA, OTEL traces (v0.8.0+)	--fixed-schedule, mooncake/bailian/baseten traces
7	Warm-up	--num-warmups (default 0)	warmup=/cooldown= in profile	Warmup phase, metrics discarded
8	Install	Part of vLLM	pip install guidellm[recommended]	pip install aiperf

Compiled from the tool documentation cited in the sections above.

Reporting template and pitfalls

The minimum publishable record:

#	Field	Example content
1	Hardware	GPU model and count, interconnect, power cap, host CPU
2	Software	Engine and version, driver, CUDA/ROCm, benchmark tool and version
3	Serving config	Model, precision, parallelism, max batch tokens, prefix caching on/off
4	Load model	Open/closed, target and achieved rate, arrival process and burstiness, concurrency cap
5	Workload	Dataset or trace, ISL/OSL distribution, ignore_eos, shared prefix length, seed

#	Field	Example content
6	Run control	Warm-up, duration or request count, number of repeats, client location
7	Results	p50/p90/p99 TTFT, TPOT/ITL, E2E (with client queue time), goodput and attainment, errors

Recurring pitfalls: quoting a closed-loop peak as "capacity"; running vLLM with the default infinite rate; reporting means; changing tools mid-baseline (the [migrating GenAI-Perf baselines to AIPerf](#) guide covers how to bridge that); running the client on the same host as the server, where it competes for CPU; and treating a run whose p99 grows with duration as a latency result. GPU Smith can review a load-test design against these criteria on request.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://www.usenix.org/legacy/event/nsdi06/tech/full_papers/schroeder/schroeder.pdf
2. <https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/OS/Littles-Law-50-Years-Later.pdf>
3. <https://github.com/giltene/wrk2>
4. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/benchmarks/serve.py>
5. <https://docs.vllm.ai/en/latest/cli/bench/serve>
6. <https://docs.nvidia.com/aiperf/reference/command-line-options>
7. <https://github.com/vllm-project/guidellm>
8. <https://docs.vllm.ai/en/latest/benchmarking/cli>
9. <https://docs.nvidia.com/aiperf/benchmark-modes/load-generator-options-reference>
10. <https://github.com/ai-dynamo/aiperf>
11. <https://github.com/vllm-project/guidellm/releases/tag/v0.8.0>
12. <https://docs.nvidia.com/aiperf/tutorials/load-patterns-scheduling/warmup-phase-configuration>
13. <https://arxiv.org/abs/2401.09670>
14. <https://docs.nvidia.com/aiperf/tutorials/metrics-analysis/benchmark-goodput-with-ai-perf>
15. <https://vllm-project.github.io/guidellm/latest/reference/guidellm/benchmark/profiles/sweep>
16. https://vllm-project.github.io/guidellm/latest/guides/knee_detection
17. https://vllm-project.github.io/guidellm/latest/guides/over_saturation_stopping

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.