GPU SMITH
COMPUTE ENGINEERING

GPU SMITH / RESEARCH & PRACTICAL GUIDANCE

Private LLM Overload Control: Queues, Admission, Retries

> 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 · 2026-11-21 · llm overload control · llm admission control · vllm overload protection · litellm admission control · load shedding · retry storm prevention · llm request queue timeout · inference backpressure · private llm capacity planning

Original article: https://gpusmith.com/articles/en/private-llm-overload-control-queues-admission-retries



In this article

1. Executive Summary
2. Introduction and Background
3. Key Changes in How Overload Is Controlled
4. Implementation Considerations and Process Changes
5. Data Analysis and Evidence
6. Implications and Future Directions
7. Frequently Asked Questions (FAQs)
8. Conclusion

Executive Summary

Overload control in a private LLM deployment is not one setting. It is a chain of independently scoped limits, each owned by a different process, and a logical request crosses all of them. This report follows one request through the client SDK, the ingress, a gateway such as LiteLLM, one or more vLLM API-server processes and the per-rank scheduler, and records for each hop what is counted, where overflow goes and who retries. As of November 2026, the evidence is a versioned documentation reference, not a measured benchmark.

The central finding is that request caps, token caps and sequence caps are different controls with different scopes. In **vLLM v0.30.0**, the request cap rejects new requests with HTTP 503 at a configured limit (docs.vllm.ai), the prefill cap counts prompt tokens of requests in the prefill phase (docs.vllm.ai), and the per-iteration token budget is a separate scheduler setting. **LiteLLM** admission is per worker process (docs.litellm.ai), and a pod with 4 workers and a cap of 64 admits up to 256 concurrent requests. The sum of local caps is a configuration bound, not proof of usable GPU capacity.

Retries are the second finding. The OpenAI Python SDK retries certain errors 2 times by default (github.com), and the Anthropic SDK does the same (platform.claude.com). Google's SRE guidance says requests should only be retried at the layer immediately above the one rejecting them (sre.google), and AWS gives an illustrative example, with five layers and three retries per layer, in which load rises 243x when every layer retries independently (builder.aws.com). Measured retry amplification is total attempts divided by unique original requests, counting rejected and cancelled attempts; the configured worst case is the product of per-layer maximum attempts.

The third finding concerns evidence. Success-only latency cannot certify overload safety, because rejected, cancelled, timed-out and unfinished requests are the outcomes overload creates. AWS defines goodput as the subset of throughput handled without errors (builder.aws.com), and a promotion decision needs the full outcome ledger. Documentation is evidence of a feature contract, not proof the local deployment enforces it: the stable vLLM documentation lists an active-sequence control absent from the v0.30.0 page (docs.vllm.ai).

The deliverables are a versioned control dictionary (Table 1) and an acceptance-test matrix (Table 2). Pass criteria come from the owner's service-level objective (SLO) and failure budget; no universal queue length is offered, and none exists in the sources. A configuration should be promoted only when every outcome reconciles, cancellation releases backend resources, nested retry bounds are known, and burst behavior stays inside the budget. When it does not, the correct response is more measured capacity, not a larger queue.

Introduction and Background

Teams that run a private LLM service usually discover overload control by accident. A traffic burst arrives, a queue grows somewhere, clients time out, SDKs retry, and the retried work arrives at the same queue. The Google SRE book describes the general shape: the most common cause of cascading failures is overload (sre.google), and capacity planning reduces the probability of triggering one but is not sufficient (sre.google). GPU-specific claims here are limited to what the vendors' own documentation and source code state.

The question for a serving platform owner is practical. Which layer admits work, which layer makes it wait, and which layer rejects it? What does each limit actually count, and over what scope? How do clients retry and cancel? And is the burst behavior safe enough to promote a configuration, or is more measured capacity required? This is not a batching optimization guide, a runtime upgrade note, a replica-routing comparison or a generic SLO primer.

Scope and method

The method is documentary and dated. Every product fact was read from a vendor or standards-body page, or from a pinned source file, on 2026-11-21. The vLLM facts are tied to **v0.30.0** unless a stable-documentation page is named, because stable pages move. The v0.30.0 release is a pinned reference, not the latest release: a community issue thread refers to later tags, including v0.32.0 and v0.33.0, so check on the installed version whether the active-sequence control is present. Where documentation and source differ, the report turns the difference into a test.

Two public explainers cover adjacent ground and are cited only for what they state. One, published 2026-09-20, says vLLM's queue "has no depth limit as of Sep 2026" (www.spheron.network), while the v0.30.0 page lists request and prefill-token admission caps (docs.vllm.ai). The difference may reflect the software version or wording each source describes, and this report does not infer which; run the help command on the installed release. A second, published 2026-08-01, defines admission as deciding whether the service can make a useful promise under current policy (jpcasabianca.com) and advises validating the deployed server (jpcasabianca.com). The contribution here is an installed-stack scope reference with outcome-based acceptance.

Key Changes in How Overload Is Controlled

The following changes describe how the control surface differs from the single-queue mental model that many teams carry over from smaller deployments.

Change 1: One logical request crosses several scoped queues

A single user prompt can be held in a client SDK's retry loop, an ingress or proxy read timeout, a gateway admission queue, an API-server in-flight counter and the engine's waiting queue. Each has its own unit and its own scope. Build a topology map first, with one row per process and one column per counter.

- **Client or SDK:** retry count, per-attempt timeout and total deadline. The OpenAI SDK defaults to a 10 minute timeout (github.com).
- **Ingress or proxy:** read timeout, retry-to-next-server rules and connection limits. In nginx, the read timeout applies between two successive reads, not to the whole response (nginx.org).
- **Gateway (LiteLLM):** per-worker in-flight cap, per-worker queue cap and queue timeout, plus per-key quotas.
- **API-server process (vLLM):** in-flight request cap and prefill-token backlog cap.
- **Per-rank scheduler (vLLM):** per-iteration sequence and token budgets, preemption and recompute.

The sum of independently scoped local caps is a configuration bound. It proves neither that the fleet has that much usable GPU capacity nor that load is balanced across the processes holding the counters.

Change 2: Request caps, prefill-token caps and sequence caps are different controls

In v0.30.0, the in-flight request cap counts unfinished requests, meaning waiting plus running (github.com). On overflow the engine raises an error whose message says it is busy and cannot accept new requests (github.com). The prefill-token cap is a different unit: the total prompt tokens of requests currently in the prefill phase, with its own overflow message (github.com). Neither is the scheduler's per-iteration limit, which bounds tokens processed in a single iteration (docs.vllm.ai).

Two behaviors matter for acceptance tests. First, the token counter is conservative: a partially processed request is counted at its full prompt length until it leaves prefill, which the documentation says overestimates the real backlog and causes earlier rejection than strictly necessary (docs.vllm.ai). A long-prompt workload under chunked prefill will therefore reject earlier than a simple tokens-per-second calculation predicts. Second, the scheduler prioritizes decode requests and fills the remaining token budget with prefills (docs.vllm.ai), so prefill progress slows when decode load is high, and the wait implied by a token cap moves with the request mix.

Change 3: Scope is process-local, and data parallelism changes what a number means

The v0.30.0 data-parallel page states that each API server process counts in-flight requests and prefill backlog across all data-parallel ranks it routes to (docs.vllm.ai). The command-line text says that, unlike the per-rank sequence limit, the request limit is enforced in the API server process (docs.vllm.ai). A deployment with four ranks and a per-rank sequence limit of 256 can therefore reject at a request cap far below the 1,024 sequences the ranks could run, if the cap is set to the per-rank number. The number of API-server processes is separately configurable (docs.vllm.ai).

An unresolved point deserves an explicit test. The v0.30.0 source contains a shared-slot admission structure described as lock-free admission statistics shared by API server processes (github.com), yet the documentation phrases the limit per API server process. This research did not trace where the sharing is enabled in the launch path. Until a test shows where the first 503 appears when N requests are sent through each API-server process, treat the request cap as process-local and do not assume it is fleet-wide.

Metrics have a related scope problem. vLLM metrics are collected in the API server process (docs.vllm.ai), so per-process scrape targets must be summed or labeled correctly before they feed a dashboard or autoscaler.

Change 4: Gateway admission is per worker and off unless configured

LiteLLM documents per-worker admission with an in-flight cap, a queue cap and a queue timeout; the feature is off unless the in-flight cap is set, and a request that finds no free slot waits in the queue until a slot frees or the timeout elapses (docs.litellm.ai). The configuration reference describes the cap as per worker process (docs.litellm.ai). Because queued requests stay with the worker that accepted them (docs.litellm.ai), uneven balancing can leave one worker rejecting while another idles, which is the skewed-balancing test in Table 2.

Two further documented properties shape the error taxonomy. Admission rejections happen before authentication and are not attributed to a key in spend logs (docs.litellm.ai), so a tenant-level view of rejected work needs gateway-level counters rather than per-key logs. And on Anthropic-format routes the rejection uses Anthropic's error body (docs.litellm.ai), so client code that parses the body must handle both shapes. The earliest release note found that mentions the feature is v1.101.0, which lists per-worker admission control that answers overload with an immediate 503 (docs.litellm.ai); pin the installed gateway version and re-test.

Change 5: Capacity rejection and quota rejection are different signals

The per-key, per-user and per-team limits in LiteLLM apply to any key, user or team that has a limit set (docs.litellm.ai), and limits that are not model-specific still return 429 on every model (docs.litellm.ai). A proxy-generated concurrency 429 is raised before any provider call, so it does not count toward a deployment's cooldown (docs.litellm.ai). Cooldowns apply to individual deployments, not entire model groups (docs.litellm.ai).

The status-code vocabulary is therefore: 429 for a tenant quota (the user has sent too many requests in a given amount of time (www.rfc-editor.org)), 503 for capacity, and gateway or proxy timeouts for everything else. RFC 9110 describes 503 as a server unable to handle the request due to a temporary overload or scheduled maintenance (www.rfc-editor.org), so a bare 503 does not separate a full worker from maintenance, and the mapping must be confirmed on the installed stack. Both vLLM and LiteLLM use 503 for capacity, which means a client or intermediary cannot tell "this worker is full" from "the service is unavailable" without the body, the header or an extra signal. The SRE guidance recommends returning a specific status when overloaded so that clients and other layers back off and do not retry (sre.google).

Change 6: Retries need one owner, a deadline and a finite budget

The OpenAI SDK retries certain errors twice with a short exponential backoff (github.com), and for server-supplied delays it prefers milliseconds, then seconds (raw.githubusercontent.com). The Anthropic SDK retries certain errors twice (platform.claude.com), and requests that time out are retried twice by default (platform.claude.com). LiteLLM has its own retry loop distinct from the provider SDK's internal retry count (docs.litellm.ai), and a retry policy can set retries per exception type (docs.litellm.ai). An nginx proxy only treats a 503 or 429 as an unsuccessful attempt when told to (nginx.org), and it can cap the total time spent passing a request onward (nginx.org).

The general guidance converges on one rule. AWS says its best practice is to retry at a single point in the stack (builder.aws.com), and the SRE book says to avoid amplifying retries by issuing retries at multiple levels (sre.google). Side effects add a correctness constraint: a proxy must not automatically retry non-idempotent requests (www.rfc-editor.org), and APIs with side effects are unsafe to retry unless they provide idempotency (builder.aws.com). For LLM calls that trigger tool actions, test request identifiers and server-side deduplication before any layer replays work.

Change 7: Cancellation is a contract to be proven per path

Cancellation reduces doomed work (sre.google), and in v0.30.0 the engine aborts a request when its generator is cancelled or collected (github.com). At the HTTP layer, the wrapper hands disconnect listening to the response object for streaming responses (github.com). LiteLLM states that a client disconnect, queued or

in flight, releases the admission slot immediately (docs.litellm.ai), but its separate option that cancels the upstream request is documented for non-streaming calls only (docs.litellm.ai). A community issue reported that a nightly build did not detect client disconnect on non-streaming chat completions (github.com); the issue was later closed as resolved by a fix that postdated that nightly. Test non-streaming and streaming disconnects separately on the installed release, because behavior can vary by release and path.

Streaming and non-streaming cancellation therefore need separate tests, and a freed gateway slot does not prove the engine stopped computing. The proof is a backend observation: running and waiting gauges fall and no tokens are generated after the disconnect.

Change 8: Documentation drift between releases

The stable vLLM command-line page documents a control that limits how many requests the scheduler admits into the running set, and the same page also documents the in-flight request cap (docs.vllm.ai). The active-sequence option was absent from the opened v0.30.0 page, so it must not be backported by assumption. The original community proposal for a waiting-queue limit used a differently named flag, and a proposal does not establish what shipped (github.com). Record, for every control, the version, the source page and the output of the help command on the installed binary.

Implementation Considerations and Process Changes

The versioned control dictionary

Table 1 below is the control dictionary, version 1.0, dated 2026-11-21. Each row names a control, its unit, its scope, its documented default and overflow behavior, the metric or log that proves it fired, and the test that exercises it. "Unsupported or unverified" is a valid cell value and is preferable to a guess.

Layer and control	Unit and scope	Default and overflow behavior	Evidence it fired	Test
vLLM request cap (v0.30.0)	Requests, waiting plus running (github.com); enforced in the API server process	Off unless set; new requests rejected with 503	503 body "engine is currently busy" (github.com); waiting and running gauges (docs.vllm.ai)	T1, T5
vLLM prefill-token cap (v0.30.0)	Prompt tokens in prefill; counted conservatively	Off unless set; 503 with its own message (github.com)	503 body; request queue-time histogram (docs.vllm.ai)	T2, T6
vLLM per-iteration budgets	Tokens per iteration; sequences per rank	Scheduler setting, not an admission cap; KV pressure triggers preemption and recompute (docs.vllm.ai)	Preemption counters, queue time	T3
vLLM active-sequence cap (stable docs only)	Requests admitted into RUNNING	Absent from the opened v0.30.0 page: mark unsupported on 0.30	Version-gated	Check help output

Layer and control	Unit and scope	Default and overflow behavior	Evidence it fired	Test
LiteLLM in-flight cap	Requests per worker process; total is cap times worker count (docs.litellm.ai)	Off unless set; waits in queue, then 503	Admission gauges; 503, with client retry guidance (docs.litellm.ai)	T4, T5
LiteLLM queue cap and timeout	Queued requests per worker; seconds	Config reference lists queue timeout default of 1.0 s and a per-worker queue cap that defaults to `null`, meaning the same value as the in-flight cap (docs.litellm.ai)	Queue wait histogram per worker	T4
LiteLLM tenant quota	RPM, TPM, parallel requests per key, user or team	429 on exceed	Per-key logs (admission 503 is not attributed)	T7
LiteLLM upstream cancel	Non-streaming only	Default false	Engine abort observed	T8
SDK retries	Attempts per call, per-attempt timeout	Two retries by default	Client attempt IDs	T9
Proxy retry and read timeout	Tries, seconds (nginx.org)	Retries on 503 or 429 only if configured (nginx.org)	Proxy access log	T9, T10

The table is deliberately a template. Cells that cite documentation record a contract; the "Evidence it fired" column records what must be observed locally before the contract is believed. Other serving layers fit the same dictionary. NVIDIA Triton's queue policy has a maximum queue size whose default of 0 means no maximum (raw.githubusercontent.com). Hugging Face TGI documents that a low concurrent-request limit refuses requests instead of making them wait (huggingface.co). SGLang's queued-request cap is ignored in disaggregation mode (docs.sglang.io). Ray Serve counts queued requests at each caller (docs.ray.io) and returns 503 by default (docs.ray.io). Envoy circuit breakers are on by default with modest limits (www.envoyproxy.io). Each has a unit, a scope, a default that often means no limit, and an overflow status to verify.

Table 1 shows that no row is both global and request-and-token aware, so the dictionary must carry scope as a first-class field, and a promotion review should reject any configuration whose scope cell reads "unknown".

Choosing a candidate waiting budget from traces

The residual waiting budget for a request class is the end-to-end deadline minus the measured non-waiting work and an explicitly chosen reserve. The non-waiting work must come from per-request traces on the target workload: network and preprocessing, prefill, first-token and decode time, and any cold-start cost, recorded separately. Subtracting unrelated p95 statistics from each other is not a valid derivation, because the percentiles do not come from the same requests. Write the formula in the dictionary as: residual wait = deadline minus (network and preprocessing + prefill + decode, measured on the same request class) minus reserve. The components to extract per request class are:

- **Network and preprocessing time:** client to ingress, gateway handling, tokenization and templating, measured separately from engine time.

- **Prefill time:** from execution start to first token, tagged with prompt length and prefix-cache state.
- **Decode time:** from first token to final token, tagged with output length.
- **Cold-start cost:** model load, warm-up or first-request penalties, counted once and excluded from steady-state budgets.
- **Reserve:** an explicitly chosen margin owned by the service owner, recorded in the dictionary with its rationale.

For the prefill-token cap, the vLLM documentation presents the cap as a quality-of-service tool for time to first token. A starting hypothesis is: candidate cap = allowed prefill-wait seconds multiplied by measured aggregate prefill tokens per second for the actual topology and request mix. This is a starting point to validate, not a p99 guarantee, for three reasons: the counter overestimates partially processed requests, decode priority slows prefill under load, and prefix-cache hits change the real work. Test hot and cold cache states separately.

A sanity check on queue depth uses Little's Law. The law is stated for long-run averages under steady-state conditions (www.ic.unicamp.br), with finite means and strictly stationary processes in the original statement (ideas.repec.org), and Little's later paper describes classical results as primarily concerned with relationships among long-run averages (people.cs.umass.edu). Use L = arrival rate x mean time as a units check on requests in the system, not as a tail bound for bursty, variable-length traffic.

Coordinating limits and error propagation

Order the limits so the cheapest rejection happens first and the most informative one reaches the client. The SRE book notes that rejected requests still consume some resources (sre.google), and says an overloaded backend should reject the excess rather than turn down and stop accepting all traffic (sre.google). AWS recommends a spillover configuration that fast-fails instead of queueing excess requests (builder.aws.com). A design to validate: tenant quota at the gateway (429), a short gateway queue bounded by the residual waiting budget, and engine caps set slightly above the gateway total so they act as a backstop.

For each rejection, document the status, body shape and header that reaches the client. Retry-After can be an HTTP-date or a number of seconds (www.rfc-editor.org), and LiteLLM's HAProxy guide tells clients to honor it with bounded, jittered retries and a total deadline. If a layer rewrites a 503 into a generic 502 or 504, retry rules keyed on 503 will silently stop working. Streaming adds a case: vLLM stable documents a keep-alive comment interval for idle streaming connections (docs.vllm.ai), which matters when a proxy read timeout is shorter than the queue wait.

Bounding retries across layers

Define two numbers and report both. The **measured retry amplification** is total attempts divided by unique original requests, counting rejected and cancelled attempts. The **configured worst-case nested attempt bound** is the product of the maximum attempts of every layer that actually retries. The second is a ceiling used for design, not a prediction, because not every worst case occurs together.

The SRE book offers external reference points for budgets: a per-request limit of up to three attempts (sre.google) and a per-client ratio of retries to requests that must stay below 10 percent (sre.google). These are Google's choices for its own systems, not LLM thresholds, and are cited as examples of the form a budget takes. Add randomized exponential backoff (sre.google), jitter to spread retries in time (builder.aws.com), and

a local token bucket to limit retries (builder.aws.com). Propagate one deadline from the top of the stack (sre.google), since gRPC does not set a deadline by default (grpc.io) and clients should always set an explicit, realistic one (grpc.io). Envoy recommends retry budgets over fixed caps (www.envoyproxy.io).

Acceptance traffic and the test matrix

Use open-loop arrivals. In a closed system a new request is only triggered by the completion of a previous request (www.usenix.org), and mean response times are significantly lower in closed systems than in open systems (www.usenix.org), so a closed-loop test understates overload. The vLLM benchmark tool's default request rate of infinity sends every request at time 0 (docs.vllm.ai), which is a single step burst; a finite rate with burstiness 1 follows a Poisson process (docs.vllm.ai), and a concurrency limit can make the achieved rate lower than the requested one (docs.vllm.ai), partially closing the loop. A ramp mode can sweep the rate (docs.vllm.ai). Coordinated omission, the flaw that occurs whenever an observed response time is longer than the interval the testing thread tries to maintain (groups.google.com), is the reason to log intended send times.

Table 2 below is the acceptance-test matrix. Pass criteria are deliberately written as references to the owner's SLO and failure budget; the sources provide no universal numbers.

ID	Scenario	What to record	Pass criterion (owner-defined)
T1	Step burst above the request cap, realistic length mix	Every original and attempt ID, outcome class, wait time, first-token and final-token times	Rejections within the failure budget; every request reconciles to one outcome
T2	Long-prompt burst against the prefill-token cap, cold then hot prefix cache	Prompt tokens, cache state, queue time, 503 count	Rejection onset explained by the conservative counter; no silent timeouts
T3	Sustained saturation at the knee, ramped rate	Completion rate (goodput), preemptions, queue time	Goodput flat or rising as offered load rises; if it falls, more capacity is needed
T4	Skewed balancing across gateway workers	Per-worker in-flight, queued and rejected counts	No worker rejects while the fleet has spare capacity inside the budget
T5	Add API-server processes or workers without adding GPUs	Admission onset versus completion rate	Reject if admitted load rises but usable completion rate does not
T6	Data-parallel imbalance: one rank slow	Per-rank running and waiting	Cap behavior matches the documented scope
T7	Tenant over quota during a burst	429 versus 503 counts by key	Quota and capacity rejections are distinguishable at the client
T8	Client disconnect, streaming and non-streaming, queued and running	Engine running and waiting gauges, tokens generated after disconnect	Backend resources released within the stated interval, on both paths
T9	Client timeout shorter than queue wait	Attempt IDs, retry timing, Retry-After handling	Measured amplification within the declared bound; no retry past the deadline

ID	Scenario	What to record	Pass criterion (owner-defined)
T10	Process loss and recovery: kill a worker, then an API server	Outcomes of in-flight requests, readiness timing	In-flight work reconciles; recovery within the objective; no replay of non-idempotent calls

Table 2 clarifies that most rows test an interaction between layers, not a single flag. T5 is the most commonly skipped and the most informative: adding processes multiplies local admission limits, and the test shows whether usable completion rate follows. For T10, Kubernetes gives pods a default termination grace period of 30 seconds (kubernetes.io) and sends a kill signal to remaining processes after expiry (kubernetes.io), so in-flight streaming work has a bounded drain window unless it is configured otherwise.

The per-logical-request ledger and outcome reconciliation

The evidence record has two keys: an original request ID that survives retries and an attempt ID per hop. Required fields per attempt are:

- **Identifiers:** original request ID, attempt ID, tenant and the hop that handled the attempt.
- **Timestamps:** admission, wait start, execution start, first token, final token, rejection, cancellation and client-side timeout.
- **Workload tags:** prompt tokens, output tokens, streaming or not, and prefix-cache state.
- **Outcome and producer:** the final outcome class and the layer that produced it.
- **Backend observation:** waiting and running counters, host memory and the observed release of backend resources after cancellation.
- **Window metadata:** pinned container, model and configuration, topology map and the time window.

Outcome classes should be mutually exclusive so that the identity below closes:

- **Completed:** a full response was delivered within the deadline.
- **Finally rejected:** the last attempt received a capacity or quota rejection.
- **Cancelled:** the client or an intermediary abandoned the request.
- **Timed out:** the deadline or a per-layer timeout ended the request.
- **Errored:** any other failure, including process loss.
- **Unfinished:** still in flight at window end.

The reconciliation identity for a measurement window is: original arrivals = completed + finally rejected + cancelled + timed out + errored + unfinished at window end. If the identity does not close, the test is invalid. A finite-window form of Little's Law holds when there are no lost customers, so arrivals equal departures (www.ic.unicamp.br), which is why the window should start and end empty or the leftovers be counted as unfinished. OpenTelemetry's HTTP conventions leave the error type unset only on success (opentelemetry.io), so rejected and cancelled requests can share a histogram with an outcome label, and histogram bucket edges should sit at the SLO thresholds (prometheus.io). Reject a configuration when success-only latency hides missing outcomes.

Data Analysis and Evidence

Documented reference numbers

Table 3 collects numeric reference points from primary sources; each is a documented value, not a recommendation.

Reference	Value	How to use it
SDK default retries	2 retries for the OpenAI and Anthropic SDKs (see Change 6)	Counts as a retrying layer in the nested bound
SRE per-request retry budget	Up to 3 attempts (sre.google)	External example of a bounded budget
SRE per-client retry ratio	Below 10% (sre.google)	External example of a ratio budget
SRE adaptive-throttling multiplier	2x preferred (sre.google)	Client-side throttling form, not an LLM threshold
Hedging overhead in Dean and Barroso	About 5% extra load (cacm.acm.org)	Duplicate work needs a stated budget
AWS five-layer retry example	243x load (builder.aws.com)	Illustrates multiplication across layers
Kubernetes termination grace	30 seconds (kubernetes.io)	Default drain window for T10

Table 3 shows that the externally documented numbers are mostly about retries and defaults, not about GPU capacity. That is consistent with the sources' scope: Google SRE and AWS guidance does not establish GPU throughput or a universal queue threshold.

Worked calculations (Hypothetical Example)

The following figures are invented to illustrate arithmetic, not measurements. Suppose a team measures an aggregate prefill rate of 40,000 prompt tokens per second on its actual topology and request mix, and a trace-derived residual budget allows 2 seconds of prefill wait. The candidate prefill-token cap is 2 x 40,000 = 80,000 tokens, to be validated by T2. The conservative counter may bring rejection onset earlier than this arithmetic suggests; record by how much.

Now suppose a client SDK allows 2 retries (3 attempts), a gateway retries up to 2 times (3 attempts) and a proxy tries 2 servers. The configured worst-case nested bound is 3 x 3 x 2 = 18 attempts per original request, which is a ceiling and is unlikely to occur at once. If a test of 1,000 original requests produced 1,420 total attempts, including rejected and cancelled ones, the measured amplification is 1.42. The acceptance question is whether 1.42 sits inside the owner's declared retry budget at the tested burst size, and whether the 18-attempt ceiling is tolerable. If not, remove retries from two of the three layers.

For a gateway with 4 workers and an in-flight cap of 64, the admitted total is 256, as in the documentation example (docs.litellm.ai). If the engine's request cap is below 256, the engine rejects traffic the gateway admitted; if far above, the gateway queue is the only protection. The dictionary must say which layer is meant to reject first.

What the numbers cannot show

Little's Law is a long-run relation under stationarity, not a percentile (people.cs.umass.edu). Goodput is the maximum request rate that can be served adhering to the SLO (arxiv.org), and the vLLM benchmark tool accepts goodput thresholds in milliseconds (docs.vllm.ai). A configuration whose goodput rises until the knee and then holds is behaving as intended; AWS cautions that discarded work creates a positive feedback loop that increases the overload (builder.aws.com) and that overload testing is crucial before building avoidance mechanisms (builder.aws.com).

Implications and Future Directions

Promote, hold or buy capacity

The decision rule follows from the ledger. Promote a configuration when all of the following hold:

- **Reconciliation:** every test in Table 2 reconciles all outcomes.
- **Budget:** rejections and timeouts at the owner's burst size are inside the failure budget.
- **Cancellation:** both streaming and non-streaming paths release the intended backend resources.
- **Retries:** the measured retry amplification and the configured nested bound are both accepted by the owner.

Hold when any scope cell in Table 1 is unknown. Buy or provision more measured capacity in any of these cases:

- **Goodput falls:** completions per second drop as offered load rises.
- **Budget exceeded:** rejections exceed the budget even after the waiting budget is tuned to the residual deadline.
- **Admission without capacity:** T5 shows admission growing without completion rate.

A larger queue converts rejections into timeouts and does not add capacity. As the SRE chapter puts it, small queues relative to the thread pool are usually better (sre.google).

Changes that require re-running the matrix

Any change to the following invalidates earlier results:

- **Software version:** the vLLM or LiteLLM release.
- **Process counts:** API-server processes or gateway workers.
- **Parallelism:** the data-parallel size.
- **Traffic shape:** the request length mix and the prefix-cache hit rate.
- **Retry policy:** SDK, gateway or proxy retry settings.
- **Timeouts:** ingress, proxy and client timeouts.
- **Scaling rules:** the autoscaler's triggers.

AWS also notes that if misconfigured, load shedding can disable reactive automatic scaling (builder.aws.com), so scale triggers should be tested alongside rejection rules. The stable vLLM docs already differ from v0.30.0, and a community disconnect report was closed as resolved by a later fix (github.com), so version drift is a routine event, not a rare one.

Ownership and rollback

Each control in the dictionary needs an owner, a rollback configuration and an alert. Rollback means a recorded set of previous values, including the "off" state, because several controls are off by default. A release record should hold:

- **Owner:** the named team accountable for the control.
- **Values:** current and previous settings, including "off".
- **Versions:** vLLM, gateway, SDK and proxy releases.
- **Evidence:** the test run identifiers and the date.
- **Trigger:** the observed condition that forces rollback.

The dictionary should be versioned like code, with the date, the software versions and the test run identifiers attached to each release.

Open questions remain outside this report's evidence: whether the shared admission slots in v0.30.0 source make the request cap effectively global in a given launch mode, how cancellation behaves on every streaming path, and how per-worker queue fairness degrades under skewed load. Each is answered by a test in Table 2, not by documentation. GPU Smith writes as an independent advisor and not as a vendor of any serving software discussed; the report is analysis to be checked against local measurements and carries no claim of third-party review.

Frequently Asked Questions (FAQs)

What is LLM admission control, and where should it run?

Admission control decides whether a request enters, waits or is rejected. In the stack studied it can run at the gateway (per-worker caps), at the API server (request and prefill-token caps) and at the tenant quota layer. Run the cheapest, most informative rejection first and keep engine caps as a backstop.

How big should an LLM inference queue budget be?

No source supports a universal number. Derive the residual waiting budget from traces, convert it to a token budget with measured prefill throughput, and validate it with T2 and T3 as a hypothesis.

How do I do self-hosted LLM load shedding without retry storms?

Shed with a clear 503 and a retry hint, make one layer the retry owner (builder.aws.com), bound attempts and total time, add jitter (builder.aws.com) and enforce a retry budget. Measure amplification as attempts divided by unique original requests.

What is a reasonable LLM request queue timeout?

Shorter than the deadline minus the work that follows the queue. Triton's queue timeout defaults to none (raw.githubusercontent.com), and the opened vLLM v0.30.0 page documents no per-request queue timeout, so enforce one at the gateway or client.

Does vLLM have overload protection?

In v0.30.0, yes: optional request and prefill-token caps that return 503, off unless configured. Preemption under KV pressure is a different mechanism, not admission.

How do I write a 429 retry backoff acceptance test?

Send an open-loop burst above the cap, record attempt IDs, verify that clients honor Retry-After (www.rfc-editor.org) within the deadline, and assert that measured amplification stays inside the declared budget (T9). Distinguish tenant 429s from capacity 503s (T7).

How does this feed private LLM capacity planning?

Use T3 and T5 to find the knee and to check that more admission slots yield more completions. If they do not, the fix is capacity, not configuration.

Conclusion

Overload control for private LLM serving is an exercise in scope accounting. The limits that exist in current software are real, documented and versioned, but they count different things in different processes: requests versus prompt tokens versus sequences, per worker versus per API-server process versus per rank. A configuration is safe only when those scopes are mapped, the error and retry behavior across layers is known, cancellation is shown to release backend resources, and all outcomes, not only completions, reconcile at the owner's burst size.

The control dictionary and acceptance matrix in this report give a repeatable way to reach that conclusion for an installed stack, and they state the limits of the evidence: documentation is a feature contract, a sum of local caps is not fleet capacity, and Little's Law is a sanity check, not a tail bound. Where tests fail the owner's SLO and failure budget, the evidence points to more measured capacity.

Source links

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

1. https://docs.vllm.ai/en/v0.30.0/cli/serve
2. https://docs.litellm.ai/docs/proxy/server_tuning.md
3. https://github.com/openai/openai-python/blob/main/README.md
4. https://platform.claude.com/docs/en/api/sdks/python
5. https://sre.google/sre-book/handling-overload
6. https://builder.aws.com/content/3EumjoZascWd1oZiEgL8ORlv3qE/timeouts-retries-and-backoff-with-jitter

7. https://builder.aws.com/content/3Eun1EEyX6p2e3VYNyRLSJzLuMV/using-load-shedding-to-avoid-overload
8. https://docs.vllm.ai/en/stable/cli/serve
9. https://sre.google/sre-book/addressing-cascading-failures
10. https://github.com/vllm-project/vllm/issues/61918
11. https://www.spheron.network/blog/llm-inference-overload-queuing-load-shedding-and-admission-c
12. https://jpcasabianca.com/journal/llm-admission-control
13. https://nginx.org/en/docs/http/ngx_http_proxy_module.html
14. https://github.com/vllm-project/vllm/blob/v0.30.0/vllm/v1/engine/async_llm.py
15. https://github.com/vllm-project/vllm/blob/v0.30.0/vllm/exceptions.py
16. https://docs.vllm.ai/en/v0.30.0/configuration/optimization
17. https://docs.vllm.ai/en/v0.30.0/serving/data_parallel_deployment
18. https://github.com/vllm-project/vllm/blob/v0.30.0/vllm/v1/engine/admission_control.py
19. https://docs.vllm.ai/en/v0.30.0/design/metrics
20. https://docs.litellm.ai/docs/proxy/config_settings.md
21. https://docs.litellm.ai/docs/proxy/deploy/ha/haproxy.md
22. https://docs.litellm.ai/release_notes/v1.101.0/v1-101-0.md
23. https://docs.litellm.ai/docs/proxy/users
24. https://docs.litellm.ai/docs/routing
25. https://www.rfc-editor.org/rfc/rfc6585.html
26. https://www.rfc-editor.org/rfc/rfc9110.html
27. https://raw.githubusercontent.com/openai/openai-python/main/src/openai/_base_client.py
28. https://github.com/vllm-project/vllm/blob/v0.30.0/vllm/entrypoints/serve/utils/api_utils.py
29. https://github.com/vllm-project/vllm/issues/18826
30. https://docs.vllm.ai/en/v0.30.0/usage/metrics
31. https://raw.githubusercontent.com/triton-inference-server/common/main/protobuf/model_config.proto
32. https://huggingface.co/docs/text-generation-inference/basic_tutorials/launcher
33. https://docs.sglang.io/advanced_features/server_arguments.html
34. https://docs.ray.io/en/latest/serve/configure-serve-deployment.html
35. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/circuit_breaking
36. https://www.ic.unicamp.br/%7Enfonseca/MO648/doc/Little%27s%20Law-Published.pdf
37. https://ideas.repec.org/a/inm/oropre/v9y1961i3p383-387.html
38. https://people.cs.umass.edu/%7Eemery/classes/cmpsci691st/readings/OS/Littles-Law-50-Years-Later.pdf
39. https://grpc.io/docs/guides/deadlines
40. https://www.usenix.org/legacy/event/nsdi06/tech/full_papers/schroeder/schroeder.pdf
41. https://docs.vllm.ai/en/latest/cli/bench/serve
42. https://groups.google.com/g/mechanical-sympathy/c/icNZJejUHfE/m/BfDekfBEs_sJ
43. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle

44. https://opentelemetry.io/docs/specs/semconv/http/http-metrics
45. https://prometheus.io/docs/practices/histograms
46. https://cacm.acm.org/research/the-tail-at-scale
47. https://arxiv.org/pdf/2401.09670

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.