GPU SMITH
COMPUTE ENGINEERING

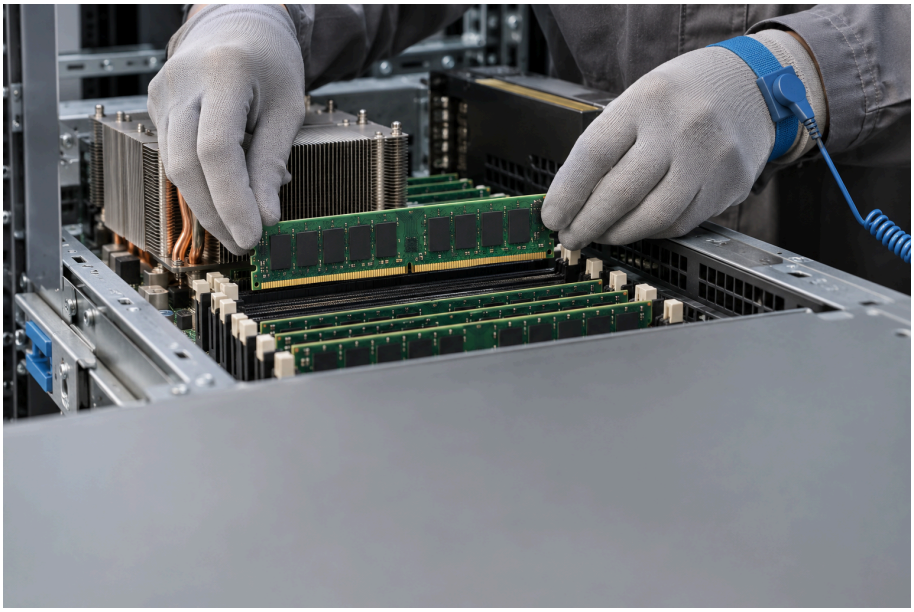
GPU SMITH / RESEARCH & PRACTICAL GUIDANCE

Multi-LoRA Serving Capacity Planning: Working Sets and RAM

> 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 · 2026-11-08 · multi-lora serving capacity planning · vllm lora · max_loras · max_cpu_loras · lora adapter working set · lora adapter eviction · multi-lora p99 latency · lora admission control · lora cold start · lora host ram

Original article: https://gpusmith.com/articles/en/multi-lora-serving-adapter-working-set-planning



In this article

1. Executive Summary
2. Introduction and Background
3. Key Changes in How Multi-LoRA Capacity Is Defined
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

The number of adapters in a registry is not the capacity of a multi-LoRA (low-rank adaptation) serving pool. In **vLLM**, `max_loras` is defined as the maximum number of LoRAs in a single batch, with a default of 1 (docs.vllm.ai), while `max_cpu_loras` caps how many are stored in CPU memory and must be at least `max_loras` (docs.vllm.ai). Registered, host-resident, GPU-resident, active and waiting adapters are five different populations, and each is limited by a different resource. As of November 2026 the latest vLLM release on PyPI is **0.33.0** (pypi.org), and every semantic below should be re-confirmed against the exact build a team pins.

Three sizing errors recur. First, serialized adapter bytes are not runtime bytes: vLLM warns that a `max_lora_rank` much larger than needed wastes memory and can cause performance issues (docs.vllm.ai), and source reading indicates that GPU slot buffers are sized to the rank ceiling rather than to each adapter. For a Llama 3.1 8B adapter on all seven linear modules in bf16, the tensor payload is **83.9 MB at rank 16** and **335.5 MB at rank 64** (derived from the published model configuration (huggingface.co)), and eight slots at a rank ceiling of 64 reserve about **2.68 GB**, versus 10.7 GB at a ceiling of 256 (derived arithmetic). Second, cold loads are not free: published measurements put a host-to-GPU adapter load at a **few to tens of milliseconds** (arxiv.org), one study attributing 17.5% of time to first token (TTFT) to loading a rank-128 adapter in an unloaded system (arxiv.org) and 68% for a rank-32 adapter on Llama-70B with tensor parallelism of 4 (TP4) (arxiv.org). Third, steady-state benchmarks do not bound admission: one public study uses random-weight adapters and counts streamed chunks (github.com), and another explicitly excludes cold starts and eviction (github.com).

The recommended method is a trace-driven ledger that replays real tenant arrivals against the pinned runtime, records the count and bytes of adapters in each state, and scores each tenant against its own latency and correct-output contract. Per-tenant goodput is the share of completed requests that met that contract per unit time, consistent with the goodput framing of **DistServe** (arxiv.org). Admission should reject or defer by queue age, as load-shedding guidance recommends (aws.amazon.com), rather than let a rare tenant wait behind a busy one.

No benchmark was executed for this report, and no universal tenants-per-GPU number is defensible. The deliverable is a sizing sheet, an admission evidence matrix and refresh triggers (changed adapter tensors, rank ceiling, graph settings or tenant workload) that let a platform owner decide whether to retain, partition, admit or scale.

Introduction and Background

Multi-LoRA serving places many fine-tuned variants of one base model behind one serving process. The original LoRA paper already framed the opportunity: a shared pre-trained model can be kept resident while many task-specific modules are swapped in and out on the fly (arxiv.org). It also flagged the difficulty, noting that batching inputs for different tasks is not straightforward once adapter weights are involved (arxiv.org). Systems research then supplied the mechanisms: **Punica** introduced a CUDA kernel design for batching GPU operations across different LoRA models (arxiv.org), and **S-LoRA** stores all adapters in main memory and fetches those used by running queries into GPU memory (arxiv.org).

Those mechanisms now ship in production engines, and the planning question has shifted from "can it run" to "how much can this pool admit." For an enterprise private-inference platform the question is concrete: how many tenants, with what burst behavior, can share an existing pool, which adapters should stay resident, and when traffic should be partitioned or hardware expanded. A registry that lists 500 adapters says nothing about that. The answer depends on which adapters arrive together, how large their real tensors are, how the runtime pads and caches them, and what each tenant's latency contract tolerates.

This report is a serving-admission and churn qualification reference. It is not about training adapters, choosing rank for accuracy, upgrading runtimes or ranking engines. It keeps one runtime, vLLM, as the worked path, and treats other engines only as optional separate profiles. The primary sources (vLLM documentation and source code, Hugging Face PEFT documentation, arXiv and USENIX papers, and two public benchmark repositories) were read in full on November 8, 2026. Where a conclusion rests on reading source code rather than documentation, the text says so. The recurring failure is treating a catalog count as a capacity promise; the remedy is a measured ledger that can be audited.

Two scope limits apply throughout. No local benchmark was run, so every figure here is either quoted from a cited source with that source's scope or derived arithmetically from cited configuration values and labeled as derived. And the older systems papers report speedups against 2023 to 2024 baselines, so their ratios are motivation, not forecasts for current vLLM.

Key Changes in How Multi-LoRA Capacity Is Defined

Change 1: Count five adapter states instead of one registry total

vLLM's own vocabulary separates the populations. A client selects an adapter through the ordinary `model` field, as if it were another model (docs.vllm.ai), so every registered name is addressable. But `max_loras` limits distinct adapters in one batch (docs.vllm.ai), and the scheduler enforces it: a waiting request is held back when admitting it would add a distinct LoRA beyond the limit (raw.githubusercontent.com), and the active set counts the LoRAs of all running requests, including those not scheduled in the current step (raw.githubusercontent.com). Host retention is a different cap again, since `max_cpu_loras` stores adapters in CPU memory (docs.vllm.ai), and when it is unset the source resolves it to `max_loras`.

The practical ledger therefore tracks **registered** (catalog and storage), **host-resident** (retained in the CPU cache), **GPU-resident** (copied into a slot), **active in a batch** (running requests) and **waiting** (queued behind the slot limit). Metrics exist for the last two: `vllm:lora_requests_info` is a gauge of running LoRA statistics (docs.vllm.ai), with a running label holding a per-adapter count of requests (docs.vllm.ai) and a waiting label counting requests still to be scheduled (docs.vllm.ai). The metrics reference also lists a "deferred" waiting reason that includes the LoRA budget (docs.vllm.ai). One caution applies: the exporter's own source warns the metric may be misleading in data-parallel deployments (raw.githubusercontent.com).

Change 2: Size padded slot buffers and host copies, not serialized files

A **PEFT checkpoint** contains only the adapter parameters, not the base model (huggingface.co), but the file is not the allocation. Serialized size follows tensor element counts and dtype, with safetensors requiring that the byte buffer be fully indexed with no holes (raw.githubusercontent.com), and header-only parsing can list

tensors, types and shapes without loading data (huggingface.co). Runtime allocation is a separate quantity. vLLM's `lora_dtype` defaults to the base model dtype (docs.vllm.ai), so an fp32 file may occupy different bytes in memory than on disk. Reading `vllm/lora/layers/base_linear.py` on the main branch indicates that GPU buffers are stacked tensors with a leading dimension of `max_loras` and a rank dimension of `max_lora_rank`, with lower-rank adapters copied into a slice. This is a code-reading inference, not a documented guarantee, and the pinned build's measured peak allocation should override it. The documentation's statement that `max_lora_rank` affects memory allocation and performance (docs.vllm.ai) is consistent with it.

Under tensor parallelism the shape also changes. By default only half of the LoRA computation is sharded, and `fully_sharded_loras` shards the rest, which the configuration notes is likely faster at high sequence length, high rank or high tensor-parallel size (docs.vllm.ai). Per-rank buffer bytes must be measured on each rank, not divided by the tensor-parallel degree.

Change 3: Treat the rank ceiling as a pool-wide cost set by the largest adapter

Because slots are sized to the ceiling, one rank-64 adapter can force every slot to rank-64 size. vLLM's guide gives the same advice in words: set the ceiling to the maximum rank actually used and avoid an unnecessarily high value (docs.vllm.ai). The configuration exposes discrete ceiling values and a default of 16 (docs.vllm.ai). Hardware cost is only part of the effect. **LoRAServe** reports that state-of-the-art systems co-batch heterogeneous adapters without accounting for rank variability (arxiv.org), that padded kernels make smaller adapters pay the computational cost of the largest one (arxiv.org), and that in its experiment co-serving ranks 8 and 128 raised P95 TTFT for rank-8 requests by 84% (arxiv.org). Those are one paper's setups, but they support partitioning high-rank tenants into a separate pool as a first-class option.

PEFT adds another reason to inventory real tensors. Ranks can differ by layer through `rank_pattern` (huggingface.co), `target_modules="all-linear"` selects every linear module and so changes the size relative to a narrower target list (huggingface.co), and DoRA adds a per-layer magnitude parameter (huggingface.co).

Change 4: Plan for cold loads and eviction, not only steady state

The runtime keeps two caches. The worker manager is documented as an LRU (least recently used) cache: every request loads the requested LoRAs, and the least recently used are unloaded if the cache is above capacity (raw.githubusercontent.com). A request for a resident adapter only refreshes its position (raw.githubusercontent.com). A new adapter is validated by loading it first, before eviction (raw.githubusercontent.com), so the host-resident count can briefly exceed the configured cap; the capacity plan must include that transient adapter. Activation then copies the adapter from CPU into a GPU buffer (raw.githubusercontent.com), and if one step needs more distinct adapters than slots the manager raises an error naming the GPU LoRA slots (raw.githubusercontent.com).

Research systems show why churn matters. **CaraServe** states that GPU loading creates a cold start that substantially delays token generation (arxiv.org); **Toppings** adds that under continuous batching it also delays the TTFT of the arriving request (www.usenix.org). **Chameleon** finds that even prefetching raises TTFT under load because it contends for the CPU-GPU PCIe link (arxiv.org), and that when GPU memory is fully used

there is no space left for caching idle adapters (arxiv.org). LRU is also not the only policy: Chameleon argues recency-based policies are inadequate (arxiv.org) and scores adapters by frequency, recency and size (arxiv.org). The LRU manager in vLLM's source is the documented policy, so any popularity or size skew must be handled by sizing and partitioning rather than by policy configuration.

Change 5: Judge admission per tenant, with proof of the correct adapter

Aggregate throughput hides rare-tenant harm. The Tail at Scale makes the general point that temporary latency episodes unimportant at moderate scale can come to dominate at large scale (research.google) and that queueing delay on the server before a request begins is a common source of variability (cacm.acm.org). In adapter serving that queue is partly an adapter queue: a cold tenant arriving beside a busy tenant waits for a slot, a load, or both. The acceptance record therefore needs, for each request, the tenant, the requested adapter name, the adapter revision actually applied, queue age, TTFT, inter-token latency (ITL), completed output tokens and whether the output met the tenant's quality contract. Correct-adapter proof is a data-plane test, not an inference from latency: send canary prompts whose outputs differ measurably per adapter and assert identity on every replay.

Implementation Considerations and Process Changes

Step 1: Write the admission contract

Fix the contract before measuring anything:

- **Pinned tuple:** base model, tokenizer and chat template, runtime version and flags, and approved adapter revisions.
- **Tenant limits:** per-tenant TTFT, ITL, completion rate and quality criteria, each with a percentile and an attainment target.
- **Separation of questions:** inventory capacity (how many adapters can be held) is declared apart from batch eligibility (how many can run together).
- **Traffic basis:** a named tenant trace, with offered, completed and rejected counts reported together.

Benchmarks publish such limits as category definitions; for instance **MLPerf Inference** defines Llama2-70b limits of 2000 ms TTFT and 200 ms TPOT (time per output token) for a conversational category and 450 ms and 40 ms for an interactive one (github.com). Those are not tenant defaults, but they show the form: two latency numbers plus a percentile and an attainment target.

Step 2: Inventory real adapter tensors

For every adapter revision, record the following:

- **Rank by module and target layers,** including any per-layer pattern.
- **Dtype** of each tensor, since the runtime dtype may differ.
- **Embeddings, saved modules and variant tensors,** such as `modules_to_save` entries and DoRA magnitude vectors.
- **Compatibility:** exact base model, tokenizer, chat template, architecture and quantization path.

- **Lineage:** adapter revision hash and the approval that admitted it.

PEFT states that `modules_to_save` lists modules apart from adapter layers that are trainable and saved in the checkpoint (huggingface.co), and trainable token indices save selected token weights alongside the LoRA weights (huggingface.co). These extras can dwarf low-rank tensors. vLLM's guide documents `modules_to_save` support for sequence-classification heads that save a complete single-layer linear classification head (docs.vllm.ai); support for other extra modules was not established by the pages read and must be tested per adapter. The same applies to the exact model, multimodal paths (default multimodal LoRAs allow only one LoRA per prompt (docs.vllm.ai)), mixture-of-experts layouts and quantization. LoRA is applied to all supported modules unless restricted (docs.vllm.ai), and disabling modules to gain speed requires requalifying quality.

Step 3: Measure every residency boundary

Measure on-disk and transfer bytes, host process memory (including pinned memory), GPU slot buffers, graph and workspace overhead, KV (key-value) cache allocation and loading peaks. Several boundaries interact:

- **KV cache competes with adapters.** `gpu_memory_utilization` is a per-instance fraction of GPU memory for the executor (docs.vllm.ai), and vLLM pre-allocates cache using that percentage (docs.vllm.ai). Slot buffers are carved from the same budget, so a larger rank ceiling shrinks KV capacity.
- **Graph specialization costs memory.** `cudagraph_specialize_lora` creates separate graphs for steps with and without active adapters (raw.githubusercontent.com), and the optional `specialize_active_lora` captures graphs for power-of-two active counts at the cost of startup time and memory (docs.vllm.ai).
- **Pinned host memory is finite.** Pinned transfers attain the highest host-to-device bandwidth (docs.nvidia.com), but pinned memory is a scarce resource and excessive use can reduce overall system performance (docs.nvidia.com). Pinning on every call can be up to twice as slow as a plain copy (docs.pytorch.org).
- **Container limits terminate processes.** In cgroup v2, `memory.max` is the main hard limit (docs.kernel.org), and exceeding the Kubernetes limit activates the kernel out-of-memory subsystem (kubernetes.io). Setting a request equal to measured peak matters because the request is mainly a scheduling input (kubernetes.io).
- **Rollout doubles residency.** Replacing an adapter under the same name is supported through in-place reload (docs.vllm.ai); budget old and new copies simultaneously, and test whether a request mid-reload can observe either revision, since the pages read do not establish atomicity.

Step 4: Derive working-set traces

Define the **arrival-window working set** as the count of distinct adapter IDs requested within a stated window. It is a workload descriptor, not automatically `max_loras`. Compute it at several windows: seconds (relevant to slots), minutes (relevant to host retention) and hours (relevant to storage). Record hot-set concentration, burst duration, rank and module mix, and revisit gaps after eviction. Skew is empirically common: LoRAServe reports that in one production region the top five adapters account for more than 70% of requests (arxiv.org).

Replay identical arrival timing against every candidate configuration; **Poisson-style** synthetic load is a legitimate harness, as in MLPerf's server scenario (github.com), but it does not replace the tenant trace. **Little's law** links concurrency and latency (the time-average number in a system equals arrival rate times average time in system) (www.columbia.edu), which bounds distinct in-flight adapters by in-flight requests.

Step 5: Sweep in small controlled steps

Sweep host retention, active slots and permitted rank ceilings one at a time, keeping prompt and output distributions comparable. The case list is:

- **Hot-only:** the concentrated set that fits in slots.
- **Cold-only:** every request names an adapter that must be loaded.
- **Hot-plus-cold:** the rare-tenant-beside-busy-tenant case.
- **Churn:** revisits after eviction at the measured gap.
- **Mixed-rank and long-context:** real rank bands with long prompts competing for KV cache.
- **Reload:** an approved rollout with old and new revisions resident.

One public study illustrates the effect of slot choice but also its limits: under skewed traffic small `max_loras` values caused tail-latency spikes while larger values reduced throughput (github.com), which is exactly why the sweep must record both latency tails and throughput. For counting, use server-reported usage. Metric names differ between tools (vLLM notes terminology is not standardized (docs.vllm.ai), and NVIDIA says to compare results only when definitions align (docs.nvidia.com)), and one streamed output can contain multiple tokens under speculative decoding (docs.vllm.ai). Streamed chunks must never be reported as tokens.

Step 6: Observe admission and queue consequences

Record these on every run, and run the explicit cold-tenant-beside-hot-tenant test:

- **Traffic accounting:** offered, completed, rejected and retried requests.
- **Queue state:** queue age and the running and waiting adapter counts.
- **Adapter churn:** load and eviction counts and failures.
- **Identity:** the adapter revision applied, checked by canary prompts.
- **Service:** per-tenant TTFT, ITL and completion distributions with output tokens counted from server usage.

TTFT includes queuing, prefill and network time (docs.nvidia.com), so adapter waits appear inside it; empty initial responses are ignored by at least one tool (docs.nvidia.com). For rejection policy, load shedding should start rejecting excess requests so the server can focus on those it accepts (aws.amazon.com), an upper bound on queue time is a recommended mechanism (aws.amazon.com), and shed requests should not flatter latency metrics because their latency is extremely low compared with other requests (aws.amazon.com). A backend should accept only what it can process and reject the rest gracefully (sre.google), with criticality-based rejection as utilization nears thresholds (sre.google). For vLLM, vendor-documented goodput options take TTFT, TPOT and end-to-end latency as request-level SLOs (docs.vllm.ai), and DistServe ties attainment to a goal such as 90% per provisioned GPU (arxiv.org).

Step 7: Choose retain, partition, admit or scale

Choose by the binding boundary, not the registry count. If slot waits dominate, raise `max_loras` only if buffer memory and KV headroom allow it. If host misses dominate, raise `max_cpu_loras` within the container limit. If a high-rank minority sets the ceiling, partition it. If SLOs fail for rare tenants, add admission control or pinned residency where the runtime supports it. Compare a shared-base pool against separate replicas only when isolation and identity requirements permit both, using matched hardware and traces. A **dLoRA** result is relevant here: its authors argue that unmerged serving is not always best when requests are skewed (www.usenix.org), a reminder that dedicated or merged capacity can win for a dominant tenant.

Step 8: Publish a release-pinned sheet and refresh triggers

Record the vLLM version and commit, flags, adapter revisions, measured per-state bytes, and the admission matrix. Re-run on any change in adapter tensors, rank ceiling, graph settings, parallelism or tenant workload. The reference build for this report: PyPI listed 0.33.0 on the access date (pypi.org), and the source files were read when the main branch pointed at commit d1879210a4b1, committed November 8, 2026, as shown by the GitHub API (api.github.com). Source-file readings in this report come from that main-branch commit and may differ from the behaviour of the v0.33.0 release, so re-check them against the tagged release you deploy.

Optional separate runtime profiles

Other engines expose similar but not identical controls, and each needs its own qualification. These are documented semantics, not rankings:

- **SGLang** documents GPU pool slots (more slots reserve more GPU memory, default 8 (docs.sglang.ai)), a CPU adapter cap (docs.sglang.ai), an eviction policy when the GPU pool is full (docs.sglang.ai), optional asynchronous loading that overlaps transfers with compute (docs.sglang.ai), and pinned adapters that are not evicted at runtime (docs.sglang.ai).
- **LoRAX** documents just-in-time loading that does not block concurrent requests (loraexchange.ai), and a launcher option whose increase reduces KV cache size (loraexchange.ai).
- **NVIDIA NIM** documents a CPU cache default of 16 adapters in its 1.11.0 configuration reference (the eviction, pre-allocation and HTTP 429 behaviours below come from the 1.0.0 PEFT page and were not re-checked against the current release) (docs.nvidia.com) and a per-batch GPU limit (docs.nvidia.com), evicts the least recently used LoRA from both caches (docs.nvidia.com), pre-allocates both caches (docs.nvidia.com), and returns HTTP 429 when the cache is full (docs.nvidia.com).
- **Triton TensorRT-LLM** documents a host LoRA cache with a 1G default (docs.nvidia.com) and a GPU cache computed from leftover memory (docs.nvidia.com).
- **Managed platforms** describe the same tiers: SageMaker's example holds 30 adapters on GPU and 70 in CPU memory (aws.amazon.com), and Ray Serve LLM keeps a per-replica LRU adapter cache (docs.ray.io).

Control-plane security for dynamic loading

Runtime loading through dedicated endpoints exists (docs.vllm.ai), but the guide says it should not be used in production unless the environment is isolated and fully trusted (docs.vllm.ai), and calls remote-download resolvers insecure for production (docs.vllm.ai). Treat loading and unloading as an authenticated control-plane action tied to approved revisions, never as a public endpoint, and log the revision hash applied to each request.

Data Analysis and Evidence

Adapter-state transition ledger

Table 1 below is the central worksheet. Each row is a state; each cell is something to record rather than assume.

State	Controlled by	Record at this state	Cost of moving to the next state
Registered (storage and catalog)	Control plane; approved revisions	Adapter ID, revision hash, rank by module, dtype, serialized bytes, extra tensors	Read plus validation; the validated load happens before any eviction (raw.githubusercontent.com)
Host-resident	`max_cpu_loras` (docs.vllm.ai)	Host process bytes per adapter, pinned bytes, headroom to container limit, hit rate	Copy into a GPU slot (raw.githubusercontent.com); measure ms and PCIe contention
GPU-resident	`max_loras` slots and rank ceiling	Measured buffer bytes per rank, KV capacity lost, graph overhead	Evicted by LRU when slots are full (raw.githubusercontent.com); reload cost returns
Active in a batch	Scheduler and running requests (raw.githubusercontent.com)	Distinct adapters per step, per-adapter running count (docs.vllm.ai)	Request completion releases the adapter for eviction
Waiting	Slot budget; "deferred" reason	Queue age, waiting count, rejects and retries	Admission when a slot frees; or rejection by queue-age bound (aws.amazon.com)
Evicted then reloaded	Churn and revisit gap	Reload count, added TTFT, tail percentile for the revisiting tenant	Same as host-to-GPU copy, plus disk read if host-evicted

The interpretive point is that every transition has two outputs, a latency and a memory delta, and the ledger needs both. Pinning is the explicit exception to eviction: vLLM's model manager supports pinning only in its LRU implementation (raw.githubusercontent.com), and SGLang documents pinned adapters that are not evicted at runtime (docs.sglang.ai). Pinned adapters consume slots permanently, so pin only the adapters of tenants whose contract requires it.

Worked size arithmetic for a Llama 3.1 8B adapter

The model dimensions come from the published configuration: hidden size 4096 (huggingface.co), intermediate size 14336 (huggingface.co), 32 layers (huggingface.co), 8 key-value heads (huggingface.co) with head dimension 128 (huggingface.co), so the key and value projections map 4096 to 1024. For ordinary

unquantized linear LoRA, a module costs r x (d_in + d_out) elements. Summed across the seven projections, one layer costs 81,920 x r elements, and 32 layers cost 2,621,440 x r elements. This is derived arithmetic, not a measurement, and actual files override it.

Table 2 below converts the element counts to bf16 bytes (2 bytes per element) and adds an ideal host-to-GPU transfer time at the per-direction PCIe figures quoted by an analyst in the PCI-SIG Gen5 announcement: 64 GB/s for Gen5 x16 (us.design-reuse.com) and 32 GB/s for Gen4 x16 (us.design-reuse.com). Real pageable-memory copies are slower than these ceilings.

Adapter shape (Llama 3.1 8B, bf16)	Elements	Tensor bytes	Ideal transfer, Gen5 / Gen4
All 7 modules, rank 8	20,971,520	**41.9 MB**	0.65 ms / 1.31 ms
All 7 modules, rank 16	41,943,040	**83.9 MB**	1.31 ms / 2.62 ms
All 7 modules, rank 64	167,772,160	**335.5 MB**	5.24 ms / 10.49 ms
Attention only, rank 8	6,815,744	13.6 MB	0.21 ms / 0.43 ms
Attention only, rank 16	13,631,488	27.3 MB	0.43 ms / 0.85 ms
Attention only, rank 64	54,525,952	109.1 MB	1.70 ms / 3.41 ms
Add one full `embed_tokens` or `lm_head` via `modules_to_save` (vocabulary 128256 (huggingface.co), untied embeddings (huggingface.co))	525,336,576	**1.05 GB each**	16.4 ms / 32.8 ms

The table shows three planning facts. Rank scales bytes linearly, so rank 64 costs eight times rank 8. Module selection moves bytes by a factor of about three at equal rank. And a single saved embedding matrix is larger than twelve rank-16 all-module adapters, which is why Step 2 inventories extras. The ideal times (about 1 to 5 ms for all-module adapters) are consistent in order of magnitude with Punica's statement that loading an entire rank-16 model over PCIe Gen4 takes about 2 ms (arxiv.org) and with CaraServe's range of a few to tens of milliseconds (arxiv.org), but they exclude queueing, contention and host-side staging. Measured cold-load effects are larger in relative terms: Chameleon attributes 17.5% of TTFT to loading a rank-128 adapter in an unloaded system (arxiv.org) and 68% for a rank-32 adapter on Llama-70B with TP4 (arxiv.org).

Slot buffers are a separate calculation. Under the code-reading inference above, GPU slot memory is about max_loras x max_lora_rank x 2,621,440 x 2 bytes for this model. Eight slots at ceiling 16 reserve about **0.67 GB**; at ceiling 64, about 2.68 GB; at ceiling 256 (the kind of oversized value the guide warns against (docs.vllm.ai)), about **10.74 GB**. Measure the real value from peak allocation before and after enabling LoRA.

Hypothetical admission calculation (Hypothetical Example)

Suppose a pool serves Llama 3.1 8B with 120 registered adapters, 90% at rank 16 and 10% at rank 64, all on seven modules in bf16. A trace shows p99 distinct adapters in flight of 9, and p99 distinct adapters per 10 minutes of 70. These numbers are invented for illustration. A planner would choose `max_loras` 16 (above the 9 observed, leaving burst margin) and `max_cpu_loras` 80. If the rank-64 minority sets the ceiling at 64, slot buffers reserve 16 x 64 x 2,621,440 x 2 B = **5.37 GB**; with the minority partitioned away and the ceiling at 16, they reserve **1.34 GB**, freeing **4.03 GB** for KV cache. Host retention for 80 adapters at the mixed ranks is 72 x 83.9 MB + 8 x 335.5 MB = **8.7 GB** plus one transient adapter, to be compared with measured host bytes and the container limit. Whether the partition is worth a second pool depends on measured p99 TTFT for the rank-64 tenants, not on this arithmetic.

What existing evidence does and does not establish

Table 3 below summarizes the public evidence by scope. It is intended to show which questions each source can answer for an admission decision, without ranking any system.

Source	Setup as stated	Useful for	Does not establish
Public multi-LoRA benchmark repository (commit 577929f (github.com))	1,000 synthetic rank-16 adapters, Llama 3.1 8B fp16 (github.com); random weights (github.com)	Working-set framing (github.com); host process memory grew about 13 GB while GPU grew about 25 MB (github.com); 63 throughput runs (github.com)	Quality (github.com); token counts (it counts streamed chunks (github.com)); per-request latency (raw files not kept (github.com))
Controlled-adapter benchmark (commit 47f5f84 (github.com))	Paired runs in one container on one GPU (github.com); zero-initialized B matrices (github.com)	Engine overhead at stated ranks	Tensor parallelism, cold starts, eviction (github.com); transfer across ranks (github.com)
S-LoRA paper	Host-resident adapters fetched on demand; unified paging of adapters and KV (arxiv.org)	Design rationale; prefetch of next-batch adapters (arxiv.org); one-GPU hardware (arxiv.org)	2026 vLLM behavior (latest arXiv version revised June 2024 (arxiv.org))
Punica	Batched kernel across adapters; rank 16 only (arxiv.org)	Order of magnitude of load time	Other ranks; tenant SLOs
CaraServe and Toppings	Cold-start focus (www.usenix.org)	Cold-start share of serving time (10%, 16%, 20% at rising load (arxiv.org))	Current vLLM defaults
Chameleon and LoRAServe	Caching (arxiv.org); rank-aware placement	Cache and rank effects on P99/P95 TTFT (arxiv.org)	A transferable threshold for a different model or trace
dLoRA	Merged versus unmerged serving (www.usenix.org)	Skew-aware mode choice	Tail metrics; it used average latency (www.usenix.org) and rank 8 (www.usenix.org)

The pattern is consistent: each source isolates a mechanism, and none supplies the combination an admission decision needs, which is real adapters, finite host residency, mixed ranks and per-tenant tail latency under churn. The public steady-state studies are useful scaffolding. For example, the first repository reports that uniform-traffic throughput fell from 1,935 to 1,489 units per second between one and ten adapters (github.com) and that skewed traffic reached about 2.45 times the uniform result at 1,000 adapters (github.com), but its units are streamed chunks per second, so those figures describe relative behavior only. The same working-set conclusion appears in the research: S-LoRA claims thousands of adapters on one GPU (arxiv.org) in its setup, which is a statement about registry scale under a bounded active set, not about concurrent tenants. A related cold-path caveat comes from serverless work, which reports that running LoRA inference on serverless platforms introduces high memory overhead and repeated cold starts (arxiv.org); container and process start-up are separate from adapter load and need their own rows.

Implications and Future Directions

For capacity planners, the first implication is that quoting "adapters per GPU" is a category error. Three different numbers exist (registered, host-retained, simultaneously active), and the binding one varies with workload. A pool with a flat popularity distribution will hit the slot limit first; a pool with a heavy tail and a hot set will hit host-miss latency for the cold tenants first; a pool with one large-rank minority will pay a memory and latency tax on every tenant. Procurement conversations should ask for the ledger, not a count.

For architecture, partitioning is likely to become routine. Research findings about rank variability and skew (www.usenix.org) point in the same direction as vLLM's own guidance on ceilings: group tenants by rank band, latency class and popularity, and size each group separately. Where replicas exist, affinity of tenant to replica raises hit rates, but that routing decision belongs to a separate design; this report only requires that the ledger be measured per replica.

For operations, the main change is treating adapter revisions like software releases. Every rollout creates a transient doubling of residency and an identity question, and runtime flags that look like tuning (graph specialization (raw.githubusercontent.com), rank ceiling, dtype) change memory. A refresh trigger list is part of the deliverable, not an appendix.

For tooling, the gap is observability. The runtime exports running and waiting adapter labels but per-tenant latency, correct-adapter proof and eviction counts per adapter must come from the client harness or additional instrumentation. Metric definitions also differ across tools, and some count tokens client-side with a tokenizer (docs.nvidia.com), so the harness should log server-reported token counts.

Finally, expect semantics to move. The LoRA guide already covers in-place reload and runtime loading through plugins and endpoints. Each addition changes what must be qualified. The method is stable; the constants are not.

Frequently Asked Questions (FAQs)

How many LoRA adapters per GPU?

No universal number is supported by the evidence reviewed. Registered capacity can be large when adapters sit in host memory or storage (S-LoRA reports thousands in its setup), but concurrent capacity is limited by `max_loras` and by the distinct adapters your traffic puts in a batch. Measure the working set at several windows and size slots to the in-flight p99, then verify with a replay.

What are the multi-LoRA host RAM requirements?

Host memory must hold the retained adapters, the process baseline, any pinned buffers and a transient extra adapter during loads (raw.githubusercontent.com), within the container limit (docs.kernel.org). One public benchmark observed about 13 GB of process memory growth for 1,000 loaded rank-16 adapters (github.com), which is a measured data point for that adapter set, not a rule. It works out to roughly 13 MB per adapter, far below the 83.9 MB derived above for a rank-16 all-module bf16 adapter, so the benchmark's target modules, dtype or measurement method differ and the figure is not comparable to that derivation. Compute host-retained bytes from your actual tensors, then add a reserve you choose and record in the ledger, for example a fixed percentage of headroom on the measured peak.

What is the LoRA adapter working set size?

It is the count of distinct adapter IDs requested within a stated window. It describes the workload; it does not automatically equal `max_loras`. Short windows inform slot sizing and long windows inform host retention.

Which cache eviction policy applies to LoRA adapters?

vLLM's documented worker manager uses LRU (raw.githubusercontent.com), as do NIM, per its 1.0.0 documentation (docs.nvidia.com) and Ray Serve LLM (docs.ray.io); SGLang documents a configurable policy (docs.sglang.ai). Research proposes cost-aware scores that weigh frequency, recency and size and only evict adapters that no running request uses (arxiv.org). Verify which manager your pinned build instantiates.

How does adapter swapping affect p99 latency?

A miss adds a load to the arriving request's TTFT and can delay in-flight decoding under continuous batching (www.usenix.org). Published shares range from 17.5% of TTFT (rank-128 adapter, unloaded system) to 68% (rank-32 adapter, Llama-70B, TP4). Prefetch helps but contends for the PCIe link.

What do vLLM `max_loras` and `max_cpu_loras` do?

`max_loras` is adapters in a single batch (docs.vllm.ai); `max_cpu_loras` is the CPU-memory store (the host-offload tier) and must be at least `max_loras` (docs.vllm.ai). Adapters beyond the host cap are evicted and must be re-read from storage. `max_lora_rank` sets the ceiling used for allocation (docs.vllm.ai).

How should admission control work?

Defer when a new distinct adapter would exceed the slot budget (the scheduler skips such requests (raw.githubusercontent.com)), reject when queue age exceeds a bound (aws.amazon.com), and score tenants separately so one busy tenant cannot starve a rare one.

How long does a cold start or hot swap take?

Ideal PCIe transfer for an 84 MB adapter is on the order of 1 to 3 ms (Table 2), measured host-to-GPU loads are reported as a few to tens of milliseconds, and a first request to an unloaded adapter may include a loading time, per NIM 1.0.0 documentation (docs.nvidia.com). One managed-service guide suggests sending a one-token dummy request to pre-load adapters (docs.anyscale.com). Measure end-to-end TTFT for the cold tenant on your hardware.

Conclusion

A multi-LoRA pool is admitted by its binding boundary, not by its catalog. The boundary may be GPU slots (`max_loras`), host retention (`max_cpu_loras` and the container limit), padded buffer memory set by the rank ceiling, KV headroom, or a tenant's tail-latency contract during churn. Documentation and source reading show these are separate controls with separate failure signatures, and the published research shows that cold loads, rank mixes and skew change tail latency in ways steady-state studies do not capture.

The recommended artifact is a release-pinned sizing sheet with five parts: the adapter inventory by real tensors, the state-transition ledger with latency and memory at every transition, the working-set trace summary, the per-tenant admission evidence matrix, and the refresh triggers. A buyer should accept a capacity claim only when it names the runtime build, the adapter revisions, the trace, and the per-tenant percentile results with output tokens counted from server usage. As of November 2026, no public source supplies a transferable tenants-per-GPU figure, and this report does not offer one. It offers the procedure for producing the figure that applies to a given pool.

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/latest/api/vllm/config/lora
2. https://pypi.org/project/vllm
3. https://docs.vllm.ai/en/latest/features/lora
4. https://huggingface.co/unsloth/Llama-3.1-8B/raw/main/config.json
5. https://arxiv.org/pdf/2401.11240
6. https://arxiv.org/pdf/2411.17741
7. https://github.com/priyaltaneja/multi-lora-serving-benchmark
8. https://github.com/BAL-Analytics/multi-lora-benchmark
9. https://arxiv.org/abs/2401.09670
10. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload

11. https://arxiv.org/html/2106.09685v2
12. https://arxiv.org/abs/2310.18547
13. https://arxiv.org/abs/2311.03285
14. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/v1/core/sched/scheduler.py
15. https://docs.vllm.ai/en/latest/usage/metrics
16. https://docs.vllm.ai/en/latest/design/metrics
17. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/v1/metrics/loggers.py
18. https://huggingface.co/docs/peft/en/developer_guides/checkpoint
19. https://raw.githubusercontent.com/huggingface/safetensors/main/README.md
20. https://huggingface.co/docs/safetensors/metadata_parsing
21. https://docs.vllm.ai/en/latest/configuration/engine_args
22. https://arxiv.org/abs/2511.22880
23. https://arxiv.org/pdf/2511.22880
24. https://huggingface.co/docs/peft/en/package_reference/lora
25. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/lora/worker_manager.py
26. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/lora/model_manager.py
27. https://arxiv.org/abs/2401.11240
28. https://www.usenix.org/conference/atc25/presentation/li-suyi-toppings
29. https://research.google/pubs/pub40801
30. https://cacm.acm.org/research/the-tail-at-scale
31. https://github.com/mlcommons/inference_policies/blob/master/inference_rules.adoc
32. https://docs.vllm.ai/en/latest/configuration/optimization
33. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/config/compilation.py
34. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
35. https://docs.pytorch.org/tutorials/intermediate/pinmem_nonblock.html
36. https://docs.kernel.org/admin-guide/cgroup-v2.html
37. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers
38. http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-LL.pdf
39. https://docs.vllm.ai/en/latest/benchmarking/cli
40. https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html
41. https://sre.google/sre-book/handling-overload
42. https://docs.vllm.ai/en/latest/cli/bench/serve
43. https://arxiv.org/pdf/2401.09670
44. https://www.usenix.org/conference/osdi24/presentation/wu-bingyang
45. https://api.github.com/repos/vllm-project/vllm/commits/main
46. https://docs.sglang.ai/advanced_features/lora.html
47. https://loraexchange.ai

48. https://loraexchange.ai/reference/launcher
49. https://docs.nvidia.com/nim/large-language-models/1.11.0/configuration.html
50. https://docs.nvidia.com/nim/large-language-models/1.0.0/peft.html
51. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/tensorrtllm_backend/docs/lora.html
52. https://aws.amazon.com/blogs/machine-learning/easily-deploy-and-manage-hundreds-of-lora-adapters-with-sagemaker-efficient-multi-adapter-inference
53. https://docs.ray.io/en/latest/serve/llm/user-guides/multi-lora.html
54. https://us.design-reuse.com/news/46222/32gt-s-pci-express-5-0-specification.html
55. https://arxiv.org/pdf/2310.18547
56. https://github.com/priyaltaneja/multi-lora-serving-benchmark/commits/main
57. https://github.com/BAL-Analytics/multi-lora-benchmark/commits/main
58. https://arxiv.org/pdf/2311.03285
59. https://arxiv.org/abs/2411.17741
60. https://www.usenix.org/system/files/osdi24-wu-bingyang.pdf
61. https://arxiv.org/abs/2505.14468
62. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/perf_analyzer/genai-perf/README.html
63. https://docs.anyscale.com/llm/serving/multi-lora

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.