

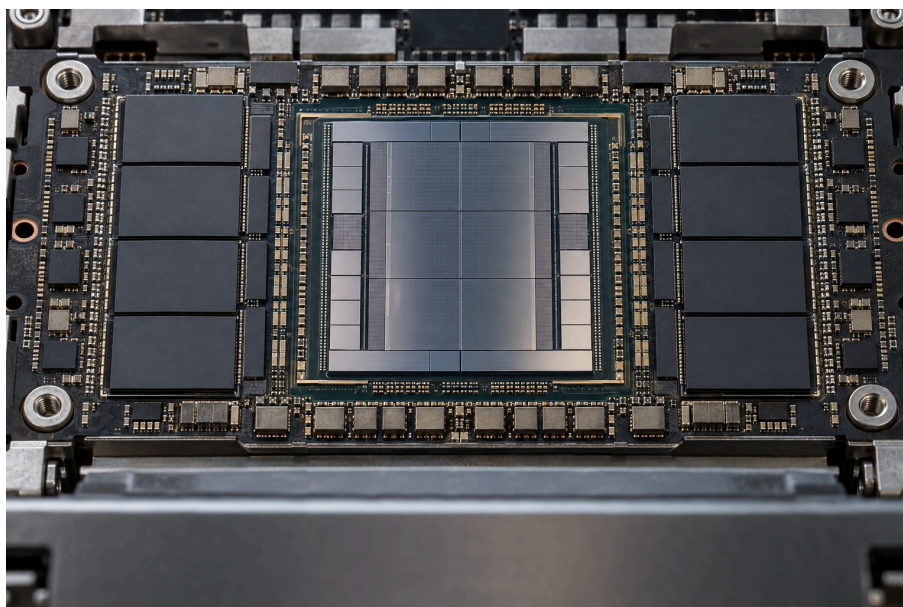


Chunked Prefill Tuning: Token Budget, TTFT and ITL

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 · chunked prefill vllm · max_num_batched_tokens tuning · chunked prefill ttft itl tradeoff · sglang chunked-prefill-size · sarathi-serve chunked prefill · ttft llm

Original article: <https://gpusmith.com/articles/en/chunked-prefill-tuning-itl-ttft-tradeoff>



In this article

1. Why prefill and decode interfere
 2. What chunked prefill does
 3. The knob: `max_num_batched_tokens` in vLLM
 4. SGLang equivalents
 5. Reading the effect: a worked example
 6. A tuning protocol
 7. Workload shapes
 8. Interaction with batch size and KV capacity
 9. When chunking is not enough
 10. Rollout and regression gates
 11. Checklist
 12. Sources
-

Why prefill and decode interfere

Each iteration of a continuous-batching LLM server runs one forward pass over whatever the scheduler admitted. Prefill processes a whole prompt at once and is compute-bound. Decode produces one token per running sequence per step, reading weights and the growing KV cache for very little compute, so it is bandwidth-bound. In the Sarathi-Serve authors' terms, prefill has high latency and saturates GPU compute, while decode has low latency per step and poor compute utilization ([Agrawal et al., arXiv 2403.02310](#)).

Problems start when both kinds of work share a GPU. Suppose a 30,000-token RAG prompt arrives while 100 chat sessions are streaming. If the scheduler runs the whole prompt in one iteration, every one of those 100 streams waits for it. The paper calls this a *generation stall*: "one or more prefills are scheduled in between consecutive decode iterations of a request" ([Sarathi-Serve, HTML v3](#)). Users see it as an inter-token latency (ITL) spike whose tail can reach seconds while the median looks fine. DistServe reaches the same diagnosis: colocating the phases causes "strong prefill-decoding interferences" and forces a system to favour either time to first token (TTFT) or time per output token (TPOT) when SLOs are tight ([Zhong et al., arXiv 2401.09670](#)).

If you need precise definitions of TTFT, ITL, TPOT and goodput before tuning, start with [TTFT, ITL and goodput definitions](#).

What chunked prefill does

Chunked prefill caps tokens per iteration: the scheduler admits running decodes (one token each), then fills the rest of a fixed budget with prompt tokens, splitting long prompts across iterations. Sarathi-Serve calls these "chunked-prefills" combined with "stall-free scheduling". New requests join a running batch without pausing the decodes, and the near-uniform iteration size also reduces pipeline bubbles under pipeline parallelism ([arXiv 2403.02310](#)).

The paper's headline results are 2.6x higher serving capacity for Mistral-7B on one A100 and up to 3.7x for Yi-34B on two A100s, both against vLLM. It also reports up to 5.6x end-to-end serving capacity for Falcon-180B with pipeline parallelism, all under tail-latency constraints ([arXiv 2403.02310](#); published at [USENIX OSDI '24](#)). These compare against 2024 vLLM, which now ships chunked prefill itself; they do not describe today's gap.

Chunking costs something. Each chunk attends over the KV cache of all earlier chunks, so splitting a prompt raises HBM reads "even though the computational cost is unchanged". The authors measured at most about 25% prefill overhead at a 512-token chunk on Yi-34B and almost none at a 2048-token budget. Chunk alignment also matters: a 257-token chunk can take 32% longer than a 256-token chunk because of tile quantization ([Sarathi-Serve, HTML v3](#)). Use multiples of 256, and go small only if the ITL SLO requires it. The paper itself used 512 for strict SLOs and 2048 for relaxed ones ([same](#)).

The knob: `max_num_batched_tokens` in vLLM

In vLLM V1, "chunked prefill is enabled by default whenever possible". With it enabled, "the scheduling policy prioritizes decode requests": the scheduler batches all pending decodes first. A prefill that does not fit into `max_num_batched_tokens` is chunked automatically ([vLLM Optimization and Tuning](#)). That makes `--max-num-batched-tokens` the per-iteration token budget **B**. It covers decode tokens and prefill tokens together.

The documentation states the trade-off directly:

- Smaller values, such as 2048, "achieve better ITL because there are fewer prefills slowing down decodes".
- Higher values "achieve better time to first token (TTFT) as you can process more prefill tokens in a batch".
- For throughput, vLLM recommends `max_num_batched_tokens > 8192`, "especially for smaller models on large GPUs".
- Setting it equal to `max_model_len` is close to the old V0 policy, except that decodes are still prioritized ([vLLM Optimization and Tuning](#)).

Defaults depend on hardware. The docs page gives no single default because there isn't one. In the v0.32.0 source, `EngineArgs.get_batch_defaults` picks the budget from the device's total memory and from whether you launched the OpenAI server (`vllm serve`) or the offline LLM class (`vllm/engine/arg_utils.py @ v0.32.0`). The same function is unchanged at v0.29.0:

Device memory (vLLM check)	vllm serve default B	Offline LLM default B	Default max_num_seqs (both)
≥ 160 GiB (code comment: B200/B300)	16384	16384	1024
≥ 70 GiB and not an A100 (comment: H100/H200)	8192	16384	1024
Other GPUs, including A100 80 GB	2048	8192	256

A code comment explains the A100 exclusion: a large budget reduced A100 throughput ([same file](#)). The consequence is that the same `vllm serve` command starts H100 and A100 servers at different points on the TTFT/ITL curve, and a config copied across GPU generations without an explicit `--max-num-batched-tokens`

silently changes B.

Related scheduler fields in v0.32.0 ([vllm/config/scheduler.py @ v0.32.0](#)):

- `--long-prefill-token-threshold` (default 0, meaning off) marks a prompt as "long" above the threshold so its per-step share can be capped. The cap is skipped when the request is alone in the batch.
- `--max-num-queued-tokens` rejects new requests with HTTP 503 once the prompt tokens waiting in prefill exceed a limit. The docstring suggests sizing it as $\text{target_TTFT} \times \text{prefill_throughput}$.
- `scheduler_reserve_full_is1` (default True) checks that the full input length fits in the KV cache before admitting a request, "rather than only checking the first chunk". This prevents over-admission and KV thrashing under chunked prefill.
- `max_num_batched_tokens` must be at least `max_num_seqs`. Otherwise config validation raises an error.

SGLang equivalents

SGLang splits the same concept across several flags ([SGLang server arguments](#)):

Concept	vLLM (v0.32.0)	SGLang (v0.5.22)
Per-step prefill chunk cap	<code>--max-num-batched-tokens</code> (shared with decodes)	<code>--chunked-prefill-size</code> (-1 disables)
Cap on a prefill batch	same budget	<code>--max-prefill-tokens</code> , default 16384
Concurrency cap	<code>--max-num-seqs</code>	<code>--max-running-requests</code>
Mix prefill and decode in one forward pass	always, decodes first	<code>--enable-mixed-chunk</code> , default False
Back-pressure on retractions	preemption (default RECOMPUTE)	<code>--schedule-conservativeness</code> , default 1.0

The documented default of None is resolved at startup: in v0.5.22, `handle_gpu_memory_settings` sets it from GPU memory: 2048 below 20 GB (comment: T4, 4080); 4096 below 35 GB (A10, 4090, 5090) and below 60 GB (A100 40GB, L40); 8192 below 90 GB (H100, A100) and below 160 GB (H20, H200); 16384 above that (B200, MI300) ([memory_hook.py @ v0.5.22](#)). The same function then reserves activation memory in proportion to the chunk size before it computes `mem_fraction_static`. Raising `--chunked-prefill-size` by hand therefore also shrinks the KV pool. For OOM during long prefills they recommend a *smaller* chunk size ([server arguments](#)).

The most important difference from vLLM is `--enable-mixed-chunk`. Its help text reads "Enabling mixing prefill and decode in a batch when using chunked prefill", and it defaults to False ([fields/schedule.py @ v0.5.22](#)). With it off, the v0.5.22 scheduler runs a pending prefill batch first ("Run prefill first if possible") and decodes only on steps with no new prefill, so every chunk step pauses all running streams for a full prefill step ([scheduler.py @ v0.5.22](#)). Equal chunk sizes therefore do not imply equal ITL cost across engines. When porting a tuned budget from one engine to the other, re-run the sweep rather than copying the number.

Reading the effect: a worked example

A first-order model predicts the curves before you spend GPU hours. **All inputs below are hypothetical**; replace them with your own measurements.

Assume one replica with 64 sequences decoding. A decode-only step takes **15 ms**, and each extra prefill token adds **0.025 ms** to the step (a marginal prefill rate of 40,000 tokens/s). Decodes use 64 of the B tokens, so each step carries B - 64 prefill tokens.

Step time when the budget is full: $t = 15 \text{ ms} + (B - 64) \times 0.025 \text{ ms}$

B	Prefill tokens/step	Full-step time (= ITL during prefill)	Prefill capacity
1024	960	15 + 24.0 = 39.0 ms	960 / 0.0390 s = 24,615 tok/s
2048	1,984	15 + 49.6 = 64.6 ms	1,984 / 0.0646 s = 30,712 tok/s
4096	4,032	15 + 100.8 = 115.8 ms	4,032 / 0.1158 s = 34,819 tok/s
8192	8,128	15 + 203.2 = 218.2 ms	8,128 / 0.2182 s = 37,250 tok/s

Service-time TTFT for one 8,000-token prompt with an empty queue:

- B = 1024: 8 full chunks of 960 (7,680 tokens) plus a 320-token tail. That is $8 \times 39.0 + (15 + 8.0) = 335$ ms.
- B = 2048: $4 \times 1,984 = 7,936$ tokens plus a 64-token tail. That is $4 \times 64.6 + (15 + 1.6) = 275$ ms.
- B = 4096: $4,032 + 3,968$ tokens = $115.8 + (15 + 99.2) = 230$ ms.
- B = 8192: one step = $15 + 8,000 \times 0.025 = 215$ ms.

On an idle server the TTFT gap is modest (335 vs 215 ms); the ITL cost to everyone else is not (39 vs 218 ms per affected step).

Under load, capacity decides TTFT. Now add 4 requests/s with 6,000-token prompts, an offered prefill load of $4 \times 6,000 = 24,000$ tok/s. Prefill utilization is offered load divided by capacity:

B	Utilization	Reading
1024	24,000 / 24,615 = 97.5%	Queue grows without bound in bursts; TTFT p99 fails
2048	24,000 / 30,712 = 78.1%	Queueing visible but bounded
4096	24,000 / 34,819 = 68.9%	Lower TTFT; ITL during prefill 115.8 ms
8192	24,000 / 37,250 = 64.4%	Lowest TTFT; ITL spikes 218.2 ms

Large budgets raise prefill capacity by spreading the fixed 15 ms decode cost over more prompt tokens, which is why vLLM recommends $B > 8192$ for throughput. Under an ITL p99 SLO of 80 ms, only B = 2048 satisfies both constraints here. Under 50 ms, only B = 1024 meets ITL, and it runs prefill at 97.5% utilization. No setting meets both, and that is the signal that chunking alone is not enough (see below).

The model omits growth of decode time with batch and context, the rising cost of later chunks (each attends over all earlier KV), chunking overhead at small budgets, and bursty arrivals. Use it to choose the sweep range, never the final value.

A tuning protocol

1. **Freeze everything except B.** Fix the model and quantization, engine version, tensor/pipeline parallel sizes, max_model_len, gpu_memory_utilization, prefix-caching setting and max_num_seqs.
2. **Build a representative trace.** Use production prompt/output length distributions, not one fixed length. `vllm bench serve` supports `--request-rate` with Poisson arrivals and a `--burstiness` factor, where values below 1 make arrivals burstier ([vllm/benchmarks/serve.py @ v0.32.0](#)).
3. **Sweep B over multiples of 256:** 1024, 2048, 4096, 8192, 16384. Run each value at three or more request rates spanning 50-95% of the knee you expect from the arithmetic above.
4. **Report percentiles and goodput together.** For example:

```
vllm serve $MODEL --max-num-batched-tokens 2048 --max-num-seqs 256
vllm bench serve --model $MODEL --dataset-name random \
  --random-input-len 6000 --random-output-len 300 --num-prompts 2000 \
  --request-rate 4 --burstiness 1.0 \
  --percentile-metrics ttft,tpot,itl --metric-percentiles 50,90,99 \
  --goodput ttft:2000 tpot:50 --save-result
```

`--goodput` accepts only request-level keys (ttft, tpot, e2e1) in milliseconds, following DistServe's definition ([serve.py @ v0.32.0](#)). ITL percentiles therefore come from `--percentile-metrics`, while TPOT stands in for ITL in the goodput count. The default is p99 only.

5. **Scrape server metrics during each run** ([vllm/v1/metrics/loggers.py @ v0.32.0](#)). Counters are exposed with a `_total` suffix.
 - `vllm:iteration_tokens_total`: histogram of tokens per engine step. If most steps sit at B, prefill is saturated.
 - `vllm:num_preemptions` and `vllm:kv_cache_usage_perc`: KV pressure.
 - `vllm:num_requests_waiting`: queue depth.
 - `vllm:request_queue_time_seconds` and `vllm:request_prefill_time_seconds`: split TTFT into waiting and computing.
6. **Pick by constraint, then by goodput.** Discard any B whose ITL p99 misses the SLO at the target rate. Among the rest, take the highest goodput. Within noise, prefer the smaller B for burst headroom.

Workload shapes

Workload	Starting point	Watch
Interactive chat (short prompts, long streams)	1024-2048	ITL p99; decodes eat budget at high concurrency

Workload	Starting point	Watch
Long-prompt RAG (long prompts, short answers)	4096-8192+	TTFT p99, prefill utilization
Agent loops (shared long prefix)	Mid-range, prefix caching on	Hit rate; only the uncached suffix costs
Offline batch	≥ 8192 per vLLM guidance	Tokens/s only

Prefix caching changes the arithmetic. vLLM's automatic prefix caching lets a new query "skip the computation of the shared part", and it "only reduces the time of processing the queries (the prefilling phase)" ([vLLM APC docs](#)). With high hit rates, offered prefill load is only the uncached tokens, so measure the hit rate ($\text{vllm:prefix_cache_hits} / \text{vllm:prefix_cache_queries}$) before you size B from raw prompt lengths.

Interaction with batch size and KV capacity

B, max_num_seqs and KV capacity form one system:

- **Decodes consume budget.** With 256 running sequences and $B = 1024$, only 768 tokens per step are left for prefill. Raising max_num_seqs without raising B quietly cuts prefill capacity and raises TTFT.
- **KV pressure causes preemption.** When the KV cache fills, vLLM preempts requests. In V1 the default mode is RECOMPUTE, and "preemption and recomputation can adversely affect end-to-end latency". The documented remedies are to raise gpu_memory_utilization, add tensor parallelism, or lower max_num_seqs or max_num_batched_tokens ([vLLM Optimization and Tuning](#)). A recomputed request re-enters prefill, so preemption storms show up as TTFT *and* ITL regressions.
- **Larger B costs activation memory.** SGLang makes this explicit by reserving memory in proportion to chunk size before it sizes the KV pool ([memory_hook.py](#)). In vLLM, compare the KV capacity reported in the startup log across budgets rather than assuming it is constant.

PagedAttention's block-level KV management is what makes this fine-grained admission practical. The original paper reported near-zero KV waste and 2-4x throughput over FasterTransformer and Orca in 2023 ([Kwon et al., arXiv 2309.06180](#)). To size the KV pool for a target concurrency, use the [vLLM KV cache capacity planner](#).

When chunking is not enough

Chunked prefill moves work along one dial; it cannot create capacity. When the ITL SLO forces a B whose prefill capacity is close to offered load, no budget satisfies both SLOs. In a sweep, the signs are:

- every B that meets ITL p99 runs iteration_tokens near the budget and shows a rising num_requests_waiting;
- TTFT p99 is dominated by queue time rather than prefill time;
- the TTFT and ITL Pareto frontier across your sweep sits entirely outside the SLO box.

At that point you have three options: add colocated replicas, accept a looser SLO, or separate the phases. vLLM's disaggregated prefilling is documented as "experimental and subject to change". It lets you tune TTFT and ITL separately and stops vLLM inserting "prefill jobs during the decoding of one request". It also states that

disaggregation "DOES NOT improve throughput" (vLLM disaggregated prefilling). The economics, including the KV transfer cost and the second minimum pool, are covered in [when to disaggregate prefill and decode](#).

Rollout and regression gates

- **Canary one replica** with the new B and compare TTFT p99, ITL p99 and preemption rate against control replicas over at least one daily peak. Roll back if either percentile regresses beyond a tolerance set before the test.
- **Alert on the mechanism, not just the SLO.** A sustained shift of the `iteration_tokens` histogram toward B, or a rising preemption rate, leads SLO breaches.
- **Pin B explicitly** in every manifest. Both engines derive defaults from GPU memory, so a node-pool change can alter B without a config diff.
- **Re-run the sweep on engine upgrades.** Scheduler fields change between releases (`--long-prefill-token-threshold-adaptive` and `--max-num-active-seqs` exist in v0.32.0 but not v0.29.0).

Checklist

Item	Decision rule
Token budget (<code>--max-num-batched-tokens / --chunked-prefill-size</code>)	Pinned explicitly; multiple of 256; highest goodput among values meeting ITL p99
SGLang <code>--enable-mixed-chunk</code>	Sweep both settings
Concurrency (<code>--max-num-seqs / --max-running-requests</code>)	Sized from KV capacity; decodes consume budget
<code>--long-prefill-token-threshold</code> , <code>--max-num-queued-tokens</code>	Use when long prompts starve short ones, or to cap TTFT backlog
Prefix cache	Size prefill load from uncached tokens
Escalation	No B meets both SLOs: add replicas or evaluate disaggregation

GPU Smith can run this sweep against your own trace and hardware as part of an inference sizing engagement.

Sources

1. vLLM, "Optimization and Tuning", docs (latest), accessed 2026-10-28. <https://docs.vllm.ai/en/latest/configuration/optimization/>
2. vLLM, `vllm/engine/arg_utils.py`, tag v0.32.0 (`get_batch_defaults`). https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/engine/arg_utils.py
3. vLLM, `vllm/config/scheduler.py`, tag v0.32.0. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/config/scheduler.py>
4. vLLM, `vllm/v1/metrics/loggers.py`, tag v0.32.0. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/v1/metrics/loggers.py>

5. vLLM, `vllm/benchmarks/serve.py`, tag v0.32.0. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/benchmarks/serve.py>
6. vLLM v0.32.0 release (2026-10-18). <https://github.com/vllm-project/vllm/releases/tag/v0.32.0>
7. vLLM, "Automatic Prefix Caching", docs (latest).
https://docs.vllm.ai/en/latest/features/automatic_prefix_caching/
8. vLLM, "Disaggregated Prefilling (experimental)", docs (latest).
https://docs.vllm.ai/en/latest/features/disagg_prefill/
9. Agrawal et al., "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve", arXiv 2403.02310 (v3, 2024-06-17). <https://arxiv.org/abs/2403.02310> (full text: <https://arxiv.org/html/2403.02310v3>)
10. USENIX OSDI '24 presentation page, Sarathi-Serve.
<https://www.usenix.org/conference/osdi24/presentation/agrawal>
11. SGLang, "Server Arguments", docs. https://docs.sglang.io/advanced_features/server_arguments.html
12. SGLang, `python/sglang/srt/arg_groups/memory_hook.py`, tag v0.5.22. https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/arg_groups/memory_hook.py
13. SGLang, `python/sglang/srt/arg_groups/fields/schedule.py`, tag v0.5.22. https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/arg_groups/fields/schedule.py
14. SGLang v0.5.22 release (2026-10-19). <https://github.com/sgl-project/sglang/releases/tag/v0.5.22>
Scheduler loop: `python/sglang/srt/managers/scheduler.py` @ v0.5.22, <https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/managers/scheduler.py>
15. Zhong et al., "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving", arXiv 2401.09670. <https://arxiv.org/abs/2401.09670>
16. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention", arXiv 2309.06180 (SOSP 2023). <https://arxiv.org/abs/2309.06180>

Source links

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

1. <https://arxiv.org/abs/2403.02310>
2. <https://arxiv.org/html/2403.02310v3>
3. <https://arxiv.org/abs/2401.09670>
4. <https://www.usenix.org/conference/osdi24/presentation/agrawal>
5. <https://docs.vllm.ai/en/latest/configuration/optimization>
6. https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/engine/arg_utils.py
7. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/config/scheduler.py>
8. https://docs.sglang.io/advanced_features/server_arguments.html
9. https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/arg_groups/memory_hook.py
10. https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/arg_groups/fields/schedule.py
11. <https://github.com/sgl-project/sglang/blob/v0.5.22/python/sglang/srt/managers/scheduler.py>
12. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/benchmarks/serve.py>

13. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/v1/metrics/loggers.py>
14. https://docs.vllm.ai/en/latest/features/automatic_prefix_caching
15. <https://arxiv.org/abs/2309.06180>
16. https://docs.vllm.ai/en/latest/features/disagg_prefill
17. <https://github.com/vllm-project/vllm/releases/tag/v0.32.0>
18. <https://github.com/sgl-project/sglang/releases/tag/v0.5.22>

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.