



OpenAI-Compatible Gateways for Private LLM Clusters

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 · litellm proxy self hosted vllm · openai compatible gateway llm · kubernetes gateway api inference extension · envoy ai gateway · llm api gateway rate limiting tokens · private ai gateway

Original article: <https://gpusmith.com/articles/en/openai-compatible-gateway-for-private-inference>



In this article

1. Why a private inference cluster needs a gateway
 2. What the model servers already provide
 3. Pattern 1: LiteLLM-style proxy
 4. Pattern 2: Envoy-based AI gateway (Agent Router, formerly Envoy AI Gateway)
 5. Pattern 3: Gateway API Inference Extension and endpoint pickers
 6. Auth and tenancy design
 7. Quotas: requests versus tokens
 8. Routing and failover
 9. Observability and chargeback
 10. Decision matrix and rollout
 11. Sources
-

Why a private inference cluster needs a gateway

A self-hosted model server such as vLLM already speaks an OpenAI-style HTTP API, so the first internal consumer can point an unmodified OpenAI client at it. The trouble starts with the second consumer: the server has no notion of tenants, budgets, per-team model allow-lists or a hosted-provider fallback, and its built-in authentication is a shared static key. Once several teams share a GPU pool, the operator needs a front door that decides per request who is calling, which models they may use, how much capacity they may consume, and which replica or provider serves them.

This guide separates two tiers that are often conflated. The **API tier** handles OpenAI compatibility, identity, quotas, spend attribution and cross-provider failover. The **endpoint-picking tier** decides which model-server pod inside a pool receives the request, using model-server signals such as queue length, KV-cache state and prefix-cache locality. LiteLLM-style proxies live mostly in the first tier, the Kubernetes Gateway API Inference Extension in the second, and Envoy-based AI gateways can span both. Routing economics are covered in the [small vs large model routing cost model](#); this article covers the plumbing that makes a routing policy enforceable.

What the model servers already provide

The vLLM OpenAI-compatible server exposes, among others, `/v1/completions`, `/v1/chat/completions`, `/v1/responses`, `/v1/embeddings`, `/v1/audio/transcriptions` and `/v1/audio/translations`, plus an Anthropic-style `/v1/messages`, Cohere-style rerank and embed routes, `/v1/models`, `/health` and `/metrics` (vLLM docs: [Online serving](#); all present in the v0.32.0 source, released 2026-10-18). A gateway in front of vLLM therefore rarely needs to translate formats for self-hosted models, only to pass them through and account for them.

Authentication is the weak point. In v0.32.0 the `--api-key` flag (or `VLLM_API_KEY`) only authenticates paths under `/v1`, `/v2`, `/inference` and `/cohere` ([authenticate.py @ v0.32.0](#)); other endpoints, including unprefixed `/rerank` and `/score` and "most notably `/invocations`", are not authenticated, and the documentation states:

"Do not rely on `--api-key` alone to secure vLLM" (vLLM docs: [OpenAI-compatible server](#)). Model-server pods should therefore be reachable only from the gateway (network policy, mesh authorization or a private subnet).

vLLM also exports the signals both tiers need, such as `vllm:time_to_first_token_seconds`, `vllm:inter_token_latency_seconds`, `vllm:num_requests_running` and `vllm:kv_cache_usage_perc` (vLLM docs: [Metrics design](#)). What it does not provide is anything keyed to a tenant.

Table 1. Responsibility split between model server and gateway tiers

Concern	Model server (vLLM)	API tier (proxy / Tier One)	Endpoint-picking tier (EPP / Tier Two)
OpenAI-format API	Yes	Pass-through or translation for hosted providers	Parses requests to read model name
Tenant identity and keys	One static key, partial path coverage	Virtual keys, JWT/OIDC mapping	No
Model allow-lists	No	Per key / team	No
Token and spend limits	No	Yes	No (admission control only)
Replica selection	No	Coarse load balancing across deployments	Load-, cache- and LoRA-aware
Priority under saturation	No	Limited	Queueing by priority (llm-d)
Hosted-provider fallback	No	Yes	No

Pattern 1: LiteLLM-style proxy

LiteLLM Proxy describes itself as "one OpenAI-compatible endpoint for 100+ LLM providers, MCP tools, and A2A agents"; existing OpenAI clients switch by changing the base URL ([LiteLLM docs: Proxy overview](#)). A self-hosted vLLM server is registered with the `hosted_vllm/<model-name>` prefix and an `api_base`; the older `vllm/` prefix is deprecated ([LiteLLM docs: vLLM provider](#)). The client-facing alias (`model_name`) is decoupled from the backend, so a checkpoint can be swapped without touching clients.

Tenancy is built on virtual keys: "Give each user, team, and project a virtual key", each carrying its own models, budgets and rate limits, with spend visible by key, team, user and tag ([LiteLLM docs: Proxy overview](#)). Virtual keys require a Postgres database (`DATABASE_URL`) and a master key beginning with `sk-`; keys are minted via `POST /key/generate`, restricted with a `models` list, and rotated via `/key/<key>/regenerate` with an optional `grace_period` such as "48h" ([LiteLLM docs: Virtual keys](#)).

Per the budgets page fetched 2026-10-28, limits are set with `tpm_limit` (tokens per minute), `rpm_limit` (requests per minute) and `max_parallel_requests` at team, internal-user or key level, plus `max_budget` with a `budget_duration` reset window ([LiteLLM docs: Budgets and rate limits](#)). TPM counts total tokens by default;

`token_rate_limit_type` can restrict counting to input or output tokens. Three operational details matter more than the parameter names:

- **Budgets need the database.** Without Postgres, budgets are not enforced.
- **Counters are shared through Redis.** If Redis is down, each proxy instance enforces limits independently, so three replicas turn a 100,000 TPM limit into up to 300,000 TPM (derived); `fail_closed_rate_limit_enforcement: true` returns 503 instead.
- **Budget reservation is on by default.** The proxy estimates a request's maximum cost, reserves it against the budget, and later replaces the reservation with the actual cost; disabling it lets concurrent requests overshoot.

The documented examples show 400, 401 and 429 for different budget breaches, so client retry logic should key on the error body, not a single status code.

Failover is configured per model group: `fallbacks`, `context_window_fallbacks` and `content_policy_fallbacks` lists, `num_retries` (retries happen before fallback), and `allowed_fails / cooldown_time` to bench a failing deployment ([LiteLLM docs: Reliability](#)).

The limitation is pool visibility: LiteLLM balances across deployments, each typically a Service URL, without seeing per-pod KV-cache occupancy or prefix-cache locality.

Pattern 2: Envoy-based AI gateway (Agent Router, formerly Envoy AI Gateway)

The project's README, fetched 2026-10-28, states: "Envoy AI Gateway is now **Agent Router**, an Agentic AI Foundation project," with the repository now at github.com/theagentrouter/agent-router and docs at theagentrouter.ai (old links redirect); CRDs, API group, `aigw` CLI and images keep their names ([Agent Router README](#); [Agent Router docs](#)). The latest stable tag at the time of writing is v1.2.0, published 2026-10-07 ([Agent Router v1.2.0 release](#)).

The README describes a two-tier pattern: "The **Tier One Gateway** handles authentication, top-level routing, and global rate limiting", while "The **Tier Two Gateway** provides fine-grained control over self-hosted model access, with endpoint picker support for LLM inference optimization" ([Agent Router README](#)).

Token-based limiting uses Envoy Gateway's Global Rate Limit API backed by Redis. An `AIGatewayRoute` declares `llmRequestCosts` of type `InputToken`, `CachedInputToken`, `OutputToken`, `TotalToken` or `CEL` (a custom expression), and a `BackendTrafficPolicy` sets the limits, typically keyed on a tenant header and the `x-ai-eg-model` header ([Agent Router docs: Usage-based rate limiting](#), v1.2). The enforcement timing matters most: tokens are charged after the response completes (for streams, when the stream ends), and a request is rejected with 429 only if usage already charged exceeds the limit. An admitted stream is not cut off. The same page distinguishes this per-client allowance from `QuotaPolicy`, which caps a backend's upstream consumption for a model across all routes and consumers.

For self-hosted pools, an `AIGatewayRoute` rule can reference a backend with `group: inference.networking.k8s.io` and `kind: InferencePool`, routing by the model name extracted from the request body, and the documentation presents mixing `InferencePool` and conventional backends in one route

as supported ([Agent Router docs: AIGatewayRoute + InferencePool](#)). That makes it a single control plane for both tiers on clusters already standardized on Envoy Gateway.

Pattern 3: Gateway API Inference Extension and endpoint pickers

The Gateway API Inference Extension (GIE) is "an official Kubernetes project that optimizes self-hosting Generative Models on Kubernetes" ([GIE home](#)). It uses Envoy's ext-proc protocol to extend any gateway supporting ext-proc and Gateway API, and positions itself underneath higher-level AI gateways such as LiteLLM, Gloo AI Gateway or Apigee ([GIE: API overview](#)).

The core resource is the **InferencePool**, `inference.networking.k8s.io/v1`, GA since GIE v1.0.0. It groups pods sharing a base model, accelerator type and model server; its spec has a pod selector, `targetPorts` and an `endpointPickerRef` pointing at the Endpoint Picker (EPP) service, and the documented example sets `failureMode: FailOpen` so the gateway picks an endpoint itself if the EPP does not respond ([GIE: InferencePool](#)). In Gateway API terms an InferencePool replaces a Service as an HTTPRoute backend.

The project's scope changed in 2026, so older tutorials are partly wrong. The v1.6.0 release (2026-08-17) moved the full-featured EPP, Body-Based Routing and the latency predictor to the llm-d organization, removed the alpha `InferenceObjective`, `InferenceModelRewrite` and `EndpointPickerConfig` APIs from GIE, made `endpointPickerRef` optional, and introduced a lightweight EPP (lwepp) intended for conformance testing ([GIE v1.6.0 release notes](#)). The latest GIE release at the time of writing is v1.6.2 (2026-09-17) ([GIE v1.6.2 release](#)). The home page still lists "serving priority"; that capability now lives in llm-d.

The production EPP is now the **llm-d Router** (formerly "Inference Scheduler"), which hosts the EPP, `InferenceObjective` and `InferenceModelRewrite` and runs either standalone with its own Envoy or behind a Gateway via InferencePool ([llm-d Router README](#)). Its v1.0.0 release (2026-10-16) graduated `InferenceObjective` to `llm-d.ai/v1` with multi-pool targeting ([llm-d Router v1.0.0 release](#)). The EPP filters candidate pods and scores them with weighted scorers for KV-cache locality, session affinity, load and model metadata; flow control (the admission and queueing layer) is off by default and enabled with the `flowControl` feature gate ([llm-d Router architecture](#)). The `InferenceObjective` priority field is an integer where higher is more critical; under scarcity, flow control "will *a/ways* allow requests of higher priority to be served first," and fairness applies only among requests of equal priority ([llm-d InferenceObjective types](#)).

The implementations page lists Alibaba Cloud ACK, Istio, Agentgateway, Higress and NGINX Gateway Fabric ([GIE: Gateway implementations](#)); the home page also names Envoy Gateway, kgateway and GKE Gateway.

Auth and tenancy design

A workable tenancy model has three layers: **identity** (ideally inherited from corporate SSO), **entitlement** (which model aliases it may call) and **allowance** (tokens, dollars or concurrent requests).

In LiteLLM, the simplest implementation is one team per consuming application, virtual keys per team, and a team-level `models` list that keys with `all-team-models` inherit ([LiteLLM docs: Virtual keys](#)). For SSO mapping, JWT auth verifies tokens from an OIDC provider via `JWT_PUBLIC_KEY_URL`, maps `user_id_jwt_field` and `team_id_jwt_field` claims, and with `enforce_team_based_model_access: true` admits a request only if

one of the user's groups maps to a team with access to the model; this is an Enterprise-licensed feature ([LiteLLM docs: JWT auth](#)). In an Envoy-based design, gateway JWT or external-authorization filters stamp a tenant header that rate-limit policies key on.

Static keys on the model servers, if used, should be gateway-to-backend credentials no human holds, and key rotation with a grace period should be rehearsed before an incident. Air-gapped deployments add their own constraints on identity providers and image provenance, discussed in the [air-gapped LLM deployment guide](#).

Quotas: requests versus tokens

A requests-per-minute limit treats a 200-token chat turn and a 30,000-token document summarization as equal. On a GPU they are not: prefill grows with prompt length, KV-cache footprint with prompt plus output, and decode time with output.

Worked example (illustrative workloads). Two tenants each receive 60 RPM.

- Tenant A, chat: 1,500 input + 300 output = **1,800 tokens per request**; $60 \times 1,800 = \mathbf{108,000}$ tokens/min.
- Tenant B, retrieval-augmented summarization: 24,000 input + 1,000 output = **25,000 tokens per request**; $60 \times 25,000 = \mathbf{1,500,000}$ tokens/min.

Identical request limits allow Tenant B about $13.9\times$ ($1,500,000 \div 108,000$) the token volume of Tenant A. A shared 200,000 tokens/min limit instead admits about 111 chat requests per minute ($200,000 \div 1,800$) but only 8 summarization requests ($200,000 \div 25,000$, rounded down), which tracks GPU work far better.

Total tokens still under-weight output if decode is the scarce resource. A weighted cost (an Agent Router CEL cost) can encode that. With an illustrative weight of 4 per output token: Tenant A costs $1,500 + 4 \times 300 = 2,700$ units; Tenant B costs $24,000 + 4 \times 1,000 = 28,000$ units, a ratio of about $10.4\times$ rather than $13.9\times$. A real weight must come from measured prefill and decode cost.

Overshoot from post-hoc charging (derived). Because Agent Router charges tokens after completion and rejects only once charged usage exceeds the limit, worst-case overshoot is roughly the requests admitted while under the limit times their maximum size. If a tenant has 10 concurrent streams in flight when it crosses a 100,000-token hourly limit, each with an 8,000-token prompt and `max_tokens` of 4,096, the window can close at up to $100,000 + 10 \times 12,096 = \mathbf{220,960}$ tokens, about $2.2\times$ the nominal limit. The remedies are a concurrency cap per tenant and a server-side ceiling on `max_tokens`. LiteLLM's budget reservation addresses the same problem for dollar budgets.

Concurrency and priority. Token limits govern volume; `max_parallel_requests` (LiteLLM) or a per-tenant concurrency limit governs instantaneous pressure on KV cache. By Little's law, Tenant B at 8 requests/min (0.133 req/s) with an illustrative 20-second latency holds about 2.7 requests in flight. Priority belongs in the endpoint-picking tier, where llm-d flow control can serve interactive traffic ahead of batch traffic once the pool saturates.

Table 2. Limit types and where to enforce them

Limit	Protects against	Enforce at	Caveat
Requests/min	Abusive clients, retry storms	API tier	Blind to prompt size
Tokens/min (total or weighted)	Volume imbalance between tenants	API tier	Post-hoc charging allows overshoot
Concurrent requests	KV-cache exhaustion, latency spikes	API tier	Needs shared state across replicas
Dollar budget per period	Chargeback overruns	API tier	Needs database; zero-cost models bypass
Priority / queueing	Interactive SLO loss under saturation	Endpoint-picking tier	Flow control off by default in llm-d

Routing and failover

Routing happens at three levels: across providers (private pool versus hosted endpoints), across pools (model alias to InferencePool or Service), and within a pool (the EPP selects the pod). Cache-aware picking matters most for long shared prefixes such as system prompts, where a pod already holding the prefix skips recomputing prefill; the llm-d EPP includes prefix-cache scoring ([llm-d Router architecture](#)).

Two failure modes need explicit decisions. On EPP failure, FailOpen keeps traffic flowing with degraded placement, while fail-closed protects SLOs at the cost of availability. Hosted-provider fallback is a data-governance decision as much as a reliability one and should be set per model alias, so regulated workloads never leave the perimeter.

Observability and chargeback

LiteLLM records the cost of each request and reports spend by key, team, user and tag, exporting to Prometheus, OpenTelemetry and other sinks ([LiteLLM docs: Proxy overview](#)). Self-hosted models need an internal price, set as `input_cost_per_token` and `output_cost_per_token` under `model_info`. Setting both to 0 makes the model free and lets it bypass budget checks, while leaving them null keeps the model billable ([LiteLLM docs: Custom pricing](#)). An operator who wants budgets enforced on private models therefore needs a non-zero internal rate.

Worked chargeback (illustrative rates). Assume an internal transfer price of \$0.20 per million input tokens and \$0.80 per million output tokens, configured as `input_cost_per_token: 0.0000002` and `output_cost_per_token: 0.0000008`. Tenant B sends 200,000 requests in a month: $200,000 \times 24,000 = 4.8$ billion input tokens $\times \$0.20/M = \mathbf{\$960}$; $200,000 \times 1,000 = 0.2$ billion output tokens $\times \$0.80/M = \mathbf{\$160}$; total **\$1,120**. Real rates should come from the cluster's fully loaded cost per token.

Latency SLOs come from vLLM's histograms (TTFT, inter-token latency, end-to-end latency) joined to tenant labels from gateway logs. The definitions and pitfalls of those metrics are covered in [inference SLO metrics: TTFT, ITL, TPS and goodput](#). Gateway-measured latency includes gateway and EPP queueing and should be reported alongside server-side numbers.

Decision matrix and rollout

Table 3. Choosing a pattern (qualitative)

Situation	Reasonable starting point	Main risk
One pool, few teams, no Kubernetes gateway yet	LiteLLM proxy in front of vLLM Services	No cache-aware pod selection; Redis and Postgres become critical dependencies
Kubernetes platform team, many pools, mixed private and hosted models	Agent Router (Tier One) + InferencePool backends (Tier Two)	Rename churn in docs; post-hoc token charging
Large shared pools where prefix reuse and priority matter	GIE InferencePool + llm-d Router EPP, with an API tier in front	Fast-moving APIs; flow control must be enabled and tuned
Existing enterprise API management	Keep it as Tier One; add InferencePool + EPP behind it	Token accounting may need custom work

A low-risk rollout sequence is: (1) lock the model servers behind network policy so the gateway is the only path; (2) introduce the API tier with model aliases, per-team keys and request limits only, in observe mode where available; (3) add token limits and concurrency caps sized from observed token distributions; (4) replace Services with InferencePools and an EPP for pools with heavy prefix reuse, starting fail-open; (5) enable priority and flow control once interactive and batch traffic are labeled. The [vLLM KV cache capacity planner](#) gives concurrency caps a defensible basis.

The versions referenced here (Agent Router v1.2.0, GIE v1.6.2, llm-d Router v1.0.0, LiteLLM v1.106.6 of 2026-10-28) are recent and changing quickly; configuration field names should be rechecked against the release in use. GPU Smith can review a proposed gateway design against a specific cluster's tenancy and SLO requirements.

Sources

1. [Online serving](#) - vLLM project, list of OpenAI-compatible and other endpoints (accessed 2026-10-28)
2. [OpenAI-compatible server](#) - vLLM project, --api-key path coverage and hardening warning (accessed 2026-10-28)
3. [Metrics design](#) - vLLM project, Prometheus metric names for TTFT, ITL, KV cache and queue (accessed 2026-10-28)
4. [LiteLLM Proxy overview](#) - BerriAI LiteLLM, OpenAI-compatible endpoint, virtual keys, spend tracking, observability sinks (accessed 2026-10-28)
5. [Virtual keys](#) - BerriAI LiteLLM, Postgres and master-key requirements, model restrictions, key rotation (accessed 2026-10-28)
6. [Budgets and rate limits](#) - BerriAI LiteLLM, tpm/rpm/parallel limits, Redis sharing, fail-closed enforcement, budget reservation (accessed 2026-10-28)
7. [vLLM provider](#) - BerriAI LiteLLM, hosted_vllm/ prefix and api_base configuration (accessed 2026-10-28)

8. [Reliability: fallbacks, retries, cooldowns](#) - BerriAI LiteLLM, fallback types, retries and cooldown settings (accessed 2026-10-28)
9. [Custom pricing](#) - BerriAI LiteLLM, per-token pricing and zero-cost budget bypass (accessed 2026-10-28)
10. [JWT-based auth](#) - BerriAI LiteLLM, OIDC claim mapping and team-based model access, Enterprise feature (accessed 2026-10-28)
11. [Agent Router README](#) - Agentic AI Foundation, rename from Envoy AI Gateway, new URLs, two-tier gateway pattern (accessed 2026-10-28)
12. [Agent Router documentation](#) - Agentic AI Foundation, product name, version 1.2 docs, unchanged CRD and CLI names (accessed 2026-10-28)
13. [Usage-based rate limiting](#) - Agent Router docs v1.2, llmRequestCosts types, post-response charging, QuotaPolicy distinction (accessed 2026-10-28)
14. [AIGatewayRoute + InferencePool guide](#) - Agent Router docs v1.2, InferencePool backend references and model-name routing (accessed 2026-10-28)
15. [Agent Router v1.2.0 release](#) - GitHub, release published 2026-10-07 (accessed 2026-10-28)
16. [Gateway API Inference Extension](#) - Kubernetes SIG Network, project definition, supported gateways, serving priority claim (accessed 2026-10-28)
17. [API overview](#) - Kubernetes SIG Network, ext-proc mechanism and positioning under AI gateways (accessed 2026-10-28)
18. [InferencePool](#) - Kubernetes SIG Network, v1 API fields, EPP reference, FailOpen example (accessed 2026-10-28)
19. [Gateway implementations](#) - Kubernetes SIG Network, list of integrating gateways (accessed 2026-10-28)
20. [GIE v1.6.0 release notes](#) - GitHub, EPP migration to llm-d, removed alpha APIs, optional endpointPickerRef, lwepp (accessed 2026-10-28)
21. [GIE v1.6.2 release](#) - GitHub, latest GIE release published 2026-09-17 (accessed 2026-10-28)
22. [llm-d Router README](#) - llm-d project, EPP, InferenceObjective, standalone and gateway modes (accessed 2026-10-28)
23. [llm-d Router v1.0.0 release](#) - GitHub, InferenceObjective graduation to llm-d.ai/v1, published 2026-10-16 (accessed 2026-10-28)
24. [llm-d Router architecture](#) - llm-d project, scorers, prefix-cache scoring, flow control feature gate (accessed 2026-10-28)
25. [InferenceObjective types](#) - llm-d project, priority field semantics under flow control (accessed 2026-10-28)

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/serving/online_serving
2. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/entrypoints/serve/middleware/authenticate.py>
3. https://docs.vllm.ai/en/stable/serving/online_serving/openai_compatible_server
4. <https://docs.vllm.ai/en/stable/design/metrics>

5. https://docs.litellm.ai/docs/simple_proxy
6. <https://docs.litellm.ai/docs/providers/vllm>
7. https://docs.litellm.ai/docs/proxy/virtual_keys
8. <https://docs.litellm.ai/docs/proxy/users>
9. <https://docs.litellm.ai/docs/proxy/reliability>
10. <https://github.com/theagentrouter/agent-router>
11. <https://theagentrouter.ai/docs>
12. <https://github.com/theagentrouter/agent-router/releases/tag/v1.2.0>
13. <https://theagentrouter.ai/docs/capabilities/traffic/usage-based-ratelimiting>
14. <https://theagentrouter.ai/docs/capabilities/inference/aigatewayroute-inferencepool>
15. <https://gateway-api-inference-extension.sigs.k8s.io>
16. <https://gateway-api-inference-extension.sigs.k8s.io/concepts/api-overview>
17. <https://gateway-api-inference-extension.sigs.k8s.io/api-types/inferencepool>
18. <https://github.com/kubernetes-sigs/gateway-api-inference-extension/releases/tag/v1.6.0>
19. <https://github.com/kubernetes-sigs/gateway-api-inference-extension/releases/tag/v1.6.2>
20. <https://github.com/llm-d/llm-d-router>
21. <https://github.com/llm-d/llm-d-router/releases/tag/v1.0.0>
22. <https://github.com/llm-d/llm-d-router/blob/main/docs/architecture.md>
23. https://github.com/llm-d/llm-d-router/blob/main/apix/v1/inferenceobjective_types.go
24. <https://gateway-api-inference-extension.sigs.k8s.io/implementations/gateways>
25. https://docs.litellm.ai/docs/proxy/token_auth
26. https://docs.litellm.ai/docs/proxy/custom_pricing

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.