Multicloud.io

GPUS EXIST · JUST NOT IN YOUR REGION

Serving LLMs Without Waiting for GPUs

Cost-efficient LLM inference with vLLM and llm-d on Multicloud — starting from a single virtual node in the cluster you already run, and scaling, if you choose, to your entire inference estate.

LLM Inference Whitepaper | July 2026 | multicloud.io

EXECUTIVE SUMMARY

The argument, in one page

If you self-host models, GPU compute is your cost of goods — and it is gated twice: by **capacity** (quota and stock-outs are per-region facts) and by **price** (the same GPU trades at up to 6× between regions, with a further 40–90% discount class in the spot market). The tools a Kubernetes team reaches for stop at one cluster in one region of one cloud.

The serving layer, meanwhile, is solved — and open source. **vLLM** is the de-facto inference engine; **llm-d** (CNCF, founded by Red Hat, Google Cloud, IBM Research, CoreWeave, and NVIDIA) adds KV-cache-aware routing, prefill/decode disaggregation, and variant autoscaling on standard Kubernetes. But read its documentation closely: llm-d schedules requests across GPU pods *that already exist*. It deliberately does not provision nodes, pick regions or clouds, or manage spot.

> Every optimization the serving layer makes is bounded by the price and availability of the nodes underneath it. The 6× regional spreads and 40–90% spot discounts live **below** the serving layer — in the layer that decides which machines to buy. That layer is what Multicloud is.

This paper describes the resulting architecture: llm-d and vLLM run unmodified on a Multicloud pool — a conformant Kubernetes cluster built continuously from the cheapest qualifying capacity across every region of AWS, Azure, and Google Cloud, in cloud accounts you own. You adopt it through **Extend your cluster** (early access): one virtual node appears in the cluster you already run; replicas you mark for it execute on the pool. Burst is the doorway, not the ceiling — the dial runs from one overflow variant to your entire inference estate, with no migration event anywhere along the way.

Claims in this paper stay falsifiable: every market number is sourced, every platform capability carries its status (production, early access, or roadmap), and the price ladder of section 10 is reproducible against public price lists — or against your own workloads, via a read-only audit.

CONTENTS

What's in this paper

This paper describes a reference architecture on the platform as it stands, with status flags where a capability is **(early access)** or roadmap. vLLM and llm-d are upstream open-source projects (Apache-2.0), installed unmodified. Ladder prices are point-in-time public list prices; ask us to run the live catalog — or the Advisor, on your own cluster — for current numbers on your workloads.

SECTION 01

01 The problem: capacity first, price second

Venture analyses put compute at up to 80% of total capital raised for AI companies (a16z), and for an inference product every routed token lands on that line. Three market facts shape it:

6×

The hourly cost of an identical H100 can differ by more than 6× between regions (Silicon Data).

−41…82%

Observed 2026 spot discounts vs. on-demand: L40S −82%, A100 −61%, H100 −41% (GPU Tracker); the big three publish 60–90% spot ranges.

0

GPUs your autoscaler can reach outside its own region when quota runs dry — Cluster Autoscaler, Karpenter, and node auto-provisioning all stop at one cluster in one region of one cloud.

Capacity is gated by region. GPU quota is granted per region; frontier SKUs sell out per region; "on-demand" frequently means *available eventually, if you planned ahead* (SemiAnalysis). Teams describe hunting accelerators across regions and providers like booking the last flight out.

The same GPU trades at wildly different prices. Beyond the regional spread, an entire discount class sits in the spot market — and the scarcer the GPU, the shallower its spot discount, which is exactly why the ability to reach *more markets* matters more than any single market's discount.

The cheap capacity asks you to absorb interruptions. Spot GPUs are reclaimed with seconds-to-minutes of notice. Operating that safely — detection, graceful drain, replacement, rescheduling — is precisely the burden that keeps teams paying on-demand prices for interruption-tolerant workloads.

> The capacity you need exists. Your cluster just can't reach it. This paper fixes the reach, not the serving stack.

SECTION 02

02 The serving layer is solved: vLLM and llm-d

You do not need to invent LLM serving. The open-source stack is mature, fast-moving, and Kubernetes-native — and it is the same stack the hyperscalers themselves productize.

vLLM — the engine

The de-facto open-source inference engine: PagedAttention, continuous batching, automatic prefix caching, speculative decoding, quantization from FP8 down to INT4, across 200+ model architectures. Two properties matter economically. **Hardware breadth:** vLLM runs on every NVIDIA GPU from T4 through L4, L40S, A100, H100/H200, and B200, on AMD MI300X as a first-class platform, and on Google TPU — so the set of machines that can serve your model is wide, which means the set of *prices* available to you is wide, if your platform can reach them. **Parallelism:** tensor parallelism inside a node and pipeline parallelism across nodes let one engine span whatever node shape is cheapest that satisfies the model.

llm-d — the layer above the engine

Founded by Red Hat, Google Cloud, IBM Research, CoreWeave, and NVIDIA; Apache-2.0; donated to the CNCF in March 2026. Not a monolith — a composition of standard Kubernetes pieces:

- **The Inference Gateway.** Builds on the Kubernetes Gateway API Inference Extension: an *InferencePool* resource plus an *endpoint picker* the gateway consults per request — routing to the vLLM replica that already holds the request's KV-cache prefix, has the shortest queue, or promises the best predicted latency.
- **Prefill/decode disaggregation.** Prefill is compute-bound; decode is memory-bandwidth-bound. llm-d runs them as separately sized and scaled pod pools, with KV state handed off over a high-speed transfer library.
- **KV-cache management.** A global prefix-cache index, with hierarchical offload of KV blocks to CPU RAM and disk — expensive GPU memory holds only what earns its place.
- **Variant autoscaling.** The Workload Variant Autoscaler treats the same model on different accelerators (L4 vs. A100 vs. H100) as variants, watches SLOs, and drives standard HPA/KEDA toward the **cheapest variant that meets the SLO**.

> The published numbers are the reason this layer exists: 3× output throughput and 2× faster time-to-first-token from prefix-cache-aware routing; 10–30% (up to 70%) more tokens/sec from prefill/decode disaggregation; 40% latency reduction from predicted-latency scheduling; in Google's production use, 35% lower TTFT and 92.8% shorter queue waits.

SECTION 03

03 What the serving layer assumes — the seam

Read llm-d's documentation closely and a boundary appears, stated plainly: llm-d schedules **requests across vLLM pods that already exist**. It assumes the cluster already has GPU nodes — drivers installed, node pools per accelerator type, capacity present or arriving. Its variant autoscaler emits *desired replica counts*; standard HPA performs the scaling; and when the resulting pod goes *Pending* because no node can hold it, llm-d's world ends.

LLM-D OPTIMIZES	LLM-D ASSUMES SOMEONE ELSE PROVIDES
Which replica serves a request	Which **nodes** exist, and where
How many replicas of which variant	Which **accelerator, region, cloud, and price** each node has
KV-cache placement across GPU / CPU / disk	Spot lifecycle: detection, drain, replacement
Prefill/decode pool balance	Cross-region and cross-cloud reach

This is the seam, and it is where the money is. Every optimization llm-d makes is bounded by the price and availability of the nodes underneath it: route requests perfectly across GPUs you overpaid for — or queue behind quota in the one region your autoscaler can see — and you have optimized the numerator while the denominator eats the margin.

> The 6× regional spreads and 40–90% spot discounts of section 01 live **below** the serving layer, in the layer that decides which machines to buy. That layer is what Multicloud is.

This division is not an accident of llm-d's youth — it is good layering. The serving layer works in milliseconds against a fixed fleet; the capacity layer works in minutes against a moving market. The rest of this paper is about running the first on the second.

SECTION 04

04 The capacity layer: one decision over the whole GPU board

Multicloud operates the capacity market the serving stack assumes. The mechanism, briefly — the full story is the companion *Architecture Whitepaper*:

A live catalog. Roughly **265,000 priced offers** across ~3,900 machine types — every region and zone of AWS, Azure, and Google Cloud, spot and on-demand, with per-zone interruption-risk scores, CPU benchmarks keyed to the actual silicon, and **76 catalogued accelerator types** for GPU-class matching.

One placement decision. The scheduler packs pending work and buys the **cheapest qualifying** machines from that whole catalog — whatever cloud, region, and zone; spot-aware where your constraints tolerate interruption, on-demand where they don't. Constraints are yours per workload: a specific GPU class, a memory floor, a benchmarked-compute floor, allowed regions, spend caps, on-demand only.

Interruptions as weather. Every node carries an agent watching its cloud's reclaim signals. On notice: drain honoring your disruption budgets, replacement bought from the live catalog — often in another zone, region, or cloud — and a new node serving in about a minute. Machines are replaced, never repaired; no machine holds durable state.

One network, one cluster. Nodes across clouds join a flat, WireGuard-encrypted, dual-stack fabric under a single conformant Kubernetes control plane. A cluster spanning three clouds is still *one cluster* — one kubeconfig, one scheduler view, no federation.

Your accounts, your bill. Machines are provisioned into dedicated cloud accounts you own. Your negotiated discounts, committed-use plans, and credits all still apply; Multicloud charges for the platform, never a markup on machines.

> Because the pool is upstream-conformant Kubernetes, llm-d is not "integrated" with it in any bespoke sense — **it simply installs**, the same Helm charts and Gateway API resources it uses anywhere else.

SECTION 05

05 The doorway: extend the cluster you already run

You do not migrate anything to adopt this. The entry point is **Extend your cluster** (early access): your existing cluster — EKS, GKE, AKS, or self-managed — gains one new node, a **virtual node** backed by your Multicloud pool. It carries a taint, so nothing lands on it by accident. Workloads you explicitly mark for it schedule there like anywhere else and execute on the pool — while remaining visible in your *kubectl*, your dashboards, and your CI/CD exactly like local pods. The connection is outbound-only and encrypted; no foreign machine ever joins your cluster.

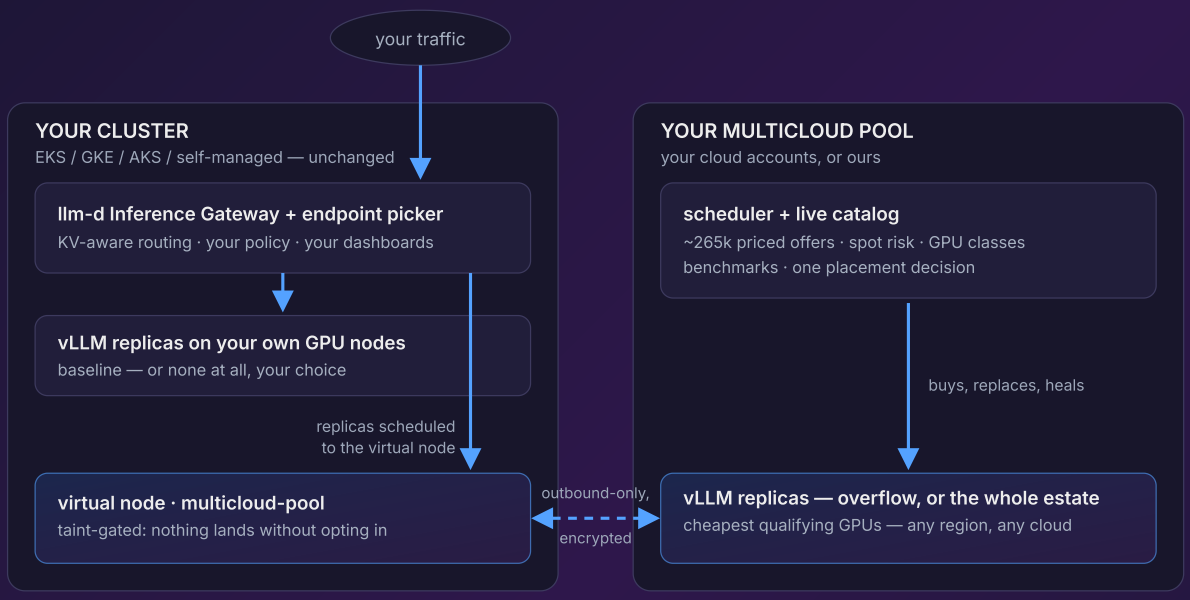

Figure 1 — the doorway: your gateway, routing policy, and dashboards stay home; pending demand behind the virtual node drives the pool's autoscaler, which buys the cheapest qualifying GPU on the whole board.

Your gateway stays home; your capacity stops being local. What changes is what happens when the variant autoscaler asks for replicas your cluster cannot place: instead of queueing behind regional quota, the demand drives the pool, and the replica comes back as a running pod on the virtual node.

A five-line diff

Bursting an inference deployment is the same burst-tier contract as any other workload — a toleration and a node selector. Nothing without that toleration can ever land on the virtual node:

```
spec:
  template:
    spec:
      tolerations:
        - key: multicloud.io/burst
          operator: Equal
          value: "true"
          effect: NoSchedule
      nodeSelector:
        multicloud.io/tier: burst
```

You adopt one model — or one variant of one model — at a time, watch its cost and latency, then turn the dial.

And the dial runs all the way to 100%

Burst is where adoption starts, not where the architecture stops — the same virtual node supports three steady states, and teams move between them by editing manifests, never by migrating:

1 **Overflow.** Your GPUs carry the baseline; the pool absorbs peaks, new variants, and experiments. The comparison is the pool versus your cloud's on-demand, for capacity you need anyway.

2 **All of it, through your cluster.** Point every inference tier at the burst tier and the entire serving estate — every replica of every variant, interactive tiers included — runs on the pool, while your cluster remains the front end: the llm-d gateway, routing policy, CI/CD, dashboards, and the *kubectl* your team already trusts. You can stop holding local GPUs entirely; every accelerator you pay for is market-priced, spot-aware, and continuously re-placed. Latency is a per-workload guardrail, not a hope: interactive tiers constrain cells to regions within their budget, and the optimizer works inside that boundary.

3 **Promoted.** When the virtual-node hop itself is no longer wanted, the pool you are already serving from is a full, isolated, conformant Kubernetes cluster — promotion to primary (standard kubeconfig, full admin, traffic served directly from the pool's edge) is a configuration step, not a migration (section 11).

SECTION 06

06 Serving cells: how llm-d runs on the pool

Prefill/decode disaggregation and KV-cache transfer want high-bandwidth, low-latency links. So the unit of llm-d deployment on the pool is what this paper calls a **serving cell**: one llm-d installation — gateway, endpoint picker, prefill pool, decode pool, KV-cache manager — placed **within one region**, where the interconnect is real. The two layers then divide the problem cleanly, each at the altitude it was built for:

Inside a cell, llm-d decides which replica serves each request — prefix-cache affinity, queue depth, predicted latency, prefill/decode balance. Milliseconds matter.

Underneath the cell, Multicloud decides which machines the cell runs on — which accelerator, which zone, spot or on-demand, at what price, replacing reclaimed nodes as weather. Dollars matter.

Above the cells, Multicloud decides where cells exist at all — which regions, which clouds, how many (section 08).

The pool never stretches a prefill/decode pair across regions, and this is deliberate: cross-region KV transfer would burn the latency the disaggregation exists to save. Cells are local; the *fleet of cells* is global.

Where the levers multiply: variants

llm-d's Workload Variant Autoscaler already prefers the cheapest accelerator variant that meets the SLO — but it can only choose among variants whose nodes exist. On a single-region cluster, "the L4 variant" has one price and one availability. On the pool, *every* variant is priced across every region and zone of three clouds, spot-aware and risk-scored, and the scheduler buys the cheapest instantiation of whichever variant the autoscaler asks for. The serving layer's cost-awareness and the capacity layer's cost-awareness multiply — neither replaces the other, because they optimize different levers on different clocks.

> Not all of an inference stack is GPUs. Gateways, endpoint pickers, cache indexes, and llm-d's hierarchical KV offload tier (KV blocks spilled to CPU RAM) all run on ordinary machines — and the same catalog places those on benchmarked, cheapest-qualifying CPU capacity, where the observed spread between "what most teams run" and the best qualifying offer is routinely above 90% (section 10).

SECTION 07

07 The loop, end to end: from SLO breach to purchased GPU

The two control loops meet at the oldest interface in Kubernetes: a *Pending* pod.

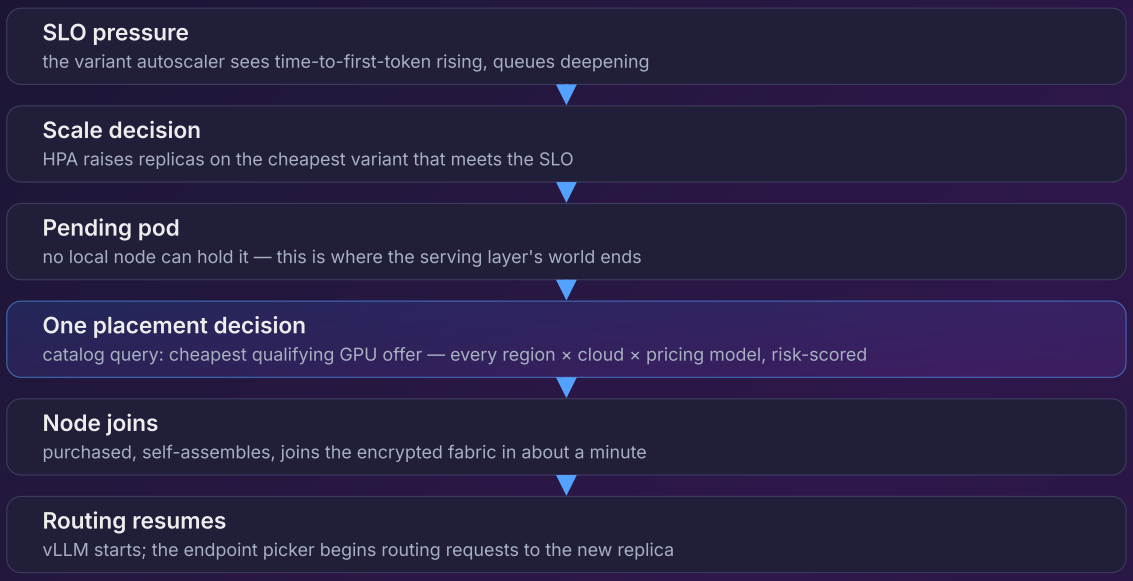

Figure 2 — scale-out: the serving layer's demand signal becomes the capacity layer's purchase, and the purchase becomes a routed replica.

And in reverse, when the cloud reclaims a spot GPU: the on-node agent catches the notice within seconds; the node drains gracefully, honoring your PodDisruptionBudgets, while the endpoint picker stops routing to the disappearing replica; a replacement is bought from the live catalog — often in another zone, sometimes another region or cloud, frequently *cheaper* than what was lost — joins, loads the model, and routing resumes.

Why inference rides spot well

Inference replicas are unusually well-suited to this weather: they hold **no durable state** — the KV cache is a performance artifact that is rebuilt, not recovered, and llm-d's prefix-aware routing simply warms the new replica as traffic arrives. The honest cost of a reclaim is a model-load: pulling weights onto the fresh node.

Mitigation, built in. Per-node image caching and configurable *warm headroom* — a chosen fraction of spare capacity kept standing so scale-out lands on existing slack instead of waiting for a purchase.

Spot is a dial, not a religion. Whatever must never ride spot — the gateway tier, a floor of decode replicas — you constrain to on-demand, per workload. The risk knob is also the spot-vs-on-demand selector; zero means on-demand only.

> Plan on 50–70% net spot savings after interruption overhead — not the sticker discount. This paper states it that way on purpose.

SECTION 08

08 Going global: many cells, one hostname

Once serving cells can exist anywhere, two placement strategies open up that a single-region stack cannot express. **Follow the users:** run cells in regions close to your traffic — time-to-first-token is a latency SLO, and a cell 40 ms from the user beats a cheaper cell 180 ms away for interactive serving. **Follow the price:** for latency-tolerant tiers — batch scoring, embedding runs, evaluation sweeps, async generation queues — cells land wherever qualifying GPUs are cheapest *that day*, and the answer changes as the market moves.

Name the geography, not the cloud. The natural unit for a cell target isn't "AWS us-east-1" or "GCE europe-west4" — it's a *place*: a latency footprint, and where it matters, a compliance boundary. "US East Coast" is the same low-latency story to a customer in Washington, D.C. whether the GPU behind it sits in an AWS zone in Ashburn, an Azure region in the same metro, or a Google Cloud zone next door — dozens of providers share that footprint, and for a latency-bound request any of them will do. And it isn't a single winner, either: a cell is one Kubernetes cluster spanning every cloud it's allowed to touch, so the scheduler buys **each node independently**, on whichever provider is cheapest for that purchase — a healthy cell routinely runs AWS, Azure, and GCE nodes side by side, at the same time, behind the same hostname. The same logic runs the other way for **Sovereign EU**: the boundary that matters is regulatory, not one cloud's region list — "any cloud, any region inside the EU" is the natural expression of the constraint, not a workaround for it.

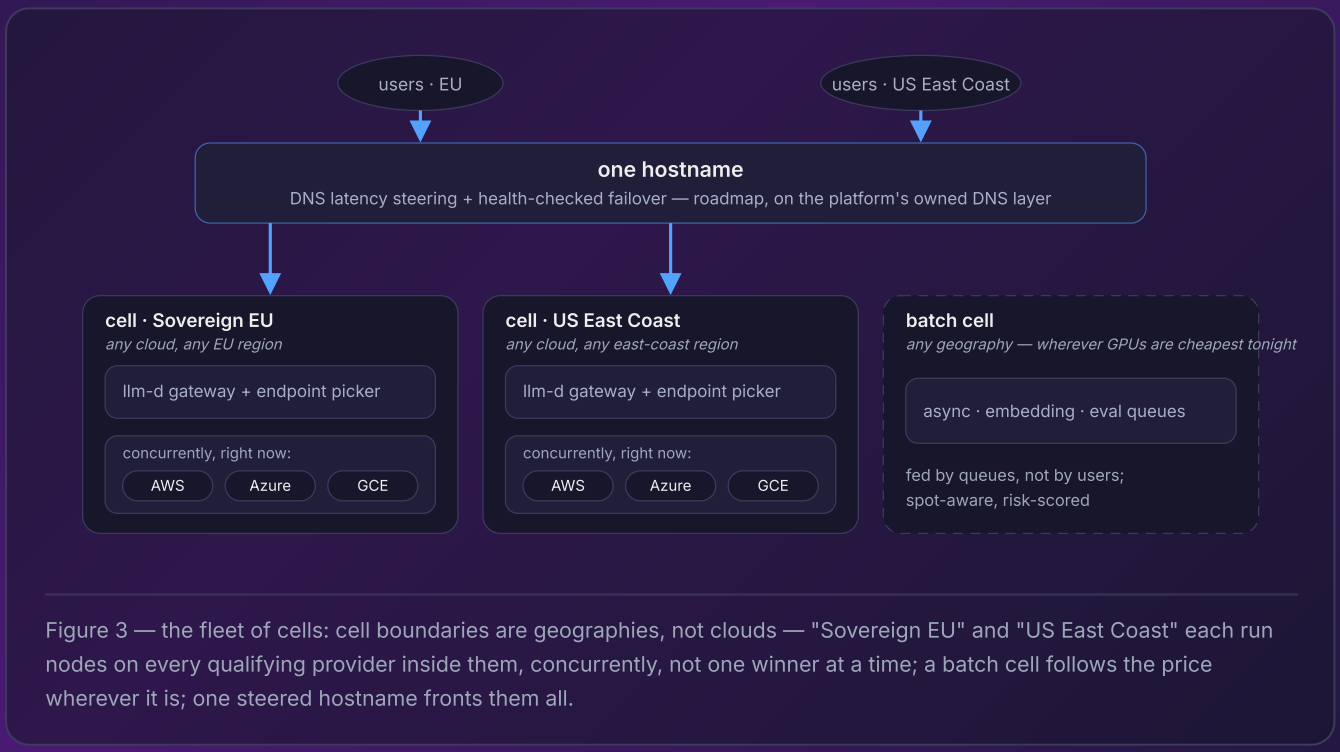

Figure 3 — the fleet of cells: cell boundaries are geographies, not clouds — "Sovereign EU" and "US East Coast" each run nodes on every qualifying provider inside them, concurrently, not one winner at a time; a batch cell follows the price wherever it is; one steered hostname fronts them all.

Read the cell names again: **Sovereign EU** and **US East Coast** name geographies, not clouds — each is a location rule (include every region of every provider that qualifies, exclude everything outside the boundary) that the scheduler resolves **node by node**, purchase by purchase, to whichever cloud and region is cheapest at that moment: the same one placement decision as everywhere else in this paper (section 04), just run once per node instead of once per cell. A customer never chooses "AWS or Azure or GCE" for a cell; they choose the geography it exists to serve, and every node inside it stays independently contestable — which is why a live cell is usually a mix, not a monoculture.

The edge underneath the cells

The platform already owns the edge that makes this one system: public DNS records are created and maintained automatically for services you expose, TLS is issued and renewed, and every node can terminate traffic. **Latency-steering and health-checked-failover DNS policies are the roadmap increment on top of that owned layer** (section 12) — the records, zones, and reapers are already platform-managed, so steering is a policy change, not a new subsystem. With it, a regional outage becomes a routing event: surviving cells absorb the traffic while the pool re-provisions the lost cell's capacity elsewhere — the same reschedule loop as section 07, at region scale.

Data locality still rules, and the Sovereign EU cell is that rule in its strictest form: a location constraint that must hold regardless of which cloud ends up serving a given replica. More generally, chatty, data-pinned, or residency-bound workloads carry location constraints and stay put; the optimizer prefers within-cloud multi-region placement by default (data movement there costs ~4–5× less than crossing clouds) and crosses clouds only when the savings clearly clear the egress cost. Model weights are the easy case — static, cacheable, replicated once per cell, not per request.

A live example, not a metaphor

Right now, Sovereign EU. A request lands and the replica that answers is an AWS node in Frankfurt; a second replica, already warm, is an Azure node in Amsterdam. Same hostname, same SLA, same compliance boundary — two invoices, at once.

Thirty seconds later. The autoscaler needs a third replica; Google Cloud happens to be cheapest for that purchase, so the new node lands there. Nothing about the gateway, the routing policy, or a customer's traffic path changes — the cell just gained a third invoice line.

SECTION 09

09 Why this stack, on this pool

Each advantage here is structural — a consequence of where the platform sits, not a bolt-on.

1 · The whole GPU board in one decision. Quota and stock-outs are per-region facts; vLLM's hardware breadth makes many SKUs qualify; the catalog prices them all across every region of three clouds and buys the cheapest actually available. A single-cluster tool chooses from one region's menu; the pool chooses from the market.

2 · Spot-aware serving, safety net included. Inference replicas are the best-behaved spot citizens in production — stateless, quick to warm, horizontally redundant behind a KV-aware router. The platform supplies what spot actually requires: risk-scored placement, reclaim detection, PDB-respecting drain, cross-region replacement.

3 · Heterogeneity is native, not a spreadsheet. Prefill wants compute; decode wants memory bandwidth; the KV offload tier wants cheap RAM; gateways want none of the above. A benchmarked, accelerator-aware catalog places each pool on the machine shape that suits it — mixing generations and vendors where vLLM qualifies them.

4 · One cluster across clouds — no federation project. Cells in three clouds are pods in one conformant Kubernetes on one flat encrypted network. The multi-cluster fleet, the mesh glue, and the DR runbooks that usually accompany "multi-region inference" simply do not exist here.

5 · Your accounts, your credits, no markup. Machines are bought in dedicated accounts you own, at your rates, stacking your committed-use discounts and credits under the catalog's savings. Managed-inference platforms charge roughly 2× bare GPU price; here the serving layer is open source and the machines are at cost.

6 · The platform around the model is already running. Observability that understands per-machine spot prices, GitOps delivery, SSO, TLS, DNS, secrets — and scale-to-zero (Knative) for the long tail of models that shouldn't hold a GPU while idle. None of it lands on your on-call.

Where this sits in the landscape, honestly: single-cluster optimizers (Karpenter, CAST AI, Spot Ocean) do real spot automation inside one region of one cloud — the boundary GPU scarcity punishes. Managed inference clouds (Modal, Baseten, Together) serve excellently on their infrastructure, at a markup. Client-side brokers (SkyPilot) find cheap GPUs per job and leave the standing platform as your build. An **operated pool, on your own accounts, spanning every region of three clouds, under conformant Kubernetes** is the combination none occupy.

SECTION 10

10 The numbers

One accelerator, four prices — the spread the placement decision harvests. Public list prices for a single H100 80GB, as compiled mid-2026:

OPTION	$/GPU-HOUR
What most teams run — AWS on-demand (p5, us-east-1)	**$6.88**
Same cloud, spot	**$2.53**
Another cloud, on-demand — Azure NC-series	**$1.63**
The whole board — Google Cloud spot (a3, best region)	**$1.35 (−80%)**

> Read the third row twice: **another cloud's on-demand beats your own cloud's spot.** No single-cloud tool, however good, can reach it.

The same mechanics apply to the CPU side of the stack — a benchmarked 16-vCPU/64 GB workload runs **$561**/month as typically bought (on-demand, us-east-1) and **$24–39**/month as the catalog places it, at equal-or-better benchmarked compute (−93% to −96%).

The honesty footnote that belongs under every such table: these are point-in-time list prices — GPU markets move weekly, which is an argument *for* continuous placement, not against it. Spot rows are reclaimable; plan on 50–70% net savings after interruption overhead, not the sticker discount. Your negotiated discounts apply first and narrow the ladder. Cross-cloud rows exclude data movement — the optimizer weighs egress before crossing, and for inference the recurring flows (prompts in, tokens out) are small; model weights replicate once per cell. The live catalog re-prices this ladder continuously — and the Advisor (section 11) computes it for *your* workloads, not a specimen.

SECTION 11

11 Adopting it: Audit → Extend → Serve

The same three phases as every Multicloud engagement, each with standalone payoff and its own small, explicit access grant — mapped here to an inference estate:

1. **Audit** *(read-only, days, zero risk).* Install the Advisor in your existing cluster: read-only (get/list/watch, never Secrets), zero-egress except one anonymous catalog query. You get the counterfactual no single-cloud tool computes — what each workload, GPU replicas included, *would* cost on the optimal instance/region/cloud/pricing-model, performance-normalized, with interactive levers for regions, clouds, and spot. It quantifies; it changes nothing.

2. **Extend** *(early access — start with one variant, stop wherever you like).* Attach the virtual node. Pick the lowest-risk inference tier you have — the async queue, the embedding batch, the eval sweep, an overflow variant of your interactive model — add the burst toleration, and watch cost, latency, and reclaim behavior with your own dashboards. Guardrails (latency budgets, allowed regions, spend caps) are per-workload, and the dial runs 0–100: many teams will keep interactive tiers local and burst the rest; a team that likes its numbers can run its **entire inference estate** through the virtual node (section 05), with no migration event anywhere along the way.

3. **Serve** *(design-partner program).* Stand up serving cells near your users, put the steered hostname in front, and run the failover game-day on your workload — region loss, then provider loss — as a scheduled milestone, not a leap of faith. When you want more than burst, the pool promotes to a sovereign cluster: standard kubeconfig, full admin, operated for you underneath.

> **The savings pay for it.** A flat platform fee, priced under your audited savings — never per-CPU, never a percentage of savings. The audit comes first precisely so the fee is justified by a number you measured.

SECTION 12

12 What's built, what's early, what's roadmap

PIECE	STATUS
Catalog, scheduler, spot-aware placement, reclaim handling, cross-cloud fabric	**In production** (AWS and Google Cloud provisioning live; Azure provisioning in final validation)
GPU-class constraints + accelerator catalog (76 types)	**Built**
Advisor (read-only savings counterfactual)	**Built** — install is a Helm chart
Extend your cluster (virtual node, burst tier, peering at signup)	**Early access** — design-partner phase
DNS latency steering / health-checked geo-failover	Roadmap — the platform already operates the DNS layer end to end; steering policies are an incremental extension being built with design partners
Whole-provider-outage failover	Mechanism built; proving it on your workload is a design-partner game-day, not a claim made in advance
GPU-specialist providers beyond the big three	Roadmap — the provider roster grows; workloads inherit new providers through the existing agreement
vLLM and llm-d	Upstream open source (Apache-2.0), installed unmodified; llm-d is CNCF Sandbox and pre-1.0 — expect its APIs to move

Sources

llm-d: github.com/llm-d/llm-d · llm-d.ai (v0.3 release blog) · Red Hat launch announcement · Google Cloud blog (GKE Inference Gateway in Vertex AI production). vLLM: github.com/vllm-project/vllm · docs.vllm.ai. GPU market: Silicon Data (geography of GPU pricing) · GPU Tracker (cloud GPU statistics 2026) · Introl (spot GPU savings) · SemiAnalysis (the GPU rental market). Economics: a16z (the high cost of AI compute) · HostFleet (serverless GPU pricing matrix). Platform: the Multicloud *Architecture Whitepaper* and docs/external/llm-inference-whitepaper.md (this paper's text source, with live links).

This paper describes a reference architecture on the platform as it stands, with status flags where a capability is early access or roadmap. Ladder prices are point-in-time public list prices — ask us to run the live catalog, or the Advisor on your own cluster, for current numbers on your workloads.

Contact: yaron@multicloud.io · multicloud.io