Cross-Region Model Serving

PyTorch Inference, Observability & LLMOps

Suraj Muraleedharan, Amazon Web Services

Single-Region is Solved; Multi-Region is Hard.

10K+
requests per second per region

3
regions

<200ms
p99 TTFT target

3x
spike absorption

Correlated Failures

Region outages can cascade.
Traffic spikes can hit multiple regions simultaneously

Quality is Invisible

LLMs can provide fast but semantically wrong responses.
Error codes don't catch this.

TorchServe Gap

No PagedAttention, no KV-cache management, no tensor parallelism.
Limited maintenance since 2025.

How We Got Here: PyTorch Serving Evolution

Era 1: 2019–2023
TorchScript + Triton
or custom servers
Static batching, stateless

Era 2: 2020–2025
TorchServe
First official PyTorch serving
LLMs broke every assumption

Era 3: 2023–now
torch.compile + vLLM/Triton
PagedAttention, KV-cache
Purpose-built runtimes

The Key Shift: Serving Split in Two
torch.compile: model optimization at load time
Runtime: batching, KV-cache, streaming, multi-GPU
These two concerns are now decoupled

Today's Stack
vLLM — LLM workloads (PagedAttention)
Triton — Non-LLM models (Vision, NLP, Recommendations)
TGI — HuggingFace-native teams
KServe — Orchestration layer (wraps any backend)

Architecture: Three-Layer Design Per Region

Layer 1: Edge Routing

Global Accelerator → NLB → Envoy L7

Latency-based routing, circuit breakers, traffic splitting

Layer 2: Inference Engine

vLLM/Triton + torch.compile + CUDA Graphs

Continuous batching, PagedAttention, speculative decoding, 8×H100

Layer 3: Post-Processing

Safety classifier, quality scoring (judge model), PII redaction

Cross-Region

Health-aware failover

Geo-fencing compliance

PrivateLink mesh

Low-latency config propagation

Key Insight

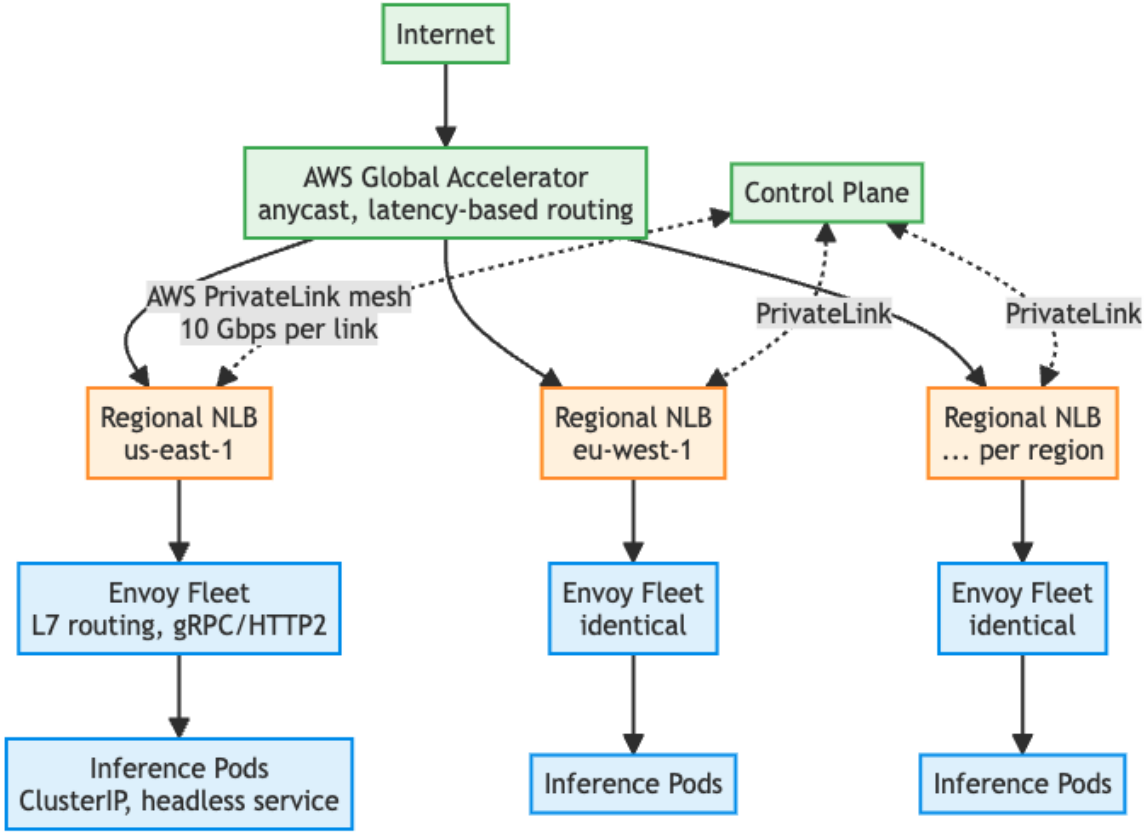
Scale layers independently

Routing: cheap, fast to spawn

GPUs: expensive, slow to spawn

15% warm pool for failover

Network Topology



Pillar 1: Serving — Deployment & Failover

vLLM	Triton	TGI	KServe
LLM workloads	Non-LLM models	HuggingFace-native	Orchestration layer

Progressive Deployment; quality-gated at each stage

Pillar 1: Serving — Deployment & Failover

Circuit Breakers

Envoy outlier detection (per-host)
App-level state machine:
CLOSED → OPEN → HALF_OPEN
Exponential backoff cooldown
Feeds into regional health score

Failover & Rollback

Health-based traffic drain (<30s)
Previous version warm-loaded
Pointer swap rollback (<1s)
N-2 version warm standby
Regional autonomy during partitions

Deployment Strategies

Progressive Canary with quality gates
Gate check at every stage
Auto-rollback on gate failure
Blue-green only for selective cases
since it needs 2x fleet

Pillar 2: Observability — Inference Telemetry

Telemetry Pipeline

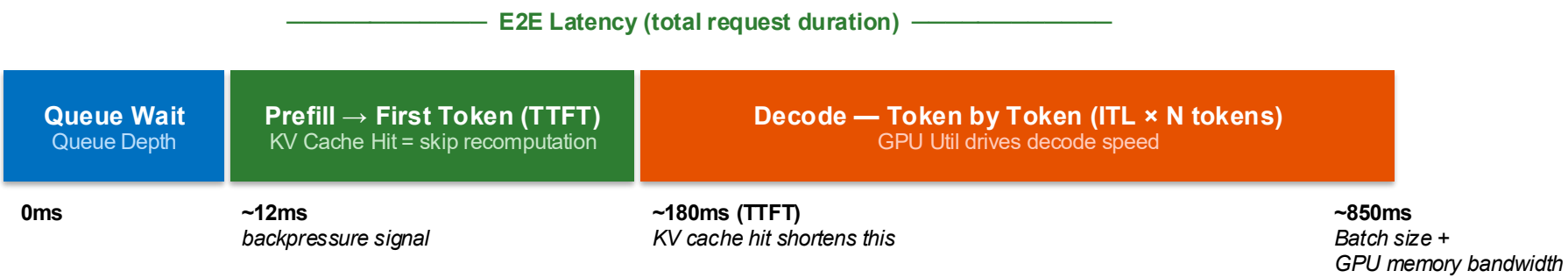
Per-Request Trace Spans

Pillar 2: Observability — The Signals

Time to First Token (TTFT) *How fast we start?*	**Inter-Token Latency (ITL)** *How smooth is the streaming?*	**Latency** *Total request duration*	**Token Throughput** *How much is the work/second?*	**Requests/second** *System capacity*
KV Cache Hit Rate *How much do we reuse?*	**GPU Utilization** *Cost v/s Headroom*	**Request Queue Depth** *Are we drowning?*	**Error Rate** *Are the requests failing?*	**Service Level Objectives (SLOs)** *Are we meeting targets?*

Green = User-facing Orange = Cost/efficiency Blue = System Purple = Reliability

Pillar 2: Observability — Anatomy of a request



Pillar 2: Observability — Dashboards

L1: On-Call

Region health
Request rate + error rate by region
TTFT latency heatmap (200ms SLO)
GPU utilization gauge (65-75%)
Queue depth + preemption rate
Canary vs stable comparison

L2: ML Engineering

Composite quality score (±1σ)
Per-dimension breakdown
Hallucination rate
KL divergence from baseline
Output entropy + finish reasons
User regeneration rate

L3: Cost & Efficiency

Cost per 1K tokens
GPU utilization vs provisioned
Monthly burn rate vs budget
Speculative decode acceptance
KV-cache hit rate
Capacity headroom per region

Pillar 3: LLMOps — Quality and Governance

Observability shows the problem. This pillar closes the loop and acts on it.

1. Drift Detection

Input: PSI, KL divergence, chi-squared

Output: token length, entropy, finish reasons

2. Quality Evaluation

Judge model on 5% sampled traffic

Dimensions scored 0.0-1.0 - Helpfulness, Accuracy, Coherence, Safety

3. Gate and Rollback

Alerts and automated rollbacks based on the quality gates

SLO burn-rate triggers and Sloth format for Prometheus

Open-Source Tooling

Inference
vLLM (LLMs)
Triton (non-LLM)
torch.compile + CUDA Graphs

Orchestration
KServe (Model CRD)
Karpenter (GPU scaling)
Envoy (L7 + circuit breaking)

CI/CD
Argo Workflows (Pipeline)
ArgoCD (GitOps)
MLflow (Model registry)

Observability
OpenTelemetry (Traces and Logs)
Prometheus + Thanos (Metrics)
Grafana + Sloth (SLOs)

Streaming
Apache Kafka (Event stream)
Apache Flink (Anomaly detect)
ClickHouse (Quality store)

Testing
Chaos Mesh (Fault injection)
Locust + k6 (Load testing)
AIPerf (Benchmarking)

Key Takeaways

- **Migrate from TorchServe**; vLLM for LLMs, Triton for everything else, KServe for orchestration.
- **Serving is split into two** – Model optimization (torch.compile) and serving orchestration (batching, KV-cache, streaming) are decoupled.
- **Design observability from day one**. Use custom OTel spans per inference phase, not just HTTP metrics.
- **Quality evaluation is not optional**. LLMs fail silently. You need judge models and drift detection, not just health checks.
- **Progressive deployment** with quality gates at every stage with automated rollback.

Thank you