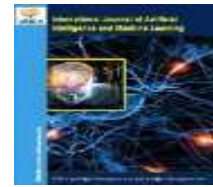




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

Absorbing Generative AI into an Existing ML Platform: An Integration Pattern for a Shared Control Plane

Naveen Kumar Vedurupaka

Independent Researcher, USA. E-mail: naveenkumarvedurupakausa@gmail.com

Abstract

Meeting generative-AI demand by standing up a separate stack beside an existing machine learning platform is a common industry response, but it tends to duplicate governance, observability, registry, lineage, rollout, and cost workflows. This paper reports an architectural pattern that instead extends the existing platform to serve both predictive and generative workloads, through a shared control plane paired with runtime-specific serving lanes. The design is organized as a 2x3 matrix: two model families (classical, generative) crossed with three serving lanes (online, near-real-time, batch). Wire-level contracts shared across all six matrix cells are identified, along with the generative-AI-specific extensions needed for serving, retrieval-augmented generation, and parameter-efficient fine-tuning, and the operational changes required for evaluation, safety, and cost attribution. The contribution claimed is an integration pattern and design recipe rather than a new inference engine, scheduler, or serving algorithm; the underlying serving mechanics, including paged attention, disaggregated prefill and decode, and low-rank adaptation, are adopted from established literature and are not claimed as novel here. A Kubernetes-based reference architecture is described, the small set of net-new control-plane code is separated from the substrate it reuses unmodified, and the evidence basis, trade-offs, and limitations are discussed for platform teams considering a similar transition. Every quantitative claim is labeled by evidence category (design target, illustrative sizing, or extrapolation) since the design has not yet been benchmarked against a production workload.

Keywords: Control Plane; Generative AI Serving; Large Language Model Integration; Machine Learning Platform Architecture; Model Governance.

1. Introduction

1.1 Motivation and Contribution Boundary

Paged attention, continuous batching, token streaming, parameter-efficient adapters, retrieval composition, and gang-admitted multi-node replicas: these runtime mechanics set generative-AI workloads apart from classical predictive models. Faced with that difference, the typical industry response has been to stand up a second platform beside the existing predictive one. The result tends to be duplicated governance, two registries, two lineage stores, two dashboard estates, two on-call rotations, and two compliance audits. This duplication is not obviously justified by the runtime difference alone; it looks more like the path of least resistance for teams working under release pressure.

The alternative reported here absorbs generative AI as runtime-specialized lanes on a control plane that already serves classical models, rather than standing up a parallel one. The architectural frame is a 2x3 inference matrix: two model families (classical, generative) crossed with three serving lanes (online, near-real-time, batch). A shared control plane covers all six cells; the differences live in lane-specific runtimes joined by wire-level contracts.

The contribution claimed here is an integration pattern and design recipe, not a new inference engine, scheduler, or serving algorithm. The runtimes and algorithms it draws on, paged attention [1], continuous batching [2], FlashAttention [3], disaggregated prefill and decode [4], [5], structured generation [6], and LoRA [7] and QLoRA [8] fine-tuning, are adopted from existing literature rather than proposed here. The novelty this paper claims is deliberately bounded to two items: full control-plane absorption across mixed classical and generative-AI workloads (registry, lineage, observability, identity, governance, rollout, and cost attribution shared across all six matrix cells), and an explicit shared-versus-lane-specific contract boundary intended to be reusable as a recipe by other teams.

The paper does not claim to invent LLM serving, paged attention, disaggregation, LoRA serving, cluster scheduling, LLM evaluation, or a unified predictive-and-generative serving runtime. Recent systems have already moved the ecosystem toward unified serving primitives, and Section 1.2 sets the boundary against them explicitly. What may distinguish this paper is a narrower, more operational question: how a mixed-

workload ML platform absorbs generative AI without a rewrite, what stays shared, what specializes by lane, and what evidence supports each part of the pattern.

1.2 Related Work and Novelty Boundary

Training, registry, serving, and observability for classical models are already well covered by industrial ML platforms such as Michelangelo [9], TFX [10], and SageMaker [11]. Most predate widespread LLM serving and do not appear to treat parameter-efficient fine-tuning, paged-attention KV caches, retrieval composition, or LLM evaluation as first-class lanes. Each of these platforms covers roughly one or two cells of the 2x3 matrix described here; the gap this paper tries to address is the integration recipe for the generative-AI column under a shared substrate, not a competing platform.

Public practice is already moving toward unified serving. KServe [12], for instance, now targets both predictive and generative AI, with a dedicated custom resource for multi-node inference, prefill and decode separation, and routing [13]. MLflow [14] covers LLM tracking, prompt engineering, agent observability, and evaluation [15]. AIBrix [16] looks like the closest adjacent system: a cloud-native LLM infrastructure project with unified runtime management, LoRA support, autoscaling, distributed KV cache, and hybrid orchestration. None of these seems to address the retrofitting problem this paper focuses on: extending a mixed classical-plus-generative platform so an existing control plane (registry, lineage, observability, RBAC, rollout, cost) absorbs the generative-AI lane without duplicating it.

The runtime layer for LLM serving is defined in the literature by paged attention [1], continuous batching [2], FlashAttention [3], disaggregated prefill and decode [4], [5], structured generation [6], quantized serving [17], and chunked-prefill scheduling [18]. This paper adopts these runtime and scheduling ideas as given; it does not propose new ones. Parameter-efficient fine-tuning follows LoRA [7] and QLoRA [8], with ZeRO [19] and Megatron-LM [20] treated as escape-hatch frameworks for full-parameter and pipeline-parallel training when needed. Classical serving lineage traces to Clipper [21] and KServe [12], which established the request-batching, multi-version, latency-bounded online-serving pattern that the platform's classical online lane still uses.

Retrieval-augmented generation composes a retriever with a generator [22]; HNSW [23] remains a dominant approximate-nearest-neighbor index and is treated in this design as a typed feature behind the existing feature-store contract. LLM evaluation, safety, and governance draw on HELM [24] as a broad open evaluation framework, RAGAS [25] for RAG faithfulness, answer relevance, and context recall, and PromptBench [26] for adversarial and prompt-injection robustness testing. The NIST AI Risk Management Framework: Generative AI Profile (NIST AI 600-1) [27] supplies the risk taxonomy used for the ethics framing in Section 3.4. Cluster scheduling and workflow orchestration in this design build on Ray [28], Kueue [29], Volcano [30], and LeaderWorkerSet [31], which provide queueing, gang admission, and multi-node serving custom resources; NVIDIA DCGM attributes GPU-second utilization. Wire-level contracts, OpenLineage [32], OpenTelemetry [33], ONNX [34], MLflow [14], SPIFFE [35], OPA [36], and Sigstore Cosign [37], are reported here as load-bearing for the generative-AI lane without modification.

Table 1 summarizes what each line of adjacent work contributes, what this paper adopts unchanged, and what it adds at the control-plane boundary.

Adjacent work	What it contributes	What this paper adopts	What this paper adds
vLLM, Orca, FlashAttention, Splitwise, DistServe, SGLang, TGI [1]–[3], [4]–[6], [17], [18]	LLM runtime and scheduling mechanics	Runtime, paged attention, continuous batching, disaggregation, structured generation	Control-plane absorption (registry, lineage, observability, governance, rollout, cost)
LoRA, QLoRA [7], [8]	Parameter-efficient fine-tuning algorithms	LoRA and QLoRA on the shared training plane	Multi-tenant LoRA serving boundaries and isolation policy by tier
KServe, MLflow current docs [13], [15]	Cross-modal serving and LLM-aware registry primitives	KServe and MLflow as substrates	Cross-lane shared lineage, evaluation, governance, and cost recipe

Adjacent work	What it contributes	What this paper adopts	What this paper adds
AlBrix [16]	GenAI-first cloud-native LLM infrastructure	Comparator, not a primitive consumed here	Retrofitting an existing mixed-workload ML platform rather than building greenfield
Fairness in Serving LLMs [38]	Per-tenant fairness within an LLM runtime	Token-budget policy at the gateway	Cross-lane RBAC and tiered fairness policy on the shared control plane
HELM, RAGAS, PromptBench [24], [25], [26]	LLM evaluation suites	Suites as containerized promotion gates	Lineage-sealed evaluation events via a shared test facet
NIST GenAI Profile [27]	Risk taxonomy and governance framing	Risk vocabulary for ethics framing	Mapping into the platform's gates, audit, and operational controls

TABLE 1. Adjacent work, what each contributes, what this paper adopts unchanged, and the integration boundary this paper adds.

2. Method

2.1 Applicability Envelope and Evidence Basis

The design pattern reported in this paper is grounded in a Kubernetes-based reference architecture, sized for the operating point described below. It is not a benchmark paper and does not report a running deployment. Each quantitative or operational claim is classified into one of three evidence categories, defined in Table 2, and labeled accordingly throughout the paper.

The applicability envelope targets platform teams running an existing classical-model ML substrate on Kubernetes who want to add a generative-AI lane without forking their platform. The reference design targets hundreds of production models across classical and generative-AI families; hundreds of peak queries-per-second of generative-AI inference and hundreds of inference GPUs of capacity; fewer than 100 deep retrains per week and fewer than 10 generative-AI fine-tunes per week; and mixed Critical, Prod, and Dev tiers with multi-region active-active service for Critical workloads.

Section 3.3 separates the small set of net-new control-plane code from the substrate the lane consumes unchanged. Where the design has no measurements, disaggregation throughput gains, governance overhead, end-to-end RAG faithfulness, the paper labels the claim a design target rather than a result. No synthetic benchmark numbers are presented.

TABLE 2. Evidence taxonomy used for claims in this paper.

Category	Definition
Design target	The quoted value is the SLO, capacity plan, or recovery objective the design commits to; it is enforced as a gate or alert.
Illustrative sizing	A worked example, with explicit assumptions, used to communicate orders of magnitude; not a guarantee.
Extrapolation / open work	A claim that exceeds the design's envelope.

2.2 Design Principles for Absorbing Generative AI

Whether a candidate component enters the platform, in what shape, and at what cost, seems to come down to five operational decision rules.

Principle 1: wire-level contracts, not framework wrappers. The platform standardizes at the wire boundary: gRPC and HTTP surfaces, OTLP for telemetry, OpenLineage JSON events, ONNX or safetensors for model artifacts, SPIFFE SVIDs for identity, and Cosign signatures for provenance. The framework itself is not

standardized; Python, JVM, Go, and Rust runtimes can coexist behind the same wire surfaces. Wire-level contracts appear to survive framework version churn better than framework wrappers do.

Principle 2: three reasons to wrap. A component is judged to justify a platform-built wrapper only if it satisfies at least three of the following: cross-team ergonomics, policy enforcement the underlying tool cannot perform on its own, two-tool bridging, governance hooks such as audit, signing, or RBAC, or upgrade decoupling. The genai-gateway, a small platform service placed in front of the LLM runtimes, passes three of these: one OpenAI-compatible surface, per-tenant token budgets, and REST, gRPC, and HTTP bridging (Section 2.4). KServe, MLflow, Ray, and Volcano are exposed unwrapped because each fails this test.

Principle 3: paved path with named escape hatches. Every contract has one paved-path implementation and a small set of named escape hatches. The online generative-AI paved path is vLLM; SGLang [6], for structured generation, and TGI [17], for specific quantization needs, are the named escape hatches. The training paved path is Ray Train on KubeRay; PyTorchJob, MPIJob, DeepSpeed, TorchTitan, and Megatron are escape hatches. The paved path carries the platform's full support contract; escape hatches inherit wire contracts but not the same level of platform ergonomics.

Principle 4: invariants in the control plane, mechanics in the runtime. Invariants are what every model shares regardless of runtime: registry, lineage, observability, RBAC, traffic management, and cost attribution. Mechanics are how a runtime works internally: continuous batching, paged attention, KV-cache layout, tensor and pipeline parallelism, prefill-decode ratios, gang-restart behavior, and disaggregation. The platform adopts mechanics rather than rebuilding them and declines any runtime that would change an invariant.

Principle 5: promotion gates over runtime guards. Every production change passes through the registry promotion gateway, which evaluates Cosign signature verification, model-card schema validation, an evaluation-threshold check, a lineage-completeness check, and, for tier-Critical workloads, human approval. Runtime guards, rate limits, circuit breakers, OPA admission, catch what the gates miss, but the platform invests primarily in gates because they can catch issues before traffic reaches the runtime.

These five principles appear to be why the generative-AI lane absorbs into the platform with one small gateway, one admission webhook, a few evaluation-suite plug-ins, and a metric set, rather than a forked control plane.

2.3 The 2x3 Inference Matrix and the Shared Control Plane

The serving plane decomposes into six cells. Two model families (classical, generative) cross three lanes (online, near-real-time, batch). The matrix is a framing device for isolating runtime mechanics while preserving platform invariants; Table 3 lists the paved-path runtime for each cell.

What differs across cells is the runtime engine and the GPU profile, which depends on model size, context length, KV budget, batching strategy, quantization, and the applicable SLO. As an illustrative sizing exercise (FP16 weights, FP8 KV cache, top-k = 8 RAG retrieval), a 7-billion-parameter model can fit on one H100, a 13-billion-parameter model with tensor parallelism of 2 on two H100s, and a 70-billion-parameter model with tensor parallelism of 4 and pipeline parallelism of 2 across two nodes. These figures do not generalize across quantization schemes or context lengths and are offered only as orders of magnitude.

The scaling mechanism also varies by lane: RPS-driven autoscaling for online, partition-driven throughput for near-real-time, and job-driven parallelism for batch. Latency design targets are under 100 ms for classical online, under 2 seconds end-to-end for generative-AI online with time-to-first-token under 1.5 seconds and time-between-tokens under 80 ms, sub-second for classical near-real-time (or asynchronous via Kafka), and throughput-bound for batch.

What does not differ across cells matters more for the argument this paper makes. Every cell dereferences models from one MLflow registry by a model URI backed by an OCI-compatible object store. Every cell also emits OpenLineage events to one Marquez backend and Iceberg archive, emits OpenTelemetry metrics under an identical name vocabulary, carries the same cost-attribution labels, runs under the same RBAC pipeline at workload identity, and honors the same promotion gates. The matrix functions as a contract: any new runtime takes a cell, inherits the invariants, and contributes only its lane-specific specialization. The near-real-time cells reject the embedded-in-stream-processor pattern teams often reach for first; the paved path is Kafka into a consumer and back to Kafka, which keeps model loading, telemetry, and rollout aligned with the online cells.

TABLE 3. The 2x3 inference matrix. All six cells share the same control plane (registry, lineage, observability, RBAC, traffic management, cost attribution); only the runtime, GPU profile, scaling mechanics, and latency target differ per cell.

Lane	Classical models	Generative AI
Online (request/response)	KServe + NVIDIA Triton [39]; KEDA on RPS; canary and shadow rollout; service-mesh ingress	vLLM behind genai-gateway; paged-attention KV cache; HTTP/2 SSE and gRPC streaming; per-tenant token budgets
Near-real-time (streaming)	Kafka → KServe consumer → Kafka; partition-driven throughput	Kafka → vLLM consumer → Kafka; LLM-as-classifier; asynchronous RAG
Batch (warehouse-scale)	Spark-on-Kubernetes with vectorized UDFs; Iceberg/Delta tables	Ray Data with LLM batch inference; Spark with embedding UDFs

Each control-plane service exposes a contract consumed identically by every matrix cell. The model registry is shared: MLflow [14] holds metadata, signatures, and the artifact URI, while model bytes live in an OCI-compatible object store fetched by the runtime at load time. The registry stays format-neutral: ONNX [34] for classical models, safetensors with an adapter configuration file for LoRA adapters, and safetensors without one for full fine-tunes; PyTorch and TensorFlow SavedModel formats and JAR archives remain first-class as well. The promotion gateway runs the same five gates across every format: Cosign signature check, model-card schema check, evaluation-threshold check, lineage-completeness check, and human approval for tier-Critical. The lineage-completeness gate resolves to one Marquez REST query and is fail-closed: a 5xx response from Marquez causes the promotion to be rejected with an HTTP 503.

Lineage is shared through OpenLineage [32] as the wire-level event schema, with emitters covering Spark, Flink, Argo Workflows, Metaflow, KServe, MLflow, and the genai-gateway. The platform adds four custom facets, each independently versioned: a test/evaluation facet covering test, eval, perf, shadow, chaos, and conformance events with a signed evaluator digest; a cost facet; a scheduling facet; and an erasure-step facet. Marquez backs the live store; a partitioned Iceberg table serves as the long-term archive with seven-year retention for finance-impacting workloads. Per-invocation lineage is intentionally not emitted for generative AI, since token-stream volume and the latency budget make it impractical; per-promotion lineage continues, paired with a per-request audit log on the gateway (Section 2.4).

Observability is shared through OpenTelemetry [33] SDKs as the invariant, with a Grafana-based stack as the paved-path backend and cloud-managed substitutes ingesting the same OTLP path. Every signal carries the same mandatory resource attributes (team, project, tier, workload, model identity and version, region, cluster). The generative-AI lane adds eight LLM-specific metrics, covering token counts, time-to-first-token, time-between-tokens, KV-cache occupancy, admission drops, disaggregated KV transfer, and multi-node readiness, without replacing the shared vocabulary. Identity and RBAC are shared through SPIFFE [35] SVIDs as the workload-identity primitive and a six-tuple RBAC contract (team, project, environment, resource kind, resource name, action). OPA [36] gates promotion, priority elevation, residency-restricted RAG namespaces, multi-tenant LoRA cross-tenant boundaries, and tier-Critical evaluator approval; Cosign [37] signatures are required on container images, model artifacts, evaluator images, and the model card itself.

Traffic management is lane-adapted but contract-shared. KServe [12] serves as the rollout primitive for both classical and generative-AI InferenceService resources, and Argo CD applies the same staged rollout policy (1 to 10 to 50 to 100 percent, with auto-rollback) to both. Batch uses a snapshot-split canary over Iceberg tables; near-real-time uses Kafka partition splits. The mechanism varies by lane, but the contract, declarative, audited, rollback-capable, with the same six operations (candidate bind, canary, shadow, diff, rollback, audit), does not. Cost attribution is likewise shared: OpenCost computes per-workload showback and NVIDIA DCGM attributes GPU-seconds; only the GPU-hours under each workload label changed when the generative-AI lane landed, and a cost facet seals per-run cost into lineage for cross-quarter regression analysis.

2.4 The Generative-AI Online Lane

Runtime specialization is most visible in the generative-AI online lane, and most of the lane-specific new code lives there as well.

vLLM [1] serves as the paved-path online runtime; SGLang [6], for structured generation, and TGI [17], for specific quantization needs, are the named escape hatches. All three expose OpenAI-compatible REST but none offers a platform-native gRPC streaming endpoint, so the genai-gateway translates between OpenAI REST

(client-facing), the platform's own streaming gRPC surface (for SDK clients), and each runtime's HTTP interface upstream. The runtime is interchangeable at the wire boundary; the gateway, registry contract, lineage emitter, and telemetry surface do not need to change when it does.

The lane depends on vLLM's paged-attention KV cache [1] and continuous batching [2]. Page size, GPU-memory utilization, prefix caching, chunked-prefill [18], and FP8 KV cache are tuned per model class. The platform's vLLM image bakes in the OpenTelemetry exporter and a Cosign-verifier init-container, and KV-cache occupancy is exported as a first-class metric with a high-occupancy alert. KV-cache memory scales with layer count, KV-head count, head dimension, and element size, sized per standard vLLM practice [1]; FP8 KV roughly halves memory footprint and so approximately doubles concurrency relative to FP16. Per-model capacity sizing is treated here as a runtime mechanic the platform adopts, not a contribution of this paper.

The genai-gateway is a small Go service in front of vLLM that satisfies the three-reasons-to-wrap test described in Section 2.2: one OpenAI-compatible surface across vLLM, SGLang, and TGI; per-tenant token-budget enforcement that vLLM and the service mesh cannot perform alone; and bridging across REST, gRPC, and HTTP. It keeps a per-team, per-project token budget in Redis, decremented as response chunks arrive; exceeding the budget returns an HTTP 429 and emits an admission-drop metric tagged with the budget reason. Budgets are advisory for tier-Critical and binding for Prod and Dev. Time-to-first-token, time-between-tokens, and token counts are measured at the gateway. Requests are signed with the gateway's SPIFFE SVID, and non-gateway cross-namespace calls are denied at the mesh.

Workloads can opt in to disaggregated prefill and decode pools when single-pool time-to-first-token or time-between-tokens at the 99th percentile sustains above budget at high resource utilization. The topology adopts the prefill-decode split described by Splitwise [4] and DistServe [5]; the platform's own contribution is not the topology itself but its integration with the gateway and the shared control plane. KV-cache blocks transfer over RDMA via NVIDIA NIXL. The connector is plugin-selectable: NIXL is the paved path, with LMCACHE, P2pNCCL, and Mooncake as named escape hatches. As a starting heuristic, the prefill-to-decode ratio runs roughly 1 to 3 for short-prompt chat and shifts toward decode for long-context RAG. The throughput gains reported in the originating literature [4], [5] motivate this design choice; production-realized gains have not yet been measured here and are labeled a design target rather than a result.

Models above roughly 70 billion parameters span multiple nodes via LeaderWorkerSet [31]. A leader pod runs vLLM's coordinator process; the remaining worker pods join the leader. The whole group is gang-admitted by a Volcano [30] PodGroup with topology constraints for NVLink islands, RDMA fabric, and expert parallelism honored at admission time. Gang-restart on worker-pod loss is automatic: vLLM's distributed inference cannot resume a sharded execution with a missing worker, so the whole replica restarts and sibling replicas absorb traffic in the interim. Tier-Critical workloads run at least two LeaderWorkerSet replicas behind one InferenceService to bound the blast radius of a single gang restart.

LoRA adapters share a base model: one vLLM deployment loads the base once and merges per-request adapters at serving time. The client selects the adapter through the standard OpenAI model field using a base-model, adapter-name, adapter-version convention, so adapter selection needs no new API surface. For tier-Dev and tier-Prod, a shared base appears acceptable: one base model serving many adapters is efficient enough to outweigh the residual risk in most cases. That risk is real. Adapters share KV-cache pages, scheduler queue positions, and a runtime process, so a misbehaving or hostile adapter could lengthen sibling tail latency, exhaust shared KV pages, or leak timing-channel signals. For tier-Critical, OPA forces tenant-dedicated deployments per LoRA adapter, which reduces the risk to process-level isolation; cross-tenant observation through metric labels or audit logs remains possible in principle and is managed through RBAC rather than eliminated. A per-tenant LoRA cache with strict request-path isolation that preserves shared-base efficiency is left as open work.

2.5 Retrieval-Augmented Generation as Composition

Rather than a new lane, RAG is treated in this design as composition over two existing platform services. A retrieval-augmented endpoint differs from a non-RAG endpoint by one annotation on the model card naming its RAG namespace. The gateway issues a vector query, assembles the prompt with citations, and streams the vLLM response.

The vector index sits behind the same feature-store contract the classical lanes already consume, using HNSW [23] approximate-nearest-neighbor search with optional metadata filters. Treating the index as a typed feature lets the existing feature-store governance, per-namespace access control, PII tagging, and residency constraints, apply without modification. Embedding generation follows two paths: a low-latency, co-located CPU embedding model in the gateway pod (a sub-50-ms p99 design target) and a higher-flexibility, separately served embedding model (a sub-90-ms p99 design target), chosen per RAG namespace. Embedding models register in MLflow with a safetensors or ONNX flavor and ride the same promotion gates as any other model.

Hybrid retrieval combines vector and lexical (BM25) scores, and re-ranking applies a cross-encoder over top-k candidates before prompt assembly; both are opt-in per namespace. The cross-encoder is itself a registered MLflow model with its own promotion gates. Per-invocation lineage is not emitted for RAG requests, for the same volume and latency reasons as the online lane generally, but the per-promotion OpenLineage run ties the base-model artifact, adapter, vector namespace, embedding model, prompt-template version, and re-ranker into one DAG. The per-request audit log records fields such as request identity, workload attribution, retrieval namespace and configuration, retrieved passage identifiers and scores, embedding-model and prompt-template provenance, token accounting, and latency, which together answer which passages a given model retrieved without needing per-invocation lineage volume.

2.6 Fine-Tuning on the Shared Training Plane

No modification to the existing training plane is needed for fine-tuning; it rides that plane as-is. The paved-path training engine is Ray Train [28] on KubeRay, gang-scheduled by Volcano [30], with mandatory checkpointing for preemption recovery and parent-child MLflow runs nested per trial. Escape hatches, Kubeflow Training Operator, PyTorchJob, MPIJob, DeepSpeed [19], TorchTitan, and Megatron [20], apply when a workload exceeds Ray Train's reach.

LoRA [7] fine-tunes use standard PEFT and transformers libraries inside a TorchTrainer; the base model loads from MLflow as safetensors and the adapter is the registered artifact. A tuning sweep over rank, learning rate, and target modules uses early-stopping schedulers, again nesting parent-child MLflow runs per trial. QLoRA [8] and full-parameter fine-tunes at 70 billion parameters or above use MPIJob plus DeepSpeed ZeRO-3 [19] with offload; Megatron [20] is the escape hatch when ZeRO-3 proves insufficient.

Fine-tune output registers with a safetensors flavor plus an adapter configuration file for LoRA and QLoRA, or safetensors alone for full fine-tunes. The model card declares the fine-tune's intent, naming the base model, dataset URI, adapter rank, and training-run URI, and lineage links the base model, dataset, hyperparameters, and adapter as one DAG. Serving binds the adapter at vLLM load time; the gating, lineage-completeness, and Cosign-signature checks are identical to the classical-side gates.

Every training run calls a checkpoint API at framework-defined boundaries, enforced by a CI static check, with a layout kept uniform across Ray Train, Lightning, Accelerate, and DeepSpeed. Scheduling fairness for fine-tunes follows the same contract as classical training: Kueue [29] queues, Volcano gang admission, tier-aware priority, and per-tenant weighted fair-share over GPU-seconds on the shared fabric, with no new scheduling primitive added for fine-tunes. The detailed scheduling design for gang admission and GPU fairness across mixed classical and generative-AI workloads is reported separately in the author's related unpublished work [40].

2.7 Generative AI in the Near-Real-Time and Batch Lanes

Beyond online, the generative-AI lane occupies two further cells: near-real-time and batch. Neither seems to need new control-plane code to absorb generative AI. The platform-supplied consumer image is the only net-new artifact for near-real-time; the Ray Data integration path is shared with the classical batch cell.

Near-real-time generative AI runs Kafka into a vLLM consumer and back to Kafka. The consumer calls the same vLLM InferenceService the online cell uses via gRPC and publishes results with an echoed event identifier. Use cases include LLM-as-classifier work, such as asynchronous moderation, intent labeling, and routing, and asynchronous RAG. Model-loading lifecycle, OpenTelemetry metrics, and the rollout primitive are inherited from the online cell. The consumer commits Kafka offsets only after the broker acknowledges the output, giving at-least-once delivery; a crash before commit reprocesses the event. Each output carries a deterministic identifier derived from model identity, version, event identifier, and prompt-template hash, so a reprocessed event produces a duplicate identifier that downstream consumers can drop, giving effectively-once semantics in practice. Exactly-once delivery via Kafka transactions is not in the paved path; teams that need it can use a transactional Kafka-Connect path as an escape hatch. The same six traffic operations described in Section 2.3 apply over Kafka, adapted to partition groups and mirrored shadow topics.

Batch generative AI uses Ray Data [28] for the GPU-heavy path (LLM batch inference, embedding refresh, content tagging) and Spark-on-Kubernetes with embedding UDFs for the CPU-heavy path; Ray Data reads and writes Iceberg and Delta tables natively, loading the model once per executor. Continuous batching [2] appears to saturate the GPU more effectively under throughput-bound work, so the batch scorecard pins tokens-per-GPU-hour as its central metric. Batch's order-of-magnitude throughput advantage over online serving follows from its relaxed per-prompt latency; this paper does not quote a specific multiplier without a measured baseline. Batch rollouts use branch-based versioning over Iceberg tables, so a new model version writes to its own snapshot without touching the live one, and the same six traffic operations map onto branches instead of partitions.

3. Results And Discussion

3.1 Evaluation, Safety, and Governance

Rather than replacing the classical-side promotion gates described in Section 2.3, LLM evaluation extends them. The substrate is the registry promotion gateway and the OPA policy bundle, both of which already existed for classical models; what is new is a set of LLM-specific evaluation suites and one additional policy class.

The evaluation-suite registry holds versioned evaluation harnesses signed with Cosign. Each suite runs at promotion time as a containerized job and contributes a pass, fail, or waived status against a model-card-declared threshold. HELM [24] backs a hallucination and factuality suite required from tier-Prod upward; RAGAS [25] backs three RAG-specific suites, faithfulness, answer relevance, and context recall, mandatory for RAG endpoints from tier-Prod upward, with faithfulness and relevance disallowed from waiver; PromptBench [26] plus an internal attack corpus backs prompt-injection and jailbreak suites, both disallowed from waiver at tier-Prod and above. Internal suites cover toxicity, with rare waivers allowed, and a tier-Critical fairness suite that hard-gates with no waiver path and a two-person fairness-approver review. An internal performance harness checks time-to-first-token, time-between-tokens, and error rate at tier-Prod and above with no waiver option.

The erasure-step lineage facet introduced in Section 2.3 records propagation steps for right-to-be-forgotten requests. Erasure appears achievable for datasets in the lakehouse, derived features in the feature store, vectors in the RAG namespace, fine-tune adapters, training-run artifacts, request and audit logs, evaluation-suite artifacts, and lineage events. Erasure cannot prove removal from a foundation model's parameter behavior without retraining; for foundation models the platform hosts, retraining is the propagation path, and for models consumed through a third-party API, the platform cannot make that guarantee. Erasure scope is declared in the model card and audited against the erasure-step facet.

A small, tier-configurable fraction of production requests is mirrored into an offline pipeline that re-runs the evaluation suites against live output and retrieved context; sustained regressions trigger a model-version freeze. Results write into the same test-facet contract used at promotion, adding no new lineage surface. This continuous-evaluation pipeline is part of the design, though its sensitivity at scale has not yet been measured. Red-team exercises run PromptBench [26] and human-curated attack corpora against tier-Critical evaluators, with findings filed as signed lineage events and recurring patterns fed back into the canonical suites.

3.2 Performance, Operations, and Failure Modes

Five service-level indicators govern the generative-AI lane, all built from the online cell's metric vocabulary. Every one of them is currently a design target rather than a measured production result: time-to-first-token p99 (target 1.5 s), time-between-tokens p99 (target 80 ms), end-to-end RAG p99 (target 2 s), availability (target 99.9% at Prod), and token-budget rejection rate (target under 1%). Tier-aware error budgets scale from 99.99% availability at tier-Critical to 99.9% at Prod and 99% at Dev, and the error budget governs deployment cadence: if more than half the monthly budget is spent in the first half of the month, non-fix deployments to that tier freeze.

Cost-per-token attribution divides OpenCost's per-workload cost by the output-token rate to yield a cost-per-million-output-tokens figure, shown on every generative-AI dashboard, with per-model alerts when the figure drifts week over week without a corresponding scale change. Online replicas autoscale on GPU utilization and queue depth relative to target depth. Cold start is dominated by weight loading, tens of seconds for a mid-size model and a few minutes for a 70-billion-parameter model on a warm image cache; tier-Critical replicas are pre-warmed in the secondary region so failover can hold a one-hour recovery-time objective.

Several failure modes recur in the design and are addressed with specific detection signals and recovery paths. Worker-pod loss in a multi-node group triggers a whole-gang restart, with a target recovery window of a few minutes and a blast radius of one replica, or a full outage if only one replica is running. KV-cache exhaustion is detected through occupancy metrics and recovered by evicting low-priority generations and shedding best-effort traffic at the gateway, at the cost of increased tail latency rather than replica loss. Prefill-pool starvation under disaggregation triggers autoscaling of the prefill pool and, if the prefill SLO is breached for a sustained window, a fallback to non-disaggregated serving. GPU out-of-memory conditions trigger a pod restart and traffic shift. Gateway crashes are absorbed by the service mesh shifting traffic to healthy replicas, with automatic rollback on error-rate correlation. Regional failover shifts traffic to a warm secondary region within the recovery-time objective. Fabric partition on the RDMA transport used for disaggregated KV transfer triggers a fallback to non-disaggregated serving until the fabric recovers. Cosign verification failure at pod start keeps the pod in an init state and triggers a registry rollback, limiting the blast radius to the affected model version.

vLLM is pinned to one minor version per region at a time; upgrades roll one model at a time through canary rollout, with auto-rollback on time-to-first-token regression or a shadow-diff above tolerance.

LeaderWorkerSet controller upgrades land in two waves (controller, then rolling replica update), and tier-Critical workloads require redundant replicas before any upgrade that touches the LeaderWorkerSet layer. A chaos-drill plan repeatedly kills a multi-node worker pod over a sustained run, with pass criteria requiring the gang to restart within its recovery target, siblings to absorb traffic, and error-budget burn to remain within budget; a quarterly regional drill drains the primary region against the regional recovery-time objective. These drills are specified in the reference design; no production-incident measurements are reported here.

3.3 Net-New Code to Absorb the GenAI Lane

As specified, the architecture is a Kubernetes-based reference design built from the named substrate components above. This section reports which parts of the existing control plane absorb the generative-AI lane unchanged and which parts are net-new. The list is an inventory of a design, not a measured code count from a shipped system.

Every shared service described in Section 2.3 appears to work on vLLM-served models unchanged: artifacts register through unmodified MLflow, promotion gates execute the same way they do for classical runtimes, lineage lands in the same Marquez and Iceberg archive, OpenTelemetry metrics land in the same dashboard stack, cost labels flow to OpenCost and DCGM identically, and the RBAC pipeline applies the same six-tuple contract. None of these paths needed modification to add the generative-AI lane.

Table 4 enumerates the net-new platform components, the test and evaluation assets, and the substrate consumed unchanged. The headline observation is that absorbing the generative-AI lane appears to need only a small set of in-house platform components, a gateway, an admission webhook, a Kafka consumer image, a metric set, and one OPA policy class, layered on a large substrate reused without modification. Each net-new component's design state is also tracked: the Helm charts for the serving stack, the per-contract conformance tests, and the performance-harness profile are specified but not yet released or run in production, and the evaluation-suite containers are specified but not yet built. The reference design topology is documented in this paper and in the author's separate scheduling work [40].

TABLE 4. Components introduced or consumed for the GenAI lane. Net-new platform code and test/evaluation assets are small; the substrate the lane reuses unmodified is large.

Component	New / Reused	Category
genai-gateway (Go service)	New	Platform control-plane code
LeaderWorkerSet-aware mutating admission webhook	New	Platform control-plane code
NRT Kafka consumer image for vLLM	New	Platform control-plane code
Eight LLM-specific metric and dashboard definitions	New	Platform observability configuration
One OPA policy class (tier-Critical evaluator fairness review)	New	Platform policy code
One perf-harness profile (k6, 24-hour soak)	New; specified, not yet run	Platform CI/test asset
Evaluator container images (RAGAS, HELM, PromptBench wrappers, internal toxicity/fairness/perf)	New; specified, not yet built	Test/evaluation asset (not control-plane code)
MLflow (registry)	Reused unmodified	Substrate consumed
Marquez + OpenLineage collectors	Reused unmodified	Substrate consumed
OpenTelemetry collectors + Grafana-based stack	Reused unmodified	Substrate consumed

Component	New / Reused	Category
KServe operators + service mesh + Argo CD	Reused unmodified	Substrate consumed
OPA bundles (existing classical policy)	Reused unmodified	Substrate consumed
OpenCost + DCGM	Reused unmodified	Substrate consumed

3.4 Ethics, Privacy, and Operational Risk

Risk categories in the NIST AI Risk Management Framework: Generative AI Profile (AI 600-1) [27] map onto the platform's operational controls. This mapping is operational rather than philosophical: each item below ties to a gate, a contract, or an audit surface already described earlier in this paper.

Fairness gates measure per-segment disparity against sub-populations declared in the model card and fail beyond a declared threshold. Tier-Critical fairness gates are hard, with no waiver and a two-person fairness-approver review; tier-Prod gates accept compensating-control waivers, themselves logged for audit. The audit log records request identity, tenant attribution, retrieval and prompt-template provenance, and token accounting, but not prompt or response text by default; text logging can be enabled per RAG namespace for debugging under retention bounds and OPA-gated access, and is never enabled for tenants under residency or PII restrictions. Prompt-derived embeddings are not stored.

Shared-base multi-tenant LoRA serving carries a cross-tenant isolation risk at tier Prod and below, addressed at tier-Critical through tenant-dedicated replicas at the cost of GPU efficiency, as described in Section 2.4. Each RAG namespace is tagged with a PII class and data-residency constraints, and OPA denies retrieval against a residency-restricted namespace for callers outside the permitted region; embedding-model provenance and index version are sealed into the audit log so a residency violation can be reconstructed after the fact. Right-to-be-forgotten scope and its limits are covered in Section 3.1; the boundary case, no parameter-level forgetting for third-party API models without retraining, is worth restating here as a genuine limitation rather than a solved problem.

Tier-Critical dedicated replicas and warm secondary-region capacity carry GPU-hour overhead, captured in cost facets and shown in per-team showback. This paper does not claim an environmental optimum; it aims only to make the cost queryable so that teams can weigh availability against carbon and dollar cost at the point where they choose a tier. The mapping from platform gates and audits to the NIST GenAI risk taxonomy [27], hallucination to the RAGAS and HELM gates, prompt injection and jailbreak to PromptBench, bias to the fairness gate, privacy to audit-log redaction and the erasure facet, is not exhaustive; gaps relative to the full taxonomy, including human-AI configuration risks for agentic workloads, are left as open work.

3.5 Lessons, Threats to Validity, and Limitations

Wherever contracts were genuinely wire-level, the shared-control-plane stance held. OpenLineage, OpenTelemetry, MLflow's format-neutral registry, OPA policy, SPIFFE workload identity, and Cosign signatures each appear to survive contact with vLLM, paged attention, continuous batching, token streaming, LoRA adapters, multi-tenant adapter serving, RAG composition, disaggregated topology, and multi-node gang admission without alteration. Any new runtime that emits OpenTelemetry, registers a safetensors-or-ONNX artifact, accepts a SPIFFE-authenticated request, and runs under a Cosign-verified image appears able to integrate as another matrix cell under this design.

In at least five places, though, runtime mechanics broke classical assumptions and the stance strained. First: GPU memory pressure on shared serving nodes does not behave like CPU memory pressure on classical nodes. KV-cache eviction under high concurrency degrades time-between-tokens before any pod-level signal moves. That is why KV-cache occupancy became a first-class signal, with admission-time shedding of best-effort work once occupancy runs high. LLM evaluation semantics diverge from classical metrics: faithfulness, hallucination, prompt injection, jailbreak, toxicity, and relevance are not single-scalar thresholds in the way classical AUC or precision-recall metrics are; each suite scores along its own dimension against a human-rated reference set, and the gateway still treats them as gates, but the definitions live in the evaluation-suite registry rather than the gateway itself. Gang-restart on a single worker-pod loss is effectively a whole-replica outage for tier-Critical workloads, so redundant replicas are required before any upgrade touching the multi-node layer. Disaggregated serving appeared to pay off only above the break-even point described in Section 2.4; below it, the operational cost of running separate prefill and decode pools, ratio tuning, fabric monitoring, dual-route

shadow during topology changes, seemed to outweigh the latency gain, while gains above the break-even point are not yet measured and follow the originating literature [4], [5]. Multi-tenant LoRA serving sits uncomfortably at tier-Critical: tenant-dedicated deployments are the cheapest defensible answer available today, while a per-tenant LoRA cache with isolated request paths that preserves shared-base efficiency remains open work [42], [41].

Several threats to validity limit how far this pattern should be expected to generalize. The reference design targets the operating point described in Section 2.1, hundreds of production models, hundreds of peak generative-AI QPS, and hundreds of inference GPUs, with a specific latency and tier mix; platforms an order of magnitude larger or smaller, or with very different prompt-length distributions, may see different operational trade-offs [43]. The sizing figures this paper does offer, KV-cache scaling, capacity, and the disaggregation break-even point, assume Llama-class architectures with grouped-query attention on H100-class hardware; FP4-quantized serving and runtimes other than vLLM may exhibit different KV economics and scheduling trade-offs. The evaluation suites named are widely used but do not constitute a complete safety surface; multi-turn jailbreak chains, agentic tool-use attacks, and indirect prompt injection via retrieved content likely require additions to the suite registry that remain open work. Multi-node serving depends on NVLink-island and RDMA-fabric topology, and cloud and on-premises operators differ in how they expose these primitives, which affects gang-admission semantics in ways this paper does not fully characterize [44]. Finally, every quantitative claim in this paper is a design target or an illustrative sizing figure rather than a measured production outcome; a future paper is expected to report measured numbers against a public benchmark suite [45].

Models above 70 billion parameters use multi-node parallelism in this design; expert-parallel mixture-of-experts architectures above roughly 200 billion parameters would extend past the design's envelope and should be read as extrapolation [46]. Long-context workloads above 128K tokens appear to saturate the KV-transfer fabric earlier than the throughput model predicts. Streaming-only workloads such as audio or video do not fit the request-response gateway as described; multimodal support is integrated only for image-in, text-out use cases, and agentic long-horizon tool use would need a sessions-and-callbacks extension that is not part of this design [47].

Conclusion

A parallel platform does not appear to be a requirement for extending an existing ML platform to serve generative AI. Organizing serving as a 2x3 matrix, classical and generative model families across online, near-real-time, and batch lanes, lets the whole control plane stay shared, with net-new code confined to a small gateway, an admission webhook, a Kafka consumer image, an LLM metric set, and one OPA policy class. The runtimes, schedulers, and evaluation suites this design depends on are adopted from established literature rather than proposed here.

A useful test for any new runtime candidate may be whether it can speak the contracts the platform already speaks: OpenTelemetry, a safetensors-or-ONNX artifact in a format-neutral registry, a SPIFFE-authenticated request, a Cosign-signed image, and an OpenLineage event at lifecycle boundaries. Runtimes that meet this test can enter as another matrix cell; those that demand framework-level coupling with the control plane may not be worth absorbing this way. The contribution claimed is this integration recipe rather than a runtime or scheduler novelty, and a measured-benchmark follow-up against production traffic remains open work.

ACKNOWLEDGMENTS

The views expressed are the author's own and do not necessarily represent the position of any employer.

AUTHOR CONTRIBUTIONS

Naveen Kumar Vedurupaka is the sole author of this paper and is responsible for its conception, design, analysis, and writing.

CONFLICTS OF INTEREST

The author declares no conflicts of interest.

References

- [1] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," in Proc. 29th Symp. Operating Systems Principles (SOSP), 2023, pp. 611–626. Available: <https://arxiv.org/pdf/2309.06180>

- [2] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in Proc. 16th USENIX Symp. Operating Systems Design and Implementation (OSDI), 2022, pp. 521–538. Available: <https://www.usenix.org/conference/osdi22/presentation/yu>
- [3] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," in Proc. Conf. Neural Information Processing Systems (NeurIPS), 2022. Available: <https://arxiv.org/pdf/2205.14135>
- [4] P. Patel et al., "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in Proc. ACM/IEEE Int. Symp. Computer Architecture (ISCA), 2024. Available: <https://arxiv.org/pdf/2311.18677>
- [5] Y. Zhong et al., "DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving," in Proc. 18th USENIX Symp. Operating Systems Design and Implementation (OSDI), 2024, pp. 193–210. Available: <https://arxiv.org/pdf/2401.09670>
- [6] L. Zheng et al., "SGLang: Efficient Execution of Structured Language Model Programs," in Proc. Conf. Neural Information Processing Systems (NeurIPS), 2024. Available: <https://arxiv.org/pdf/2312.07104>
- [7] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in Proc. Int. Conf. Learning Representations (ICLR), 2022. Available: <https://arxiv.org/pdf/2106.09685>
- [8] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in Proc. Conf. Neural Information Processing Systems (NeurIPS), 2023. Available: <https://arxiv.org/pdf/2305.14314>
- [9] J. Hermann and M. Del Balso, "Meet Michelangelo: Uber's Machine Learning Platform," Uber Engineering Blog, 2017. [Online]. Available: <https://www.uber.com/in/en/blog/michelangelo-machine-learning-platform/>
- [10] D. Baylor et al., "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform," in Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD), 2017, pp. 1387–1395. Available: <https://dl.acm.org/doi/10.1145/3097983.3098021>
- [11] E. Liberty et al., "Elastic Machine Learning Algorithms in Amazon SageMaker," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2020, pp. 731–737. Available: <https://ssc.io/pdf/modin711s.pdf>
- [12] KServe Project, "KServe: A Standard Cloud-Native Model Inference Platform on Kubernetes," Cloud Native Computing Foundation, 2024. [Online]. Available: <https://kserve.github.io/website/>
- [13] KServe Project, "LLMInferenceService and Generative AI on KServe," KServe Documentation, 2024–2025. [Online]. Available: <https://kserve.github.io/website/latest/>
- [14] M. Zaharia et al., "Accelerating the Machine Learning Lifecycle with MLflow," IEEE Data Eng. Bull., vol. 41, no. 4, pp. 39–45, 2018. Available: https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf
- [15] MLflow Project, "MLflow for LLMs, Agents, and Evaluation," MLflow Documentation, 2024–2025. [Online]. Available: <https://mlflow.org/docs/latest/llms/index.html>
- [16] AIBrix Team, "AIBrix: Cost-Effective and Scalable Kubernetes Control Plane for vLLM," ByteDance, 2025. [Online]. Available: <https://github.com/vllm-project/aibrix>
- [17] Hugging Face, "Text Generation Inference," 2024. [Online]. Available: <https://github.com/huggingface/text-generation-inference>
- [18] A. Agrawal et al., "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," in Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI), 2024. Available: <https://arxiv.org/pdf/2403.02310>
- [19] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," in Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC), 2020. Available: <https://arxiv.org/pdf/1910.02054>
- [20] M. Shoeybi et al., "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," arXiv preprint arXiv:1909.08053, 2019. Available: <https://arxiv.org/pdf/1909.08053>
- [21] D. Crankshaw et al., "Clipper: A Low-Latency Online Prediction Serving System," in Proc. 14th USENIX Symp. Networked Systems Design and Implementation (NSDI), 2017, pp. 613–627. Available: <https://dl.acm.org/doi/10.5555/3154630.3154681>
- [22] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Proc. Conf. Neural Information Processing Systems (NeurIPS), 2020. Available: <https://arxiv.org/pdf/2005.11401>
- [23] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," IEEE Trans. Pattern Anal. Mach. Intell., vol. 42, no. 4, pp. 824–836, 2020. Available: <https://arxiv.org/pdf/1603.09320>

- [24] P. Liang et al., "Holistic Evaluation of Language Models," *Trans. Mach. Learn. Res.*, 2023. Available: <https://arxiv.org/pdf/2211.09110>
- [25] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "RAGAS: Automated Evaluation of Retrieval Augmented Generation," in *Proc. Conf. European Chapter of the Association for Computational Linguistics (EACL), System Demonstrations*, 2024, pp. 150–158. Available: <https://aclanthology.org/2024.eacl-demo.16.pdf>
- [26] K. Zhu et al., "PromptBench: A Unified Library for Evaluation of Large Language Models," *J. Mach. Learn. Res.*, vol. 25, 2024. Available: <https://arxiv.org/pdf/2312.07910>
- [27] Chloe Autoio, et al., "Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile," National Institute of Standards and Technology, 2024. [Online]. Available: <https://www.nist.gov/publications/artificial-intelligence-risk-management-framework-generative-artificial-intelligence>
- [28] P. Moritz et al., "Ray: A Distributed Framework for Emerging AI Applications," in *Proc. 13th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 2018, pp. 561–577. Available: <https://arxiv.org/pdf/1712.05889>
- [29] kubernetes-sigs, "Kueue: Kubernetes-Native Job Queueing," 2024. [Online]. Available: <https://kueue.sigs.k8s.io/>
- [30] Volcano Project, "Volcano: Cloud-Native Batch Scheduling System," Cloud Native Computing Foundation, 2024. [Online]. Available: <https://volcano.sh/>
- [31] kubernetes-sigs, "LeaderWorkerSet API," 2024. [Online]. Available: <https://lws.sigs.k8s.io/>
- [32] OpenLineage Project, "OpenLineage Specification," Linux Foundation, 2024. [Online]. Available: <https://openlineage.io/>
- [33] OpenTelemetry Project, "OpenTelemetry Specification," Cloud Native Computing Foundation, 2024. [Online]. Available: <https://opentelemetry.io/>
- [34] ONNX Project, "Open Neural Network Exchange," Linux Foundation, 2024. [Online]. Available: <https://onnx.ai/>
- [35] SPIFFE Project, "Secure Production Identity Framework for Everyone," Cloud Native Computing Foundation, 2024. [Online]. Available: <https://spiffe.io/>
- [36] T. Sandall, "Open Policy Agent: A General-Purpose Policy Engine," Cloud Native Computing Foundation, 2024. [Online]. Available: <https://www.openpolicyagent.org/>
- [37] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software Signing for Everybody," in *Proc. 29th ACM Conf. Computer and Communications Security (CCS)*, 2022, pp. 2353–2367. Available: <https://dl.acm.org/doi/10.1145/3548606.3560596>
- [38] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in Serving Large Language Models," in *Proc. 18th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 2024. Available: <https://arxiv.org/pdf/2401.00588>
- [39] NVIDIA Corporation, "Triton Inference Server," 2024. [Online]. Available: <https://github.com/triton-inference-server/server>
- [40] N. K. Vedurupaka, "Multi-Tenant Gang Scheduling and GPU Fairness for an ML Platform with Mixed Classical and Generative-AI Workloads," unpublished manuscript, 2026.
- [41] H. Gaggar, "MemGov: Policy-Governed Memory Lifecycle Management for Enterprise LLM Agents," *Journal of Intelligent Decision Making and Information Science*, vol. 3, no. 11s, pp. 2199–2214, 2026.
- [42] C. T. Bobba, A. G. Vijay, V. Aher, J. Singh, A. Kejriwal, K. Srinivasan, and A. Ranjan, "Radiographic Evaluation of Periapical Healing Following Single vs Multiple Visit Root Canal Treatment: A PAI Based Study," *Journal of Pharmacy & Bioallied Sciences*, vol. 18, Suppl. 1, p. S206, 2026.
- [43] K. Rekulapalli, H. Nilagiri, and M. Vakkanthula, "A Comparative Evaluation of National FHIR Implementation Guides: Strategic Methodologies for Enhancing Interoperability, Regulatory Alignment, and Clinical Comprehensiveness in Global Health IT," *Health Systems*, pp. 1–13, 2026. Available: <https://doi.org/10.1080/20476965.2026.2728621>
- [44] A. Bhatnagar and V. R. K. Bonagiri, "Operational Excellence through Cloud-Native Distributed Systems," *International Journal of Computational and Experimental Science and Engineering*, vol. 12, no. 2, 2026. Available: <https://doi.org/10.22399/ijcesen.5207>
- [45] H. Gaggar, "Architecting Scalable Enterprise AI Agent Infrastructure for Distributed Intelligent Systems," *International Journal of Advances in Signal and Image Sciences*, pp. 193–208, 2025. Available: <https://doi.org/10.29284/da9w0117>
- [46] C. T. Bobba, "Advancements in Cosmetic Dentistry," *Innovation*, vol. 1, p. 2.

- [47] H. Gaggar, "PromptDLP: Deterministic Pre-Inference Egress Control for Preventing Secret and PII Leakage in LLM Agents," *Journal of Information Systems Engineering and Management*, vol. 9, no. 1, 2024. Available: https://www.jisem-journal.com/download/59_JISEM_9_1.pdf