Securing the Agent: Vendor-Neutral, Multitenant Enterprise Retrieval and Tool Use

Varsha Prasad
Red Hat AI
Boston, USA
vnarsing@redhat.com

Francisco Javier Arceo
Red Hat AI
Boston, USA
farceo@redhat.com

Abstract

Retrieval-Augmented Generation (RAG) and agentic AI systems are increasingly prevalent in enterprise AI deployments. However, real enterprise environments introduce challenges largely absent from academic treatments and consumer-facing APIs: multiple tenants with heterogeneous data, strict access-control requirements, regulatory compliance, and cost pressures that demand shared infrastructure.

A fundamental problem underlies existing RAG architectures in these settings: dense retrieval ranks documents by semantic similarity, not by authorization—so a query from one tenant can surface another tenant's confidential data simply because it is the nearest neighbor. We formalize this gap and analyze additional failure modes—including tool-mediated disclosure, context accumulation across turns, and client-side orchestration bypass—that arise when agentic systems conflate relevance with authorization. To address these challenges, we introduce a layered isolation architecture combining policy-aware ingestion, retrieval-time gating, and shared inference, enforced through server-side agentic orchestration. Unlike client-side agent patterns, this approach centralizes tool execution, state management, and policy enforcement on the server, creating natural enforcement points for multitenant isolation and eliminating per-tenant infrastructure duplication.

An open-source implementation using Llama Stack—a vendor-neutral framework realizing the Responses API paradigm with server-side multi-turn orchestration—and its Kubernetes Operator demonstrates that secure multitenancy, cost-efficient sharing, and the flexibility of modern agentic systems are simultaneously achievable on shared infrastructure.

CCS Concepts

• **Computing methodologies** → *Machine learning*; • **Information systems** → Data management systems.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords

Llama Stack, Kubernetes operator, LLM systems, MLOps, LLMOps

ACM Reference Format:
Varsha Prasad and Francisco Javier Arceo. 2026. Securing the Agent: Vendor-Neutral, Multitenant Enterprise Retrieval and Tool Use. In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

Enterprise adoption of generative AI has evolved beyond simple prompt-response interactions toward agentic systems—AI applications that autonomously reason, use tools, retrieve information, and execute multi-step workflows to accomplish complex tasks [6, 14, 16]. This evolution reflects a fundamental shift: rather than treating large language models (LLMs) as sophisticated text generators, organizations now deploy them as reasoning engines capable of taking actions in the world.

An API-first paradigm for these systems is to unify multi-turn inference, tool use, and retrieval behind a single endpoint. OpenAI's Responses API has emerged as a de facto interface pattern for building such systems by providing a unified interface for chat-style inference, tool invocation, retrieval tools, and stateful workflows [13]. In parallel, open implementations increasingly expose OpenAI-compatible endpoints to decouple applications from a specific model provider.

However, real-world enterprise deployments differ sharply from the assumptions embedded in consumer-facing APIs and many academic prototypes. Enterprise environments exhibit characteristics that demand specialized architectural consideration:

- **Multiple tenants:** distinct business units, customers, or partners served from shared infrastructure, with strict isolation requirements.
- **Heterogeneous data:** document collections vary in format, sensitivity classification, and access requirements.
- **Strict access control:** regulatory frameworks require fine-grained governance with auditable access patterns.
- **Operational control:** visibility into agent behavior, tool execution sequences, and data access patterns is required for debugging and compliance, consistent with lessons from production ML engineering [2].

- **Vendor independence:** lock-in to a single AI provider creates business risk; enterprises require on-prem and hybrid options.

Naïve approaches to addressing these requirements replicate the entire agentic stack per tenant: separate vector stores, dedicated inference endpoints, and isolated tool configurations. This strategy incurs substantial costs — infrastructure scales linearly with tenants rather than with actual usage — and creates operational fragmentation that amplifies common ML systems maintenance risks [15].

1.2 Problem statement

This paper addresses a fundamental tension in enterprise agentic AI deployment:

> Enterprises require both shared infrastructure for cost efficiency and strict tenant isolation for security and compliance — while maintaining the flexibility of modern agentic architectures.

Standard agentic AI deployments exhibit assumptions incompatible with this requirement:

(1) **Client-side orchestration:** the application manages the inference–tool–inference loop, distributing security-critical logic to potentially untrusted clients and increasing operational complexity [2].
(2) **Homogeneous data access:** retrieval stacks assume uniform access to a corpus; dense retrieval methods (e.g., DPR) optimize similarity ranking rather than authorization [7].
(3) **Implicit trust boundaries:** tool execution is often treated as a capability extension without systematic verification of who may invoke tools or consume tool outputs, despite the centrality of tool use in modern agent designs [6, 14, 16].
(4) **Stateless isolation:** requests are treated independently, ignoring how conversation state and cached tool results can leak across boundaries; such hidden couplings are a classic source of ML systems fragility [15].

In multitenant settings, these assumptions create serious vulnerabilities. A document highly similar to a query may belong to a different tenant. A tool call may access resources outside the user's authorization scope. Conversation history may accumulate context that crosses security boundaries.

1.3 Contributions

This paper makes the following contributions:

(1) We formalize the problem of *multitenant enterprise RAG* under shared infrastructure, showing why semantic similarity alone (as used by dense retrievers and ANN indexes) is insufficient to enforce isolation and authorization [5, 7].
(2) We analyze failure modes of existing RAG and agentic systems in multitenant settings, including cross-tenant retrieval leakage, unauthorized context construction, and policy violations arising from client-side orchestration.
(3) We propose a layered isolation architecture for enterprise RAG that combines policy-aware ingestion, retrieval-time gating, and shared inference to achieve strict tenant isolation without per-tenant duplication of storage or models.
(4) We introduce server-side orchestration as a unifying enforcement layer that centralizes retrieval, tool execution, and state management, reducing the trusted computing base for secure multitenant RAG and aligning with production ML engineering best practices [2, 15].
(5) We present an open-source, vendor-neutral framework based on Llama Stack [10] and Kubernetes deployment via an operator [3, 11] that enables pluggable models, vector stores, and tools, demonstrating the feasibility of secure multitenant RAG on shared infrastructure.

Primary sources. Llama Stack [10]: https://github.com/llamastack/.
Kubernetes operator [11]: https://github.com/llamastack/llama-stack-k8s-operator.

2 Background and Related Work

2.1 The evolution of LLM application architectures

LLM application architectures have evolved through distinct phases, each introducing new capabilities and security considerations. **Completion APIs** exposed simple prompt $\rightarrow$ text interfaces with perimeter-oriented security.

Retrieval-augmented generation (RAG) [9] introduced retrieval and new attack surfaces such as retrieval manipulation and context poisoning. **Dense retrieval and learned retrievers** (e.g., DPR [7] and retrieval-augmented pretraining such as REALM [4]) established the core technical basis for modern RAG. **Vector search infrastructure** (e.g., FAISS [5]) enabled ANN retrieval at scale. **Tool-using agents** [6, 14, 16] extended LLMs with tool calls and an inference $\rightarrow$ tool $\rightarrow$ inference loop. **Autonomous multi-step agents** execute multi-tool workflows with limited human oversight. In multitenant deployments, isolation must span retrieval, tool invocation, state accumulation, and orchestration.

Multitenancy in agentic AI differs from traditional database or ML serving patterns. Beyond data isolation, agentic systems require policy-aware tool execution, stateful conversation management, and orchestration controls. Production ML systems require systematic lifecycle management [1] and experience operational fragility when security invariants are distributed across components [2, 15]—a problem amplified in agentic deployments where reasoning, retrieval, and tool execution interleave autonomously.

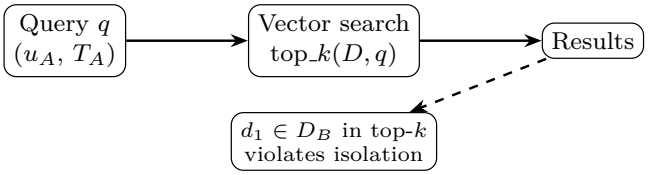

Figure 1: Similarity-only retrieval: top-k from shared corpus D can include documents from another tenant ($d_1 \in D_B$), violating isolation.

3 Problem Definition: Multitenant Enterprise Agentic AI

3.1 System model and threat analysis

We formalize the multitenant agentic AI environment: tenants (T) share infrastructure, each with associated data, users, tools, and policies. Users possess attributes (roles, clearances, group memberships). Documents live in vector stores with metadata (tenant ownership, classification, access policy). Agent execution follows the tool-using agent pattern [6, 16]: $E = [(i_1, \phi_1, r_1), (i_2, \phi_2, r_2), \ldots, (i_n, \emptyset, r_n)]$.

Key threats include: cross-tenant data leakage via retrieval, tool-mediated disclosure, context accumulation attacks, client-side orchestration bypass, and audit failure. Client-side orchestration is fundamentally unsuitable for enterprise multitenancy because it distributes trust to every client, enabling fabricated tool results, unauthorized context injection, and manipulation of conversation state.

4 Multitenant Agentic AI Challenges and Solutions

Standard agentic AI deployments exhibit fundamental incompatibilities with enterprise multitenancy. We analyze these failure modes and propose a layered isolation architecture that addresses them.

4.1 Similarity-authorization gap in retrieval

Every major RAG framework retrieves documents by semantic similarity using dense retrievers such as DPR [7] and ANN indexing [5]. Similarity is orthogonal to authorization: a document from tenant B may be the most semantically similar result for tenant A's query, yet returning it violates isolation.

Formally, for tenants T_A and T_B sharing corpus $D = D_A \cup D_B$, the isolation invariant requires that only documents with $\text{tenant}(d) = \text{tenant}(u_A)$ be returned. For any similarity threshold θ, there exist queries q such that $\arg\max_{d \in D} \text{sim}(q, d) \in D_B$. Thus similarity-based retrieval cannot enforce tenant isolation without an authorization predicate: $\mathcal{R}(q, u) = \{d \in D : \text{sim}(q, d) > \theta \wedge P(u, d) = \text{permit}\}$.

4.2 Tool-mediated disclosure and client-side orchestration risks

Agentic systems executing tool sequences [14, 16] face unique multitenant threats: **Tool-mediated disclosure:** Agents invoke tools with agent credentials rather than end-user authorization, potentially accessing unauthorized data across tenant boundaries. **Context accumulation:** Multi-turn conversations persist context without per-turn policy re-validation, enabling cross-tenant data leakage. **Client-side bypass:** When orchestration runs client-side, malicious clients can skip authorization checks, manipulate tool invocation, or extract unauthorized data.

These failures stem from distributing security-critical logic outside the trust boundary, consistent with production ML system fragility patterns [2, 15].

4.3 Layered isolation architecture

We propose a three-layer solution: policy-aware ingestion, retrieval-time gating, and shared inference.

Layer 1: Policy-aware ingestion. Tenant metadata must be attached at document ingestion time. An ingestion function $\mathcal{I}(d, t) \to D_t$ tags document d with tenant t's attributes so that every chunk inherits ownership metadata.

Layer 2: Retrieval gating. Two-tier enforcement: (1) resource-level authorization before search, and (2) chunk-level metadata filtering after retrieval. This composes similarity search with mandatory authorization predicates, separating semantic ranking from access control.

Layer 3: Shared inference. The LLM layer is shared across tenants, with isolation enforced at input construction. Since layers 1–2 ensure only authorized content enters prompts, inference can be safely shared, reducing cost from $O(N \cdot M)$ to $O(M)$ for N tenants and M model endpoints.

4.4 Server-side orchestration as enforcement layer

Server-side orchestration centralizes retrieval, tool execution, and state management within the trust boundary, eliminating client-side bypass risks. This design aligns with tool-using agent formulations [6, 16] while enabling centralized policy enforcement, audit logging, and state isolation.

We propose a three-layer architecture that directly addresses the failure modes identified in Section **??**: policy-aware ingestion, retrieval-time gating, and shared inference.

4.5 Policy-aware ingestion (Layer 1)

Tenant metadata must be attached at document ingestion time, not retrofitted. This reduces the risk of accidental cross-tenant coupling and missing invariants during system evolution [15]. Define an ingestion function $\mathcal{I}(d, t) \to D_t$ that tags document d with tenant t's attributes so that every chunk inherits ownership metadata. In Llama Stack, when files are attached to vector stores, the `attributes` parameter carries tenant ownership, classification, and access policy; these are stored as chunk metadata.

4.6 Retrieval gating (Layer 2)

Retrieval is gated in two tiers. **(1) Resource-level ABAC:** before any search, the system checks that the user is authorized to read the vector store; if not, no search is performed. **(2) Chunk-level metadata filtering:** after retrieval, structured filters are applied on chunk metadata so that only documents satisfying the policy are admitted. This composes similarity search (dense retrieval [7] over ANN indexes [5]) with a mandatory authorization predicate:

$$\mathcal{R}(q, u) = \{d \in D : \mathrm{sim}(q, d) > \theta \wedge P(u, d) = \mathrm{permit}\}.$$

Backends such as Elasticsearch and SQL-backed vector stores can support predicate pushdown for efficiency on large corpora; otherwise, post-retrieval gating still enforces the semantic/authorization separation.

4.7 Shared inference (Layer 3)

The LLM inference layer is shared across tenants; the model itself does not require per-tenant isolation, only the context fed to it. Because layers 1 and 2 ensure that only authorized documents and tool results enter the prompt, the inference layer can be safely shared. At the serving layer, modern systems show that batching, scheduling, and memory management dominate throughput and latency for generative transformers [8, 17]. With N tenants and M model endpoints, cost scales as $O(M)$ rather than $O(N \cdot M)$.

5 Server-Side Orchestration as Enforcement Layer

Server-side orchestration is essential for multitenant security: it centralizes retrieval, tool execution, and state management, reducing the trusted computing base and addressing failure modes in Section **??**. This design also aligns with the execution structure implied by tool-using agent formulations [6, 16].

5.1 The case against client-side orchestration

In client-side patterns, the client controls the inference–tool–inference loop. A compromised or buggy client can skip retrieval filters, invoke unauthorized tools, or accumulate cross-tenant context. Formally, client-side orchestration expands the TCB to include untrusted client code, so server-side security invariants cannot be enforced from the server alone. This is consistent with broader findings that ML systems become fragile when critical invariants are distributed across components without clear ownership or enforcement points [2, 15].

5.2 Server-side orchestration in Llama Stack

Llama Stack implements the Responses API with the full loop executed on the server [10, 13]. Tool execution (`file_search`, `web_search`, MCP, and function tools) runs inside the server trust boundary, and state is managed server-side with tenant-scoped access control. Streaming events expose tool calls and text deltas, improving observability in line with production ML monitoring and governance needs [2].

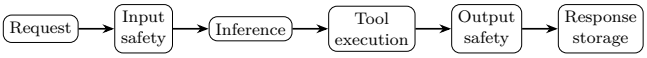

Figure 2: Server-side orchestration flow: every step runs inside the server trust boundary.

Failure mode	Enforcement point
Cross-tenant retrieval leakage	Layer 2 retrieval gating (ABAC + metadata filters)
Context accumulation	Tenant-scoped state storage and per-turn authorization
Tool-mediated disclosure	Server-side tool execution with authorization propagation
Client-side bypass	Server-side orchestration (reduced TCB)
Audit failure	Server-side telemetry and tracing

Table 1: Mapping from failure modes to architectural enforcement points.

5.3 Enforcement points

Table 1 maps each failure mode from Section **??** to the enforcement point that mitigates it.

6 Reference Implementation: Llama Stack

Llama Stack [10] provides an open-source, vendor-neutral implementation of the proposed architecture. Unlike client-side orchestration libraries, Llama Stack executes the complete agentic control loop on the server, creating natural enforcement points for multitenancy and security [13].

The framework implements a provider abstraction enabling hybrid deployments: inference providers (vLLM [8], Ollama, OpenAI), vector stores (Chroma, pgvector, Elasticsearch), and tool runtimes (file_search, web_search, MCP). Together with open models such as gpt-oss [12], this provides a complete open-source alternative to proprietary agentic AI platforms.

6.1 Multitenant deployment patterns

The Llama Stack Kubernetes Operator [11] supports multiple deployment topologies: **Shared server:** Single instance with logical isolation via ABAC policies and metadata-gated retrieval. **Per-tenant instances:** Separate deployments with namespace isolation and shared model backends. **Hybrid:** Mixed deployment balancing cost efficiency with stronger isolation for high-security tenants.

Authorization is enforced at API routes, routing resolution, and tool execution, with tenant-scoped state storage and per-turn policy checks preventing cross-tenant leakage.

Second, the framework adopts a pluggable provider model. Each API is implemented by interchangeable providers that share a common interface, enabling vendor independence and hybrid deployments across on-premise and cloud environments.

Third, Llama Stack implements server-side agentic orchestration. The full agentic execution loop—including inference, tool invocation, and subsequent inference—executes within the server trust boundary. This design contrasts with client-side orchestration approaches and enables centralized enforcement of access control, safety policies, and execution constraints.

Finally, the framework introduces a distribution model. Pre-configured combinations of APIs and providers are packaged as distributions, enabling turnkey deployment while preserving flexibility for customization and provider substitution.

6.2 Layered Architecture

Llama Stack follows a layered architecture that separates concerns across distinct tiers, as illustrated in Figure 3. At the top, an HTTP/REST layer handles request routing, authentication, quota enforcement, and streaming. Beneath this layer, a set of domain-specific APIs expose functionality for inference, agentic execution, vector I/O, safety, and tool integration.

A routing layer mediates between API calls and concrete provider implementations, resolving logical resource identifiers (e.g., model identifiers or vector store identifiers) to physical provider instances. Below this, a provider layer encapsulates both inline providers executing in-process and remote providers that adapt external services. Persistent state is maintained in a storage layer comprising key–value stores, relational stores, and vector databases.

This separation enables independent evolution of APIs, providers, and storage backends while preserving a unified control plane for policy enforcement.

6.3 Core APIs and Agentic Execution

Llama Stack defines a comprehensive set of APIs covering the full lifecycle of agentic applications, including inference, agents, vector I/O, safety, tools, file management, and evaluation. Each API is designed with multitenancy as a first-class concern, enabling tenant-scoped resource management and access control.

The Agents API, which implements the OpenAI Responses API paradigm, is particularly significant for multitenant agentic systems. Unlike traditional chat completion APIs that terminate after a single inference call, the Responses API orchestrates complete agentic workflows. A single request may trigger multiple inference calls, tool executions, safety checks, and state transitions before producing a final response.

All such operations are executed within the server boundary. Conversation state is retrieved and persisted server-side, tools are invoked under centralized authorization, and safety guardrails are applied at each step. This design ensures that intermediate context, tool outputs, and execution state remain subject to uniform access control policies.

6.4 Provider Architecture

Extensibility in Llama Stack is achieved through its provider architecture. Each API may be backed by multiple providers, which are categorized as inline or remote. Inline providers execute within the Llama Stack process and are suitable for sensitive operations requiring in-process execution. Remote providers adapt external inference engines, vector databases, or services through standardized interfaces.

This separation enables hybrid deployments in which sensitive data paths remain local while computationally intensive operations are delegated to scalable external services. Crucially, provider substitution is transparent to clients, as all interactions occur through the unified API layer.

6.5 Routing Layer

The routing layer dispatches API requests to provider instances based on logical resource identifiers. For example, inference requests are routed according to model identifiers, while vector queries are routed based on vector store identifiers. This indirection enables fine-grained control over resource access and placement.

From a multitenancy perspective, the routing layer serves as a critical enforcement point. Routing decisions can incorporate authorization checks, tenant identity, and policy constraints before delegating requests to providers. Different tenants may thus be routed to distinct provider instances or storage backends while sharing the same API surface.

6.6 Distribution Model

A distribution packages a specific set of APIs, provider configurations, and registered resources into a deployable unit. Distributions support turnkey deployment for common scenarios, environment-specific configuration (e.g., development versus production), and seamless provider substitution without application changes.

By decoupling application logic from provider selection, the distribution model enables organizations to evolve their infrastructure and vendor choices while preserving stable interfaces for agentic applications.

6.7 Access Control Framework

Llama Stack includes a declarative, policy-based access control framework that evaluates authorization decisions at runtime. Policies specify permitted and forbidden actions over resource scopes, with conditions based on user identity, resource attributes, and ownership relationships.

Authorization checks are enforced at multiple layers, including API routes, routing table resolution, and tool execution. A default-deny model ensures that access is only granted when explicitly permitted by policy. This design enables consistent enforcement across inference, retrieval, and agentic execution.

6.8 Server Architecture

The Llama Stack server is implemented atop a modern asynchronous web framework and provides OpenAI-compatible endpoints for inference and agentic execution. Authentication middleware supports pluggable identity providers, while quota management enables per-principal rate limiting and usage tracking. Streaming execution is supported for long-running agentic workflows, and telemetry integration enables end-to-end observability.

Importantly, all agentic orchestration occurs within the server process, enabling comprehensive audit logging of inference calls, tool executions, and data access events.

6.9 Implications for Multitenant Agentic AI

Taken together, these architectural choices yield several properties essential for multitenant enterprise deployments. First, centralized orchestration creates uniform enforcement points for access control and policy compliance. Second, provider-level isolation enables tenant-specific routing while preserving shared infrastructure. Third, server-managed state prevents cross-tenant context leakage across multi-turn interactions. Finally, centralized execution enables comprehensive auditing and compliance monitoring.

These properties form the foundation for the layered isolation architecture described in the following section, where we show how policy-aware ingestion, retrieval gating, and server-side orchestration can be composed to achieve secure, cost-efficient multitenant agentic AI.

6.10 High-level architecture

We propose a layered architecture for secure multitenant agentic AI with an authentication layer, a server-side agentic orchestrator, policy evaluation, and a shared-but-logically-isolated vector store.

6.11 Authentication and tenant context

Requests begin with authentication establishing a user and tenant context. Claims (tenant/namespace, roles/groups, projects) are mapped to authorization attributes and propagated through the request lifecycle.

6.12 Server-side agentic orchestration

The orchestrator executes the multi-turn loop server-side: inference, tool-call detection, tool classification, policy evaluation, tool execution, and context construction. Server-side tools (e.g., `file_search`, `web_search`, MCP tools) are executed within the trust boundary; client-side function tools are explicitly delegated.

6.13 Policy-aware ingestion

Ingestion attaches mandatory metadata (tenant attribution, classification) and records lineage for audit. Documents without required metadata are rejected.

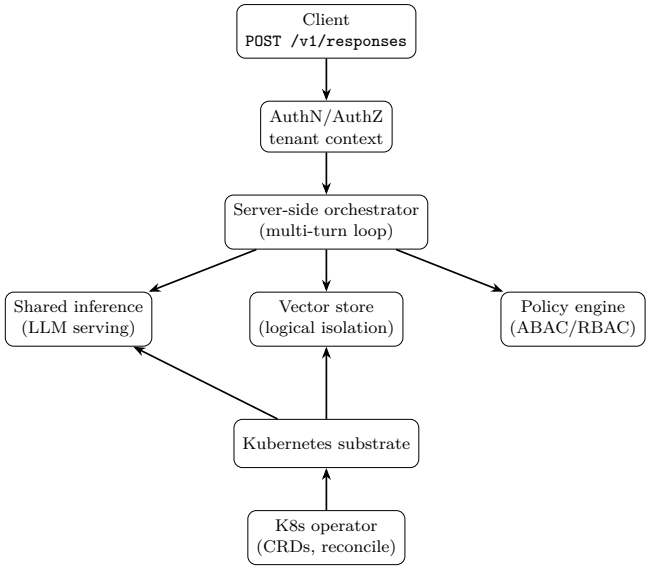

Figure 3: Reference architecture for multitenant enterprise agentic AI on shared Kubernetes infrastructure.

6.14 Shared vector store with logical isolation

A single physical vector index is shared, but each chunk carries tenant and policy metadata. Retrieval uses query-time predicates (tenant predicates and classification filters) to prevent cross-tenant candidate leakage, separating similarity ranking from authorization checks [5, 7].

6.15 Two-phase retrieval: search and admission

Phase 1 performs filtered similarity search, grounded in dense retrieval and ANN indexing methods [5, 7]. Phase 2 applies policy-based admission; only admitted documents enter the generation context, and denials are logged.

6.16 Tool execution with policy gating

Every tool invocation is preceded by policy evaluation over user attributes, tool identity, and parameters. This is particularly important for agent loops where the model chooses actions and tools iteratively [6, 14, 16].

6.17 Shared inference with isolated contexts

LLM instances are shared across tenants, but each request constructs an isolated context containing only authorized inputs. Serving efficiency and scheduling are handled by the inference backend; modern LLM serving systems show that batching and memory management dominate throughput [8, 17]. Because isolation is enforced at ingestion and retrieval, the inference layer itself does not need per-tenant

replicas—reducing cost to $O(M)$ model endpoints for N tenants while preserving strict logical isolation.

6.18 Overview and implementation

Llama Stack is an open-source framework implementing the Responses API paradigm with server-side orchestration [10, 13]. It is vendor-neutral via a provider abstraction, and is designed for Kubernetes-native deployment and operator-based lifecycle management [3, 11].

6.19 Responses API implementation

The Responses API exposes operations to create responses (execute an agent), retrieve responses, list responses, inspect input items, and delete responses; creation supports both streaming and non-streaming modes. Internally, the implementation converts request input to chat messages and runs the inference–tool loop via a streaming orchestrator or a synchronous path. Results are persisted through a responses store backed by an authorized SQL layer so that rows are tagged with owner and access attributes and filtered on read by the default ABAC policy. Streaming events expose reasoning, tool calls, and text deltas, giving operators and auditors fine-grained visibility into agent behavior without requiring client-side instrumentation.

6.20 Vector store abstraction and filters

Llama Stack defines a vector store protocol (vector I/O API) implemented by multiple providers (e.g., Chroma, pgvector, Elasticsearch, in-process SQLite). Each vector store is a logical resource registered in a routing table and subject to the same access control as other resources; creation assigns an owner from the authenticated user. The protocol supports a structured filter language for query-time metadata constraints: comparison operators (`eq`, `ne`, `gt`, `gte`, `lt`, `lte`) and compound `and`/`or` filters. When the agent uses `file_search`, the server applies the user's tenant and policy attributes so that retrieval is gated by both resource-level and chunk-level checks, enabling metadata-driven isolation without requiring a separate physical index per tenant.

6.21 Attribute-based access control

Llama Stack supports ABAC-style policies for tenant isolation, role-based capabilities, and classification enforcement. The access control engine evaluates `AccessRule`s with permit/forbid scopes and optional conditions (e.g., "user in owners roles", "user is owner", "resource is unowned"); the default policy permits access when the user is the resource owner or when the user's attributes (roles, teams, projects, namespaces) match the resource's access attributes. Authorization is enforced at API routes (e.g., `RouteAuthorizationMiddleware`), at routing table resolution (e.g., before resolving a vector store or model), and at storage read time via `AuthorizedSqlStore`, which builds SQL `WHERE` clauses from the current user so that tenants only see their own or attribute-matched rows. JWT or Kubernetes auth providers map external claims (e.g., `tenant`, `groups`) into these attributes, so enterprise identity systems can drive isolation without embedding tenant IDs in application logic.

7 Kubernetes Deployment via the Llama Stack Operator

The architectural guarantees described in previous sections—layered isolation, server-side orchestration, and attribute-based access control—must ultimately be realized on shared infrastructure. Kubernetes has become the de facto substrate for such deployments, building on lessons from cluster managers such as Borg [3]. The Llama Stack Kubernetes Operator [11] bridges the gap between the logical isolation model of the Llama Stack server and the physical resource management provided by Kubernetes, automating deployment, lifecycle management, and network-level isolation for Llama Stack distributions.

7.1 Operator overview and reconciliation model

The operator follows the standard Kubernetes operator pattern: it extends the Kubernetes API with a custom resource definition (CRD), `LlamaStackDistribution`, and runs a controller that continuously reconciles the desired state declared in each CR instance with the actual cluster state. When a user creates, updates, or deletes a `LlamaStackDistribution` resource, the operator's reconciliation loop automatically provisions or tears down the corresponding pods, services, persistent volumes, and (optionally) network policies. This declarative model ensures that deployments are reproducible and auditable, reducing the configuration drift that is a well-known source of operational risk in production ML systems [15].

7.2 The `LlamaStackDistribution` custom resource

Each `LlamaStackDistribution` CR captures the full specification of a Llama Stack server deployment. Key fields include:

- **Distribution:** the distribution name (e.g., `starter`, `starter-gpu`), which the operator resolves to a container image. Image mapping overrides via a ConfigMap allow independent patching for security fixes or bug fixes without requiring a new operator release.
- **Replicas and container spec:** replica count and environment variables that configure the server, including pointers to inference backends (e.g., `OLLAMA_URL`, `VLLM_URL`) and model identifiers.
- **Storage:** persistent volume size and mount path for model artifacts and server state, ensuring data survives pod restarts.
- **ConfigMap reference:** an optional reference to a ConfigMap containing the server's `config.yaml`. Updates to the ConfigMap trigger an automatic pod restart so the running server always reflects the latest configuration.

API type	Providers
Inference	remote::vllm, remote::ollama, remote::openai, remote::bedrock, remote::watsonx, inline::sentence-transformers
Agents	inline::meta-reference
VectorIO	inline::milvus, remote::milvus, inline::chromadb, inline::sqlite-vec
Safety	remote::trustyai_fms
Tool runtime	inline::rag-runtime, remote::brave-search, remote::tavily-search, remote::model-context-protocol
Files	inline::localfs
Telemetry	inline::meta-reference

Table 2: API types and providers available in a `LlamaStackDistribution`. Inline providers run in-process; remote providers adapt external services.

An example CR deploying a Llama Stack server backed by Ollama:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: tenant-a-stack
spec:
  replicas: 1
  server:
    distribution:
      name: starter
    containerSpec:
      env:
      - name: OLLAMA_INFERENCE_MODEL
        value: "llama3.2:1b"
      - name: OLLAMA_URL
        value: "http://ollama.svc:11434"
    storage:
      size: "20Gi"
      mountPath: "/home/lls/.lls"
```

7.3 API providers and distribution capabilities

Each distribution bundles a set of API providers that map to the Llama Stack API surface described in Section 6. Table 2 summarizes the key API types and their provider implementations. The distribution supports both inline providers (executing in-process) and remote providers that adapt external services, enabling hybrid deployments where sensitive data paths remain local while compute-intensive operations are delegated to scalable backends.

7.4 OpenAI-compatible API surface

The deployed Llama Stack server exposes OpenAI-compatible endpoints under the `/v1/openai/v1/` path prefix, enabling existing OpenAI SDKs and tooling to target the deployment by changing only the `base_url`. Supported endpoints include Chat Completions, Completions, Embeddings, Files, Vector Stores, Vector Store Files, Models, and the Responses API. This compatibility layer is significant for enterprise adoption: organizations can migrate workloads from proprietary APIs to a self-hosted, policy-enforced deployment without modifying client code, while gaining the multitenancy and isolation guarantees described in this paper.

7.5 Network-level isolation

Beyond the application-layer isolation enforced by Llama Stack's ABAC engine, the operator provides optional Kubernetes `NetworkPolicy` resources for each distribution instance. When enabled via a feature flag in the operator's ConfigMap, ingress traffic to each `LlamaStackDistribution` pod is restricted by default to:

- Pods labeled as part of the Llama Stack deployment in the same namespace.
- The operator's own namespace.

Administrators can further customize access through the CR's `allowedFrom` field, specifying permitted namespaces by name or by label selector, and can optionally expose the service externally via an Ingress resource. This defense-in-depth approach layers network-level segmentation on top of application-level ABAC, limiting the blast radius of a compromised component and ensuring that even if an attacker gains access to a pod in one namespace, they cannot reach another tenant's Llama Stack server at the network layer.

7.6 Deployment topology for multitenancy

The operator supports multiple deployment topologies depending on organizational requirements:

- **Shared server, logical isolation:** A single `LlamaStackDistribution` serves all tenants, with isolation enforced entirely through ABAC policies and metadata-gated retrieval. This minimizes infrastructure cost and is appropriate when tenants share a trust domain.
- **Per-tenant distribution instances:** Separate CRs are deployed in tenant-specific namespaces, combining Kubernetes RBAC and namespace isolation with application-level ABAC. Shared model-serving backends (e.g., vLLM) remain in a common namespace, accessible through Kubernetes service networking.
- **Hybrid:** Some tenants share a distribution while high-security tenants receive dedicated instances, blending cost efficiency with stronger isolation where required.

In all topologies, the operator's reconciliation loop ensures that the declared state is continuously enforced, and standard GitOps workflows (e.g., Argo CD, Flux) can manage

`LlamaStackDistribution` resources alongside other Kubernetes manifests, providing versioned, reviewable, and auditable infrastructure-as-code for the entire agentic AI deployment.

8 Analysis and Discussion

The layered isolation architecture addresses the failure modes identified above. Cross-tenant retrieval leakage is mitigated by composing similarity retrieval with mandatory policy predicates [5, 7]. Tool-mediated disclosure is reduced through centralized, policy-gated execution aligned with tool-using agent patterns [14, 16]. Context accumulation and state leakage are prevented by tenant-scoped storage and per-turn authorization [2, 15].

Performance depends on the inference backend [8, 17]; predicate pushdown into vector stores is desirable for scale but post-retrieval gating still enforces security invariants.

Limitations: ABAC policies can become complex at scale; metadata filtering timing varies by backend; model prior knowledge is orthogonal to RAG isolation; client-side function tools execute outside the server trust boundary.

9 Conclusion

Agentic AI systems introduce multitenancy, access control, and compliance challenges that exceed the assumptions in standard client-orchestrated patterns. We formalized the insufficiency of similarity-based retrieval for tenant isolation, grounded in dense retrieval and ANN vector search [5, 7], and analyzed failure modes of existing RAG and agentic frameworks. We proposed a layered isolation architecture combining policy-aware ingestion, retrieval-time gating, and shared inference, and argued that server-side orchestration is the unifying enforcement layer that centralizes retrieval, tool execution, and state management and reduces the trusted computing base. These design choices also reduce operational risk by turning implicit assumptions into enforced invariants, consistent with lessons from production ML systems engineering [2, 15].

Llama Stack [10] demonstrates that vendor-neutral, open-source tooling can support enterprise-grade multitenancy through provider abstraction, policy enforcement, and a server-side Responses API implementation [13]. Together with open models such as gpt-oss [12] and efficient serving via vLLM [8], this provides a complete open-source alternative to proprietary agentic AI platforms. The Llama Stack Kubernetes Operator [11] enables reproducible deployments on shared cluster infrastructure [3]. The patterns presented here—server-side orchestration, policy-aware retrieval, defense in depth, and operator-driven infrastructure—provide a foundation for secure, scalable agentic AI deployments that maintain enterprise control over autonomous agent behavior.

References

[1] [n. d.]. MLflow: A Machine Learning Lifecycle Platform. Open-source project. https://mlflow.org/ Accessed: 2026-02-23.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 291–300. doi:10.1109/ICSE-SEIP.2019.00042

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* (2016). https://cacm.acm.org/practice/borg-omega-and-kubernetes/

[4] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: Retrieval-Augmented Language Model Pre-Training. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2002.08909

[5] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-Scale Similarity Search with GPUs. *arXiv preprint arXiv:1702.08734* (2017). https://arxiv.org/abs/1702.08734

[6] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. 2022. MRKL Systems: A Modular, Neuro-Symbolic Architecture that Combines Large Language Models, External Knowledge Sources and Discrete Reasoning. *arXiv preprint arXiv:2205.00445* (2022). https://arxiv.org/abs/2205.00445

[7] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781. doi:10.18653/v1/2020.emnlp-main.550

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. 611–626. doi:10.1145/3600006.3613165

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2005.11401

[10] Llama Stack Contributors. 2025. Llama Stack. GitHub repository. https://github.com/llamastack/ Accessed: 2026-01-28.

[11] Llama Stack Contributors. 2025. Llama Stack Kubernetes Operator. GitHub repository. https://github.com/llamastack/llama-stack-k8s-operator Accessed: 2026-01-28.

[12] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[13] OpenAI. 2025. Responses API Reference. Online documentation. https://platform.openai.com/docs/api-reference/responses Accessed: 2026-02-16.

[14] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2302.04761

[15] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems*. 2503–2511. https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757b546b22a-Abstract.html

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2210.03629

[17] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 521–538. https://www.usenix.org/conference/osdi22/presentation/yu