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: retrieval systems rank documents by relevance—whether through semantic similarity, keyword matching, or hybrid approaches—not by authorization, so a query from one tenant can surface another tenant's confidential data simply because it scores highest. We formalize this gap and analyze additional shortcomings—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. This approach centralizes security-critical operations—tool execution authorization, state isolation, and policy enforcement—on the server, creating natural enforcement points for multitenant isolation while allowing client-side frameworks to retain control over agent composition and latency-sensitive operations.

We validate the proposed architecture through an open sourced implementation in Llama Stack—a vendor-neutral framework realizing the Responses API paradigm with server-side multi-turn orchestration—demonstrating that secure multitenancy, cost-efficient resource sharing, and autonomous agent capabilities are simultaneously achievable on shared infrastructure.

Conference'17, Washington, DC, USA

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

CCS Concepts

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

Keywords

multitenancy, retrieval-augmented generation, access control, agentic AI, server-side orchestration, Llama Stack, LLM systems, 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 [12, 30, 36]. 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 is emerging to unify multi-turn inference, tool use, and retrieval behind a single endpoint. OpenAI's Responses API is one prominent example, providing a unified interface for chat-style inference, tool invocation, retrieval, and stateful workflows [27], while open-source frameworks increasingly expose OpenAI-compatible endpoints to decouple applications from any single provider.

However, real-world enterprise deployments differ sharply from the assumptions embedded in consumer-facing APIs and many academic prototypes, exhibiting 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 [1].

- **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 [31].

1.2 Problem statement

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

> Autonomous agents require flexible tool access and multi-turn reasoning capabilities, yet enterprise environments demand strict tenant isolation and policy enforcement—requirements that existing agentic architectures cannot simultaneously satisfy.

Standard agentic AI deployments exhibit security assumptions incompatible with enterprise multitenancy:

(1) **Client-side orchestration:** the application manages the inference—tool—inference loop, distributing security-critical logic to potentially untrusted clients and increasing operational complexity [1].
(2) **Homogeneous data access:** retrieval stacks assume uniform access to a corpus; retrieval methods (whether dense, sparse, or hybrid) optimize relevance ranking rather than authorization [13].
(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 [12, 30, 36].
(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 [31].

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.

This paper focuses on agentic RAG—the intersection of retrieval-augmented generation and autonomous tool use—as the primary case study, since retrieval is where the relevance-authorization gap is most acute. However, the layered isolation architecture and server-side enforcement patterns we propose are general and extend to other agentic capabilities including inference, tool execution, code generation, and multi-step workflows.

1.3 Contributions

This paper makes the following contributions:

(1) We formalize the problem of *multitenant enterprise RAG* under shared infrastructure, showing why relevance ranking alone (whether vector-based, keyword-based, or hybrid) is insufficient to enforce isolation and authorization without explicit authorization predicates [11, 13].
(2) We analyze shortcomings 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 [1, 31].
(5) We validate the proposed architecture through an implementation in Llama Stack [18], an open-source, vendor-neutral framework with Kubernetes deployment via an operator [3, 19], demonstrating the feasibility of secure multitenant agentic RAG on shared infrastructure with pluggable models, vector stores, and tools.

Primary sources. Llama Stack [18]: https://github.com/llamastack/.
Kubernetes operator [19]: https://github.com/llamastack/llama-stack-k8s-operator.

2 Background

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 → text interfaces with perimeter-oriented security.

Retrieval-augmented generation (RAG) [17] introduced retrieval and new attack surfaces such as retrieval manipulation and context poisoning, building on foundational work on prompt injection vulnerabilities [35]. **Dense retrieval and learned retrievers** (e.g., DPR [13] and retrieval-augmented pretraining such as REALM [9]) established the core technical basis for modern RAG. **Retrieval infrastructure** evolved to support multiple search modalities: dense vector search across diverse vector databases [11], sparse keyword matching (BM25), and hybrid approaches combining both with neural rerankers [29]. However, all of these modalities optimize for relevance; none enforce authorization natively. **Tool-using agents** [12, 30, 36] extended LLMs with tool calls and an inference → tool → inference loop. **Autonomous multi-step agents** execute multi-tool workflows with limited human oversight.

The Responses API paradigm [24, 27] represents a convergence of these phases, unifying inference, tool use, retrieval, and state management behind a single endpoint.

Each phase introduced new capabilities but also inherited the security gaps of prior phases. The Responses API paradigm unifies these capabilities under a single interface but assumes single-tenant deployment. Enterprise multitenancy requires policy-aware tool execution, stateful conversation management, and orchestration controls—challenges amplified in agentic deployments where reasoning, retrieval, and tool execution interleave autonomously [1, 31].

3 Multitenant Agentic AI: Challenges and Solutions

We formalize the multitenant agentic AI environment: tenants (T) share infrastructure, each with associated data, users, tools, and policies. Agent execution follows the tool-using pattern [12, 36], where an execution sequence E consists of alternating inference steps i, tool calls ϕ, and responses r:

$$E = i_1, \phi_1, r_1, i_2, \phi_2, r_2, \ldots, i_n, \emptyset, r_n \tag{1}$$

where the final step has no tool call ($\emptyset$) and terminates with response r_n.

Standard agentic deployments exhibit several security assumptions that are fundamentally incompatible with enterprise multitenancy.

3.1 Key security challenges

Relevance-authorization gap: Retrieval systems—whether vector-based, keyword-based, or hybrid—optimize for relevance metrics rather than authorization policies. This creates a fundamental gap: search ranking considers semantic similarity, term frequency, or combined relevance signals, but authorization decisions depend on access control policies that are orthogonal to these relevance measures.

For tenants T_A and T_B sharing corpus $D = D_A \cup D_B$, retrieval methods cannot enforce tenant isolation without an authorization predicate. Let q denote a query, u denote a user, d denote a document, θ denote a relevance threshold, and Pu, d denote an authorization policy that returns permit or deny. Then secure retrieval requires:

$$\{d \in D : \text{relevance}\big(q, d\big) > \theta \wedge P\big(u, d\big) = \text{permit}\} \tag{2}$$

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 shortcomings stem from distributing security-critical logic outside the trust boundary [1, 31, 32].

3.2 Proposed solution: layered isolation with server-side orchestration

To address these shortcomings, we propose a two-part solution. A three-layer isolation architecture secures the *data path*: how documents are ingested, retrieved, and fed to the model. Server-side orchestration secures the *control path*: how tools are invoked, state is managed across turns, and policies are enforced (detailed in Section 4). The three data-path layers are:

Layer 1: Policy-aware ingestion. Tenant metadata attached at ingestion: $\mathcal{I}d, t \rightarrow D_t$ tags document d with tenant t's attributes.

Layer 2: Retrieval gating. Two-tier enforcement: resource-level authorization before search and chunk-level filtering after retrieval, composing similarity search with authorization predicates.

Layer 3: Shared inference. LLM layer shared across tenants with isolation enforced at input construction, reducing cost from $ON \cdot M$ to OM.

The following subsections detail each layer.

3.3 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 [31]. The ingestion function $\mathcal{I}$ defined above ensures that every chunk inherits ownership metadata, so downstream retrieval and authorization can operate on consistent tenant attributes.

3.4 Retrieval gating (Layer 2)

Retrieval is gated in two tiers across multiple search modalities. **(1) Resource-level attribute-based access control (ABAC):** before any search, the system checks that the user is authorized to read the search 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 two-tier gating applies regardless of search modality (dense, sparse, or hybrid): each retrieval path enforces authorization before applying relevance ranking, implementing the secure retrieval formulation from Equation (2). Where the backend supports it, predicate pushdown can enforce filtering efficiently at scale; otherwise, post-retrieval gating still enforces the relevance/authorization separation.

3.5 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 [14, 37]. With N tenants and M model endpoints, cost scales as OM rather than $ON \cdot M$.

4 Server-Side Orchestration as Enforcement Layer

As introduced in Section 3, the control path—tool invocation, state management, and policy enforcement—benefits from server-side orchestration to reduce the trusted computing base (TCB). This section examines the limitations of purely client-side orchestration for multitenant settings, the trade-offs of server-side orchestration, and how hybrid approaches may combine the strengths of both.

4.1 Limitations of client-side orchestration

Client-side orchestration offers important advantages: low latency for local tool calls, flexibility in composing custom agent logic, and the ability to leverage rich client-side frameworks. However, in multitenant enterprise settings, purely client-side patterns introduce specific shortcomings. When 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. 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 [1, 31].

4.2 Trade-offs of server-side orchestration

Server-side orchestration centralizes policy enforcement but introduces its own trade-offs: added latency for tool calls that could execute locally, reduced flexibility for client-specific agent logic, and increased server-side complexity. For workloads where all tenants are trusted and isolation is not a concern, client-side orchestration remains a pragmatic and effective choice.

In practice, hybrid architectures are likely to emerge as the dominant pattern: server-side orchestration enforces security-critical invariants (authorization, state isolation, audit logging), while client-side frameworks retain control over agent composition, user interaction, and latency-sensitive tool calls that do not cross trust boundaries. Llama Stack's design supports this model—clients orchestrate their agent logic using any OpenAI-compatible framework, while the server enforces policy at the API boundary.

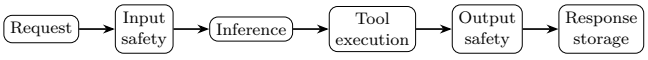

Figure 1: Server-side orchestration flow: every step runs inside the server trust boundary.

4.3 Enforcement points

Table 1 maps each shortcoming from Section 3 to the enforcement point that mitigates it.

Shortcoming	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 shortcomings to architectural enforcement points.

5 Llama Stack Implementation

Llama Stack [18] provides an open-source implementation of the architecture proposed in Sections 3 and 4. The framework executes the complete agentic control loop server-side via the Responses API [27], with pluggable providers for inference (vLLM [14], Ollama, OpenAI), vector stores (Chroma, pgvector, Elasticsearch), and tools (`file_search`, `web_search`, Model Context Protocol [2]). Together with open models such as gpt-oss [26], this provides a complete open-source alternative to proprietary agentic AI platforms. The Kubernetes Operator [19] automates deployment across multiple topologies, from shared instances with logical isolation to per-tenant deployments with namespace isolation.

5.1 Layered Architecture

Llama Stack follows a layered architecture that separates concerns across distinct tiers, as illustrated in Figure 2. 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.

5.2 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.

As discussed in Section 4, Llama Stack operates as a *platform layer*—the server-side execution target that client-side agent frameworks (LangChain [15], LangGraph [16], CrewAI [4], and others) call into via OpenAI-compatible endpoints, gaining server-side policy enforcement, multitenancy, and provider portability without changes to agent code.

5.3 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. A developer can prototype locally with inline providers (e.g., sqlite-vec for vector search, an in-process safety model), then move to production-grade remote providers (e.g., pgvector, vLLM) without changing application code or API calls—only the distribution configuration changes. This lowers the barrier from experimentation to production deployment while preserving consistent security and isolation guarantees across environments.

5.4 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.

5.5 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.

5.6 Access Control Framework

Llama Stack includes a declarative, policy-based access control framework that evaluates authorization decisions at runtime. Access rules specify permit or forbid scopes with optional conditions based on ownership and attribute matching. 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. A default-deny model ensures that access is only granted when explicitly permitted by policy.

Authorization is enforced at three levels: API route middleware, routing table resolution (before resolving a vector store or model to a provider), and storage read time, where query filters are constructed from the current user's attributes so that tenants only see their own or attribute-matched rows. JWT or Kubernetes authentication providers map external identity claims into these attributes, so enterprise identity systems can drive isolation without embedding tenant IDs in application logic.

5.7 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.

5.8 Responses API Implementation

Llama Stack implements the Responses API not only as an agentic execution endpoint but as a resource model encompassing vector stores, files, and conversations—each a first-class, tenant-scoped API object subject to the same ABAC policies as responses themselves (Figure 3). The core endpoint supports creating responses (executing an agent), retrieving, listing, inspecting input items, and deleting responses, in both

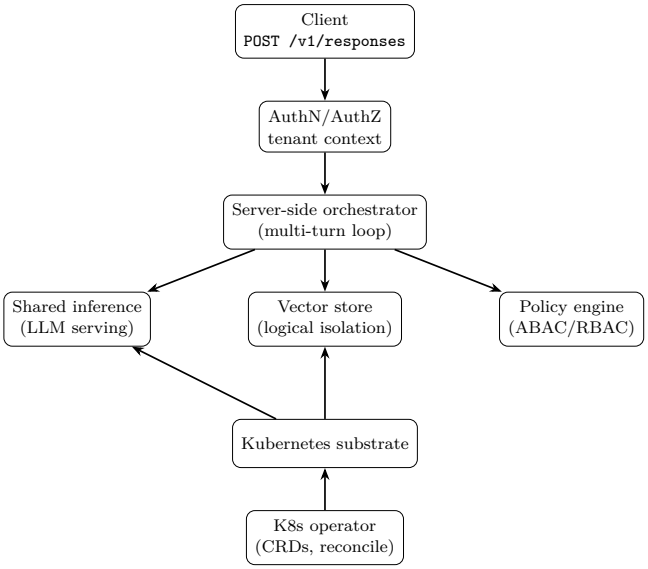

Figure 2: Llama Stack architecture for multitenant enterprise agentic AI on shared Kubernetes infrastructure.

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. Built-in tools (`file_search`, `web_search`, MCP) are resolved and executed server-side during this loop.

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.

5.9 Search abstraction and filters

Llama Stack defines a search abstraction layer that provides uniform access to multiple retrieval modalities: vector databases (ChromaDB, PGVector, Elasticsearch, Qdrant, Weaviate, Milvus, Oracle Cloud Infrastructure, sqlite-vec), keyword search engines, and hybrid pipelines with optional reranking. Advanced chunking strategies (including contextual enrichment) are supported at ingestion time. Each search 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 abstraction supports structured filtering through metadata predicates applied consistently across all search approaches: 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.

5.10 Cloud-Native Deployment

The architectural guarantees described above must be realized on shared infrastructure. Kubernetes has become the de facto substrate for enterprise deployments [3].

The Llama Stack Kubernetes Operator [19] automates deployment and lifecycle management through custom resources that declaratively specify server configurations, backend connections, and isolation policies. The operator supports multiple deployment topologies:

- **Shared instances**: A single deployment serves multiple tenants with application-level ABAC isolation. For example, multiple business units in a regulated organization share one Llama Stack instance, avoiding per-unit infrastructure costs while satisfying compliance requirements.
- **Per-tenant instances**: Separate deployments in tenant-specific namespaces combine Kubernetes RBAC with application-level policies. A platform team offering AI-as-a-service internally can provision per-team distributions, combining namespace-level network isolation with ABAC for defense in depth.
- **Hybrid approaches**: High-security tenants receive dedicated instances while lower-sensitivity tenants share infrastructure.

In all topologies, the provider abstraction allows organizations to start with lightweight backends (e.g., Ollama, sqlite-vec) during development and migrate to production-grade infrastructure (e.g., vLLM [14], pgvector) without modifying agent code or security policies—only the distribution configuration changes.

The operator's provider abstraction enables deployment of heterogeneous inference backends (vLLM [14], Ollama, proprietary endpoints) and vector databases (FAISS [11], ChromaDB, PGVector, Elasticsearch, Qdrant, Weaviate, Milvus, Oracle Cloud Infrastructure, sqlite-vec) as shared Kubernetes services. Multiple Llama Stack instances reference the same backends, achieving cost savings through infrastructure sharing while maintaining logical isolation through the authorization layer.

This design scales economically: adding tenants requires only logical configuration (authorization policies, database schemas) rather than full infrastructure duplication, crucial for cost-constrained enterprise deployments.

6 Analysis and Discussion

6.1 Design trade-offs and limitations

The architecture involves several trade-offs:

- **ABAC policy complexity.** Policy complexity grows with the number of tenants, roles, and resource types. Organizations with deeply nested permission structures may face policy management overhead.
- **Metadata filtering performance.** Filtering performance varies across vector database backends: some support predicate pushdown natively, while others require post-retrieval filtering that adds latency on large corpora.

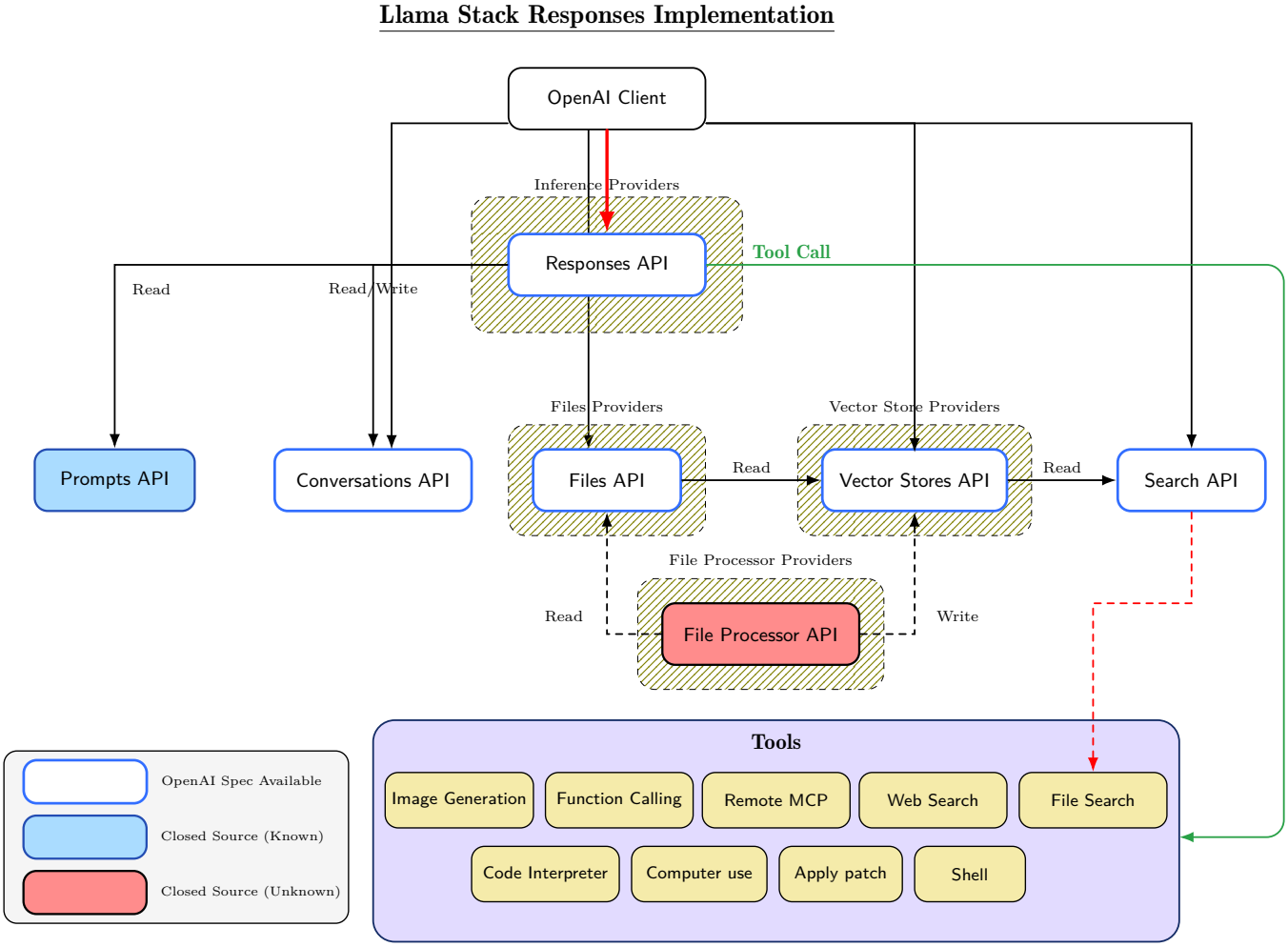

Figure 3: Llama Stack's Responses surface area and its relation to other APIs, providers, and tools.

- **Model prior knowledge.** The inference layer shares models across tenants, which reduces cost but means that model prior knowledge—information learned during pretraining—is orthogonal to RAG isolation and cannot be controlled by the authorization layer.
- **Client-side function tools.** Client-side function tools, by design, execute outside the server trust boundary. Llama Stack mitigates this through explicit tool classification but cannot enforce server-side invariants on client-executed code.

6.2 Related work

Several systems aim to standardize LLM-based agentic applications. The closest to our work are API-layer platforms and enterprise agent runtimes.

OpenAI's Responses API [27], function-calling conventions [25], and the Model Context Protocol (MCP) [2] establish *what* an agent can do but are silent on *who* may do it—they assume a single-tenant caller. Databricks' Mosaic AI Agent Framework [6] wraps agents in an MLflow ResponsesAgent [5, 38] with managed MCP tools and Unity Catalog governance, providing the closest enterprise parallel, but couples the experience to the Databricks stack. Llama Stack adopts the same interface conventions while providing multitenant isolation through an open, vendor-neutral API layer deployable on any infrastructure.

LLM serving engines such as vLLM [14] and Orca [37] optimize inference throughput but are agnostic to tenant isolation; vector databases such as Milvus [39] and Weaviate [34] and platforms such as Vectara [33] rank by relevance without authorization enforcement. Llama Stack treats these as pluggable providers beneath its authorization layer. Agent orchestration frameworks—LangGraph [16], Microsoft's Agent Framework [23] (successor to Semantic Kernel [22] and AutoGen [21]), Google ADK [8], Haystack [7], LlamaIndex [20], Smolagents [10], and Pydantic AI [28]—provide developer-facing abstractions but orchestrate from the client side; Llama

Stack serves as the server-side execution target these frameworks call into.

Production ML research has documented the fragility of distributed invariants [1, 31], and lifecycle platforms such as MLflow [38] address deployment but not agentic multitenancy. To our knowledge, no prior work addresses the intersection of standardized agentic API design, server-side orchestration, and multitenant isolation on shared infrastructure—the gap this paper aims to fill.

7 Conclusion

Enterprise deployment of agentic AI systems introduces security and compliance challenges that existing architectures—designed for single-tenant, consumer-facing use—do not address. This paper formalized the relevance-authorization gap in retrieval [11, 13] and identified shortcomings that arise when agentic systems conflate relevance with authorization in multitenant settings.

To address these shortcomings, we proposed a layered isolation architecture that separates the data path (policy-aware ingestion, retrieval gating, shared inference) from the control path (server-side orchestration for tool execution, state management, and policy enforcement). Rather than positioning server-side orchestration as a replacement for client-side frameworks, we argued that hybrid architectures—where server-side enforcement secures trust-boundary-crossing operations while client-side frameworks retain control over agent composition—represent a practical path forward.

Our open-source implementation through Llama Stack [18] demonstrates that the proposed architecture is realizable in a vendor-neutral framework with server-side Responses API execution [24, 27] and pluggable providers. Combined with efficient serving infrastructure [14] and open models [26], this provides a complete open-source alternative to proprietary agentic platforms, enabling secure multitenancy on shared infrastructure without per-tenant duplication.

References

[1] 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

[2] Anthropic. 2024. Model Context Protocol. Online documentation. https://modelcontextprotocol.io/docs/getting-started/intro Accessed: 2026-02-24.

[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] CrewAI, Inc. 2024. CrewAI: Framework for orchestrating role-playing autonomous AI agents. Open-source project. https://github.com/crewAIInc/crewAI Accessed: 2026-02-23.

[5] Databricks. 2025. Author an agent in code using MLflow ResponsesAgent. Online documentation. https://docs.databricks.com/en/generative-ai/agent-framework/create-agent.html Accessed: 2026-02-24.

[6] Databricks. 2025. Mosaic AI Agent Framework. Online documentation. https://docs.databricks.com/en/generative-ai/agent-framework/index.html Accessed: 2026-02-24.

[7] deepset. 2023. Haystack: End-to-end LLM framework for building production-ready applications. GitHub repository. https://github.com/deepset-ai/haystack Accessed: 2026-02-24.

[8] Google. 2025. Agent Development Kit (ADK). GitHub repository. https://github.com/google/adk-python Accessed: 2026-02-24.

[9] 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

[10] Hugging Face. 2025. smolagents: A smol library to build great agents. GitHub repository. https://github.com/huggingface/smolagents Accessed: 2026-02-24.

[11] 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

[12] 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

[13] 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

[14] 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

[15] LangChain, Inc. 2023. LangChain: Build context-aware reasoning applications. Open-source project. https://github.com/langchain-ai/langchain Accessed: 2026-02-23.

[16] LangChain, Inc. 2024. LangGraph: Build resilient language agents as graphs. Open-source project. https://github.com/langchain-ai/langgraph Accessed: 2026-02-23.

[17] 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

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

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

[20] LlamaIndex. 2022. LlamaIndex: Data framework for LLM applications. GitHub repository. https://github.com/run-llama/llama_index Accessed: 2026-02-24.

[21] Microsoft. 2023. AutoGen: A programming framework for agentic AI. GitHub repository. https://github.com/microsoft/autogen Accessed: 2026-02-24.

[22] Microsoft. 2023. Semantic Kernel: Integrate cutting-edge LLM technology quickly and easily into your apps. GitHub repository. https://github.com/microsoft/semantic-kernel Accessed: 2026-02-24.

[23] Microsoft. 2025. Microsoft Agent Framework. GitHub repository. https://github.com/microsoft/agents Accessed: 2026-02-24.

[24] Open Responses Community. 2026. Open Responses. Online resource. https://www.openresponses.org/ Accessed: 2026-02-23.

[25] OpenAI. 2023. Function Calling. Online documentation. https://platform.openai.com/docs/guides/function-calling Accessed: 2026-02-24.

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

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

[28] Pydantic. 2024. Pydantic AI: Agent Framework / shim to use Pydantic with LLMs. GitHub repository. https://github.com/pydantic/pydantic-ai Accessed: 2026-02-24.

[29] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. 2024. Blended RAG: Improving Retrieval-Augmented Generation through Hybrid Search. *arXiv preprint arXiv:2404.07220* (2024). https://arxiv.org/abs/2404.07220

[30] 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

[31] 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

[32] Natalie Shapira, Chris Wendler, Avery Yen, Gabriele Sarti, Koyena Pal, Olivia Floody, Adam Belfki, Alex Loftus, Aditya Ratan Jannali, Nikhil Prakash, Jasmine Cui, Giordano Rogers, Jannik Brinkmann, Can Rager, Amir Zur, Michael Ripa, et al. 2026. Agents of Chaos. *arXiv preprint arXiv:2602.20021* (2026). https://arxiv.org/abs/2602.20021

[33] Vectara. 2023. Vectara: RAG-as-a-Service Platform. Online. https://vectara.com/ Accessed: 2026-02-24.

[34] Weaviate. 2019. Weaviate: Cloud-native vector database with structured filtering. GitHub repository. https://github.com/weaviate/weaviate Accessed: 2026-02-24.

[35] Simon Willison. 2022. Prompt injection attacks against GPT-3. Blog post. https://simonwillison.net/2022/Sep/12/prompt-injection/ Accessed: 2026-02-24.

[36] 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

[37] 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

[38] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin* 41, 4 (2018), 39–45. https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf

[39] Zilliz. 2019. Milvus: A cloud-native vector database. GitHub repository. https://github.com/milvus-io/milvus Accessed: 2026-02-24.