International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P106
Eureka Vision Publication | Volume 7, Issue 1, 30-47, 2026

Original Article

Silo, Pool, and Bridge for Multi-Tenant RAG: Measuring Isolation, Noisy-Neighbor Effects, and Cost in SaaS Microservices

Ritesh Kumar
Independent Researcher, Pennsylvania, USA.

Received On: 26/11/2025 ***Revised On: 27/12/2025*** ***Accepted On: 04/01/2026*** ***Published On: 17/01/2026***

Abstract - Multi-tenant Retrieval-Augmented Generation (RAG) enables enterprise SaaS platforms to ground large language model outputs in customer-specific data while sharing infrastructure across tenants. This deployment model introduces a hard requirement for strict tenant isolation across storage, embedding generation, vector indexing, retrieval orchestration, and response construction, without unacceptable cost or performance variance under mixed workloads. This paper formalizes three isolation patterns for multi-tenant RAG systems, Silo, Pool, and Bridge, and introduces an isolation taxonomy across four planes: data plane, vector plane, orchestration plane, and LLM plane. A threat model specific to multi-tenant RAG is presented, covering cross-tenant embedding leakage through similarity search, membership inference risk, retrieval contamination from incorrect scoping or poisoned content, and metadata inference. A Kubernetes-native reference architecture is specified to implement tenant-aware controls and explicit policy enforcement points across ingestion and retrieval. The paper also defines an evaluation approach for comparing isolation patterns using leakage testing under adversarial retrieval scenarios, mixed-tenant latency measurements (P50 and P95) to quantify noisy-neighbor effects, cost-per-query decomposition, and operational overhead.

Keywords - Retrieval-Augmented Generation, Multi-tenancy, Tenant isolation, Enterprise SaaS, Vector databases, Embeddings, Access control, Threat modeling, Microservices, Kubernetes, Noisy neighbor effects.

1. Introduction

Retrieval-Augmented Generation (RAG) [1] is widely adopted for grounding large language model outputs in enterprise knowledge sources such as product documentation, support content, contracts, and internal policies. In a typical RAG pipeline [1], user queries are transformed into retrieval requests, relevant content is fetched from a document store or vector index, and selected context is assembled into a prompt that constrains the model response. This design improves factuality and domain alignment relative to prompting alone [3], but it also expands the system boundary. Data flows through ingestion, embedding generation, indexing, retrieval, orchestration, and generation components, each of which can introduce security and performance failure modes.

Most enterprise deployments of RAG are delivered as Software as a Service (SaaS). That delivery model typically requires multi-tenancy [9], where multiple customers share infrastructure to achieve acceptable unit economics, simplified operations, and faster onboarding. Introducing RAG can increase the complexity of multi-tenant isolation because retrieval and prompt assembly add additional enforcement points. The degree of difficulty depends on implementation choices, existing isolation infrastructure, and the capabilities of the selected vector database and serving stack. A correct multi-tenant RAG system must ensure that tenant identity and authorization constraints remain intact across every stage that can influence generated output.

Tenant isolation in this paper is defined as three properties. First, retrieval isolation requires that queries from one tenant must not retrieve content owned by another tenant. Second, context assembly isolation requires that retrieved context included in the prompt must be scoped to the requesting tenant and principal permissions. Third, inference exposure resistance requires that the system reduce the risk of cross-tenant exposure through model outputs, logs, and observable side channels including retrieval behavior and response artifacts.

1.1. Motivation

Cloud and vector database vendors describe practical multi-tenant patterns for SaaS systems and provide prescriptive guidance for RAG components. AWS, Microsoft Azure, Milvus [25], and Pinecone [24] publish architecture documentation identifying Silo, Pool, and Bridge approaches. These materials are useful for implementation, but they stop at architectural recommendations and do not provide a repeatable methodology to compare isolation patterns under adversarial conditions, mixed-tenant load, and explicit cost-per-query accounting. Multi-tenant RAG introduces failure modes less prominent in traditional SaaS designs, such as cross-tenant retrieval leakage from mis-scoped similarity search [10], [11], [12] or retrieval contamination when context assembly includes unauthorized chunks.

A second motivation concerns security modeling. Conventional multi-tenant security analysis focuses on storage isolation and request authorization. RAG requires extending the threat model to cover embeddings [15], vector indices, retrieval-time filtering, and the orchestration layer that constructs the model prompt. Without a threat model specific to these components [14], it is difficult to justify where enforcement must occur and how to validate that isolation holds under normal and adversarial workloads.

Finally, there is an operational motivation. Architects must decide where to place boundaries, which services can be safely shared, which data stores must be partitioned, and what controls are mandatory for safe pooling. Those decisions directly affect tail latency under contention, noisy-neighbor behavior [18], cost drivers such as index footprint and token usage, and the operational overhead of onboarding and maintaining tenants. A structured comparison of Silo, Pool, and Bridge patterns makes those decisions explicit and testable.

1.2. Research Questions

This paper focuses on three questions that arise when building a RAG-powered SaaS product serving multiple tenants on shared infrastructure. The first question asks how Silo, Pool, and Bridge isolation patterns differ in isolation guarantees across storage, embedding generation, vector indexing, retrieval orchestration, and prompt construction. The second question asks what noisy-neighbor effects are measurable under mixed-tenant workloads and which shared components dominate tail latency behavior. The third question asks what the cost-per-query profile of each pattern is and which cost drivers dominate as tenant count, corpus size, and query volume change. These questions are framed to support engineering decisions, and each corresponds to measurable properties that can be validated through leakage testing, latency percentile analysis, and cost decomposition.

1.3. Contributions

This paper makes four contributions. First, it introduces an isolation taxonomy for multi-tenant RAG across four planes: data plane, vector plane, orchestration plane, and LLM plane. The taxonomy provides a consistent vocabulary for specifying what is isolated, where isolation is enforced, and what failure modes remain. Second, it defines three isolation patterns for multi-tenant RAG pipelines: Silo, Pool, and Bridge. Each pattern is described in terms of shared versus tenant-scoped components and the isolation invariants that must hold.

Third, it provides a threat model tailored to multi-tenant RAG. The threat model covers embedding-space and retrieval-specific risks such as cross-tenant retrieval leakage via similarity search, membership inference risk, vector index poisoning, retrieval contamination through incorrect scoping, and metadata inference through observable behavior. Fourth, it specifies a Kubernetes-native reference architecture that implements tenant-aware controls using explicit policy enforcement points across ingestion and retrieval. The paper also defines an evaluation methodology to compare patterns using leakage rate under adversarial retrieval scenarios, latency percentiles (p50 and p95) under mixed-tenant workloads, cost-per-query decomposition, and operational overhead indicators.

2. Background And Related Work

2.1. RAG Pipeline Decomposition

A Retrieval-Augmented Generation (RAG) system [1] transforms raw enterprise documents into grounded model responses through a sequence of stages. The ingestion stage accepts documents from upstream sources, segments them into chunks, attaches metadata required for isolation and governance, and generates embeddings that encode chunk semantics. Chunking strategies include fixed token windows, boundary-aware segmentation, semantic chunking based on embedding similarity, recursive chunking, and parent-child hierarchical approaches. This list is not exhaustive; chunking affects retrieval granularity and index size but does not change the isolation requirements defined in this paper.

The storage stage persists two distinct data types. A document store retains raw text or chunk payloads and associated metadata, commonly using object storage or a document database. A vector store maintains embeddings and identifiers that link vectors back to source chunks. Some deployments co-locate payload and vector data in one system, while others separate them to scale and secure each tier independently. The indexing stage builds structures that accelerate similarity search in high-dimensional embedding spaces. Approximate nearest neighbor (ANN) methods are common, including graph-based indexing such as Hierarchical Navigable Small World (HNSW) [4] and cluster-based inverted file methods such as Inverted File Index (IVF). HNSW organizes vectors into a navigable graph to improve query latency [4]. IVF partitions vectors into coarse clusters and searches a subset of clusters per query to reduce comparisons. Some systems also use hybrid retrieval [2], where dense similarity search is combined with lexical retrieval. This can improve robustness for certain query classes but adds orchestration complexity and cost accounting at query time.

The retrieval and orchestration stage processes incoming queries, encodes them into the embedding space, executes similarity search, optionally reranks results using cross-encoder models that typically require access to chunk text rather than embeddings alone, and assembles retrieved chunks into a context window. This stage is a primary isolation boundary because any cross-tenant retrieval error can directly introduce unauthorized content into the prompt. The generation stage constructs the final prompt by combining system instructions, retrieved context, and the user query, then invokes the LLM for inference and applies post-processing such as output filtering, citations, and audit logging. Canonical RAG formulations [1] explicitly treat retrieval as a first-class component whose outputs condition generation, which is why retrieval-time controls must be treated as part of the security and isolation model.

For clarity, a document refers to the original ingested content. A chunk is a segmented unit derived from a

document and used as the retrieval unit. A payload refers to the text content of a chunk, distinct from its embedding and metadata. Tenant scoping refers to enforcing that data access and retrieval operations are constrained to a single tenant; the terms tenant filter and tenant discriminator are used as equivalent mechanisms for tenant scoping.

2.2. Multi-Tenant SaaS Model

Multi-tenancy in SaaS systems [9] is commonly implemented using one of three isolation approaches: database-per-tenant, schema-per-tenant, or shared database with row-level separation. Stronger physical separation generally improves isolation and reduces blast radius but increases cost and operational overhead as tenant counts grow. Shared storage with row-level separation improves resource efficiency but shifts isolation responsibility into application logic, query correctness, and enforcement depth.

Tenant identity in enterprise SaaS is typically represented by a tenant identifier for the customer organization plus principal identifiers for users or service accounts, with roles and scopes that constrain access. In distributed microservice architectures, this identity must propagate across service boundaries so downstream components can enforce authorization consistently. RAG systems intensify this requirement because multiple services participate in retrieval and context assembly, and any stage that loses or misroutes tenant context can create cross-tenant exposure.

A common architectural separation divides a shared control plane from a tenant-scoped data plane. The control plane manages onboarding, configuration, and platform services. The data plane hosts customer workloads and data, and is where retrieval and prompt assembly must enforce tenant isolation. The boundary between these planes determines which components can be pooled and which require per-tenant deployment, and it strongly influences cost allocation, observability, and incident response.

2.3. Vector Database Isolation Mechanisms

Vector databases provide multi-tenancy primitives that parallel traditional isolation models, but operate over embedding stores and similarity search paths. One approach is namespace-style logical separation, where each tenant's vectors are stored in a distinct namespace within shared infrastructure. Pinecone documents namespace-based multitenancy [24], where queries are scoped to a namespace (within a single index) to prevent cross-namespace retrieval by construction. Pinecone provides namespace-based isolation within a single index, allowing logically separate vector sets to share the underlying index infrastructure.

Another approach is partition-based separation within collections. Milvus documents multi-tenancy strategies [25] using partitions or partition keys to target queries to tenant-specific partitions. Partition-based strategies must account for platform limits; Milvus documentation notes that a collection can hold up to 1,024 partitions per collection [25], which constrains partition-per-tenant designs at high tenant counts and can influence pattern selection. Collection-level separation, where each tenant has a dedicated collection, can provide clearer boundaries but can increase operational and memory overhead as tenant count grows.

Metadata filtering [6], [7] is a widely used mechanism across vector systems. In this model, all tenant vectors coexist in shared indexes and each query includes a predicate such as tenant_id equals X. This maximizes index sharing and can reduce per-tenant overhead, but it raises the consequence of filter omission or misapplication. It also increases the importance of defense-in-depth validation, such as verifying that returned results match the request tenant context before context assembly.

Relational vector stores such as PostgreSQL with pgvector [21] can use database-native access control. PostgreSQL row-level security [20] allows policies that restrict which rows can be returned based on roles or session context. AWS explicitly describes using row-level security to enforce tenant isolation in a pgvector-based multi-tenant design. These mechanisms can provide strong enforcement at the data access layer, but they still require correct tenant context propagation and auditing in the orchestration layer.

2.4. Related Work and Gap Analysis

Prior research addresses important aspects of multi-tenant behavior in systems adjacent to the end-to-end architecture question. For multi-tenant RAG efficiency, recent work [18] analyzes caching and fairness in multi-tenant RAG deployments and quantifies efficiency and tenant-level fairness trade-offs under shared workloads. This motivates treating performance isolation and tail latency as first-class evaluation dimensions, not secondary concerns.

For multi-tenant vector indexing, Curator [8] examines the trade-off between per-tenant indices and shared indices with filtering, proposing indexing techniques intended to reduce overhead while preserving tenant-level performance characteristics. This is directly relevant to the Pool versus Silo decision at the vector plane and provides a basis for analyzing index footprint and query-time behavior under multi-tenant load.

For RAG security, membership inference attacks against retrieval corpora and RAG systems are studied in the literature [10], [11], [12], including settings where attackers attempt to infer whether specific documents are present in the retrieval database by probing the system and observing outputs. Additional work proposes membership inference frameworks for RAG-based systems under black-box and grey-box assumptions. These results support treating retrieval behavior and response artifacts as part of the attack surface, alongside conventional access control.

The gap addressed in this paper is the lack of an integrated, end-to-end treatment that aligns three elements in a single engineering framework:

1) A plane-based isolation taxonomy covering data, vector, orchestration, and LLM concerns.

2) A threat model mapped to concrete enforcement points in a microservice RAG architecture.
3) A repeatable evaluation methodology for comparing Silo, Pool, and Bridge patterns across leakage testing, noisy-neighbor effects, cost-per-query decomposition, and operational overhead.

3. Isolation Taxonomy across Four Planes

Multi-tenant RAG isolation is an end-to-end property. It must hold across every component that can influence retrieval results, prompt construction, and generated output. Treating isolation as a single database setting or a single gateway check is insufficient because RAG systems [1] introduce additional state and decision points, including embeddings, vector indices, reranking, and context assembly. To make requirements explicit and testable, this section defines a four-plane isolation taxonomy. Each plane represents a category of resources and operations where tenant separation must be maintained, and where specific failure modes tend to appear in production SaaS deployments.

3.1. Isolation Planes Definition

3.1.1. Data Plane

The data plane covers persistence and access of tenant documents, derived chunks, and authoritative metadata, including access control attributes and lineage. Isolation at this plane determines whether tenant data is separated through dedicated storage boundaries or through logical scoping in shared stores. In a store-per-tenant model, each tenant is assigned a dedicated database, bucket, container, or equivalent boundary. This reduces blast radius and simplifies some audits, but it increases operational overhead and can duplicate baseline capacity. In a shared-store model [9], multiple tenants share the same storage system and isolation relies on a tenant discriminator plus authorization policies that are applied on every access path.

Chunking introduces a practical requirement that is easy to miss. Retrieved units are typically chunks, not whole documents, so chunks must carry tenant identity and authorization-relevant metadata end-to-end. If chunk payloads and chunk metadata are stored in different systems, the architecture must define which system is the source of truth for enforcement, and how consistency is maintained under updates and deletions.

Authorization patterns at this plane are commonly row-level for relational stores [20] and object-level for document and object stores. Row-level approaches can bind access to session context or database roles, while object-level checks often execute in a storage adapter or policy service. Encryption boundaries also belong in the data plane. Per-tenant keys reduce the impact of key compromise for certain threat scenarios. Shared keys with per-tenant derivation can reduce key management overhead, but increase reliance on correct derivation, rotation, and correct use across all services.

Common data plane failure modes are operational and consistency-related. Examples include missing tenant scoping predicates, stale ACL state after permission changes, and inconsistent metadata between the document store and the retrieval metadata store. These failures are high impact in RAG because they can directly affect which chunks become eligible for retrieval and prompt inclusion.

3.1.2. Vector Plane

The vector plane covers embeddings, vector persistence, index organization, and similarity search behavior under tenant scoping. Isolation choices at this plane have strong cost and performance implications because vector indices are often memory intensive and retrieval latency sensitive. The primary architectural decision is between per-tenant indices and shared indices with tenant-aware filtering [8]. Per-tenant indices reduce reliance on query-time filtering for isolation and can simplify correctness validation, but they increase index duplication and operational work as tenant counts grow. Shared indices improve resource sharing and can reduce baseline memory footprint, but place strict requirements on filter correctness, filter placement in the query path, and validation of results before context assembly.

Indexing mechanisms influence performance under scoping. For example, graph-based [4] and cluster-based ANN approaches reduce search cost by pruning the candidate set. Tenant scoping can be implemented by selecting the correct tenant-specific index or partition, or by filtering within a shared index [6], [7]. From an isolation perspective, the retrieval path must not return cross-tenant candidates to orchestration. From a performance perspective, tenant scoping must not destabilize tail latency. This risk is highest when large and small tenants share the same retrieval infrastructure and compete for the same index and compute resources.

Partitioning strategies include namespace separation [24], collection separation [25], shard-level separation, and key-based segmentation. Each strategy shifts the default isolation properties and changes operational scaling behavior. The embedding pipeline also introduces isolation considerations because embedding generation is often centralized for efficiency. If embedding computation and caching are shared, boundaries must prevent incorrect reuse across tenants and must ensure that tenant context is not lost during asynchronous ingestion.

A recurring vector plane failure mode is late or inconsistent filtering. When tenant filters are applied only after a broad candidate set is generated, cross-tenant vectors may be processed before being discarded, which increases side-channel exposure risk. Post-filtering can also increase latency substantially when a tenant's vectors are a small fraction of a shared index because the system may scan many irrelevant candidates before collecting enough tenant-scoped results [7]. This complicates correctness validation and increases the importance of defense-in-depth checks in orchestration before any retrieved content can influence prompt construction.

3.1.3. Orchestration Plane

The orchestration plane spans the services that coordinate ingestion workflows, query processing, retrieval, reranking, and context assembly. Isolation at this plane depends on tenant identity being a first-class control signal. Tenant identity is established at the authentication boundary, typically at an API gateway, and must propagate through every downstream service call, message, and asynchronous job that can influence retrieval or prompt assembly.

Policy enforcement point placement must be explicit. Enforcement often occurs at the gateway for authentication and coarse authorization, at the retrieval service for query scoping and validation, at the vector query layer for tenant-aware search constraints, and at storage adapters for payload fetch authorization. Defense in depth is essential because a single missed check in a pooled system can lead to cross-tenant retrieval or context mixing.

Query routing and scoping are core orchestration responsibilities. In per-tenant index designs, routing must select the correct tenant index or partition deterministically. In shared-index designs, routing must attach the correct tenant filter and ensure that the filter is applied on all query variants, including hybrid retrieval and reranking paths. Context assembly is the highest sensitivity step because it determines what enters the model prompt. Context assembly rules must require provenance validation, must reject any chunk that fails tenant or principal authorization checks, and must ensure that only authorized chunks can be included in the final context window.

A deterministic audit record is required for validation and incident response. At minimum, it should capture tenant identity, authorization context, filters applied, retrieved item identifiers, and which items were included in the final prompt.

3.1.4. LLM Plane

The LLM plane covers prompt construction, inference execution, response filtering, and telemetry. Multi-tenancy at this plane is typically implemented through policy scoping rather than physical separation, although tenant-dedicated endpoints are possible in high-isolation deployments. Shared model endpoints can be cost efficient, but require strict controls on tenant-specific prompt templates, tool access, quota enforcement, and logging practices. Tenant-dedicated endpoints reduce shared resource contention, but increase operational overhead and can complicate model lifecycle management across a broad tenant population.

Prompt construction isolation requires that system prompts, tool configurations, and retrieved context are scoped to the tenant and the requesting principal's permissions. Rate limiting, quotas, and fair scheduling [18] belong in this plane because inference capacity is often a dominant contributor to tail latency and cost-per-query. Response filtering and tenant-scoped output validation are required to reduce the risk of unintended disclosure through generated output artifacts. Logging and redaction requirements must be explicit because prompts and retrieved context can contain sensitive tenant data, and shared operational tooling can become an indirect exposure path if logs are not properly scoped and protected.

Inference-time isolation extends beyond prompt content to execution state. In shared serving deployments, key-value (KV) cache optimizations [17] such as prefix sharing and cache reuse can introduce side channels [16]. If cache entries are not tenant-partitioned, one tenant may infer information about another tenant's prompt through cache timing or hit-rate behavior [16]. This risk is most relevant when multiple tenants share the same model serving stack.

3.2. Pattern Definitions

3.2.1. Silo Pattern

The silo pattern dedicates resources per tenant across all four planes for components that process tenant content. A shared control plane that handles tenant onboarding, configuration distribution, and operational tooling, but never processes tenant documents or tenant queries, may still be used to reduce operational overhead. Document and metadata storage are tenant-scoped. Embedding indices and vector storage are tenant-scoped. Orchestration is either tenant-dedicated or implemented with strict tenant-specific routing and state boundaries. Model inference can be tenant-dedicated or strongly partitioned through policy and resource controls. The main advantage is reduced blast radius and reduced dependence on correct runtime filtering for isolation. The main trade-offs are higher baseline cost due to duplicated infrastructure and higher operational overhead due to per-tenant lifecycle management, scaling, and configuration.

3.2.2. III.B.2. Pool Pattern

The Pool pattern shares infrastructure across tenants and enforces isolation through tenant discriminators, authorization checks, and runtime filtering. The document store, vector index, orchestration services, and model endpoints may all be shared. Isolation is therefore primarily logical and depends on the correctness of identity propagation and enforcement at multiple checkpoints. Pool can offer strong efficiency and simplified infrastructure, but it increases the consequences of misconfiguration and tends to expose tenants to higher noisy-neighbor risk [18] because retrieval and inference resources are contended. In Pool deployments, defense-in-depth validation, provenance-based context assembly, and auditable enforcement are mandatory controls, not optional enhancements.

3.2.3. Bridge Pattern

The Bridge pattern is a hybrid that combines pooled services with selected tenant-scoped components. Typical variants include shared orchestration with tenant-scoped vector indices, or shared retrieval services with tenant-scoped document stores. Bridge is used when Pool is too risky for some tenants or workloads and Silo is too costly for the full tenant population. Tiering criteria often include regulatory requirements, data sensitivity, workload predictability, and performance SLO strictness. Because Bridge spans both pooled and tenant-scoped boundaries, routing, policy enforcement, and auditability must remain consistent across tiers. Migration paths must be designed explicitly because

moving tenants between tiers can require index rebuilds, key management changes, and configuration updates across services.

3.3. Pattern Comparison Matrix

Table I summarizes the three patterns across the four planes and key operational dimensions. The matrix is intended as a decision aid and as a checklist for evaluation. Isolation properties should be treated as expected characteristics that still require validation, especially in Pool and Bridge configurations where enforcement depends on correct propagation and policy placement.

Table 1:Isolation Models in Multi-Tenant AI Architectures: Comparative Analysis of Silo, Pool, and Bridge Approaches

Dimension	Silo	Pool	Bridge
Data plane isolation	Tenant-scoped stores	Shared store with tenant scoping	Tier-dependent
Vector plane isolation	Tenant-scoped indices	Shared index with filtering	Often tenant-scoped for high-isolation tiers
Orchestration plane isolation	Tenant-dedicated or strongly partitioned routing	Shared services with strict identity propagation and validation	Shared services with tier-aware routing and controls
LLM plane isolation	Tenant-dedicated or strongly partitioned policies	Shared endpoint with tenant-scoped policies	Tier-dependent
Primary failure mode	Provisioning and routing errors within a tenant boundary	Filter omission, context mis-scoping, identity propagation defects	Tier boundary errors and inconsistent enforcement across tiers
Blast radius	Primarily per tenant	Potentially multi-tenant	Typically bounded to a tier population
Cost drivers	Infrastructure duplication, per-tenant index footprint	Shared index and shared inference capacity	Mixed duplication and shared contention
Operational overhead	High per-tenant lifecycle management	Lower shared operations, higher validation burden	Moderate, plus tier management
Compliance fit	Strongest by default	Requires strong controls and evidence	Tier-dependent

4. Threat Model for Multi-Tenant Rag

This section defines a threat model for multi-tenant RAG systems with an emphasis on where isolation breaks in practice and how those failures propagate into prompts and generated outputs. The intent is not to exhaustively enumerate every security issue in distributed systems, but to focus on threats that are either unique to RAG [14] or amplified by retrieval, embeddings, and context assembly.

4.1. Threat Model Scope and Assumptions

The protected assets in scope include tenant documents, derived chunks, chunk metadata, embeddings, vector indexes, prompts, model outputs, logs and traces, and encryption keys. Loss of confidentiality is the primary concern because cross-tenant exposure is the critical failure mode in multi-tenant SaaS. Integrity is also in scope because poisoned or manipulated content [13] can alter retrieval results and lead to incorrect or unsafe responses.

The threat actors considered are a malicious tenant acting through legitimate APIs, an external attacker who has obtained tenant credentials or can exploit exposed interfaces, and an insider with elevated operational access. Adversary capabilities range from repeated probing of the retrieval and generation interfaces, to content injection via the ingestion pipeline, to attempts to exploit misconfigurations in identity propagation, routing, filtering, or logging. The model treats the adversary as capable of generating large numbers of requests and observing system behavior, including response content, latency, and error messages, within the limits of SaaS rate controls.

Trust boundaries are defined at the API boundary where authentication and tenant context are established, at service-to-service boundaries inside the microservice mesh, at the

TABLE 1. PATTERN COMPARISON MATRIX

data plane boundary for document and metadata access, at the vector database boundary for similarity search, and at the LLM endpoint boundary where prompts are submitted and outputs are returned. Logging and telemetry pipelines are treated as an additional operational boundary because they can contain sensitive prompts and retrieved context.

The baseline assumption is that transport is encrypted in transit using Transport Layer Security (TLS) and that standard cryptographic primitives are not broken. The model does not assume perfect correctness in policy configuration or perfect correctness in distributed propagation of tenant context. Misconfiguration and partial failure are treated as realistic, because they are common root causes of multi-tenant incidents.

4.2. Embedding-Space Vulnerabilities

Embedding-space vulnerabilities arise because retrieval depends on similarity search over shared or partially shared vector structures. This creates failure modes where the system can expose cross-tenant content directly through retrieved chunks, or indirectly through observable retrieval behavior and output artifacts. OWASP's guidance on generative AI security [19] explicitly calls out weaknesses associated with vectors and embeddings, which aligns with treating the vector plane as part of the attack surface rather than a neutral storage layer.

4.2.1. Cross-Tenant Embedding Leakage

Cross-tenant embedding leakage occurs when similarity search returns vectors or chunk identifiers that belong to a different tenant than the requester. In pooled deployments, the dominant attack vector is a missing, malformed, or bypassed tenant filter, or a misrouted tenant identifier that causes a query to execute against the wrong namespace, partition, or index. A second class of failures appears when filtering is applied inconsistently across retrieval variants, such as hybrid retrieval, reranking, or fallback paths.

The impact ranges from direct disclosure of content, if payload fetch is performed without an additional tenant check, to indirect disclosure of document identifiers, titles, or metadata if those fields are returned in retrieval results or logs. The most useful detection signal is retrieval provenance that includes tenant identity and retrieved item identifiers. If a retrieval log or trace shows chunk identifiers mapped to a different tenant than the request tenant, the system has a measurable isolation violation. This is why deterministic audit records are treated as part of the isolation model, not a monitoring convenience. Additionally, embedding inversion attacks [15] show that embeddings can leak substantially more than similarity metadata. Under some conditions, reconstructed text can be recovered from embeddings with meaningful fidelity [15], so embeddings should be treated as sensitive tenant data rather than benign derived features.

4.2.2. Membership Inference

Membership inference in RAG refers to attempts to infer whether a target document, or a semantically related document, exists in another tenant's retrieval corpus by probing the system and observing outputs or retrieval behavior. Prior work [10], [11], [12] studies membership inference against RAG systems, including black-box and grey-box settings, and demonstrates that retrieval behavior and downstream outputs can leak information about the presence of documents in the underlying corpus. In a multi-tenant context, this becomes a cross-tenant concern when an attacker can influence or observe retrieval outcomes beyond its own tenant boundary, or when system-level telemetry and error behavior reveal corpus characteristics.

The primary attack vectors include repeated probing with semantically targeted queries, observing response differences that correlate with retrieval hits, and exploiting confidence signals or debugging fields if the system exposes them. Even when content is not directly disclosed, corpus membership can leak competitive information, such as whether a tenant has documents related to a product line, acquisition, or incident.

Detection signals include anomalous query patterns, repeated near-duplicate queries, and probing workloads that sweep a semantic neighborhood. Mitigation is primarily architectural and operational. It requires strict tenant scoping, strict suppression of cross-tenant retrieval artifacts, careful control of debug outputs, rate limits tuned for probing resistance, and audit trails sufficient to identify probing behavior.

4.2.3. Vector Index Poisoning

Vector index poisoning [13] occurs when an attacker injects crafted content through the ingestion pipeline to influence retrieval results, degrade retrieval quality, or cause systematic misdirection of responses. In pooled indexes, poisoning can also become a cross-tenant integrity issue if shared retrieval infrastructure allows poisoned vectors to appear in the candidate set for other tenants due to filtering errors or shared reranking paths. The direct impact can include degraded relevance, denial of service through index bloat or retrieval hotspots, and response manipulation if poisoned chunks are repeatedly selected into prompts.

This threat is best addressed with layered controls. Ingestion must enforce tenant-scoped authorization, content validation, and rate controls. Indexing must ensure that tenant scoping is correct and that updates are auditable. Retrieval must validate provenance and enforce deny-by-default behavior when scope is ambiguous. Operationally, the system should support rollbacks or quarantine of recently ingested content for a tenant when abnormal retrieval patterns are detected.

This subsection focuses on integrity and isolation risks. A related availability risk is computational denial of service, where adversarial embeddings or documents are crafted to increase retrieval cost, expand candidate sets, or degrade index performance. These attacks impact tail latency and shared resource stability and should be evaluated as part of capacity protection and abuse controls.

4.2.4. LLM Serving Side Channels

In shared inference deployments, optimization techniques for KV-cache management [17] can create cross-tenant leakage paths [16]. Prefix sharing and cache reuse improve throughput by avoiding redundant computation for common prompt prefixes, but they may allow one tenant to infer portions of another tenant's prompt through cache timing or hit-rate observation [16]. Mitigations include tenant-isolated inference sessions, strict cache partitioning, or disabling cross-tenant cache reuse. Dedicated model endpoints eliminate this vector by construction.

4.3. Data-Plane Vulnerabilities

Data-plane vulnerabilities often appear as traditional authorization failures, but their consequence in RAG is amplified because they can propagate into prompts and

outputs. Two classes are particularly relevant to multi-tenant RAG: retrieval contamination and metadata inference.

4.3.1. Retrieval Contamination

Retrieval contamination occurs when chunks from the wrong tenant, or chunks that the requesting principal is not authorized to access, enter the context window used for generation. This can happen even if vector retrieval returns correct candidates, for example if payload fetch uses a different authorization path, if ACL evaluation is inconsistent across services, or if asynchronous pipelines produce stale permission state. Orchestration bugs, such as race conditions around ACL updates or incorrect cache scoping, can also cause contamination.

The impact is direct data leakage in generated responses and potential regulatory non-compliance, since the prompt includes unauthorized content. The most effective defense is to treat context assembly as a policy-enforced operation. Before any chunk is included in the prompt, the system should validate tenant ownership and principal authorization using an authoritative policy decision path, and record the provenance decision in an audit log.

4.3.2. Metadata Inference

Metadata inference refers to learning sensitive information about another tenant without directly accessing its content. Examples include inferring tenant activity levels, document counts, update frequency, or query volume. Attack vectors include timing analysis, observing resource consumption patterns, and exploiting error message differences. In multi-tenant systems, metadata leakage can occur through shared rate limit behavior, shared queue latency, shared index maintenance events, or unscoped operational metrics.

The impact is competitive intelligence and usage profiling. Mitigation includes scoping operational metrics by tenant and access role, reducing high-cardinality exposure in shared dashboards, standardizing error responses, and using quotas and scheduling policies that reduce observable coupling between tenants.

4.3.3. Controls Mapped to Enforcement Points

Controls are most effective when mapped to explicit enforcement points and treated as invariants. Tenant identity should be established once at the authentication boundary and propagated as immutable request context through service-to-service calls. In practice, this commonly uses signed tokens or signed headers with strict validation at each hop. Identity propagation must also cover asynchronous ingestion paths, including job queues and batch processors, because ingestion is a write path into the retrieval corpus.

Authorization checks should execute at multiple layers [9]. The API gateway should enforce authentication and coarse access controls. The retrieval service should enforce tenant-scoped query construction and validate the scope of results. Storage adapters should enforce tenant and principal authorization on payload fetch, even if vector retrieval already applied filters. Where supported, database-level policies such as row-level security [20] can provide an additional layer of enforcement, but they should be treated as defense-in-depth rather than the only control. Defense-in-depth is widely recommended [9], [14] because single enforcement points are vulnerable to misconfiguration, and redundant checks reduce the likelihood that one defect results in cross-tenant exposure.

Vector filtering must be designed so that tenant scope is not optional. When possible, filters should constrain candidate generation [6], [7], not only filter after scoring. When filters cannot be applied early due to datastore limitations, post-filter validation must be strict and must fail closed. Fail-closed behavior prioritizes isolation over availability. Operators should monitor rejection rates, configure alerting thresholds, and define fallback policies or graceful degradation paths for transient identity propagation failures. Any retrieved item that does not match tenant scope should be rejected before context assembly, and the event should be logged as a policy violation signal. Output controls reduce exposure through generated text and telemetry. These include redaction policies, citation and provenance constraints, and response policy checks aligned to tenant configuration. Logs and traces must be scoped, access-controlled, and redacted to prevent operational exposure of prompts and retrieved context. Audit logging should capture retrieval provenance, policy decisions, and trace identifiers so that violations can be detected and reconstructed.

4.4. Pattern Resilience Analysis

Resilience differs across Silo, Pool, and Bridge patterns primarily through blast radius and dependence on correct runtime filtering. Silo reduces cross-tenant exposure risk by limiting shared data and shared indexes, but it still requires correct identity and authorization within each tenant boundary. Pool has the highest dependence on correct tenant context propagation, correct filtering, and strict context assembly validation. Bridge inherits both modes. It can reduce blast radius for tenants placed in higher isolation tiers, but it introduces tier boundary risks where routing, enforcement, and auditability must remain consistent across pooled and tenant-scoped components.

Table II summarizes expected resilience properties. The entries are expressed as expected characteristics that still require validation and continuous testing.

Table 2: Threat Resilience by Isolation Pattern

Threat	Silo	Pool	Bridge
Cross-tenant embedding leakage	Lower likelihood due to tenant-scoped indices and stores	Higher likelihood if filters or routing fail	Tier-dependent, reduced for tenant-scoped vector tiers
Membership inference risk	Reduced cross-tenant exposure paths, still requires output and telemetry controls	Higher risk if retrieval artifacts or behavior leak across tenants	Tier-dependent, shared inference and telemetry can dominate
Vector index poisoning	Contained to a tenant boundary when ingestion and indexing are tenant-scoped	Can affect shared infrastructure and shared quality signals, cross-tenant impact if scoping fails	Typically contained within tier, but shared services can propagate effects
Retrieval contamination	Primarily within-tenant if boundaries are correct	Cross-tenant impact possible if orchestration validation is weak	Tier-dependent, boundary and routing correctness is critical
Metadata inference	Reduced coupling between tenants	More coupling through shared resources unless mitigated	Reduced for isolated tiers, shared components still leak metadata without controls

5. Kubernetes-Native Reference Architecture

This section specifies a Kubernetes-native reference architecture for multi-tenant RAG that supports Silo, Pool, and Bridge isolation patterns. The design goal is to make tenant isolation enforceable and auditable by placing policy checks at multiple points in both ingestion and retrieval paths, and by treating tenant identity as immutable request context rather than an optional application field.

5.1. Architecture Overview

The architecture decomposes the system into a small set of microservices with explicit responsibilities. The ingestion service accepts documents and produces chunk payloads plus metadata. The embedding service computes embeddings for chunks and writes vectors with tenant-scoped metadata. The indexer manages vector index updates and compaction. The retrieval service executes tenant-scoped similarity search and optional reranking, and returns provenance-tagged candidates. The prompt builder performs context assembly and constructs the final prompt based on tenant policy. The LLM gateway invokes the model endpoint and applies output controls, including redaction and audit logging. A policy service provides authorization decisions and policy configuration, while a tenant registry resolves tenant tier, routing targets, and keying material references.

Data is persisted in three logical stores. The document store retains raw documents and chunk payloads. The metadata store retains authoritative chunk metadata, including tenant ownership, ACL attributes, lineage, and timestamps. The vector database stores embeddings and supports similarity search with tenant scoping. These stores can be deployed as tenant-dedicated or shared depending on the isolation pattern. A shared control plane supports tenant lifecycle operations through a tenant registry, configuration service, and secrets management system.

Tenant identity is a first-class control signal. It is established at the authentication boundary and propagated end-to-end as immutable context [9]. Services do not accept tenant identity from untrusted request fields. They accept only a validated tenant context derived from authenticated credentials, then enforce scoping at every data access and retrieval action. This design aligns with the threat model [14] by reducing filter omission risk, limiting blast radius, and enabling deterministic auditing.

A practical way to keep the design verifiable is to define explicit enforcement points and require fail-closed behavior. If tenant context is missing, inconsistent, or unverifiable, the request is rejected before any retrieval or payload fetch occurs. Fail-closed behavior prioritizes isolation over availability. Operators should monitor rejection rates, configure alerting thresholds, and define fallback policies or graceful degradation paths for transient identity propagation failures. If retrieved candidates fail provenance validation, they are rejected and the event is recorded as a policy violation signal.

5.2. Ingestion Path

The ingestion path is a write path into the retrieval corpus and must enforce tenant ownership, access control metadata integrity, and auditability. In multi-tenant deployments [9], ingestion is also a common source of cross-tenant contamination because content is transformed into chunks and then indexed for later retrieval.

5.2.1. Document Intake Service

The document intake service is the entry point for tenant content. It authenticates the caller, resolves tenant identity, and validates that the caller is authorized to ingest content for that tenant. It then performs document normalization and chunking, and attaches required metadata to each chunk. The minimum metadata set includes tenant ID, document ID, chunk ID, ACL attributes, lineage identifiers, and timestamps for creation and update. The service routes payloads and metadata to the correct storage boundary based on the tenant's isolation tier. In Silo, routing targets tenant-dedicated stores. In Pool, routing targets shared stores with tenant discriminators. In Bridge, routing targets are tier-specific and must be derived from the tenant registry rather than configuration embedded in the client.

5.2.2. Embedding Generation Service

The embedding service accepts tenant-scoped chunk references and computes embeddings. It must treat tenant context as mandatory input and must not generate or cache embeddings in a way that allows cross-tenant reuse. If batching is used for efficiency, batching must not merge tenant contexts in a way that weakens auditability or causes ambiguous attribution. Each embedding write must include tenant ID and chunk identifiers that allow downstream provenance checks to validate ownership. The embedding service should write embeddings to the vector DB and write embedding metadata to the metadata store, enabling later verification that a retrieved vector corresponds to an authorized chunk.

5.2.3. Vector Indexing Service

The indexing service manages index updates, compaction, and any background maintenance that affects retrieval behavior. It selects the correct index boundary based on the isolation pattern. In Silo, each tenant has a dedicated index boundary. In Pool, tenants share index infrastructure and rely on filtering and validation. In Bridge, index boundaries are tier-specific, and the indexer must enforce that vectors are written only into the tenant's permitted tier. Index update operations must be auditable and reversible in the operational sense. At minimum, indexer actions should be traced with tenant context, index identifiers, and the source batch lineage so that poison or contamination events [13] can be investigated and scoped.

5.3. Retrieval Path

The retrieval path is the highest sensitivity path because it selects content that will be inserted into the prompt. Isolation failures at this stage can lead to cross-tenant retrieval leakage and retrieval contamination [14].

5.3.1. Query Gateway

The query gateway authenticates the request and constructs an immutable tenant context from validated credentials, such as JSON Web Token (JWT) claims or API key mappings. It applies per-tenant rate limits and quotas to reduce noisy-neighbor effects [18] and probing risk [10], [11], [12], and it normalizes request inputs to reduce injection and parsing ambiguity. The gateway also enforces coarse authorization, such as whether the principal can invoke retrieval for a given tenant and which collections or knowledge sources are in scope. The gateway emits a trace identifier that is propagated through the full request path to support deterministic auditing.

5.3.2. Retriever Service

The retriever service performs tenant-scoped search against the vector database and any optional sparse index. Tenant scoping is applied before similarity search when the datastore query model supports it [6], [7]. If pre-filtering is not available, the service performs post-filter validation and rejects any cross-tenant candidates. Results are forwarded to context assembly only after tenant ownership and authorization constraints are satisfied. The retriever should return candidates together with provenance fields required for downstream validation, including chunk ID, document ID, tenant ID, and the filter predicate applied. A defense-in-depth measure [9], [14] is to perform cross-tenant result detection as a separate validation step, where the retriever explicitly checks that the returned candidate set matches tenant ownership and logs any mismatch as a policy violation signal.

If reranking is used, reranking must not weaken scoping guarantees. The reranker should operate only on tenant-validated candidates and should not introduce additional retrieval calls that bypass the tenant filter path. If the reranker relies on external models, the request payloads must be treated as sensitive and must follow the LLM plane logging and redaction constraints.

5.3.3. Context Assembly

Context assembly is responsible for constructing the context window that will condition generation. It is the final gate before the model sees any retrieved content, so it must enforce tenant and principal authorization deterministically. Context assembly should verify chunk provenance against the metadata store [20], reject any chunk that fails tenant ownership or ACL checks, and record the provenance decision trace. Context size management should be tier-aware. Bridge deployments often allocate larger context windows or higher retrieval depth to certain tiers, but this must be driven by tenant policy rather than request-controlled parameters.

5.3.4. LLM Gateway

The LLM gateway constructs the final prompt using tenant-scoped system prompts, tenant-specific tool configuration, and the validated context window. It enforces per-tenant quotas for inference and token usage [18], and applies response post-processing such as redaction, policy checks, and citation formatting when enabled. It must produce audit logs that associate the response with the tenant context, the provenance identifiers of included chunks, and the policy checks applied. Logging must be configured to avoid storing raw prompts or retrieved context unless required for debugging under tightly controlled access paths.

5.4. Kubernetes Implementation Patterns

The architecture maps directly onto Kubernetes primitives [22] so that isolation and governance can be enforced at the platform layer in addition to application logic. The goal is not to claim Kubernetes alone provides tenant isolation, but to use it to reduce blast radius, constrain communication paths, and make misconfiguration harder.

5.4.1. Namespace Strategy

In Silo, namespace-per-tenant is used to separate workloads, secrets, service accounts, and network policy scopes. In Pool, a shared namespace is used for shared services, and tenant isolation is enforced primarily through request context and data-plane controls, with Kubernetes labels used for operational grouping rather than as the primary security boundary. In Bridge, namespaces are organized by tier, with tenant-scoped resources placed in

tenant namespaces or tier-specific namespaces, while pooled services remain shared.

5.4.2. Resource Controls

Resource controls are required to manage noisy-neighbor effects [18] and prevent a single tenant from exhausting shared compute. ResourceQuota and LimitRange define per-namespace limits for CPU, memory, and object counts. PriorityClass can ensure retrieval pods are scheduled preferentially and are less likely to be preempted during resource contention, though it influences scheduling and preemption rather than runtime resource allocation. These controls should be tier-aware in Bridge patterns and should be aligned with the evaluation methodology for tail latency and fairness.

5.4.3. Network Policies

NetworkPolicy rules [22] constrain east-west traffic and reduce cross-namespace communication by default. In Silo and Bridge, policies should restrict traffic so tenant namespaces can communicate only with shared platform services that are explicitly required. Where a service mesh is used, mutual TLS and identity-based routing can strengthen service-to-service authentication and improve observability, but it must be configured to preserve tenant context propagation and to avoid leaking sensitive headers into logs or traces. NetworkPolicy enforces isolation at layers 3 and 4. It does not validate application-layer tenant context, so tenant scoping and authorization enforcement remain required in the services and data access layers.

5.4.4. Policy Enforcement

Admission control policies can prevent unsafe configurations from entering the cluster. Open Policy Agent (OPA) Gatekeeper [23] can enforce that workloads include required labels, that privileged pods are disallowed, and that only approved network policy patterns are used. For multi-tenant RAG, a practical admission policy is to enforce the presence and correctness of tenant and tier labels on tenant-scoped resources, and to ensure that secrets and service accounts are not shared across tenant namespaces in Silo configurations. Audit logging should capture admission decisions so platform-level violations can be correlated with application-level audit trails.

Table 3: Enforcement Point Checklist

Enforcement Point	Identity Validation	Tenant Scoping	ACL Check	Provenance Validation	Logging Required
Query Gateway	Required	Required	Coarse	N/A	Trace ID emission
Retriever Service	Verify context	Filter injection	N/A	Candidate validation	Filter decisions
Storage Adapter	Verify context	Required	Required	N/A	Access decisions
Context Assembly	Verify context	Required	Required	Required	Inclusion decisions
LLM Gateway	Verify context	Required	Policy check	Chunk provenance	Full audit trail

5.5. Observability and Auditability

Observability is part of the isolation story because it enables detection and proof of enforcement. Distributed tracing should provide an end-to-end trace from the API gateway through retrieval and context assembly to the LLM gateway, with tenant identity represented as controlled metadata that is not exposed to unprivileged operators. Metrics should be emitted per tenant and per tier for latency percentiles, error rates, filter rejection rates, and queueing delays. Retrieval recall proxies, such as hit rates at top-k after filtering and reranking acceptance rates, are useful for performance diagnosis but must be scoped and access-controlled to avoid metadata inference [14]. Logs must be designed for least exposure. The system should log retrieval provenance identifiers and policy decisions rather than raw chunk text. Where prompt logging is necessary for debugging, logs should be redacted and protected with strict operational access controls. Audit logs should capture the minimal set needed to reconstruct the decision path for a request, including tenant context, policy decision identifiers, retrieved chunk identifiers, and trace IDs.

6. Evaluation Methodology

This section defines a repeatable evaluation methodology for comparing Silo, Pool, and Bridge patterns across isolation strength, performance under contention, cost-per-query, and operational overhead. The goal is to measure properties that matter to SaaS architects and that map directly to the enforcement points and failure modes defined in Sections III to V. The methodology is designed to be implementable in a Kubernetes testbed with deterministic datasets and controlled workloads so that results can be reproduced and compared across pattern variants.

6.1. Metrics Definition

Metrics are defined so they can be computed from recorded traces and logs without relying on subjective judgment. Each metric is measured for both retrieval-only and end-to-end request paths, because noisy-neighbor behavior [18] often appears in the slowest shared component, which may differ across patterns.

6.1.1. Leakage Rate (Isolation Strength)

Leakage is measured using two metrics that separate retrieval-path violations from end-to-end isolation failures. The cross-tenant candidate-return rate is the fraction of adversarial retrieval attempts where the vector search returns at least one cross-tenant candidate prior to any downstream filtering. The prompt contamination rate is the fraction of adversarial attempts where a cross-tenant chunk appears in the final context assembled for the model. The candidate-return rate captures retrieval-path violations and side-channel exposure risk, while the prompt contamination rate captures the most severe end-to-end failure. Both metrics should target zero.

Leakage is detected by validating retrieved candidate identifiers against an authoritative mapping in the metadata store. The check is performed before context assembly, and again after payload fetch, to distinguish vector-layer leakage from payload-layer authorization failures. A request is counted as leaked if any returned candidate violates tenant ownership, even if the candidate is later filtered out, because the event indicates an isolation failure in the retrieval path. Assertions should verify that required scoping constraints were applied at each enforcement point [14], for example that tenant filters were present in vector queries and that storage fetches executed with tenant-scoped authorization context.

6.1.2. Latency Under Mixed Workloads (Noisy-Neighbor Effects)

Noisy-neighbor effects [18] are measured using latency percentiles under controlled mixed-tenant load. The primary metrics are p50 and p95 latency for retrieval and for end-to-end response. Latency should be decomposed into gateway processing, vector search, reranking if enabled, context assembly, and model inference to identify which component dominates tail behavior. The workload includes multiple tenants with different load profiles to simulate contention. A high-load tenant is driven to a sustained target throughput while one or more co-tenants operate at low and moderate throughput. Workload profiles include uniform load, bursty load with short spikes, and skewed distributions where one tenant contributes the majority of requests. A noisy-neighbor index can be defined as the relative degradation of a low-load tenant's p95 latency when the high-load tenant is active, compared to the low-load tenant's p95 latency in isolation under the same request rate. This metric is computed per pattern and per tier in Bridge configurations to quantify isolation effectiveness for performance, not only for security.

6.1.3. Cost-Per-Query Decomposition

Cost-per-query is decomposed so architects can attribute cost to specific pipeline stages. The cost model includes compute cost for embedding generation on ingestion, vector search and orchestration on retrieval, and LLM inference on response generation. Storage cost includes document storage, metadata storage, and vector index footprint. The index footprint component should explicitly account for memory-resident structures when applicable [4], [5], [8], because that is often a dominant cost driver for high-performance Approximate Nearest Neighbor (ANN) indexes.

A practical cost-per-query estimate is computed as monthly infrastructure cost divided by monthly query volume, with ingestion cost either amortized by ingestion volume or reported separately. The model should include pattern-specific overhead such as duplicated control plane components in Silo, shared networking and observability overhead in Pool, and tier management overhead in Bridge. The objective is not to produce a universal cloud bill, but to produce a comparable cost decomposition across patterns under the same workload and capacity targets.

For planning purposes, end-to-end cost per query can vary by orders of magnitude, often ranging from roughly $0.01 to $1.00 depending on model selection, context window size, retrieval depth, and workload shape. In many deployments, LLM inference dominates variable cost, while vector index footprint drives fixed monthly cost. For example, storing a single 768-dimensional float32 embedding requires about 3 KB, so a 10 million-vector corpus implies on the order of tens of gigabytes before accounting for approximate nearest neighbor index overhead [4], [5], replication, and metadata.

Absolute cost magnitudes are deployment-specific and should be measured using the cost model and workload defined in this section. In many deployments, model inference dominates variable cost, while vector index footprint and memory provisioning dominate fixed cost, but the balance depends on corpus size, retrieval configuration, and token budgets.

6.1.4. Operational Overhead

Operational overhead is measured as the effort and time required to onboard and operate tenants under each pattern. Tenant onboarding time is defined as provisioning latency from an onboarding request to a tenant being able to ingest documents and serve queries with policy enforcement active. Deployment complexity can be measured by counting distinct deployments, configuration objects, and secret objects required per tenant or per tier, and by identifying which objects must be customized per tenant. Maintenance burden includes routine upgrades, scaling actions, key rotation, index rebuilds, and policy changes such as ACL updates. Automation potential is measured as the proportion of these actions that can be executed through deterministic automation without manual steps. This metric is particularly relevant for Silo and Bridge patterns where lifecycle operations scale with tenant count.

Table 2: Metric Definitions & Computation Sources

Metric	Definition	Data Source	Ground Truth Required
Leakage Rate	Fraction of adversarial queries returning cross-tenant candidates	Retrieval logs, provenance traces	Chunk-to-tenant ownership mapping
p50/p95 Latency	Latency percentiles for retrieval and end-to-end response	Distributed traces	N/A
Noisy-Neighbor Index	Relative p95 degradation under co-tenant load vs isolation	Traces with tenant attribution	Baseline measurements
Cost-Per-Query	Monthly infrastructure cost / monthly query volume	Resource metrics, billing data	Capacity targets

Onboarding Time	Provisioning latency to operational readiness	Automation logs, timestamps	N/A

6.2. Test Environment Specification

The test environment is specified so measurements are comparable across patterns. The system runs on a Kubernetes cluster [22] with fixed node allocation and fixed resource limits per service, unless the scenario explicitly tests autoscaling behavior. Node types, CPU and memory allocations, and storage classes should be held constant across patterns. Each pattern variant should be deployed using the same service implementations and configuration structure, differing only in isolation configuration such as namespace strategy, index boundaries, and policy placement.

The evaluation should use a reproducible vector database deployment, and the methodology should support alternative backends such as a relational vector store [21] or a dedicated vector database [24], [25]. Embedding model selection should favor open-source models to support deterministic benchmarking. LLM serving should be configured locally when possible to reduce variability from external API rate limits and service changes. Tenant simulation parameters include number of tenants, corpus size per tenant, chunk size distribution, and ACL complexity. These parameters must be recorded as part of the benchmark artifacts.

6.3. Workload Design

The workload dataset is constructed to control tenant separation while allowing realistic retrieval behavior. Tenant corpora should be primarily disjoint, with optional controlled overlap scenarios where similar topics appear across tenants without sharing identical documents. This allows testing whether semantic similarity can expose cross-tenant leakage when scoping is incorrect. ACL variations should be included so that authorization is not equivalent to tenant ownership. For example, within a tenant, some documents can be restricted to specific roles to validate principal-scoped enforcement during context assembly.

The query mix should include baseline queries intended to retrieve relevant chunks, burst patterns that stress queues and shared caches, and heavy-versus-small tenant mixes that expose contention. Adversarial queries [10], [11], [12] are included specifically for leakage testing and should be tagged so they can be analyzed separately from baseline traffic. All queries and expected ownership assertions should be generated deterministically from the dataset so that leakage checks are repeatable.

6.4. Evaluation Scenarios

Isolation validation scenarios execute adversarial retrieval tests for each pattern, including explicit filter bypass attempts, boundary-condition queries that target scoping weaknesses, and misrouting tests that validate fail-closed behavior when tenant context is inconsistent [14]. Expected outcomes should be stated as assertions rather than numeric results, for example that no cross-tenant candidate identifiers are returned, and that any violation triggers a policy event and request rejection.

Performance characterization scenarios measure baseline latency for each pattern under single-tenant load and then measure mixed-tenant degradation under controlled contention [18]. The methodology should include runs with and without background ingestion and indexing activity, because indexing can materially affect tail latency in pooled architectures [8]. Scalability scenarios increase tenant count and corpus size while holding per-tenant request rate constant to observe whether shared components exhibit superlinear degradation.

Cost analysis scenarios compute cost-per-query decomposition using measured resource consumption and recorded index footprint. A break-even analysis framework compares patterns as tenant count and utilization change. Sensitivity analysis varies tenant count, corpus size, query rate, and context window size, because these parameters shift which pipeline stage dominates cost and which isolation boundary becomes a bottleneck.

Table 3: Evaluation Scenario Matrix

Scenario	Workload Profile	Patterns Compared	Metrics Collected
Isolation Validation	Adversarial queries	Silo, Pool, Bridge	Leakage rate, policy violations
Baseline Performance	Single tenant, steady state	Silo, Pool, Bridge	p50/p95 latency
Noisy-Neighbor	Mixed load, high-load tenant	Pool, Bridge	Noisy-neighbor index
Scalability	Increasing tenant count	Silo, Pool, Bridge	Latency, throughput
Cost Analysis	Fixed workload, varying scale	Silo, Pool, Bridge	Cost-per-query decomposition
Break-Even	Varying tenant count	Silo vs Pool	Cost crossover point

6.5. Benchmark Harness and Repeatability

The benchmark harness consists of a traffic generator that can emulate multiple tenants, a tenant identity injection mechanism that produces authenticated requests for each tenant and principal role, and an instrumentation layer that collects traces, logs, and metrics. The harness should record the exact deployment configuration, including Kubernetes manifests, policy definitions, and datastore configuration, as versioned artifacts. Datasets should be deterministic, and evaluation prompts should be fixed so that changes in results are attributable to architecture changes rather than prompt drift.

Validation checks should run as part of every benchmark execution. These include assertions that tenant filters are present in vector queries, that storage fetches are tenant-

scoped, that context assembly performs provenance validation, and that audit logs contain required identifiers and policy decision traces. These checks ensure that a benchmark run cannot silently pass with missing enforcement [14], which is widely recommended when comparing isolation patterns where correctness depends on configuration.

This paper specifies an evaluation methodology rather than reporting empirical benchmark results. The metrics, scenarios, and benchmark harness are defined to enable reproducible comparison of isolation patterns, but execution and measurement on representative deployments remains future work. Any discussion of tradeoffs in later sections is derived from architectural analysis and documented platform behavior rather than measured performance data. Independent empirical validation using the methodology defined in this section is encouraged.

7. Discussion

This section interprets the implications of the isolation taxonomy, threat model, and reference architecture for real SaaS deployments. The intent is to translate the analysis into actionable selection guidance, highlight minimum controls required for safe pooling, and clarify where conclusions depend on assumptions or environment-specific constraints.

7.1. Pattern Selection Decision Framework

Pattern selection should be driven by explicit requirements rather than defaulting to whichever architecture is easiest to deploy. The primary criteria are regulatory requirements, tenant trust level, workload variability, cost constraints, and service-level objectives for latency and availability. Regulatory constraints typically set the minimum acceptable isolation boundary for the data plane and, in some cases, constrain where inference and telemetry can be processed. Tenant trust level matters when tenants can actively probe the system [10], [11], [12] or upload untrusted content [13], which increases the likelihood of adversarial behavior against retrieval and context assembly. Workload variability and burstiness determine whether shared infrastructure can meet tail latency targets [18] without expensive overprovisioning. Cost constraints are best evaluated using cost-per-query decomposition because the dominant driver may differ across deployments, for example index footprint versus inference cost. SLA targets, especially p95 latency and availability, determine how much performance isolation is required in practice.

A practical decision tree begins with the strictest requirement. If regulatory or contractual constraints require tenant-dedicated storage boundaries and auditable separation for retrieval artifacts, then a Silo configuration for the data plane and vector plane is typically the default, with shared control plane services allowed only if they do not access tenant content. If regulatory constraints allow shared storage with strong access control [20], then the next decision point is whether pooled retrieval can be made defensible with defense-in-depth enforcement [9], [14] and continuous validation. If the organization can operationalize strict policy enforcement, provenance validation, and automated leakage testing, then Pool can be viable for lower-risk tiers, provided performance isolation is managed through quotas and capacity controls. If the deployment requires mixed guarantees, such as a subset of tenants needing strict isolation while the majority are cost sensitive, Bridge becomes the default because it allows tiered placement without forcing one pattern for the entire tenant population.

Migration paths matter because tenant requirements change. Pool-to-Bridge migration is often driven by increasing regulatory demands or by tenants whose workloads cause sustained contention. Bridge-to-Silo migration is typically triggered by the need to reduce blast radius further or to guarantee performance isolation for a small set of high-value tenants. These migrations should be treated as first-class design requirements. The architecture should support moving a tenant between tiers with deterministic routing, explicit index rebuild procedures, and auditable verification that data-plane and vector-plane boundaries were updated correctly.

7.2. Implementation Guidance

Implementation guidance focuses on the practical controls required to make the patterns safe and operable. The emphasis is on preventing silent isolation failures and on minimizing the number of paths through the system that can bypass tenant scoping.

7.2.1. Minimum Viable Controls for Pool

Pool has the strongest dependence on correctness of identity propagation and enforcement placement [9]. The minimum viable control set should be defined as mandatory invariants. Tenant identity must be established at authentication and propagated as immutable request context. Every retrieval request must include tenant scoping constraints, and those constraints must be validated at the retriever and again during context assembly using provenance checks. Payload fetch must enforce tenant and principal authorization even if vector search was scoped, because retrieval contamination can occur when authorization logic differs between vector and payload paths. The system should fail closed [14] when tenant context is missing, inconsistent, or unverifiable.

Filter enforcement requires both prevention and detection. Prevention includes enforcing that all retrieval code paths apply scoping, including hybrid retrieval, reranking, and fallback flows. Detection includes logging and metrics that record the applied scoping predicate, returned candidate ownership, and rejection counts. Audit logging must capture enough information to prove that the system applied scoping consistently and to support investigation if a suspected leakage occurs. Configuration validation should include automated tests that intentionally attempt bypass scenarios and assert that no cross-tenant candidates can reach context assembly. These tests should run continuously in staging and as part of deployment pipelines so that drift in configuration or code does not silently weaken isolation.

7.2.2. When Bridge is the Pragmatic Choice

Bridge is typically the pragmatic choice when Pool is too risky for a subset of tenants and Silo is too expensive or too operationally heavy for the full population. In practice, Bridge often isolates the components with the highest confidentiality and blast radius risk, while pooling components where sharing is less risky and where strong enforcement is easier. A common approach is to silo the data plane and, for higher-risk tiers, silo the vector plane by using tenant-scoped indices, while keeping orchestration services and the LLM gateway shared but strictly tenant-aware. Another approach is to keep shared storage [20] but silo vector indices for tenants that require stronger retrieval isolation or more predictable performance.

Choosing what to silo versus pool should be based on the threat model [14] and the cost decomposition. If cross-tenant retrieval leakage is the dominant risk, tenant-scoped vector boundaries are often the first escalation step. If metadata inference [14] through shared observability is a primary concern, then tier-specific telemetry isolation and operational access controls may be required even if compute remains pooled. Migration strategies should prioritize reversibility and auditability. A tenant's tier change should produce an auditable sequence of actions that includes data movement or index rebuilds if required, routing updates, key and secret updates, and a post-migration validation run that executes leakage tests and baseline latency checks.

7.2.3. Operational Considerations

Operational workflows influence isolation outcomes because most real failures are triggered by misconfiguration, drift, or incomplete lifecycle procedures [9]. Tenant onboarding workflows should be deterministic and automated where possible. Onboarding should include creation of tenant registry entries, provisioning of any tenant-scoped stores or indices, application of policies and network rules [22], [23], and a validation step that confirms scoping enforcement and audit logging are active. Incident response for suspected leakage should be planned explicitly. The system should support rapid containment actions, such as temporarily disabling retrieval for a tenant, restricting pooled services to known-safe tiers, or isolating a suspect ingestion batch [13]. Audit trails must support reconstructing which chunks entered prompts for specific requests, which is widely recommended [19] for scoping the impact of a suspected event.

Key rotation, index rebuilds, and ACL updates are recurring operations that can break isolation if not designed carefully. Key rotation must be coordinated with storage access and ingestion pipelines so that encryption boundaries remain consistent. Index rebuild procedures must preserve tenant scoping and must avoid accidental mixing during backfills. ACL updates must propagate deterministically to both metadata enforcement and retrieval-time filters, and caching layers must respect the new authorization state to avoid stale access decisions.

7.3. Limitations

Several limitations bound the generality of the results. Implementations and performance characteristics vary across LLM serving stacks [17] and across vector databases [21], [24], [25], which can change the magnitude and location of noisy-neighbor effects. The evaluation workload is focused on document-based RAG [1] and may not represent other retrieval modalities such as code, images, or structured knowledge graphs. The threat model [14] is formalized around the most relevant multi-tenant RAG threats, but not every attack class is empirically validated in the evaluation methodology, and some threats are treated through architecture and controls rather than through demonstrated exploitability. Finally, deployment environments differ in network topology, observability tooling, and operational access models, which can influence metadata inference risk and the practicality of certain controls.

7.4. Future Directions

Future work can strengthen isolation and reduce operational risk in several ways. Confidential computing can provide stronger guarantees for certain processing steps by reducing exposure of plaintext data in memory under specific adversary models, though its applicability depends on deployment constraints and performance impact. Federated RAG approaches could enable selective cross-tenant knowledge sharing under explicit privacy constraints, but require rigorous policy models and robust auditing. Automated tenant placement and dynamic pattern selection are promising for Bridge deployments, where the system could assign tenants to tiers based on measured workload behavior and risk classification, then migrate tenants safely as conditions change.

Standardization is also a practical direction, particularly for defining evaluation interfaces and minimum conformance requirements for tenant isolation in RAG pipelines. A conformance testing suite that exercises leakage probes, scoping assertions, and auditability checks would reduce reliance on ad hoc validation and would make comparisons across implementations more meaningful.

8. Conclusion

Multi-tenant RAG enables enterprise SaaS platforms to ground model outputs in tenant-specific knowledge while operating on shared infrastructure. This deployment model introduces a strict requirement for tenant isolation that spans storage, embeddings, retrieval, orchestration, and inference, and it creates failure modes that are not addressed by traditional multi-tenant patterns alone. The core result of this work is a structured way to reason about isolation in RAG systems, implement it in a cloud-native stack, and evaluate tradeoffs across security, performance, cost, and operational overhead.

8.1. Summary of Contributions

This paper introduced a formal taxonomy for multi-tenant RAG isolation across four planes: data, vector, orchestration, and LLM. The taxonomy makes isolation requirements explicit, identifies where enforcement must

occur, and provides a consistent vocabulary for comparing architectural choices. A threat model tailored to multi-tenant RAG was defined, covering embedding-space vulnerabilities and data-plane risks, and grounding the discussion in concrete trust boundaries and enforcement points. A Kubernetes-native reference architecture was specified to implement tenant-aware controls through explicit policy enforcement points across ingestion and retrieval paths, with auditability treated as a first-class requirement. Finally, the paper defined an evaluation methodology that measures isolation strength through leakage testing, quantifies noisy-neighbor effects using latency percentiles under mixed-tenant load, decomposes cost-per-query into attributable components, and characterizes operational overhead through measurable lifecycle metrics.

8.2. Key Takeaways

No single isolation pattern dominates across all dimensions. Silo, Pool, and Bridge represent fundamentally different tradeoffs between blast radius, performance isolation, cost efficiency, and operational complexity. Isolation in multi-tenant RAG is an end-to-end property. It cannot be achieved by a single datastore setting or a single gateway control because retrieval and context assembly can introduce cross-tenant exposure even when storage boundaries appear correct. In pooled architectures, leakage risk is driven primarily by configuration and enforcement correctness, including identity propagation, scoping consistency across retrieval variants, provenance validation, and auditability. Pattern selection therefore requires explicit tradeoff analysis across isolation strength, latency targets, cost drivers such as index footprint and inference consumption, and the operational maturity required to run continuous validation.

8.3. Closing Statement

Multi-tenant RAG is a practical foundation for enterprise AI adoption because it aligns grounding with SaaS delivery economics, but it demands rigor in how isolation is defined, enforced, and verified. The taxonomy, threat model, reference architecture, and evaluation methodology presented here provide a defensible basis for selecting and operating Silo, Pool, and Bridge patterns in production SaaS environments. A clear next step is broader independent benchmarking and standardization of isolation and evaluation interfaces so that multi-tenant RAG systems can be compared consistently and validated continuously as platforms evolve.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[2] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense Passage Retrieval for Open-Domain Question Answering," in Proc. 2020 Conf. Empirical Methods in Natural Language Processing (EMNLP), 2020, pp. 6769–6781. doi: 10.18653/v1/2020.emnlp-main.550.

[3] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval Augmentation Reduces Hallucination in Conversation," in Findings of the Association for Computational Linguistics: EMNLP 2021, 2021, pp. 3784–3803. doi: 10.18653/v1/2021.findings-emnlp.320.

[4] 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, Apr. 2020. doi: 10.1109/TPAMI.2018.2889473.

[5] S. J. Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," in Advances in Neural Information Processing Systems (NeurIPS), vol. 32, 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html

[6] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. Simhadri, "Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters," in Proc. ACM Web Conference (WWW), 2023, pp. 3406–3416. doi: 10.1145/3543507.3583552.

[7] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data," Proc. ACM Manag. Data, vol. 2, no. 3, art. 120, pp. 1–27, 2024. doi: 10.1145/3654923.

[8] Y. Jin, Y. Wu, W. Hu, B. M. Maggs, X. Zhang, and D. Zhuo, "Curator: Efficient Indexing for Multi-Tenant Vector Databases," arXiv preprint, 2024. doi: 10.48550/arXiv.2401.07119.

[9] C. D. Weissman and S. Bobrowski, "The Design of the Force.com Multitenant Internet Application Development Platform," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2009, pp. 889–896. doi: 10.1145/1559845.1559942.

[10] M. Anderson, G. Amit, and A. Goldsteen, "Is My Data in Your Retrieval Database? Membership Inference Attacks Against Retrieval Augmented Generation," in Proc. 11th Int. Conf. Information Systems Security and Privacy (ICISSP), 2025, pp. 474–485. doi: 10.5220/0013108300003899.

[11] G. Wang, J. He, H. Li, M. Zhang, and D. Feng, "RAG-leaks: Difficulty-calibrated membership inference attacks on retrieval-augmented generation," Sci. China Inf. Sci., vol. 68, art. no. 160102, 2025. doi: 10.1007/s11432-024-4441-4.

[12] S. Zeng, J. Zhang, P. He, Y. Xing, Y. Su, T. Zhao, and W. Lu, "The Good and The Bad: Exploring Privacy Issues in Retrieval-Augmented Generation (RAG)," in Findings of the Association for Computational Linguistics: ACL 2024, 2024, pp. 4505–4524. doi: 10.18653/v1/2024.findings-acl.267.

[13] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge Corruption Attacks to Retrieval-Augmented Generation of Large Language Models," in Proc. 34th USENIX Security Symposium, 2025, pp. 3827–3844. [Online]. Available: https://arxiv.org/abs/2402.07867

[14] A. Arzanipour, R. Behnia, R. Ebrahimi, and K. Dutta, "RAG Security and Privacy: Formalizing the Threat Model and Attack Surface," arXiv preprint, 2025. doi: 10.48550/arXiv.2509.20324.

[15] J. X. Morris, V. Kuleshov, V. Shmatikov, and A. M. Rush, "Text Embeddings Reveal (Almost) As Much As Text," in Proc. 2023 Conf. Empirical Methods in Natural Language Processing (EMNLP), 2023, pp. 12448–12460. doi: 10.18653/v1/2023.emnlp-main.765.

[16] G. Wu, Z. Zhang, W. Wang, Y. Zhang, G. Chen, and M. Yang, "I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving," in Proc. Network and Distributed System Security Symposium (NDSS), 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/i-know-what-you-asked-prompt-leakage-via-kv-cache-sharing-in-multi-tenant-llm-serving/

[17] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in Proc. 29th ACM Symposium on Operating Systems Principles (SOSP), 2023, pp. 611–626. doi: 10.1145/3600006.3613165.

[18] B. Iftekhar, V. Viswanath, S. Guo, Z. Li, S. Agarwal, and A. Akella, "Ensuring Fair LLM Serving Amid Diverse Applications," arXiv preprint, 2024. doi: 10.48550/arXiv.2411.15997.

[19] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications, Version 2025," 2025. [Online]. Available: https://genai.owasp.org/llm-top-10/

[20] PostgreSQL Global Development Group, "CREATE POLICY: Define a New Row-Level Security Policy for a Table," PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs/current/sql-createpolicy.html

[21] pgvector Contributors, "pgvector: Open-source Vector Similarity Search for Postgres," GitHub repository. [Online]. Available: https://github.com/pgvector/pgvector

[22] Kubernetes, "Declare Network Policy," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/

[23] Open Policy Agent, "OPA Gatekeeper: Policy Controller for Kubernetes," Open Policy Agent Documentation. [Online]. Available: https://www.openpolicyagent.org/docs/latest/

[24] Pinecone, "Implement Multitenancy," Pinecone Documentation. [Online]. Available: https://docs.pinecone.io/guides/index-data/implement-multitenancy

[25] Milvus, "Implement Multi-tenancy," Milvus Documentation. [Online]. Available: https://milvus.io/docs/multi_tenancy.md