TIER120 PBC

Architecture Reference

Multi-Tenant RAG

A Reference Architecture for Isolating Tenant Data in Retrieval-Augmented Generation Systems

PUBLISHED BY

Tier120 PBC

Generative AI Consulting Practice

tier120pbc.com

Version 1.0 · 2026

Executive Summary

Retrieval-Augmented Generation (RAG) has become the dominant pattern for deploying generative AI in regulated environments. By grounding model responses in an organization's own documents, RAG reduces hallucination risk, improves auditability, and lets agencies move on AI use cases that would otherwise be blocked by data-handling concerns.

RAG introduces a problem that single-organization deployments rarely surface: when one platform serves multiple tenants — separate agencies, business units, or contracts — the system must guarantee that no tenant ever retrieves, references, or is influenced by another tenant's data. This is harder than it looks. Tenant context is established at authentication and must then flow, unbroken, through every retrieval call, every database write, every cache read, and every UI handler that touches a document. A single missing field in a session object can silently collapse the boundary.

This reference describes a defense-in-depth architecture for multi-tenant RAG that treats **tenant identity as a first-class field** carried by every session, enforced at every database boundary, and validated at every retrieval call. The pattern is technology-agnostic but is illustrated using a Python / FastAPI / PostgreSQL / vector-store stack typical of agency deployments. It is intended for solution architects, IV&V reviewers, and program leadership evaluating or building RAG platforms that will serve more than one organization.

> **Who this is for**
>
> Solution architects designing shared-infrastructure AI platforms; IV&V teams assessing tenant isolation claims; program managers writing PWS/SOW language for multi-tenant AI procurements; and engineering leads inheriting a single-tenant RAG prototype that now needs to support multiple customers.

1. The Multi-Tenancy Problem in RAG

1.1 Why single-tenant patterns fail

A single-tenant RAG application has a comfortably simple data flow: the user authenticates, a session is created, and every downstream query implicitly belongs to that one organization. The vector index, the document store, and the audit log all describe the same population. There is no boundary to cross because there is no second side.

Standing up a second tenant on the same infrastructure breaks this assumption everywhere at once. The session object, the retrieval handler, the document upload pipeline, the chat history table, the example-prompt library, and the administrative UI all now need to know which tenant the current user belongs to and refuse to act on anything else. In practice, the failure mode is not a single missing check — it is many checks, distributed across the codebase, that quietly default to "all data" when tenant context is absent.

1.2 The five places isolation breaks

Across multi-tenant RAG implementations, isolation failures cluster in five recurring locations:

- Session boundary — the authenticated session does not carry tenant identity, so handlers must re-derive it or guess.
- Retrieval boundary — vector and keyword searches execute against an index that contains all tenants' chunks, filtered only by an optional metadata field.
- Write boundary — document ingestion, chat history, and feedback writes accept a tenant_id from the request rather than from the trusted session.
- UI selector boundary — dropdowns, pickers, and clone-source lists are populated from unscoped queries, exposing other tenants' resources by name.
- Uniqueness-check boundary — pre-insert validations (e.g., "does a cohort with this name already exist?") run unscoped, leaking existence information across tenants.

Each of these is independently sufficient to leak data. An architecture that addresses four of five still fails an IV&V review.

2. Architectural Principles

The reference architecture rests on four principles. Each is implementable in any reasonable language and database; the discipline is in applying all four consistently.

2.1 Principle 1 — Tenant identity is a session attribute, not a request parameter

Tenant identity is established once, at authentication, from a trusted source (identity provider claims, a tenant directory lookup keyed by username, or a signed token). It is then stored on the session object alongside the user identity. No handler reads tenant_id from a query string, a form field, a cookie, or a request body. This single rule eliminates an entire class of forgery attack — a user cannot ask the system to operate on someone else's data because the system never asks the user which tenant they are in.

> **Anti-pattern**
>
> Accepting tenant_id as a hidden form field or URL parameter "because the frontend already knows it." The frontend is not a trusted source. Anything the frontend sends, an attacker can also send.

2.2 Principle 2 — Every query is tenant-scoped at the database boundary

Tenant filtering is not the responsibility of the application layer alone. It belongs in the data access layer, ideally enforced by the database itself through row-level security (RLS) policies or, at minimum, by a thin repository layer that refuses to accept queries without a tenant_id argument. The application layer may add filters; it may not remove them.

2.3 Principle 3 — Writes carry tenant identity from the session, never from the payload

Every insert and update writes the session's tenant_id into the row. The payload may not override it. If a request body contains a tenant_id field that disagrees with the session, the request is rejected — not corrected, rejected — and the attempt is logged.

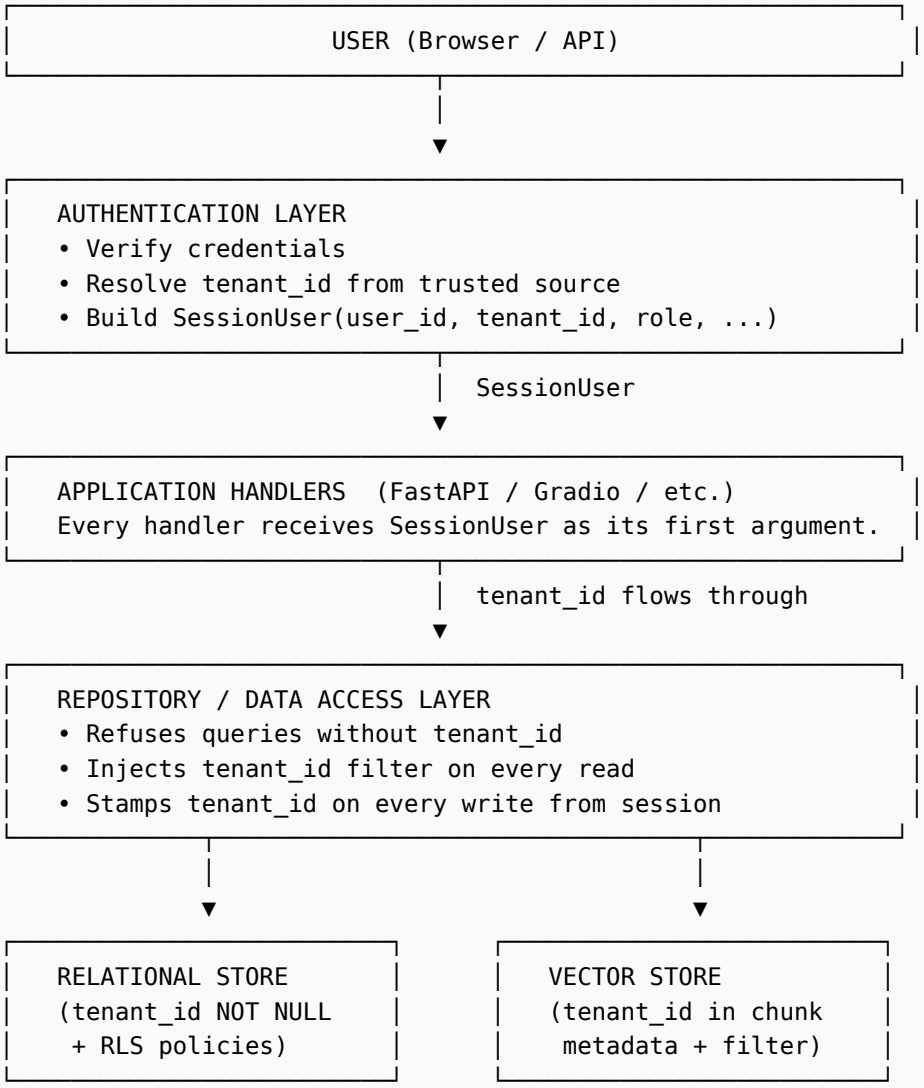
2.4 Principle 4 — Tenant isolation is verified, not assumed

Cross-tenant access attempts are explicitly tested. NOT NULL constraints on tenant_id columns are enforced at the schema level. Periodic audit queries scan for rows where tenant_id is null, mismatched, or duplicated across tenants. IV&V test plans include negative cases: "User A in Tenant 1 attempts to retrieve resource owned by Tenant 2 — system returns 404, not 403, and logs the attempt."

3. Reference Architecture

3.1 Logical view

Figure 1 — Logical layers and the path of tenant_id



3.2 The Session object

The session object is the single source of tenant identity for all downstream code. At minimum, it carries:

Field	Type	Notes
user_id	string	Unique user identifier from the

		identity provider.
tenant_id	string (UUID)	Resolved at login from a trusted source. Immutable for the session lifetime.
role	enum	Tenant-scoped role: admin, contributor, viewer.
username	string	Display only; never used for authorization.
display_name	string	Optional, for UI.

Critical: the tenant_id field is not optional. If the data model permits a null tenant_id on the session, every handler must defensively check for it, and one will eventually forget. Make it required at the type level.

4. Data Model

4.1 Tenant column on every tenant-owned table

Every table that stores tenant-owned data carries a tenant_id column. The column is NOT NULL, indexed, and references a tenants table. Tables that are genuinely global (e.g., system configuration, model registry) do not — but the default should be tenant-scoped, and exceptions should require justification.

4.2 Schema constraints

The following constraints belong at the schema level, not the application level:

- tenant_id NOT NULL on every tenant-owned table.
- Composite unique constraints include tenant_id (e.g., a "cohort name" is unique within a tenant, not globally).
- Foreign keys to tenant-owned tables are accompanied by tenant_id and validated by trigger or check constraint to ensure the parent and child belong to the same tenant.
- Row-level security policies on the relational store restrict SELECT and UPDATE to rows where tenant_id matches the current session's tenant_id.

4.3 Example schema fragment

Figure 2 — Schema fragment with RLS

```
CREATE TABLE tenants (
    tenant_id   UUID PRIMARY KEY,
    name        TEXT NOT NULL UNIQUE,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE cohorts (
    cohort_id   UUID PRIMARY KEY,
    tenant_id   UUID NOT NULL REFERENCES tenants(tenant_id),
    name        TEXT NOT NULL,
    created_by  TEXT NOT NULL,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    UNIQUE (tenant_id, name)        -- name unique WITHIN tenant
);

CREATE INDEX idx_cohorts_tenant ON cohorts(tenant_id);

ALTER TABLE cohorts ENABLE ROW LEVEL SECURITY;

CREATE POLICY cohorts_tenant_isolation ON cohorts
    USING (tenant_id = current_setting('app.tenant_id')::uuid);
```

Note: RLS depends on the application setting app.tenant_id on the database session before any query runs. This setting is established from the trusted SessionUser, not from the request payload.

4.4 Vector store considerations

Vector stores vary in their isolation primitives. The architecture supports three patterns, in descending order of strength:

Pattern	Description and trade-offs
Index-per-tenant	Strongest isolation. Each tenant has its own physical index. Eliminates cross-tenant retrieval at the storage layer. Highest operational overhead; index counts grow linearly with tenants.
Namespace-per-tenant	Most modern vector stores (Pinecone, Qdrant, Weaviate) support logical namespaces. Strong isolation with one physical index. Recommended default.
Metadata filter	Single shared index with tenant_id as a required metadata filter on every query. Cheapest, weakest. Acceptable only if the retrieval layer cannot construct a query without the filter.

5. Common Failure Patterns

The following patterns are observed repeatedly in multi-tenant RAG implementations that began life as single-tenant prototypes. They are listed in order of frequency.

5.1 Session object missing tenant_id

The most common root cause. The session dataclass or model omits tenant_id, so handlers must look it up some other way — usually from a request parameter, a global, or a cache keyed by username. Each lookup is a place where the wrong answer can be returned silently.

Fix: Add tenant_id to the session at the type level. Make it required. Update the authentication handler to populate it. Update every handler that constructs a session — in tests, in admin tools, in background workers — to provide it.

5.2 Unscoped uniqueness check

Application code asks "does a resource with this name already exist?" without filtering by tenant. The check returns true for a resource in another tenant, the operation is rejected, and the user learns that some other tenant owns a resource by that name — a small but real information leak.

Fix: Pre-insert checks must be tenant-scoped. The composite unique constraint at the schema level (Section 4.2) backstops application-level mistakes.

5.3 Unscoped UI selector

A dropdown labeled "Clone from existing cohort" populates from SELECT name FROM cohorts with no tenant filter. Users see other tenants' resource names. They cannot select them — the action handler enforces tenant scoping — but the names themselves leak.

Fix: Every list-returning query passes through the repository layer, which refuses to execute without a tenant_id argument.

5.4 Write path accepts tenant_id from payload

An ingestion endpoint accepts a JSON body that includes tenant_id, and writes that value into the row. A user in Tenant A constructs a request with tenant_id = Tenant B and writes a document into Tenant B's corpus.

Fix: Writes derive tenant_id exclusively from the session. If the payload contains tenant_id and it disagrees, reject the request with a 400 and log it.

5.5 Background jobs run without tenant context

A nightly reindex job queries all documents and rebuilds the vector store. The job has no session, so RLS does not apply, and the rebuild proceeds correctly only because the developer remembered to include tenant_id in the chunk metadata. The next person to touch the job forgets.

Fix: Background jobs iterate tenants explicitly, setting app.tenant_id on each iteration. They do not run "as superuser" against the data tables.

6. Implementation Checklist

Use this checklist when standing up a new multi-tenant RAG platform or hardening an existing single-tenant prototype.

Authentication and session

- SessionUser type includes a non-optional tenant_id field.
- tenant_id is resolved from a trusted source at login (IdP claim or directory lookup), never from request input.
- Session storage is server-side or signed; clients cannot tamper with tenant_id.

Data model

- Every tenant-owned table has a NOT NULL tenant_id column referencing the tenants table.
- Composite unique constraints include tenant_id where uniqueness is intended to be tenant-scoped.
- Indexes on tenant_id exist on all tenant-owned tables.
- Row-level security policies are enabled on the relational store.

Repository layer

- Repository functions require tenant_id as an argument; calls without it fail to compile or fail at runtime.
- Read queries inject the tenant_id filter; application code cannot remove it.
- Write operations stamp tenant_id from the session, ignoring payload values.

Vector store

- Namespace-per-tenant or index-per-tenant is used. If metadata filtering is the only option, the filter is added at the retrieval-layer level, not the call-site level.
- Chunk metadata includes tenant_id for defense in depth.

UI and API surface

- Every list endpoint, picker, and dropdown is tenant-scoped at the repository layer.
- Resource IDs are scoped or opaque; sequential or guessable IDs do not expose other tenants' resources.
- Error responses for cross-tenant access return 404, not 403, to avoid leaking resource existence.

Verification

- Automated tests include negative cross-tenant access cases.
- A periodic audit query scans for rows with null or mismatched tenant_id and alerts on findings.
- Logs capture tenant_id on every request for forensic reconstruction.

7. Considerations for IV&V Reviewers

Independent reviewers assessing a multi-tenant RAG platform should treat tenant isolation as a discrete review area with its own evidence requirements. The following questions surface most isolation defects:

- Show me the SessionUser type definition. Is tenant_id present and non-nullable?
- Show me three randomly chosen read handlers. How does each one obtain tenant_id?
- Show me the schema for three tenant-owned tables. Is tenant_id NOT NULL and indexed?
- Show me a write handler. Where does the tenant_id that gets persisted come from?
- Run a negative test: as a user in Tenant A, attempt to read a known resource ID owned by Tenant B. What is returned?
- Show me the most recent cross-tenant audit query results.
- For background jobs and admin tools, how is tenant context established?

A platform that answers all seven cleanly is in good shape. A platform that answers six but pauses on one usually has a real defect in the area it paused on.

8. Closing

Multi-tenancy is not a feature to be added late. It is a property of the data model and the session that has to be designed in from the first table and the first login handler. The pattern in this reference — tenant identity as a required session attribute, enforced at the database boundary, verified by negative testing — is not novel. It is, however, applied unevenly. Most multi-tenant RAG defects are not exotic; they are familiar patterns showing up in a new architectural context where the practitioner did not yet have the muscle memory to look for them.

Tier120 PBC works with agencies and businesses standing up generative AI platforms that have to serve more than one customer. If this reference is useful to your team, we would be glad to talk about how it applies to your specific stack.

> **About Tier120 PBC**
>
> Tier120 PBC is a Florida-based consulting firm helping government agencies and businesses modernize through generative AI, cloud, and compliance solutions. Our Generative AI practice builds and reviews production RAG platforms, with a focus on tenant isolation, auditability, and regulatory fit.

Contact: tier120pbc.com · Schedule a consultation through the website.