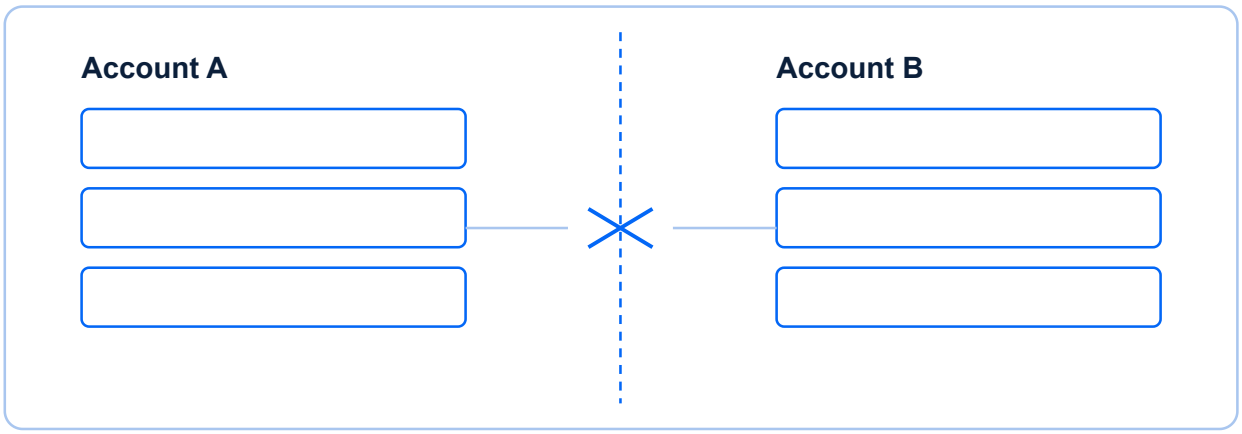



880 attempts to cross, on production. 0 rows reached.

WHITE PAPER

# Keeping Customers Apart in a Multi-Tenant AI Service

Database-enforced isolation, encryption bound to its record, and treating retrieved text as data.

Ron Alterio, Founder

November 2026

EXECUTIVE SUMMARY

# Isolation should not depend on remembering a filter

A multi-tenant AI service holds the things businesses care about most: their customers' conversations, the credentials that post in their name, and the knowledge their assistant speaks from. Most isolation rests on every query remembering its filter, and an AI adds a new way to reach the same outcome, because text it reads can try to give it instructions.

This paper presents how Xorventa makes isolation a property of the database, not a habit of the code, binds encrypted values to the record they belong to, and treats everything an assistant retrieves as data.

## 0 of 880

Attempts to read, change or delete another account's rows on the production database reached anything.

## 160 of 160

Tables with row-level security and policies, under a role that cannot bypass them.

## AES-256-GCM

Envelope encryption, with end-user content bound to its record so a moved value fails to decrypt.

## 222

Automated tests covering isolation, prompt injection and past review findings.

**Written for:** engineers and security leaders building or evaluating multi-tenant AI services.

# 1. The problem

Three properties of a multi-tenant AI service make isolation harder than it looks.

## 1.1 Filters fail silently

Application-level isolation means a `WHERE account_id = ?` on every query, written by a person, on every path. Its failure mode is the worst one available: a missed filter returns plausible results, raises no error and passes every test that only looks at one account. The more tables and features a service gains, the more places that condition has to be right.

## 1.2 Encryption at rest does not know whose value it is

Ordinary encryption protects against a stolen disk. It does nothing about a value copied to the wrong row: anything that can decrypt one record can decrypt it anywhere, so a bug that moves ciphertext can quietly hand one person's data to another.

## 1.3 An assistant reads untrusted text

A website, an uploaded document or a visitor's message can all contain text written to look like instructions. Prompt injection is ranked first in the OWASP Top 10 for LLM applications for this reason. If retrieved text and instructions reach the model in the same form, anyone who can get text into the knowledge base can try to steer the assistant.

# 2. Why the conventional approaches fall short

| Common approach | Why it falls short for this problem |
|---|---|
| **Filter every query in application code** | It is needed and not sufficient. A single mistake is silent, and nothing below it would catch the error. |
| **Row-level security connecting as the table owner** | Owners and privileged roles commonly bypass policies, so the policy can exist and never apply. The application has to run as a role that cannot. |
| **Session settings on a pooled connection** | With a transaction pooler, a session-level setting can carry over to the next request on that connection, which would put one customer's scope on another's query. |
| **One key for all encrypted data** | Key rotation becomes a rewrite of everything, and one exposed key exposes every value. |
| **Storing plaintext when encryption is unavailable** | A missing setting becomes personal data on disk, with no error to notice. |
| **Asking the model to ignore instructions in documents** | That is a request, not a boundary, and fixed delimiters can be imitated by the text they enclose. |

# 3. The solution

The design assumes each layer can be wrong, so a mistake in one does not become an exposure. Application code scopes every request, the database enforces that scope independently, and the encryption and model layers each hold a boundary of their own.

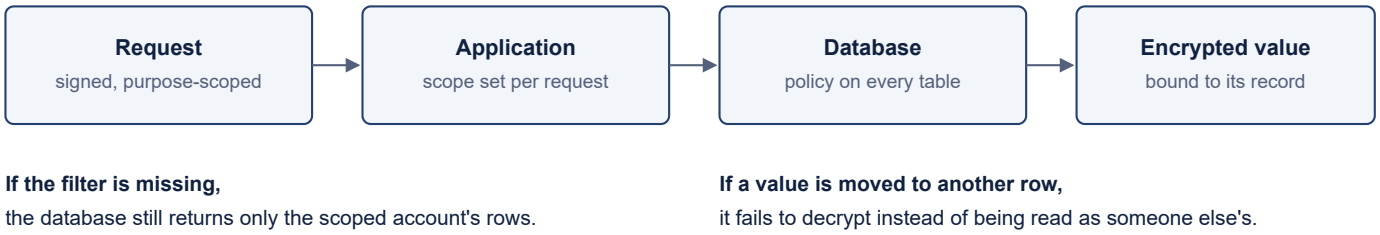



Figure 1. Independent layers. Each one holds even if the one before it has a defect.

## 3.1 Isolation enforced by the database

- **Every table has row-level security on.** All 160 tables in the application schema, with policies keyed to the account or tenant, or through a parent record where a child table has neither.
- **The application cannot bypass it.** Requests run under a dedicated role without bypass rights, so policies apply to every query the service makes.
- **Scope is set per transaction, in one statement.** Role and scope are transaction-local, and revert on commit or rollback. A pooled connection carries nothing to the next request, and combining them into one statement keeps the round trips low.
- **Scope comes from identity.** When a customer's session or a chat window's token is authenticated, its account or tenant scope is bound to the whole request, across every thread that serves it.
- **If isolation cannot be established, the request is refused.** A failure while setting up a customer request's scope stops the operation and discards the connection, so its query never runs without isolation.
- **Background work is separate.** Scheduled and operator work that must cross accounts runs in its own path, apart from customer requests.
- **New tables cannot slip through.** An automated check fails the build if a migration creates a table no policy covers.

## 3.2 Encryption with envelope keys, bound to the record

Secrets such as access tokens use envelope encryption: each value is encrypted with its own data key under AES-256-GCM, and the data key is wrapped by a master key held in a managed key service. Rotating the master key re-wraps data keys without touching the data, and a short-lived in-memory key cache keeps the live path fast without calling the key service each time.

Content belonging to a business's end users has stricter rules:

- **No plaintext fallback, in any environment.** If encryption is unavailable, the write fails. A misconfiguration causes an error, never personal data on disk.
- **The ciphertext is bound to its record.** AES-GCM authenticated data covers the purpose, account, user, field and record, length-prefixed so that a value cannot shift a boundary. A value copied to another row fails authentication instead of decrypting there.

### 3.3 Text the assistant reads is data

Retrieved knowledge and other untrusted text reach the model between delimiters containing a random value generated for that turn. Because the delimiter cannot be known in advance, text cannot be written to close it, and the instructions state that what lies within is data. When facts are extracted from sources, the output is constrained by a schema enforced during generation, so a source cannot make extraction return anything except records. Answers are also limited to the business's approved knowledge by default. No single measure stops prompt injection, so the design also limits what a manipulated answer could do.

### 3.4 Requests prove what they are for

| Surface | Control |
|---|---|
| Chat window | The public embed key only identifies the business. It is exchanged for a short-lived signed token bound to the site's origin, with a stated purpose, audience and issuer, all validated. A voice token is issued separately and cannot be used for chat. |
| Accounts | Two-factor sign-in, with its secrets stored encrypted. |
| Connected AI assistants | Access credentials expire after 15 minutes. Reading, changing settings, publishing and deleting are separate tools, each refusing the others' actions, and the owner's settings decide whether an action runs or waits for confirmation. Billing and credentials are out of reach. |
| Incoming webhooks | Signatures are checked in constant time, and an event is marked processed only after it has been handled, so a failure is retried and not lost. |
| Outbound fetches | URLs the service fetches on someone's behalf have their DNS pinned and checked, so a link cannot be pointed at internal addresses. |
| Browser | Strict transport security with preload, framing denied and a content security policy allowing no unlisted inline script. |
| Build and deploy | Dependency audits and a check for tracked secrets run automatically, and the production image build fails if a secrets file is present. |

## 4. The evidence

### 4.1 The cross-account test

Policies are only worth what they do when challenged, so we attacked them directly on the production database. For each table, in a transaction that was always rolled back, the test took the same role and settings a real request uses, scoped to one account, and tried to reach another account's rows. We then repeated it the other way round. It recorded counts only, read no content and committed nothing.

| Attempt, as the other account | Attempts | Rows reached |
|---|---|---|
| **Read another account's rows** | **282** | **0** |
| **Update another account's rows** | **282** | **0** |
| **Delete another account's rows** | **282** | **0** |
| **Read another tenant's rows, on a tenant scope** | **34** | **0** |
| **Total** | **880** | **0** |
| Read from the request role with no scope set, all 160 tables | 320 | 0 |

141 tables hold an account or tenant key directly, tested from two accounts in both directions. Child tables keyed through a parent showed each account only its own share of the rows.

> **A test that finds nothing proves little unless it could have found something. As a control, the same queries under the cross-account scope used for background work returned the other accounts' rows, and the same update matched them. Only the scope changed.**

## 4.2 Automated tests

| Area | What is checked |
|---|---|
| Row-level security | Policy coverage of every table, scope reaching the route, scope failing closed, and elevations holding only where intended. |
| Tenant isolation | Retrieval, widget endpoints and social data stay within their tenant. |
| Prompt injection | The delimiter contract and the guards around untrusted content. |
| Security reviews | Each finding from past reviews is pinned by a test, so a fix cannot quietly regress. |

222 tests in these areas pass, as part of the suite run before a release.

## 4.3 Internal review

A code review across the service's attack surfaces, with live non-destructive checks of our hosts, verified every serious finding adversarially, by trying to disprove it. No finding in that review was confirmed at Critical or High severity, and its remaining items have been closed and checked against the code.

## 4.4 Scope of these measurements

The cross-account test covers database isolation, from the role a request uses, for reads, updates and deletes. The controls in Section 3.4 are verified by the automated tests and the review. Security is continuing work, and a clean result describes the state at the time of testing, which is why these checks run repeatedly.

# 5. How this behaves in use

| If this happens | Then |
|---|---|
| A query in a customer request forgets its account filter | The database returns only the scoped account's rows. |
| A customer request's scope cannot be set up | The request is refused, and its query never runs. |
| A migration adds a table with no policy | The build fails. |
| Encrypted content is copied to another record | It fails to decrypt. |
| The encryption service is unavailable | End-user content is not written, and an error is raised. |
| A web page tells the assistant to ignore its rules | It reaches the model as delimited data, and answers stay within approved knowledge. |
| Someone copies a site's chat embed key | It only identifies the business. A page on another site cannot use it, because tokens are bound to that site's origin. |

# 6. Guidance for teams building similar systems

| If you are deciding | Our recommendation | What would change it |
|---|---|---|
| Where isolation lives | In the database as well as the code, under a role that cannot bypass it. | A separate database per customer, which moves the boundary. |
| How to pass scope | Transaction-local settings, never session-level, on a pooled connection. | No connection pooling. |
| What a failed scope does | Refuse the request and discard the connection, and keep cross-account work on its own path. | Nothing. |
| How to prove it | Attack your own policies with your real role on real tables, with a control showing the test can succeed. | Nothing. Test it, do not assume it. |
| How to encrypt user content | Envelope keys, no plaintext fallback, and authenticated data binding each value to its record. | Fields that must be searchable, which need another design. |
| How to handle untrusted text | Per-turn random delimiters, schema-constrained extraction, and limits on what an answer can do. | A model with an enforced separation of instructions and data. |

# 7. Conclusion

Isolation that depends on every developer remembering every filter will eventually fail, without anyone noticing. Enforcing it in the database, under a role that cannot bypass it, means a forgotten filter in a customer request does not become a leak. Binding ciphertext to its record and treating retrieved text as data apply the same idea at other layers: assume a mistake will happen, and decide in advance what it is allowed to cost.

The test in this paper is simple enough for any team to run against its own system, and we encourage it: use your real role and real tables, try to cross the boundary, and prove the test can succeed.

# References

The standards and guidance this work draws on.


- PostgreSQL documentation, Row Security Policies
- OWASP, Top 10 for LLM Applications
- OWASP, Multi-Tenant Security Cheat Sheet
- NIST SP 800-38D, Galois/Counter Mode
- NIST, AI Risk Management Framework
- IETF RFC 8725, JSON Web Token Best Current Practices

THE PRACTICAL CONCLUSION

# Assume a mistake will happen, and decide in advance what it is allowed to cost.

**Isolation in the database,** not only in the code

**Refuse the request** if its scope cannot be set

**Bind ciphertext** to the record it belongs to

**Attack your own policies,** with a control

Explore Xorventa at xorventa.ai