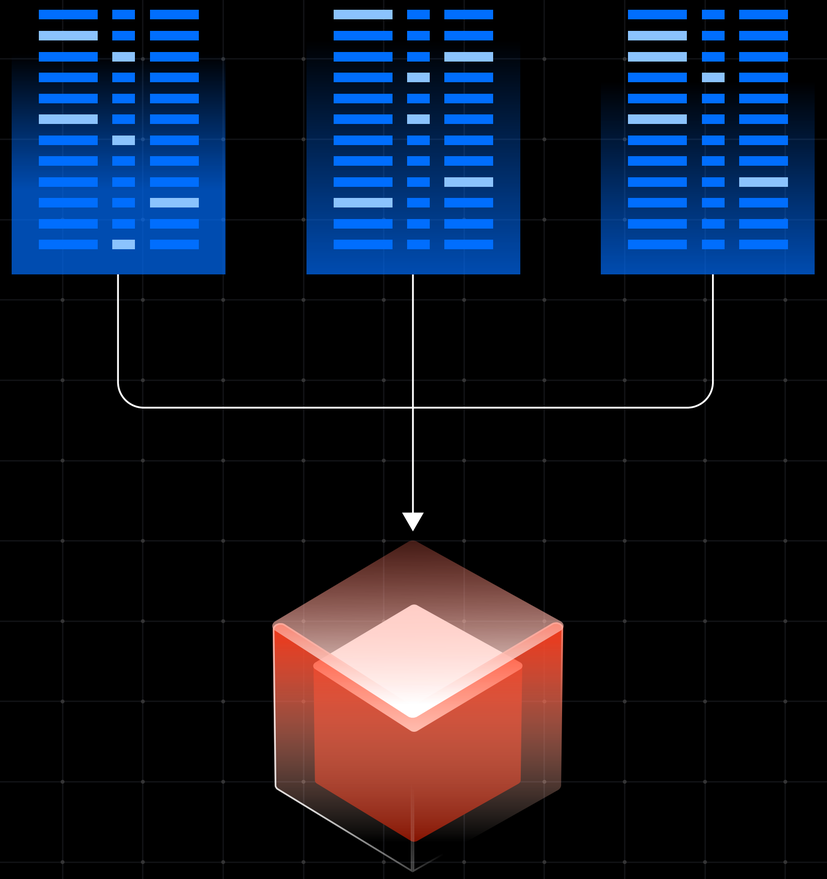
Covasant
AI-Driven. Human-Inspired.

Configurable Organization Meets the Lakehouse

How Tenant-Scoped Delta Paths Enable Single-Source-of-Truth Retrieval



Editor's note (2026-05-12). This paper was written when Auraa used **Databricks Synced Tables** (managed DLT pipelines) to mirror Delta → Lakebase. The third decision narrative below still refers to that mechanism. In May 2026, UC-323 retired Synced Tables in favor of **GovernanceWriter parallel fan-out**, the repository layer dual-writes to silver Delta, Lakebase, and bronze audit as a single unit of work. The architectural decision (selective Lakebase mirroring for retrieval performance, Delta as sole write authority) is unchanged; only the mechanism shifted from infrastructure-managed sync to application-managed fan-out. This eliminates DLT compute cost per table and removes sync latency from the read path. See UC-323 and the superseded ADR-010 for context.

Auraa is Covasant's Databricks-native, agent-driven data platform that automates ingestion, data quality, governance, and analytics across the Lakehouse through a modular suite of components orchestrated by AI agents. This paper examines the architectural synergy between three foundational decisions, how to store platform state, where to organize tenant data, and when to optimize retrieval through selective sync, and shows how they compound into a unified architecture where isolation, consistency, and discoverability emerge from structure rather than bolted-on enforcement.

Executive Summary

Multi-tenant data platforms face a fundamental tension: every tenant wants isolation guarantees, but every agent, analyst, and API consumer wants seamless retrieval. Traditional platforms resolve this tension with layers of access-control middleware, row-level security, tenant-scoped views, query rewrite interceptors, each adding latency, complexity, and surface area for misconfiguration.

Auraa takes a different approach, grounded in three foundational design choices that were made early and reinforced through production experience:

- **Metadata is data.** All platform state, user roles, tool registries, quality rules, lineage graphs, flows through the same bronze/silver medallion pattern as business data, with Delta Lake as the single write authority. There is no separate operational database; the Lakehouse holds everything. (See ADR-000.)
- **Organization is configuration.** Project structure within a tenant is not hardcoded; it is a per-tenant choice, resolved at runtime by TenantPathResolver into fully qualified Unity Catalog paths. Different tenants can use different isolation models without platform code changes. (See ADR-001.)
- **Retrieval adapts to the consumer.** Delta remains the sole write authority, but Lakebase serves as a read-optimized mirror, populated by **GovernanceWriter parallel fan-out (UC-323)**, which dual-writes selected tables to silver Delta and Lakebase as a single unit of work. (See UC-323 and the superseded ADR-010.)

These are not independent features. The synergy between them is multiplicative:

1. **Isolation by construction.** Because metadata and business data both flow through the medallion and are addressed by tenant-scoped paths, isolation is physical, there is no “wrong tenant” row to filter out, because it was never written to the same location.
2. **Write once, retrieve many ways.** A single Delta write authority feeds multiple retrieval surfaces, Spark SQL for batch analytics, Lakebase for sub-second API reads, Structured Streaming for continuous consumers, without duplicating the write path.
3. **Retrieval inherits isolation.** The Lakebase mirror inherits tenant-scoped paths from its Delta source. A resolver-scoped query against Lakebase returns the same isolated results as a query against Delta, no additional access-control layer required.

This paper examines each dimension of this synergy, the trade-offs it introduces, and how Auraa’s component architecture exploits it in production.

The Architectural Foundation

Metadata Is Data

The first decision was the most consequential. Traditional data platforms accumulate operational state, role assignments, tool registries, agent configurations, quality rules, execution logs, lineage graphs, and store it in application databases separate from the business data they manage. This creates dual infrastructure, inconsistent quality guarantees, and retrieval fragmentation.

Auraa treats all platform metadata as data. It flows through the same medallion pattern as business data:

Layer	Business Data	Platform Metadata
Bronze	Raw ingested records (append-only)	Raw state-change events: role assignments, tool registrations, config changes
Silver	Deduped, schema-validated, DQ-checked	Current-state records: active roles, current tool definitions, latest rules
Gold	Business-ready aggregates	Operational aggregates: role matrices, tool coverage reports, quality scorecards

Delta Lake is the single write authority for both payloads. No component writes operational state to an external database, a config file, or an in-memory-only store. This invariant means every piece of platform state is queryable via SQL, versionable via Delta time travel, and subject to the same quality checks as a customer record.

Organization as Configuration

The second decision addressed a problem that surfaces in every multi-tenant platform: different tenants have legitimately different needs. A three-person startup and a regulated healthcare organization should not be forced into the same isolation model.

Rather than choosing one strategy, Auraa provides three configurable strategies for how a tenant's data is organized within Unity Catalog:

Strategy	Catalog Scope	Schema Scope	Best For
Schema per Project	Single tenant catalog	project_{pid}_{layer}	Cross-project analytics, moderate isolation
Catalog per Project	Dedicated catalog per project	bronze, silver, gold	Regulatory compliance, maximum isolation
Env-Catalog, Source-Schema	Catalog per environment	Source system or business domain	ETL-centric, data mesh, multi-source

The critical design constraint: **table names are stable identities.** A table named customer remains customer regardless of which catalog, schema, or environment it lives in. The surrounding path (catalog + schema) encodes context; the table name encodes meaning. This stability is what makes retrieval portable across strategies.

Selective Sync for Retrieval Performance

The first two decisions answer how and where data is stored. The third addresses how it is retrieved when direct Delta reads are insufficient.

Not all retrieval consumers can read from Delta with acceptable performance. API-served workloads (MCP tool invocations, REST endpoints) need sub-second point queries with connection pooling. High-concurrency reads (multiple agents querying role assignments) benefit from a relational serving layer. The naive solution, writing to both Delta and a relational database, would violate the single-source-of-truth invariant established by the first decision.

The resolution: **dual-write selectively to Lakebase at the repository layer, when retrieval performance requires it.** The fan-out is:

- **One-directional in intent:** Delta is the source of truth; Lakebase is a downstream read mirror. There is no "two masters" problem because Delta still authorizes every write, Lakebase mirrors it as part of the same write transaction.
- **Selective:** Only tables with API-serving or high-concurrency retrieval patterns get the Lakebase write. Bronze tables consumed only by Spark jobs do not.
- **Application-managed:** GovernanceWriter performs the parallel fan-out in-process (silver Delta + Lakebase + bronze audit). No DLT pipeline; no sync latency.

This separates write optimization (consistency, auditability, append-only bronze) from read optimization (sub-second point queries, connection pooling, standard SQL). The write path is always the same: component → repository → GovernanceWriter → fan-out. The read path adapts to the consumer, Spark SQL for batch analytics, Lakebase for API endpoints, Structured Streaming for continuous consumers.

- Editor's note: The original third decision used **Databricks Synced Tables** (managed DLT pipelines, SNAPSHOT mode) as the sync mechanism. That worked but introduced per-table DLT compute cost and sync lag in the read path. UC-323 replaced it with application-level parallel fan-out in May 2026; the rest of this paper still uses the older terminology for historical context.

Why These Decisions Compound

Each decision is valuable on its own. The compounding happens at the intersections:

Synergy	Decisions	Effect
Isolation by construction	Metadata-as-data + configurable organization	Metadata and business data share the same tenant-scoped paths, no separate access-control layer
One pattern, two payloads	Metadata-as-data + configurable organization	Platform state and business data flow through the same medallion, resolved by the same paths
Write once, retrieve many ways	Metadata-as-data + selective sync	Single Delta write authority feeds multiple read surfaces (Spark, Lakebase, streaming)
Environment portability	Configurable organization + selective sync	Same code deploys everywhere; resolver injects environment catalog, Lakebase mirrors are environment-scoped

Isolation by Construction

In a traditional multi-tenant system, isolation is enforced by filtering, a query interceptor rewrites SELECT * FROM customers to SELECT * FROM customers WHERE tenant_id = ?. This works, but it is fragile: a missed filter, a direct table access, or a new query path can leak data across tenants.

In Auraa, isolation is enforced by addressing. When a component calls TenantPathResolver.resolve("customer", project_id="alpha", layer="silver"), the resolver returns a fully qualified path like:

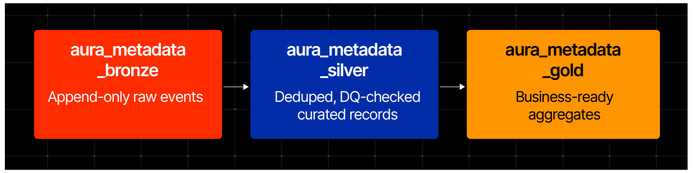
```
# Strategy 1 (Schema per Project)
aura_tenant_acme.project_alpha_silver.customer

# Strategy 2 (Catalog per Project)
aura_project_acme_alpha.silver.customer

# Strategy 3 (Env-Catalog, Source-Schema)
aura_tenant_acme_prod.crm.customer
```

The path itself encodes the tenant boundary. There is no tenant_id column to forget. There is no row-level filter to misconfigure. A component that resolves paths correctly cannot access another tenant's data, because the other tenant's data lives at a different Unity Catalog address.

This property holds for both writes and reads. When DataDuct writes ingested data, it writes to the resolved path. When an agent retrieves metadata, it queries the resolved path. The same resolver, the same path, the same isolation, applied uniformly to every operation.

One Pattern, Two Payloads

The metadata-as-data principle established that platform state follows the same medallion as business data. Configurable organization operationalized it across all three strategies. Regardless of whether a tenant uses schema-per-project, catalog-per-project, or environment-catalog, their platform metadata lives in the same resolved path namespace as their business data:



This means retrieval consumers do not need to distinguish between "platform queries" and "business queries." An agent asking "which tables have data quality failures?" queries the same Delta tables, through the same resolver, with the same isolation guarantees, as an analyst asking "what were last month's sales?" The retrieval pattern is uniform; only the payload differs.

For tool and agent registries specifically, Foundra publishes ToolSpec records to a Delta table (registry.tools) during platform initialization. At runtime, the MCP server retrieves tool definitions from this same table. The publish-then-retrieve loop is:

```
Foundra (init) --write→ Delta (registry.tools) →read→ MCP Server (runtime)
```

Because the registry lives in Delta (not a config file, not an in-memory cache), it is: - **Queryable:** agents can search for tools by capability, domain, or keyword using SQL - **Versionable:** Delta time travel provides audit trail of registry changes - **Tenant-scoped:** the resolver ensures each tenant sees only their registered tools

Write Once, Retrieve Many Ways

Delta as write authority does not mean Delta is the only retrieval surface. The Lakehouse provides multiple retrieval modes, each optimized for a different consumer:

Consumer	Retrieval Mode	Latency	Pattern
Spark SQL analytics	Direct Delta read	Seconds	spark.sql("SELECT ... FROM {resolved_path}")
Agent tool invocation	MCP → SQL Warehouse	Sub-second	Tool function queries Delta via JDBC/SQL Warehouse
REST API (Kona)	SQL Warehouse endpoint	Sub-second	FastAPI → Databricks SQL connector → Delta
Synced Tables	Lakebase mirror	Near-real-time	Automatic DLT pipeline, SNAPSHOT mode
Semantic search	Embedding index on Delta	Sub-second	Vector similarity over Delta-stored embeddings

The write path is always the same: component → DeltaWriter → Delta Lake. The read path adapts to the consumer. This asymmetry is deliberate, writes optimize for consistency and auditability (append-only bronze, merge-on-key silver), while reads optimize for the access pattern of each consumer.

Environment Portability Without Code Changes

The environment-as-catalog model (Strategy 3) means each environment has its own catalog, its own data, its own roles, and its own configuration, but the same code:

```
aura_tenant_acme_dev.crm.customer     ← dev data, dev roles, dev config
aura_tenant_acme_staging.crm.customer ← staging data, staging roles, staging config
aura_tenant_acme_prod.crm.customer    ← prod data, prod roles, prod config
```

Environments are not copies of each other. Data flows from source systems independently per environment. Configuration (connection strings, warehouse sizes, secret scopes) is environment-specific, managed by DABs variable substitution. Roles differ, a developer may have admin rights in dev but viewer access (if any) in prod.

What does promote is **code and schema definitions.** The same DABs bundle deploys to all environments. The same pipeline code, the same quality rules, the same tool definitions run everywhere. The TenantPathResolver injects the correct environment catalog at runtime, so pipeline code never contains environment-specific paths.

Because table names are stable identities, a query like SELECT * FROM crm.customer resolves to the correct environment without code changes, only the catalog prefix differs, and the resolver handles that. This means:

- **Pipeline code is environment-agnostic,** no hardcoded dev/staging/prod paths
- **Quality rules defined in dev apply in prod,** same rule, different data, different catalog
- **Agents tested in dev work in prod,** same tool definitions, resolver provides the right context

Selective sync extends this portability: each environment's Lakebase mirror is configured independently against that environment's Delta tables. Dev mirrors serve dev APIs with dev data; prod mirrors serve prod APIs with prod data. No cross-environment leakage, no shared mirrors.

Trade-offs and Constraints

Path Resolver as Critical Infrastructure

The TenantPathResolver becomes a single point of correctness. If it resolves incorrectly, writes land in the wrong location and reads return wrong results. Auraa mitigates this through:

Frozen dataclass design: The resolver is a pure function over immutable inputs (tenant_id, strategy, environment). No mutable state, no side effects, no I/O.

Exhaustive strategy testing: Each strategy is tested independently with known inputs and expected paths.

Factory-only construction: Components obtain resolvers through resolver_for_tenant(), which reads the tenant registry and validates the strategy before returning a configured instance.

Write Latency vs. Retrieval Freshness

Delta is optimized for batch and micro-batch writes, not sub-millisecond transactional inserts. For platform metadata that changes infrequently (tool registries, tenant configurations, quality rules), this is a non-issue, writes happen at initialization or on explicit user action, and reads hit a stable snapshot.

For higher-frequency metadata (execution logs, lineage events, DeltaBus messages), the append-only bronze pattern provides durability without write contention. Retrieval consumers that need near-real-time freshness can use:

- Structured Streaming over Delta changelog for continuous consumers
- Synced Tables for Lakebase-mirrored API endpoints
- SQL Warehouse with photon acceleration for low-latency point queries

Cross-Strategy Query Complexity

A platform operator who manages tenants across multiple strategies faces different path structures. Auraa's component abstraction hides this, components call the resolver, not raw SQL, but ad-hoc cross-tenant analytics (e.g., "how many tables across all tenants?") requires strategy-aware query construction. The TenantPathResolver.resolve_catalog() and resolve_schema() methods support this, but the caller must iterate over tenants and strategies.

Sync Cost and Selective Judgment

Each Synced Table incurs DLT compute for continuous mirroring. This is the primary cost of selective sync, justified only when retrieval patterns demand it. Engineers must decide which tables to sync, guided by a straightforward heuristic: tables serving API endpoints or high-concurrency reads are synced; tables consumed only by Spark batch jobs are not. Over-syncing wastes DLT compute; under-syncing forces API consumers onto slower Delta reads. The judgment is not difficult for clear cases (tool registry = sync, bronze event log = skip) but requires evaluation for tables with mixed access patterns.

Additionally, Synced Tables introduce sync latency, changes written to Delta are not immediately visible in Lakebase. For platform metadata that changes infrequently (tool registries, role assignments), the latency (seconds to low minutes) is acceptable. For state that must be read-your-own-write consistent, consumers should read from Delta directly.

Strategy Migration

Migration between organization strategies post-onboarding is non-trivial by design. Because the physical path structure differs between strategies, changing a tenant's strategy requires data movement (DEEP CLONE or CREATE TABLE AS SELECT), metadata re-registration, and grant re-application. The choice of organization strategy is a foundational decision, not a runtime toggle.

Architectural Patterns in Practice

Tool Registry: Publish to Delta, Discover at Runtime

Foundra's platform initialization publishes tool definitions to a Delta table:

```
ToolSpec (Python dataclass)
  → FoundraToolPublisher.publish()
    → INSERT INTO registry.tools (tool_id, name, module_path, ...)
      → Delta Lake (write authority)
```

At MCP runtime, the agent framework discovers available tools by querying the same table:

```
MCP Server (runtime)
  → SELECT * FROM registry.tools WHERE tenant_id = ? AND active = true
    → Delta Lake (read)
      → Hydrate ToolSpec → Register with MCP handler
```

The tool registry is tenant-scoped by construction: registry.tools lives within the tenant's resolved metadata path. A tenant using Strategy 1 sees tools at aura_tenant_acme.aura_metadata_silver.tools; a tenant using Strategy 2 sees them at aura_project_acme_platform.silver.tools. Same table name, same schema, different physical location, isolation without filtering.

Metadata Extraction: Write Structured, Retrieve Semantically

AION discovers metadata from source systems (tables, columns, constraints, comments) and writes structured records to Delta:

```
Source System (BigQuery, PostgreSQL, etc.)
  → AION Connector.discover()
    → DatasetRecord, ColumnRecord, RelationshipRecord
      → Delta Lake (aura_metadata_bronze)
        → DQ rules → Delta Lake (aura_metadata_silver)
```

Downstream consumers retrieve this metadata in different modes:

Structured retrieval: SELECT column_name, column_comment FROM metadata.columns WHERE table_name = ?

Semantic retrieval: NEXARA embeds column descriptions into vectors stored in Delta, enabling similarity search ("find columns related to customer address")

Agent retrieval: AIVARA queries metadata tables to build context for troubleshooting or analysis tasks

All three retrieval modes read from the same Delta tables, through the same tenant-scoped paths. The write happened once; retrieval adapts to the consumer.

Role-Based Auth: Delta Writes, Lakebase Reads

User role assignments illustrate all three design decisions working together. When an administrator assigns a user to a project role:

```
Admin action (via MCP tool or REST API)
  → Write role event to Delta (governance.tenant_principal_roles,  bronze)
    → DQ validation + dedup → Delta (governance.tenant_principal_roles,
silver)
      → Synced Table → Lakebase mirror
```

On every subsequent API request, the MCP auth middleware must resolve the caller's tenant and role. This is a high-concurrency, sub-second lookup, exactly the pattern selective sync targets. The middleware queries the Lakebase mirror of governance.tenant_principal_roles, not Delta directly:

```
MCP request arrives
  → MCPAuthMiddleware reads user_email
    → SELECT tenant_id, role FROM governance.tenant_principal_roles
      WHERE principal_email = ? (via Lakebase,  sub-second,
connection-pooled)
      → Resolve tenant context → proceed with request
```

The write path (Delta) preserves the audit trail: every role assignment, revocation, and change is an append-only bronze event. The read path (Lakebase) optimizes for the auth middleware's access pattern: point lookups by email, hundreds of times per minute, with connection pooling. The configurable organization ensures the role table lives at the tenant's resolved path, so the Lakebase mirror is tenant-scoped by construction.

Data Quality: Rules and Results in the Same Lakehouse

AICURA stores data quality rules and their execution results in Delta:

```
Rule definition (user or agent authored)
  → Delta (aura_metadata.quality_rules)

Rule execution (scheduled or on-demand)
  → Read rule from Delta
  → Execute against resolved table path
  → Write result to Delta (aura_metadata.quality_results)
```

Because rules reference tables by name (not by full path), they are portable across environments. A rule defined as "column email in table customer must match RFC 5322" works in dev, staging, and prod, the resolver provides the correct full path at execution time.

Comparison: With and Without These Decisions

Dimension	Traditional Approach	With Auraa's Unified Architecture	Primary Principle
Data vs. metadata storage	Separate stores (app DB for metadata, lake for data)	Single store (Delta) for both, same medallion	Metadata is data
Quality guarantees	Business data has DQ; metadata does not	Both payloads flow through same DQ rules	Metadata is data
Tenant isolation	Row-level security filters on every query	Physical path separation by construction	Configurable organization
New tenant onboarding	Same structure for all tenants	Strategy chosen at onboarding, optimized for use case	Configurable organization
Environment portability	Environment-specific code paths or config rewrites	Same code everywhere; resolver injects environment catalog at runtime	Configurable organization
API retrieval latency	Dedicated relational DB (second write target)	Lakebase mirror via Synced Tables (one-directional)	Selective sync
Agent context building	Query multiple systems, merge results	Single Delta surface, Lakebase for hot paths	Metadata is data + selective sync
Audit trail	Application-level logging	Delta time travel for all state (data + metadata)	Metadata is data

Conclusion

The synergy between these three decisions is not incidental, it is architectural.

The first decision, metadata is data, established a single storage discipline. Every piece of platform state flows through the same medallion pattern as business data, with Delta as the sole write authority. This eliminated dual infrastructure, gave metadata the same quality guarantees as business records, and made every operational fact queryable via SQL and versionable via time travel.

The second decision, organization as configuration, gave each tenant a path structure suited to their requirements, resolved at runtime by TenantPathResolver. This made isolation a structural property of the addressing scheme rather than a filtering overlay. A component that resolves paths correctly cannot access another tenant's data, because the other tenant's data lives at a different Unity Catalog address.

The third decision, selective Lakebase mirroring for retrieval, separated write optimization from read optimization. Delta remains the sole write authority, preserving consistency and auditability. The mirror is populated by GovernanceWriter parallel fan-out (UC-323), which writes selected tables to silver Delta and Lakebase in the same unit of work, providing sub-second API reads and connection pooling for the workloads that need them. The Lakebase mirror inherits tenant-scoped paths from its Delta source, no additional access-control layer required. (The mechanism replaced Databricks Synced Tables in May 2026; the decision is unchanged.)

Each decision is independently valuable. Treating metadata as data alone eliminates the dual-infrastructure problem. Configurable organization alone eliminates row-level tenant filtering. Selective sync alone eliminates the "two masters" problem for API-served workloads. But the compounding effect, metadata and business data, organized by the same paths, written to the same store, retrieved through the same optimized surface, creates a system where isolation, consistency, and discoverability are structural properties, not bolted-on enforcement.

The TenantPathResolver bridges the first two decisions: it does not distinguish between resolving a path for the customer business table and the tools registry table. Selective sync extends the chain: the Lakebase mirror inherits the same tenant-scoped isolation from the Delta source. No separate access-control middleware at any layer.

For platform teams evaluating multi-tenant Lakehouse architectures, the lesson is this: **decide how to store early, decide where to store flexibly, and decide how to retrieve pragmatically.** The result is a system that is simpler to reason about, harder to misconfigure, and more flexible for tenants with different requirements.

The path is the policy. Delta is the authority. The Lakehouse is the surface.

References

- Metadata Is Data, Platform State Through the Medallion Pattern (ADR-000: docs/architecture/ADR-000-metadata-is-data.md)
- Configurable Project Organization Strategy (ADR-001: docs/architecture/ADR-001-configurable-project-organization.md)
- Delta-to-Lakebase Sync for Retrieval Performance (ADR-010: docs/architecture/ADR-010-delta-to-lakebase-sync.md)
- TenantPathResolver Design (docs/architecture/tenant_path_resolver_design.md)
- DeltaBus: A Lakehouse-Native Event Bus Pattern (docs/whitepapers/WP-004 DeltaBus-Lakehouse-Native-Event-Bus-Whitepaper.md)
- Auraa: An Agentic Data Activation Platform (docs/whitepapers/WP-002 Auraa-Platform-Whitepaper.md)
- Databricks Unity Catalog Documentation, Catalog, Schema, and Table Isolation Models
- Delta Lake Documentation, Time Travel, DEEP CLONE, Structured Streaming
- Databricks Synced Tables Documentation, Delta-to-Lakebase Mirroring

Covasant

AI-Driven. Human-Inspired.

About Covasant

Covasant Technologies is an emerging global leader in delivering Agentic AI-led services that address complex, industry-specific challenges. Through its pioneering Services-as-Software model, Covasant brings together capabilities in data engineering, digital and cloud, AI engineering, and Enterprise Risk Management (ERM). Strategically located in innovation hubs including Hyderabad, Plano, New York, Los Angeles, London, and Dubai, Covasant leverages a premier talent ecosystem to deliver scalable, autonomous solutions. Our "Governance-Led" approach ensures that global enterprises can automate decision-making and orchestrate business processes with the reliability and speed required in today's market.

Plano, TX (USA) • London, UK • Hyderabad, India • Dubai, UAE

kathy.harnois@Covasant.com

alan.dennis@Covasant.com