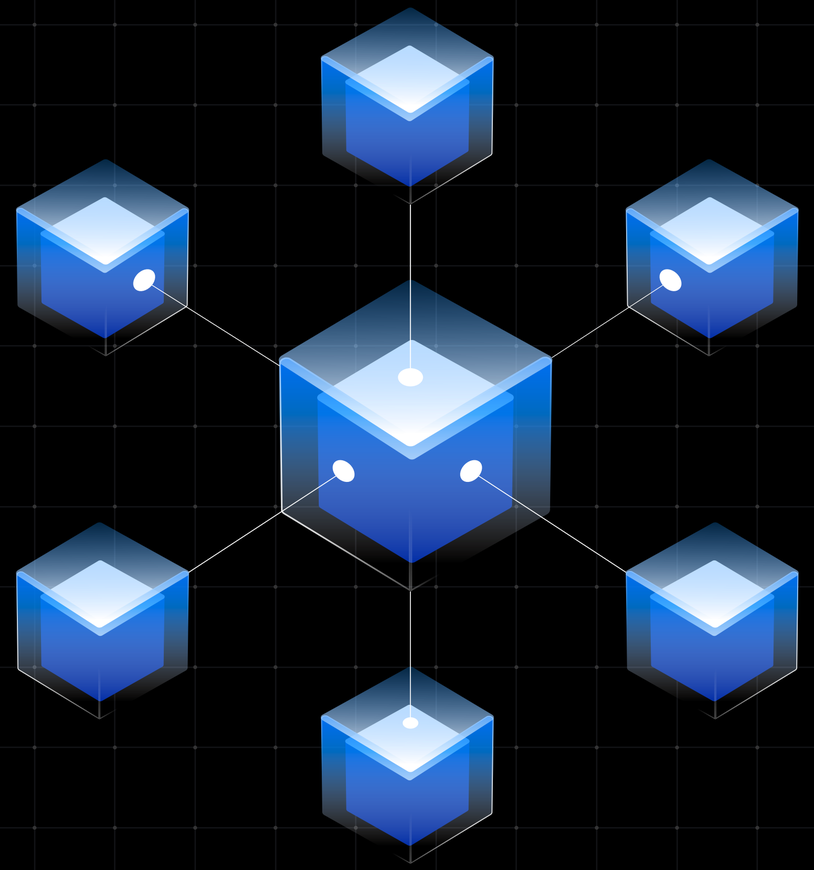
Covasant
AI-Driven. Human-Inspired.

Sub-Second on the Lakehouse

How Auraa Serves APIs from Databricks Without Breaking the Single Source of Truth

Alan Dennis

Databricks MVP,
VP - AI and Data Innovation,
Covasant



Executive Summary

Auraa is an agentic data platform built natively on Databricks. Its architecture rests on a foundational principle, captured in our zeroth Architecture Decision Record (ADR-000) and titled **Metadata Is Data**, which holds that all platform metadata and configuration (user roles, tool registries, agent definitions, data quality rules, project settings, execution logs, lineage) flows through the same Delta-backed bronze → silver → gold medallion as business data, governed by Unity Catalog, isolated by tenant path, with Delta Lake as the single write authority for both. Treating metadata as a first-class form of data is what gives Auraa one governance perimeter, one audit trail, and one quality regime across the platform.

Taking that principle seriously introduces a latency problem. Databricks Apps allows us to host APIs inside the same governance perimeter as the data they serve, but Delta accessed through a SQL Warehouse is not a sub-second point-read engine. The alternative of placing a separate operational database alongside the lakehouse would create a second write authority and fracture the very governance model that the principle was designed to protect.

This paper documents the architecture Auraa converged on to resolve that tension: a **dual-surface read model** in which Delta Lake remains the single write authority and Databricks Lakebase ,described in the official documentation as “a fully managed Postgres database integrated into the Databricks platform” ,operates as a downstream read-only co-surface populated by application-level fan-out at write time. We describe the GovernanceWriter pattern that performs the fan-out, explain why we retired the most obvious alternative (Databricks Synced Tables), enumerate the trade-offs we accepted, and characterize the consequences for API workloads running on Databricks Apps. The intended audience is platform architects, data engineers, and technical leaders evaluating how to serve low-latency APIs from a lakehouse-native platform without compromising on the single source of truth.

The Principle
Metadata is Data

Auraa's architecture is anchored by a single load-bearing claim, captured in ADR-000. Before describing the latency problem or the serving architecture that solves it, it is worth setting the principle out clearly, because every subsequent design decision in this paper is constrained by it. This section first states what the principle says, then explains why it matters, then surfaces the architectural bill that taking the principle seriously eventually leaves on the table.

What ADR-000 says

Data platforms accumulate operational state that is distinct from the business data they manage: user role assignments, tool registries, agent configurations, data quality rules, execution logs, lineage graphs, project settings. Traditional platforms store this state in application databases, configuration files, or in-memory caches that sit outside the lakehouse and are governed by a different set of mechanisms.

ADR-000 rejects that separation. Its core claim is the following:

All platform metadata is data. It flows through the medallion pattern (bronze → silver → gold) with the same rigor, isolation, and tooling as business data. Delta Lake is the single write authority for both business data and platform state.

This is not a guideline or a recommendation. It is an architectural invariant. Components that need to persist state do so through Delta. No component writes operational state to a side database, a local configuration file, or an in-memory-only store.

Why the principle matters

Treating metadata and configuration as data collapses three problems that recur in conventional data platforms. Each of the three is a problem we encountered in earlier systems, and each of them disappears almost entirely once metadata flows through the same medallion as business data.

The first problem is **dual infrastructure.** Without ADR-000, the platform must operate and secure two storage systems: the lakehouse for business data and a relational database for platform state. Each has its own backup posture, failover plan, access-control surface, and monitoring stack. With ADR-000, there is one write system, one access-control model rooted in Unity Catalog, one backup story, and one audit trail.

The second problem is **inconsistent quality guarantees.** Business data routinely flows through schema validation, deduplication, and null-detection checks; platform metadata historically does not. A corrupt role assignment or a malformed tool definition can therefore silently degrade the system because it was never subjected to the same rigor as a customer record. ADR-000 closes that gap: a role assignment with an invalid tenant identifier fails data-quality validation before it becomes operational truth, exactly as a customer record with a malformed email would.

The third problem is **retrieval fragmentation.** When metadata lives in a separate store from business data, an agent assembling context for a task must query multiple backends, each with a different connection, query language, and access-control model. Under ADR-000, that agent queries tool definitions, quality results, and business data through the same SQL surface with the same tenant-scoped path resolution. The data model is uniform regardless of which kind of state is being read.

Beyond these three, ADR-000 has a deliberate fourth benefit that is easy to overlook: it eliminates the need to design, build, and maintain separate audit tables. Because every governance write lands in Delta, the medallion already records the full history of every state change. Bronze captures the immutable event stream, and Delta's native versioning enables time-travel queries against silver and gold. A question such as "what tools were registered at two o'clock yesterday afternoon?" becomes a SELECT … AS OF TIMESTAMP query against the registry table rather than a search through application logs, and no auxiliary audit schema must be designed, populated, or kept in sync with the operational tables. Environment promotion and lineage-as-data fall out of the same property: the audit trail and the operational state are the same object, viewed at different points in time.

The bill that comes due

The benefits described above are not free. The bill that ADR-000 leaves on the table is API latency. Delta Lake is optimized for batch and micro-batch writes and for analytical reads through Spark; it is not optimized for sub-second point reads from a connection-pooled API. Every component in Auraa that must serve state through an HTTP request ,authentication middleware, MCP tool servers, project-scoped REST endpoints ,sits on the wrong side of that optimization.

This paper is the story of how Auraa paid that bill without abandoning the principle. The principle is the constant; everything that follows is the engineering required to honor it.

The Problem
Serving APIs from Databricks Apps

Having established the principle that constrains us, we turn to the specific operational context in which the latency problem appears. Auraa hosts a number of its APIs on Databricks Apps, the platform capability that lets us deploy services inside the same governance perimeter as the data they serve. This section describes what Databricks Apps gives us, what it does not give us, and why the most obvious workaround would undo the very principle of previous section.

What Databricks Apps does not provide

What Databricks Apps does not provide is a sub-second point-read engine for arbitrary state stored in Delta. The default read path for a Delta table is through a SQL Warehouse, which incurs JVM startup, query planning, and per-query session overhead. Cold-start latency for a warehouse query is measured in seconds rather than milliseconds. For API workloads, that latency is not a budget that can be spent on incidental lookups.

Auraa's request hot path is full of reads that must be fast in order for the platform to feel interactive:

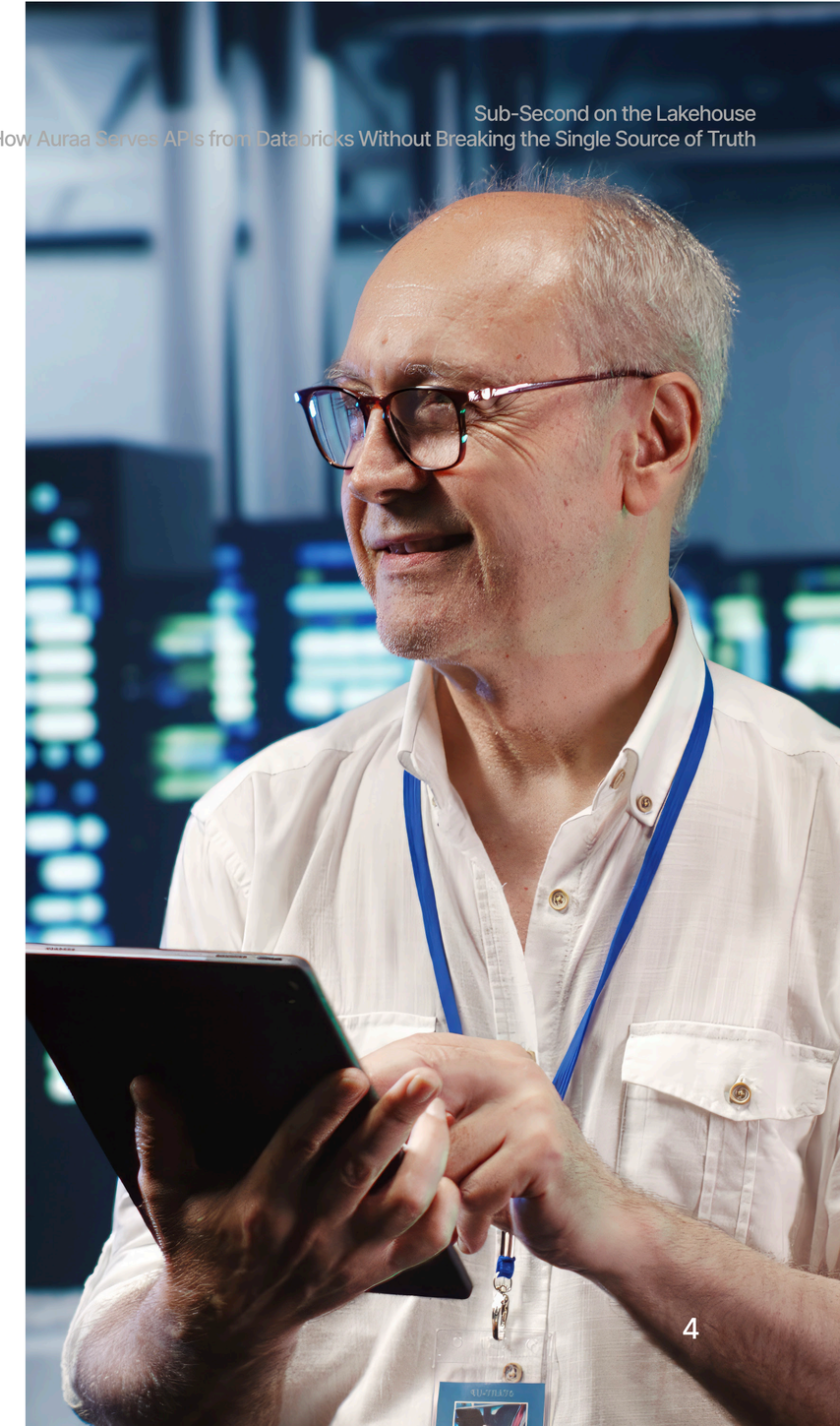
- **Authentication checks** on every incoming request, which must determine whether a given principal holds a given role on a given tenant.
- **Tool registry lookups** that resolve the set of tools an MCP server exposes and the current definition of each.
- **Project settings** that determine which warehouse, catalog, and retention policy apply to the request being served.
- **Agent definitions** that supply the skills, prompts, and tool bindings associated with the agent handling the call.

Each of these reads touches between two and ten governance tables. If any of those reads must wait for a SQL Warehouse to cold-start, the API call extends by roughly two to six seconds on a Serverless SQL warehouse ,the typical startup window Databricks publishes for that warehouse type ,and by approximately four minutes on a Pro or Classic warehouse. For a web application rendering interactive views, that delay is the difference between an experience that feels responsive and one that requires patience from the user. For an MCP server fielding agent tool calls, the cost is real but more easily absorbed, because the latency hides within the larger response time that language models already require. Either way, the cumulative effect across many requests measurably degrades the perceived quality of the platform.

Why the obvious workaround is not a workaround

The most obvious response to the problem above is to provision a relational database alongside the lakehouse and use it to serve the hot path. Mechanically this works. Architecturally it is a regression. The moment a second store is introduced as a write target, there are two write authorities. When a role is granted, which one holds the truth? Schema drift between the two stores becomes inevitable. Audit gaps open up wherever the two are not perfectly synchronized. Unity Catalog's governance reaches one of them but not the other.

The entire purpose of treating metadata as data is to avoid having a second master. Adding a Postgres next to the lakehouse, then, is not a solution to the latency problem. It is the latency problem traded for the very governance fragmentation that ADR-000 was designed to prevent. A real solution must deliver sub-second reads while keeping Delta as the single write authority. That constraint is what shaped everything in the remainder of this paper.



What We Tried First
Databricks Synced Tables

Before arriving at the architecture this paper documents, Auraa pursued a different approach that, on paper, appeared to satisfy both constraints. Databricks itself offers a feature, Synced Tables, designed precisely for the scenario of mirroring Delta state into a low-latency relational surface. This section describes that feature, explains why we initially adopted it, and details the operational realities that led us to retire it.

What Databricks Apps does not provide

Synced Tables is a managed Databricks feature that uses Lakeflow Spark Declarative Pipelines under the hood to mirror a Delta table into Lakebase, the platform's managed Postgres serving layer. Readers familiar with earlier Databricks releases may recognize this framework by its prior name, Delta Live Tables. The sync is unidirectional: data flows from Delta into Lakebase and never the other way around. Consumers read from Lakebase using standard Postgres drivers and connection pools; nothing writes to Lakebase except the sync pipeline.

On the surface, this arrangement appeared to satisfy ADR-000 perfectly. Delta would remain the single write authority. Lakebase would simply be a read-optimized projection of that authority. Auraa adopted Synced Tables, provisioned definitions for approximately thirty-five governance tables, and built the initial generation of API endpoints against the Lakebase read path.

Where Synced Tables fell short in practice

What looked elegant in design proved expensive and brittle in operation. The shortcomings clustered into several categories that, individually, might have been tolerable but together made the approach untenable.

The first category was **cost.** Continuous mode operates a streaming Lakeflow pipeline twenty-four hours a day on a per-table basis, with a minimum sync interval measured in seconds. With more than thirty-five governance tables under management, the platform was running more than thirty-five always-on pipelines, and the compute cost remained essentially flat regardless of whether any data was actually changing. The economics did not match the workload.

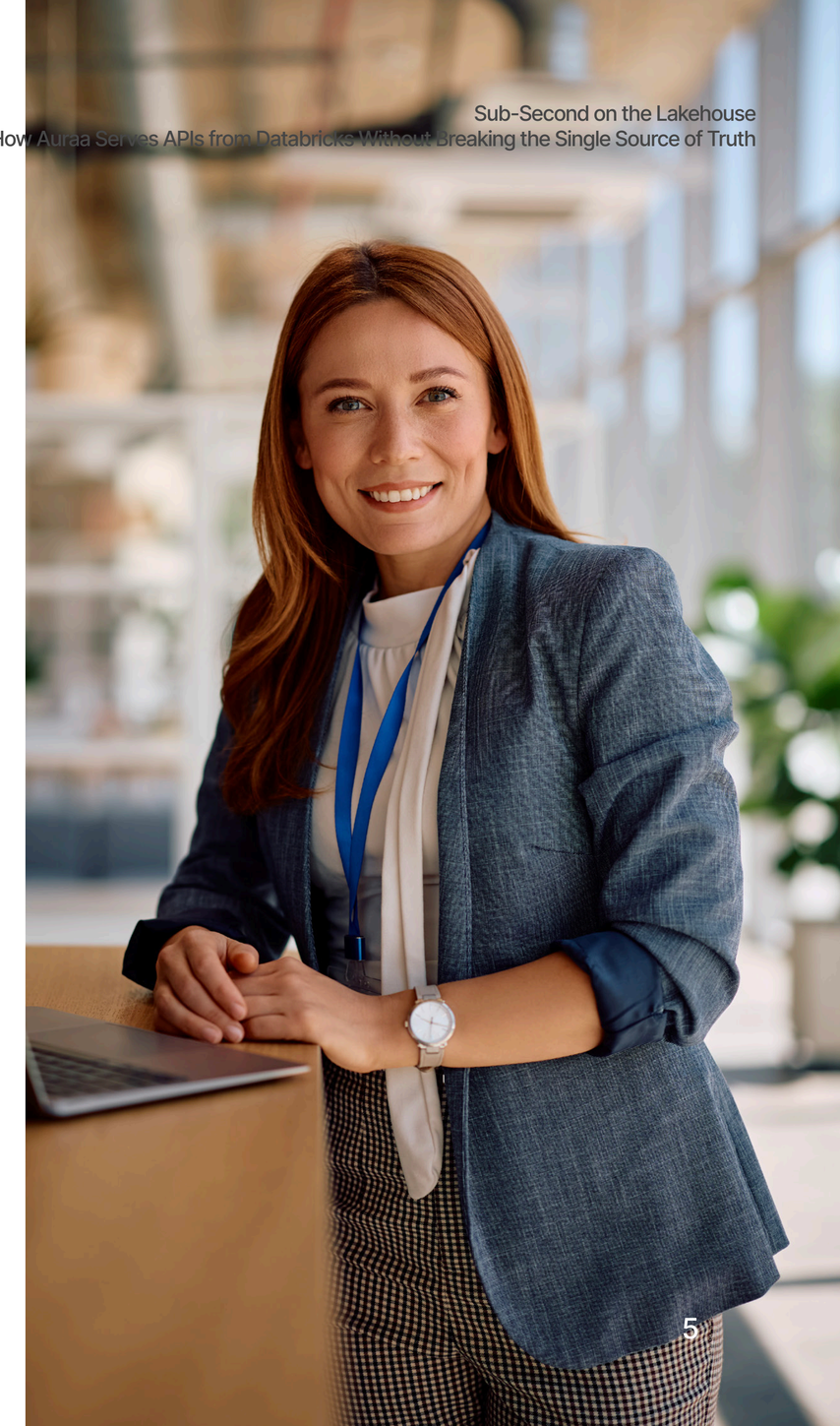
The second category was **freshness and trigger semantics.** The alternative modes, Snapshot and Triggered, sound event-driven but are not. The Databricks documentation describes Triggered mode as "scheduled updates that run on demand or at intervals" and states that for both Snapshot and Triggered modes, "subsequent syncs must be triggered explicitly." Between those explicit triggers, Lakebase is allowed to drift behind Delta. The naming of "triggered" suggested an event-bus model that the implementation did not actually deliver.

The third category was **read-your-own-write.** API callers frequently write a record and then immediately read it back as part of the same logical operation. Synced Tables introduces a propagation delay between Delta and Lakebase that breaks this expectation. The workarounds available, explicit waits, or a fallback to Delta on the next read,were both visibly ugly and tended to spread complexity throughout the calling code.

The fourth category was **extensibility.** The sync was a blind mirror. There was no point at which the platform could validate the propagated record, deduplicate against in-flight writes, or enrich the Lakebase row with derived columns. Any transformation logic had to live somewhere else in the system, which complicated reasoning about what the served data actually represented.

The verdict

Taken together, the infrastructure was doing too much in some dimensions (running a continuous pipeline per table even when no data changed) and not enough in others (offering no read-your-own-write guarantee and no transformation hook). We were paying for both ends of that asymmetry, and we were not getting a system whose behavior matched what API workloads actually needed. The decision to retire Synced Tables followed directly. The architecture that replaced it is the subject of next section.



What Worked
Application-Layer Fan-Out

The architecture that succeeded did so because it relocated propagation from infrastructure to application code. The repository layer already holds the data at the moment of every governance write; asking a downstream pipeline to re-derive that data is, on reflection, wasted motion. This section describes the resulting pattern, names the component that implements it, and explains how the approach preserves ADR-000.

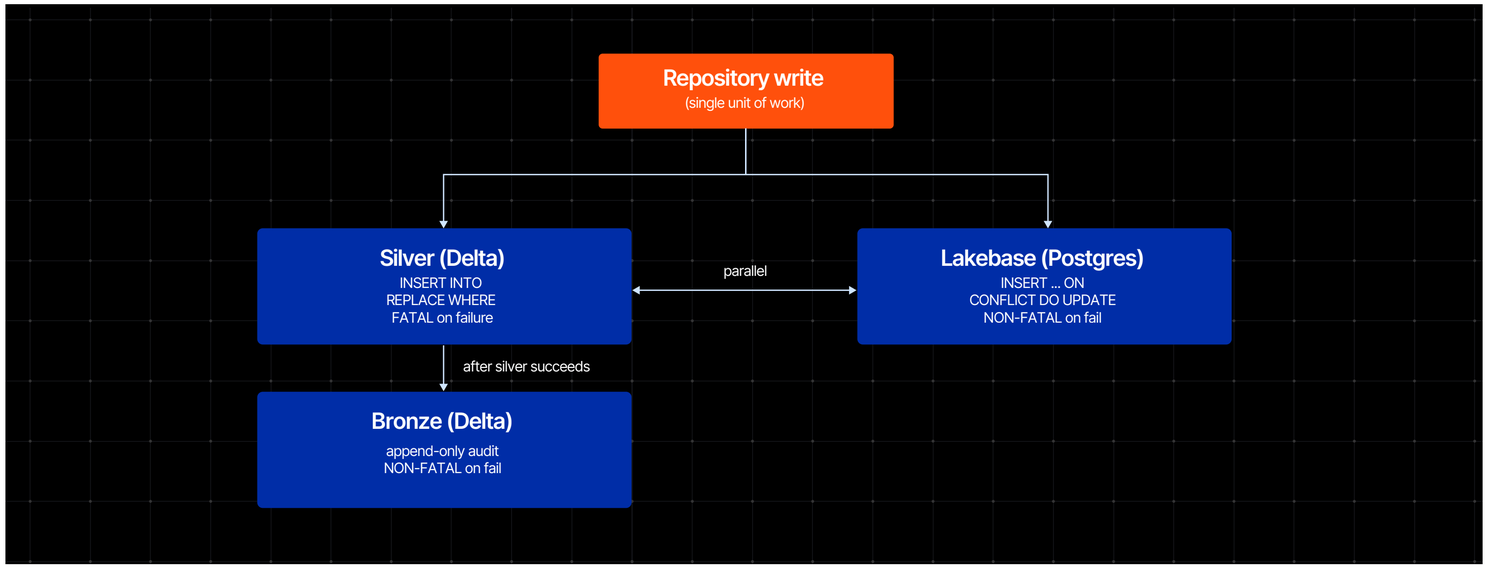
The insight that reshaped the design

Every mutation to a governance table in Auraa flows through a small number of repository classes. At the moment of write, the repository possesses the canonical representation of the new row: it knows the primary key, the column values, the tenant context, and the originating request identity. The classical justification for using an infrastructure-level sync, that the application does not know when the data has changed, does not hold here, because the application is the cause of the change.

Once this is recognized, the question reframes itself. Rather than writing to Delta and then asking a Databricks-managed pipeline to mirror it into Lakebase, the repository can write to both destinations directly, in parallel, as a single unit of work. The propagation delay collapses to whatever latency the parallel write adds, which is measured in milliseconds rather than seconds.

The GovernanceWriter pattern

The component that realizes this insight is named GovernanceWriter. It is a single class in aura_core that all governance repositories invoke when persisting a write. Its behavior is illustrated in the diagram below.



Each of the three writes carries an explicit role within the unit of work. The silver write is the contract that the caller depends on. The Lakebase write is the serving surface that the API tier reads from. The bronze write is the immutable audit trail that lets the platform reconstruct what should be in Lakebase at any point in the past.

The silver write executes against Delta using INSERT INTO REPLACE WHERE keyed on the table's primary key. This write is fatal: if it fails, the caller receives an error and no state is considered persisted. The Lakebase write executes against Postgres using INSERT … ON CONFLICT DO UPDATE on the same primary key, runs in parallel with the silver write via a shared thread pool, and is treated as non-fatal. If Lakebase fails for any reason, silver remains authoritative and a background self-heal pass reconciles Lakebase from silver. The bronze write fires only after the silver write has succeeded, appends an event record to the audit log, and is similarly non-fatal so that audit-side failures never block the caller.

How the pattern preserves ADR-000

A reasonable reader might ask whether fanning out to a second store violates the ADR-000 invariant. It does not, and the reasoning matters.

ADR-000 requires that Delta Lake be the single write authority. Under the GovernanceWriter pattern, the only component in the platform that writes to Lakebase is GovernanceWriter itself, and GovernanceWriter only writes to Lakebase as a downstream consequence of having first committed the same record to silver Delta. Lakebase is therefore logically downstream of silver: the same record, served through a different physical surface. When Lakebase falls behind, silver is the source of truth, and the self-heal pass closes the gap. Lakebase never originates state.

The mechanism by which Delta state propagates to the serving surface shifted from infrastructure-managed (Synced Tables) to application-managed (GovernanceWriter), but the architectural invariant, Delta is the contract, the lakehouse is the authority, did not shift. Consumers of API endpoints continue to read state that, in principle, has Delta as its origin and Unity Catalog as its governance perimeter.

Read-your-own-write, restored

Because the Lakebase write occurs during the originating request rather than via a downstream pipeline, the next read from the same caller observes its own write. There is no sync lag in the read path that the caller perceives. This property is what made API workloads viable on the resulting architecture; Synced Tables, by construction, could not provide it.

The combined effect of these design choices is an architecture in which a single application-layer component performs propagation that is both cheaper than a managed sync pipeline and more responsive to the access patterns of API callers. Before turning to the read side of that architecture, however, one question deserves a direct answer.

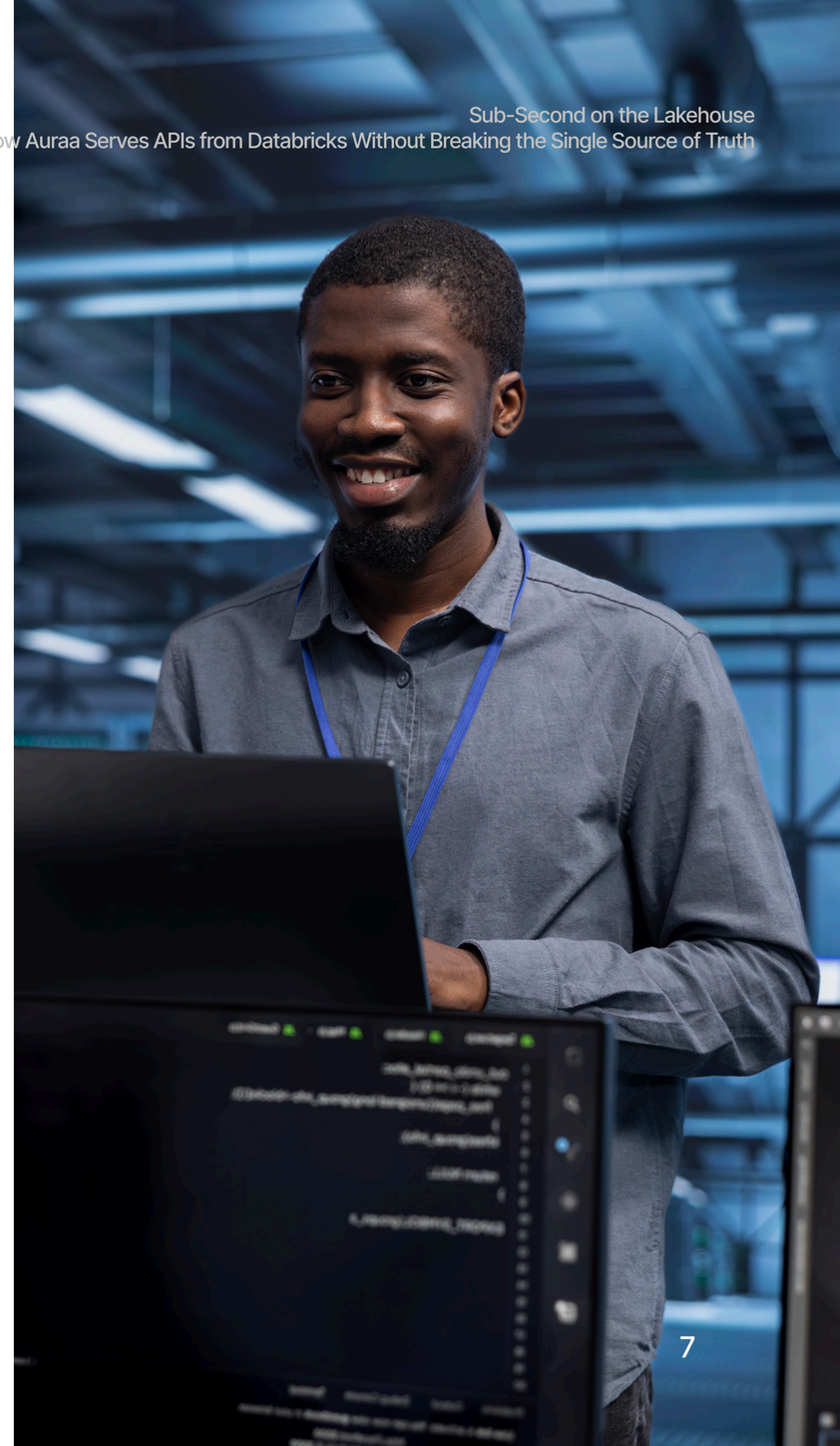
Why this pattern fits this workload

An experienced engineer reading the previous subsections would reasonably ask whether the additional cost of a per-write fan-out is actually tolerable. The answer depends entirely on the workload, and the workload at issue here, governance and configuration state, has a profile that makes the fan-out cost essentially negligible. This subsection makes that argument explicit, because the suitability of the pattern depends on it.

Governance and configuration writes are rare in absolute terms. A tenant onboarding writes a few dozen rows across the registry tables; a tool registration writes a single row; a role grant writes a single row; a project setting change writes a single row. In aggregate, Auraa sees hundreds to low thousands of governance writes per day, not millions per second. Reads against the same state, by contrast, occur on every incoming API request, tens or hundreds of thousands of reads per day, each operating under a strict latency budget.

The asymmetry between write volume and read volume is what makes the fan-out economical. Spending a few additional milliseconds on a rare write so that every subsequent read becomes much faster is a trade that compounds heavily in favor of the system. The same pattern applied to a high-throughput ingestion workload, billions of rows per day, each carrying real-time analytical value, would be the wrong shape entirely. In that setting, per-write cost would dominate the calculus and per-read latency would matter less, and a managed continuous-sync mechanism would more naturally fit the bill. ADR-000 anticipates this distinction in its retrieval recommendations: governance state and business data are treated differently precisely because their write-versus-read access profiles differ so starkly.

The match between the architectural pattern and the workload profile is the kind of “right tool for the job” judgment that is easy to over-generalize once it works. The pattern documented in this section is correct for governance and configuration data on Auraa; it should not be presumed correct for every workload that needs a Postgres serving surface. The next section turns to the read side of the architecture, in which the same dual-surface model that serves Lakebase from Delta for governance state can be selectively applied to business data only where the access pattern warrants it.



The Dual-Surface Read Model

With propagation handled at write time by GovernanceWriter, the read side of the architecture organizes itself into two co-equal surfaces. Consumers do not choose between them on the basis of governance or authority, both surfaces reflect the same logical state, governed by the same Unity Catalog perimeter, but on the basis of access pattern. This section describes the model, then shows how it maps onto Auraa's actual workloads.

Two surfaces, one source of truth

The first read surface is **silver Delta,** accessed through SQL Warehouse. It is the natural fit for batch joins, time-travel queries, cross-table analytics, and any workload that can tolerate warehouse cold-start latency in exchange for the full expressive power of Spark SQL. The second read surface is **Lakebase,** accessed through standard Postgres drivers and connection pools. It is the natural fit for point reads from APIs and agents, where latency budgets are measured in milliseconds and connection pooling is a hard requirement.

A third path also exists for consumers that need to react to state changes continuously: **Delta Change Data Feed** consumed through Structured Streaming. This path is outside the dual-surface point-read model but participates in the same overall architecture, and is mentioned here for completeness.

The decision table below summarizes the mapping.

Access Pattern	Read Path	Why
Point reads from API and agent requests	Lakebase (Postgres)	Sub-second latency, connection pooling, no warehouse cold-start
Batch joins, analytics, time-travel queries	Silver Delta (SQL Warehouse)	Full SQL power, AS OF TIMESTAMP, cross-table joins
Streaming reactions to state changes	Delta CDF (Structured Streaming)	Continuous processing of change events

The important property of this table is that the rows do not describe different data. They describe different paths to the same data. The same primary keys, the same tenant isolation, and the same Unity Catalog perimeter apply regardless of which surface a consumer reads through.

How the model maps onto Auraa's workloads

In practice, the surface chosen by each Auraa subsystem is dictated by that subsystem's latency budget rather than by any deeper architectural commitment. The pattern is consistent enough to be worth enumerating.

The MCP authentication middleware reads role bindings from Lakebase, because every incoming agent call must be authorized in sub-second time. The MCP server reads the tool registry from Lakebase, because the list of tools available to a session must be resolved before the session can begin serving. Project-scoped services read their configuration from Lakebase at request time, eliminating the need for an initialization-phase pre-load.

On the analytical side, the data-quality dashboards read execution results from silver Delta, because the queries involve cross-rule joins and time-travel comparisons that benefit from warehouse-native SQL. Lineage exploration reads the lineage graph from silver Delta for similar reasons: the queries are recursive and benefit from the optimizer's ability to plan across multiple tables. Streaming consumers that react to platform events subscribe to Delta Change Data Feed.

The pattern that emerges is a clean separation by access shape. APIs and agents take the Lakebase path; analytics and dashboards take the Delta path; reactive consumers take the streaming path. None of the three categories require an exception to the model.

What It Buys Us

The architecture described in the preceding sections is not pursued for its own sake. It exists because of the concrete operational outcomes it produces. This section enumerates those outcomes, in the order of how directly they bear on the original problem statement.

The most important outcome is that the **API latency budget is restored.** Hot-path reads return from Lakebase in milliseconds. Warehouse cold-start is no longer in the request path. For an agentic platform that depends on tight feedback loops between users, agents, and tools, this restoration is what makes the platform feel interactive rather than ponderous.

The second outcome is that **governance and tenancy remain coherent across the two surfaces.** The Lakebase tables are registered into Unity Catalog as a mirrored catalog, which means lineage, audit metadata, and permission information are visible alongside the underlying Delta tables. Unity-Catalog-governed access controls apply when Lakebase is read through a SQL warehouse or through a federated query. A qualification is worth naming explicitly, however: when an API reads Lakebase directly over the Postgres wire, which is the hot path this paper is designed to optimize, authorization is enforced by Postgres roles rather than by Unity Catalog ACLs. Auraa handles this by provisioning and rotating those Postgres roles through the same UC-governed automation that manages the Delta side, so that the two access-control mechanisms are administered as one system even though they enforce at different layers. The result is a single coherent governance posture, but achieving it requires deliberate plumbing rather than coming for free.

The third outcome is that **time travel and audit history remain intact.** Delta retains the full history of every governance write through its native versioning, and bronze retains an explicit append-only audit log of every state change. The API tier did not have to surrender either capability in order to obtain its latency budget.

The fourth outcome is that i**nfrastructure cost has been meaningfully reduced.** Approximately thirty-five Delta Live Tables pipelines that were running continuously to support Synced Tables have been retired. The propagation cost that remains is the few additional milliseconds that the parallel Lakebase write adds to the originating request. The fan-out is, in effect, free in steady state.

The fifth outcome is that the architecture provides **an extensibility point** that infrastructure-managed sync did not. Because fan-out is application code, the platform can validate the row, deduplicate against in-flight writes, enrich the Lakebase row with derived columns, or route writes differently per environment, all without modifying infrastructure or invoking platform-level features.

The sixth outcome is the one that originally motivated the redesign: **read-your-own-write is restored.** API callers can write a record and immediately read it back as part of the same operation. No explicit waits, no fallback paths, no quiet inconsistency between two stores.

Taken together, these outcomes describe a serving model in which the lakehouse is no longer in tension with the API tier. The same governed state powers both surfaces.

The Honest Caveats

No architecture is free of trade-offs, and this one is no exception. The section that follows surfaces the costs that Auraa accepted in choosing application-layer fan-out over managed infrastructure sync. They are not trivial, and a team evaluating a similar architecture should weigh them deliberately.

Schema discipline is now an application concern

The most visible cost is that schema discipline has moved from the platform to the application. Every additive Delta column must be matched by a corresponding Postgres DDL change with compatible types and nullability constraints. Synced Tables absorbed this responsibility automatically; GovernanceWriter does not. Auraa pays the cost in code review and migration scripts.

The most common pitfall in this category is the Delta column-defaults two-step. Delta does not allow a DEFAULT clause on a column added through ALTER TABLE ADD COLUMN; the column must be added first and then receive its default through a subsequent ALTER COLUMN SET DEFAULT. Inline DEFAULT declarations on the initial CREATE TABLE statement are supported in Delta 3.1 and later, provided that the table's allowColumnDefaults property is enabled, so the restriction is narrower than it first appears. It nevertheless caught the team early on additive migrations and is now codified as part of the standard schema-change checklist.

Two engines, two type systems

Lakebase is Postgres 16 compatible, the official documentation states plainly that "Databricks database instances only support Postgres 16" ,which means that on the serving side the platform has access to every Postgres feature through that version, including JSONB operations, native array types, and the ON CONFLICT upsert syntax that the fan-out depends on. Within that compatibility envelope, however, Delta and Postgres still have rich but different type systems. Delta provides struct, array, map, and arbitrary-precision decimal types; Postgres provides JSONB, native arrays, and a different precision model for numerics. The mapping between the two is mostly natural, structs become JSONB, arrays of primitive's map to Postgres arrays, high-precision decimals require an explicit precision and scale declaration, but the mapping must be made explicit rather than discovered per table. Auraa codified the rules once and applies them uniformly.

Initial setup has real cost

Standing up the fan-out architecture required a non-trivial amount of one-time work. Lakebase Autoscaling projects had to be provisioned. DDL parity had to be established for more than forty governance tables. A backfill path was needed to seed Lakebase from the existing silver state. A self-heal job was needed to reconcile drift between the two surfaces. None of this work is recurring once it is complete, but it was not free at the outset.

Failure observability requires investment

Because Lakebase failures are non-fatal by design, they must be visible through telemetry rather than through caller-visible errors. Auraa logs every fan-out outcome (silver, Lakebase, bronze) with structured fields, surfaces Lakebase failure rates as a first-class metric, and runs the self-heal pass on a schedule so that any drift is bounded in time. The bronze audit trail is what allows the platform to reconstruct what Lakebase should contain at any given moment.

Connection pool lifecycle on a serverless tier

Databricks Apps runs on a serverless tier in which containers cold-start and may be reaped during periods of low traffic. Connection pool warm-up, idle-connection handling, and reuse across requests had to be designed deliberately rather than inherited from a long-running server process. This is a manageable concern, but not one that solved itself.

These caveats, taken together, define the real ongoing cost of the architecture. None of them is severe enough to outweigh the latency benefits the architecture delivers, but each of them must be acknowledged and addressed in any environment that adopts a similar pattern.

Conclusion

The architecture described in this paper was driven by a single tension: ADR-000 establishes Delta Lake as the single write authority for all platform state, and that commitment imposes a latency cost on any workload that must serve state through an API. Resolving the tension required honoring the principle in full while finding a way to deliver sub-second reads on top of it.

Two paths were available. The first, Databricks Synced Tables, kept the principle intact through an infrastructure-managed sync but exacted a cost in continuous compute, sync latency, and a lack of extensibility that the team ultimately found unacceptable. The second, the application-layer fan-out implemented by GovernanceWriter, kept the principle intact through a deliberate write-time propagation that costs only a few additional milliseconds per write and yields read-your-own-write consistency on the serving surface. The team chose the second.

The resulting architecture is one in which the same governed Delta state is reachable through two physical surfaces: silver Delta through SQL Warehouse for analytical workloads, and Lakebase through Postgres for API workloads. Delta remains the contract. Lakebase is a downstream read-only co-surface, populated at write time, that gives API callers the latency profile they require without compromising the lakehouse as the source of truth.

This dual-surface model is the substrate that every subsequent Auraa capability, the agent framework, the MCP tool layer, the DeltaBus event bus, the semantic-flow runtime, stands on. None of those capabilities would have shipped on a platform that forced a choice between truth and speed. The contribution of GovernanceWriter is to make that choice unnecessary.

References

- Auraa: An Agentic Data Activation Platform for Databricks
- DeltaBus: A Lakehouse-Native Event Bus Pattern for Data Platforms
- Configurable Organization Meets the Lakehouse: How Tenant-Scoped Delta Paths Enable Single-Source-of-Truth Retrieval
- How We Made Databricks Apps APIs Fast Without Breaking Our Single Source of Truth

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