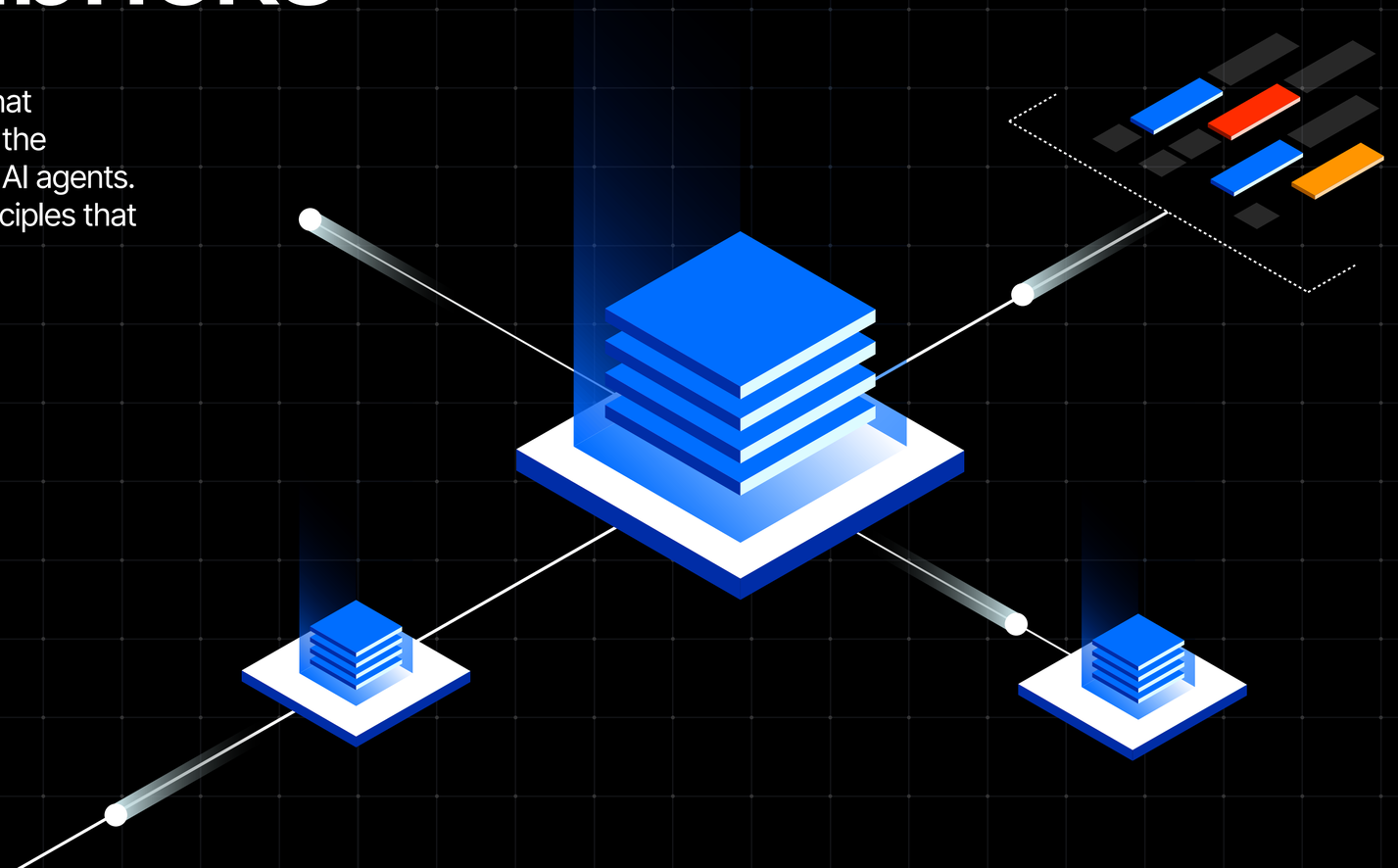
Covasant
AI-Driven. Human-Inspired.

Rethinking Data Engineering

An AI-First, Metadata-Driven Platform for Databricks

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 whitepaper explains why the platform was built, the design principles that distinguish it, and the outcomes those choices produce.

Alan Dennis
VP, AI & Data Innovation,
Covasant



Executive Summary

Modern data engineering faces significant scalability challenges. Despite enormous investment — six-to-twelve-month delivery cycles, multimillion-dollar budgets, large teams of specialized engineers — the results can be fragile. Pipelines break when schemas drift. Quality rules often exist in scattered SQL scripts that are hard to maintain. Governance is frequently treated as an afterthought. And many new projects start from scratch, because logic is trapped in code rather than captured in reusable, machine-readable form.

The root cause is structural: the industry largely treats data engineering as a software engineering problem. Build code, deploy code, monitor code. But data engineering is fundamentally a data management problem. The decisions that drive pipelines — what to ingest, how to clean it, what quality rules to apply, how to transform it — are not inherently code. They are metadata: structured, versionable, queryable records that describe what should happen, leaving the how to the platform.

Auraa is built on this insight. It is a Databricks-native platform where major data engineering decisions — from source connection parameters to quality rule definitions to transformation specifications — are captured as structured metadata stored in Delta Lake tables. This metadata follows the same medallion architecture (bronze → silver → gold) as the data itself: raw metadata is discovered, cleansed metadata is validated, and enriched metadata is annotated with business context.

This design produces three key consequences:

1. AI agents can autonomously create, manage, and optimize data engineering workflows, because they operate on structured data rather than parsing code.
2. Pipelines become highly reproducible, because behavior is driven by versioned metadata records rather than mutable code.
3. Governance is structural, because Unity Catalog governs both the data and the metadata that drives it.

The result: what traditionally takes months and significant budgets, Auraa can often deliver in weeks with a leaner team — while maintaining strong governance, high quality, and the ability for AI agents to continuously improve the system.

Why We Built Auraa

The Observation

After years of building enterprise data solutions — fraud detection systems, regulatory reporting platforms, customer analytics lakes — a pattern became unmistakable: at least 70% of the engineering effort on any project is repetitive pattern-matching, not creative problem-solving.

Connecting to a PostgreSQL database looks almost identical to connecting to SQL Server. Writing a null-check quality rule for a financial column is structurally the same as writing one for a healthcare column. Mapping source fields to a canonical model follows the same logic whether the source is SAP or Salesforce. Engineers were spending the majority of their time on work that could be described in a paragraph but took weeks to implement, test, and deploy.

The question was not "can we automate this?" — automation tools exist. The question was: **why do those tools fail to deliver on their promise?**

Why Current Approaches Fall Short

The answer lies in how existing platforms represent data engineering decisions.

Code-first platforms embed decisions in Python scripts, SQL procedures, and notebook cells. This makes them flexible but opaque: an agent cannot inspect a 200-line notebook and reliably determine which tables are being ingested, what quality rules are applied, or what the transformation logic does. The knowledge is there, but it is locked in procedural syntax that resists machine reasoning.

Configuration-first platforms extract decisions into YAML or JSON files. This is better — an agent can parse a YAML config — but configurations are typically incomplete. They capture what but not why. They describe the current state but carry no history. And they are disconnected from the runtime: the config lives in a git repo, the data lives in a warehouse, and the relationship between them is maintained by convention, not by the platform.

Low-code platforms provide visual interfaces that generate code or configurations behind the scenes. This accelerates human operators but does nothing for agents. An LLM cannot drag and drop in a visual pipeline builder.

The fundamental issue is the same in all three approaches: data engineering decisions exist outside the data platform. They live in code files, config files, or UI state — not in the governed, queryable, versioned data store where agents and humans can both reason about them.

The Founding Thesis

This led to the thesis that Auraa is built upon:

If core data engineering decisions can be captured as **structured metadata** — stored in the same governed data platform as the data itself — then AI agents can autonomously create, manage, and optimize those decisions at a scale and consistency that is difficult for human teams to maintain manually.

This is not about replacing data engineers. It is about elevating their role from writing repetitive pipeline code to directing intelligent agents that execute from curated metadata. The engineer becomes the architect of intent; the platform handles the execution.

Three bets define the implementation of this thesis:

1. **Databricks as the substrate.** Delta Lake for storage, Unity Catalog for governance, SQL Warehouse and Spark for compute, MLflow for model management. No middleware layer; build directly on the platform.
2. **AI-and-agent-first from day one.** Not a traditional tool with AI added later, but a system designed from the ground up so that agents are the primary operators and humans provide intent and oversight.
3. **Metadata as data.** Not metadata about data — metadata as data. Stored in tables, governed by the same policies, flowing through the same medallion pipeline, queryable by the same engines.

Design Principle
AI-First and Agent-First

Most platforms that claim to be "AI-powered" have added a chatbot or copilot to an existing human-first tool. The user interface was designed for humans; the AI is an assistant that helps humans use it faster. This is agent-augmented design, and it is valuable — but it is limited. The AI can only be as capable as the human interface it wraps.

Auraa inverts this relationship. The platform is designed for agents first, and human interfaces are built on top of the same abstractions agents use.

Tools Are the Primary Interface

Every platform capability — registering a data source, running a quality check, generating a transformation spec, approving an ingestion plan — is implemented as a tool: a function with typed inputs, typed outputs, a version number, and governance metadata. These tools are registered in NEXARA, Auraa's tool and agent registry, where they can be discovered by any agent at runtime.

This is not just a convenience wrapper around an internal API. The tool interface is the primary interface. This provides a unified path for execution. When a human user clicks "Run Quality Check" in the web interface, the UI calls the same tool that an agent would invoke. This generally ensures that operations — regardless of who initiates them — follow the same authorization, audit, and execution path.

Agents Orchestrate, Tools Execute

The separation between agents and tools is deliberate and strict:

- **Agents** handle planning, discovery, and decision-making. They are probabilistic — an LLM-based agent might propose different quality rules for the same dataset depending on context. They reason about goals and constraints.
- **Tools** handle execution. They are deterministic — given the same inputs, they produce the same outputs. They implement the business logic of ingestion, quality analysis, transformation, and governance.
- This separation produces a critical property: **reproducibility**. An agent's plan might be generated by an LLM, but the plan itself is a sequence of tool invocations with explicit parameters. Re-executing that sequence produces the same result. The planning is creative; the execution is mechanical.

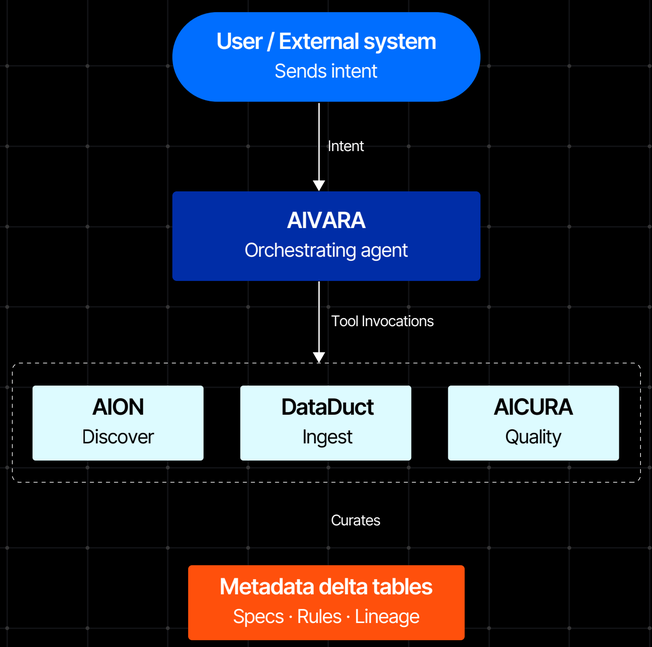
Human-in-the-Loop, Not Human-in-the-Path

Auraa's default execution model is agent-driven. An orchestrating agent (AIVARA) receives a high-level intent — "connect to my PostgreSQL database and ingest the orders schema" — and autonomously:

- Discovers available tools for source connection and metadata probing
- Connects to the source and discovers schemas, tables, and columns
- Generates ingestion specifications based on discovered metadata
- Proposes quality rules based on column types and data patterns
- Presents the complete plan to a human for review and approval

The human reviews the plan, approves it (or modifies it), and the agent executes. The human provides direction and oversight; the agent handles the mechanical work. This is fundamentally different from a system where humans build and click through each step while an AI suggests optimizations on the side.

For scenarios where trust is established and the patterns are well-known, the human approval step can be minimized or fully automated — allowing the agent to execute autonomously under established governance and audit policies.



Design Principle
Built for Databricks

Auraa is not cloud-agnostic middleware layered on top of Databricks. It is built for Databricks, leveraging the platform's native capabilities rather than reimplementing them.

This is a deliberate architectural decision with strategic implications. Every capability that Databricks provides natively — storage, compute, governance, identity, scheduling — is one fewer component to build, operate, and secure. And every Auraa operation generates Databricks consumption: compute cycles for ingestion, storage for Delta tables, SQL Warehouse queries for quality checks. The platform is a **consumption amplifier**, not a competing cost center.

Delta Lake as the Universal Substrate

All data in Auraa lives in Delta Lake tables organized in the medallion pattern:

- **Bronze -** Raw, append-only data as received from source systems. No transformations, no filtering. The immutable record of what was ingested and when.
- **Silver -** Cleansed, deduplicated, type-conformed data with quality rules enforced. The trusted analytical base.
- **Gold -** Purpose-built datasets for specific consumption patterns: reporting, machine learning, operational analytics.

Critically, Auraa's metadata follows the same pattern. Ingestion specifications, quality rule definitions, transformation models, audit records, and lineage data all live in Delta tables within the tenant's catalog. This is not a convenience — it is a core design decision explored in detail in the next section.

What Auraa Leverages Natively

Capability	Databricks Feature	What Auraa Avoids Building
Storage	Delta Lake (ACID, time travel, CDF)	Custom data store
Governance	Unity Catalog (catalogs, schemas, ACLs)	Custom RBAC engine
Compute	SQL Warehouse + Spark clusters	Dedicated ETL servers
Secrets	Databricks Secret Scopes	External vault management
Event bus	DeltaBus (Delta tables + CDF)	Kafka / Azure Event Hubs
Identity	On-behalf-of user tokens	Custom OAuth provider
ML/AI	MLflow, Model Serving, FMAPI	External model hosting
Scheduling	Databricks Jobs and Workflows	External orchestrator
Lineage	Unity Catalog lineage tracking	Custom lineage graph

Event-Driven Without External Infrastructure

Auraa's components communicate through events — a data source registered, an ingestion completed, a quality rule violated. Rather than introducing Kafka or Azure Event Hubs, Auraa uses **DeltaBus:** a pattern that treats Delta Lake tables with Change Data Feed (CDF) enabled as native message buses.

Publishing an event is a Delta table write. Consuming events is a CDF read from the last processed version. The result: a complete publish-subscribe event bus with at-least-once delivery semantics, permanent queryability, and native Unity Catalog governance — at a fraction of the cost of managed message queue services.

This pattern is documented in detail in the companion whitepaper: *DeltaBus: A Lakehouse-Native Event Bus Pattern for Data Platforms.*

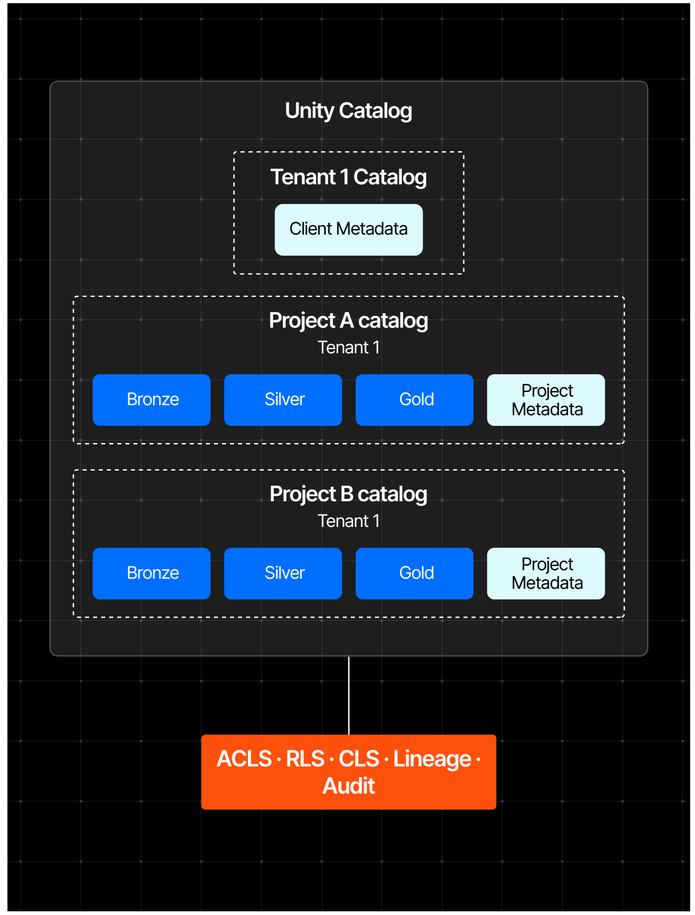
Design Principle
Unity Catalog as the Security Foundation

Multi-tenancy and security are not features added to Auraa — they are structural properties of its architecture. The entire security model is built on Unity Catalog, which means that no Auraa-specific security code stands between a user and their data. The platform delegates security entirely to the engine it runs on.

Catalog-per-Project Isolation

Each project (or data domain) within a tenant receives a dedicated Unity Catalog catalog, while the tenant itself also receives a top-level catalog containing the client's global metadata. This creates a fundamentally stronger isolation boundary than schema-level approaches:

- A project's data, execution logs, quality metrics, and audit records all live within its own catalog. There is no shared data plane across projects.
- The tenant-level catalog serves as a unified registry for cross-project and client metadata, ensuring global context is preserved without comingling transactional data.
- Unity Catalog ACLs enforce access at the catalog level. A user with access to Project A's catalog has zero visibility into Project B's catalog — enforced by the engine, not by application-layer filters.
- Administrative operations (catalog creation, schema provisioning, initial ACL setup) are automated by Auraa's control plane but governed by Unity Catalog's native permission model.



On-Behalf-Of User Identity

Most multi-tenant platforms use an elevated service principal to access data on behalf of users. This is an anti-pattern: the service principal has broad permissions, and the platform must implement its own authorization layer to restrict access.

Auraa uses Databricks' **on-behalf-of** pattern: operations execute with the calling user's own identity. When a data engineer runs a quality check, the query is logged under their identity, and Unity Catalog enforces their specific permissions. The platform delegates the decision of whether a user should see a particular table directly to Unity Catalog.

Row-Level and Column-Level Security

Sensitive data — personally identifiable information, financial details, health records — is governed by Unity Catalog's Row-Level Security (RLS) and Column-Level Security (CLS) policies. These are defined once and enforced across every access path: Spark queries, SQL Warehouse queries, API calls, agent tool invocations. No application-layer filtering that clever queries might bypass.

Governance as an Execution Property

Auraa's governance component, AIGENE, enforces policies at the tool invocation layer. Before any tool executes — before an ingestion runs, before a transformation applies, before data is exported — AIGENE checks the policy:

- Is this user authorized to invoke this tool for this tenant?
- Does this operation comply with the tenant's data residency requirements?
- Is the target data classified at a sensitivity level this user can access?

If the check fails, the operation does not execute. This strict enforcement model significantly reduces the risk of unauthorized access. Governance is not just a report generated after the fact — it acts as a primary gate for operations.

The audit trail follows the same principle: tool invocations are recorded — who, what, when, why — as a Delta table in the tenant's catalog. Audit data is treated as first-class data, queryable with the same tools used for analytics.

Data-Driven Grants Management

Managing permissions at scale in a multi-tenant environment often devolves into a web of manual configuration steps that drift from corporate policy over time. Auraa resolves this by extending Unity Catalog with **Data-Driven Grant Management:**

- **Automated, Templated Application:** Whenever a new tenant or data domain (project) is created, Auraa automatically applies pre-defined grant templates. This highly automated approach ensures that a catalog's permissions consistently align with the governance baseline from the moment it is provisioned, significantly reducing manual error.
- **Simplified Visualization:** Understanding effective permissions across nested groups and privileges can be notoriously difficult. Auraa abstracts Unity Catalog's granular grants into simplified, data-driven visual representations, giving platform administrators clearer visibility into access maps without requiring complex SQL queries.
- **Active Drift Detection:** A common vulnerability in governed platforms is configuration drift—when administrators manually add or remove grants outside of managed processes to solve immediate issues. Auraa continuously monitors Unity Catalog against its declarative templates, identifying and reporting grants that deviate from the defined baseline.

This approach ensures that security is structurally sound at creation and remains highly auditable and strictly enforced throughout the lifecycle of the data platform.

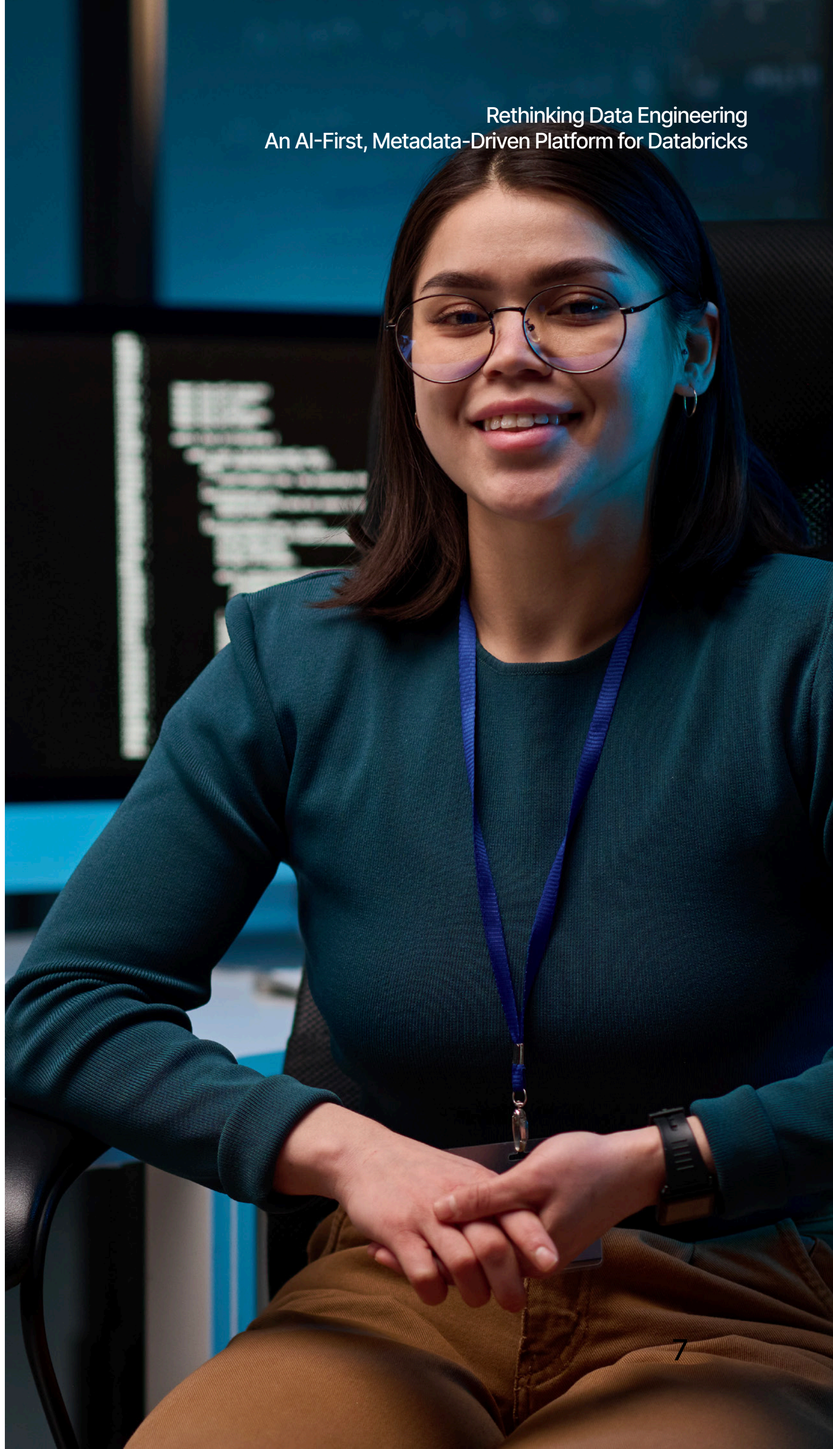
Design Principle
Data-Driven Consistency and Reproducibility

This is the design choice that most distinguishes Auraa from conventional platforms: the decision to drive all data engineering activities from structured metadata rather than procedural code.

The Problem with Code-Driven Pipelines

In a traditional data platform, an ingestion pipeline is a Python script or a notebook. A quality rule is a SQL statement in a stored procedure. A transformation is a dbt model or a PySpark transformation function. Each of these is code — and code has properties that make it hostile to automation, audit, and agent reasoning:

- **Code is opaque.** An LLM can read Python, but reliably extracting "this pipeline ingests the orders table from PostgreSQL with a 5-minute incremental window using the updated_at column" from a 200-line script is fragile.
- **Code drifts silently.** A developer changes a filter condition, adds a column mapping, or tweaks a retry policy. The change is in git, but the relationship between the code change and its business impact is implicit.
- **Code is not queryable.** You cannot ask "how many pipelines use incremental ingestion?" or "which quality rules apply to financial tables?" without parsing every file in the repository.
- **Code is not composable for agents.** An agent cannot modify a Python function the way it can update a row in a metadata table. Code changes require understanding syntax, context, and side effects.



Metadata-Driven Everything

Auraa replaces code-driven pipeline logic with metadata-driven declarations:

Traditional Approach	Auraa's Data-Driven Approach
Ingestion logic in Python notebooks	Pipeline behavior driven by ingestion specification records
Quality rules in SQL scripts	Quality rules as versioned rows in a quality_rules table
Transformation logic in dbt models	Transformations driven by UDM specification records
Configuration in YAML files on disk	Configuration as rows in metadata.settings tables
Lineage reconstructed from logs	Lineage captured as metadata at execution time
Change history in git commits	Change history in Delta time-travel and audit tables

An ingestion specification, for example, is a record that describes: the source system, the source table, the target bronze table name, the ingestion mode (full refresh or incremental), the watermark column (if incremental), the batch assignment, the schedule, and the approval status. This record is stored in a Delta table, governed by Unity Catalog, and versioned by Delta's time-travel capability.

When the ingestion engine runs, it reads the specification and executes accordingly. The engine does not contain source-specific logic — it delegates to a connector (identified by the spec's source type) and writes to the target table (identified by the spec's target name). The engine is generic; the specification is specific.

Why This Produces Consistency

Because most ingestion specifications share the same schema, pipelines — regardless of source type, target layer, or tenant — become structurally consistent. A PostgreSQL ingestion and an API ingestion differ primarily in their metadata values, rather than in bespoke execution paths. This drastically reduces an entire category of bugs that arise when different pipelines are coded differently by disparate teams.

The same principle applies to quality rules, transformation models, and governance policies. The platform promotes consistency not just through code reviews or style guides, but through the structured enforcement of the metadata itself.

Why This Produces Reproducibility

If you know the metadata that drove a pipeline run — the ingestion spec, the quality rules, the checkpoint state — you can reproduce that run exactly. The metadata is immutable (Delta time-travel), the engine is deterministic (tools produce the same output for the same input), and the execution environment is captured in audit records.

This is not theoretical: each pipeline run records the version of the ingestion spec it executed against, the version of the quality rules it applied, and the checkpoint it started from. Rolling back to a previous state is a metadata operation, not a code deployment.

Design Principle
Metadata as Data Curated and Enriched

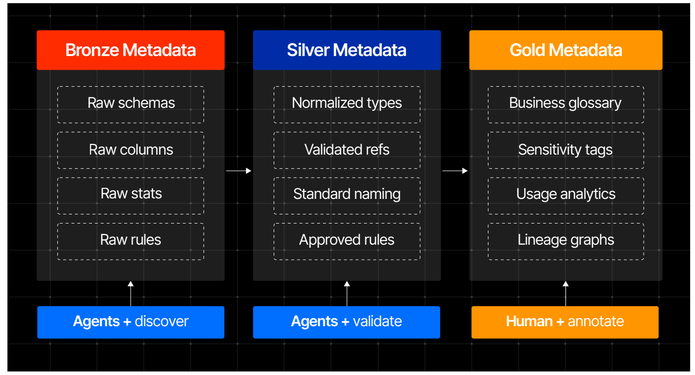
If previous section argued that data engineering should be driven by metadata, this section goes further: metadata itself deserves the same treatment as the data it describes. It should be ingested, cleansed, validated, enriched, and governed through the same medallion pipeline.

Metadata Follows the Medallion Pattern

When an Auraa agent connects to a new data source and discovers its schemas and tables, the resulting metadata does not go directly into a "catalog" or "registry." It enters the **bronze metadata layer** — raw, as-discovered, with whatever inconsistencies or gaps the source system has.

From bronze, the metadata flows to **silver** — where types are normalized, naming conventions are standardized, cross-references are resolved, and validation rules are applied. A column reported as VARCHAR(255) in PostgreSQL and NVARCHAR(MAX) in SQL Server is mapped to a canonical type. A table name in camelCase is normalized to snake_case. Foreign key relationships are validated against actual data.

From silver, the metadata reaches **gold** — enriched with business context. Business glossary terms are linked. Sensitivity classifications are assigned. Usage analytics (which tables are queried most frequently? which columns drive the most quality failures?) are computed. This is the layer where human data stewards add the knowledge that machines cannot discover on their own.



Why Agents Need Curated Metadata

An AI agent reasoning about a data estate needs more than raw column names and types. It needs to know:

- Which tables are related to which business processes?
- Which columns contain sensitive data that requires masking?
- Which ingestion patterns have produced the best quality outcomes?
- Which source systems are prone to schema drift?
- Which quality rules have high false-positive rates and should be re-tuned?

These are questions that can only be answered when metadata has been curated — cleansed, structured, enriched — to the same standard as the data it describes. An agent operating on raw, uncurated metadata makes poor decisions. An agent operating on gold-tier metadata makes decisions that rival those of an experienced data engineer.

This is the virtuous cycle at the heart of Auraa's design:

1. Agents discover raw metadata from source systems (bronze)
2. Agents and automated rules validate and normalize metadata (silver)
3. Humans and agents enrich metadata with business context (gold)
4. Agents use gold metadata to make better decisions about ingestion, quality, and transformation
5. Those decisions produce more metadata, which feeds back into the cycle

Agents as Metadata Curators

In traditional platforms, metadata curation is a human responsibility — and it is usually neglected, because engineers are busy building pipelines. In Auraa, agents are first-class metadata curators:

- **AION** (the discovery component) probes source systems and generates metadata: schemas, tables, columns, types, constraints, relationships, row counts, size estimates. This metadata is created programmatically, at machine speed, with machine consistency.
- **AICURA** (the quality component) analyzes data patterns and proposes quality rules: null checks, range validations, uniqueness constraints, referential integrity rules. Each proposed rule is itself metadata — a record in the quality rules table — that a human reviews and approves.
- **AITERA** (the transformation component) examines source metadata and target model requirements, then generates transformation specifications. Again: metadata in, metadata out.
- **AIGENE** (the governance component) classifies data sensitivity, assigns access policies, and enforces compliance rules — all based on metadata about the data's content and context.

The agents do not merely consume metadata to do their work. They produce metadata as a primary output of their work. The platform's metadata estate grows richer with every operation.

Gaining Insights from Metadata

Because metadata is stored in Delta tables — the same tables that Databricks' SQL analytics, dashboards, and ML tools operate on — it is immediately available for analysis:

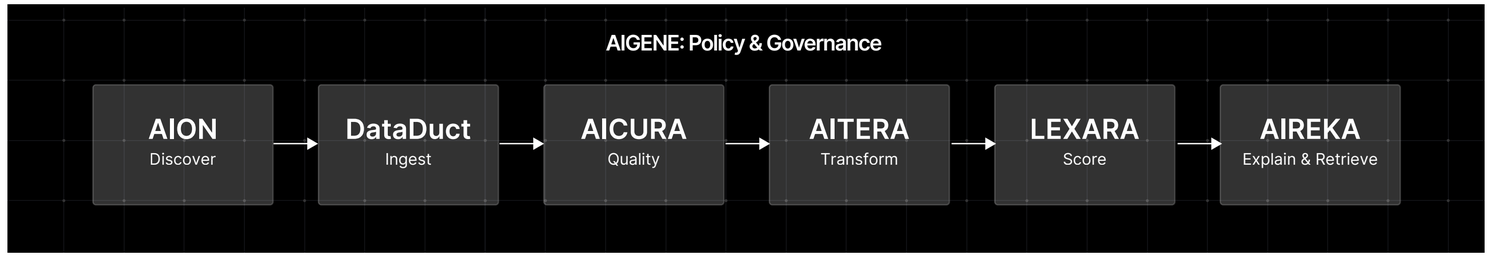
- **Operational insights:** Which sources are ingesting the most data? Which pipelines have the longest runtimes? Where are the bottlenecks?
- **Quality insights:** Which tables have the highest quality failure rates? Which rules are most frequently violated? Are quality trends improving or degrading?
- **Governance insights:** Which data is classified as sensitive? Is all sensitive data properly masked? Are access policies consistently enforced?
- **Optimization insights:** Which tables could benefit from incremental ingestion instead of full refresh? Which quality rules are redundant? Where is compute being wasted on unnecessary transformations?

These insights are not produced by a separate analytics tool reading logs. They are the natural result of querying the metadata tables that drive the platform. The platform's operational data is its analytical data.

The Architecture

Auraa's component architecture follows from the principles above. Each component has a clear responsibility, exposes its capabilities as tools via NEXARA, and communicates through DeltaBus events.

Data Plane - The Pipeline



AION discovers source systems: schemas, tables, columns, relationships. Generates ingestion specifications for review.

DataDuct executes ingestion: reads from sources via pluggable connectors, writes to bronze Delta tables. Supports full refresh, incremental (watermark-based), and CDC modes.

AICURA enforces data quality: runs rules against bronze data, captures metrics, routes clean records to silver and failed records to quarantine.

AITERA applies transformations: maps silver data to gold-layer models (Universal Data Model) based on transformation specifications.

LEXARA applies business rules and scoring: computes risk scores, rule hits, and aggregates over the structured data models.

AIREKA powers explainability and semantics: generates natural-language narratives for scores, tracks lineage, and serves as the semantic layer for RAG (Retrieval-Augmented Generation) and agentic search over curated data products.

Control Plane — The Brain

- **FOUNDRA** provisions infrastructure: creates tenant catalogs, schemas, tables, and initial configurations. Highly automated onboarding.
- **NEXARA** manages the tool and agent registry: every tool is discoverable at runtime with its contract, version, and governance requirements.
- **AIGENE** enforces governance: authorization checks on every tool invocation, policy enforcement, audit trail capture, sensitivity classification.
- **AIVARA** orchestrates workflows: receives intents, discovers tools, builds plans, delegates execution, monitors completion.

Foundation Layer

- **aura_core** provides shared utilities: DeltaBus messaging, secret resolution, engine abstraction, logging, error hierarchy, and base connector classes.
- **INFRAIA** manages Databricks infrastructure: cluster provisioning, secret scope management, workspace configuration.

Solution Layer

- **Kona AI APIs** FastAPI microservices exposing platform capabilities as RESTful endpoints
- **aura-web** React/TypeScript application providing the human interface, built on the **Auraa Semantic Flow** pattern for data-driven, agent-generated UI surfaces.

Outcomes

The design choices described in this paper are not theoretical. They are implemented, tested, and operating. Here is what they produce:

Speed

Activity	Traditional	With Auraa
Connect a new data source	2–4 weeks (connector development)	<1 hour (agent-assisted probing)
Generate ingestion pipelines for 50 tables	4–8 weeks (per-table development)	Hours (metadata-driven spec generation)
Implement quality rules for a new dataset	1–2 weeks (custom SQL development)	Hours (agent-proposed, human-approved)
Onboard a new tenant	1–2 weeks (manual provisioning)	<1 hour (highly automated)
First data in bronze table	2–4 weeks	Days
Production silver layer	2–3 months	2–4 weeks

Cost

Metric	Traditional	With Auraa
Engineering effort per dataset onboarded	High ($15k–$50k equivalent)	Low ($500–$2k equivalent)
Infrastructure overhead (messaging, orchestration)	Moderate to High	Low (Native DeltaBus)
Governance implementation	Separate project phase	Structural (built-in)
Ongoing pipeline maintenance	20–40% of engineering time	<15% (metadata-driven automation)

Quality and Governance

Activity	Traditional	With Auraa
Data quality rule coverage	Limited (10–20% of tables)	Extensive (agent-proposed coverage)
Governance compliance	Partial, relies on manual audits	Structural, enforced at execution layer
Lineage completeness	Reconstructed, gaps common	Highly automated, captured at execution
Audit trail coverage	Inconsistent	Comprehensive tracking of tool invocations
Pipeline reproducibility	Low (code-dependent, configuration drift)	High (metadata-versioned)

Conclusion

Auraa exists because we believe the next generation of data platforms will not be built by adding AI assistants to human-first tools. They will be built for agents from the ground up — with humans providing intent, domain expertise, and oversight.

The key enabler of this shift is the treatment of metadata as data: structured, versioned, curated, and enriched through the same governed pipeline as the business data itself. When metadata is this clean and this accessible, agents can do more than generate SQL or suggest optimizations — they can reason about entire data ecosystems, propose improvements, and execute changes with full governance and audit.

Auraa demonstrates a tangible shift on Databricks today, empowering agents to discover sources, generate ingestion specifications, propose quality rules, and orchestrate end-to-end pipelines — driven by structured metadata, governed by Unity Catalog, and highly auditable down to the tool invocation.

The traditional data engineering model — hand-coding pipelines, manually configuring rules, bolting on governance at the end — produced the data platforms we have. Auraa is designed to produce the data platforms we need: highly consistent, reproducible, governed, and intelligent.

Built for Databricks. Secured by Unity Catalog. Driven by metadata. Orchestrated by agents.

Your Next Steps with Auraa

Auraa is rapidly evolving to meet the needs of modern, AI-first data organizations. To explore how the platform can transform your data engineering workflows, consider the following next steps:

- **Schedule a Deep Dive:** Contact our architecture team for a personalized walkthrough of the Auraa component suite and its integration with your Databricks workspace.
- **Request a Proof of Value:** Identify a high-impact data domain and experience first-hand how agentic orchestration can accelerate your delivery timelines. (Email: kathy.harnois@Covasant.com)

Transform your data engineering from manual building to intelligent directing.

Further Reading

- **DeltaBus: A Lakehouse-Native Event Bus Pattern for Data Platforms** How Auraa eliminates external message queues by treating Delta tables as native event buses.
- **Auraa Semantic Flow: Transient, Data-Driven Interfaces for Agent-Human Collaboration** How Auraa's agents communicate with humans through rich, structured interaction surfaces.
- **Semantic Documentation Search for Agentic Platforms** How Auraa makes its own documentation discoverable to AI coding agents using Delta-native vector search.
- **Auraa: An Agentic Data Activation Platform for Databricks** Detailed technical architecture of every Auraa component.

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