Covasant

AI-Driven. Human-Inspired.

Auraa

An Agentic Data Activation 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 paper describes its architecture, design philosophy, and production deployment.

Executive Summary

The enterprise data platform market is currently navigating a transition from human-centric to agent-centric operations. Many existing platforms were designed primarily for manual workflows, UIs, and scheduled jobs. While adding AI assistants to these environments provides immediate productivity gains, the underlying architectural constraints often limit the degree of autonomy and governance that agents can achieve.

Auraa takes a fundamentally different approach.

Rather than bolting AI onto yesterday's platform, Covasant built tomorrow's data platform for tomorrow's challenges. Auraa is a Databricks-native platform designed with a **Prompt-First** philosophy. While the underlying architecture is tool-first, composed of typed, discoverable, independently testable functions - the primary interface for users is natural language. These prompts drive autonomous agents that discover and invoke the right tools at runtime to achieve complex business outcomes. Agents are the orchestrators, but the prompt is the interface.

This single architectural decision has cascading consequences. Users express intent; agents plan workflows; tools execute them. If an agent makes a bad plan, it produces a failed tool call - not a corrupted dataset. New capabilities are discovered by agents at runtime through a central registry, so extending the platform does not require rewriting the agents or the prompt-handling logic. All components program against stable abstract contracts, making the system testable without standing up infrastructure and extensible without breaking existing consumers. And components communicate through a lakehouse-native event bus built entirely on Delta Lake, eliminating external message queue infrastructure.

The result is a platform that decomposes the data lifecycle into well-defined components - INFRAIA, Foundra, AION, DataDuct, AICURA, AITERA, LEXARA, AIREKA, AIGENE, NEXARA, AIVARA, and AURA_CORE - each exposed as tools that agents compose into solutions. Delivery that once took months can be compressed to weeks. Governance that was once bolted on as an afterthought is built in from the first line of code. Capabilities that were once reimplemented for every project are registered once and reused everywhere.

Kona AI Databricks - Covasant's fraud detection and risk analytics solution - is the first production application built on Auraa. It validates the platform's ability to deliver a complete, domain-specific analytical solution through composition of modular components, and serves as both a flagship deployment and a reference architecture for future applications across compliance, risk, optimization, and analytics domains.

The Problem: Why Legacy Platforms Cannot Keep Pace

The Traditional Model and Its Symptoms

Most organizations building modern data and AI platforms run into the same structural problems, regardless of industry, scale, or cloud provider. The symptoms are remarkably consistent.

Delivery is slow. Standing up a usable lakehouse or data warehouse typically takes six to twelve months. Multi-domain or multi-line-of-business rollouts can stretch to eighteen months or longer. By the time the platform is ready, the business requirements that motivated it have evolved, or the executive sponsors have moved on.

Costs are high and poorly contained. Large teams of senior architects and data engineers spend months designing and building pipelines that are functionally similar to pipelines they built on the last project. Rework is constant as requirements shift against weak modularity. Each new data source or use case triggers another cycle of custom engineering.

Tooling is fragmented. ETL, data quality, catalog management, business rules, and governance live in separate products with ad-hoc integration. There is no unified contract surface, no common way to discover what the platform can do, compose capabilities into workflows, or enforce policies consistently across tools.

Governance arrives late. Policies are bolted on after pipelines are built. Lineage and access controls are inconsistently applied. Identity management operates in a different system from data operations. Compliance teams audit artifacts that were never designed to be auditable.

Reusability is minimal. Each project effectively reimplements ingestion, data quality, transformations, and scoring from scratch. Knowledge stays in notebooks and tribal memory. The investment in solving a problem once does not compound across the next project.

These are not failures of execution. They are structural properties of the traditional model: project-centric delivery on architectures designed for manual operation.

The AI Retrofit Problem

The industry's response to these symptoms has been to add AI capabilities to existing platforms. Every major data platform vendor now offers some form of AI assistant, copilot, or intelligent automation layer.

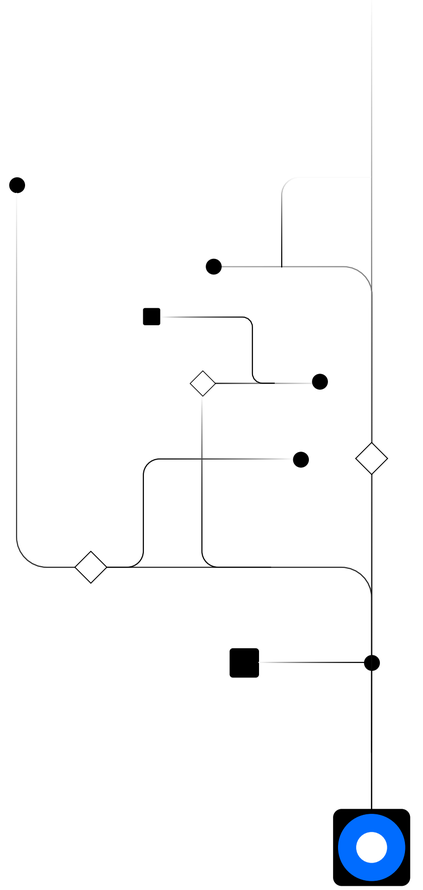
But the underlying architecture is unchanged. These platforms were designed for human operators working through UIs, notebooks, and scheduled jobs. The AI layer can suggest SQL, generate documentation, or recommend optimizations, but it cannot act autonomously, because the platform's capabilities are not expressed in a form that agents can reliably invoke. A chatbot that generates a SQL query still requires a human to review it, run it, check the results, and decide what to do next.
This creates a capability ceiling. AI assistants can help a human work faster, but for a platform to autonomously compose, execute, and govern multi-step workflows, the underlying architecture must be designed to accommodate machine-scale orchestration.

The Data Delivery Gap

Together, the traditional model and the AI retrofit problem create what we call the data delivery gap: the distance between what the business needs from its data platform and what the platform can deliver in the time and budget available.

AI initiatives stall while foundational pipelines are being built. Business stakeholders lose trust in delivery timelines. Governance and compliance lag implementation. The cost of experimentation is too high, so teams default to conservative, slow-moving approaches that further widen the gap.

Closing this gap requires more than better tooling on an old architecture. It requires a platform designed from the ground up for the way data work will be done, by agents, at scale, with governance built in from the start.



Auraa's Design Philosophy: Built for Agents, Not Retrofitted

Auraa's architecture is grounded in a single thesis: **a data platform designed for agent-driven orchestration will outperform a human-designed platform retrofitted with AI.** Every architectural decision in the platform flows from this thesis.

This section describes the design principles that make Auraa fundamentally different from platforms that add AI capabilities after the fact. These are not features; they are structural properties of the system that would be prohibitively expensive to retrofit into an existing architecture.

Prompt-First, Tool-First

Auraa's design logic follows a clean hierarchy: **Users prompt, Agents plan, Tools execute.**

The primary entry point for any operation in Auraa is a natural language prompt. This prompt-first model democratizes access to complex data capabilities, allowing users to express intent without needing to understand the underlying tool interfaces.

Every capability in Auraa, ingestion, data quality, transformation, scoring, infrastructure provisioning, governance enforcement, is exposed as a tool: a function or class with explicit typed inputs, typed outputs, and a machine-readable contract.

This is not a wrapper around a human-facing UI. The tool is the capability. When a user prompts the platform to "onboard a new SAP source," an agent interprets the intent, discovers the necessary metadata-discovery and infrastructure tools through NEXARA, and plans a multi-step workflow. The implications of this design are transformative. Tools can be composed programmatically by agents without fragile UI automation or prompt engineering against unstructured outputs. Execution is deterministic, typed inputs, predictable outputs, auditable side effects. This separation makes agentic orchestration safe and testable. The blast radius of a bad plan is a failed tool call, not a corrupted dataset.

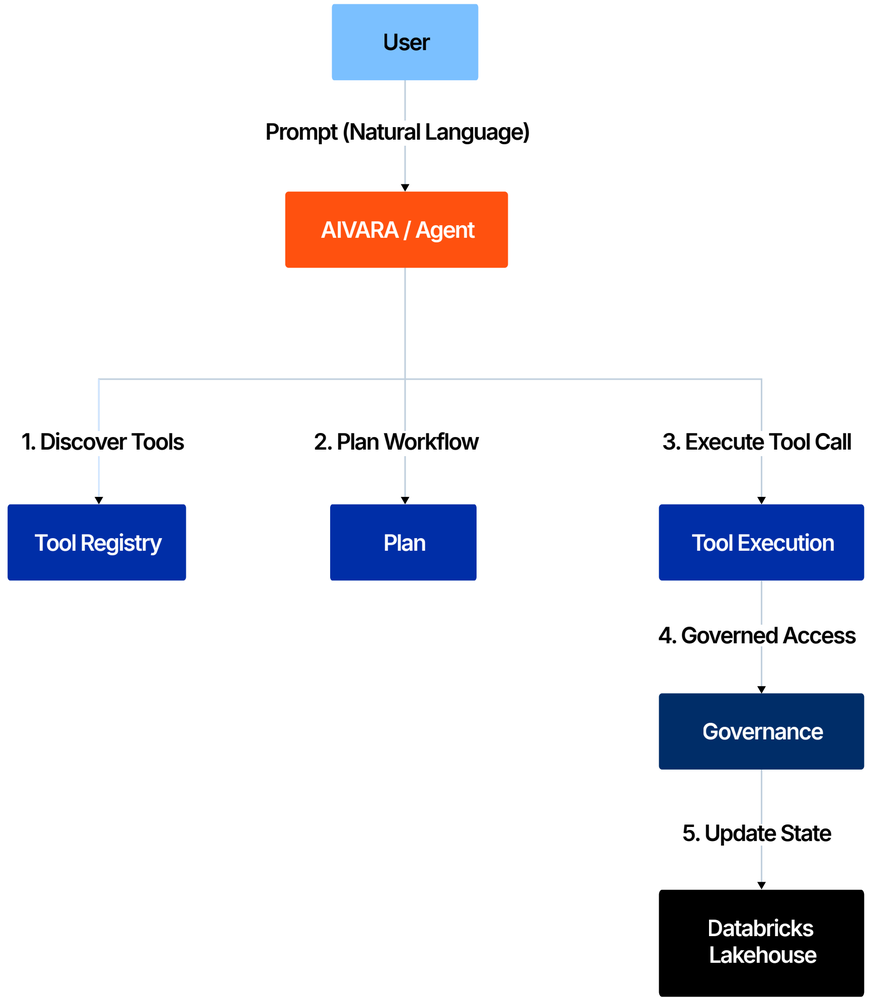
Interface-Driven Composition

Beneath the tools lies the architectural foundation that makes the entire system composable: a rigorous interface layer.

Auraa's shared foundation library, AURA_CORE, defines over thirty-five abstract interfaces, IMessageBus, IGrantManager, IToolRegistry, ICatalogManager, ITenantRepository, IGroupManager, and many others, that form the contract surface between components. This is hexagonal architecture applied to agent-tool interaction. Ports (abstract interfaces) define what the platform can do. Adapters (concrete implementations) define how it does it: Delta tables for messaging, Spark SQL for grant execution, Unity Catalog for governance enforcement.

Agents and tools program against ports. They never reference adapters directly.

The practical consequence is profound. Any component can be replaced, extended, or tested in isolation without affecting anything else. The production DeltaMessageBus adapter, which publishes events to Delta tables via gRPC streaming or in-memory buffering, is substituted with an InMemoryMessageBus during testing, and no consuming code changes. The SparkGrantManager could be swapped for a different grant execution engine, and no agent would notice. A new ingestion engine can be added behind the existing IIngestionEngine interface, and every workflow that uses ingestion gets access to it automatically.



This is not an accident of good engineering. It is the fundamental requirement for a platform that agents can safely operate. Agents must be able to reason about capabilities, what the platform can do, without needing to understand infrastructure, how it does it. The interface layer makes that possible.

Lakehouse-Native

Auraa is built on and for the Databricks Lakehouse. This is not a deployment choice; it is an architectural commitment that shapes every component.

Data flows through a medallion architecture. Bronze tables hold raw, append-only data written by DataDuct, with Change Data Feed (CDF) enabled for downstream incremental processing. Silver tables hold cleansed, deduplicated, and standardized data, curated by AICURA's quality rules and shaped by AITERA's Unified Data Model transformations. Gold tables hold purpose-built outputs, risk scores, aggregates, data products, ready for consumption by applications, dashboards, and external systems.

All metadata, configuration, and registry state is stored in Delta tables governed by Unity Catalog. Incremental processing is implemented via CDF and checkpoint tables, so jobs process only newly changed data. The lakehouse is not just where data lives. It is where the platform itself lives, its configuration, its event stream, its governance state, and its operational history.

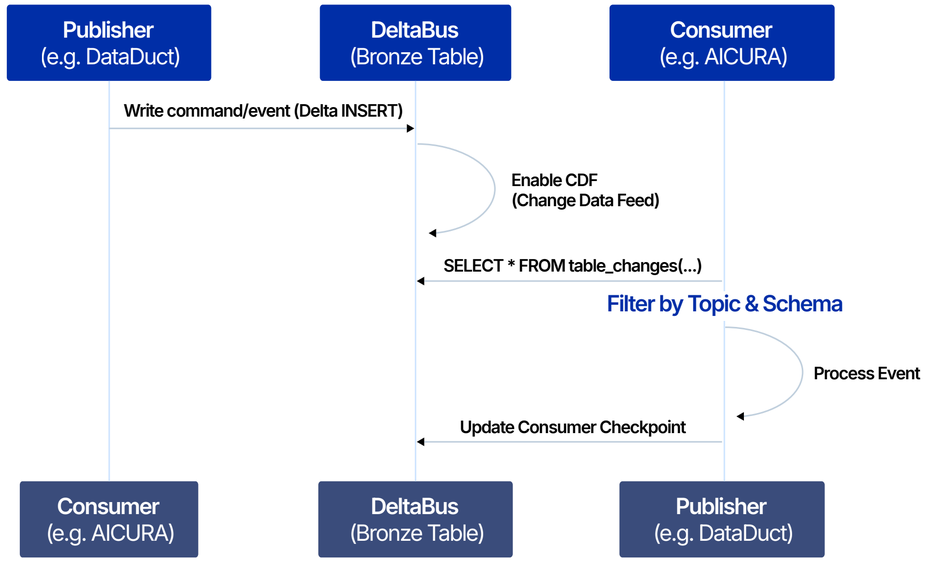
Event-Driven by Default

Components in Auraa do not call each other directly. They communicate through **DeltaBus**, a lakehouse-native event bus built entirely on Delta Lake tables with Change Data Feed.

This is a deliberate architectural choice. Direct inter-component calls create tight coupling, cascading failure paths, and synchronous dependencies that are difficult for agents to reason about. Event-driven communication decouples components so that each can operate, fail, and recover independently.

DeltaBus eliminates external message queue infrastructure, Kafka, SQS, Azure Service Bus, while providing the capabilities that platform operations require: at-least-once delivery with checkpoint-based deduplication, dead-letter queues for failed messages, permanent and SQL-queryable event history, and native Unity Catalog governance over the event stream. The estimated steady-state cost is approximately $50 per month in Delta storage, compared to $2,000–15,000 per month for managed message queue services at comparable event volumes.

When a tenant is provisioned, the API publishes a command to DeltaBus and returns immediately. A consumer processes the provisioning workflow asynchronously, updating a durable progress table at each phase. When an ingestion specification is ready, a DeltaBus event triggers the ingestion consumer. When provisioning completes or fails, downstream consumers are notified through completion events. Each consumer maintains its own checkpoint and failure handling. No component needs to know which other components are listening.



The full technical design of DeltaBus is described in the companion whitepaper *DeltaBus: A Lakehouse-Native Event Bus Pattern for Data Platforms.*

Multi-Tenant by Design

Auraa is built for multi-tenant operation from its foundation, not as a feature added after initial deployment.

Each tenant operates in a dedicated Unity Catalog catalog (aura_tenant_<id>), provisioned automatically by INFRAIA. This catalog-per-tenant design provides strong isolation at the storage and governance layer. Identity and access management is handled by AIGENE, which manages principals, tenant role bindings, and platform-wide groups.

Auraa employs a **Strict Registry-Driven Governance** model. Grant policies, group definitions, and volume configurations are stored as versioned registry tables seeded from declarative configuration files. There are no hardcoded fallbacks or "default" permissions hidden in the code. The platform operates on a **Fail-Fast** principle: if a required registry entry for a group or permission is missing, the system raises a runtime error rather than falling back to an insecure or unpredictable state. This ensures that agents always operate within a deterministic, governed boundary.

How This Compares

The following table summarizes the architectural differences between a legacy platform augmented with AI and a platform built from the ground up for agent-driven orchestration.

Dimension	Legacy Platform + AI Copilot	Auraa (Agent-Native)
Primary consumer	Human operator via UI or notebook	Agent via typed tool invocation
AI role	Assistant that suggests; human executes	Agent that plans and executes; human governs
Capability surface	UI actions, SQL scripts, notebooks	Typed tools with machine-readable contracts
Discovery	Documentation, tribal knowledge	Runtime registry (NEXARA)
Composition	Manual workflow assembly	Agent-driven tool composition
Governancet	Applied after the fact	Built in from the first line of code
Communication	Direct calls, shared databases	Event-driven (DeltaBus)
Testability	End-to-end only	Per-tool unit testing against interfaces
Extensibility	Modify existing code	Register a new tool; agents discover it automatically

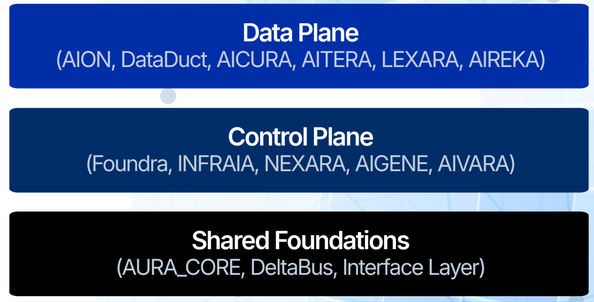
The differences are not incremental. They are structural. Retrofitting these properties into an existing platform would require rewriting the platform, which is exactly what Auraa is.

Architecture

Auraa's architecture is organized into three layers: the **Data Plane**, which processes data through the medallion architecture; the **Control Plane**, which manages provisioning, governance, orchestration, and platform lifecycle; and **Shared Foundations**, which provide the interface contracts, messaging infrastructure, and utilities that all components depend on. This section provides an overview of each layer and how they interact.

Data Plane

The Data Plane comprises the components that discover, ingest, curate, transform, score, and explain data as it moves through the lakehouse.



AION handles metadata discovery and ingestion design. It connects to source systems, SQL databases, SAP, file systems, Unity Catalog volumes, discovers schemas, tables, and relationships, and generates structured ingestion manifests. AION does not move data. It designs how data should be moved, creating a clean separation between design and execution that enables agents to reason about ingestion strategy without triggering data movement.

DataDuct is the metadata-driven ingestion engine that operationalizes AION's designs. Its Python class-based engines (FileEngine, JDBCEngine, SAPEngine) read AION manifests and execute ingestion into Bronze Delta tables, handling schema evolution, error recovery, lineage tracking, and batch observability. DataDuct publishes DeltaBus events on ingestion start, completion, and failure, enabling downstream components to react without polling.

AICURA provides data quality, curation, and completeness management. It profiles tables to understand statistical properties, proposes quality rules using heuristics and LLM-assisted generation, detects drift across successive profiles, and manages the full rule lifecycle from proposal through approval to active enforcement and eventual retirement. AICURA generates engine-specific quality specifications, for DataDuct, Lakeflow Declarative Pipelines, dbt, and other engines, through adapters, so the same quality rules can be enforced regardless of execution engine.

AITERA manages the Unified Data Model (UDM) and the transformations that produce it. It defines canonical entity models, invoices, vendors, payments, customers, general ledger entries, and performs the denormalization, consolidation, and derived-attribute computation needed to produce a clean semantic layer in Silver. Downstream components and applications consume UDM entities without needing to understand the source system's native structure.

LEXARA will provide centralized rule management and scoring. Its domain model, policies and policy sets, is defined, and it will evaluate business rules, fraud patterns, and key risk indicators over UDM entities, producing transaction-level and entity-level scores with configurable weights and thresholds. LEXARA is in active development; Kona AI Databricks currently implements scoring logic directly and will migrate to LEXARA as the component matures.

AIREKA provides information retrieval and search capabilities, including vector search for similarity-based retrieval and documentation discovery services. Its planned expansion includes entity-level explainability, generating structured narratives that explain why a specific invoice, vendor, or payment has been flagged, and portfolio-level summaries that surface trends and emerging patterns with data quality context from AICURA.

Each Data Plane component reads and writes managed Delta tables under tenant catalogs provisioned by INFRAIA. Each exposes its capabilities as registered tools discoverable through NEXARA.

Control Plane

The Control Plane manages the operational infrastructure that makes the Data Plane possible: provisioning, governance, orchestration, and the platform's own lifecycle.

INFRAIA is the infrastructure and catalog provisioning engine. It creates per-tenant Unity Catalog catalogs and standard schemas, manages volumes and environment-specific configuration, applies grants from data-driven registry tables, and generates Databricks Asset Bundle scaffolding. INFRAIA makes zero-touch tenant onboarding possible, a capability that becomes critical in multi-tenant deployments where manual setup would create an operational bottleneck.

Foundra is the platform's control plane and operational backbone. It orchestrates platform initialization across seventy discrete phases, catalog creation, schema setup, group provisioning, grant application, service principal configuration, registry seeding, and manages the async tenant onboarding workflow powered by DeltaBus. When a client provisioning request arrives, Foundra publishes a command, returns a task identifier immediately, and a background consumer processes the full onboarding sequence, updating a durable provisioning task table at each phase boundary. Foundra also manages the data-driven registries, grant policies, group definitions, volume definitions, tenant roles, that allow the platform's permission model to evolve through data changes rather than code deployments.

NEXARA is the agent and tool registry, the platform's capability directory. It catalogs tools from all Auraa components with metadata, input/output schemas, tags, and version information. It catalogs agents and their relationships to tools. It syncs Unity Catalog functions and models as first-class tools and agents. And it serves runtime discovery queries, so an agent can ask "find ingestion tools for SAP sources" and receive a structured answer without hardcoded knowledge of the platform's internals. NEXARA is what makes agent-tool composition dynamic rather than hard-wired. When a new tool is registered, every agent can discover it. No code changes. No redeployment.

AIGENE manages identity, access management, and governance. It handles principals and tenant entities, tenant role bindings, JWT-based authentication, and cross-tenant service authorization. On the governance side, it provides policy-as-code evaluation for both design-time checks (can this agent use this tool on this tenant's data?) and runtime authorization (is this principal permitted to perform this operation?). AIGENE integrates directly with Unity Catalog for table, column, and row-level permissions and masking, and with the grant policy engine for multi-tenant catalog resolution. AIGENE ensures that governance participates in every tool invocation, not as a gate that humans remember to check, but as a structural property of how tools execute.

AIVARA is the agent-facing operational interface to the platform. It serves as the primary interpreter of user intent, converting natural-language prompts into structured pipeline plans. AIVARA discovers and composes tools via NEXARA, validates plans via AIGENE, and orchestrates the end-to-end execution across the platform. Its scope includes operational troubleshooting, connectivity validation, and autonomous remediation. The critical enabler of AIVARA's intelligence is not just the underlying LLM, but the tool-first architecture of the platform itself, which provides a predictable, governed surface for agentic action.

Shared Foundations

The shared foundations layer provides the connective tissue that gives the platform its structural integrity.

AURA_CORE is the foundational library that all components depend on. At its center is the interface layer: over thirty-five abstract interfaces that define the contract surface between every component in the system. IMessageBus defines how components communicate. IGrantManager defines how permissions are applied. IToolRegistry defines how tools are discovered. ICatalogManager, ITenantRepository, IGroupManager, IMetadataRepository, ISourceRegistry, IJwtProvider, each defines a specific capability boundary that can be implemented, tested, and replaced independently.

AURA_CORE also contains the production **DeltaMessageBus** implementation: dual-mode publishing (ZeroBus SDK for low-latency confirmed delivery, buffered writes as an always-available fallback), CDF-based consumption with checkpoint tracking, dead-letter management, and the DeltaProvisioningTaskStore for durable progress tracking of long-running operations. For testing, it provides InMemoryMessageBus and InMemoryCheckpointStore, test adapters that allow any component to be tested in complete isolation from infrastructure.

Shared utilities round out the library: logging and configuration management, context models for tenant, user, and execution state, Spark and Delta utilities for safe writes, CDF helpers, and checkpoint patterns, and abstract base classes for tools and engines that enforce consistent behavior across the platform.

CI/CD patterns are standardized across all components: PyTest with coverage thresholds at seventy percent or higher, linting, and Declarative Automation Bundles for packaging and deployment. Every component follows the same build, test, and deploy lifecycle.

Component Detail

This section provides a closer look at each component's responsibilities, current capabilities, and relationship to the platform's agent-first architecture.

INFRAIA, Infrastructure and Catalog Provisioning

INFRAIA is the engine that turns a bare Databricks workspace into a governed, multi-tenant data platform.

Its responsibilities span the full infrastructure lifecycle. It creates per-tenant Unity Catalog catalogs with a standard schema structure (bronze, silver, udm, gold, dq, logs), provisions and manages Unity Catalog volumes, applies grants and permissions from data-driven registry tables, and generates Databricks Asset Bundle scaffolding and job templates. It also creates and maintains the shared aura_system catalog that holds platform metadata, registry tables, and cross-tenant configuration.

The key design principle behind INFRAIA is that infrastructure provisioning should be a tool call, not a manual procedure. An agent, or the async onboarding consumer, can provision a complete tenant environment by invoking INFRAIA tools in sequence: create catalog, create schemas, configure volumes, apply grants, seed registries. No tickets. No runbooks. No human intervention.

This capability is what makes Auraa's multi-tenant model operationally viable. Without it, onboarding a new tenant would require the same manual setup that makes traditional platforms slow and error-prone.

Foundra, Platform Control Plane

If INFRAIA provides the tools for infrastructure provisioning, Foundra is the orchestrator that decides when and how to use them.

Foundra manages the full lifecycle of an Auraa installation. Platform initialization spans seventy discrete phases, from catalog creation through schema setup, group provisioning, grant application, service principal configuration, and registry seeding. Each phase is logged, auditable, and independently recoverable if a failure occurs partway through.

Tenant onboarding is fully asynchronous. When a provisioning request arrives via the API, Foundra publishes a command to DeltaBus and returns a task identifier immediately. The OnboardingTaskHandler, a DeltaBus consumer running in the background, processes the full onboarding sequence, updating a durable provisioning_tasks Delta table at each phase boundary. Clients poll this table for progress. If the process crashes mid-provisioning, it resumes from the last completed phase on restart. No work is lost. No manual intervention is required.

Foundra also manages the data-driven registries that make Auraa's governance model flexible and predictable:

- **grant_policies**, Declarative grant definitions with scope filtering for platform, tenant, and project levels.
- **group_definitions,** Platform and tenant-scoped group specifications.
- **volume_definitions,** Storage volume configurations per tenant.
- **tenant_roles,** Role definitions for tenant-scoped access control.

By externalizing these definitions into registry tables, Auraa eliminates the risks associated with hardcoded identity logic. The platform's **Registry-Only Resolution** ensures that every permission check is a lookup against a governed record. If a registration is missing, the system fails explicitly and immediately, providing clear feedback to both human operators and agents. Governance is not just "applied" to the system; it is the system's operating system.

AION, Metadata Discovery and Ingestion Design

AION occupies a distinctive position in the architecture: it sits between the source systems that hold data and the ingestion engines that move it, designing the bridge without crossing it.

AION connects to SQL databases, SAP systems, file stores, and Unity Catalog volumes to discover schemas, tables, columns, relationships, and constraints. It stores this metadata in CDF-enabled Delta tables, creating a persistent, versioned record of what exists in each source system. From this metadata, it generates structured ingestion manifests, machine-readable specifications that describe exactly which tables to ingest, how to handle schema evolution, which columns to include, and what change-detection strategy to use.

These manifests are consumed by DataDuct (for Python-based ingestion) or Lakeflow Declarative Pipelines (for Databricks-native ingestion). AION also detects and records schema drift events, so that changes to source systems are captured and surfaced before they cause downstream failures.

Every AION capability is exposed as a tool registered in NEXARA. An agent can ask AION to discover a source, generate a manifest, or check for drift, all through typed tool calls with structured outputs.

DataDuct, Metadata-Driven Ingestion Engine

DataDuct is the operational counterpart to AION's design focus. Where AION decides what to ingest and how, DataDuct executes the plan.

Its engine architecture is extensible by design. Each source type is handled by a dedicated Python class, FileEngine for flat files, JDBCEngine for relational databases, SAPEngine for SAP systems, that implements a common ingestion interface. Engines read AION manifests and write to Bronze Delta tables with full support for batch processing (multiple datasets ingested concurrently with batch identifiers), schema evolution, error handling, retries, and lineage tracking.

DataDuct integrates with AICURA for inline data quality checks during ingestion. It publishes DeltaBus events at key lifecycle points, ingestion started, completed, failed, so that downstream components can react without polling or scheduling dependencies. A quality check that fails during ingestion produces an event that AICURA can consume and act on; a successful ingestion produces an event that AITERA can consume to trigger transformation.

This event-driven model eliminates the brittle scheduling dependencies that plague traditional ETL pipelines. Components react to what has happened, not to what a scheduler assumes will have happened by a certain time.

AICURA, Data Quality, Curation, and Completeness

Data quality in traditional platforms is typically one of two things: an afterthought (applied inconsistently, if at all) or a bottleneck (a separate team reviews every pipeline before it goes to production). AICURA offers a third option: a unified, cross-table quality framework that participates continuously in the data lifecycle.

AICURA profiles tables to understand their statistical properties, row counts, null rates, cardinality distributions, value ranges, and string patterns. From these profiles, it proposes quality rules using statistical heuristics and, increasingly, LLM-assisted generation. A column that is never null gets a NOT_NULL rule. A column with values between 0 and 100 gets a RANGE rule. A column matching a pattern like an email address or phone number gets a PATTERN rule. These proposals are not automatically enforced; they enter a lifecycle, PROPOSED, APPROVED, ACTIVE, DEPRECATED, ARCHIVED, that allows domain experts to review and refine before activation.

Once active, rules are enforced as data moves through the pipeline. AICURA generates engine-specific quality specifications through adapters: DataDuct ingestion specs, Lakeflow Declarative Pipeline expectations, dbt tests, and others. The same logical rule, "this column must not be null", is expressed in the syntax that each engine understands, applied consistently regardless of which engine processes the data.

AICURA also monitors for drift. When successive profiles show meaningful changes, a null rate that jumps from 2% to 15%, a cardinality that doubles, a value range that shifts, AICURA flags the drift and can trigger review or alerting workflows. This catches data quality degradation before it propagates to downstream consumers.

AITERA, Unified Data Model and Transformations

Raw data arrives in many shapes. Source systems have their own naming conventions, normalization strategies, and business logic embedded in table structures. AITERA's job is to impose order: transforming the heterogeneous data in Silver into a consistent, well-documented Unified Data Model that downstream components and applications can rely on.

AITERA defines canonical entity models, invoices, vendors, payments, customers, general ledger entries, and performs the denormalization, consolidation, and derived-attribute computation needed to produce them. It supports both full-refresh and incremental patterns using CDF and MERGE, so that UDM entities stay current without requiring expensive full-table recomputation.

The UDM is stored in tenant-specific udm schemas. Downstream components, LEXARA for scoring, AIREKA for search and explanation, and applications like Kona AI, consume UDM entities through a clean, documented interface. They do not need to understand how SAP represents a vendor differently from how Oracle does. AITERA has already resolved those differences.

This semantic layer is what makes cross-domain reuse possible. A fraud detection application and a compliance monitoring application can share the same UDM entities, invoices, vendors, payments, even though they apply different rules and produce different outputs.

LEXARA, Rules and Scoring (In Development)

Every analytical application eventually needs to apply rules to data and produce scores. Fraud detection needs risk indicators. Compliance monitoring needs policy violation checks. Financial analytics needs threshold-based alerts. Today, this logic is typically embedded in application code, duplicated across projects, and difficult to audit or modify.

LEXARA is designed to centralize this capability. Its domain model defines policies and policy sets that can express fraud patterns, key risk indicators, business controls, and threshold-based scoring logic. Rules are evaluated over UDM entities and data quality signals. Scores are computed with configurable weights, thresholds, and escalation logic, and aggregated to entities, vendors, customers, GL accounts, over time.

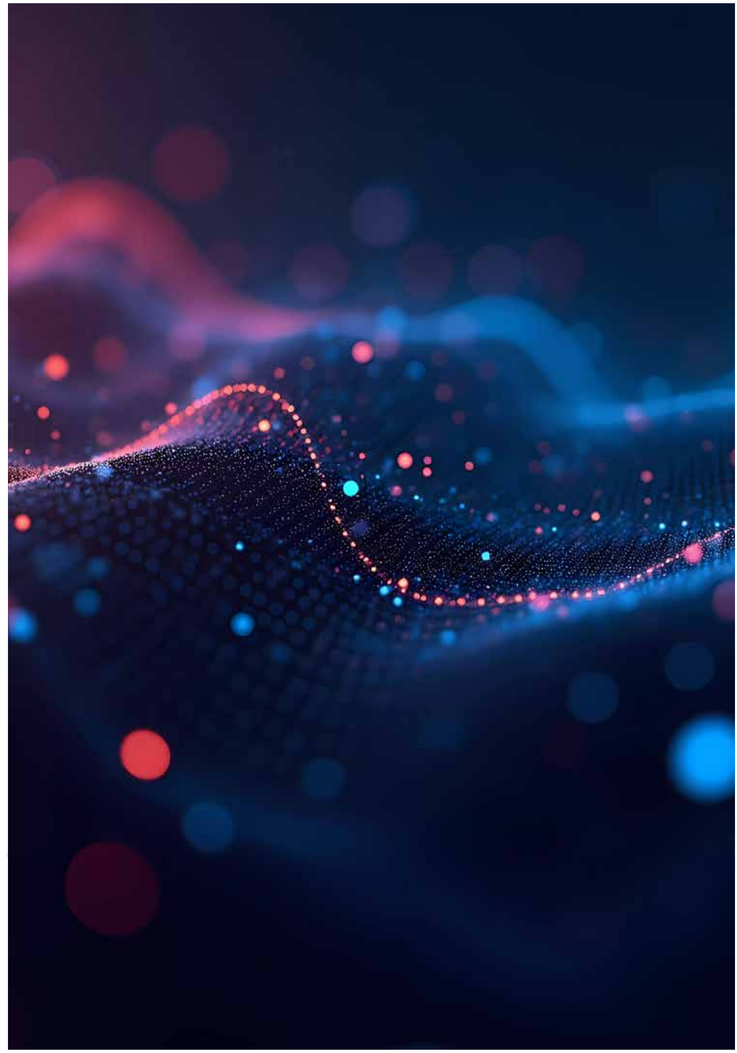
LEXARA is currently in development. Its domain model is defined, and the rule evaluation engine is on the roadmap. Kona AI Databricks currently implements scoring logic directly within its application layer; as LEXARA matures, this logic will migrate to the platform, making it reusable across future applications.

AIREKA, Information Retrieval and Search

As the volume of data, documentation, and analytical output grows, the ability to find relevant information becomes a capability in its own right.

AIREKA provides vector search integration for similarity-based retrieval and documentation discovery services that index and surface relevant content across the platform. It manages features and feature sets that support search and retrieval workflows.

AIREKA's planned expansion will move into explainability: generating structured, human-readable narratives that explain why a specific entity has been flagged, what rules were triggered, which data quality signals are relevant, and how the score compares to historical patterns. This capability is particularly important in regulated domains, where auditors and compliance officers need to understand not just what was flagged but why.



AIGENE, Identity, Access Management, and Governance

Governance in Auraa is not a feature. It is a property of the system, woven into every tool invocation, every event, every provisioning step.

AIGENE is the component responsible for making this real. On the identity side, it manages principals and tenant entities, tenant role bindings, JWT-based authentication, and cross-tenant service authorization. It provides break-glass access for emergency scenarios and password reset workflows. On the governance side, it enforces policy-as-code at both design time (can this agent use this tool on this tenant's data?) and runtime (is this principal authorized for this operation?).

AIGENE integrates directly with Unity Catalog for table, column, and row-level permissions and masking. It works with the grant policy engine to resolve permissions across multi-tenant catalogs, using scope filtering to ensure that platform-level grants and tenant-level grants are applied correctly and do not interfere with each other.

The result is a governance model where adding a new tenant, modifying access policies, or onboarding a new role is a data operation, not a deployment. AIGENE ensures that the policies are enforced consistently, whether the consumer is a human user, an agent, or an automated background process.

NEXARA, Agent and Tool Registry

In a traditional platform, knowing what the platform can do requires reading documentation, asking colleagues, or exploring code. In Auraa, it requires a query.

NEXARA is the platform's capability directory. It maintains a registry of every tool across all Auraa components, storing metadata, input/output schemas, tags, and version information in Delta-backed registry tables. It maintains a parallel registry of agents and their relationships to tools. It syncs Unity Catalog functions and models as first-class entries, so that Databricks-native capabilities are discoverable alongside Auraa-native ones.

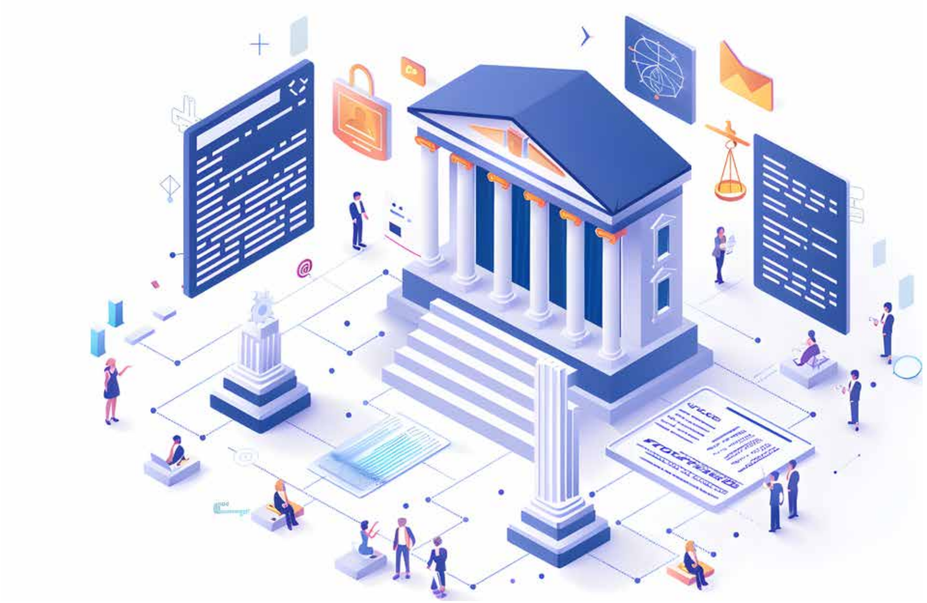
When an agent needs to find a tool, it queries NEXARA: "What tools are available for ingesting data from SAP?" "Which tools can apply data quality rules to a Bronze table?" "What agents are registered for operational troubleshooting?" NEXARA returns structured results that the agent can use to compose a workflow, no hardcoded tool lists, no static imports, no configuration files to update.

This is the mechanism that makes Auraa's extensibility practical. A new data quality adapter, a new ingestion engine, a new scoring provider, each is a tool registered in NEXARA. Every agent that queries the registry gains access to the new capability. The platform grows without anyone rewriting the agents that use it.

AIVARA, Agent-Driven Operations (Expanding)

AIVARA is where Auraa's agent-first architecture meets the user.

Today, AIVARA provides a focused set of capabilities. Its configure_pipeline tool accepts a natural-language description of what a user wants to achieve and produces a structured pipeline plan by discovering relevant tools and composing them into a workflow. Its approve_pipeline tool lets users review and approve generated plans before they execute. Its troubleshooting tools diagnose errors in running pipelines, validate connectivity to source systems, detect schema drift, and review quarantined records.



The vision for AIVARA is broader: an agent that can interpret high-level business intents ("build a fraud detection pipeline that reads from SAP and Oracle, applies quality checks, builds a UDM, and scores vendors against these risk indicators"), design the end-to-end architecture by discovering tools via NEXARA, validate the plan via AIGENE, and deploy the result as a Databricks Asset Bundle.

The important thing to understand about AIVARA's trajectory is that it is not the bottleneck. The bottleneck for agentic orchestration is always the platform underneath: are capabilities expressed as tools? Are they discoverable? Are they governed? Are they independently testable? Auraa answers yes to all of these. AIVARA's scope can expand incrementally, at the pace of the AI models that power it, because the platform is already ready.

AURA_CORE, Shared Foundational Library

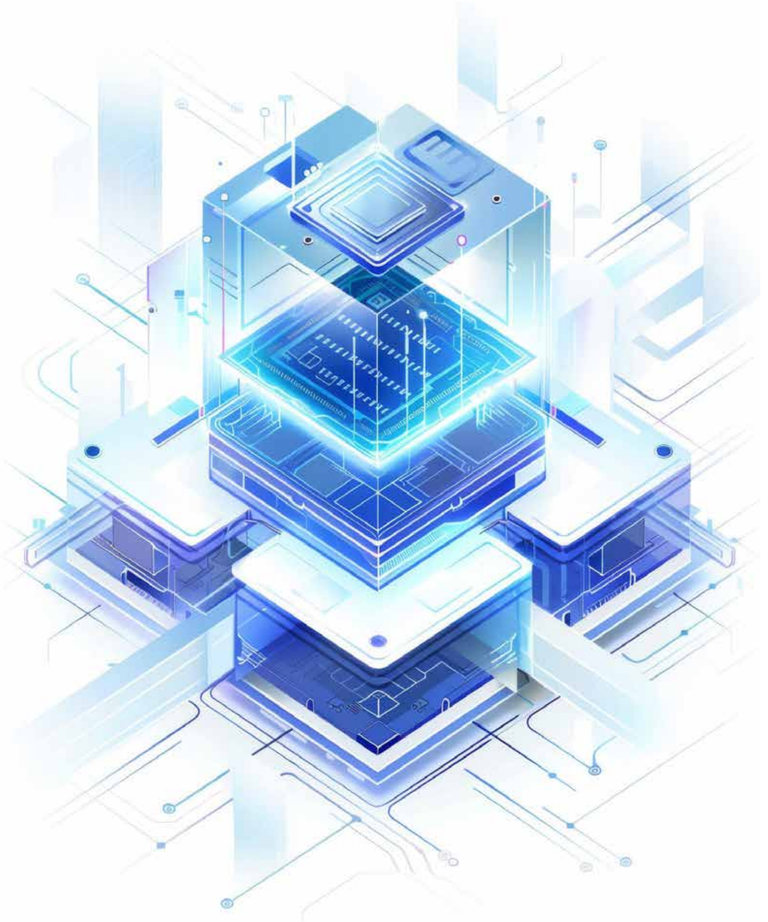
AURA_CORE is the component that no user sees and every component depends on. It deserves attention because the properties that make Auraa structurally different from traditional platforms, interface-driven composition, event-driven communication, durable progress tracking, isolated testability, all originate here.

The **interface layer** defines over thirty-five abstract ports: IMessageBus for event-driven communication, IGrantManager for permission application, IToolRegistry for capability discovery, ICatalogManager for infrastructure operations, ITenantRepository for identity, and many others. Every component programs against these interfaces. No component references a concrete implementation directly. This is the foundation of Auraa's testability and extensibility.

The **DeltaMessageBus** adapter implements the IMessageBus port using Delta Lake tables with Change Data Feed. It provides dual-mode publishing, ZeroBus SDK for confirmed low-latency delivery, buffered writes as an always-available fallback, CDF-based consumption with per-consumer checkpoint tracking, dead-letter management for failed messages, and SQL-queryable event history. The DeltaProvisioningTaskStore provides durable progress tracking for long-running operations like tenant onboarding, with each phase update persisted to a Delta table and queryable in real time.

The **testing infrastructure** provides in-memory adapters for every major interface, so components can be tested in complete isolation from Databricks, Delta Lake, and Spark. InMemoryMessageBus and InMemoryCheckpointStore allow messaging behavior to be tested synchronously in milliseconds. This is how Auraa maintains test coverage above seventy percent across a multi-component platform.

Shared utilities handle the cross-cutting concerns that every component needs: logging, configuration management, context models (tenant, user, execution state), Spark/Delta helpers (safe writes, CDF readers, checkpoint patterns), and abstract base classes for tools and engines that enforce consistent behavior.



What Makes Auraa Different

This section distills the architectural differences between Auraa and traditional platforms into five concrete shifts. These are not theoretical distinctions. They are observable differences in how the platform operates, how solutions are delivered, and how governance is enforced.

6.1 From Hand-Coded Pipelines to Tool-First Components

In a traditional stack, each project writes its own ingestion logic, quality checks, transformation scripts, and scoring routines. The result is a codebase where the same problems are solved differently, and usually incompletely, every time.

Auraa replaces this pattern with tool-first components. AION and DataDuct standardize ingestion. AICURA centralizes data quality. AITERA standardizes UDM construction. Each capability is a registered tool with a defined contract, so the investment in solving a problem once compounds across every future project. A new data source does not require a new ingestion pipeline; it requires a new AION manifest and a DataDuct engine invocation.

From Project-Centric to Platform-Centric

The traditional model treats each use case as an independent project with its own architecture, pipelines, and governance. Auraa separates platform from solution.

The platform team owns the components: INFRAIA, Foundra, AION, DataDuct, AICURA, AITERA, LEXARA, AIREKA, AIGENE, NEXARA, AIVARA, and AURA_CORE. Solution teams build domain-specific applications, fraud detection, compliance monitoring, risk analytics, by composing platform tools. Kona AI Databricks demonstrates this model: it is a solution built on Auraa, not a fork of it. The next solution inherits the same ingestion, quality, governance, and orchestration capabilities without rebuilding them.

From Manual Architecture to Agentic Orchestration

Traditional delivery depends on senior architects assembling pipelines by hand. Auraa enables agents to compose tools into workflows.

AIVARA interprets business intent, discovers tools via NEXARA, validates plans via AIGENE, and executes through deterministic tool calls. Human architects still define the platform's capabilities and governance policies. But the repetitive work of assembling capabilities into solutions, the work that takes months in traditional delivery, is increasingly automated. The architect's role shifts from building to governing, from assembling to reviewing.

From Bolt-On Governance to Governance by Design

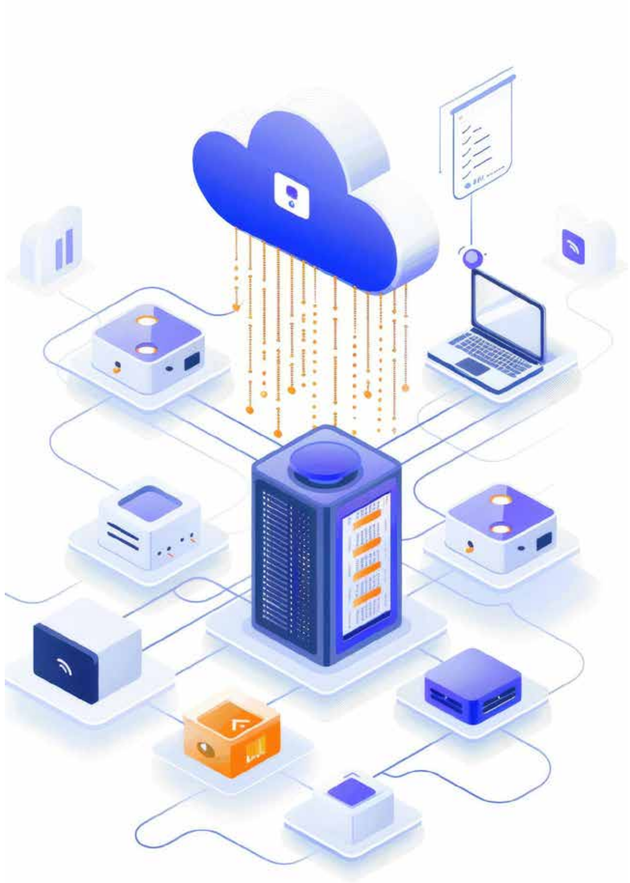
Governance in most environments is a final step: something applied after the pipeline is built, checked before the release goes out, audited after an incident occurs. In Auraa, governance is a structural property.

AIGENE participates at design time and runtime. Grant policies and role bindings are data-driven, stored in Delta registry tables, and applied consistently by tools, not manually configured by operators. Unity Catalog provides the enforcement layer for table, column, and row-level security. Every event published to DeltaBus is a permanent, queryable audit record. Governance is not something the platform does. It is something the platform is.

From Opaque Systems to Observable Platforms

Traditional platforms are difficult to observe. Pipeline failures surface as cryptic error messages. Provisioning status requires checking multiple systems. The relationship between a business outcome and the data operations that produced it is difficult to trace.

Auraa is observable by design. Every operation publishes events to DeltaBus, creating a permanent, SQL-queryable audit trail. Provisioning progress is tracked in durable Delta tables, visible in real time. Quality signals, drift events, and scoring outputs are all stored in governed tables and surfaceable through standard SQL queries. A custom DeltaSpanExporter writes OpenTelemetry traces to Delta for environments that lack native observability support. The platform is designed to be observed by both humans (SQL queries, dashboards) and agents (DeltaBus events, tool outputs).



Reference Application: Fraud Detection on Auraa

Kona AI Databricks, Covasant's fraud detection and risk analytics solution, is the first production application built on Auraa. It validates the platform's component model with a real, domain-specific workload and serves as a reference architecture for future applications.

The Kona AI workflow on Auraa follows the natural flow of data through the platform:

Infrastructure and Tenant Setup. Foundra orchestrates the full onboarding sequence asynchronously via DeltaBus. INFRAIA provisions the tenant catalog and standard schemas. AIGENE configures tenant-scoped principals, role bindings, and grants. The entire process runs without human intervention, with progress visible in real time through the provisioning task table.

Ingestion. AION discovers SAP FI, AP, and GL tables and generates structured ingestion manifests. DataDuct ingests into Bronze using CDF and checkpoints. DeltaBus events coordinate downstream processing.

Data Quality and Completeness. AICURA profiles Bronze tables and proposes quality and completeness rules. Rules are enforced as data is promoted to Silver, catching anomalies before they propagate.

Unified Data Model. AITERA builds UDM entities, udm.invoices, udm.vendors, udm.payments, udm.gl_entries, providing a consistent semantic layer for scoring.

Rules and Risk Scoring. Fraud patterns and key risk indicators are applied to UDM entities. Transaction-level and entity-level risk scores are computed and stored in Gold. This scoring logic currently lives in the application layer and will migrate to LEXARA as the component matures.

Outputs and Consumption. Scored entities are exposed via API services, Delta Sharing, or BI tools as Gold data products. The full audit trail of all platform operations is queryable via SQL against DeltaBus event history.

The same pattern, provision, ingest, curate, model, score, serve, applies to any analytical domain. The components are reusable. Only the domain-specific configuration changes: which source systems to connect to, which UDM entities to define, which rules to apply. The platform investment compounds with every application built on it.

Implementation Patterns

This section describes the core patterns that Auraa's components follow. These patterns ensure consistency across the platform and make it straightforward for teams to extend Auraa with new capabilities.

Tool Design Pattern

When implementing a new capability in Auraa, the process follows a consistent sequence.

First, define an abstract interface (port) in AURA_CORE that specifies the contract, what the capability does, what inputs it requires, what outputs it produces. Second, implement one or more concrete adapters behind that interface, each tailored to a specific engine or data source. Third, wrap the capability in tool functions with typed parameters and structured outputs. Fourth, register the tools in NEXARA with metadata, input/output schemas, and version information.

This sequence ensures that every new capability is immediately discoverable by agents, testable in isolation against the abstract interface, and governable by AIGENE. It also means that extending the platform is an additive operation: new tools are registered alongside existing ones, without modifying or risking existing workflows.

Incremental Processing Pattern

Across AION, DataDuct, AICURA, and AITERA, a consistent pattern governs how data changes flow through the platform.

Change Data Feed (CDF) provides table-level change detection. Checkpoint tables track the last-processed Delta version for each consumer. MERGE patterns in transformations update Silver and UDM tables incrementally, applying only the changes since the last run. This pattern minimizes recomputation, simplifies late-arriving data handling, and ensures that pipeline costs scale with data change volume rather than total data volume.

Multi-Tenant Pattern

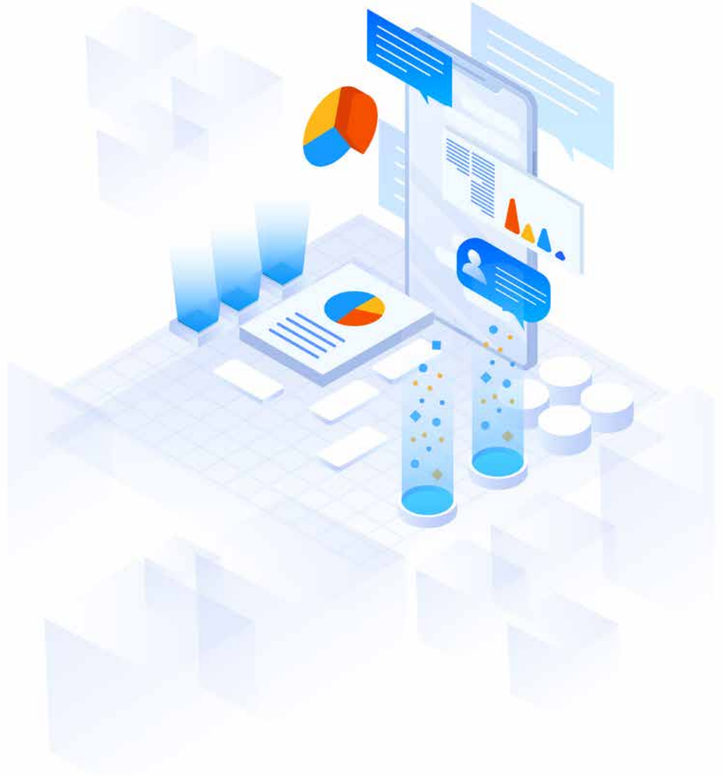
Multi-tenancy follows a catalog-per-tenant model. INFRAIA creates a dedicated catalog (aura_tenant_<tenant_id>) with a consistent set of schemas. Foundra registers the tenant in aura_system.tenants. AIGENE enforces that tools operate only within the active tenant's catalog, using scope-filtered grant policies from the data-driven registries.

This model provides strong isolation, each tenant's data is physically separated and independently governed, while allowing the platform to manage hundreds of tenants with the same codebase and the same operational model.

Event-Driven Coordination Pattern

Components publish events to DeltaBus topics rather than calling each other directly. commands.onboarding.start triggers async tenant provisioning. events.ingestion.spec.ready triggers ingestion execution. events.onboarding.completed and events.onboarding.failed notify downstream consumers of outcomes.

Each consumer maintains an independent checkpoint. Failures in one consumer do not cascade to others. Dead-lettered messages are preserved with full error context for inspection and replay. The complete event history is queryable by SQL, making operational debugging, compliance reporting, and performance analysis straightforward.



Operational Model

Build, Test, and Deploy

All Auraa components follow a standardized CI/CD lifecycle. Declarative Automation Bundles (DABs) package code and configuration for each component. GitHub Actions pipelines run lint checks, PyTest suites with seventy percent or higher coverage thresholds, and bundle validation. Wheel packages are versioned and deployed to Databricks volumes; bundle deployment references these packages.

This standardization means that a new component or a new version of an existing component follows the same build, test, and deploy path as every other component. There is no special-case deployment logic.

Monitoring and Observability

Observability in Auraa is built on the same Delta infrastructure that powers the platform itself.

All pipeline runs, ingestion, data quality, UDM transformation, scoring, are logged to tenant-specific logs schemas. DeltaBus event history provides a permanent, queryable audit trail across all platform operations. The provisioning_tasks table offers real-time visibility into long-running operations. Predictive Optimization and Lakehouse Monitoring are leveraged where available. A custom DeltaSpanExporter exports OpenTelemetry traces to Delta tables for environments, such as Databricks Apps, that lack native observability support.

Because all of this data lives in Delta tables, operational dashboards are SQL queries. No additional monitoring infrastructure is required.

Incident Response

When something goes wrong, the platform provides several mechanisms for rapid diagnosis. Standardized correlation identifiers link tool calls, events, and provisioning tasks into traceable chains. DeltaBus dead-letter tables preserve failed messages with full error context. NEXARA enables rapid identification of which tools and agents are involved in a failing workflow. And the event history table answers the most basic incident response question, "what happened, in what order, and what failed?", with a SQL query.

Risks and Mitigation

Architectural Complexity

Risk: A platform with twelve named components can overwhelm teams accustomed to monolithic ETL.

Mitigation: Start with a minimal slice. INFRAIA, Foundra, AION, DataDuct, and AICURA provide a complete ingestion-and-quality platform. Additional components are introduced as needs arise. Opinionated solution templates, modeled after Kona AI on Auraa, provide a clear path from "I have the platform" to "I have a working solution." AURA_CORE hides cross-cutting complexity behind stable interfaces, so teams interact with clean, well-documented tool contracts rather than the full component graph.

Agent Non-Determinism

Risk: LLM-driven planning introduces probabilistic behavior into a system that manages critical data.

Mitigation: The tool-first architecture is the primary mitigation. All business logic lives in deterministic tools. Agents focus on planning and tool selection, the probabilistic part of the workflow. AIGENE provides approval gates for deployment plans and sensitive operations. Every tool call is auditable and its effects are reversible or at least inspectable. The worst outcome of a bad plan is a failed tool call, not a corrupted dataset.

Performance and Cost

Risk: Complex rule evaluation and large-scale UDM joins can inflate Databricks compute costs.

Mitigation: CDF and incremental processing patterns minimize recomputation. INFRAIA cluster policies select appropriate compute (serverless, spot, or fixed) for each workload. DeltaBus eliminates external message queue infrastructure costs. Cost and performance are monitored at pipeline and rule-pack granularity, so hotspots are identifiable and addressable.

Ecosystem Integration

Risk: Customers with existing ETL, data quality, or analytical platforms may view Auraa as competing with their current investments.

Mitigation: Auraa's interface-driven architecture turns this concern into a strength. External tools can operate behind Auraa interfaces: AICURA can emit quality specs for external platforms, DataDuct can delegate to external ingestion systems, and LEXARA can integrate with external scoring providers. External capabilities can be registered in NEXARA and governed by AIGENE, effectively turning adjacent products into platform tools. Integration is a configuration exercise, not a rewrite.

Organizational Adoption

Risk: Moving from project-based delivery to a platform model requires cultural change.

Mitigation: Kona AI Databricks provides a concrete demonstration of platform value. A clear organizational boundary between the platform team (Auraa) and solution teams prevents confusion about ownership. Foundra, INFRAIA, and AIVARA provide onboarding tooling and templates that lower the barrier to building the first solution on the platform.

Replatforming Legacy Applications

Organizations with existing data and analytics solutions can adopt Auraa incrementally. The following stages describe a typical replatforming journey, using fraud detection as the worked example. The pattern applies broadly to any analytical domain.

Stage 1: Foundation.
Stand up INFRAIA, Foundra, AION, DataDuct, and minimal AICURA to replicate the existing ingestion and quality layer. This stage validates that Auraa can ingest the same data from the same sources and apply equivalent quality checks.

Stage 2: Curation and Modeling.
Implement AITERA transformations equivalent to the existing staging and mapping logic. This stage produces UDM entities that match the semantic model of the legacy system, establishing functional parity for downstream consumption.

Stage 3: Rules and Scoring.
Port existing business rules into LEXARA rule packs, or retain application-specific scoring until LEXARA matures. This stage is where the legacy system's core analytical logic moves to the platform.

Stage 4: Search and Explainability.
Introduce AIREKA for document search and, as capabilities expand, for entity-level explanation generation. This stage adds capabilities that the legacy system likely did not have.

Stage 5: Governance.
Onboard AIGENE for identity management and policy enforcement. Integrate with existing governance processes. Apply data-driven grant policies and role bindings.

Stage 6: Agentic Orchestration.
Gradually introduce AIVARA to plan and execute flows currently managed by human operators and scheduled jobs. This stage is where the platform begins to deliver the productivity gains that justify the migration investment.

At each stage, success should be measured in two dimensions: functional parity (does the new system produce equivalent results?) and platform improvement (faster delivery, better observability, stronger governance, lower operational cost).

The deeper insight is that replatforming on Auraa is not a lift-and-shift. It is a decomposition. A monolithic application is broken into tool-first components that become reusable across the next application, and the one after that. The cost of replatforming one application is partially amortized by the reduced cost of building every subsequent application on the same platform.

Roadmap

Auraa's roadmap reflects the dual priority of hardening what exists and expanding what is possible.

Near-term: Hardening and Production Readiness

- Expand AION and DataDuct with additional source connectors and improved error resilience for production workloads.
- Refine AICURA's rule lifecycle and LLM-assisted rule generation based on production feedback.
- Strengthen Foundra's onboarding automation and build operational dashboards for platform health monitoring.
- Extend DeltaBus adoption across additional platform workflows, moving toward fully event-driven coordination between all components.

Medium-term: Capability Expansion

- Deliver the **LEXARA Rules and Scoring Framework**, enabling application-independent business rules and risk indicator management.
- Expand **AIREKA Explainability Services** to provide natural-language narratives for scored entities and anomalies.
- Develop **AITERA Domain Templates** for automated Unified Data Modeling across standard ERP and CRM systems.
- Enhance **AIVARA Autonomous Solution Design**, allowing agents to provision complete analytical environments from high-level business requirements.

Long-term: Platform Maturity

- Evolve **NEXARA Marketplace**, enabling a secure exchange for shared tools and specialized agents across organizational boundaries.
- **Heterogeneous Governance Integration** with external platforms such as Microsoft Purview and Collibra.
- **Cross-Region Federation** for distributed data landscapes and multi-cloud resiliency.
- **Advanced DeltaBus Patterns**, including high-performance parallel consumer groups and native schema versioning within the event stream.

Because components are interface-driven, new engines, tools, and external integrations can be added without refactoring the platform. The architecture is designed to grow incrementally, not to require periodic rewrites.

Conclusion

The data platform industry is at an inflection point. The platforms that will define the next decade are not the ones adding AI assistants to architectures designed for human operators. They are the ones built from the ground up for a world where agents compose, execute, and govern data workflows, and humans set the strategy, define the policies, and audit the outcomes.

Auraa is that platform.

Every capability is a tool, typed, discoverable, testable, and governed. Components communicate through events, not direct calls. Governance is a structural property, not an afterthought. The architecture is interface-driven, so it grows without breaking. And the entire system runs on the Databricks Lakehouse, with no external infrastructure to manage, no separate message queues to operate, and no governance gaps between the data platform and the event stream.

Kona AI Databricks, the first application built on Auraa, demonstrates that this architecture delivers real solutions, not just elegant diagrams. The same ingestion, quality, transformation, and governance components that power fraud detection will power the next application, and the one after that, without starting from scratch.

Auraa was not built by adding AI to an old platform. It was built by asking a different question: what would a data platform look like if agents were the primary consumer from day one?

This paper is the answer.

Companion Documents

- DeltaBus: A Lakehouse-Native Event Bus Pattern for Data Platforms, Technical whitepaper on the lakehouse-native event-driven messaging infrastructure.
- DeltaBus Adoption Opportunities in the Auraa Platform, Analysis of eleven concrete adoption areas across the platform.

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

alan.dennis@Covasant.com