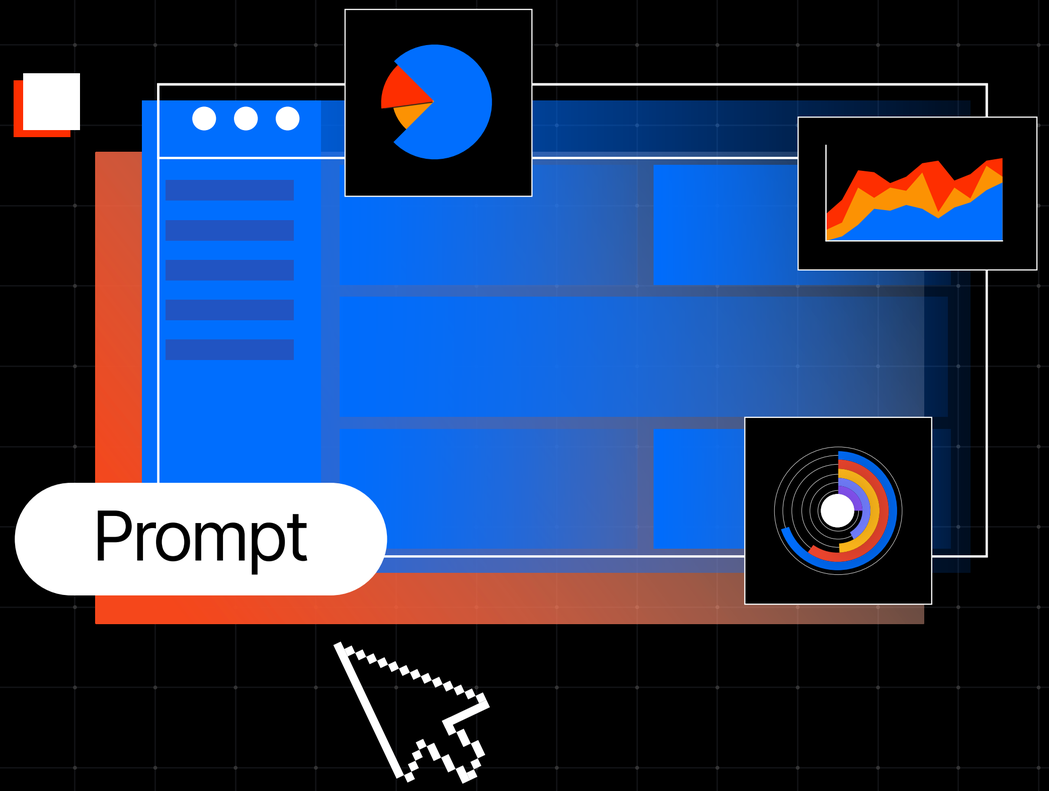
Covasant
AI-Driven. Human-Inspired.

Auraa Semantic Flow

Transient, Data-Driven Interfaces for Agent-Human Collaboration

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. The Auraa Semantic Flow pattern is a core primitive in Auraa's aura_core component. It enables any backend service, pipeline configuration, grant management, quality rule review, tenant provisioning, to present structured, time-bounded decision surfaces to human users without building custom frontend pages.



Alan Dennis

Databricks MVP,
VP - AI and Data Innovation,
Covasant

Executive Summary

AI agents in enterprise data platforms face a fundamental interaction design problem. They can produce sophisticated analyses, discovering 47 source tables, grouping them by ingestion frequency, proposing 28 quality rules with varying confidence levels, but they have only two ways to communicate this to humans: free-text prompts (expressive but unstructured, offering no guardrails or actionable controls) and buttons (structured but rigid, unable to convey nuance, context, or confidence).

Neither is adequate. A pipeline agent that discovers complex source metadata needs to say: "Here are 47 tables grouped into three frequency tiers based on query activity analysis. I'm 95% confident about the NOT NULL rules but only 65% confident about the legacy range checks. Review the groupings, you can drag tables between tiers, then approve or reject the plan. You have 30 minutes before this proposal expires."

That is not a prompt. That is not a button. That is an Auraa Semantic Flow .

This whitepaper introduces the Auraa Semantic Flow pattern, a JSON-serializable envelope that enables backend services to define rich, multi-step, time-bounded interaction workflows rendered identically across graphical and conversational interfaces. The pattern rests on three pillars:

- Rich agent vocabulary, A catalog of 14 domain-specific section types (grouped tables with drag-to-reassign, confidence-scored checkbox groups, real-time progress with polling, metric cards, timelines) that give agents a structured language for expressing complex proposals
- Semantic Surfaces, Interfaces that are born from a specific context, carry their own TTL, and die on completion or expiry. No permanent pages, no stale state, no orphaned routes. The next interaction for a different source produces a completely different screen
- Dual rendering, The same descriptor drives both a web UI (lazy-loaded React components) and a conversational agent (text-formatted), eliminating parallel presentation logic

The result: backend teams ship new agent-driven features without frontend work. The Auraa platform integrated grant template approvals, a multi-step wizard with policy preview, conflict detection, and confirmation, in under two hours using existing section types, compared to an estimated two days for a custom page.

The Interaction Bandwidth Problem

The Prompt-Button Spectrum

Current agent-human interfaces occupy two extremes:

	Free-text Prompts	Static Buttons
Expressiveness	High, natural language, arbitrary length	Low, fixed labels, binary choices
Structure	None, humans must parse prose	Complete, one action per button
Nuance	Possible but unreliable, "I'm somewhat confident"	Impossible, approve or reject, no middle ground
Editability	None, humans can only respond with more text	None, humans can only click or not
Context	Implicit, the human must remember prior turns	Implicit, the button doesn't explain why
Temporal bounds	None, the conversation persists indefinitely	None, the button persists indefinitely

Enterprise data engineering decisions, "should we ingest these 47 tables on this schedule with these quality rules?", fall in the gap between these extremes. They require structure (tables, groupings, rules) but also nuance (confidence scores, AI reasoning, editable defaults). They need temporal bounds (this proposal is based on current source metadata; it becomes stale) but also persistence across async review cycles (the approver may not be the person who triggered the analysis).

The Frontend Bottleneck

Traditional application architecture compounds the problem. Each new agent capability requires:

- A dedicated React page with routing
- TypeScript type definitions mirroring the backend response
- An API service client with fetch calls
- Custom state management for multi-step flows
- Progress polling infrastructure for async operations

This creates a frontend bottleneck: backend agents can discover metadata, propose rules, and generate pipeline specs in minutes, but the UI to review and approve these proposals takes days to build. The agent's capabilities are gated by frontend velocity.

The Transience Gap

Existing UI frameworks assume permanence. Routes, pages, and components are deployed and persist across sessions. But agent-human collaboration is inherently transient:

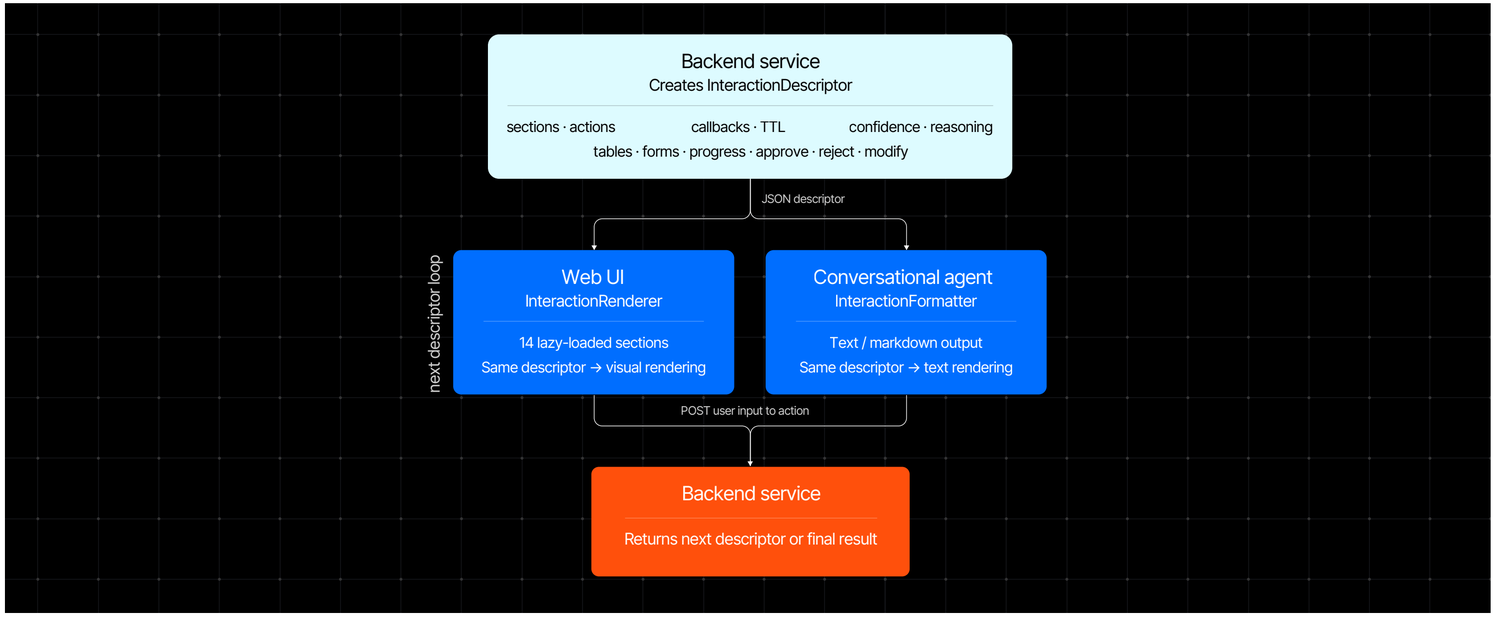
- An agent's proposal is contextual, "these rules for this data at this moment"
- The proposal has a shelf life, source metadata changes, credentials rotate, other users may act on the same resources
- The interaction is disposable, once approved or rejected, there is no reason for the UI to exist

Building permanent pages for transient interactions wastes engineering effort and creates maintenance burden. Worse, permanent pages become stale, showing outdated proposals with no indication that the underlying data has changed.

The InteractionDescriptor Pattern

Core Concept

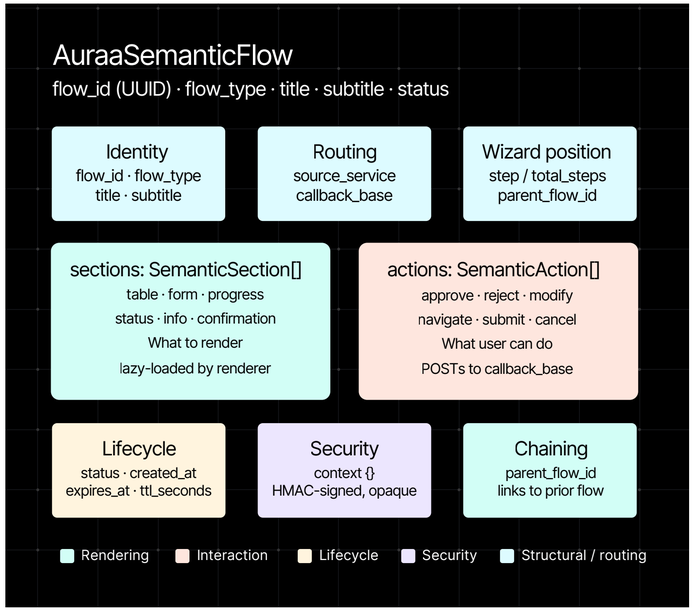
An InteractionDescriptor is a JSON-serializable data structure that completely describes a human-facing interaction: what to display, what the user can do, where to send input, and when the interaction expires. It is created by any backend service, stored transiently, and rendered by a generic frontend engine.



The critical distinction from form schema libraries (JSON Forms, React JSON Schema Form, Retool): an Auraa Semantic Flow is not a form. It is a workflow envelope that carries:

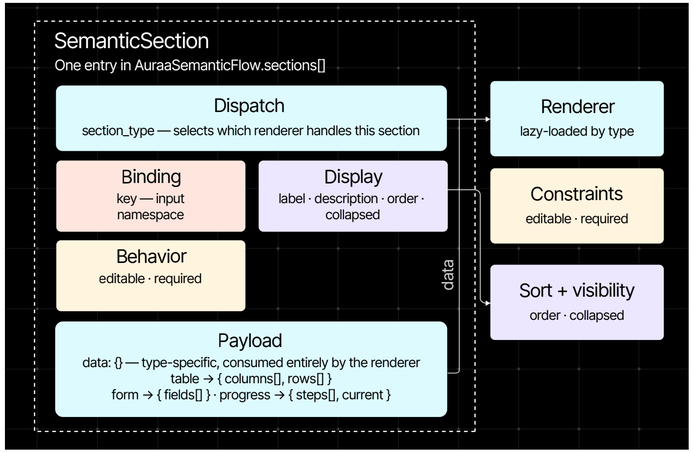
- **Lifecycle state,** active, processing, completed, expired, dismissed
- **Temporal bounds,** created_at, expires_at, ttl_seconds
- **Chain position,** parent_flow_id, step, total_steps
- **Signed context,** HMAC-protected opaque state that travels with the descriptor
- **Action protocol,** Typed responses: next (continue flow), complete (done), validation_error (retry)

Data Model



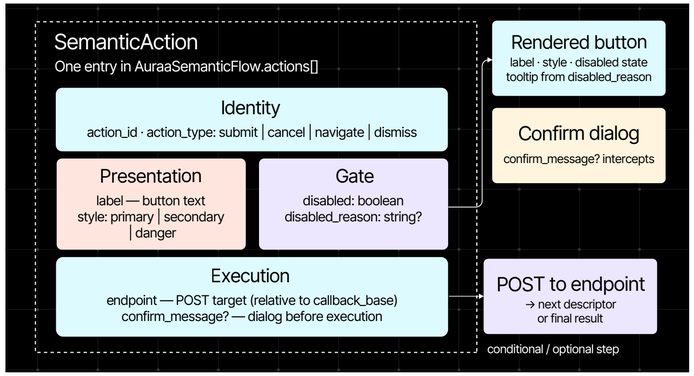
SemanticSection,

A typed, renderable content block:



SemanticAction,

A user-triggerable operation:



Section Type Catalog

The section catalog is the agent's vocabulary, each type encodes a specific interaction pattern:

Section Type	Purpose	Key Innovation
text_input	Free text with constraints	sensitive flag for masked input
select / multi_select	Choice from options	Agent pre-selects defaults
checkbox_group	Items with confidence	Confidence scores visible per item; auto-check threshold
table	Selectable data grid	Row-level selection for batch operations
grouped_table	Categorized rows	Drag-to-reassign between groups (e.g., frequency tiers)
key_value	Summary pairs	Read-only context display
progress	Real-time tracking	Built-in polling with exponential backoff; auto-advances to next step
markdown	Formatted text	Agent reasoning, documentation
confirmation	Approve/deny gate	Warning display for destructive actions
clarification	Follow-up questions	Agent asks for disambiguation
json_view	Raw data inspection	Collapsible for debugging
metric_cards	KPI badges	At-a-glance status (specs persisted, rules proposed)
timeline	Event stream	Real-time event synchronization from progress polling

The catalog is deliberately domain-specific rather than generic. grouped_table exists because pipeline frequency assignment is a real problem that requires drag-and-drop categorization. checkbox_group with confidence exists because AI rule proposal requires humans to see how confident the agent is, not just what it proposes. These are not general-purpose form widgets, they are interaction patterns extracted from real agent-human collaboration scenarios.

Transient Interaction Lifecycle

Why Transience Matters

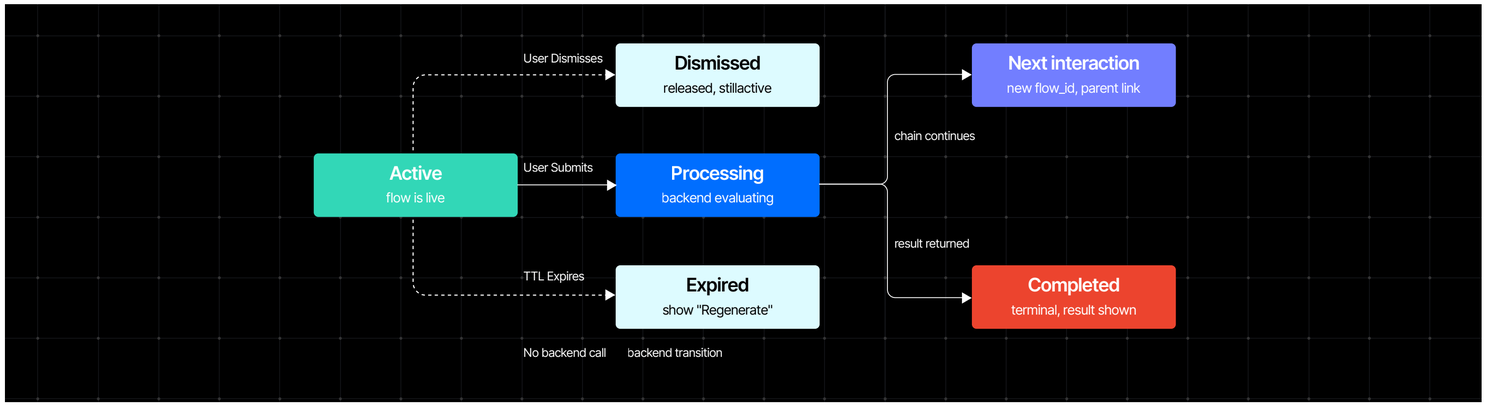
Traditional applications build permanent interfaces. A “Quality Rules” page exists whether there are 0 rules or 10,000. It shows the same columns, the same filters, the same actions, regardless of context.

An InteractionDescriptor creates an interface that exists only for a specific decision. When an agent proposes 28 quality rules for a newly discovered source:

- The interaction is born from that specific discovery context
- It carries a TTL (default 30 minutes), visible as a countdown badge
- It is tailored to exactly these rules, this source, this confidence distribution
- It dies on approval, rejection, or expiry
- It leaves no permanent artifact, the next discovery produces a new interaction

This mirrors how human collaboration actually works. When you ask a colleague to review a pull request, they don’t build a permanent review dashboard, they look at the specific diff, leave comments, approve or request changes, and move on. The Auraa Semantic Flow models that: here’s what I need you to look at, here’s what you can do, here’s how long you have.

Lifecycle States



All terminal states release resources. The in-memory store evicts the descriptor; the Delta store marks it as inactive. No cleanup jobs, no orphan detection, the TTL is the garbage collector.

Semantic Flow Store

Semantic flows are stored in one of two backends depending on durability requirements:

Store	Implementation	Use Case	TTL
In-memory	LRU cache with TTL eviction	Short-lived flows, single-session	30 min default
Delta table	{system_catalog}.governance.semantic_flows	Async approvals, audit trail	Configurable

The Delta store is the novel element. By persisting semantic flows to Delta Lake:

- Multi-instance durability, Databricks Apps can restart or scale horizontally without losing in-flight interactions
- Auditability, Every flow event (proposal, approval, rejection) is queryable via standard SQL
- No external infrastructure, No Redis, no Postgres, no SQS. The lakehouse is the semantic flow store

This aligns with the DeltaBus philosophy (see companion whitepaper): the lakehouse already provides durable, queryable, governed storage, using it for interaction state eliminates an entire infrastructure category.

Agent-Human Collaboration Patterns

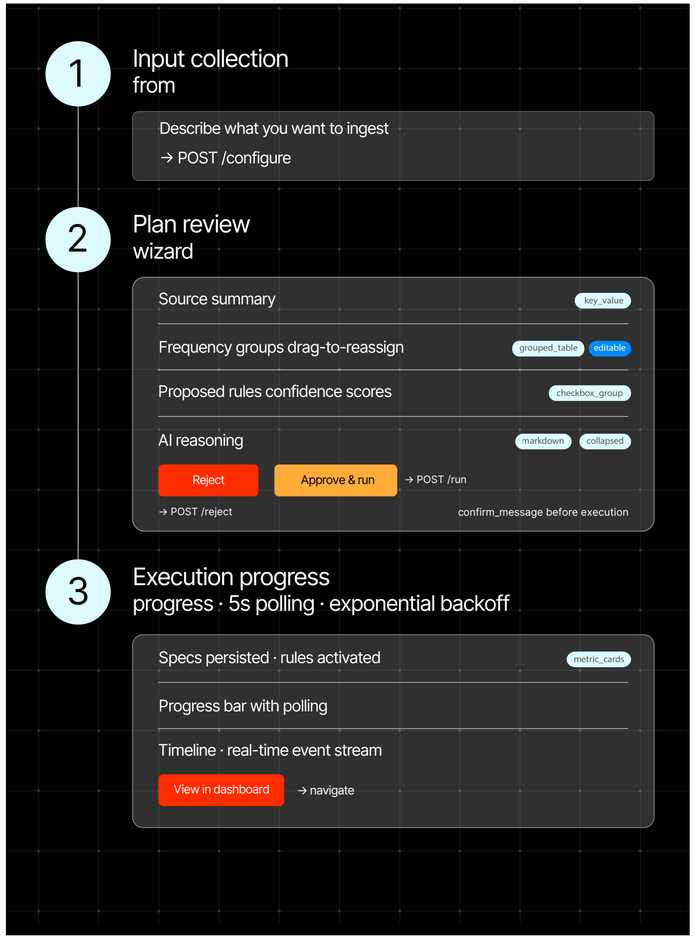
Confidence-Aware Proposals

When an AI agent proposes actions, certainty varies. A NOT NULL rule for an order_id column is near-certain; a range check for a legacy field is speculative. The checkbox_group section type makes this visible:

```
{
  "section_type": "checkbox_group",
  "key": "quality_rules",
  "label": "Proposed Quality Rules (28)",
  "data": {
    "items": [
      {"key": "r1", "label": "order_id NOT NULL", "checked": true, "confidence": 0.95},
      {"key": "r2", "label": "email pattern match", "checked": true, "confidence": 0.92},
      {"key": "r3", "label": "legacy_field range", "checked": false, "confidence": 0.41,
       "description": "Low confidence, field has irregular distribution"}
    ]
  }
}
```

Multi-Step Wizard Chains

Complex agent workflows are expressed as chains of descriptors linked by parent_flow_id:



Each step is a complete Auraa Semantic Flow. The action response protocol determines flow:

```
{ type: "next", flow: AuraaSemanticFlow }, Advance to next step
{ type: "complete", result: {...} }, Flow finished
{ type: "validation_error", result: {errors, message} }, Stay on current step
```

The frontend maintains no multi-step state. The backend owns flow logic. The frontend renders whatever descriptor it receives.

HMAC-Signed Context

Multi-step flows require state between steps. Traditional approaches use server-side sessions or client-side state management. Auraa Semantic Flows carry state in the context field, an opaque dict signed with HMAC-SHA256:

```
# Backend: sign context before sending
context = {"plan_id": "plan-uuid", "tenant_id": "acme", "approved_specs": [...]}
descriptor.context = sign_context(context, secret_key)

# Backend: verify on action submission
verified_context = verify_context(submitted_context, secret_key)
# Raises if tampered
```

This enables stateless action handlers. The handler receives the descriptor's signed context, verifies integrity, extracts state, and processes the action. No server-side sessions, no distributed cache, no sticky routing.

The context is opaque to the frontend, it passes it through without reading or modifying it. This provides a clean separation: the frontend handles rendering and input collection; the backend handles state, authorization, and business logic.

Approval Gates with Confirmation

For destructive or consequential actions, the confirm_message field on an action triggers a confirmation dialog before submission:

```
{
  "action_id": "approve",
  "label": "Approve & Run",
  "action_type": "submit",
  "endpoint": "/configure/plan-uuid/approve",
  "style": "primary",
  "confirm_message": "This will create 47 ingestion specs and start pipeline execution. Estimated processing time: 15 minutes."
}
```

The human sees the confirmation with the agent's impact assessment before committing. This is not a generic "Are you sure?", it is a context-specific warning generated by the agent based on its analysis.

Dual-Surface Rendering

The Divergence Problem

Enterprises increasingly offer both graphical and conversational interfaces to their platforms. A data engineer might use the web UI for routine monitoring but the chat agent for ad-hoc analysis. These interfaces typically diverge:

- The web team builds React components for visual rendering
- The agent team builds prompt templates for text rendering
- They drift apart, showing different data in different formats

Auraa Semantic Flows solve this by providing a **single source of truth** for what to present. The backend emits one descriptor; two renderers interpret it:

Visual Renderer (Web UI)

The web renderer uses a **lazy-loaded semantic section registry:**

```
const SECTION_REGISTRY: Record<SectionType, LazySectionComponent>
= {
  text_input: lazy(() => import('./sections/TextInputSection')),
  grouped_table: lazy(() => import('./sections/GroupedTableSection')),
  progress: lazy(() => import('./sections/ProgressSection')),
  // ... 14 components
};
```

Each section component is loaded only when needed, wrapped in an error boundary (one section crashing doesn't break the flow), and renders with Suspense fallbacks.

The **SemanticFlowShell** provides chrome: title, subtitle, step indicator, TTL countdown badge, validation errors, and the action button bar. The **SemanticFlowRenderer** dispatches each section to its registered component.

Three Rendering Surfaces

The web UI provides three surfaces for interaction rendering, selected by context:

Surface	When Used	Implementation
Dedicated page	Bookmarkable flows (pipeline config)	Route with SemanticFlowRenderer
Agent side panel	In-context agent workflows	Sheet drawer with inline rendering
Modal dialog	Transient confirmations from any page	SemanticFlowProvider portal

Any component can invoke useSemanticFlow().showFlow(descriptor) to present a flow as a modal, no routing, no navigation, no page reload. The flow appears, the human responds, and the modal disappears.

Real-World Application
Pipeline Configuration

The first production consumer of the Auraa Semantic Flow pattern is UC-314: Prompt-Based Pipeline Configuration. This flow demonstrates all three pillars.

The Scenario

A data engineer says: "Ingest all tables from our R&D SQL Server for the supply chain project."

The pipeline agent:

1. Discovers 47 tables via Aion metadata connectors
2. Profiles query activity to determine ingestion frequency
3. Groups tables into Daily (32), Weekly (12), Hourly/SLA (3) tiers
4. Proposes 28 quality rules with confidence scores
5. Generates DataDuct ingestion specs with watermark detection

Without Auraa Semantic Flow

This would require:

- A custom "Pipeline Review" page with routing
- TypeScript types for pipeline plans, frequency groups, proposed rules
- React components for frequency group tables with drag-and-drop
- A confidence-scored rule list with approve/reject per rule
- Progress polling for async pipeline execution
- State management for the 3-step wizard flow

Estimated effort: **2-3 days of frontend development.**

With Auraa Semantic Flow

The backend adapter (plan_to_flow()) transforms the PipelinePlan domain object into three AuraaSemanticFlow instances, one per step. The frontend renders them using existing section components. No new pages, no new types, no new state management.

The grouped_table section handles frequency tier review with drag-to-reassign. The checkbox_group section handles quality rules with confidence scores. The progress section handles async execution polling with exponential backoff.

Effort: **~2 hours**, writing the backend adapter function.

Second Consumer: Grant Template Approval

When the grant management service (UC-304) needed an approval workflow for grant templates, with policy preview tables, conflict detection warnings, and scope confirmation, it followed the same pattern:

1. Backend adapter: template_preview_to_flow()
2. Sections: key_value (template summary), table (policy list), table (conflicts), confirmation (scope warning)
3. Actions: Approve (apply grants) / Reject

No frontend changes. The existing SemanticFlowRenderer handled it. The grant adapter was the only new code.

Architecture Properties

Zero Frontend Code for New Features

The section catalog is fixed and generic. Backend services compose descriptors from existing section types. Adding a new agent capability, source registration, user role assignment, quality rule approval, requires only a backend adapter that transforms domain objects into AuraaSemanticFlow instances.

The frontend is a rendering engine, not a feature host. It doesn't know about pipelines, grants, or quality rules. It knows about tables, checkboxes, progress bars, and actions.

Stateless Action Handlers

HMAC-signed context eliminates server-side session management. Action handlers are pure functions: verify context, extract state, process action, return next descriptor or result. This enables horizontal scaling without sticky routing or distributed session stores.

Lakehouse-Native Persistence

The DeltaFlowStore persists flows to a governed Delta table. Benefits:

- **Durability,** Survives Databricks App restarts and replica scaling
- **Auditability,** SELECT * FROM semantic_flows WHERE status = 'completed' for compliance reporting
- **Time travel,** Delta's versioning provides natural interaction history
- **Governance,** Unity Catalog ACLs apply to flow data like any other table
- **Zero ops,** No Redis cluster, no Postgres replica, no SQS queue

Graceful Degradation

Each section is wrapped in an error boundary. If the frontend encounters a section_type it doesn't recognize (because the backend has been updated but the frontend hasn't), it renders a fallback ("Unsupported section type: X") instead of crashing the entire flow.

Similarly, if the progress polling endpoint is unavailable, the ProgressSection applies exponential backoff (up to 30s) rather than flooding the backend with retries.

Section Catalog as Agent Tool Schema

The section catalog described in §3.3 is not only a rendering vocabulary, it is also exposed as a **structured tool definition** (compose_flow) that the LLM agent can invoke at runtime. The tool's param_schema encodes all 14 section types, their data shapes, action types, and lifecycle fields as JSON Schema. This means the agent learns the full vocabulary through the same mechanism it learns any other tool: by reading the schema at routing time.

This closes a gap between §8.1 (zero frontend code for new features) and the agent's actual capabilities. Without tool-level access to the section catalog, new interaction patterns still require a hand-coded backend adapter, the agent can render descriptors but cannot compose them. With compose_flow, the agent can dynamically build ad-hoc interaction surfaces (data comparisons, confidence-scored approvals, multi-step wizards) from queried data without any new backend or frontend code.

Two composition modes are supported:

1. **Agent-internal composition,** The agent calls compose_flow as a function call within its reasoning loop, populating sections with real data queried during the conversation. The result is a descriptor embedded directly in the response. No user confirmation step is needed because the tool only presents an interaction surface, it does not execute business logic.
2. **Callback-wired composition,** The agent sets callback_tool_id to an existing registered tool (e.g., grant_catalog_access). The descriptor serves as a custom confirmation UI: when the user submits, the standard tool_action_handler dispatches to the callback tool with the collected input and HMAC-verified context. This lets the agent compose bespoke approval workflows in front of any platform operation.

The tool validates descriptors before signing: section types are checked against the catalog, keys must be unique, navigate action endpoints are grounded against the frontend route registry, and callback_tool_id is verified against the factory registry. Invalid descriptors are rejected with actionable error messages rather than rendered with missing or broken sections.

Comparison to Prior Art

	JSON Schema Forms	Workflow Engines (Temporal)	Approval Systems (PagerDuty)	Auraa Semantic Flow
Primary purpose	Form generation	Task orchestration	Alert routing	Agent-human collaboration
State model	Schema only	Durable workflow state	Alert lifecycle	Transient flow with TTL
Temporal bounds	None	Workflow timeouts	Alert TTL	Native TTL + countdown
Agent awareness	None	Worker model	None	Confidence scores, pre-filled values
Multi-step	Not native	Native	Not native	parent_flow_id
Real-time	None	Event-driven	Webhook	parent_flow_id + step Built-in polling + backoff
Rendering	Single (HTML form)	External	Email/SMS/App	Dual (web + chat)
Persistence	None (client-side)	Durable (database)	Database	**Delta Lake** (lakehouse-native)
Expiry	None	Timeout → fail	TTL → escalate	TTL → "Regenerate"

The closest analogue is Slack's Block Kit, a structured message format that supports interactive elements (buttons, selects, date pickers) within a chat interface. InteractionDescriptor extends this concept in three ways:

1. Multi-step wizard chains, Block Kit messages are stateless; Auraa Semantic Flows chain across steps with signed context
2. Domain-specific section types, Block Kit provides generic UI blocks; Auraa Semantic Flows provide grouped_table, checkbox_group with confidence, progress with polling
3. Dual rendering, Block Kit targets Slack only; Auraa Semantic Flows render on web and chat surfaces from a single definition

Limitations and Future Work

Current Limitations

- **Section catalog is fixed,** Adding a new section type requires a frontend component. The catalog grows over time but is not user-extensible
- **Synchronous action submission,** Actions are submitted via HTTP POST and block until the backend responds. Long-running actions should return a progress interaction immediately
- **No conditional visibility,** Section visible is static. Dynamic show/hide based on other section values would require frontend logic

Future Directions

- **Autonomy levels,** Not all agent proposals need human review. A project-level autonomy_setting (manual / supervised / autonomous) would allow agents to auto-approve interactions below a confidence threshold, presenting them to humans only when uncertainty is high. The descriptor already carries confidence data; the missing piece is the policy layer.
- **DeltaBus-driven flows,** Instead of synchronous API responses, agents could publish flow.proposed events to DeltaBus. Multiple rendering surfaces (web, chat, mobile, email) would subscribe and present the flow through their respective renderers. This decouples flow creation from rendering entirely.
- **Collaborative flows,** Multiple humans reviewing the same flow simultaneously, with real-time state synchronization via Delta CDF. The DeltaFlowStore already provides the persistence layer; the missing piece is change notification to connected clients.
- **Flow templates,** Pre-built flow templates for common patterns (confirmation, CRUD review, batch approval) that backend services can instantiate with minimal code, further reducing the adapter effort.

Conclusion

The Auraa Semantic Flow pattern addresses a gap in the current landscape of agent-human collaboration tooling. As AI agents become increasingly capable, discovering metadata, proposing pipelines, suggesting quality rules, the bottleneck shifts from what agents can do to how effectively they can communicate proposals to humans for review and approval.

Free-text prompts are too unstructured for complex, multi-faceted proposals. Static buttons are too rigid for nuanced, confidence-varying recommendations. Permanent pages are too slow to build and too inflexible for contextual, time-bounded decisions.

Auraa Semantic Flows provide a third path: **transient, data-driven interaction surfaces** that give agents a rich vocabulary for structured proposals, die when their purpose is fulfilled, and render identically across visual and conversational interfaces. The pattern is implemented, in production, and has demonstrated significant reduction in time-to-feature for new agent capabilities.

The source of its power is not any single technical innovation, but the composition of several: schema-driven rendering, transient lifecycle management, HMAC-signed stateless context, confidence-aware defaults, dual-surface rendering, and lakehouse-native persistence. Together, these enable a fundamentally different relationship between AI agents and human users, one where the agent doesn't just tell the human what it found, but shows them in a structured, reviewable, time-bounded, actionable format.

References

Human-AI Interaction

- Amershi, S., Weld, D., Vorvoreanu, M., et al. (2019). "Guidelines for Human-AI Interaction." CHI 2019, ACM. https://dl.acm.org/doi/10.1145/3290605.3300233
- Horvitz, E. (1999). "Principles of Mixed-Initiative User Interfaces." CHI 1999, ACM. https://dl.acm.org/doi/10.1109/5254.796083
- Gomez, C., Cho, S.M., Ke, S., Huang, C.M., and Unberath, M. (2024). "Human-AI Collaboration Is Not Very Collaborative Yet: A Taxonomy of Interaction Patterns in AI-Assisted Decision Making." Frontiers in Computer Science, 6. https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2024.1521066/full
- Weisz, J.D., He, J., Muller, M., et al. (2024). "Design Principles for Generative AI Applications." CHI 2024, ACM. https://dl.acm.org/doi/10.1145/3613904.3642466

Confidence and Uncertainty Visualization

- Banovic, N., Yang, Z., et al. (2024). "Are You Really Sure? Understanding the Effects of Human Self-Confidence Calibration in AI-Assisted Decision Making." CHI 2024, ACM. https://dl.acm.org/doi/10.1145/3613904.3642671
- Kizilcec, R.F., and Schneider, E. (2025). "Trusting AI: Does Uncertainty Visualization Affect Decision-Making?" Frontiers in Computer Science. https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2025.1464348/full
- Xu, W. (2023). "Using AI Uncertainty Quantification to Improve Human Decision-Making." arXiv:2309.10852. https://arxiv.org/html/2309.10852v1

Ephemeral and Transient Interfaces

- Doring, T., Sylvester, A., and Schmidt, A. (2013). "A Design Space for Ephemeral User Interfaces." TEI 2013, ACM. https://dl.acm.org/doi/10.1145/2460625.2460637
- Doring, T., and Sylvester, A. (2013). "Ephemeral User Interfaces." ACM Interactions, 20(4). https://interactions.acm.org/archive/view/july-august-2013/ephemeral-user-interfaces
- Findlater, L., Moffatt, K., McGrenere, J., and Dawson, J. (2009). "Ephemeral Adaptation: The Use of Gradual Onset to Improve Menu Selection Performance." CHI 2009, ACM. https://dl.acm.org/doi/10.1145/1518701.1518956

Agent Communication Protocols

- Anthropic. (2024). "Model Context Protocol (MCP)." Open specification. https://modelcontextprotocol.io/specification/2025-11-25
- Google. (2025). "Agent2Agent Protocol (A2A)." Open specification. https://a2a-protocol.org/latest/
- FIPA. (1996–2005). "Agent Communication Language (FIPA-ACL)." IEEE Computer Society.
- OpenAI. (2025). "Structured Outputs and Function Calling." OpenAI API Documentation. https://developers.openai.com/api/docs/guides/structured-outputs

Agent-Driven and Generative UI

- Google. (2025). "A2UI: An Open Project for Agent-Driven Interfaces." https://developers.googleblog.com/introducing-a2ui-an-open-project-for-agent-driven-interfaces/
- Leviathan, Y., Valevski, D., et al. (2025). "Generative UI: LLMs are Effective UI Generators." Google Research. https://generativeui.github.io/static/pdfs/paper.pdf
- Tanaka, S., et al. (2025). "GenerativeGUI: Dynamic GUI Generation Leveraging LLMs for Enhanced User Interaction on Chat Interfaces." CHI EA 2025, ACM. https://dl.acm.org/doi/10.1145/3706599.3719743
- Vercel. (2024). "AI SDK 3.0 with Generative UI Support." https://vercel.com/blog/ai-sdk-3-generative-ui
- CopilotKit. (2025). "AG-UI: The Agent-User Interaction Protocol." https://docs.ag-ui.com/
- Stieglitz, D., et al. (2025). "A Multimodal GUI Architecture for Interfacing with LLM-Based Conversational Assistants." arXiv:2510.06223. https://arxiv.org/html/2510.06223v2

Structured Messaging and Schema-Driven UI

- Slack Technologies. (2024). "Block Kit: A UI Framework for Slack Apps." https://docs.slack.dev/block-kit/
- Microsoft. (2024). "Adaptive Cards: Platform-Agnostic UI Snippets." https://adaptivecards.io/
- RJSF Team. (2024). "React JSON Schema Form." Open-source project. https://github.com/rjsf-team/react-jsonschema-form

Companion Documents

- Auraa Semantic Flow Literature Review, Detailed review of 28 sources across the three pillars (rich agent vocabulary, transient surfaces, dual-surface rendering) and cross-cutting themes.
- DeltaBus: A Lakehouse-Native Event Bus Pattern, Companion whitepaper on the lakehouse-native messaging pattern referenced in §4.3 and §10.2.

Document Version: 1.0 Status: Production Last Updated: April 2, 2026 Authors: Covasant Platform Team Audience: Architects, Developers, Product Managers, Technical Leaders, AI Engineers

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