Data-Sync Platform

Data Oriented Systems

Guilherme Rosa 113968
Rui Machado 113765

15/12/2025

1. Problem Statement

Modern organizations increasingly rely on **heterogeneous data architectures**, where different data storage technologies are employed based on specific operational requirements. While this polyglot persistence approach offers advantages in terms of performance optimization and system specialization, it introduces significant challenges in **data integration and analysis**. This project addresses three fundamental problems inherent to federated data environments.

1.1. Data Fragmentation and Integration Complexity

In contemporary data ecosystems, enterprise data is distributed across multiple database management systems (DBMS), each optimized for distinct workloads. The core challenge lies in the lack of native **interoperability** between these heterogeneous systems. Cross-database queries cannot be executed through standard query languages. Current approaches require implementing custom application-layer logic that queries each data source **independently,** performs data reconciliation and joins, **aggregates results** across disparate schemas

This process introduces substantial development overhead, computational inefficiency, and maintenance complexity, particularly as the number of data sources scales.

1.2. Technical Accessibility Barriers

Effective data analysis in federated environments demands extensive technical expertise across multiple domains:

- **Query language proficiency:** Users must master various query languages (SQL for relational databases, MongoDB Query Language for document stores)
- **Schema knowledge:** Deep understanding of physical table structures, column definitions, and data relationships across all systems
- **System-specific nuances:** Awareness of data type differences, naming conventions, and architectural constraints

This high barrier to entry effectively excludes non-technical stakeholders (product managers, business analysts, and domain experts) from direct data exploration.

1.3. Metadata Discovery and Schema Visibility

Data systems lack a **unified metadata catalog** that provides comprehensive visibility into available data assets. Without centralized schema introspection, users face several challenges:

- **Discovery inefficiency:** Determining the existence of specific tables, columns, or data attributes requires manual inspection of individual databases
- **Schema drift blindness:** Changes to underlying data structures are not propagated or tracked across the federation
- **Integration friction:** Building user interfaces for data exploration requires real-time metadata queries, which are impractical without a centralized metadata layer

This metadata opacity impedes both automated system integration and human-driven data discovery, limiting the agility of data-driven operations.

2. Concept

The core philosophy of the platform is **Data Abstraction**. We solve the problem of physical data separation by creating a logical layer that sits above the database infrastructure.

2.1. Virtual Data Federation

The system abstracts the physical reality of the underlying databases. Instead of interacting with "MySQL" or "MongoDB" directly, the platform creates a set of **Virtual Global Tables**. These are logical definitions that unify disparate data sources into a single, federated namespace. This allows users and the system to query data as if it were all in one place, completely hiding the complex physical boundaries between the document store and relational databases.

2.2. Decoupled Metadata Management

To make this virtual layer performant, we decouple discovery from execution. A background process proactively "syncs" the schemas of these virtual tables into a high-speed local cache. This ensures that the system has an instant, up-to-date map of the global data structure without needing to query the slow, underlying database connections every time a user wants to browse a table.

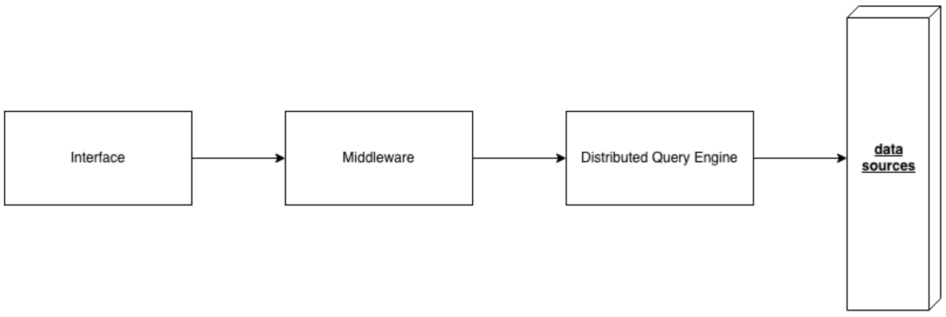
2.3. AI-Driven Semantic Layer

We replace the need for technical query languages with a semantic interface. The AI agent is fed the context of the **Virtual Global Tables**, allowing it to understand the relationships between data sources. It translates natural language questions into precise federated queries against this virtual layer, allowing non-technical users to join data across systems without knowing they are doing so.

3. Solution: System Architecture

3.1. High-Level Design

The system implements a tiered architecture designed to abstract the physical storage from the user.



3.1.1. Interface Layer

The Interface Layer serves as the central command center for all data operations, streamlining the workflow from information discovery to query execution. Its primary architectural function is **State Management and Visualization**; it acts as a translation layer that converts user actions into structured intent for the middleware, without performing local data processing.

- **Natural Language Querying:** The primary interaction model allows users to query data using plain English (e.g., *"Show me sales by region"*). This abstracts the underlying query language, leveraging the middleware to generate executable SQL automatically.

- **Direct SQL Execution:** The platform supports a hybrid interaction model by enabling raw SQL inputs. This allows technical users to bypass the semantic layer when necessary, offering granular control for query construction and system validation
- **Global Table Presentation:** Virtual Global Tables are exposed as immediate, accessible resources. This design eliminates the need to navigate distributed data sources, allowing users to instantly identify and interact with federated data.
- **Result Rendering:** The layer transforms raw query results into consumable formats. It manages data visualization, table formatting, and pagination, ensuring that even large datasets are presented clearly and efficiently.

3.1.2. Middleware Layer

This layer acts as the query federation and metadata abstraction brain of the platform, creating an intelligent bridge between the user interface and the underlying data engine. Its primary architectural function is Query Translation and Metadata Management; it transforms logical data requests into executable queries across distributed data sources, while maintaining a comprehensive understanding of the entire data landscape.

- **Metadata Discovery and Synchronization:** The backend automatically discovers and catalogs the complete hierarchy of data sources - catalogs, schemas, tables, and columns - from the underlying query engine. It maintains an in-memory metadata registry that provides instant access to schema information, enabling the interface to deliver immediate feedback without repeatedly querying data sources. This registry continuously synchronizes with connected databases, ensuring the platform always reflects the current state of available data.
- **Global Table Abstraction:** The layer introduces "global tables" - logical views that unify multiple physical tables from different data sources into a single queryable entity. A user querying a global table doesn't need to know whether data resides in PostgreSQL, MySQL, or MongoDB; the abstraction completely hides this complexity. These global tables serve as business-oriented representations of data, decoupling user queries from physical storage locations.
- **Intelligent Query Translation:** When queries target global tables, the backend translates them into executable queries against physical tables. This translation operates in progressive phases, handling everything from simple single-table queries with filtering and limits to complex multi-source operations involving unions and joins.

The system automatically resolves column references, maps global columns to their physical counterparts, and generates optimized execution plans across distributed sources.

- **Relationship-Based Federation:** The platform enables definition of explicit relationships between tables through union and join operations. When relationships are defined, the system auto-provisions the necessary global tables, mappings, and column configurations, making complex logical views immediately queryable. This allows queries to seamlessly span multiple databases - for instance, unioning regional transaction data while joining with centralized customer information - all through a single logical query.
- **Natural Language Query Generation:** Recognizing that not all users write SQL, the layer incorporates an AI-powered agent that translates natural language into executable queries. The agent leverages the global table abstractions to generate queries, accessing metadata discovery and schema exploration capabilities to understand what data is available and how it connects. This conversational interface allows business users to ask questions in plain English while the backend handles the complexity of multi-database query construction.

The result is an architecture that transforms complex multi-database queries into operations as simple as querying a single table, while offering both programmatic APIs for application integration and conversational interfaces for business users.

3.1.4. Data Persistence Layer

This layer consists of the heterogeneous data sources that store the platform's actual data. In this architecture, these are treated purely as storage engines with no awareness of the federation layer above them. Each data source operates independently, managing its own data according to its native storage model and access patterns.

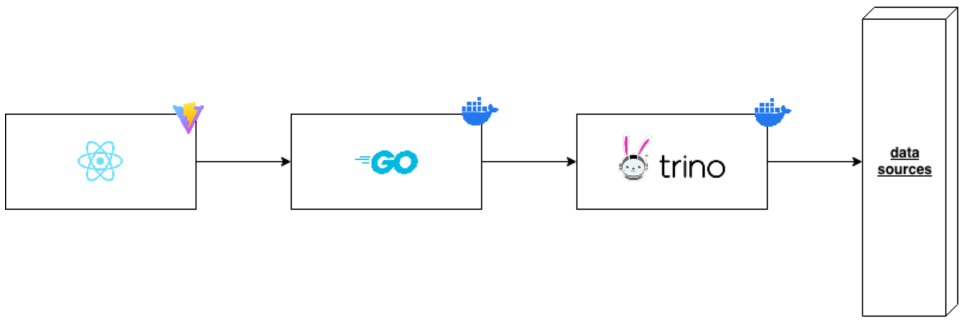
- **Connector-Based Extensibility:** The architecture employs a connector-based abstraction that enables integration with virtually any data source. Connectors handle the fundamental differences between storage systems - translating between relational schemas and tabular representations, mapping document structures to queryable tables, bridging object stores to columnar formats, or adapting time-series databases to standard query interfaces. This connector layer ensures that despite storing data in fundamentally different formats and supporting different query languages, all sources appear as uniform tabular structures to the upper layers.

- **Physical Data Distribution:** Data sources may be distributed across regions, cloud providers, or organizational boundaries, each managing different domains or subsets of the overall dataset. One source might store customer data with full relational integrity, while another houses transactional records optimized for high-throughput writes. A document store could maintain product catalogs with complex nested attributes, while a columnar database holds analytical aggregates. The federation layer unifies these disparate sources, allowing queries to seamlessly span physical and organizational boundaries.
- **Distributed Query Execution:** The query engine operates as a coordinator-worker cluster that orchestrates query execution across multiple data sources. When a federated query spans different storage systems, the coordinator distributes workload across workers, each executing portions against different sources in parallel. Workers push computation down to the storage layer where possible, leveraging each source's native query capabilities, then aggregate results at the federation layer. This distributed execution model enables the platform to query petabyte-scale datasets across dozens of sources with performance characteristics similar to querying a single database.

This architecture transforms disparate storage systems into a unified data platform, where the complexity of multi-database operations, data model translation, protocol differences, and distributed query execution is completely abstracted from the application layer - allowing organizations to add, remove, or modify data sources without impacting the user experience.

3.2. Technology Stack

The following diagram details the specific technologies chosen to implement each layer of the architecture.



3.2.1. Orchestration & Infrastructure

- **Docker:** Ensures consistency across environments by packaging each component with its complete runtime, eliminating dependency conflicts in a distributed system where the query engine, backend, and data sources have different requirements.
- **Docker Compose:** Manages complex startup sequencing (data sources → query engine → backend) and provides automatic internal networking with stable service discovery, enabling the entire distributed system to spin up with a single command.

3.2.2. Query Engine Layer

- **Trino:** Distributed SQL query engine specifically designed for federated analytics - queries data in place across heterogeneous sources without moving it, with a connector architecture supporting dozens of data sources and horizontal scalability through coordinator-worker distribution.

3.2.3. Application Layer (Backend)

- **Go (Golang):** Exceptional performance with lightweight goroutines for handling thousands of concurrent operations (metadata sync, query translation, AI requests), strong static typing to prevent runtime failures during complex query translation, and single-binary compilation for simplified deployment.

3.2.4. Interface Layer (Frontend)

- **React 19:** Component-based architecture enabling UI composition through reusable building blocks, with declarative rendering patterns that simplify complex state management and a mature ecosystem providing pre-built solutions for common interface challenges.
- **TypeScript:** Static type safety across complex data structures (global tables, column mappings, metadata hierarchies), catching integration errors at development time rather than runtime.
- **Vite:** Near-instantaneous development feedback with hot module replacement and optimized production builds with automatic code splitting and tree shaking.

4. Development

The project follows a well-established architectural pattern that enforces separation between domain logic, application-specific code, and presentation layers. The overall structure is organized as follows:

Backend Services (/cmd, /pkg, /internal): The backend implements a Go-based middleware layer following the Standard Go Project Layout. This structure distinguishes between:
Core domain logic (pkg/data-sync/): Reusable packages containing business logic, exposing clean interfaces while maintaining implementation encapsulation
Application layer (internal/api/): HTTP transport implementation including request routing, payload serialization, and middleware composition
Entry points (cmd/data-sync/): Application initialization and dependency injection

Frontend Application (/interface/data-sync): A React-based Single Page Application (SPA) implementing a modern component-driven architecture with TypeScript for type safety and maintainability.

Data Source Configuration (/data-sources): Docker Compose configurations and initialization scripts for local development and testing environments.

This architectural separation ensures that core business logic remains implementation-agnostic, facilitating independent testing, deployment flexibility, and future extensibility.

4.1 Backend

The backend architecture is implemented in **Go**, serving as the intelligent middleware layer between heterogeneous data sources and client applications. The system adopts a modular, layered architecture based on the Standard Go Project Layout pattern, establishing a strict separation between domain logic and transport mechanisms. This architectural decision ensures that core business logic remains **implementation-agnostic**, facilitating independent testing and promoting code reusability.

4.1.1 Organization

The codebase architecture implements a clear separation of concerns through two primary directory structures:

Domain Logic Layer (pkg/data-sync/);
This layer encapsulates core system functionality as a collection of domain-specific packages. Each package represents a bounded context within the system:

- **Discovery:** Schema introspection and metadata synchronization
- **Storage:** Metadata persistence and caching mechanisms
- **Query:** Query translation and execution orchestration
- **Chatbot:** AI-powered natural language query interface
- **Matching:** Automated schema relationship detection
- **Sync:** Metadata synchronization workflow management

Each package exposes well-defined interfaces while maintaining implementation encapsulation, adhering to principles of low coupling and high cohesion. This organization enables independent unit testing and supports future extensibility without affecting consumer code.

Application Layer (internal/api/):
This layer provides HTTP transport implementation, isolated from external package dependencies through Go's internal visibility mechanism. Key responsibilities include:

- **Request routing (routers/):** RESTful endpoint definitions organized by domain
- **Middleware composition:** Cross-cutting concerns (CORS, logging, error handling)
- **Server initialization (server.go):** HTTP server configuration and lifecycle management

By constraining transport-specific code to this layer, the architecture maintains protocol independence and facilitates alternative interface implementations (e.g., gRPC, WebSocket) without modifying core logic.

4.1.2. Core Services

The middleware functionality is orchestrated through several specialized services within the domain logic layer, each addressing distinct architectural challenges.

4.1.2.1 Metadata Discovery and Synchronization Engine

The Metadata Discovery service addresses a fundamental challenge in federated query environments: the performance overhead of real-time schema introspection across distributed data sources.

Architectural Approach: The service implements a proactive metadata synchronization strategy rather than reactive on-demand queries. This design pattern materializes schema information in an in-memory cache, trading storage for query latency reduction.

Schema Discovery Protocol:A scheduled background process executes periodic introspection queries against the Trino Coordinator's system catalog tables:

- *system.metadata.catalogs*
- *system.metadata.schemas*
- *system.metadata.tables*
- *system.metadata.columns*

The discovery process traverses this hierarchical structure, extracting complete metadata including table names, column definitions, data types, and schema relationships.

Metadata Persistence Layer: Discovered schema information is persisted through the Storage Service (pkg/data-sync/storage), which implements:

- **Thread-safe in-memory caching:** Concurrent read access with write synchronization
- **Hierarchical indexing:** Nested map structures enabling O(1) lookup by catalog/schema/table path
- **Change detection:** Comparison algorithms to identify schema drift between synchronization cycles

This architecture achieves sub-millisecond metadata lookup latency, enabling responsive frontend interactions independent of backend database load.

Synchronization Strategy: The service employs an incremental synchronization algorithm that compares discovered metadata against cached state, performing differential updates to minimize computational overhead while maintaining cache coherence. The system detects additions, modifications, and deletions of database objects, triggering appropriate cache invalidation and refresh operations

4.1.2.2 Query Translation and Execution Engine

The Query Engine (pkg/data-sync/query) abstracts the complexity of the Trino federation layer, providing a **unified query execution interface** while managing protocol-specific communication details.

Key Responsibilities:

- **Query Translation:** Transforms logical query representations (global table references) into valid Trino SQL statements, resolving abstract table names to physical catalog-qualified identifiers based on metadata mappings.
- **Execution Management:** Orchestrates the query lifecycle through the Trino client protocol, including statement submission, asynchronous result polling, and result set materialization. The engine implements streaming result processing to prevent memory saturation when handling large datasets, maintaining bounded memory consumption regardless of result set size.
- **Error Handling and Resilience:** Implements retry logic and graceful degradation strategies to handle transient failures in the federated query environment.

4.1.2.3 AI-Powered Query Assistant

The Chatbot Service (pkg/data-sync/chatbot) integrates Google's **Gemini** large language model to provide a natural language interface for data exploration. The primary architectural challenge addressed by this service is **grounding**: constraining the generative model's output space to prevent hallucinations and ensure generated queries are valid within the specific federated schema.

Implementation Strategy:

- **Dynamic Context Construction:** Upon user interaction, the service retrieves current schema metadata from the Metadata Discovery Engine. This information is dynamically injected into the model's system prompt, creating a schema-aware context that describes available tables, columns, and their relationships.
- **Constrained Generation:** The service prompt engineering strategy instructs the model to function as a Trino SQL expert with strict output formatting requirements. Generated responses are parsed using regular expression extraction to isolate SQL code blocks, which undergo syntactic validation before execution.
- **Tool-Augmented Generation:** The service implements a function-calling pattern, exposing metadata discovery and query execution as tools available to the model.

This enables the agent to perform multi-step reasoning: first discovering available tables, then generating appropriate queries based on actual schema structure.

Safety and Validation:

- Generated SQL statements undergo validation to prevent destructive operations (only *SELECT* statements are permitted)
- Query results are returned with execution metadata (row count, execution time) to provide transparency
- The service maintains conversation state to enable contextual follow-up queries

4.1.2.4 Automated Relationship Discovery

The Matching Service (pkg/data-sync/matching) implements AI-powered schema relationship detection to automate the creation of global table definitions through JOIN and UNION operations.

Strategy Pattern:

The service adopts a strategy pattern enabling multiple relationship detection algorithms:

- **Current Implementation:** Gemini-based semantic matching
- **Extensibility:** Supports future rule-based or ML-based strategies

The Gemini strategy analyzes table metadata to identify:

- UNION Candidates (Vertical Stacking):
  - Tables with identical or highly similar schemas
  - Temporal partitions (e.g., orders_2023, orders_2024)
  - Regional splits with consistent structure

- JOIN Candidates (Horizontal Enrichment):
  - Tables representing the same entity across different data sources
  - Semantic similarity in table names (e.g., customers vs. customer_profiles)
  - Presence of compatible join keys (matching column names/types)

Matching Algorithm:

1. Metadata Collection: Gather complete schema information for all physical tables
2. Context Construction: Serialize table structures into JSON format
3. AI Analysis: Submit to Gemini with structured prompt defining relationship criteria
4. Result Parsing: Extract relationship suggestions from JSON-formatted response
5. Confidence Scoring: Filter suggestions based on confidence thresholds (≥0.5)

Auto-Creation Workflow:
Discovered relationships are automatically materialized as global table definitions, with:

- Entity-based naming (e.g., "customers", not "postgres_mysql_customers_join")
- Appropriate join column selection based on primary key analysis
- Validation against physical schema to prevent invalid relationships

4.1.2.4 Trino Configuration

The Trino distributed SQL query engine serves as the **federation layer**, enabling unified querying across heterogeneous data sources. The deployment architecture consists of one **coordinator node and three worker nodes**, orchestrated via Docker Compose to ensure consistent networking and service discovery.

The coordinator node is configured with specific resource constraints (5GB maximum query memory, 1GB per node) and operates on port 8080. Dynamic catalog management is enabled, allowing runtime registration of data source connectors without requiring service restarts. This configuration approach supports the system's metadata synchronization requirements, where catalogs may be discovered and registered programmatically.

Three catalog connectors are configured to establish federation across the heterogeneous data landscape:

- **PostgreSQL Connector:** Connects to the relational database containing core operational data (customers, products, historical orders)
- **MySQL Connector:** Integrates the CRM system with extended customer profiles and inventory management data
- **MongoDB Connector:** Provides access to document-based collections storing unstructured data such as product reviews and customer interactions

The worker nodes share the same catalog configurations but operate in execution mode, processing query fragments distributed by the coordinator. This distributed query execution architecture ensures scalability while maintaining the abstraction of a unified data source for the application layer.

4.2. Frontend

The frontend is architected as a Single Page Application (SPA) providing an intuitive control plane for the federated data platform. Built with React and TypeScript, the application abstracts backend complexity behind a modern, responsive user interface.

4.2.1 Project Structure

The frontend architecture implements a layered organization optimized for component reusability and clear separation between presentation, business logic, and infrastructure concerns:

Page Layer (pages/): This layer contains route-level components that compose the application's primary views. Each page represents a distinct user workflow:

- **ChatPage:** Conversational AI interface for natural language querying
- **QueryPage:** Direct SQL editor with result visualization
- **SchemaStudioPage:** Global table and relationship management interface

Pages orchestrate feature-specific logic, coordinate data fetching through TanStack Query, and compose reusable components into coherent user experiences. They remain isolated from implementation details of API communication or low-level UI primitives.

Component Library (components/): This layer provides a hierarchical component system organized by abstraction level:

- **Primitive Components (ui/):** Foundational, unstyled components built on Radix UI primitives - buttons, dialogs, tables, inputs, and form controls. These implement accessibility, keyboard navigation, and ARIA patterns while remaining style-agnostic through composition.
- **Domain Components (assistant-ui/):** Specialized components for the AI chat interface - message threading, tool execution visualization, markdown rendering, and attachment handling. These encapsulate complex interaction patterns specific to conversational interfaces.
- **Shared Components:** Cross-cutting components like QueryResultTable that provide feature-specific functionality reused across multiple pages.

Each component maintains single responsibility, accepts configuration through typed props, and remains framework-agnostic in its internal logic.

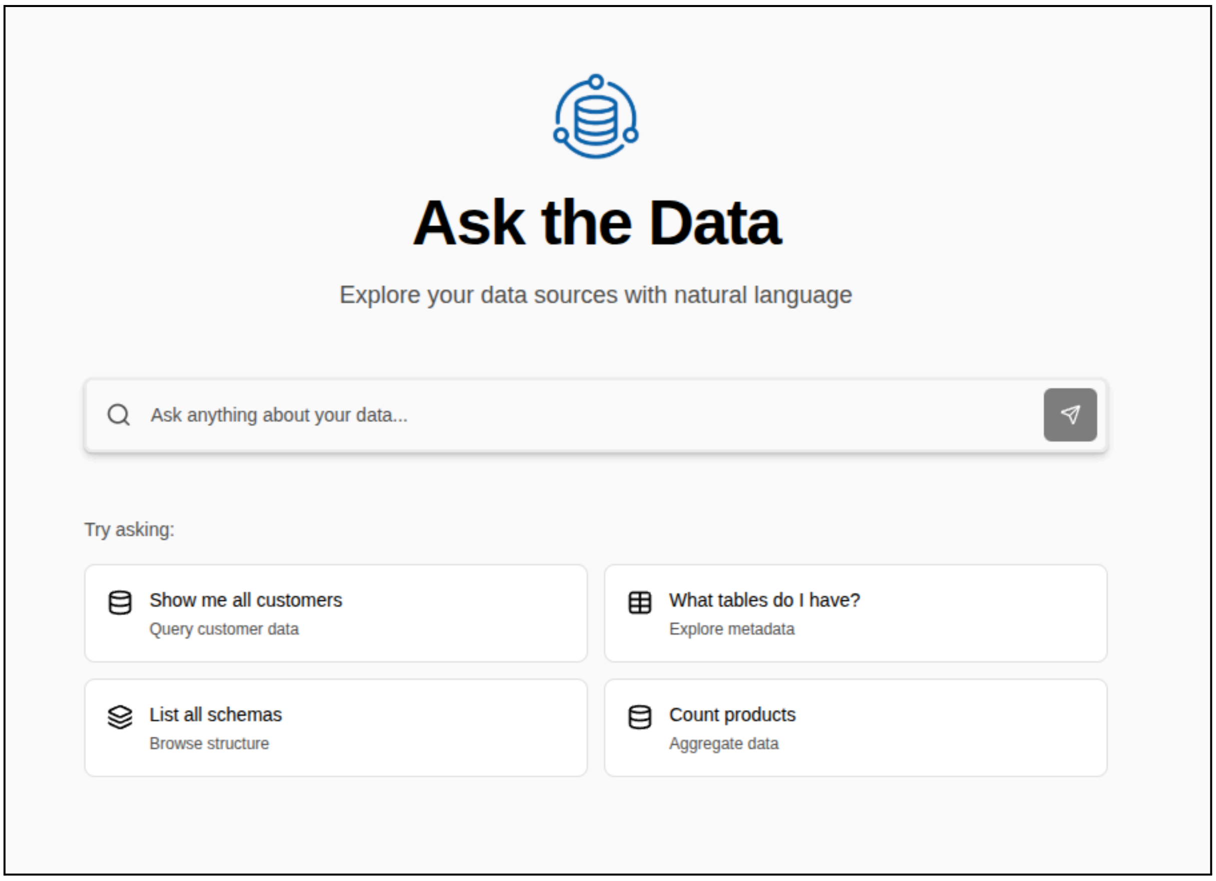
Infrastructure Layer (lib/): This layer encapsulates cross-cutting concerns and external integrations:

- **API Client (api.ts):** Type-safe HTTP client providing strongly-typed methods for all backend endpoints. Defines TypeScript interfaces for all request/response payloads ensuring compile-time type safety across the frontend-backend boundary. Centralizes error handling and request configuration.
- **Layout Layer (layouts/):** Provides application shell components that define navigation structure, page composition, and responsive behavior. The MainLayout component implements the sidebar navigation, routing integration, and theme management, remaining consistent across all page views.
- **Routing Architecture (App.tsx):** Type-safe routing through TanStack Router with compile-time route validation. Routes are defined declaratively with parent-child relationships, enabling automatic code splitting per route and type-safe navigation throughout the application.

This structure enforces unidirectional data flow: pages fetch data through the API client, manage state through TanStack Query hooks, and render UI by composing components from the component library. Components remain pure and stateless where possible, with state management isolated to pages or dedicated state management hooks using Zustand for client-side state.

4.2.2 Chat Interface (Conversational Analytics)

The Chat Interface implements a conversational paradigm for data exploration, enabling natural language interaction with the federated data platform.



Ask the Data

Explore your data sources with natural language

Ask anything about your data...

Try asking:

Show me all customers
Query customer data

What tables do I have?
Explore metadata

List all schemas
Browse structure

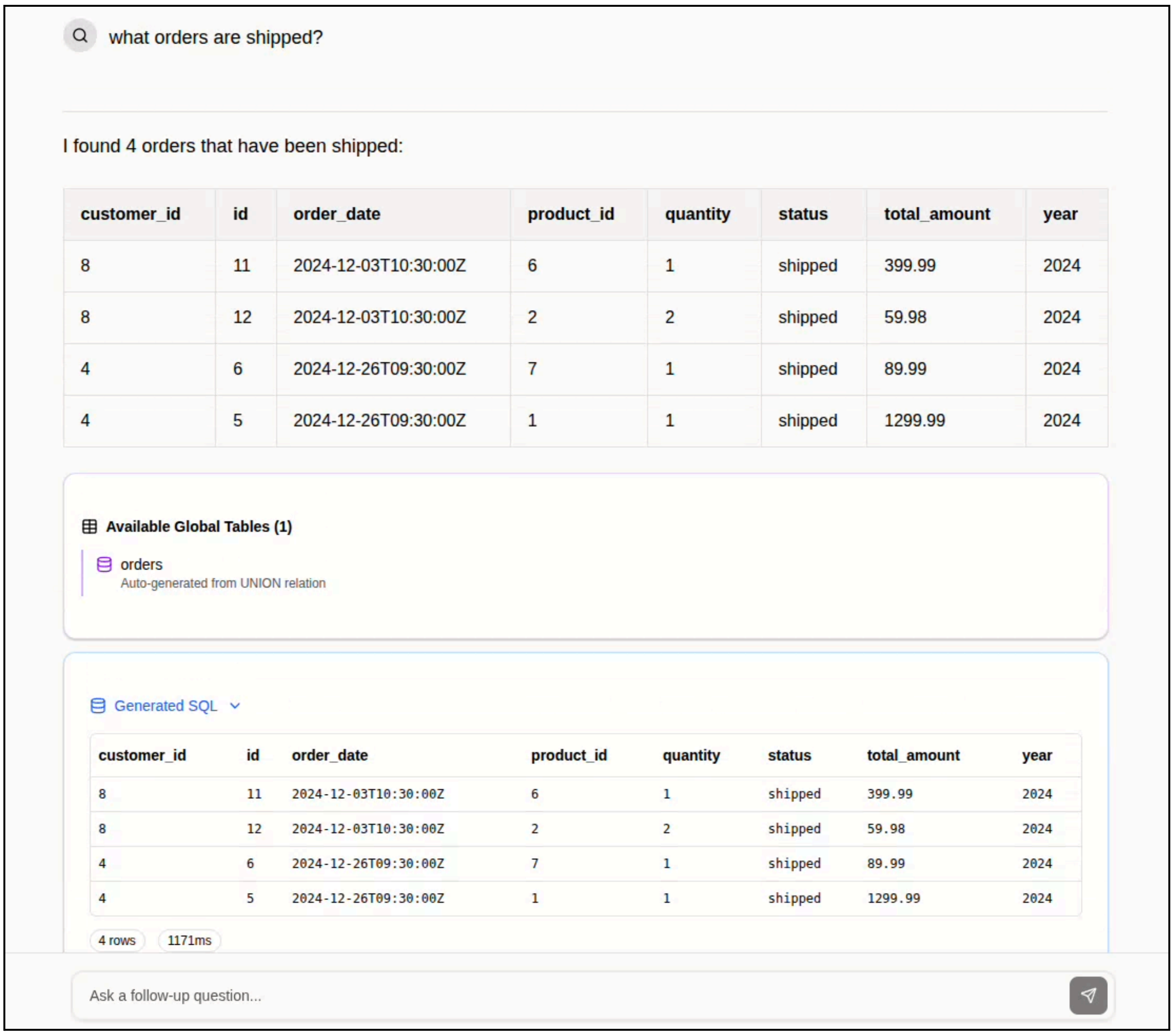
Count products
Aggregate data

Conversation Management: The interface maintains multiple conversation threads, each preserving:

- Complete message history (user queries and assistant responses)
- Contextual state enabling follow-up queries ("filter that by region")
- Tool execution results (query outputs, metadata discoveries)

Message Flow:

1. User submits natural language query
2. Frontend dispatches request to /api/chatbot/message endpoint
3. Backend streams response with potential tool executions
4. Frontend renders response incrementally with typing indicators
5. Results materialize in specialized components based on message type



what orders are shipped?

I found 4 orders that have been shipped:

customer_id	id	order_date	product_id	quantity	status	total_amount	year
8	11	2024-12-03T10:30:00Z	6	1	shipped	399.99	2024
8	12	2024-12-03T10:30:00Z	2	2	shipped	59.98	2024
4	6	2024-12-26T09:30:00Z	7	1	shipped	89.99	2024
4	5	2024-12-26T09:30:00Z	1	1	shipped	1299.99	2024

Available Global Tables (1)

orders
Auto-generated from UNION relation

Generated SQL

customer_id	id	order_date	product_id	quantity	status	total_amount	year
8	11	2024-12-03T10:30:00Z	6	1	shipped	399.99	2024
8	12	2024-12-03T10:30:00Z	2	2	shipped	59.98	2024
4	6	2024-12-26T09:30:00Z	7	1	shipped	89.99	2024
4	5	2024-12-26T09:30:00Z	1	1	shipped	1299.99	2024

4 rows 1171ms

5.2. Query Editor (SQL Workspace)

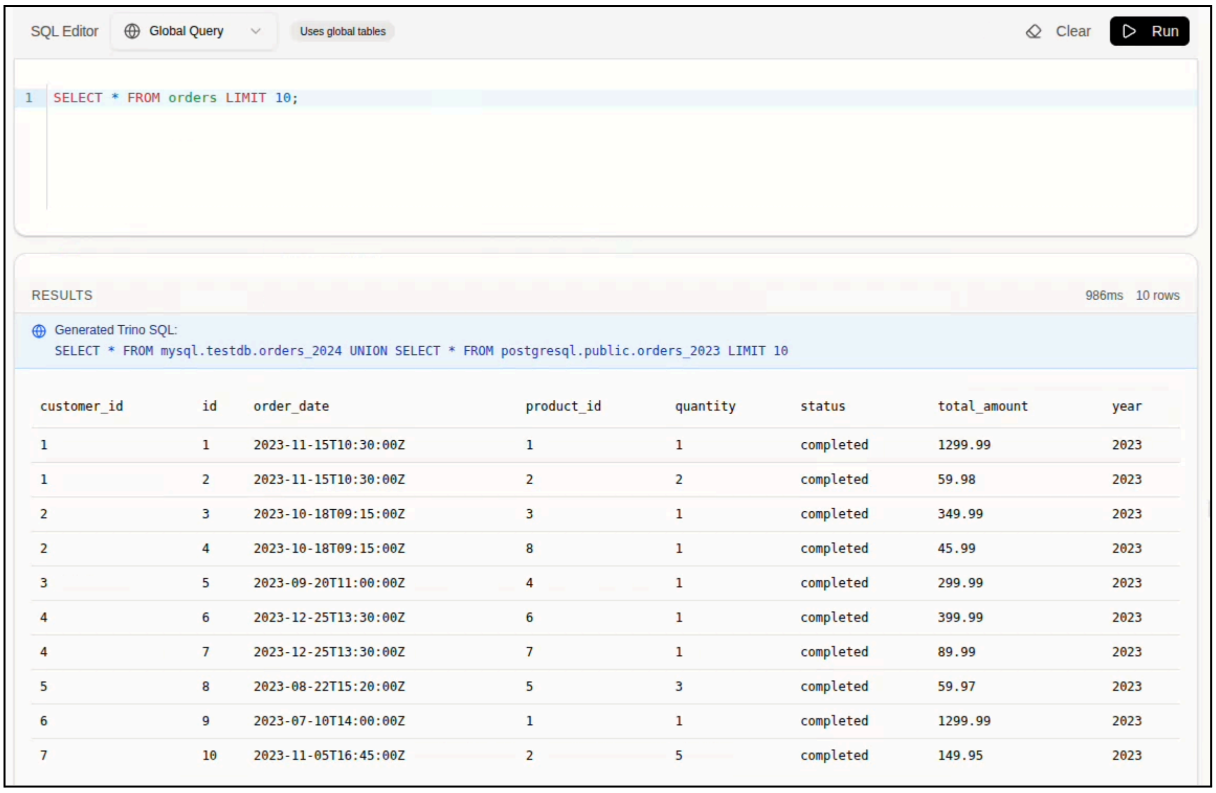
The Query Editor provides a traditional SQL interface for users preferring direct query composition over conversational interaction.

Code Editor Integration:

- Syntax Highlighting: SQL-aware code highlighting using CodeMirror
- Auto-Completion: Context-aware suggestions for tables, columns, SQL keywords
- Error Indication: Inline syntax error detection and highlighting

Execution Management:

- Query Execution: Direct submission to query execution engine
- Result Visualization: Tabular result display with export capabilities
- Query History: Persistent storage of executed queries for reuse



```
SELECT * FROM orders LIMIT 10;
```

RESULTS 986ms 10 rows

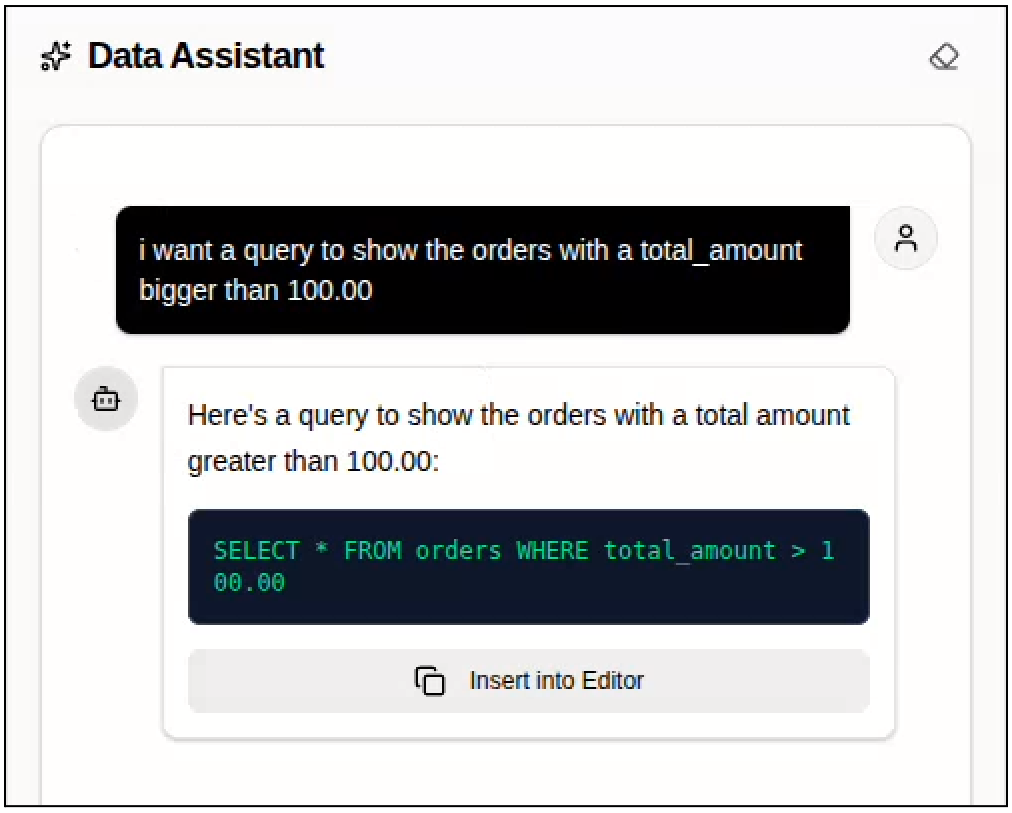
Generated Trino SQL:

```
SELECT * FROM mysql.testdb.orders_2024 UNION SELECT * FROM postgresql.public.orders_2023 LIMIT 10
```

customer_id	id	order_date	product_id	quantity	status	total_amount	year
1	1	2023-11-15T10:30:00Z	1	1	completed	1299.99	2023
1	2	2023-11-15T10:30:00Z	2	2	completed	59.98	2023
2	3	2023-10-18T09:15:00Z	3	1	completed	349.99	2023
2	4	2023-10-18T09:15:00Z	8	1	completed	45.99	2023
3	5	2023-09-20T11:00:00Z	4	1	completed	299.99	2023
4	6	2023-12-25T13:30:00Z	6	1	completed	399.99	2023
4	7	2023-12-25T13:30:00Z	7	1	completed	89.99	2023
5	8	2023-08-22T15:20:00Z	5	3	completed	59.97	2023
6	9	2023-07-10T14:00:00Z	1	1	completed	1299.99	2023
7	10	2023-11-05T16:45:00Z	2	5	completed	149.95	2023

AI-Assisted Query Generation:

- Natural language query description input
- AI-generated SQL with explanation
- Editable generated queries before execution
- Iterative refinement through conversational feedback



i want a query to show the orders with a total_amount bigger than 100.00

Here's a query to show the orders with a total amount greater than 100.00:

```
SELECT * FROM orders WHERE total_amount > 1
00.00
```

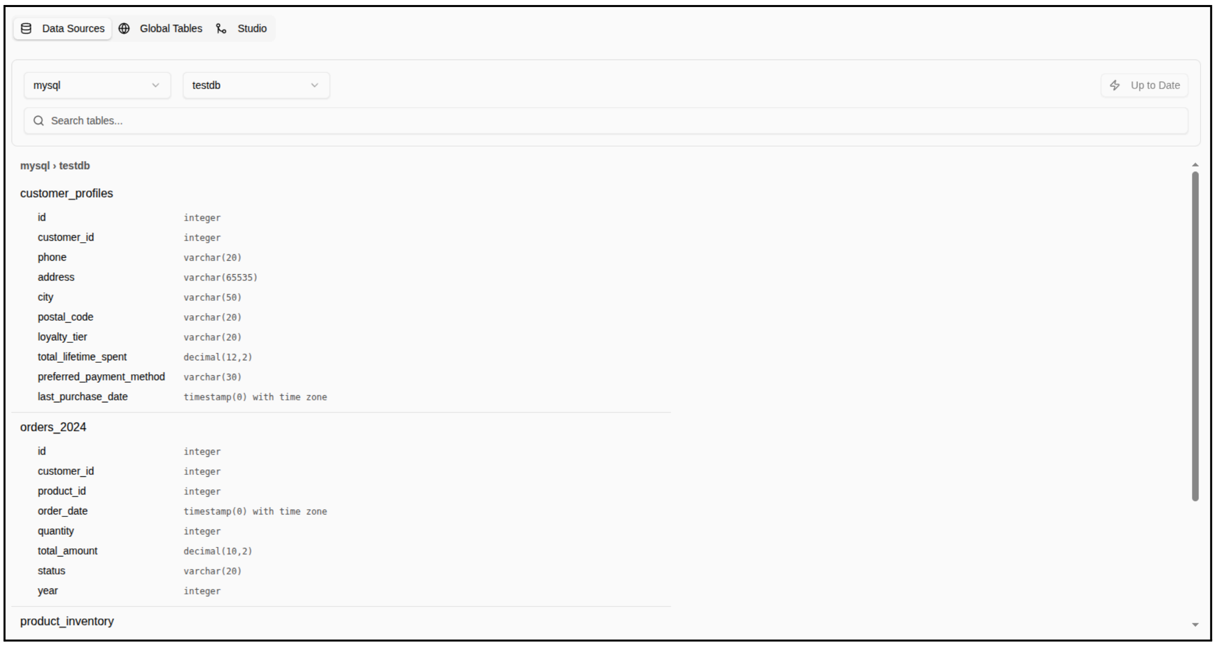
5.2. Schema Studio (Visual Data Discovery)

The Schema Studio addresses metadata visibility challenges through interactive schema exploration, providing a comprehensive "map" of the federated data landscape.
This is divided in **3 tabs**, that have different purposes and use cases:

- Data Sources
- Global Tables
- Studio

5.2.1 Data Sources Tab

This tab is designed to give users a clear overview and allow them to inspect the presented data sources.



mysql › testdb

customer_profiles

id	integer
customer_id	integer
phone	varchar(20)
address	varchar(65535)
city	varchar(50)
postal_code	varchar(20)
loyalty_tier	varchar(20)
total_lifetime_spent	decimal(12,2)
preferred_payment_method	varchar(30)
last_purchase_date	timestamp(0) with time zone

orders_2024

id	integer
customer_id	integer
product_id	integer
order_date	timestamp(0) with time zone
quantity	integer
total_amount	decimal(10,2)
status	varchar(20)
year	integer

product_inventory

Hierarchical Navigation System:

The interface implements a three-level navigation hierarchy reflecting the Trino catalog structure:

- Level 1 - Catalog Selection:
  - Dropdown selector displaying all available catalogs (PostgreSQL, MySQL, MongoDB). Includes "All Catalogs" option for unified view across all data sources.
- Level 2 - Schema Filtering:
  - Upon catalog selection, displays available schemas with search/filter capabilities. Schema-level metadata (table count) provides navigational context.
- Level 3 - Table Inspection:
  - Column name and data type enumeration
  - Table metadata (row count, last modified timestamp where available)
  - Quick actions (preview data, add to global tables)

Real-Time Metadata Synchronization:

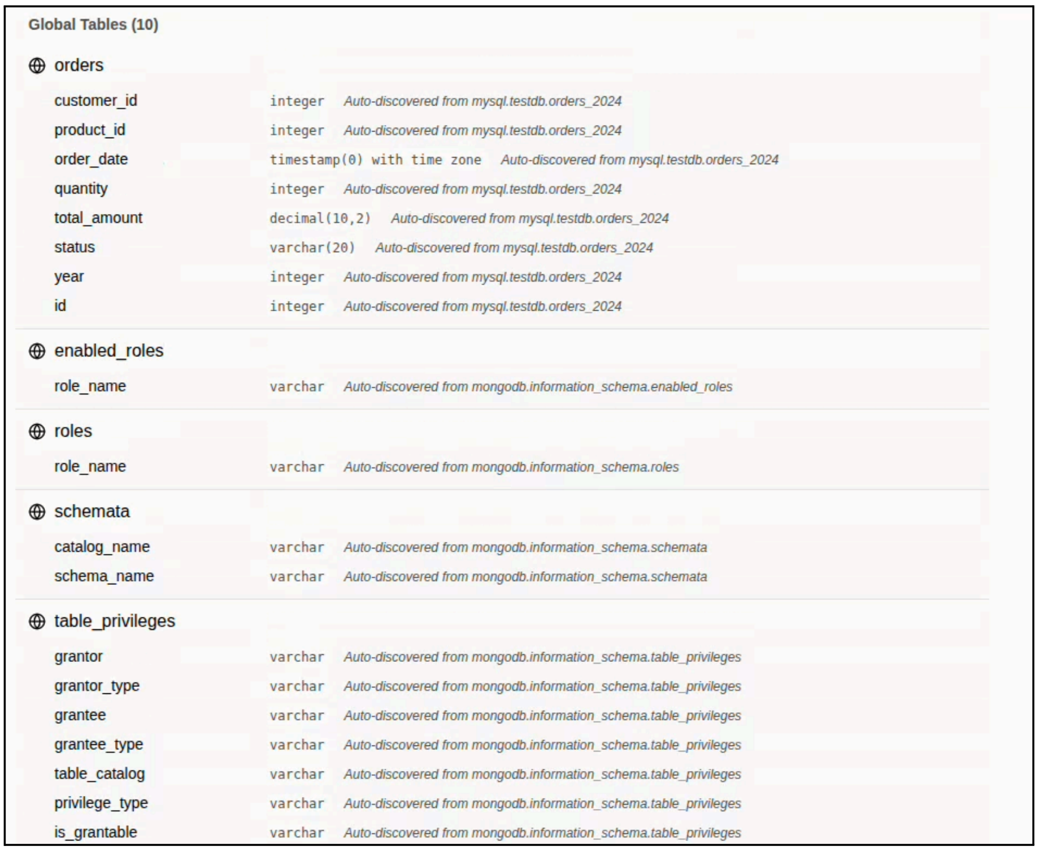
The interface maintains live synchronization with backend metadata:

- React Query cache invalidation on metadata updates
- Polling-based refresh for schema change detection

- Optimistic updates for user-initiated metadata modifications

5.2.1 Global Tables Tab

Global Table Management:
The studio provides tools for defining global tables (logical abstractions mapped to physical tables), in this tab it is possible to get an overview of it.



Global Tables (10)

orders

customer_id	integer	Auto-discovered from mysql.testdb.orders_2024
product_id	integer	Auto-discovered from mysql.testdb.orders_2024
order_date	timestamp(0) with time zone	Auto-discovered from mysql.testdb.orders_2024
quantity	integer	Auto-discovered from mysql.testdb.orders_2024
total_amount	decimal(10,2)	Auto-discovered from mysql.testdb.orders_2024
status	varchar(20)	Auto-discovered from mysql.testdb.orders_2024
year	integer	Auto-discovered from mysql.testdb.orders_2024
id	integer	Auto-discovered from mysql.testdb.orders_2024

enabled_roles

role_name	varchar	Auto-discovered from mongodb.information_schema.enabled_roles

roles

role_name	varchar	Auto-discovered from mongodb.information_schema.roles

schemata

catalog_name	varchar	Auto-discovered from mongodb.information_schema.schemata
schema_name	varchar	Auto-discovered from mongodb.information_schema.schemata

table_privileges

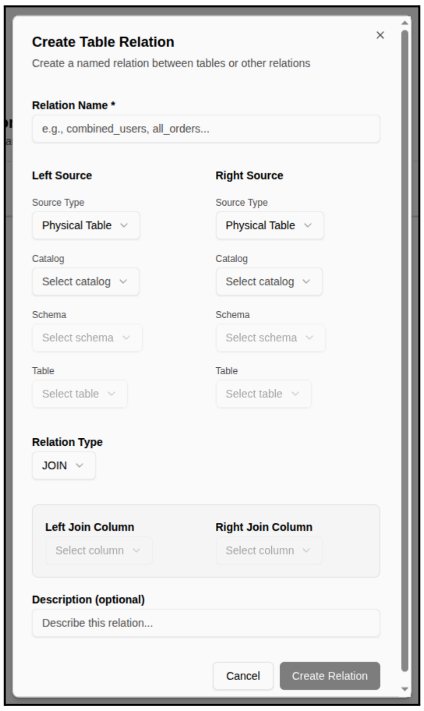
grantor	varchar	Auto-discovered from mongodb.information_schema.table_privileges
grantor_type	varchar	Auto-discovered from mongodb.information_schema.table_privileges
grantee	varchar	Auto-discovered from mongodb.information_schema.table_privileges
grantee_type	varchar	Auto-discovered from mongodb.information_schema.table_privileges
table_catalog	varchar	Auto-discovered from mongodb.information_schema.table_privileges
privilege_type	varchar	Auto-discovered from mongodb.information_schema.table_privileges
is_grantable	varchar	Auto-discovered from mongodb.information_schema.table_privileges

5.2.1 Global Tables Tab

In this tab, it is possible to create table relations and create global tables of entities. There are two ways to do so:

1. **Manual Relation Definition:**
    - Interface for creating JOIN and UNION relationships:
        - Source Selection: Cascading dropdowns for left/right table selection
        - Relation Type: Toggle between JOIN and UNION semantics

- Join Configuration: Column pair selection for JOIN operations
- Validation: Real-time schema compatibility checking



Create Table Relation

Create a named relation between tables or other relations

Relation Name *

e.g., combined_users, all_orders...

Left Source	**Right Source**
Source Type: Physical Table	Source Type: Physical Table
Catalog: Select catalog	Catalog: Select catalog
Schema: Select schema	Schema: Select schema
Table: Select table	Table: Select table

Relation Type

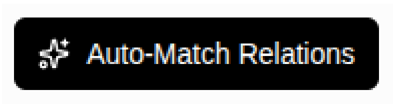
JOIN

Left Join Column	**Right Join Column**
Select column	Select column

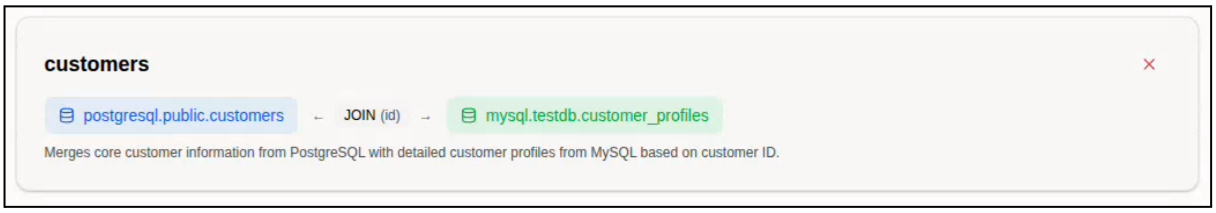
Description (optional)

Describe this relation...

2. **Automated Relationship Discovery (AI-powered):**
   - Loading state with progress indication
   - Results summary (suggestions found, relations created, errors)
   - Immediate visualization of newly created relationships



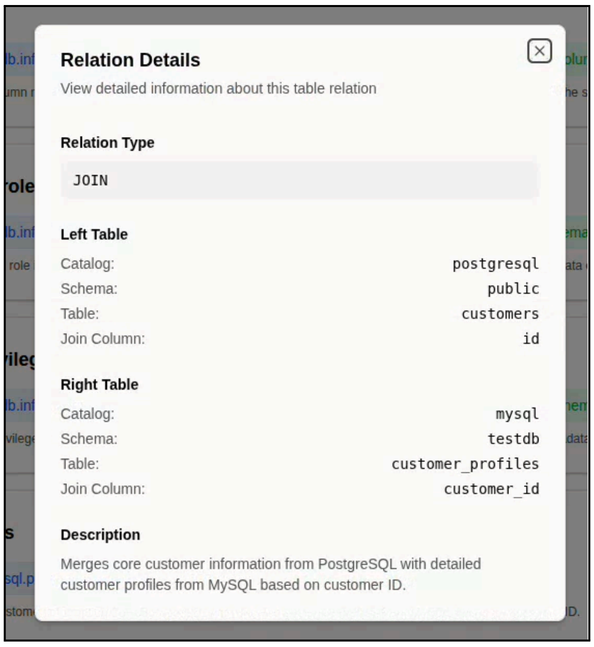
After creating the table relations, they should look like this:



customers

postgresql.public.customers ← JOIN (id) → mysql.testdb.customer_profiles

Merges core customer information from PostgreSQL with detailed customer profiles from MySQL based on customer ID.



Relation Details

View detailed information about this table relation

Relation Type

JOIN

Left Table

Catalog:	postgresql
Schema:	public
Table:	customers
Join Column:	id

Right Table

Catalog:	mysql
Schema:	testdb
Table:	customer_profiles
Join Column:	customer_id

Description

Merges core customer information from PostgreSQL with detailed customer profiles from MySQL based on customer ID.

6. Key Workflows

This section details the critical operational sequences that enable the platform's logical integration and querying capabilities.

6.1 Natural Language Query Workflow

This workflow enables non-technical users to explore federated data through conversational interaction, abstracting away query language complexity.

User Interface Flow:

The *ChatPage* implements a dual-state interface. Initially, users encounter a hero view with a centered search input and curated suggestion cards. Upon submitting the first query, the interface transitions to a threaded conversation view with message history, assistant responses, and a fixed input bar at the bottom.

Message Processing Pipeline:

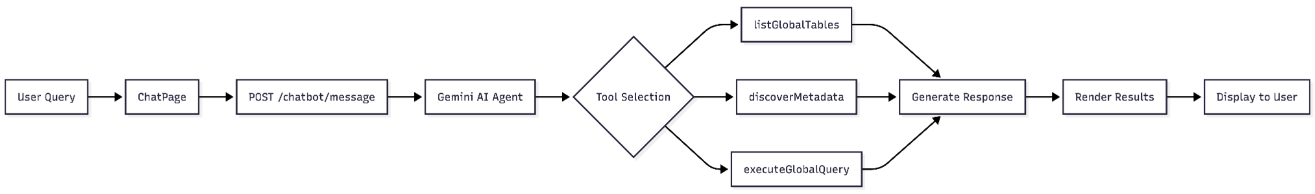
When a user submits a natural language query, the frontend dispatches a POST request to **/api/chatbot/message** containing the query text and complete conversation history. The backend forwards this to the *Gemini AI agent* configured with system instructions defining its role as a data exploration assistant. The agent has access to four specialized tools through function calling:

- **listGlobalTables():** Retrieves available global tables with descriptions
- **discoverMetadata()**: Explores catalogs, schemas, tables, and columns from physical sources
- **executeGlobalQuery(sql)**: Executes SQL against global tables, returning rows and metadata
- **getTableColumns(tableName)**: Fetches column details for query construction

The AI agent analyzes the natural language query, determines which tools to invoke, generates appropriate SQL queries, and executes them.

Response Rendering:

The backend returns a *ChatResponse* containing the assistant's natural language explanation and an array of *ToolResults*. The frontend employs polymorphic message rendering based on tool types.



6.2 AI-Powered Global Table Federation Workflow

This workflow automates the complex task of identifying relationships between physical tables and creating unified global table abstractions, leveraging AI to analyze schema patterns.

Automated Matching Trigger:

Users initiate automated matching through the Schema Studio interface or directly via POST to **/api/relations/auto-match** with optional parameters: *maxSuggestions* (default 5) controls result quantity, while *autoCreate* (boolean) determines whether suggestions are automatically instantiated or returned for review.

Metadata Context Assembly:

The backend constructs a comprehensive *MatchingContext* by traversing the entire metadata hierarchy - all catalogs, schemas, tables, and their column definitions (name, data type). This produces a complete snapshot of the physical data landscape. The context also includes existing *TableRelations*, enabling the AI to suggest nested relations that build upon previously created abstractions.

AI Analysis and Suggestion Generation:

The *GeminiMatchingStrategy* formats this metadata as JSON and constructs a detailed prompt instructing the AI to identify table relationships. The prompt distinguishes between two relation types:

- **UNION Relations:** Combine tables representing the same entity with identical schemas
- J**OIN Relations:** Merge different representations of the same entity from different sources

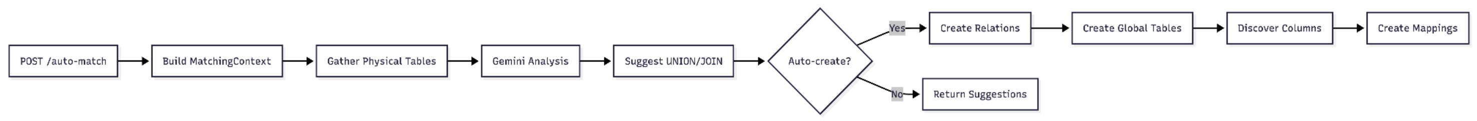
The AI responds with structured JSON containing relation suggestions, each including source tables, relation type, join columns (for JOINs), descriptive rationale, and a confidence score (0.0-1.0). Only suggestions with confidence ≥0.5 are returned.

Automatic Provisioning:

When *autoCreate* is enabled, the system processes each suggestion sequentially, creating a TableRelation record and automatically provisioning the supporting infrastructure:

1. **Global Table Creation**: A *GlobalTable* is created using the relation name
2. **Column Discovery:** The system queries Trino to discover columns from the first physical table in the relation
3. **Global Column Creation**: Each discovered column becomes a *GlobalColumn* with auto-generated descriptions
4. **Column Mapping:** ColumnMappings are created linking global columns to their physical counterparts across all tables in the relation

The system handles naming conflicts by appending version suffixes and validates that all referenced physical tables exist before creating relations. The *AutoMatchResponse* returns both the original suggestions (with confidence scores and rationale) and the successfully created relations.



6.3 Metadata Discovery & Synchronization Workflow

This workflow establishes and maintains the platform's understanding of available data sources, creating the foundation for query federation and global table abstraction.

Initial Discovery

Discovery is triggered via POST to **/api/sync**, initiating a hierarchical metadata crawl through Trino's federation layer. The *MetadataSync* service orchestrates this multi-level traversal:

- **Catalog Discovery:** Queries Trino's *information_schema* to enumerate configured catalogs (postgresql, mysql, mongodb, ...)
- **Schema Discovery:** For each catalog, discovers available schemas
- **Table Discovery:** Within each schema, enumerates tables (or collections for MongoDB)
- **Column Discovery:** For each table, retrieves column metadata including name, data type, and nullability

Metadata Storage

Discovered metadata is stored in the *MemoryMetadataStorage*, an in-memory registry using Go's *sync.RWMutex* for thread-safe concurrent access. This in-memory approach provides microsecond-latency access to schema information, eliminating the need to query Trino for every UI interaction or query translation operation.

Metadata Utilization

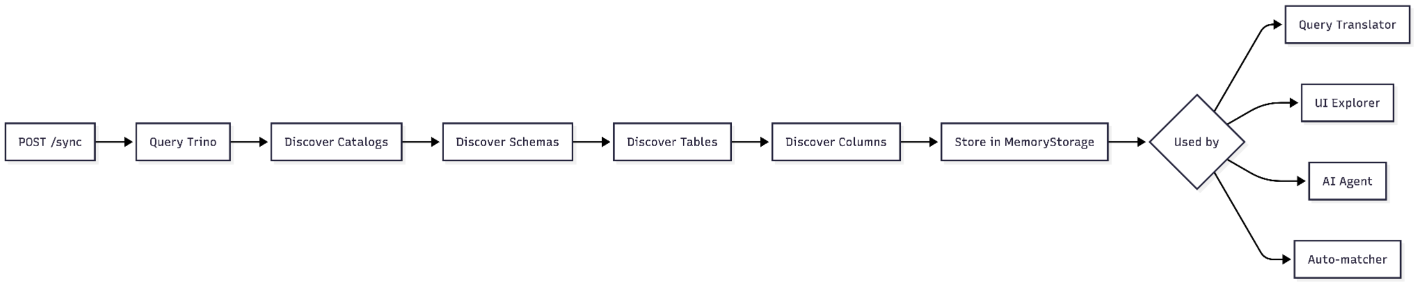
Once synchronized, the metadata registry powers multiple platform capabilities:

- **Query Translation:** The translator validates column references, resolves data types, and generates correct SQL syntax
- **UI Schema Explorer:** The interface renders the c**atalog/schema/table/column** hierarchy without additional Trino queries
- **AI Agent Context:** The chatbot tools access metadata to answer schema questions

- **Automated Matching:** The matching system analyzes column schemas to identify relationship candidates

Incremental Synchronization

The sync endpoint supports re-execution to capture schema changes in connected data sources. When invoked, it performs a fresh discovery traversal, updating the in-memory registry with new tables, modified schemas, or removed objects. The synchronization is non-blocking - queries continue executing against the current metadata snapshot while the sync updates the registry in the background. The sync process is fault-tolerant, continuing to process remaining catalogs even if individual sources become unavailable.



7. Setup & Deployment

The platform is containerized for simplified deployment. Data sources, Trino cluster, and backend API run in Docker containers, while the frontend operates as a Vite development server or static build.

7.1. Prerequisites

- **Docker & Docker Compose**: Container orchestration for infrastructure components
- **Node.js 18+:** Frontend development and build tooling
- **Gemini API Key:** Required for AI-powered features (chatbot, automated matching)

7.2. Quick Start

A startup script automates the entire stack initialization: **./start.sh**

- The script orchestrates the following sequence:
    1. **Docker Network**: Creates isolated trino-network for inter-service communication

2. **Data Sources**: Launches PostgreSQL, MySQL, and MongoDB with initialization scripts
3. **Trino Cluster**: Starts coordinator and three worker nodes with catalog configurations
4. **Backend API**: Builds and launches the Go application server
5. **Frontend**: Installs dependencies and starts the Vite development server

Once complete, services are accessible at:

- **Frontend UI**: http://localhost:5173
- **Backend API**: http://localhost:8081
- **Trino Coordinator**: http://localhost:8080

7.3. Configuration

Environment Variables (set in shell or **.env** file):

- **GEMINI_API_KEY:** Google Gemini API key for AI features (required)
- **PORT**: Backend server port (default: 8081)
- **TRINO_PORT**: Trino coordinator port (default: 8080)

Catalog Configuration (**trino/coordinator-config/catalog/*.properties**):

- Each .properties file defines a connector to a data source. Example for PostgreSQL:
  - connector.name=postgresql
  - connection-url=jdbc:postgresql://postgres:5432/testdb
  - connection-user=testuser
  - connection-password=testpass

Adding new data sources requires creating a new catalog properties file and restarting the Trino cluster.

Frontend Configuration (**interface/data-sync/src/lib/api.ts**):

- The API base URL defaults to **/api** (proxied in development). For production deployments, update the **API_BASE** constant to point to the backend server.

8. Limitations

8.1. Query Translation Scope

The translator supports common query patterns but lacks advanced SQL features: subqueries, window functions, CTEs, and complex aggregations. Query optimization is minimal - the system generates syntactically correct SQL without execution plan optimization.

8.2. Data Modification Constraints

No Global Table Writes: The platform is read-only for global tables. Users cannot INSERT, UPDATE, or DELETE data through global table abstractions - all write operations must target physical tables directly through their native interfaces (psql, mysql client, mongosh).
No Type Coercion in Mappings: Column mappings require exact data type matches. The system cannot map compatible but different types (e.g., bigint in PostgreSQL to int in MySQL, or varchar(100) to varchar(255)). Automated matching may fail to identify valid relationships due to minor type differences, requiring manual mapping configuration.

8.3. Data Source Management

No Runtime Catalog Addition: Adding new data sources requires modifying Trino catalog configuration files and restarting the Trino cluster - no API exists for dynamic catalog registration. This makes real-time data source onboarding impractical for multi-tenant or rapidly evolving environments.

9. Conclusion

The Data-Sync platform addresses the fundamental challenge of querying data distributed across heterogeneous storage systems without the operational overhead and latency of traditional ETL pipelines. By implementing a three-layer architecture - query federation through Trino, intelligent abstraction via global tables, and conversational access through AI agents - it transforms complex multi-database operations into intuitive user interactions.
The platform's key innovations extend beyond basic federation. AI-powered automated matching analyzes physical schemas to suggest and provision global table abstractions, eliminating manual relationship configuration. The query translation engine seamlessly

handles UNION and JOIN operations across different database systems, presenting federated data as unified entities. Natural language querying removes the technical barrier entirely, enabling business users to explore distributed data through conversation rather than SQL syntax.

This architecture demonstrates that data integration need not require data movement. By federating queries at runtime and abstracting complexity through global tables, organizations can maintain data in its native storage while providing unified access. The result is a scalable, transparent solution that breaks down data silos without creating new ones - making distributed data as accessible as a single database, and as queryable as a conversation.