AI Data Analyst

System Documentation

A natural-language interface over a retail sales database. A user asks a business question in plain English; the system turns it into a safe, validated SQL query, runs it against real data, and comes back with results, a chart, and a written business insight, instead of a raw table of numbers.

This document is written for two audiences at once: if you're non-technical, read the **Business Flow** and **Guardrails** sections and skip the more technical ones. If you're technical, the **Tech Stack**, **Architecture**, and **Data Model** sections describe the system design in detail.

Contents

1 What This System Does

Think of it as a data analyst who is on call 24/7, always shows their work, and refuses to guess when a question is unclear.

A user types something like *"Which products had the highest revenue growth last quarter, and which region drove it?"* into a chat box. Behind the scenes, the system moves through eight stages, shown below:

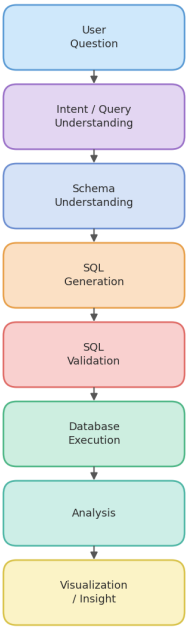

Figure 1: End-to-end processing pipeline, from question to insight.

1. Decides whether the question is answerable as asked, or too vague to answer responsibly.
2. Writes a SQL query that reflects the business's own definitions of terms like "revenue" and "quarter", not generic guesses.
3. Checks that query against a strict rulebook before it's allowed anywhere near the database.
4. Runs it against the real sales data (read-only - it is physically incapable of changing anything).
5. Reads the results and explains, in plain sentences, what they mean for the business.
6. Picks a sensible way to visualize the answer.
7. Remembers the conversation, so a follow-up like "compare that to last quarter" doesn't require repeating context.
8. Logs every single request - answered, blocked, or failed - for audit purposes.

2 AI Tech Stack (Technical)

Layer	Technology	Notes
Backend framework	**FastAPI** (Python)	A single question-answering endpoint, plus the web interface.
LLM provider	**Azure OpenAI**	Two model roles, both configured via environment settings (no hardcoded model names).
Reasoning model	A GPT-class deployment	Used for SQL generation and result analysis, the two tasks that need real reasoning.
Fast/cheap model	A smaller GPT-class deployment	Used only for the lightweight ambiguity classification step.

Layer	Technology	Notes
LLM call pattern	JSON-mode constrained completions	Deterministic settings for SQL generation and ambiguity classification; slightly higher creativity for the narrative insight.
SQL safety validation	**sqlglot** (SQL parser/AST library, not regex or string matching)	Parses the LLM's SQL into a real syntax tree and validates it structurally.
Database	**SQLite**, opened through a **read-only connection**	The process cannot write to the database at the driver level, independent of anything the LLM generates.
Execution safety net	Row-fetch cap + wall-clock query timeout	Defense-in-depth in case an expensive query slips past validation.
Chart selection	Plain heuristic (**no LLM call**)	Picks KPI / bar / line / table purely from the shape of the result set (row count, numeric vs. text columns, date-like column names).
Conversation memory	In-process, keyed by session, last 10 turns kept, last 4 fed into prompts	Not persisted to disk; resets if the server restarts.
Audit logging	Separate audit database	Every request is logged regardless of outcome (answered, blocked, errored).
Frontend	Vanilla JS + Chart.js + hand-written CSS	No framework, no build step.

Why two models instead of one: ambiguity classification runs on every single question and doesn't need deep reasoning, so a cheap, fast model keeps latency and cost down. SQL generation and business analysis genuinely need a stronger reasoning model to get joins, date logic, and business definitions right.

Why sqlglot instead of trusting the LLM's SQL directly: an LLM can be instructed not to write `DROP TABLE` or invent a column, but instructions are not a guarantee. sqlglot gives a second, independent, non-LLM layer that structurally parses the query and checks it against the *real, live-introspected* database schema, so even a fully "jailbroken" or hallucinating model output cannot execute anything unsafe.

3 Architecture / Request Flow

The diagram below shows how a single request moves through the system's layers: Presentation, Application, AI/Intelligence, Data, and Governance:

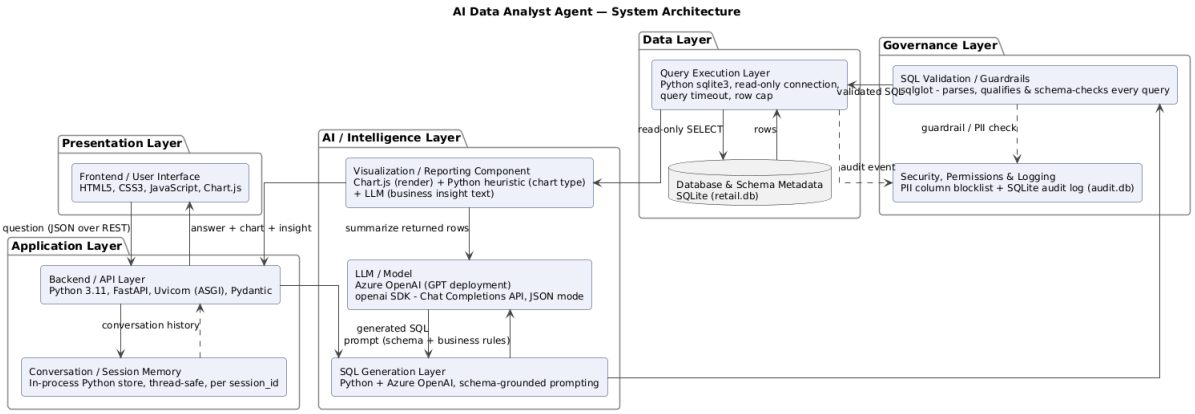

Figure 2: System architecture: presentation, application, AI, data, and governance layers.

4 Data Model

The system reads from a small, realistic retail schema: four tables, no synthetic "AI-only" fields.

Table	Key Columns
Customers	ID, name, email *(restricted PII, never selectable)*, region, signup date
Products	ID, name, category, unit price
Orders	ID, customer ID, order date, status
Order Items	ID, order ID, product ID, quantity, unit price *(charged price, may differ from catalog)*

Relationships: a customer places many orders; an order contains many order items; each order item references one product.

4.1 Business Definitions the AI Is Told to Use (Not Left to Guess)

These are injected into every prompt automatically, so the AI applies the business's actual rules rather than a generic interpretation:

- **Revenue** = quantity × unit price, summed across items, counting only orders with a "completed" status (cancelled/refunded orders are excluded unless the question explicitly asks otherwise).
- **Quarter** = a calendar quarter (Q1 = Jan-Mar, etc.).
- **Region** always means the *purchasing customer's* home region, not a shipping or warehouse region.
- **Growth** between two periods = the change in revenue (absolute or percentage) from the earlier period to the later one.

5 Guardrails & Safety

At a glance: every query is read-only, checked against the real schema, stripped of PII, capped in size and runtime, and logged, no matter the outcome.

rowgray **Guardrail**	**What It Stops**
Read-only DB connection	Any write to the database, at the driver level
rowgray SELECT-only allowlist	`DROP` / `DELETE` / `UPDATE` / `ATTACH` / `PRAGMA`, and chained statements
Live schema allowlist	Hallucinated table or column names
rowgray Scope-aware column checks	Missed or false-flagged columns inside CTEs and subqueries
PII column blocklist	Exposure of restricted personal data, even via a valid query
rowgray Row limit	Unbounded, oversized result sets
Query timeout	Slow or overly expensive queries that pass validation
rowgray "Can't be answered" path	Fabricated answers when the data simply doesn't exist
Bounded retries	Infinite retry loops and raw error/stack-trace leaks
rowgray Full audit log	Any request, answered, blocked, or failed, going unrecorded

On "safe parameterization": in a typical app, user input gets inserted into a SQL template, so parameterized queries matter a lot. Here, the LLM generates the *entire* query itself, so there's no string template being filled with user text. The equivalent safety mechanism is the AST-based validation layer above: nothing the LLM outputs runs until it's been parsed and checked, structurally, against the real schema and rule set.

What's intentionally not implemented (this is a single-tenant analyst prototype, not a multi-user production system): user authentication, per-user roles/permissions, and rate limiting on LLM calls. The restricted-column mechanism is designed so table/column-level rules for additional roles could be added later without redesigning the validator.

6 Conversation Memory & Follow-Ups

Each browser session gets its own identifier, which resets if the tab is closed or "New chat" is clicked. The last 10 turns of that session are kept in memory; the last 4 are formatted and fed into both the ambiguity check and the SQL generation prompt.

This is what makes a follow-up like *"Compare that to the previous quarter"* work: the model sees the prior question, the SQL that answered it, and the insight given, and resolves "that" and "the previous quarter" against that context rather than treating the follow-up as a fresh, standalone question.

Because this memory lives only in server process memory, it is **not** persisted: restarting the server clears every active conversation.