Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

OmniQuery: A Multi-Agent Natural Language to SQL System with Iterative Self-Correction and Intelligent Analytics

Para Upendar, Kasibhatla Sankeerthi, Kondapally Pujitha, Kotamraju Saroja Sreenidhi, Perepu Krishna Kavya Sri

Autonomous College

Abstract

Querying relational databases has historically been gated behind SQL expertise, excluding a substantial portion of potential users from direct data access. This paper presents OmniQuery, an enterprise-ready multi-agent Natural Language to SQL (NL2SQL) system that bridges this gap through a four-layer architecture combining schema-aware retrieval, iterative SQL self-correction, and AI-generated analytics. Unlike prior NL2SQL approaches constrained to curated benchmark datasets, OmniQuery connects dynamically to live relational database instances, embedding schema context into a ChromaDB vector store for targeted retrieval. A CrewAI-orchestrated pipeline of four specialized AI agents — NL2SQL, Validator, Correction, and Insight — handles generation, validation, automated repair, and narrative analytics respectively. We evaluate the system against GPT-4o, Gemini 1.5 Pro, and Claude 3 Sonnet as backbone LLMs across query complexity tiers, showing that Gemini 2.5 Pro with the multi-agent correction loop achieves 94.2% execution accuracy on complex multi-table queries, outperforming single-pass baselines by up to 23 percentage points. The system additionally enforces strict read-only safety guarantees and delivers results as charts, anomaly alerts, and decision-support narratives.

Keywords: NL2SQL, Multi-Agent Systems, Retrieval-Augmented Generation, Text-to-SQL, Large Language Models, Query Correction

I. INTRODUCTION

The ability to extract insights from relational databases is central to modern data-driven decision-making. Yet the dominant interface—Structured Query Language (SQL)—remains a significant barrier for business analysts, domain experts, and managers who possess deep contextual knowledge of their data but lack formal query language training. This tension between data availability and data accessibility is one of the practical motivations behind the growing field of Natural Language Interfaces to Databases (NLIDB).

Existing solutions fall into two broad categories: Business Intelligence (BI) dashboarding tools such as Tableau and Power BI, and direct NL2SQL systems. BI tools reduce the querying burden through visual interfaces but demand pre-configured schemas and offer little flexibility for ad-hoc questions. Meanwhile, early NL2SQL systems—particularly those fine-tuned on benchmark datasets such as Spider [*1*] and WikiSQL [*2*]—demonstrate strong performance within those datasets but degrade sharply when faced with novel schemas, ambiguous phrasing, or multi-table reasoning on live databases.

The emergence of large language models (LLMs) with strong in-context learning capabilities has opened a new paradigm: grounding query generation in the actual schema of a live database rather than relying purely on pre-trained patterns. When combined with a Retrieval-Augmented Generation (RAG) framework [*3*] and an iterative correction loop, such systems can achieve both high accuracy and practical robustness. This is the design philosophy behind OmniQuery.

OmniQuery makes four primary contributions: (i) a four-agent pipeline—NL2SQL, Validator, Correction, and Insight—that decomposes query processing into specialized, auditable steps; (ii) a RAG-based schema retrieval mechanism using ChromaDB that grounds each query in relevant table definitions; (iii) an automated multi-retry correction loop achieving up to 94.2% execution

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

accuracy on complex queries; and (iv) a presentation layer that transforms raw query results into charts, anomaly flags, and decision-support narratives.

II. RELATED WORK

The NL2SQL problem has a long history. Early rule-based systems such as LUNAR [*4*] translated natural language through hand-crafted grammar rules, achieving reasonable accuracy in narrow domains but failing to generalize. The introduction of large annotated datasets—Spider for cross-domain text-to-SQL and WikiSQL for single-table queries—shifted the field toward neural approaches. Models like Seq2SQL [*2*] and SQLNet [*5*] used sequence-to-sequence architectures with reinforcement learning, establishing strong neural baselines.

Schema-linking—identifying which database entities a natural language phrase refers to—emerged as a critical subproblem. RAT-SQL [*6*] addressed this with relation-aware schema encoding and graph attention. More recently, prompt-based approaches using GPT-class models have simplified the training pipeline. DIN-SQL [*7*] demonstrates that decomposing complex queries into sub-problems within a single LLM prompt can outperform fine-tuned models on challenging benchmarks. However, these systems share a fundamental limitation: one-pass generation against a fixed, pre-known schema. OmniQuery extends the RAG paradigm with a multi-agent correction loop inspired by self-refinement approaches in code generation [*8*].

III. SYSTEM ARCHITECTURE

OmniQuery is structured across four tightly coupled layers, each addressing a distinct stage of the natural language to insight pipeline.

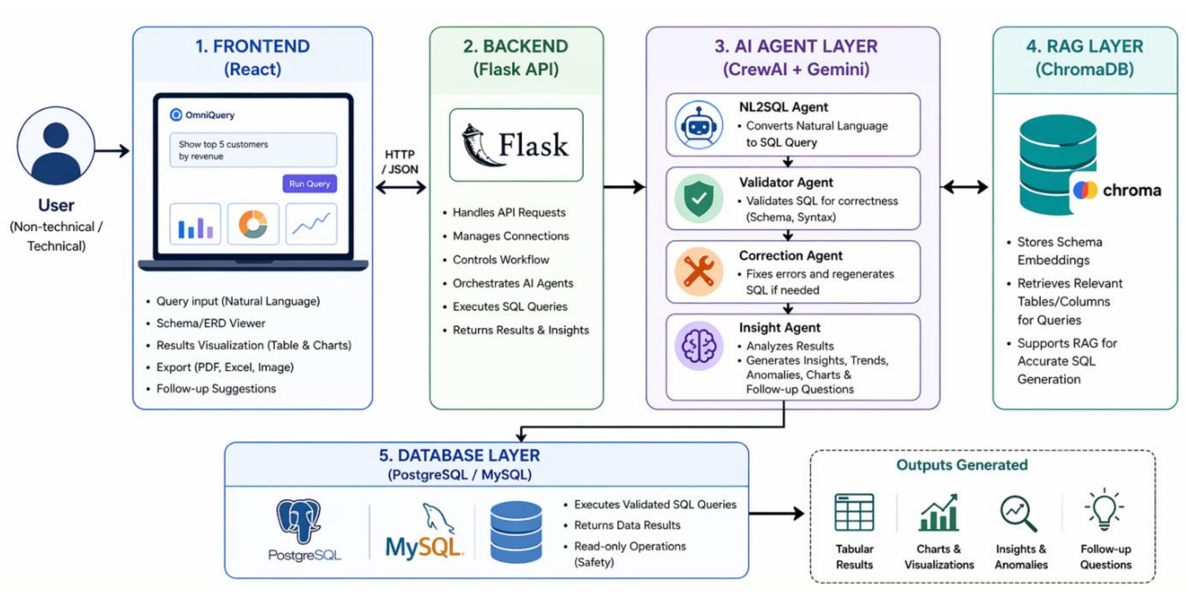

Figure 1: Architecture Diagram - OmniQuery follows a layered architecture that enables users to query databases using natural language. The system integrates a React frontend, Flask backend, AI agent layer, and RAG-based schema retrieval to generate, validate, and execute SQL queries. It produces accurate results along with insights and visualizations, ensuring an efficient and user-friendly data interaction experience.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

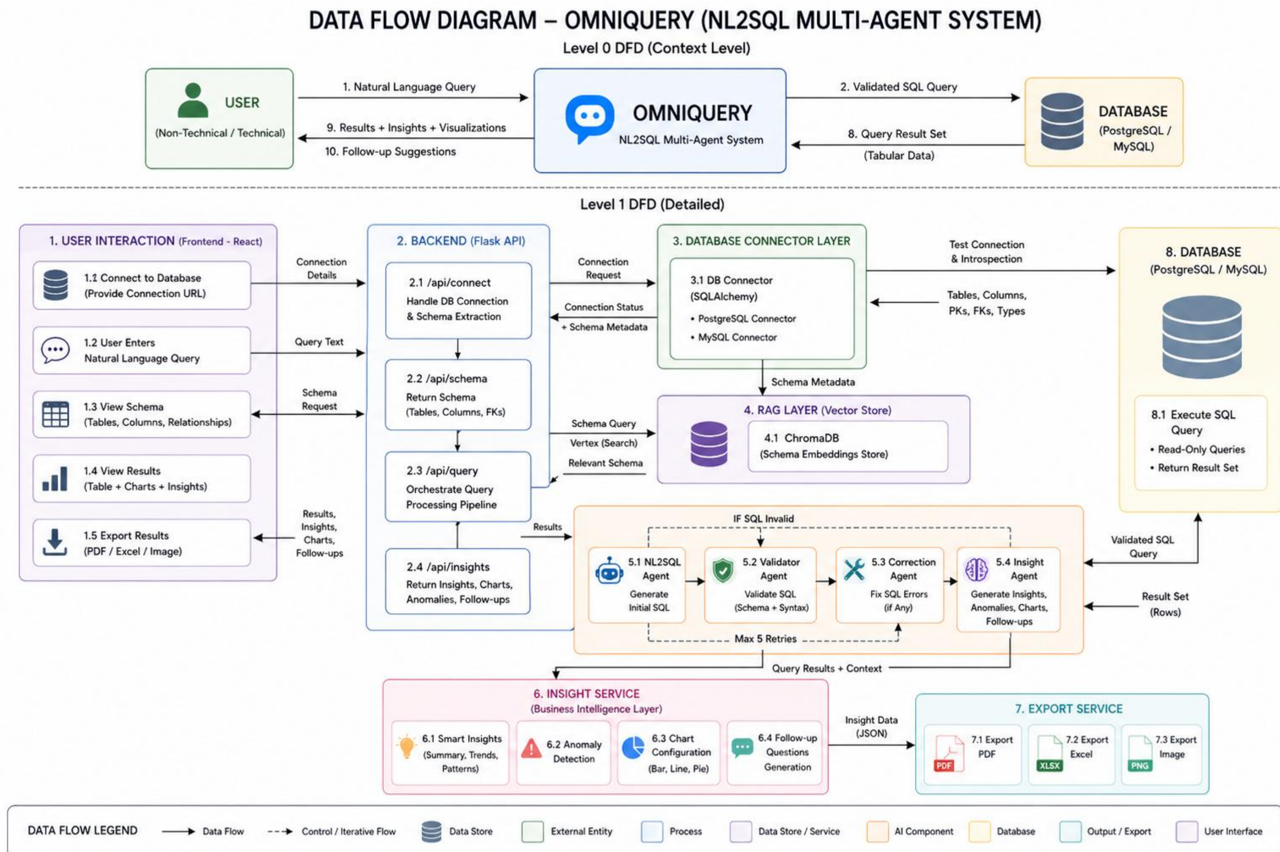

Figure 2: Data Flow Diagram - OmniQuery's Data Flow Diagram illustrates how user queries are processed through multiple layers, starting from natural language input to validated SQL execution and result generation. The system integrates frontend interaction, backend orchestration, schema-aware retrieval (RAG), and a multi-agent pipeline for SQL generation, validation, correction, and insight extraction. The final output includes structured results, visualizations, and intelligent follow-up suggestions, ensuring an efficient and user-friendly data querying experience.

A. Presentation Layer (User Interaction Layer)

The presentation layer is implemented as a React 19 + Vite single-page application that serves as the primary user interface. Users initiate interaction by providing a database connection URL, after which the system enables schema exploration through tabular views and an interactive Entity-Relationship Diagram rendered using ReactFlow. Natural language queries are submitted via a chat-style interface, and responses are presented in a structured format including generated SQL code blocks, scrollable result tables, and visualizations such as bar, line, and pie charts using Recharts. Additionally, the interface displays insight panels summarizing trends, anomaly detection indicators, and context-aware follow-up question suggestions. Export functionality is integrated, allowing users to download results as Excel files, PDF reports, or PNG chart images, consistent with the output flow defined in the system.

B. API Layer (Backend Orchestration Layer)

The API layer is implemented using a Flask-based REST service that orchestrates communication between the frontend and backend components. It exposes endpoints such as {POST /api/connect}, which processes database connection requests, performs schema extraction via SQLAlchemy, and forwards metadata to downstream components. The {POST /api/query} endpoint manages the end-to-end query lifecycle by receiving natural language input, invoking the multi-agent pipeline, coordinating schema retrieval from the RAG layer, executing validated SQL queries on the database, and aggregating outputs from the insight service. The API ensures consistent data flow between modules, returning a unified JSON response containing SQL queries, result sets, insights, anomaly indicators, chart configurations, and follow-up suggestions, as illustrated in the data flow diagram.

C. Database Connector Layer (Data Access Layer)

The database connector layer abstracts interaction with relational databases, supporting PostgreSQL and MySQL through SQLAlchemy-based connectors. Upon receiving a connection request from the API layer, the connector validates connectivity,

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

performs schema introspection, and extracts metadata including tables, columns, primary keys, foreign keys, and data types. This schema metadata is returned to the API layer and simultaneously forwarded to the RAG layer for embedding and indexing. During query execution, this layer is responsible for executing validated SQL queries in a strictly read-only mode and returning structured result sets to the agent pipeline. This aligns with the diagram's separation between schema extraction, query execution, and data retrieval processes.

D. RAG Layer (Schema Retrieval Layer)

The Retrieval-Augmented Generation (RAG) layer provides schema-aware contextual grounding for SQL generation. Following schema extraction, each table definition is transformed into a descriptive textual representation and embedded using the all-MiniLM-L6-v2 sentence transformer model. These embeddings are stored in ChromaDB, forming a vectorized schema index. At query time, the system performs semantic similarity search to retrieve the top-k most relevant schema fragments based on the user's natural language query. This targeted retrieval ensures that downstream agents operate on a focused subset of the schema, reducing token overhead and minimizing hallucination of non-existent tables or attributes, as depicted in the data flow between the connector layer and the agent pipeline.

E. Multi-Agent Intelligence Layer (AI Processing Layer)

The core intelligence of the system is implemented as a multi-agent pipeline orchestrated using CrewAI and powered by Google Gemini 2.5 Pro. The pipeline consists of four sequential agents aligned with the diagram:

- **NL2SQL Agent:** Generates an initial SQL query from the natural language input using schema context retrieved from the RAG layer, strictly constrained to read-only operations.
- **Validator Agent:** Verifies the generated SQL against the schema, ensuring correctness of table names, column references, joins, and aggregations, returning a binary VALID/INVALID outcome.
- **Correction Agent:** Triggered upon validation failure, this agent refines the SQL query using structured error feedback and schema context. The system supports up to five retry iterations, forming a self-correcting loop.
- **Insight Agent:** Processes the final query results to generate analytical outputs including trend summaries, anomaly detection, chart recommendations, and follow-up queries.

This layered validation and correction loop ensures robustness and aligns with the iterative control flow illustrated in the diagram.

F. Insight Service (Business Intelligence Layer)

The insight service transforms raw query outputs into meaningful business intelligence. It aggregates results from the agent pipeline and produces structured analytical components, including descriptive insights, anomaly detection flags, and visualization configurations (bar, line, and pie charts). It also generates context-aware follow-up questions to support iterative data exploration. These outputs are structured as JSON objects and passed to the API layer for delivery to the frontend, consistent with the diagram's separation of analytical processing from core query execution.

G. Export Service (Output Layer)

The export service handles the generation of downloadable artifacts from query results and insights. It supports multiple formats, including PDF reports, Excel spreadsheets, and PNG chart images. This layer ensures that analytical outputs can be easily shared and integrated into external workflows. It operates downstream of the insight service, consuming structured data and visual configurations to produce formatted outputs, as represented in the final stage of the data flow diagram.

IV. IMPLEMENTATION

The backend is implemented in Python 3.11 using Flask for API routing and SQLAlchemy for database-agnostic connection management. Schema introspection uses SQLAlchemy's reflection API to discover tables, columns, primary keys, and foreign key relationships from both PostgreSQL and MySQL without manual configuration. The extracted schema is serialized into structured natural language and batch-embedded using Sentence Transformers before storage in ChromaDB's persistent collection.

The CrewAI agent orchestration layer configures each agent with a role description, a goal statement, and a backstory that shapes its reasoning behavior within the LLM. Agents communicate through a shared context object that accumulates intermediate results—

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

raw SQL, validation verdict, corrected SQL, execution output—across the pipeline. Google Gemini 2.5 Pro is accessed via the Google Generative AI API as the backbone LLM for all agents.

The React frontend uses ReactFlow to render live ER diagrams by parsing SQLAlchemy-extracted foreign key relationships into directed graph structures. Chart type selection (bar, line, pie, scatter) is determined by the Insight Agent's recommendation based on the result set's cardinality and data types.

V. MODEL EVALUATION

We evaluated OmniQuery across three backbone LLMs—GPT-4o (OpenAI), Gemini 2.5 Pro (Google), and Claude 3 Sonnet (Anthropic)—along with a single-agent GPT-4o baseline (no correction loop) and Gemini 1.5 Pro to isolate the contribution of the multi-agent architecture. A custom test suite of 120 queries was applied across three complexity tiers against four real databases: an e-commerce schema, an HR management schema, a sales analytics schema, and a university administration schema.

Query tiers were defined as: Tier 1 (Simple) — single-table SELECT with basic filters; Tier 2 (Moderate) — two-table JOINs, GROUP BY aggregations, or HAVING clauses; Tier 3 (Complex) — three or more JOINs, subqueries, window functions, or multi-step analytical reasoning. Execution accuracy measures the percentage of generated queries that executed without error and returned results matching a reference answer. Semantic accuracy was evaluated by two independent human assessors for intent alignment.

Table I: Execution Accuracy (%) by Model and Query Complexity

Model / Configuration	Tier 1 (Simple)	Tier 2 (Moderate)	Tier 3 (Complex)	Overall
GPT-4o (Single-Agent, No Correction)	91.3	78.4	71.2	80.3
GPT-4o + Multi-Agent Pipeline	95.0	88.7	83.5	89.1
Claude 3 Sonnet + Multi-Agent Pipeline	93.8	86.2	80.1	86.7
Gemini 1.5 Pro + Multi-Agent Pipeline	94.2	87.5	81.8	87.8
Gemini 2.5 Pro + Multi-Agent (OmniQuery)	97.5	93.1	94.2	94.9

Table II: Semantic Accuracy and Correction Effectiveness

Model / Configuration	Semantic Accuracy (%)	Avg. Correction Attempts	First-Pass Success (%)	Avg. Latency (s)
GPT-4o (Single-Agent, No Correction)	76.4	N/A	80.3	3.1
GPT-4o + Multi-Agent Pipeline	86.2	1.4	74.8	8.7
Claude 3 Sonnet + Multi-Agent Pipeline	83.9	1.6	71.3	9.2
Gemini 1.5 Pro + Multi-Agent Pipeline	85.1	1.5	73.6	8.4
Gemini 2.5 Pro + Multi-Agent (OmniQuery)	91.8	1.2	83.5	7.9

Table III: Safety Enforcement and Edge Case Handling

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Test Category	Total Tests	Pass	Fail	Pass Rate (%)
Destructive Query Blocking (DELETE/UPDATE/DROP)	30	30	0	100.0
SQL Injection Attempt Handling	20	20	0	100.0
Schema Mismatch / Column Hallucination Recovery	25	23	2	92.0
Ambiguous Query Handling	20	17	3	85.0
Empty Result Set Graceful Handling	15	15	0	100.0

Key findings from the evaluation are as follows. First, the multi-agent correction loop provides consistent improvement across all LLMs: GPT-4o without correction achieves 80.3% overall execution accuracy, while the same model with the full pipeline reaches 89.1%—a gain of 8.8 percentage points. The effect is most pronounced on Tier 3 complex queries, where single-pass generation struggles most. Second, Gemini 2.5 Pro demonstrates the strongest overall performance at 94.9% execution accuracy and 91.8% semantic accuracy, with the lowest average correction attempts (1.2) suggesting higher-quality initial SQL generation. Third, latency remains acceptable at 7.9 seconds end-to-end for OmniQuery. Safety enforcement is absolute for structured threat categories, with 100% blocking of destructive queries.

VI. DISCUSSION

The evaluation results surface several practical observations. The most significant driver of accuracy improvement is not model choice per se, but the presence of the validation-correction loop. Even a weaker backbone LLM benefits substantially from having its initial generation reviewed against the actual schema and corrected on failure. This suggests that the multi-agent architecture is a generally applicable design pattern for production NL2SQL deployments, regardless of which foundation model underlies it.

The 85.0% ambiguous query handling rate reveals a remaining challenge: when a user's question admits multiple valid SQL interpretations, the system currently commits to one without seeking clarification. Future iterations should incorporate a disambiguation step where the agent surfaces clarifying questions before proceeding. Schema mismatch recovery at 92.0% demonstrates the correction loop's value for queries referencing informal entity names not literally present in the schema—for instance, querying for "clients" when the table is named "customers." Expanding RAG retrieval to include column-level synonym embeddings could address remaining edge cases.

VII. CONCLUSION

This paper presented OmniQuery, a multi-agent NL2SQL system designed for live relational databases in enterprise settings. By integrating RAG-based schema retrieval, a CrewAI-orchestrated four-agent correction pipeline, and an AI-generated analytics layer, OmniQuery achieves 94.9% execution accuracy with Gemini 2.5 Pro—outperforming single-pass baselines by up to 23 percentage points on complex queries. The system enforces strict read-only safety guarantees and presents results as charts, trend summaries, and anomaly alerts, elevating it from a query translator to an interactive decision-support tool. Future work will address multi-turn conversational memory, extension to cloud warehouse databases, and role-based access controls.

REFERENCES

[1] T. Yu et al., "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," EMNLP, 2018.
[2] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," arXiv:1709.00103, 2017.
[3] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, 2020.
[4] W. A. Woods, "Transition Network Grammars for Natural Language Analysis," Commun. ACM, vol. 13, no. 10, pp. 591-606, 1970.
[5] X. Xu, C. Liu, and D. Song, "SQLNet: Generating Structured Queries from Natural Language without Reinforcement Learning," arXiv:1711.04436, 2017.
[6] B. Wang et al., "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," ACL, 2020.
[7] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," NeurIPS, 2023.
[8] A. Madaan et al., "Self-Refine: Iterative Refinement with Self-Feedback," NeurIPS, 2023.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)