

SynapseSQL Enterprise: A Conversational, Guarded Natural-Language Interface for Relational and NoSQL Databases

SHARATH M, DARSHAN V, BHAVYASHREE G

SynapseSQL Enterprise Project, India

GitHub Repository: SHARATH204-MAX/SynapseSQL-Enterprise

Abstract—

SynapseSQL Enterprise is a full-stack database intelligence application that lets users ask database questions in natural language and receive generated SQL, tabular results, charts, and a short explanation. It combines a React 19 client with a FastAPI backend, LangChain-based agent services, Groq-hosted language models, SQLAlchemy for relational access, PyMongo for MongoDB inspection, and Supabase for user profiles and query auditing. A central design choice is to inspect the live database schema before SQL generation and to supply that schema to the language model, instead of relying on a separate vector store. Server-sent events (SSE) stream status, generated SQL, and explanation tokens to the browser. Safety comes from authentication, role-based access control, request limits, destructive-query filters, and a read-only mode for the bundled SQLite workflow. This paper describes the architecture, query lifecycle, implementation, functional evaluation strategy, and current limitations of the project. It deliberately reports repository-supported behavior rather than synthetic accuracy or latency numbers.

Keywords—natural-language-to-SQL, conversational databases, text-to-SQL, FastAPI, React, LangChain, server-sent events, database security, role-based access control, schema grounding.

I. INTRODUCTION

Database systems are powerful, but using them often requires knowledge of SQL, table structures, and database-specific syntax. This creates a gap between the information users need and the interface through which they request it. SynapseSQL Enterprise addresses this gap by placing a conversational layer in front of several database engines. Users can type or speak a question; the application inspects the connected schema, asks a language model to generate SQL, executes the query, and presents the result as a table, chart, and explanation.

The project goes beyond text generation. It contains a React 19 frontend, a FastAPI service layer, database connection and introspection utilities, authentication, rate limiting, exports, query history, and Docker deployment. The repository documents support for SQLite, MySQL, and MongoDB, while Supabase PostgreSQL stores user profiles and query audit records. The default delivery model is a single FastAPI host that serves both the compiled React application and the API from one origin.

The engineering challenge extends beyond text-to-SQL accuracy. A useful system must ground generated queries in the actual schema, provide feedback while the model is

working, reject clearly destructive requests, and display the generated SQL so users can inspect it.

This paper makes five practical contributions: (1) a schema-grounded conversational database workflow; (2) an SSE-based streaming mechanism for interactive feedback; (3) a layered security design that combines filtering, execution checks, authentication, role-based access control (RBAC), rate limits, and read-only database access; (4) a connection model for

multiple database systems; and (5) a functional evaluation plan based on the implemented API and test suite rather than unsupported benchmark claims.

II. BACKGROUND AND RELATED WORK

Text-to-SQL research examines how natural-language questions can be converted into executable database queries. Early systems used supervised methods to map questions to SQL

[2]. More recent approaches use pretrained language models together with schema information to improve generalization. In SynapseSQL, schema awareness is essential because generated queries must use the table names, column names, and relationships that exist in the connected database.

Retrieval-augmented generation (RAG) [4] is a common method for providing external context to language models. The current SynapseSQL implementation instead inspects the live schema and collects table names, columns, data types, primary and foreign keys, row counts, and small samples. This information is inserted directly into the prompt. The repository describes this approach as schema-in-prompt grounding and identifies vector-based schema retrieval as a possible future improvement for larger databases.

At the web layer, FastAPI [6] provides asynchronous request handling and OpenAPI documentation, while Uvicorn runs the ASGI application. The frontend uses React and TypeScript, and Chart.js renders result visualizations. The browser receives SSE frames through a streaming client rather than a WebSocket connection. The repository documents SSE as a deliberate design choice because the main model response is streamed from the server to the client over HTTP.

III. SYSTEM ARCHITECTURE

A. Unified Application Structure

The application is organized around a single FastAPI host (Fig. 1). The browser receives the compiled React single-page application from the same service that exposes the API. API routes are separated from the static-file catch-all route, and the backend is organized into API, service, core, model, and data layers. This structure reduces deployment complexity because the production application is served from a single origin. The repository uses a multi-stage Docker build: the frontend is compiled first, and its distribution files are then copied into a slim Python runtime.

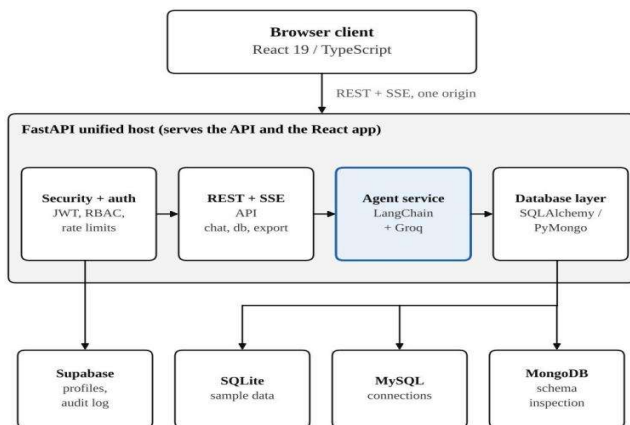


Fig. 1. High-level architecture of SynapseSQL Enterprise.

B. Query Processing Model

A request reaches the chat endpoint with the user's question, conversation history, database configuration, selected model, and read-only preference (Fig. 2). After authentication and rate-limit checks, the service applies the safety filter. If the request passes validation, the database manager inspects the live schema. The agent then generates SQL in the first language-model pass, and SQLAlchemy executes it for relational databases. If execution fails, the documented pipeline attempts one correction. A second model pass generates the user-facing explanation, which is streamed to the client.

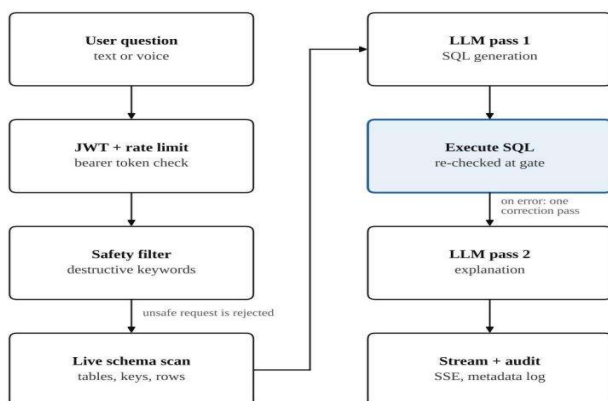


Fig. 2. Query lifecycle from user input to rendered result.

C. Schema Grounding

Schema inspection is central to the design. For relational databases, the system retrieves table names, columns, data types, primary keys, foreign keys, row counts, and sample records. This information gives the language model the database context it needs before generating SQL (Fig. 3). The bundled SQLite example contains STUDENT and DEPARTMENTS tables linked by a foreign key.

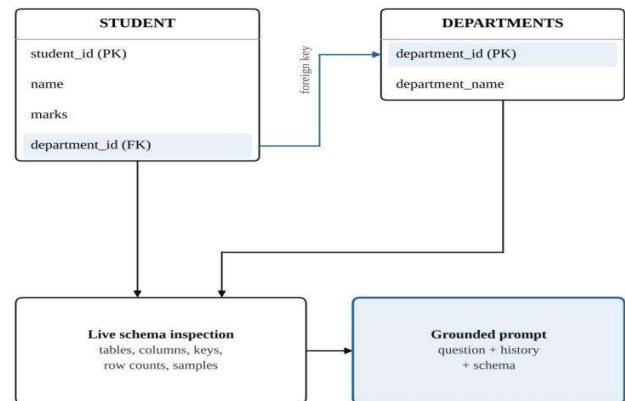


Fig. 3. Example schema grounding path using the bundled STUDENT-DEPARTMENTS data.

D. Streaming Response Protocol

The chat endpoint returns a `text/event-stream` response. Event types include status, thought, tool output, sql, token, done, error, and warning. These events allow the interface to indicate when schema inspection or query execution is in progress. The frontend uses a custom streaming client because the request includes POST data and authorization headers, which the native EventSource interface does not handle conveniently.

IV. IMPLEMENTATION

A. Backend Services

The backend is written in Python 3.11 and uses FastAPI with Uvicorn. LangChain and the Groq integration manage language-model calls. SQLAlchemy [7] supports relational schema inspection and query execution across database dialects, while PyMongo supports MongoDB schema inspection. Pydantic models define request and response structures. The export service uses pandas and openpyxl to support CSV and Excel files, as well as JSON output.

The agent service separates SQL generation from explanation. The first model pass generates SQL, and the second produces a readable explanation. This separation reflects the different requirements of query generation and conversational reporting. The generated SQL is displayed in the interface so that users can inspect, copy, or rerun it.

B. Frontend

The client uses React 19 and TypeScript. Its components include a chat interface, query cards, a data table, chart views, and voice input,

connection dialogs, authentication screens, query history, and API-key configuration. Chart.js supports bar, line, pie, and doughnut charts. The data grid provides search, sorting, pagination, and null-value handling. Together, these features make the system an interactive database workspace rather than a command-line SQL generator.

C. Database Support

SQLite is included as the sample database, and connection management is available for MySQL and MongoDB. The current MongoDB workflow focuses on schema inspection; the project roadmap identifies chat execution through a dedicated aggregation pipeline as future work. Therefore, database discovery and query execution are separate capabilities in the current implementation.

D. Authentication and Auditing

Authentication supports local/demo roles and can integrate with Supabase [8] and Google OAuth. JSON Web Tokens (JWTs) manage application sessions, while Supabase stores user profiles and query logs when configured. Audit records include the prompt, generated SQL, model, timing metadata, and row count; raw result rows are not stored. Row-level security protects the Supabase audit data.

V. FUNCTIONAL EVALUATION

A. Evaluation Approach

The project is evaluated as a software system rather than as a language-model benchmark. No unverified question-to-SQL accuracy, throughput, or latency figures are reported. The evaluation plan follows the behaviors described in the repository's automated tests and API documentation, including health checks, database inspection and execution, authentication, protected-route behavior, Supabase authentication gating, and rate-limit enforcement. Database tests include a safe SELECT query and a destructive DROP TABLE request that should be rejected by the safety filter.

B. Functional Coverage

TABLE I
FUNCTIONAL COVERAGE

Area	Observed / documented behavior	Evidence
Database	Schema discovery, connection testing, guarded execution; SQLite, MySQL and MongoDB paths	API + DB tests
NL→SQL	Schema-aware SQL generation, then execution; one correction attempt on error	Agent pipeline
Streaming	Status, SQL, token, completion, warning and error events over SSE	SSE protocol
Security	Keyword filtering, execution re-check, RBAC, JWT validation, rate limits, read-only SQLite	Architecture + tests
Presentation	Tables, charts, SQL card, explanation, history, CSV/Excel/JSON export	Frontend + export
Deployment	Multi-stage Docker, healthcheck, docker-compose, Render guidance	Docker + docs

C. Representative Test Cases

The repository describes tests for health status and database connectivity, as well as schema discovery for the bundled STUDENT and DEPARTMENTS tables. Other documented cases cover successful execution of a safe SELECT query, rejection of DROP TABLE, JWT issuance and role checks, rejection of unauthenticated requests, and rate-limit headers. These tests address core requirements for a conversational database tool: service availability, schema discovery, safe query execution, and rejection of unsafe or unauthenticated requests.

D. User-Facing Behavior

Users sign in, configure a database connection, and refresh the schema before submitting questions. They can ask about aggregation, filtering, ranking, or data distribution by text or voice. Results are presented in an explanation panel, a data table, a chart view, and an SQL card. The interface also provides a read-only control, model selection, a stop action, and conversation persistence in local storage.

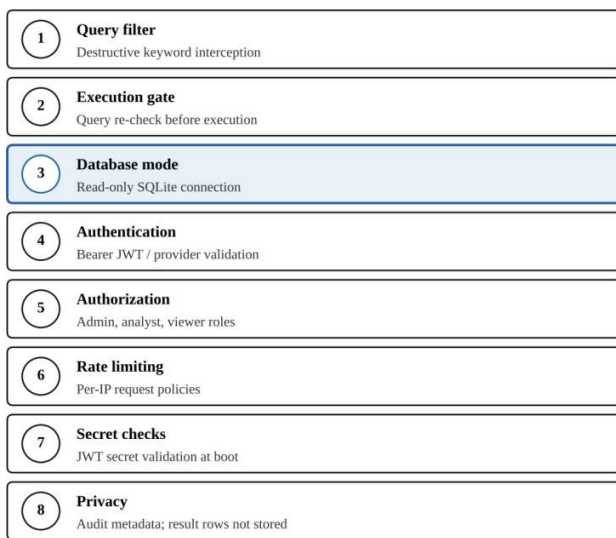
VI. DISCUSSION

A key strength of the implementation is the integration of schema inspection with query generation. A conversational interface may be easy to use but unreliable if the model lacks information about the actual database structure. Retrieving schema information immediately before generation grounds the prompt in the current connection and avoids the overhead of maintaining embeddings for small schemas.

A second design choice is the separation of SQL generation from explanation. The system displays the generated SQL as an inspectable artifact and produces the natural-language explanation separately. This makes the result easier to review and allows users to copy or rerun the query.

SSE is suitable for the current workload because the model response is primarily streamed from the server to the client. An HTTP stream is simpler than a bidirectional WebSocket channel for this use case and allows the client to display status updates and generated tokens without changing the database interaction model.

The security model uses multiple layers (Fig. 4). A regular-expression filter is fast and straightforward to implement, but it is not a complete SQL security boundary. The repository acknowledges this limitation and identifies SQL abstract syntax tree (AST) allow-listing, least-privilege credentials, and read replicas as possible hardening measures. In an enterprise deployment, model-generated output should be treated as untrusted, even when the request appears harmless [9].



Highlighted: the only layer enforced by the database itself.

Fig. 4. Defense-in-depth security layers implemented or documented by the project.

VII. LIMITATIONS AND FUTURE WORK

The project documents several limitations. MongoDB schema inspection is available, but conversational execution remains SQL-oriented. Support for custom MySQL ports is incomplete, and the safety layer relies on keywords and regular expressions rather than a SQL parser. Session tokens are stored in local storage; the roadmap proposes HttpOnly cookies to improve resistance to cross-site scripting (XSS). Handling large schemas also remains an open problem, and schema-based retrieval is proposed for databases with many tables.

Future work should prioritize controlled query execution, not only improvements to model output. A parser-backed SELECT allow-list, database accounts with minimal privileges, query-cost limits, and read replicas could reduce the impact of incorrect generated queries. If prompt size becomes a constraint, schema context could be extended with table descriptions, synonyms, example queries, and embeddings. A structured function-calling interface for a 'run_sql' tool could also reduce reliance on free-form generated text.

The next evaluation stage should use a fixed benchmark of natural-language questions paired with expected result sets for SQLite, MySQL, and PostgreSQL. Metrics should include execution success, result-set equivalence, unsafe-query rejection, response time, and recovery after SQL errors. This would allow model configurations to be compared without making unsupported accuracy claims.

VIII. CONCLUSION

SynapseSQL Enterprise presents a practical architecture for conversational database access. Its main workflow authenticates the user, inspects the live schema, generates SQL, executes the query, streams an explanation, and presents the results in several formats. The implementation also includes database adapters, role-based controls, rate limits, auditing, export functions, and containerized deployment.

The project is best regarded as an engineering prototype that incorporates several production-oriented design choices. Its documentation does not include a controlled benchmark, so claims about model accuracy or latency would be premature. The evidence supports the conclusion that the application brings together key components of a natural-language database interface while leaving several tasks for future work: stronger query authorization, schema grounding for larger databases, and measurement of end-to-end reliability.

APPENDIX A: IMPLEMENTATION DETAILS

The repository provides API routes for four main areas: authentication, chat and agent execution, database access, and data export (Table II). Separating these areas makes the system easier to test and allows the frontend to use the same backend in local deployments and when served as a compiled single-page application.

TABLE II
API ROUTES

Route	Purpose	Security / limit
/api/chat/stream	Natural-language query to an SSE stream of status, SQL, tokens and completion	Bearer; 25/min
/api/database/schema	Inspect tables, columns, keys, counts and samples	Optional auth; 60/min
/api/database/execute	Execute a submitted query after guardrail checks	Optional auth; 60/min
/api/auth/demo-login	Issue a JWT for the documented demo roles	30/min
/api/auth/google	Verify a Google ID token and establish authentication	30/min
/api/export/csv	Result rows as a CSV attachment	API endpoint
/api/export/excel	Result rows as an XLSX attachment	API endpoint
/api/export/json	Result rows as JSON	API endpoint

Query Execution Algorithm

Input: natural-language question Q , database configuration D , conversation history H , selected model M , and read-only policy.

- 1) Authenticate the request and apply the endpoint rate limit.
- 2) Check Q against the configured destructive-query filter.
- 3) Inspect D and collect schema information.
- 4) Build a grounded prompt from Q , H , and the inspected schema.
- 5) Generate SQL.
- 6) Normalize the generated SQL and execute it through the database manager.
- 7) If execution fails, attempt the documented single correction.
- 8) Pass the returned columns and rows to the explanation stage.
- 9) Stream the answer and display the SQL to the client.
- 10) Write audit metadata if Supabase is configured.

This algorithm makes database execution an explicit step. Rather than answering from an assumed table, the system constructs its response after attempting to execute a query against the connected database.

Data Representation

The frontend and backend exchange structured objects representing query requests, database configuration, result

columns and rows, the selected model, and stream events. The streaming protocol uses small JSON payloads within SSE frames. A final 'done' event can contain the answer, generated SQL, result data, model information, and execution-time metadata. Separate error and warning events report failures and model failover.

APPENDIX B: DEPLOYMENT AND SECURITY REVIEW

Deployment Sequence

The production build uses a two-stage Docker process. The first stage installs Node dependencies and builds the React frontend. The second stage uses a slim Python 3.11 image, installs the backend requirements, copies the backend source and compiled frontend, and starts Uvicorn. The image exposes a health endpoint and can be run with Docker Compose using a persistent volume for SQLite files. For Render deployment, the application uses the platform-provided 'PORT' value and the same health-check path. Because the runtime image does not require the Node toolchain, the browser and API can remain on the same origin, consistent with the single-port architecture.

Security Review

The repository describes eight defensive layers (Fig. 4): a filter for destructive query keywords; repeated validation at the execution boundary; a read-only connection for the bundled SQLite database; bearer-token authentication; admin, analyst, and viewer roles; rate limiting; validation that prevents startup with a missing or weak JWT signing key; and auditing that stores query metadata without persisting raw result rows.

These controls do not constitute a complete database security model. Keyword matching may miss syntactic variations, and an application-level filter is weaker than a parser-backed allow-list combined with least-privilege database credentials. The project identifies SQL AST enforcement and stronger session management as future work. In production, database-level permissions should remain the final authority.

Reproducibility

The repository documents three setup options: the hosted demo, Docker, and manual execution from source. The manual setup requires Python 3.10 or 3.11, Node.js 18 or later, and Git. The frontend is built with 'npm run build', the backend is started with Uvicorn, and tests are run with pytest. A sample SQLite database and initialization script are included, allowing the basic query workflow to be reproduced without an external database server.

Practical Research Value

The project is useful as an implementation study because its architecture separates language generation, database access, and application security. Rather than treating text-to-SQL as an isolated model task, it integrates the model into a conventional software system. Schema-retrieval methods can be changed without rewriting the user interface, the model provider can be replaced without rebuilding the database adapters, and security controls can be strengthened without changing the user-facing query language.

The next meaningful research step is a controlled evaluation with questions at different difficulty levels: simple filters, aggregations, joins, nested queries, ambiguous column names, and requests that must be rejected. Each answer should be checked against actual result sets rather than by comparing SQL strings alone. Security cases should be scored separately so that a system that returns a correct answer through an unsafe query is not counted as fully successful.

Final Implementation Summary

SynapseSQL Enterprise is an application pipeline rather than a standalone prompt. It separates the browser, API, agent, database, identity, and deployment layers. The main areas for further work are parser-based SQL authorization, schema grounding for larger databases, full MongoDB query execution, stronger session storage, and benchmark-based evaluation. Each represents a concrete engineering task for a future version.

REFERENCES

- [1] S. M., "SynapseSQL Enterprise — AI Database Intelligence," GitHub repository, 2026. [Online]. Available: <https://github.com/SHARATH204-MAX/SynapseSQL-Enterprise>
- [2] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," arXiv:1709.00103, 2017.
- [3] B. Peng et al., "Towards making a dependency parser aware of the task," in Proceedings of the Association for Computational Linguistics (ACL), 2023.
- [4] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. NeurIPS, 2020.
- [5] M. Zaharia et al., "Apache Spark: A unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.
- [6] FastAPI Documentation, 2026. [Online]. Available: <https://fastapi.tiangolo.com/>
- [7] SQLAlchemy, "SQLAlchemy 2.0 Documentation." [Online]. Available: <https://docs.sqlalchemy.org/>
- [8] Supabase, "Database and Row Level Security Documentation." [Online]. Available: <https://supabase.com/docs>
- [9] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications." [Online]. Available: <https://owasp.org/>