

Data Analyst Agent with Flowise using PS-SQL

Nitin Diwewar¹, Nikhil Kumar², Paras Tembhare³, Mitesh Baryerkar⁴,
Dr. Pragati Fatinge⁵

^{1,2,3,4}Department of Computer Science and Engineering, G H Raisoni College of Engineering and Management, Nagpur, India

⁵Assistant Professor, Department of Computer Science and Engineering, G H Raisoni College of Engineering and Management, Nagpur, India

Abstract

Traditionally, gaining access to relational databases has required a knowledge of Structured Query Language (SQL) and a clear understanding of the underlying database schemas, which presents a significant operational hurdle for people who are not technical. Although conversational Text-to-SQL systems based on Large Language Models (LLMs) provide a way of allowing natural language queries to be sent to databases, conventional implementations generally suffer from schema hallucination, ambiguous mappings between tokens and attributes, and failures when carrying out complex relational joins. In this paper, a conversational Data Analyst Agent is presented, the agent having been developed using the Flowise visual orchestration framework together with Phrasebased Schema Linking (PS-SQL). The architecture in question makes use of an autonomous Thought-Action-Observation loop which correctly matches natural language expressions to the various elements of the relational schema (such as tables, views, and attributes) in order to produce SQL queries that are both syntactically valid and ready for execution. The framework includes RetrievalAugmented Generation (RAG) so that it can refer to auxiliary unstructured enterprise documentation (for example, data dictionaries, PDFs, and CSV records) and makes use of Python libraries (Pandas, Matplotlib, Seaborn) to automate the process of analytical synthesis as well as the production of graphical dashboards. By maintaining conversational memory throughout multiple turns of interaction and by offering transparent and verifiable query execution, the proposed agent greatly reduces inaccuracies in schema linking, removes the need to rely on a dedicated database engineering team for regular reporting, and thus brings intelligent data exploration within reach of a wider audience. eliminates reliance on dedicated database engineering teams for routine reporting, and democratizes intelligent data exploration.

Keywords: Text-to-SQL, LLM Agent, Flowise, PS-SQL, Schema Linking, Retrieval-Augmented Generation (RAG), Automated Data Analytics.

I. INTRODUCTION

Modern businesses and institutional organizations place a great deal of trust in relational database management systems (RDBMS) when it comes to handling important transactional data, business intelligence, and operational records. Even though large amounts of data are collected, obtaining useful insights is still limited by the technical requirements associated with data querying. Since non-technical

domain experts, project managers, and organizational decision-makers are generally not proficient in Structured Query Language (SQL), they are unable to deal on their own with complicated relational schemas that involve primary-foreign key relationships. As a result, the everyday analytics processes have to rely on specialized data engineering teams, which in turn causes significant delays in the analytical pipelines, creates bottlenecks in the manual preparation of reports, and leads to inefficient reporting cycles. In order to overcome these limitations, natural language interfaces to databases (NLIDB) based on generative Large Language Models (LLMs) have become increasingly popular. Yet, there are important weaknesses associated with using foundational LLMs directly for Text-to-SQL parsing. The queries generated by such LLMs often suffer from schema hallucination, meaning that the models create column or table identifiers that do not actually exist, and they have difficulty achieving a finegrained alignment between tokens and columns when the user's prompts use domain-specific phrasing. Moreover, standard Text-to-SQL systems operate without any external context and thus are unable to make use of auxiliary organizational documentation, data dictionaries, or unstructured business definitions that are needed for interpreting domain-specific information.

In order to deal with these structural difficulties, the present study creates an end-to-end, autonomous Data Analyst Agent using the Flowise orchestration engine together with Phrase-based Schema Linking (PS-SQL). The Flowise platform offers a modular, lowcode solution for coordinating agentic workflows, dynamic tool routing, and the retention of memory during multi-turn analytical interactions. The agent employs PS-SQL to link conversational phrases to actual database entities, thus preventing schema mismatch errors before any code is executed.

The key contributions and core architectural capabilities of the proposed system include:

- Resolving ambiguities involving phrase structures by using PSSQL to compare conversational user queries directly with the current database metadata and thus avoiding syntactic hallucination.
- Using Flowise to implement a modular agentic workflow: coordinating an autonomous thought-action-observation cycle which directs tasks through database connectors, code interpreters, and retrieval pipelines.
- Hybrid Context Augmentation (RAG): Combine vector database retrieval with unstructured operational documentation, data dictionaries, and static tabular files to provide context for database queries.
- Automated data processing and visualisation: using automated execution pipelines in Python (with Pandas, Matplotlib and Seaborn) to produce both numerical summaries and graphical charts of publication quality together with validated SQL queries.
- Multi-turn contextual continuity: remembering the conversation so that iterative exploratory data analysis (EDA) can be carried out without users having to repeat the previous contextual constraints.

II. LITERATURE SURVEY

Research on converting natural language into SQL has moved through several distinct phases, and the survey below is organized around the strand of work that most directly shaped the design choices made in this project: schema linking, LLM-based generation, and the practical gap that still exists between a working prototype and a fully conversational data-analysis assistant.

1. Lan and Li [1] proposed PS-SQL, the phrase-based schemalinking method this project takes its name from. Rather than masking single tokens the way earlier pre-trained-model approaches did, they group

question words into phrases before matching them against table and column names, arguing that a phrase carries enough context to tell whether it refers to a table or a column, something a lone token often cannot do. Their experiments show a clear improvement in linking precision, but the paper itself stops at generating a schema mapping — it does not connect that mapping to an executed query, a returned result, or a user-facing answer, which is the gap the present system tries to close.

2. Gao et al. [2] carried out a large benchmark study, later known as DAIL-SQL, on how much prompt design actually matters for LLM-based SQL generation. They tested different ways of representing the schema and selecting few-shot examples and found that these choices swing execution accuracy by a wide margin on the Spider benchmark, more than most model upgrades do on their own. The paper is essentially a controlled comparison of prompting strategies, so it does not build or evaluate an end-to-end assistant that a non-technical user could actually talk to.
3. Sun et al. [3] introduced SQL-PaLM, which looks at the same problem from the fine-tuning side rather than the prompting side. Using Google's PaLM-2 as the base model, they combine few-shot, execution-consistency prompting with instruction finetuning and get strong results across several Spider variants, including versions of the benchmark designed to be harder or more realistic. Because the emphasis is on model adaptation, the work does not really address how such a model would sit inside a larger application with a database connection, a safety layer, or a chat interface.
4. Maddila et al. [4], writing about an internal tool built at industry scale, showed that AI-assisted SQL writing genuinely helps developers in day-to-day work. Their system suggests SQL as an engineer types, using the surrounding schema and code as context, and their rollout data indicates fewer mistakes and less time spent on routine queries. This is a valuable data point that such tools work in practice, but the tool is aimed at professional developers editing code, not at a business user asking a question in plain language and expecting a finished analysis back.
5. Liu et al. [5] published a wide-ranging survey that maps out where text-to-SQL research has ended up in the LLM era and where it still needs to go. They group the field around a handful of open problems, schema linking, retrieval-augmented context, reasoning over ambiguous questions, and explaining why a query was written the way it was, and note that most published systems tackle one of these in isolation. That observation lines up with what this project ran into directly: schema linking, retrieval, and explanation are usually studied separately, even though a real user needs all three at once.
6. Wang et al. [6] proposed RAT-SQL, one of the more influential schema-linking architectures before the LLM prompting era took over. Their relation-aware attention mechanism lets the model reason jointly about the question, the tables, and the columns instead of treating schema linking as a separate pre-processing step, and it was for a long time close to the state of the art on Spider. The architecture is specific to a
7. trained encoder-decoder setup, though, so its ideas had to be reinterpreted rather than reused directly for a system built around prompting a general-purpose LLM, which is the approach taken here.
8. Katsogiannis-Meimarakis and Koutrika [7] reviewed the deep-learning side of text-to-SQL before LLMs became the default choice. Their survey is useful mainly as a map of the field's history, tracing how schema-encoding methods evolved and where the earlier benchmarks fell short, but naturally it predates the current wave of prompting-based agents and does not speak to how such an agent should be deployed or evaluated.
9. Li et al. [8] introduced RESDSQL, which makes the case that schema linking and generating the SQL skeleton are two different jobs and should not be forced through the same decoder step. Splitting them,

first ranking the relevant schema items and only then filling in the SQL structure, gave them a solid accuracy jump on Spider and its harder variants. It is a strong argument for treating schema linking as its own stage, which is more or less the role the Query Parser plays in the pipeline described later in this paper, though again the paper stops short of building a usable end-to-end tool.

10. Pourreza and Rafiei [9] took a different angle with DINSQL, breaking a hard SQL query into a sequence of smaller subproblems that GPT-4 can solve one at a time, with a self-correction pass at the end. The decomposition idea, classify the query's difficulty, then generate and check it in stages, was a useful reference point when deciding how many discrete stages the SQL route in this project should have, even though DIN-SQL itself is evaluated purely on generation accuracy rather than as part of a running application.

Finally, Yu et al. [10] created Spider, the cross-domain dataset that most of the papers above are ultimately measured against. Its defining feature is that the databases used for testing are never seen during training, which forces a system to generalize its schema linking rather than memorize a fixed set of tables. Spider does not itself propose a method, but it set the evaluation standard that shaped almost every paper cited here, and it explains why schema generalization, not just SQL syntax, keeps coming up as the central challenge in this line of work.

Taken together, these studies point to the same conclusion that motivated this project: good schema linking (as in PS-SQL, RATSQ, and RESDSQ) and good SQL generation (as in DAILSQ, SQL-PaLM, and DIN-SQL) have both been studied in depth, and industry reports such as Maddila et al. confirm that AI-assisted querying holds up outside the lab, yet none of these individually cover the full path from a spoken question to a validated query, an executed result, and a plain-language answer that also reaches into unstructured files. Building that complete loop, using PS-SQL for schema linking and Flowise to tie the stages together, is the specific contribution this project attempts.

III. METHODOLOGY

A. Overall Approach

The system was designed with a fairly well-defined set of aims in mind: it should enable a user to pose a question to the database in ordinary English, convert that question into the correct SQL statement without having to be guided through the process, execute the statement in a safe manner, and then provide both the query and the answer via a simple interface, all while ensuring that the workflow remains modular so that it's possible to switch to a different database later on. Flowise was selected to form the core of the system since it allows each of the various steps—understanding the question, generating the SQL, checking the SQL, running it, and preparing the result—to be carried out as a separate and inspectable node rather than being combined into one big script. For instance, if a particular stage needs to be modified, such as tightening the safety rules or replacing the retrieval logic, the relevant part can be altered on the Flowise canvas without affecting the rest of the pipeline.

There are two types of questions. Those concerning the organization's own transactional data are dealt with via the SQL method, which communicates with a PostgreSQL database and is explained in Section III-B. Questions involving material that has been uploaded, such as spreadsheets, CSV exports, or reference documents not already contained in the database, are handled by the Retrieval-Augmented Generation approach, which is described in Section III-C; in this way the same conversational interface is able to answer questions based on either of the two sources without the user having to know which one is being used underneath. When the system was being developed, the SQL method was tested using a small

sample sales database and the RAG method was tested using a public ecommerce order export (Amazon Sale Report.csv); both tests were carried out simply to ensure that the pipeline behaved sensibly when presented with realistic data and were not used as a formal benchmark.

B. SQL Route: PS-SQL Schema Linking and Query Execution

The SQL aspect of the agent goes through five stages: Query Parser, SQL Engine, Safety Validator, Database Query, and Executive Summary.

It is at the Query Parser that PS-SQL makes its contribution. Drawing on the phrase-based linking approach proposed by Lan and Li [1], the parser does not attempt to match individual words to column names as older methods had done; instead it extracts short phrases from the question—such as "top selling products" or "last month's revenue"—and then compares each of these phrases against the current schema (including table names, column names, and a few sample values taken from the database) rather than against a fixed list that was prepared in advance. As a result, a compact, structured summary of what the question is really asking for is produced: namely, which metric, which tables and columns are involved, any filters, any grouping, and a time window if one has been mentioned. It is this summary, not the original sentence, that is passed on to the following stage. The reason for carrying out the linking in this way—rather than by masking individual tokens—follows the same line of argument as that given by RAT-SQL [6] and RESDSQL [8] in the case of fine-tuned models: a model that reasons about both the schema and the question simultaneously is able to link the information more reliably than one that treats linking as an afterthought.

Once the relevant tables and columns have been identified, the SQL Engine only needs to look at that smaller part of the schema rather than the entire database when asking the language model to produce a SQL statement. By keeping the prompt limited in this way, the model is less likely to invent a column that doesn't exist, since that is one of the more common types of failure mentioned in the text-to-SQL literature. Before the draft statement even reaches the database, the Safety Validator examines it to make sure that it is a single read-only SELECT statement, that it does not include INSERT, UPDATE, DELETE, DROP, ALTER, or TRUNCATE, that it is not

chained to a second statement, and that it includes a row limit; any statement that fails this check is sent back without being executed. It is only after a statement has passed the Safety Validator that it proceeds to the Database Query stage, where it is run against PostgreSQL using a read-only account and the resulting rows are returned. In the Executive Summary stage, those rows are then converted into a form that a non-technical reader can understand: a brief natural-language answer, the KPI figures that are worth highlighting, a suggested chart, and if a number appears to be unusually high or low when compared to a baseline, a flag is raised to point this out.

C. The RAG route: Providing answers based on uploaded files

Since it is not possible to answer every useful question just by referring to the database, the agent also provides a route involving documents and spreadsheets which consists of four stages: Document Parser, Passage Retrieval, Data Synthesis, and Citation Mapping.

The Document Parser is able to deal with any type of file that is uploaded. If the file is a CSV sheet, it reads the header row, determines the type of each column, and considers each row as a record; if the file is a text document, it divides the content into chunks the size of paragraphs. In both cases, each record or chunk is given a tag indicating its source, the name of the file and, in the case of spreadsheets, the row to which it belongs, and is stored in such a way that it can be searched later on. Upon receiving a question, the Passage Retrieval process retrieves the records or chunks that are most relevant to it rather than passing

the entire file to the language model. The Data Synthesis stage then carries out the task of formulating the answer: it combines the retrieved rows into totals, averages, and breakdowns whenever the question requires this, and asks the language model to produce an answer that is based on those figures, applying the same kind of anomaly check that is used on the SQL side when a particular figure stands out. Citation Mapping is the final step and, arguably, the most important one for establishing trust: it links each claim in the answer back to the specific row or passage from which it came, so that a user can check "why does it say that" rather than having to take the summary on faith.

D. Shared Presentation Layer and Practical Notes

No matter which method is used to answer a question, the response is passed to the same presentation layer, so whether the answer comes from a database or from a document it appears and feels the same to the user: in the form of a result table, KPI cards, a chart, a short list of findings, and, where appropriate, an anomaly flag together with a few suggested next steps. The chart component offers seven types—Bar, Ranking, Line, Area, Donut, Pie and 3D Space—and at the Executive Summary or Data Synthesis stage the one that best suits the question is selected, such as a Ranking chart for 'who's on top', a Line or Area chart for a trend and so forth.

It is worthwhile to be exact about a few implementation details since not all of the things that the architecture is designed to do were carried out in each test run. The language model is accessed via Flowise's model-connector node, meaning that the specific model being used is a matter of deployment and not something determined by the workflow design. The Safety Validator is a simple rule-based function node and not a component that has been learned, which ensures that its behaviour remains predictable and easy to audit. The aggregation of data before creating the charts makes use of Python's Pandas library, while the seven types of chart that are actually displayed to the user—Bar, Ranking, Line, Area, Donut, Pie, and 3D Space—are presented as interactive features in the web dashboard (as can be seen in Figs. 6 and 7) rather than as static plots, so Pandas' function is limited to preparing the data rather than producing the chart that the user sees. The design also includes the agent keeping a record of the conversation over successive turns, so that a follow-up question could make use of the table, the filters, or the uploaded file from an earlier turn without requiring the user to repeat all the information; since the sessions described in Section V each consisted of a single question, this ability to handle multiple turns is part of the intended design rather than something that has been demonstrated by the results reported here. The entire arrangement has not been formally benchmarked against a labelled dataset; at this stage of the project the aim had been to produce a working pipeline whose behaviour could be checked one question at a time, which is precisely what Section V reports.

IV. BLOCK DIAGRAM

Figure 1 shows the block diagram of the Data Analyst Agent as it has been implemented. The user poses questions or uploads CSV/Excel files via the Frontend web interface; the Frontend then sends the query or the file data to Flowise, which hosts the LLM Node (responsible for query understanding, SQL generation, and result analysis) as well as the Tools/Agents layer (comprising Text-to-SQL, Database Connector, Data Analysis, Chart Generation, and KPIs and Suggestions). Flowise sends the generated SQL query to the Database, which holds the structured sales and customer data and returns the results of the query; Excel/CSV files and APIs are imported into or synchronized with the Database. The response, consisting of the results, charts, and insights, is sent back through Flowise to the Frontend, which then produces the final Output comprising Charts and Visualizations, Key Findings, Suggestions, and KPIs.

Figures 2 and 3 give a more detailed view of the internal structure of the SQL and RAG chatflows that are used to implement the Tools/Agents layer.



Fig. 2. Block diagram of the five-stage SQL workflow with PS-SQL schema linking.

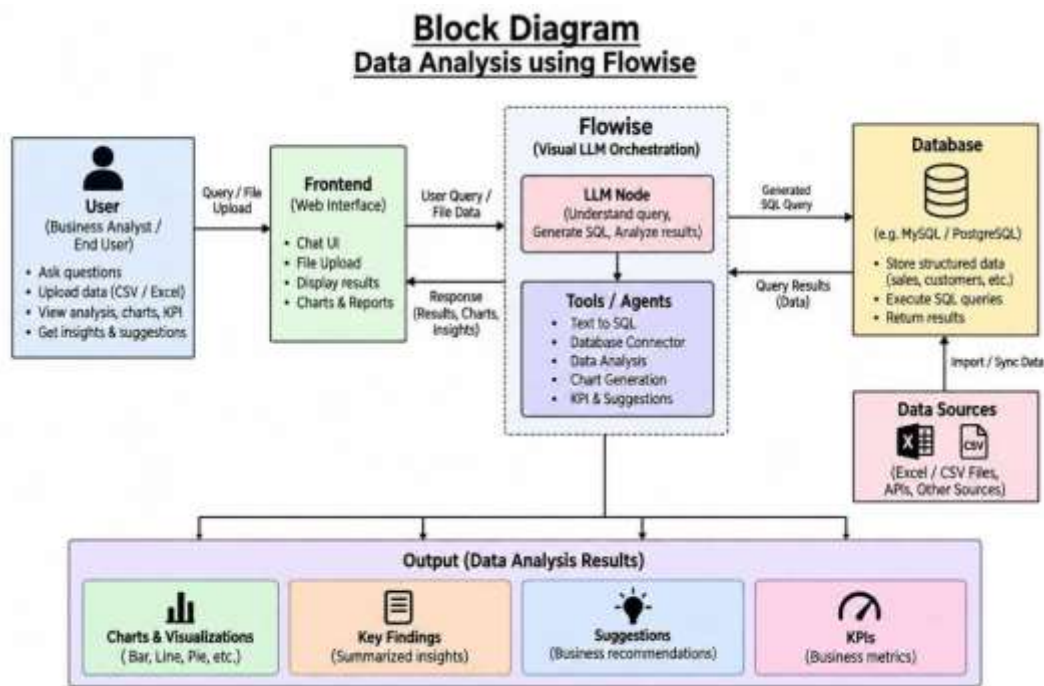


Fig. 1. Block diagram of the Data Analyst Agent using Flowise.

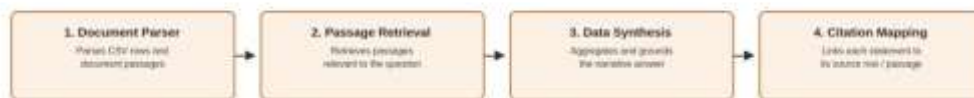


Fig. 3. Block diagram of the four-stage RAG workflow.

Table I situates the proposed agent relative to conventional, manually authored SQL querying before Section V reports outputs captured from the running system.

TABLE I Comparison of Traditional SQL Querying and the Proposed Natural-Language Data Analyst Agent

Aspect	Traditional SQL Querying	Proposed NL Data Analyst Agent
User expertise required	Requires SQL syntax and schema knowledge.	Plain natural language question; no SQL knowledge required.
Query authoring	Manually translated into SQL per question.	Generated automatically via phrase-based schema linking (PS-SQL).

Safety checks	Dependent on developer/DBA discipline.	Automated Safety Validator inspects every query before execution.
Data source scope	Structured data already modelled in the database.	PostgreSQL data (SQL route) and auxiliary CSV/document data via vector-based RAG.
Result interpretation	Raw tabular output; manual interpretation.	Result tables, KPI cards, charts, and findings generated automatically.
Traceability of claims	Implicit in query logic; not surfaced to user.	RAG route provides explicit citation mapping to source rows/passages.
Anomaly identification	Requires manual inspection of figures.	Automated anomaly flags raised within the Executive Summary/Synthesis stage.
Workflow modification	Requires application or reporting code changes.	Achieved by visually reconfiguring nodes on the Flowise canvas.

V. RESULTS AND DISCUSSION

The following section describes the actual results returned by the two routes when the system was tested using the sample PostgreSQL database and the uploaded Amazon Sale Report.csv file, including screenshots of the running application beside each result as a real-example of what a user would actually see. The figures cited below were taken directly from the application; they refer to a single test session involving one dataset only, not to a formal assessment of accuracy, and must be understood in this context.

A. SQL Route

When asked to analyse the sample sales table, the agent went through each of the five PS-SQL stages, interpreted the question, produced a SELECT statement, sent it through the Safety Validator, carried out the execution, and then summarised the results, before giving both the SQL statement and the answer so that the figures could be checked rather than merely accepted on trust. Figure 4 illustrates this interaction as it appeared in the application: at the top, the answer in plain language; in the middle, the SQL Pipeline panel showing that all the stages had been completed; and at the bottom, the exact statement that had been executed. The query grouped the sales table by sale count, revenue, profit, average revenue per sale, and by the number of distinct customers and products; the headline figures it returned are listed in Table II.

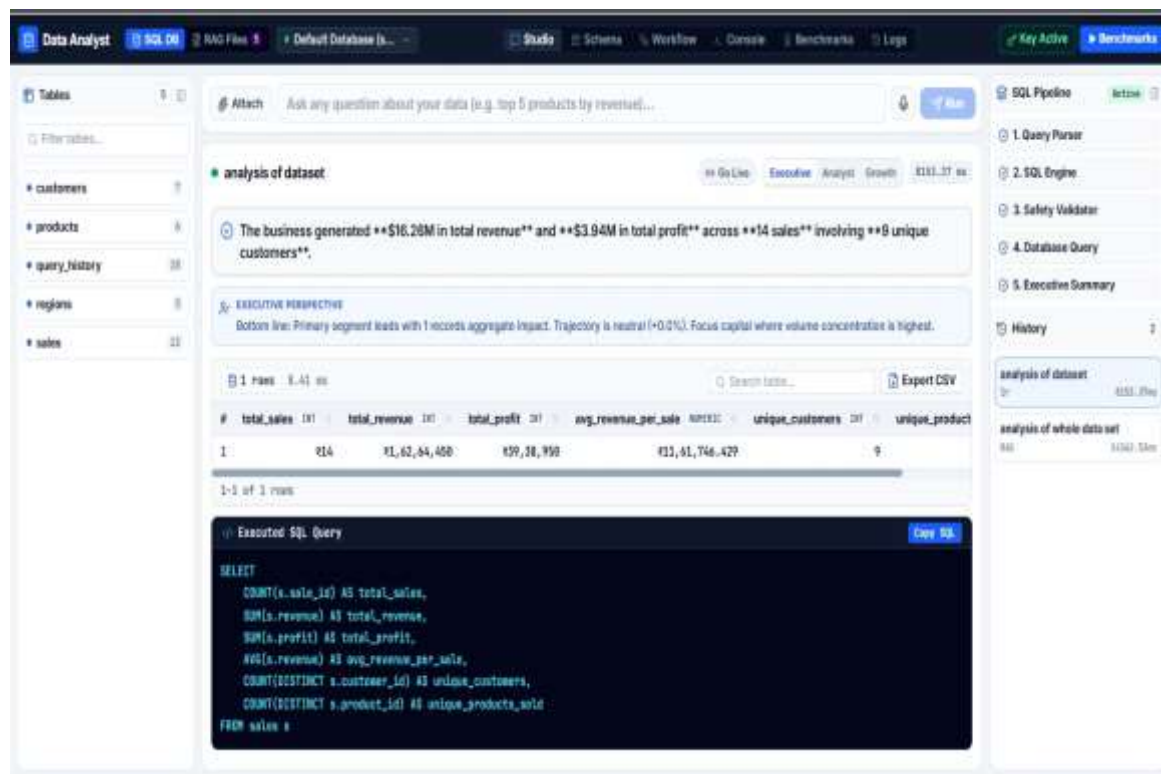


Fig. 4. SQL route output, showing the executive summary and the executed SQL query.

TABLE II SQL Route Output for the Sample Sales Database

Metric	Value Returned
Total Revenue	\$16.26M
Total Profit	\$3.94M
Rows Returned (Sales)	14
Unique Customers	9

It is worth pointing out rather than merely brushing over the following inconsistency: the raw result returned by the query when executed, as seen at the bottom of Fig. 4, gives the revenue and profit figures in rupees (₹1,62,64,450 and ₹39,38,950), whereas the natural-language summary above it presents the same figures in dollars (\$16.26M and \$3.94M). The two figures do not represent a correct conversion of one another, which indicates that the problem lies in the currency-formatting step in the Executive Summary stage and not in the SQL statement itself. This discrepancy is included here as an observation because making the correction afterwards would misrepresent what the system actually returned during this test session, and it is precisely the type of bug that a more thorough testing process would be expected to identify.

B. RAG Route

With respect to the Amazon Sale Report.csv file, the agent was requested to provide a summary based on the entire dataset. All four stages of the RAG process were completed, and the Data Synthesis stage produced a plain-language summary of the file, the number of orders, the revenue, the average order value, and the range of the catalogue, this being supported by an aggregation query that was executed internally

against the parsed file rather than being obtained from memory, as illustrated in Fig. 5. The figures obtained are given in Table III.

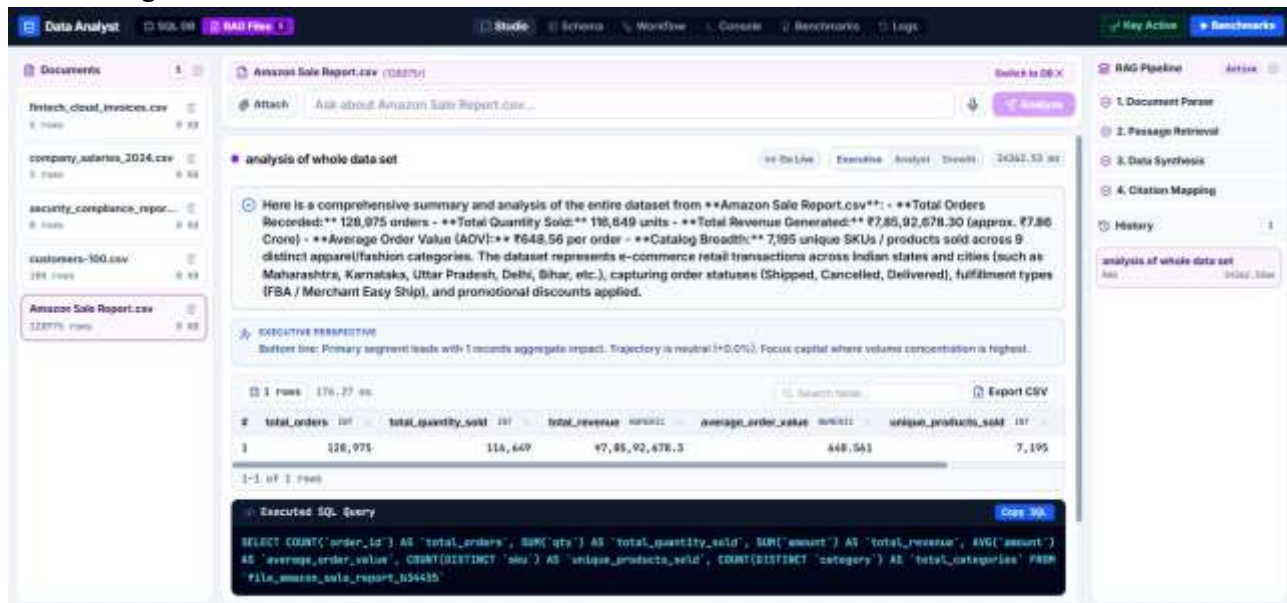


Fig. 5. RAG route output, showing the full-dataset summary for Amazon Sale Report.csv.

TABLE III RAG Route Output for Amazon Sale Report.csv

Metric	Value Returned
Total Orders	128,975
Units Sold	116,649
Total Revenue	₹7,85,92,678.30
Average Order Value	₹648.56
Unique Products	7,195
Product Categories	9

C. Category and Product-Level Findings

The same file was also subjected to two more specific questions. When asked to identify the five products that together generated the highest combined revenue and profit, the agent stated that J0230-SKD-M was the top individual product, having accounted for ₹5.28 L out of the total ₹23.94 L earned by the top five products (which averaged ₹4.79 L each), and noted a -15.0% contraction trend within that group, as illustrated in the KPI cards and the line chart in Fig. 6. When instead requested to provide a category-level breakdown, the agent found that Set was the leading category in terms of volume with 50,284 orders compared to an average of 14,330.56 across the nine categories available, and in this case the anomaly check triggered: Fig. 7 displays the resulting area chart which highlights a +250.9% spike for Set compared to the dataset's mean, together with an overall -98.3% contraction trend. Table IV gives the underlying category breakdown associated with that result.

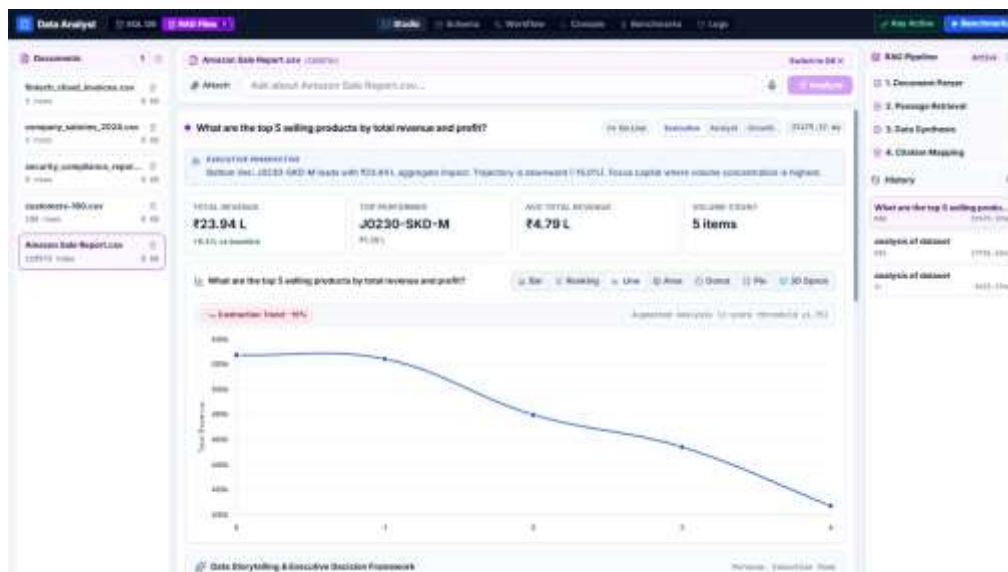


Fig. 6. KPI cards and line chart for the top-5-products-by-revenue query.

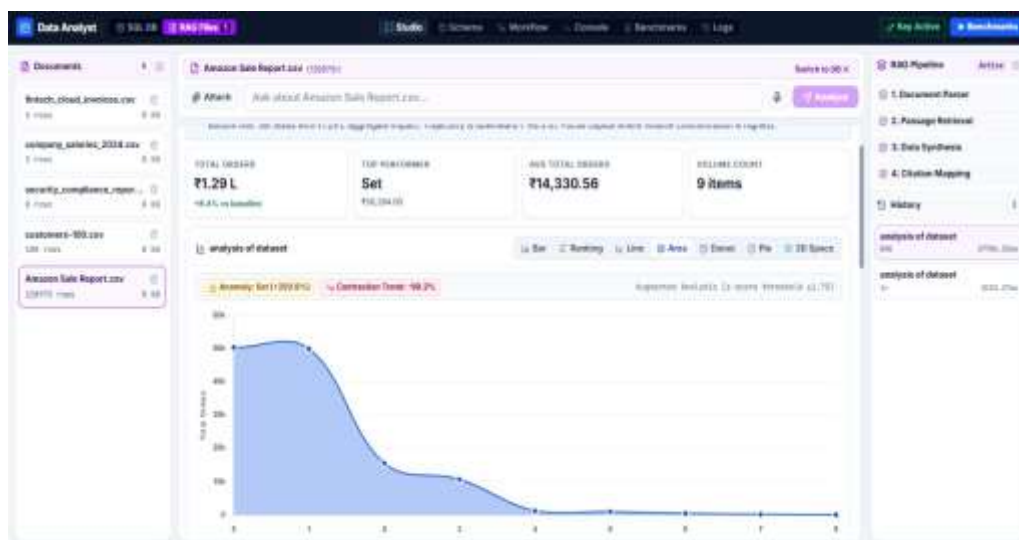


Fig. 7. Area chart flagging the anomaly and contraction trend for the category-level query.

TABLE IV Top Five Categories by Revenue — Amazon Sale Report.csv

Category	Orders	Revenue	Avg. Order Value
Set	50,284	₹3,92,04,124.03	₹833.39
kurta	49,877	₹2,12,99,546.70	₹455.93
Western Dress	15,500	₹1,12,16,072.69	₹762.79
Top	10,622	₹53,47,792.30	₹526.10
Ethnic Dress	1,159	₹7,91,217.66	₹723.90

Looking at the two queries together is helpful because "top products" and "top categories" identify different leaders (one being a single SKU and the other an entire category) since they are in fact responding to completely different questions, and the agent correctly maintained that distinction rather than treating

them as the same. Figure 8 displays the Key Findings and Strategic Suggestions section that was produced from the category-level result, together with the category table it was based on. Based on that finding, the Strategic Suggestion stage suggested allocating more marketing and inventory focus to Set as it is growing quickly, testing bundle offers on the categories that are moving more slowly such as Ethnic Dress, and setting up an automatic alert for large fluctuations so that a spike like the one shown here will be noticed earlier the next time. Although these suggestions seem reasonable, they are automatically generated from a single data snapshot and are intended to serve as a starting point for an analyst to examine, not as recommendations to act on directly.

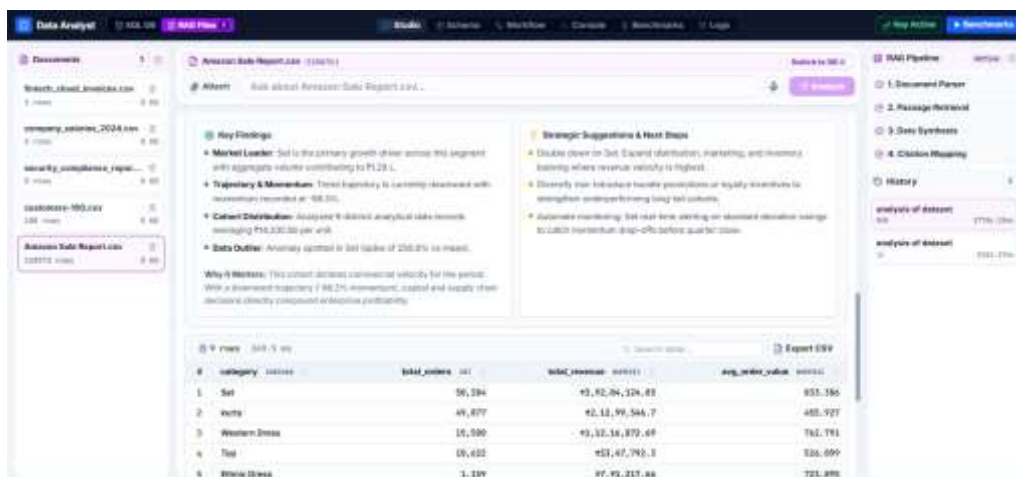


Fig. 8. Key Findings and Strategic Suggestions panel with the underlying category table.

D. Discussion

When the results are compared with those in Table I, they demonstrate that the two methods are carrying out the purpose for which they were designed: a straightforward question is entered and in return you get a results table, a set of summary figures, a suggestion of a chart, and a written conclusion, all without the person asking having to write SQL or manually go through a spreadsheet. A few cautions are worth stating clearly. PSSQL's phrase matching works most effectively when the way the question is worded is fairly close to how the schema is named, so a question that uses unusual wording may still cause it to fail. The Safety Validator consists of a fixed set of rules and is not a filter that has been learned from data, so it should be used in addition to read-only database credentials rather than relied upon as the only safety measure. Since no labelled benchmark was carried out, the figures mentioned above should not be interpreted as indicating accuracy; they only show that the entire pipeline produced a reasonable answer to the questions it was tested on. The next logical step would be to extend the testing to include a proper benchmark, and also to strengthen the Flowise deployment for use in any context beyond a classroom environment, a point which is explored further in future work.

REFERENCES

1. Z. Lan and S. Li presented a paper entitled "PS-SQL: Phrase-based Schema-Linking with Pre-trained Language Models for Text-to-SQL Parsing" at the 2024 6th International Conference on Natural Language Processing (ICNLP).

2. D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation," *Proc. VLDB Endowment*, vol. 17, no. 5, pp. 1132–1145, 2024.
3. R. Sun, S. Ö. Arik, H. Nakhost, H. Dai, R. Sinha, P. Yin, and T. Pfister, "SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL", *Trans. Machine Learning Research (TMLR)*, 2024.
4. C. Maddila, N. Ghorbani, K. Jabre, V. Murali, E. Kim, P. Thakkar, N. P. Laptev, O. Harman, D. Hsu, R. Abreu, and P. C. Rigby, "AI-Assisted SQL Authoring at Industry Scale," in *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2025, pages 148 to 158.
5. X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, and Y. Luo, "A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going?," *IEEE Transactions on Knowledge and Data Engineering*, 2025.
6. B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, Online, 2020, pages 7567 to 7578.
7. G. Katsogiannis-Meimarakis and G. Koutrika, 'A Survey on Deep Learning Approaches for Text-to-SQL', *The VLDB Journal*, volume 32, number 4, pages 905 to 936, 2023.
8. H. Li, J. Zhang, C. Li, and H. Chen, "RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL", in the *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, number 11, 2023, pages 13067 to 13075.
9. M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction", *arXiv preprint arXiv:2304.11015*, 2023.
10. T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, and D. Radev, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Brussels, Belgium, 2018, pp. 3911–3921.