UNIVERSITY OF MYSORE

(Re-accredited by NAAC with 'A' Grade)
(NIRF-2022: Ranked 33rd in University Category and 54th in Overall Category)

MYSORE UNIVERSITY SCHOOL OF ENGINEERING

Manasagangothri campus, Mysuru-570006
(Approved by AICTE, New Delhi)

A Mini Project (21ADP67)
On

"Revolutionising Database Interaction using Agentic AI."

Submitted in partial fulfilment for the award of the degree of
Bachelor of Engineering
In
Artificial Intelligence and Data Science

Submitted By

RANJAN U	**22SEAD53**
ROHITH DS	**22SEAD56**
SUDHANVAA H KASHYAP	**22SEAD64**

Under Faculty Incharge
Dr. Sayeda Umera Almas
Asst. Professor,
Dept. of AI & DS,
MUSE, UOM, Mysuru - 570006

Head of the Department
Mrs Poornima K
Asst. Professor and HOD
Dept. of AI & DS,
MUSE, UOM, Mysuru - 570006

Dr. M. S. Govinde Gowda
Director
MUSE, UOM, Mysuru - 570006

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE,
MYSORE UNIVERSITY SCHOOL OF ENGINEERING,
UNIVERSITY OF MYSORE,
Manasagangothri campus, Mysuru-06.

UNIVERSITY OF MYSORE

(Re-accredited by NAAC with 'A' Grade)
(NIRF-2022: Ranked 33rd in University Category and 54th in Overall Category)

MYSORE UNIVERSITY SCHOOL OF ENGINEERING

Manasagangothri campus, Mysuru-570006
(Approved by AICTE, New Delhi)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

CERTIFICATE

This is to certify that the Mini-Project (21ADP67) entitled "**Revolutionising Database Interaction using Agentic AI** " is a bonafide work carried out by **Ranjan U, Rohith DS**, and **Sudhanvaa H Kashyap,** students of **VI Semester**, bearing Register No. **22SEAD53, 22SEAD56, and 22SEAD64** from the **Department of Artificial Intelligence and Data Science**, in partial fulfillment of the requirements for the award of the **Bachelor of Engineering** degree at the **Mysore University School of Engineering, University of Mysore, Mysuru.**

It is further certified that all corrections and suggestions indicated during the evaluation have been duly incorporated by the aforementioned candidate.

Signature of the Head of the Department
Mrs Poornima K
Asst. Professor and HOD
Dept. of AI & DS,
MUSE, UOM, Mysuru - 570006

Signature of the Faculty Incharge
Dr. Sayeda Umera Almas
Asst. Professor,
Dept. of AI & DS,
MUSE, UOM, Mysuru - 570006

Signature of the Director
Dr. M. S. Govinde Gowda
Director
MUSE, UOM, Mysuru - 570006

Name of the Examiners: **Signature with date**

1. .
2.

DECLARATION

We, Ranjan U, Rohith DS and Sudhanvaa H Kashyap, bearing Register Nos. **22SEAD53, 22SEAD56 and 22SEAD64** of VI Semester, of **Department of Artificial Intelligence and Data Science, University of Mysore, Mysuru**, hereby declare that the **Mini-Project (21ADP67)** entitled **"Revolutionalising Database Interaction using Agentic AI"** has been duly carried out by us under the guidance of **Dr. Sayeda Umera Almas**, Asst.Professor, Department of Artificial Intelligence and Data Science, University of Mysore, Mysuru.

This Mini-Project report is submitted in partial fulfillment of the requirements for the award of the Bachelor of Engineering degree in the Department of Artificial Intelligence and Data Science by the University of Mysore, Mysuru, during the academic year **2024–2025**.

We further declare that the content of this report has not been submitted previously by anyone for the award of any degree.

Place: Mysuru

RANJAN U	**22SEAD53**
ROHITH DS	**22SEAD56**
SUDHANVAA H KASHYAP	**22SEAD64**

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of the **Mini-Project (21ADP67)** would be incomplete without acknowledging the blessings of the Almighty and the support of those who made this achievement possible. Their guidance and encouragement have been instrumental in the successful presentation of our project.

We are grateful to the **Department of Artificial Intelligence and Data Science, Mysore University School of Engineering, University of Mysore, Manasagangotri Campus, Mysuru**, for providing us the opportunity to enhance our knowledge through this Mini-Project.

We extend our sincere thanks to **Dr. M. S. Govinde Gowda, Director, Mysore University School of Engineering, University of Mysore**, for providing us with the necessary support, encouragement, and facilities to carry out and present our Mini-Project.

We express our heartfelt gratitude to **Mrs Poornima K, Head of the Department**, **Mysore University School of Engineering, University of Mysore**, for her constant encouragement and valuable suggestions throughout the Mini-Project.

We are deeply thankful to **Dr. Sayeda Umera Almas, Asst. Professor, AI & DS, Mysore University School of Engineering, University of Mysore**,for her guidance, motivation, and dedicated support, which played a vital role in the successful completion of our Mini-Project.

Finally, we would like to thank our family members and friends for their unwavering support and encouragement throughout this journey.

RANJAN U	**22SEAD53**
ROHTIH DS	**22SEAD56**
SUDHANVAA H KASHYAP	**22SEAD64**

ABSTRACT

The necessity of expertise in Structured Query Language (SQL) has long created a significant barrier, limiting direct database interaction to technically proficient users. This project addresses this challenge by developing an advanced agentic AI framework that revolutionizes database interaction, making it intuitive, conversational, and reliable. At its core, the system is a stateful multi-agent workflow orchestrated by LangGraph. It leverages a specialized multi-LLM strategy: Google's Gemini for initial intent analysis, Mistral's Codestral for precise SQL query generation, and DeepSeek R1 for a query verification, token size management, correction and query execution. This ensures that every query is not only syntactically valid but also semantically aligned with the user's intent. The agent employs a Retrieval-Augmented Generation (RAG) approach, dynamically fetching table schemas from a Chroma-DB vector store to provide context for the SQL generator. These key innovations with the interactive human-in-the-loop mechanism are managed through a lightweight Stream-lit interface. By successfully integrating these components, the project delivers a feasible solution that democratizes data access. It transforms the complex task of database querying into a simple conversation, establishing a new paradigm for trustworthy and user-centric database interaction.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

UI	User Interface
API	Application Programming Interface
LSTM	Long Short-Term Memory
LLM	Large Language Models
SOTA	State of the Art
NLP	Natural Language Processing
RAG	Retrieval-Augmented Generation
SQL	Structured Query Language
NLID	Natural Language Interfaces to Databases
CSV	Comma separated Values (Document)
NLI4DB	NL interfaces for databases
HIL	Human-in-the-Loop
NER	Named Entity Recognizer
NEL	Neural Entity Linker
NSP	Neural Semantic Parser

CHAPTER 1
INTRODUCTION

In an increasingly data-driven world, relational databases stand as the backbone of traditional information systems, storing vast quantities of valuable information. However, accessing this data has been gated by the need for proficiency in SQL, creating a significant barrier for non-technical users and hindering the democratization of data access . The field of NLP has long sought to bridge this gap through Agentic systems, which aim to analyse natural language questions and generate executable SQL queries. This endeavour promises to empower users from all domains to interact with complex databases as easily as having a conversation .

The evolution of Text-to-SQL has been marked by several distinct paradigms. Early efforts relied on rule-based systems and handcrafted templates, which were effective in constrained environments but lacked scalability and robustness when faced with complex database schemas or linguistic variations . The advent of deep learning brought about a significant shift, with models based on LSTM and Transformer architectures demonstrating the ability to learn the mapping between natural language and SQL syntax automatically . Then sequence-to-sequence models (Transformer era) represented a major leap forward, yet they still struggled with the nuances of complex, nested queries and cross-domain generalization. The recent emergence of LLMs has revolutionized the field, introducing a new contextual analysis with thinking power and fine-tuning capabilities. Despite their prowess, even advanced LLMs are not a panacea; they are susceptible to challenges such as generating syntactically incorrect queries, hallucinating table or column names, and misinterpreting ambiguous user intent. To address these shortcomings, contemporary research has pivoted towards more sophisticated architectures, including RAG to provide LLMs with accurate, real-time schema context, and multi-agentic frameworks that decompose the complex task into a series of manageable sub-problems solved via sequential yet cyclic agentic workflow.

1.1 History

Early efforts to let humans query databases in everyday language emerged alongside relational databases in the 1970s, when SQL itself evolved from SEQUEL at IBM to operationalize Codd’s relational model. The first NLIDB were rule-based and domain-specific: linguists and system builders hand-crafted grammars, lexicons, and semantic maps from user utterances to logical forms, then to SQL, which worked in narrow domains but was brittle, costly to port, and struggled with linguistic variability. Through the 1990s and 2000s, statistical parsing began to supplement symbolic pipelines, yet systems still relied on lexicon/schema alignment heuristics and template-like mappings that relied on complex joins

and nested queries. A major shift came with neural sequence-to-sequence models, initially with RNN/LSTM decoders and later Transformers, which learned to generate SQL from natural language but often produced ill-formed or non-executable queries and mis-linked question tokens to schema elements without structural awareness. To address these gaps, structure and table-aware decoders incorporated SQL grammar constraints and schema linking, improving executability and robustness on benchmarks like WikiSQL and table QA settings. Recent advances introduced fine-grained query understanding and modular pipelines (NER, entity linking, neural parsing) to better align user intent with schema and values. At present, systems increasingly combine retrieval-augmented schema/context grounding with large language models and verification steps, reflecting a maturation from handcrafted rules to hybrid, execution-aware neural and agentic architectures.

1.1.1 Background and Evolution

Text-to-SQL has evolved through four distinct eras, each addressing the limitations of its predecessors while moving closer to reliable, conversational database querying. The earliest systems (1970s–1990s) were rule-based NLIDBs built on handcrafted grammars, lexicons, and semantic parsers that mapped natural language to logical forms and then to SQL. These pipelines performed well in narrow domains but were brittle, expensive to maintain, and hard to port across schemas. They also struggled with ambiguity, paraphrase, and complex relational reasoning.

The statistical era (2000s) introduced probabilistic parsing and machine learning for semantic role labeling, schema alignment, and lexical mapping, reducing manual engineering but retaining heavy reliance on pattern-based features and domain-specific supervision. While more adaptive, these systems still faltered on unseen schemas, multi-table joins, and nested queries.

Neural sequence-to-sequence models (2016–2019) marked a pivotal shift: LSTMs and then Transformers learned to translate questions directly into SQL. Benchmarks like WikiSQL catalyzed rapid progress, but generic seq2seq models often hallucinated columns, produced non-executable SQL, or ignored database structure. This led to structure-aware innovations—explicit SQL grammars, syntax-constrained decoding, schema linking, and pointer networks—significantly improving executability and compositional generalization on harder datasets (e.g., Spider).

The LLM era (2020s–present) brought powerful in-context reasoning and cross-domain generalization, yet introduced new challenges: silent failure modes, overconfident hallucinations, and sensitivity to ambiguous intent. As a result, the field moved toward hybrid and agentic designs that combine strengths: Retrieval-Augmented Generation (RAG) to ground models in real schemas and metadata; multi-step planning and tool use; and verification or self-correction loops (e.g., execution checks, critique-and-revise). Human-in-the-loop workflows further enhance reliability by resolving ambiguity at decision points.

This project embraces that evolution: a multi-agent, RAG-grounded, verification-first pipeline orchestrated with LangGraph, using specialized LLMs for analysis (Gemini), generation (Codestral), and verification (DeepSeek R1), plus an interactive UI to keep humans in control. The result is a practical synthesis bridging linguistic flexibility with database exactness designed for trustworthy, end-to-end agentic database interaction.

1.2 Origin of Idea

The idea originated from a desire to learn and apply Agentic AI to real-world, multi-step problems with complex relational structures, while practicing end-to-end system architecture. We set out to explore how specialized agents can plan, collaborate, and verify across stages intent analysis, schema grounding, SQL generation, and validation showcasing the capabilities of modern agentic patterns. This project became a hands-on framework to evaluate the capabilities and benchmarks of the latest LLMs and agent frameworks in concert, not isolation. By integrating retrieval, multi-agent orchestration, and verification, we aimed to internalize best practices and build a trustworthy, deployable text-to-SQL pipeline.

CHAPTER 2

Related Works

Text-to-SQL research has progressed from early NLIDB to modern hybrid architectures integrating LLMs, retrieval-augmented generation (RAG), and agentic verification, enabling the production of executable, intent-aligned queries at scale. Foundational surveys synthesize decades of work, consistently highlighting three persistent challenges: **(1)** natural language understanding, **(2)** schema grounding, and **(3)** syntax- and semantics-accurate SQL generation—along with the difficulty of generalizing across heterogeneous schemas and bridging evaluation gaps [1]. Deng et al. [1] provide a comprehensive overview of datasets, methods, and evaluation metrics that have shaped the field, serving as a critical baseline for understanding its trajectory.

Early neural approaches demonstrated that generic sequence-to-sequence models—LSTM and Transformer architectures—could learn NL-SQL mappings directly from paired training data. However, surveys by He et al. [3] and Katsogiannis-Meimarakis & Kermanidis [4] report that such models struggled with structural compositionality, schema linking, and execution robustness. These shortcomings often resulted in syntactically plausible but semantically incorrect queries due to inadequate modeling of relational structure. To address this, research advanced toward grammar-constrained decoding, pointer mechanisms, and schema-aware encoders that enforce both syntactic correctness and schema alignment. Wang et al. [8] made a pivotal contribution by introducing a **modular framework** for improving semantic parsing through "fine-grained query understanding," comprising:

- NER to detect query entities,
- NEL to map entities to schema columns and cell values, and
- NSP to synthesize the SQL query.

  This modular approach enhanced schema linking precision over basic fuzzy string matching and directly influenced our project's early analysis phase.

Liu & Xu [5] offer a systematic review of NLI4DB that examines task decomposition, interaction design, and reliability considerations for real-world deployment. Their work emphasizes that beyond model accuracy, system usability and iterative refinement loops are essential for adoption in production environments. Earlier surveys [3][4] also map the evolution from rule-based systems to deep learning, noting persistent issues even in neural settings—particularly handling ambiguous queries and adapting to unseen schemas.

The advent of LLMs marked a paradigm shift in methodology. Surveys by Gao et al. [2] and Pourreza & Rafiei [7] document how prompt engineering, in-context learning, and fine-tuning enable LLM-based models to outperform earlier cross-domain systems. However, they

also introduce risks: hallucinated columns/tables, token inefficiency, and high sensitivity to linguistic ambiguity. Gao et al. [2] in particular present a benchmark evaluation of LLM-based solutions, stressing the importance of token-efficient prompt design and exploring the viability of open-source LLMs—an approach aligned with our use of multiple specialized models to balance cost and performance.

Complementary research explores RAG to directly mitigate LLM hallucinations. Liu [6] demonstrates that retrieving schema-relevant information or similar query examples at runtime grounds the model, significantly improving accuracy and reliability. This has become a cornerstone of modern Text-to-SQL systems, ensuring the model operates on the actual database schema rather than flawed internal representations. Industry perspectives, such as those from Siddiqui et al. [10], echo these findings, advocating for RAG-powered agent-style orchestration and API governance frameworks to improve reliability and secure enterprise data.

A growing body of work focuses on multi-agent frameworks that decompose the Text-to-SQL pipeline into specialized roles for selection, generation, and refinement/verification. Zeng et al. [11] present the Multi-Agent Collaborative Framework (MAC-SQL), featuring a core decomposer agent supported by auxiliary agents that:

- retrieve smaller, relevant sub-databases, and
- refine erroneous SQL queries.

  Agents are dynamically activated as needed, enhancing flexibility and robustness. This directly inspires our architecture, which employs Gemini for analysis, Codestral for generation, and DeepSeek for verification. This multi-agent philosophy recognizes that a team of specialized components can outperform a single monolithic model.

This trajectory—from rule-based parsing to LLM-powered, retrieval-augmented, and agent-coordinated systems—underscores a consistent aim: creating robust, scalable, interpretable Text-to-SQL solutions that generalize well across diverse databases while minimizing operational risks. Our project synthesizes these advances into a multi-agent, RAG-grounded, verification-first pipeline embedded in a fully interactive user interface.

Author(s)	Framework / Focus	Journal / Conference
Deng, N., Chen, Y., & Zhang, Y. (2022)	Survey of datasets, methods, and evaluation metrics in Text-to-SQL; challenges in NLU, schema grounding, and SQL accuracy	*Proceedings of the 29th International Conference on Computational Linguistics*
Gao, D., Chen, W., &	LLM-empowered Text-to-SQL;	*Proceedings of the VLDB*

Author(s)	Framework / Focus	Journal / Conference
Wang, T. (2023)	benchmark evaluations, token-efficient prompt design, open-source model exploration	*Endowment, 17*
He, Y., Bai, D., & Jiang, W. (2018)	Neural Text-to-SQL translation using LSTM/Transformer architectures; schema linking and compositionality challenges	*Stanford University*
Katsogiannis-Meimarakis, G., & Kermanidis, K. L. (2022)	Survey on deep learning approaches for Text-to-SQL; handling ambiguous queries and unseen schemas	*The VLDB Journal, 32(1)*
Liu, M., & Xu, J. (2025)	NLI4DB systematic review; task decomposition, interaction design, reliability considerations	*arXiv preprint arXiv:2503.02435*
Liu, J. (2023)	Combining Text-to-SQL with semantic search (RAG) to reduce hallucinations and improve grounding	*LlamaIndex Blog*
Pourreza, M., & Rafiei, D. (2023)	Survey on employing LLMs for Text-to-SQL; prompt engineering, in-context learning	*ACM SIGMOD Record, 52(4)*
Wang, J., Ng, P., Li, A. H., Jiang, J., Wang, Z., Nallapati, R., Xiang, B., & Sengupta, S. (2022)	Modular semantic parsing framework with NER, NEL, and NSP for fine-grained query understanding	*EMNLP 2022: Industry Track*
Zhang, Z., et al. (2019)	Deep learning-based SQL query generation from text	*SSRN*
Siddiqui, A., Srivastava, P., & Vadupu Lakshman Manikya, P. K. (2025)	RAG-powered multi-agent orchestration for enterprise Text-to-SQL	*AWS Machine Learning Blog*
Zeng, Z., et al. (2023)	Multi-Agent Collaborative	*arXiv preprint*

Author(s)	Framework / Focus	Journal / Conference
	Framework (MAC-SQL) with decomposer and refinement agents	*arXiv:2312.11242*

Table 1: "Related Works."

CHAPTER 3

Problem Statement

Accessing relational data remains a persistent hurdle for non-technical members, especially when queries involve multi-table joins, nested subqueries, window functions, or temporal logic. Without an intelligent intermediary, users must translate business intent into precise SQL understanding schema nuances, naming conventions, and edge-case semantics. This forces reliance on data engineers or analysts, creating bottlenecks, context-switching overhead, and delays for routine information needs. Even for semi-technical users, evolving database schemas introduce fragility: previously working queries break silently, and intent drifts from execution. In practice, teams shoulder an ongoing burden of query maintenance, ad hoc support, and institutional knowledge transfer.

Designing a system architecture that decomposes the problem into analysable, verifiable stages demands careful orchestration. Managing state across agents for intent analysis, schema retrieval, SQL generation, verification, and execution. Cost-efficient API management is a core constraint: multi-LLM pipelines can escalate usage if not governed with caching, retries, backoff, max-token controls, and short, purpose-built prompts. Token size management is critical: schemas and prompts must be scoped and retrieved just-in-time (RAG) to avoid context bloat, truncation, or degraded model performance. Workflow design must guard against infinite loops, partial generations, and ambiguous handoffs. Absence of such a framework, organizations either accept slow, expert-mediated access or risk unreliable, unverified SQL generation both of which undermine scalable, trustworthy database interaction.

This project addresses these pain points through an agentic, Contextual analysed, verification-first pipeline with controlled API usage for cost efficiency. Resulting a SOTA framework application with several contributions to the user platform.

CHAPTER 4

Objectives

The primary objective of this project is to design and implement a production-ready, conversational database interface that eliminates the technical barriers preventing non-technical users from accessing and querying relational databases. In an era where data-driven decision making is paramount, the inability of domain experts, business analysts, and stakeholders to directly interact with databases creates bottlenecks, dependencies on technical personnel, and delays in critical insights. This project aims to democratize database access by creating an intelligent intermediary that understands natural language intent, translates it to precise SQL, and executes queries with verification safeguards.

The core objectives encompass developing a feasible system that maintains the scalability and reliability expected in production environments while providing an intuitive, conversational user experience. The system must handle complex database schemas, ambiguous natural language queries, and edge cases gracefully, while ensuring that generated SQL is both syntactically correct and semantically aligned with user intent. Additionally, the project seeks to establish a cost-efficient, scalable architecture that leverages language models judiciously through strategic API management, memory management and intelligent caching mechanisms.

Building upon these foundational objectives, this project delivers a feasible solution for natural language-driven database interaction and management. Our work makes several key contributions to the field, including:

1. **A Specialized Multi-Agent Framework:** Design and implementation of a stateful multi-agent system using LangGraph. The architecture moves beyond a monolithic approach by assigning distinct roles such as context analysis, query generation, and contextual validation/correction to specialized agents, resulting in a precise, modular, and maintainable workflow.
2. **A Strategic Multi-LLM Approach:** Systematic use of multiple LLMs to leverage their strengths. The system employs Google's Gemini for nuanced intent understanding, Mistral's Codestral for highly precise SQL query generation, and DeepSeek R1 as a dedicated verification cum database operator agent to correct, validate, and execute the generated query end-to-end.
3. **A Fully Interactive Human-in-the-Loop Interface:** Integration of human-in-the-loop mechanism via LangGraph, exposing key decision points through a user interface to enable guided intervention during the agentic workflow, ensuring clarity, control, and alignment with the desired outcome.

4. **Trustworthy Query Generation:** Inclusion of a dedicated verification and correction agent as a core trustable feature. This step ensures the final query executed on the database is not only syntactically valid but also semantically aligned with the user's request, substantially enhancing trustworthiness and backup.

CHAPTER 5

System Design.

This system employs a reliable agentic pipeline by orchestrating specialized agents over a stateful/cyclic LangGraph, grounded by a vector database for schema recall and a SQLite database for execution illustrated in Figure 1. The design emphasizes modularity, verification-first execution, human control points, and efficient API usage.

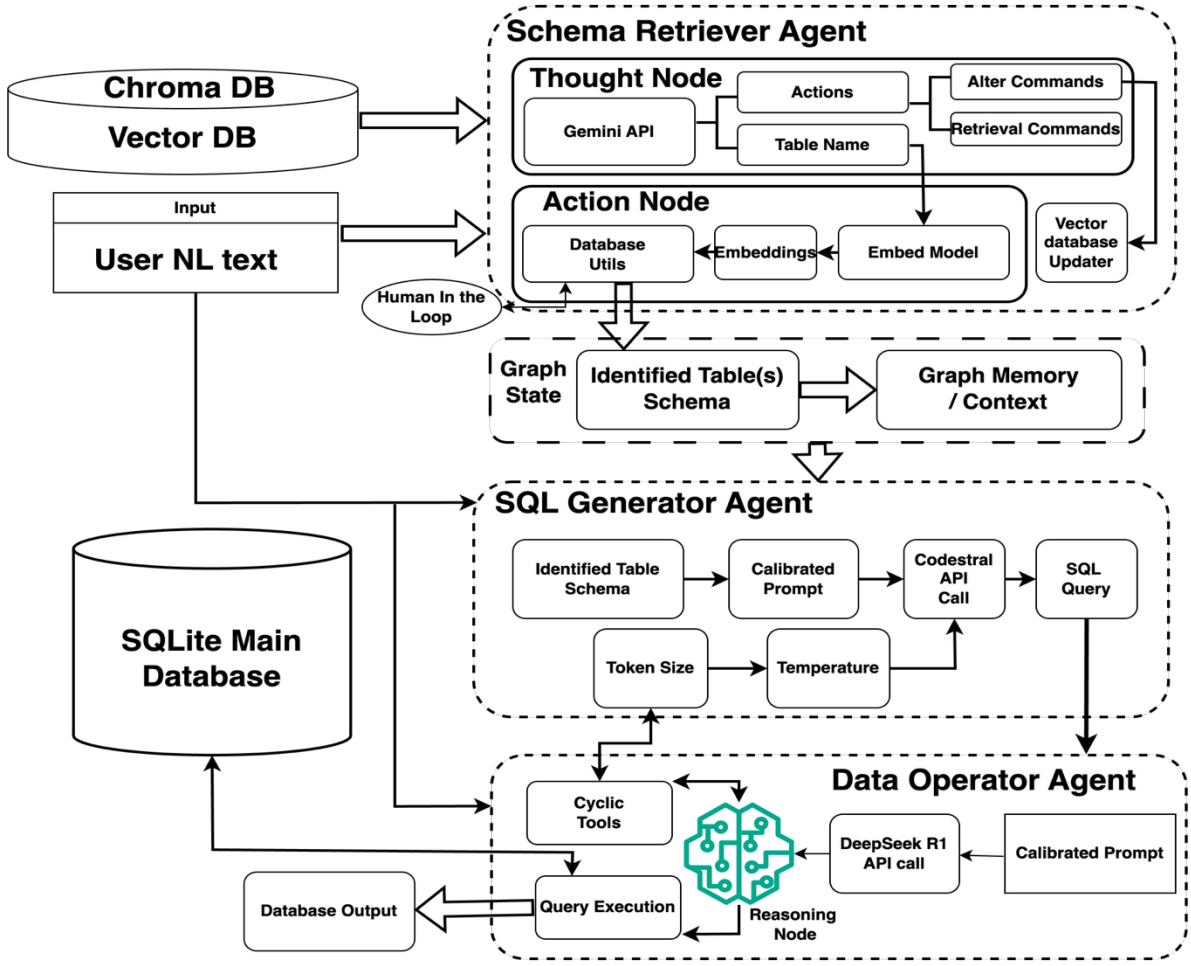

Fig1. "Agent Architecture."

Each component in the architecture is elaborated in subsequent sections.

5.1 Hardware Requirements

1. **CPU:** Modern multi-core CPU (4–8 cores) recommended to handle concurrent LLM calls, vector search, and SQLite execution; more cores improve vector indexing/search throughput for ChromaDB.
2. **RAM:** Recommended minimum 6-8 GB RAM to allocate sufficient memory for in-memory indexes.
3. **Storage:** Provision 100 GB for Chroma persistence plus database and logs.
4. **GPU:** Not required until embeddings are generated using host system.

5. **Network:** Stable broadband for API calls to LLM providers and LangGraph interactions.
6. **OS:** Designed to support all OS platform hence this is a Cross-platform framework.

5.2 Software Requirements

1. **Python:** Python 3.10+ recommended.
2. **Agent orchestration:** LangGraph for stateful, conditional, cyclic workflows; compatible with LangChain Core for message types and tool integration.
3. **UI layer:** Streamlit for conversational interface to manage chat flow and human-in-the-loop confirmations.
4. **Vector store:** ChromaDB for schema/document embeddings with in-memory HNSW index and disk persistence. Preferred to follow performance guidance for deployment.
5. **LLM connectivity:** Gemini, Mistral Codestral, and DeepSeek/OpenRouter endpoints, environment-managed API keys and retry policies.
6. **Packaging:** Requirements stated in requirements.txt that adhere to LangGraph deployment guidance for version compatibility.

CHAPTER 6

Implementation.

This system implements Text-to-SQL pipeline using a multi & sequential agent architecture driven by LangGraph, grounded by RAG over a Chroma vector store, and safeguarded by DeepSeek R1 verification. It ingests scraped exam results, builds SQLite database and chroma vector databases, generates schema-aware SQL, verifies correctness, executes, and presents results via Stream-lit UI.

6.1 Data Collection.

Data collection was conducted via web scraping of university results portals to assemble per-candidate records across exam batches. The scraper, implemented uses Selenium WebDriver with headless. It programmatically navigates to the results site, fills required inputs (registration number and a valid default date of birth), submits the form, and extracts name, semester, and GPA fields across multiple exam batches.

For dynamic elements such as result summary, dropdowns, etc. Re-location of stale elements before interacting with dropdowns, and fallback logic are used. The scraper iterates sequential registration numbers. GPAs are parsed with a simple converter and aggregated per student with an average computed over available batches. The output is persisted to CSVs by cohort and branch (AIDS/CSD/AIML) for downstream processing. This approach balances volume with throughput, resulting in producing clean, uniform CSVs ready for database construction.

6.2 Data Pre-processing.

The pre-processing stage comprises two artifacts: (1) construction of the SQLite database and (2) creation of a Chroma vector database for schema retrieval which is depicted in Figure 2.

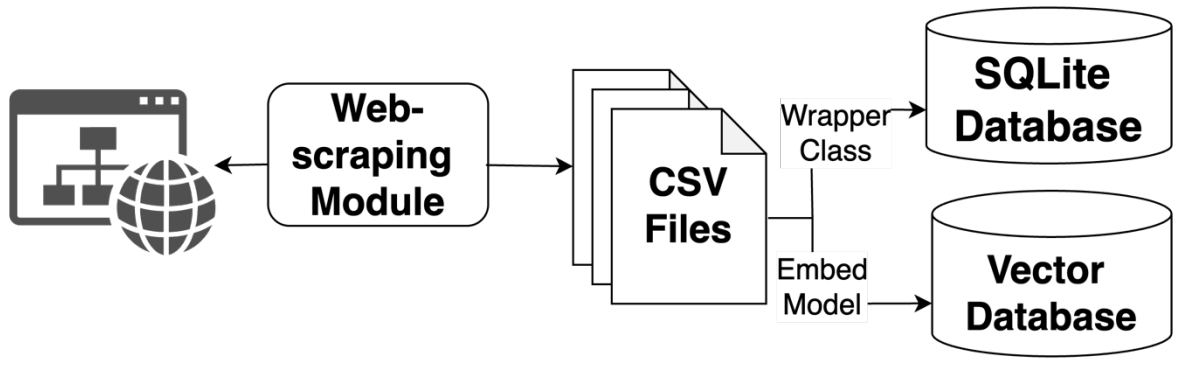

Fig 2. "Data Creation pipeline."

SQLite database creation transforms the scraped CSVs into normalized tables with coherent naming and consistent keys. Each batch/branch file is loaded into a dedicated table and, where present, the regno column is set as the primary key to support unambiguous joins and fast lookups. The loader script walks the results directory tree, reads each CSV via pandas,

and writes tables into database. This phase of data pre-processing step results in creation of SQLite database with 9 tables. Each branch with 3 batches (1st, 2nd, 3rd consecutive batches) are considered to be separate tables.

Chroma vector store is constructed to power retrieval-augmented schema grounding. The script connects to the SQLite database, extracts table schema from each table of the database and synthesizes compact documents of the form "Table: <name> Schema: <SQL>," plus a global overview listing all objects. Embeddings of these extracted table schema is computed using SentenceTransformer, then persisted into a chroma PersistentClient collection. This pipeline also manages versioned backups of the persist directory to keep previous indices safe, and ensures atomic insertion with batched adds.

This RAG layer allows the agent to retrieve only the relevant table schema(s) at generation time, minimizing token usage, reducing hallucination risk, and increasing SQL Query precision. Together, the SQLite store and the Chroma vector database form a clean, queryable substrate that the agentic layer can reliably leverage.

6.3 Project Structure.

This project follows SOLID principles for programming. Figure 3 illustrates the project structure overview.

```
├── cache
├── config
│   └── settings.yaml
├── data
│   ├── archives
│   │   └── Results.db
│   ├── Results_table_schema.txt
│   └── Results.db
├── improvements_to_be_done.txt
├── notebooks
│   ├── database.ipynb
│   └── scrape.ipynb
├── requirements.txt
├── Results_csv
│   ├── 1st_batch
│   │   ├── 1_batch_AIDS_Results.csv
│   │   ├── 1_batch_AIML_Results.csv
│   │   └── 1_batch_CSD_Results.csv
│   ├── 2nd_batch
│   │   ├── 2_batch_AIDS_Results.csv
│   │   ├── 2_batch_AIML_Results.csv
│   │   └── 2_batch_CSD_Results.csv
│   └── 3rd_batch
│       ├── 3_batch_AIDS_Results.csv
│       ├── 3_batch_AIML_Results.csv
│       └── 3_batch_CSD_Results.csv
├── src
│   ├── __init__.py
│   ├── agents
│   │   ├── __init__.py
│   │   ├── data_operator.py
│   │   ├── schema_retriever.py
│   │   ├── sql_query_generator.py
│   │   └── state.py
│   ├── main.py
│   ├── test.py
│   ├── tools
│   │   ├── __init__.py
│   │   ├── db_utils.py
│   │   ├── human_in_loop.py
│   │   └── vector_search.py
│   └── utils
│       ├── __init__.py
│       └── llm_client.py
└── vector_store
    └── Results
        └── chroma_data
            ├── 385bc4c0-c7ea-46b8-89f5-5f6c04ed2fe7
            │   ├── data_level0.bin
            │   ├── header.bin
            │   ├── length.bin
            │   └── link_lists.bin
            └── chroma.sqlite3
```

Fig 3. Project Structure.

6.4 Agentic Pipeline.

The core of this project is its agentic pipeline, a sophisticated workflow designed to provide a trustworthy, modular, and efficient path from a user's natural language request to an executed SQL query with verifiable intent alignment. The architecture decomposes the complex Text-to-SQL problem into specialized, manageable roles: analysed schema retrieval, query generation, verification, and execution while preserving state and enabling human intervention at critical ambiguity points. This implementation uses LangGraph to structure the workflow as a directed, conditional graph with clearly defined nodes and transitions,

effectively preventing infinite loops and managing API calls to ensure both robustness and cost efficiency.

6.4.1 Schema Retriever and Analysis Agent.

The Schema Retriever and Analysis Agent acts as the initial cognitive layer and gatekeeper of the entire pipeline. Its primary objective is to deeply understand the user's intent and ground the request in the context of the available database schema before any attempt is made to generate SQL. This front-loaded analysis is critical to preventing the premature generation of ungrounded, inaccurate, or irrelevant queries, which is a common failure mode in simpler Text-to-SQL systems. This "agent" is not a single monolithic function but rather a sub-workflow within the LangGraph, composed of several interconnected nodes that work in concert including: (1) thought node, (2) semantic search node, (3) table confirmation node, and (4) action node.

The workflow begins at the thought node, which is powered by Google's Gemini model. This node ingests the initial natural language query from the user and performs a comprehensive first-pass analysis. Its core responsibility is to reconstruct the user's request into a structured representation. Using a carefully engineered prompt, it extracts key entities, determines the likely SQL command type such as: SELECT, ALTER and makes an initial inference about the target database table(s). Gemini was specifically chosen for this task due to its sematic analytical capabilities and generating structured JSON output, which is essential for passing clean, machine-readable data into the agentic states.

The output of this node dictates the next step in the graph. If Gemini identifies a table with high confidence, the workflow proceeds towards confirmation. However, if the table name is assessed as "UNCERTAIN" but relevant entities have been extracted, the system intelligently routes the process to the semantic search node. This node acts as a powerful fallback mechanism, leveraging a Chroma-DB vector store and a Sentence-Transformer embedding model (which was used for vector database creation). It takes the entities extracted by Gemini and performs a semantic search against the embedded database schemas. This Retrieval-Augmented Generation (RAG) approach allows the system to discover potentially relevant tables even if the user’s language is fuzzy or does not exactly match the table names. For example, a query about "first-year failures" might be semantically linked to tables containing supplementary exam columns for the first batch.

The most critical juncture in this phase is ambiguity resolution, which is handled by the table confirmation node. This node embodies the HIL principle and is a cornerstone of the system's trustworthiness. Instead of making an autonomous, and potentially incorrect, decision when multiple candidate tables are found or when confidence is low, the agent pauses the

LangGraph execution. It does this by populating the state with the candidate tables and setting an awaiting user input flag. The Stream-lit UI detects this state and presents the options to the user in a clean, interactive format. This user confirmation step is vital, as it resolves ambiguity at its source and ensures the subsequent stages of the pipeline operate on a user-validated foundation.

Once a table has been definitively identified, either directly by the thought node or through user confirmation the action node is triggered. This node performs the final action of the retrieval phase: it connects to the Chroma-DB vector store using the confirmed table name as a precise key and retrieves the exact Table Schema definition. This retrieved schema, along with the entities and command type from the initial analysis, is then written into the Agent-State. This enriched, grounded context is then passed to the next agent in the pipeline, ensuring that the SQL generator has all the necessary information to perform its task accurately.

6.4.2 SQL Query Generator Agent.

Following the successful retrieval of database table schema, the workflow transitions to the SQL Generation Agent. This agent is designed as a specialized craftsman, with the singular, focused responsibility of translating the well-defined, context-rich problem specification from the previous stage into a precise and executable SQL query. This specialization is a key architectural choice, allowing the system to leverage a model highly optimized for code generation, rather than relying on a general-purpose model that may be less adept at handling the rigid syntax and structural nuances of SQL. This agent is embedded as the SQL query generator node within the LangGraph.

The activation of this node is strictly conditional. It only runs after the Schema Retriever agent has successfully populated the Agent-State with a confirmed table name and its corresponding table schema. This ensures that the generator never operates on incomplete or unverified information. The core tool utilized by this agent is Mistral's Codestral model, accessed via its API. Codestral was deliberately selected for this role due to its SOTA performance on code generation tasks. It is fine-tuned on a massive corpus of code, including various SQL dialects, makes it exceptionally proficient at understanding schema definitions and generating syntactically correct and logically sound queries that adhere to the specific dialect in use, which in this case is SQLite.

The interaction model for this agent is linear and straightforward. It consumes the existing state which includes the original natural language query, the extracted entities such as command-type, the confirmed table name(s), and the retrieved schema and its sole output is a string containing the generated SQL query. This output is then written back to the Agent-

State under the generated SQL key. The agent does not interact with the user, the database, or any other external tools; its function is purely translational.

The effectiveness of this agent hinges on meticulous prompt engineering. The prompt sent to Codestral is dynamically constructed to be both comprehensive and concise. It includes the high-level database context, the user's exact query, the list of relevant entities, and, most importantly, the precise CREATE TABLE schema retrieved in the previous step. The prompt contains explicit instructions tailored to the project's database, guidance on how to construct UNION clauses for queries spanning multiple tables, and the specific business logic for interpreting supplementary exams. To ensure deterministic and high-quality output, the API call is configured with a low temperature parameter.

This agent also plays a role in the system's robustness through its relationship with the downstream verification agent. If the Data Operator detects that a generated query is incomplete or truncated (often due to token limits), it can trigger a retry of this SQL Generation Agent.

In such cases, the state is updated with an increased token budget, and this node is called again, allowing it to generate a more complete query on the second attempt. This feedback loop makes the generation process more resilient to the inherent limitations of LLM token outputs.

6.4.3 Data Operator Agent.

The Data Operator and Verification Agent represents the final, and arguably most critical, stage of the agentic pipeline. It serves as the system's quality assurance engine and executioner, embodying the "verification-first" principle that underpins the project's commitment to trust and safety. Its primary function is to ensure that no unvetted, syntactically flawed, or semantically incorrect SQL query is ever executed against the live database. This agent provides a final checkpoint that validates the output of the SQL Generation Agent, corrects it if necessary, executes it securely, and captures both the results and the reasoning behind its verification process for user transparency. This agent is implemented as the data operator node and employs DeepSeek-R1 for its quick reasoning capabilities.

The workflow of this agent is multi-faceted. Upon receiving the generated SQL key from the agentic state, it first initiates a verification call to the DeepSeek R1 model. The prompt for this call is structured as a verification task: it provides the model with the original natural language query, the database schema, and the SQL query generated by Codestral, and asks for a structured JSON response evaluating the query's correctness. DeepSeek's role here is to act as an impartial "code reviewer." It assesses the query against two main criteria: syntactic

validity (is it executable in SQLite?) and semantic alignment (does it accurately reflect the user's intent?).

The agent's next action is determined by the status field in DeepSeek's JSON response.

- If the status is perfect, the agent proceeds directly to the execution phase.
- If the status is corrected, indicating that DeepSeek found and fixed a minor error (e.g., a missing quote, a slight syntax deviation), the agent discards the original query and adopts the corrected SQL provided by DeepSeek. This self-correction capability significantly enhances the system's resilience.
- If the status is incomplete, which typically occurs if the SQL generator's output was truncated, the agent initiates a controlled retry loop. It increments a retry count in the state and increases the current max tokens budget. The “should continue”(conditional edge) router in LangGraph detects this state change and sends the workflow back to the SQL Generation Agent for another attempt with more resources which makes this workflow cyclic and effective in automation. This prevents the system from getting stuck on truncated outputs.
- If the status is error or any other failure condition, the process is halted, and an error is logged.

Once a query is deemed valid (either "perfect" or "corrected"), the agent moves to its execution role. Using Python's native sqlite3 library, it establishes a connection to the database and executes the verified SQL query. This direct interaction with the database is only permitted after the AI-powered verification step, providing a crucial safety layer. The agent fetches all resulting rows and column names from the cursor and formats the data into a list of dictionaries, which is a clean and easily displayable format for the Stream-lit UI.

A key feature of this agent is its commitment to transparency. In addition to the query results, it extracts the detailed reasoning text from the DeepSeek R1 API response. This reasoning explains why DeepSeek approved or corrected the query, providing invaluable insight into the verification process. Both the query results and this reasoning narrative are written to the Agent-State. This allows the Stream-lit UI to not only show the user the final answer but also explain how the system arrived at it and verified its correctness, which is fundamental to building user trust.

6.4.4 Agent Prompt Calibration.

Prompting a model in agentic work flow demands a perfect blend of LLM understanding, roles to be managed by the agent and optimisation. This blend is called as Prompt Engineering. We propose our work in calibrating the agents according to their roles and

optimising them by utilising our novel system design. All the three agents are calibrated using custom prompts using the prompts listed in the subsequent sections.

6.4.4.1 Schema Retriever Agent.

Prompt for schema retriever agent defining their goals and methodology.

"""

Database Context: This is a custom SQLite database for student exam results with 9 tables grouped by batch (1_batch, 2_batch, 3_batch) and branch (AIML, CSD, AIDS). Table names: 1_batch_AIML_Results, 1_batch_CSD_Results, 1_batch_AIDS_Results, 2_batch_AIML_Results, 2_batch_CSD_Results, 2_batch_AIDS_Results, 3_batch_AIML_Results, 3_batch_CSD_Results, 3_batch_AIDS_Results.

- Common columns across tables: 'regno' (TEXT, primary key), 'name' (TEXT), 'semester' (TEXT), 'avg gpa' (REAL).

- Other columns are GPA for specific exam periods (e.g., 'DECEMBER - 2023 gpa' as REAL). Column names may have spaces and use double quotes in SQL.

- Supplementary exams: 'NOVEMBER 2022 gpa'/'NOVEMBER 2023 gpa' for 1st batch, 'NOVEMBER 2023 gpa'/'AUGUST 2024 gpa' for 2nd batch, 'AUGUST 2024 gpa' for 3rd batch. Missing regular GPA implies failure; supplementary GPA shows recovery.

- Gold medal: 'avg gpa' > 9.00.

- Queries may imply multiple tables (e.g., compare batches) or filters on GPA, supplementary status, or gold medals.

Given the natural language query: "{nl_text}"

Perform comprehensive analysis using the database context:

1. Identify the database table name(s) referenced (exact match preferred, or infer from batch/branch/exam mentions; list multiple as comma-separated if query implies joins; use 'UNCERTAIN' if unclear).

2. Extract key entities/attributes mentioned (column names like 'avg gpa', values like '9.00', exam periods, batches, branches).

3. Determine the SQL command type (SELECT, INSERT, UPDATE, DELETE, ALTER, CREATE, DROP, etc.), considering filters like GPA > 9.00 for gold medals or supplementary logic.

4. Check if it's an ALTER/modification command.

5. Generate the SQL command if it's an ALTER operation (SQLite-compatible, e.g., ALTER TABLE "table_name" ADD COLUMN ...).

Return a JSON object with:

- 'table_name': exact table name(s) as string (comma-separated if multiple) or 'UNCERTAIN'
- 'extracted_entities': list of column names, attributes, or key terms mentioned (e.g., ['avg gpa', 'AUGUST 2024 gpa', 'gold medal'])
- 'sql_command_type': the primary SQL operation type
- 'is_alter': boolean for structural modifications
- 'alter_command': SQL ALTER statement if applicable (SQLite-compatible)
- 'confidence': confidence score (0-1) for table identification

Examples:

For "Get students with avg gpa > 9.00 in 2nd batch AIML for gold medal":

{{
"table_name": "2_batch_AIML_Results",
"extracted_entities": ["avg gpa", "gold medal", "2nd batch", "AIML"],
"sql_command_type": "SELECT",
"is_alter": false,
"alter_command": "",
"confidence": 0.95
}}

For "Compare avg gpa between 1st batch CSD and 3rd batch AIDS":

{{
"table_name": "1_batch_CSD_Results,3_batch_AIDS_Results",
"extracted_entities": ["avg gpa", "1st batch", "CSD", "3rd batch", "AIDS"],
"sql_command_type": "SELECT",
"is_alter": false,
"alter_command": "",
"confidence": 0.8
}}

For "Add a column for notes in 3_batch_AIML_Results":

{{
"table_name": "3_batch_AIML_Results",

```
    "extracted_entities": ["notes", "column"],
    "sql_command_type": "ALTER",
    "is_alter": true,
    "alter_command": "ALTER TABLE \"3_batch_AIML_Results\" ADD COLUMN notes
TEXT",
    "confidence": 0.95
  }}

  For "Get names with missing regular GPA but passed supplementary in November 2023
for 2nd batch":
  {{
    "table_name":
"2_batch_AIML_Results,2_batch_CSD_Results,2_batch_AIDS_Results",
    "extracted_entities": ["name", "NOVEMBER 2023 gpa", "supplementary", "2nd batch"],
    "sql_command_type": "SELECT",
    "is_alter": false,
    "alter_command": "",
    "confidence": 0.7
  }}
  """
"""
  Database Context: Custom SQLite database for student exam results with 9 tables grouped
by batch (1_batch, 2_batch, 3_batch) and branch (AIML, CSD, AIDS). Table names:
1_batch_AIML_Results, 1_batch_CSD_Results, 1_batch_AIDS_Results,
2_batch_AIML_Results, 2_batch_CSD_Results, 2_batch_AIDS_Results,
3_batch_AIML_Results, 3_batch_CSD_Results, 3_batch_AIDS_Results.
  - Common columns: 'regno' (TEXT, primary key), 'name' (TEXT), 'semester' (TEXT), 'avg
gpa' (REAL).
  - Exam columns: GPAs as REAL (e.g., 'DECEMBER - 2023 gpa'). Suggest tables based
on batch/branch mentions or exam periods.
  - Supplementary exams: Infer tables for recovery GPAs (e.g., 'NOVEMBER 2023 gpa'
suggests 1st/2nd batch tables).
  - Gold medal: Related to 'avg gpa' > 9.00.

  Given the natural language query: "{nl_text}"
```

And extracted entities: {entities}

Analyze what database tables might be relevant based on these entities and the query context.

Consider batch, branch, exam periods, and common naming patterns.

Return a JSON object with:

- 'suggested_tables': list of likely table names (e.g., '2_batch_AIML_Results')
- 'confidence': confidence score (0-1) for each suggestion
- 'reasoning': why each table might be relevant

Example:

For query "Students with high GPA in 2nd batch AIML" with entities ["high GPA", "2nd batch", "AIML"]:

```
{{
  "suggested_tables": [
    {{"name": "2_batch_AIML_Results", "confidence": 0.95, "reasoning": "Matches batch and branch, contains 'avg gpa'"}},
    {{"name": "2_batch_CSD_Results", "confidence": 0.4, "reasoning": "Same batch but different branch"}}
  ]
}}
"""
```

6.4.4.2 SQL Query Generator Agent.

```
"""
```

Database Context: Custom SQLite database for student exam results with tables by batch (1_batch, 2_batch, 3_batch) and branch (AIML, CSD, AIDS). Tables: 1_batch_AIML_Results, 1_batch_CSD_Results, 1_batch_AIDS_Results, 2_batch_AIML_Results, 2_batch_CSD_Results, 2_batch_AIDS_Results, 3_batch_AIML_Results, 3_batch_CSD_Results, 3_batch_AIDS_Results.

- Common columns: "regno" (TEXT, primary key), "name" (TEXT), "semester" (TEXT), "avg gpa" (REAL).
- Exam columns: GPAs as REAL (e.g., "DECEMBER - 2023 gpa"). Use double quotes for columns with spaces (e.g., "AUGUST - 2024 gpa").

- Supplementary exams: "NOVEMBER 2022 gpa"/"NOVEMBER 2023 gpa" for 1st batch, "NOVEMBER 2023 gpa"/"AUGUST 2024 gpa" for 2nd batch, "AUGUST 2024 gpa" for 3rd batch. Missing regular GPA means failure (backlog); use IS NULL for missing and >0 for passed supplementary.

- Gold medal: WHERE "avg gpa" > 9.00.

- Register numbers of the candidates are in the format **se(ad||ai||cd)***. where the se(ad||ai||cd) are in the lower case. Example register number: 22sead53. Note that while query generation, register number is generated in this format.

- For queries about all batches/branches (e.g., "all candidates"), include ALL relevant tables with UNION or joins on "regno". Be concise: group by batch if possible.

Given the natural language query: "{nl_text}"
Detected command type: {sql_command_type}
Relevant entities: {', '.join(extracted_entities)}
Table schema:
{schema}

Generate a valid, executable SQLite3 SQL query that matches the query intent:

- Use exact table/column names from schema (double-quote columns with spaces, e.g., "AUGUST - 2024 gpa").

- For SELECT, include WHERE for filters (e.g., "avg gpa" > 9.00 for gold medals, IS NULL for missing regular GPA indicating backlogs).

- Handle backlog/supplementary logic: Check IS NULL on regular exam columns to detect backlogs; if query is global, union across all batches.

- Use UNION or joins if multiple tables (e.g., INNER JOIN on "regno" for comparisons, UNION for combining results from all batches).

- Keep queries concise: Avoid listing every column manually if not needed; use * if appropriate, but prefer specific columns.

- For backlogs: Identify candidates with any IS NULL in regular GPA columns (indicating failure/backlog).

- Output ONLY the complete SQL query string (must end with ';'), no explanations, code blocks, or partial queries. Ensure it's valid SQLite3 syntax and not truncated.

Examples (output only the SQL):
For "Get gold medal eligible students in 2nd batch AIML":

SELECT "name", "avg gpa" FROM "2_batch_AIML_Results" WHERE "avg gpa" > 9.00;

For "Students who failed regular but passed supplementary in November 2023 for 2nd batch CSD":
SELECT "name" FROM "2_batch_CSD_Results" WHERE "APRIL 2023 gpa" IS NULL AND "NOVEMBER 2023 gpa" > 0;

For "Get all candidates with backlogs across all batches":
SELECT "regno", "name" FROM "1_batch_AIML_Results" WHERE "JUNE 2022 gpa" IS NULL OR "SEPTEMBER 2022 gpa" IS NULL -- (shortened for example)
UNION SELECT "regno", "name" FROM "1_batch_CSD_Results" WHERE ... -- continue for all tables;
"""

6.4.4.3 Data Operator Agent.

"""
Database Context: Custom SQLite database with student results tables (e.g., 1_batch_AIML_Results). Use double quotes for spaced columns.
Schema: {schema}

Natural Language Query: "{nl_text}"
Command Type: {sql_command_type}
Generated SQL: {cleaned_sql}

Task: Verify if this SQL satisfies the query intent and is valid SQLite3 syntax.
- If perfect, return JSON: {{"status": "perfect", "corrected_sql": "{cleaned_sql}"}}
- If small syntax error, correct it and return: {{"status": "corrected", "corrected_sql": "fixed_query"}}
- If incomplete/truncated (e.g., missing parts, cutoff), return: {{"status": "incomplete", "corrected_sql": ""}} (we'll re-generate)
- Output ONLY valid JSON, no extra text.
"""

Conclusion and Future Work

This project delivers a comprehensive, verification-first Text-to-SQL system that fundamentally transforms how users interact with relational databases by converting natural language queries into safe, executable SQL through a sophisticated agentic, retrieval-grounded, and human-centric sequential / cyclic workflow. The implementation represents a significant advancement in democratizing database access, effectively bridging the gap between human intent and database execution without requiring technical SQL expertise.

The system's architecture successfully orchestrates specialized agents, each optimized for distinct phases of the query lifecycle. Google's Gemini serves as the cognitive foundation, performing nuanced intent analysis and entity extraction to understand user requirements with contextual awareness. Mistral's Codestral, leveraging its code-generation expertise, produces schema-aware SQL queries that are syntactically precise and semantically aligned with the extracted intent. DeepSeek R1 functions as the critical verification, token size management and execution layer, ensuring query correctness through reasoning-based validation before database interaction. This multi-agent approach, supported by a Chroma-backed RAG layer for dynamic schema retrieval, achieves reliability and usability in automated SQL generation.

The Stream-lit user interface epitomizes user-centric design, placing complete control in the hands of non-technical users while maintaining professional-grade functionality. The interface seamlessly handles ambiguity resolution through intuitive table confirmation dialogs, exposes finalized SQL queries for transparency and learning, and surfaces DeepSeek's reasoning processes to build user trust and understanding. The feature, human-in-the-loop ensures that users remain informed and empowered throughout the query generation process.

Architecturally, the system demonstrates exceptional engineering balance between precision and operational efficiency. Through carefully scoped calibrated prompts, intelligent node memory management, strategic caching mechanisms, adhering retry policies, and controlled token budgets, the design maintains cost-effectiveness while preserving accuracy. The system's operational versatility spans from simple single-table lookups to complex multi-table analytical patterns, consistently preserving trust through explicit verification checks and human oversight.

Future Work:

1. Vision-Language Integration for Data Visualization: Incorporating specialized VL (Vision-Language) models capable of automatically generating contextual visualizations from large datasets, transforming the system from query execution to

comprehensive insight delivery through intelligent chart generation and dashboard creation.

2. Adaptive Query Complexity Routing: Implementing separate API pathways for easy and complex SQL generation, utilizing difficulty-aware classification to route simple queries to lightweight, cost-efficient models while directing complex queries to high-capacity models with specialized prompts for optimization.

References

[1]. Deng, N., Chen, Y., & Zhang, Y. (2022). Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. Proceedings of the 29th International Conference on Computational Linguistics.

[2]. Gao, D., Chen, W., & Wang, T. (2023). Text-to-SQL Empowered by Large Language Models: A Survey. Proceedings of the VLDB Endowment, 17.

[3]. He, Y., Bai, D., & Jiang, W. (2018). Text-to-SQL Translation with Various Neural Networks. Stanford University.

[4]. Katsogiannis-Meimarakis, G., & Kermanidis, K. L. (2022). A survey on deep learning approaches for text-to-SQL. The VLDB Journal, 32(1), 103-128.

[5]. Liu, M., & Xu, J. (2025). NLI4DB: A Systematic Review of Natural Language Interfaces for Databases. arXiv preprint arXiv:2503.02435.

[6]. Liu, J. (2023). Combining Text-to-SQL with Semantic Search for Retrieval Augmented Generation. LlamaIndex Blog.

[7]. Pourreza, M., & Rafiei, D. (2023). A Survey on Employing Large Language Models for Text-to-SQL. ACM SIGMOD Record, 52(4), 25-36.

[8]. Wang, J., Ng, P., Li, A. H., Jiang, J., Wang, Z., Nallapati, R., Xiang, B., & Sengupta, S. (2022). Improving text-to-SQL semantic parsing with fine-grained query understanding. Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track.

[9]. Zhang, Z., et al. (2019). Text to SQL Query Conversion Using Deep Learning. SSRN.

[10]. Siddiqui, A., Srivastava, P., & Vadupu Lakshman Manikya, P. K. (2025). Building a custom text-to-SQL agent using Amazon Bedrock and Converse API. AWS Machine Learning Blog.

[11]. Zeng, Z., et al. (2023). A Multi-Agent Collaborative Framework for Text-to-SQL. arXiv preprint arXiv:2312.11242.