© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

Natural Language Querying of DUCKDB using LLM

CHANDANA M PALLEGAR[1], G K HARSHITHA[2], POOJITHA K D[3], RAKSHITHA B M[4], PADMAPRIYA H N[5]

[1, 2, 3, 4]Department Of CSE (Artificial Intelligence & Machine Learning) GSSS Institute of Engineering & Technology for Women

[5] Assistant Professor, Dept. of CSE(AI&ML), GSSS Institute of Engineering & Technology for Women

Abstract- Many users work with CSV files to store data, but analyzing this data usually requires knowledge of SQL, which can be difficult for non-technical users. To overcome this problem, this project presents a web-based application that allows users to interact with their data using simple English queries instead of writing SQL commands. The system converts natural language input into SQL queries, making data analysis faster and easier without requiring any programming skills. The proposed system runs completely offline and is developed using the Flask web framework along with a locally hosted pre-trained Large Language Model. This approach ensures data privacy and avoids dependency on cloud-based services. When a CSV file is uploaded, it is processed using Pandas to clean column names, identify data types, and dynamically generate a relational structure inside DuckDB, an in-memory analytical database. The user's query is combined with the dataset schema and processed by the language model to generate accurate SQL statements. These queries are validated to prevent unsafe operations and are then executed on the dataset, with results returned in a structured JSON format. The application provides a simple and interactive user interface built using HTML, CSS, and JavaScript, supporting drag-and-drop file uploads, data preview, and real-time query responses. This project demonstrates an efficient and secure solution for querying structured data using natural language, making it useful for academic, business, and inventory-related datasets. Future improvements may include support for multiple datasets, graphical data visualization, and enhanced model optimization for handling more complex queries.

I. INTRODUCTION

1.1 OVERVIEW

Today, large amounts of structured data are generated in the form of CSV files across different domains such as education, finance, and business. These files store useful information, but extracting insights from them often requires technical skills. Most users are not trained in database languages, which makes data analysis difficult and time-consuming.

Although CSV files are easy to store and share, analyzing them usually requires knowledge of SQL queries. Non-technical users such as teachers or small business owners often depend on tools like Excel or external technical support to answer simple questions, which limits independent decision-making.

To address this problem, this project proposes a local web-based system that allows users to query CSV data using natural language. By combining a locally hosted language model with an in¬memory database engine, the system converts English questions into SQL queries and executes them securely without internet dependency.

1.2 BACKGROUND OF THE STUDY

1.2.1 Evolution of Data Interaction

The way users interact with digital data has changed significantly over time. In the early stages, database systems were highly restrictive, allowing users to access only predefined reports created by developers. Any new requirement required technical modification, making data access slow and inflexible.

Later, graphical tools such as business intelligence dashboards made data exploration more user-friendly. These tools allowed users to filter and visualize data using drag-and-drop features. However, they still required users to understand table structures, column names, and relationships, which limited their usability for non-technical users.

Recently, a more user-centric approach to data interaction has emerged, where users can express their requirements in natural language. Instead of writing queries or using complex interfaces, users can simply state their intent, such as requesting a list of top-performing students. Natural Language Processing plays a crucial role in enabling this approach. Earlier natural language systems depended on strict keyword

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

rules and failed when user terminology did not match the dataset structure, highlighting the need for more flexible and intelligent solutions.

1.2.2 Emergence of Large Language Models
Advancements in deep learning have significantly improved natural language understanding. Transformer-based models introduced a new way of processing text by capturing contextual relationships between words. Models such as BERT, GPT, and T5 demonstrated strong performance in understanding intent and generating structured outputs.

Among these, the T5 architecture treats all tasks as text-to-text problems, making it suitable for converting natural language queries into SQL statements. Instead of translating between spoken languages, the model translates user questions into structured database commands.

Although cloud-based AI models offer high accuracy, they rely on external servers, which raises concerns related to response time, cost, and data security. Uploading sensitive datasets to third-party platforms may violate privacy regulations. To overcome these limitations, this project adopts a local language model approach, ensuring that data processing and query generation occur entirely on the user's system.

1.3 PROBLEM STATEMENT
Despite the availability of advanced Business Intelligence (BI) tools, the process of analyzing adhoc CSV data remains inefficient for non-technical users. The current landscape presents several specific problems:

1. The Syntax Barrier: A user holding a marks.csv file cannot easily answer the question "Who scored above 90 in Math?" without loading it into Excel and applying filters, or importing it into a database and writing SQL. This creates a dependency on technical staff for ad-hoc queries.
2. Static Schema Limitations: Most existing NL-to-SQL solutions are hardcoded for specific databases. They require a pre-defined schema (e.g., a known set of tables and columns). They fail when a user uploads a completely new file with unknown headers.
3. Data Privacy and Security: The most capable AI tools (like ChatGPT) are cloud-based. Uploading sensitive CSV files (containing PII like student names or proprietary sales data) to a public cloud model is a security risk and is often non-compliant with regulations like GDPR or FERPA.
4. Latency and Dependency: Cloud-based solutions require an active internet connection. If the connection drops, the analysis tool becomes useless. Furthermore, API costs can scale unpredictably with usage.

This project seeks to resolve these issues by building a local, offline, dynamic, and schemaagnostic system.

1.4 OBJECTIVES OF THE STUDY
The primary objective of this project is to develop a full-stack web application that allows users to upload raw CSV files and query them using natural language, with the system automatically generating and executing the corresponding SQL queries.

1.4.1 Specific Objectives
1. To develop a Dynamic Preprocessing Module capable of ingesting arbitrary CSV files, sanitizing column headers for SQL compatibility, and inferring data types (Integer, Varchar, Float) without prior knowledge of the file structure.
2. To implement an In-Memory Database Engine using DuckDB to instantly convert static CSV files into queryable relational tables, ensuring sub-second query performance.
3. To integrate a Local Large Language Model (FLAN-T5) optimized for sequence-to sequence tasks, capable of translating natural language prompts into syntactically correct SQL queries within a local environment.
4. To design a Schema Injection Mechanism that dynamically constructs prompts by combining user questions with the extracted metadata (column names) of the uploaded file.
5. To create a User-Friendly Interface using Flask (Backend) and HTML/CSS (Frontend) that visualizes data previews, displays the generated SQL for transparency, and renders query results in an intuitive tabular format.
6. To ensure System Safety by implementing query validation layers that prevent destructive SQL commands (DROP, DELETE) and handle model hallucinations gracefully.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

1.5 SIGNIFICANCE OF THE STUDY

This project holds importance across several areas:

- Educational Institutions: Teachers and school administrators can quickly examine student results. For instance, instead of performing complicated Excel filters, they could simply ask, "Which students failed Physics but passed Math?" and get instant answers.
- Small Business Owners: Retailers can upload their daily sales data and easily ask questions like, "Which product sold the most on Tuesday?" This eliminates the need for expensive business intelligence software or hiring specialized analysts.
- Data Privacy Advocates: By running AI analysis directly on a local machine, the project demonstrates that sensitive data can be processed securely without leaving the user's device, promoting privacy-conscious analytics.
- Developer Community: The project provides a working example of lightweight RAG (Retrieval-Augmented Generation) systems for structured data, demonstrating dynamic schema integration and serving as a guide for developers interested in this emerging AI approach.

1.6 SCOPE AND LIMITATIONS

1.6.1 Scope

- Data Format: The system focuses specifically on .csv (Comma Separated Values) files, as this is the most common export format for legacy systems and Excel users.
- Language: The NLP model is optimized for English input queries.
- Query Type: The system is designed for Read-Only Analytical Queries (SELECT statements). It supports filtering (WHERE), sorting (ORDER BY), limiting (LIMIT), and basic aggregation (COUNT, SUM, AVG).
- Environment: The application is designed to run on a local server (localhost) without external API calls to OpenAI or Google Gemini.

1.6.2 Limitations

- Complex Joins: The current iteration handles single-file analysis. While DuckDB supports joins, the NLP logic for multi-table inference is outside the current scope.
- Hardware Resources: Running a local LLM (even a quantized version) requires a reasonable amount of RAM (minimum 8GB recommended). Performance may vary based on the host machine's CPU/GPU capabilities.
- Context Window: Extremely large CSVs with hundreds of columns may exceed the context window of the prompt passed to the T5 model, potentially requiring advanced truncation strategies not covered in this phase.

1.7 THEORETICAL FRAMEWORK

This project is based on concepts from database systems and modern natural language processing models. It combines structured data querying principles with sequence-based language understanding to convert user queries into executable SQL statements.

- Relational Data Processing:In this system, uploaded CSV files are interpreted as structured tables containing rows and columns. Each row represents a single data record, while each column represents a specific attribute. When a user submits a query, the generated SQL performs basic database operations such as filtering records, selecting required columns, and computing summary values. These operations form the foundation of structured data retrieval and enable accurate analysis of tabular datasets.
- Sequence-to-Sequence Language Modeling:The natural language component of the system is designed using a sequence-to-sequence learning approach. The model takes the user's question along with information about the dataset structure as input and processes it internally to understand intent. Based on this understanding, the model produces a valid SQL query as output. This approach allows the system to handle different query styles and adapt to varying dataset schemas without manual configuration.
- In-Memory Analytical Processing:For efficient execution of generated queries, the system uses an in-memory analytical database engine. Instead of processing data row by row, the engine operates on column-based data structures, which improves execution speed for analytical queries. This method enables faster filtering, aggregation, and sorting operations, even when working with large CSV files.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

1.8 METHODOLOGY OVERVIEW

The system development follows the Agile Software Development Life Cycle (SDLC), specifically the Iterative model. The system is modularized into five distinct components to ensure separation of concerns:

1. Frontend (UI): Handles user interaction, file capture, and result visualization.
2. Preprocessing Engine: Handles the extraction of metadata and data type inference using Pandas.
3. Inference Engine: The "Brain" containing the FLAN-T5 model and tokenizer.
4. Execution Engine: The "Muscle" containing the DuckDB instance for running queries.
5. Controller: The Flask backend orchestrating the flow between these modules.

The development utilizes Python as the primary language due to its rich ecosystem for both Data Science (Pandas, DuckDB) and AI (PyTorch, Transformers).

1.9 ORGANIZATION OF THE PROJECT

The remainder of this report is organized as follows:

- Chapter 2: Literature Review surveys existing approaches to NL-to-SQL generation, comparing rule-based systems, early neural networks, and modern Transformer approaches. It also reviews the capabilities of DuckDB compared to SQLite and PostgreSQL.
- Chapter 3: Methodology and System Design details the architectural diagrams, data flow diagrams (DFD), and the specific schema injection logic used to prompt the AI.
- Chapter 4: Implementation describes the coding phase, the specific libraries used (Transformers, Flask), and the testing methodologies employed to verify SQL accuracy and system stability.
- Chapter 5: Tools and Technologies explains the development environment, the key libraries employed such as Transformers and Flask, and the methods used to ensure SQL query accuracy and overall system reliability.
- Chapter 6: System Testing presents the testing strategies implemented to validate system functionality, including verification of SQL queries, stability assessments, and performance evaluations.
- Chapter 7: Results and Discussion presents the performance metrics of the system, including query accuracy rates and execution times, alongside screenshots of the working application.
- Chapter 8: Conclusion and Future Scope summarizes the findings and proposes future enhancements such as multi-file support and voice-input integration.

1.10 OPERATIONAL DEFINITIONS

To ensure clarity, the following technical terms used throughout this report are defined:

- NLP (Natural Language Processing): A subfield of AI focused on the interaction between computers and human language.
- Inference: The process of using a trained machine learning model to make predictions (in this case, predicting SQL code) based on new input data.
- Schema: The structural definition of a database table, specifically the column names and their data types.
- Tokenization: The process of breaking down text (sentences) into smaller units (tokens) that the AI model can process.
- Zero-Shot Learning: The ability of the model to perform a task (generating SQL for a specific CSV) without having been explicitly trained on that specific CSV file beforehand.
- DuckDB: An in-process SQL OLAP (Online Analytical Processing) database management system used for high-performance data analysis.

II. LITERATURE REVIEW

2.1 OVERVIEW

The domain of Natural Language Interfaces for Databases (NLIDB) sits at the convergence of two major fields in computer science: Database Management Systems (DBMS) and Natural Language Processing (NLP). This chapter surveys the scholarly and technical literature surrounding the development of systems that translate human language into Structured Query Language (SQL). It traces the historical trajectory from early rule-based parsers to contemporary Large Language Models (LLMs). Furthermore, it examines the underlying database technologies—specifically contrasting traditional row-oriented storage with modern in-memory columnar stores like DuckDB—to establish the theoretical justification for the proposed system's architecture.

2.2 EVOLUTION OF NATURAL LANGUAGE INTERFACES FOR DATABASES (NLIDB)

The quest to allow non-technical users to query databases using natural language dates back to the late 1960s. The literature categorizes the evolution of these systems into four distinct generations: Pattern-Matching Systems, Semantic Grammar Systems, Intermediate Representation Systems, and finally, Deep Learning-based Systems.

2.2.1 Early Rule-Based and Pattern-Matching Approaches

Early systems like LUNAR (Woods, 1973) and LADDER (Hendrix et al., 1978) relied heavily on rigid pattern matching. LUNAR, designed to query a database of moon rocks brought back by Apollo 11, utilized an Augmented Transition Network (ATN) parser. While it achieved high accuracy within its microscopic domain, the literature notes its "brittleness"—it could not function outside its specific vocabulary. These systems required hard-coded rules mapping specific English verbs (e.g., "list", "show") to database commands. As noted by Androutsopoulos et al. (1995), the maintenance cost of such systems was prohibitive; adding a new column to the database often required rewriting the parsing rules.

2.2.2 Semantic Grammars and Intermediate Representations

In the 1980s, research shifted toward Semantic Grammars. Systems like TEAM (Grosz et al., 1987) attempted to decouple the linguistic processing from the database schema. They introduced the concept of an Intermediate Logical Form—a simplified, logic-based representation of the user's intent that sits between the English sentence and the final SQL query. This era highlighted a critical challenge that persists today: the "mismatch problem", where the user's conceptual view of data (e.g., "best students") does not map 1:1 to the physical schema (e.g., marks > 90).

2.3 THE DEEP LEARNING ERA: SEQUENCE-TO-SEQUENCE MODELS

The paradigm shifted radically around 2014 with the introduction of Sequence-to-Sequence (Seq2Seq) neural networks.

2.3.1 The Encoder-Decoder Framework

Sutskever et al. (2014) introduced the Encoder-Decoder architecture, primarily for machine translation (English to French). Researchers quickly adapted this for Text-to-SQL.

- The Encoder (typically an LSTM or GRU at the time) processes the input question word by word, compressing it into a fixed-length "context vector."
- The Decoder generates the SQL query token by token based on that context vector. However, early Seq2Seq models struggled with "long-term dependency" issues. If a question was long/complex, the fixed-length vector failed to capture all necessary details, leading to incorrect SQL generation.

2.3.2 Attention Mechanisms and Pointer Networks

To resolve the bottleneck of the fixed-length vector, Bahdanau et al. (2015) introduced the Attention Mechanism. This allowed the model to "focus" on specific words in the input question while generating specific parts of the SQL query. For instance, when generating the SQL snippet WHERE city =, the model attends heavily to the word "city" in the user's question.

A pivotal advancement cited in recent literature is the Pointer Network (Vinyals et al., 2015). In standard text generation, the model predicts words from a fixed vocabulary. In Text-to-SQL, however, many words (like column names or specific values) come directly from the input. Pointer Networks allow the model to "copy" tokens from the input sequence directly to the output sequence. This is crucial for the "Dynamic Schema" requirement of this project; the model doesn't need to "know" the column name Student_Age in advance—it simply learns to copy it from the provided schema context.

2.4 THE TRANSFORMER REVOLUTION AND LARGE LANGUAGE MODELS

The introduction of the Transformer architecture (Vaswani et al., 2017) rendered LSTM-based approaches obsolete by enabling parallel processing and capturing deeper contextual relationships via Self-Attention.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

2.4.1 BERT and Pre-training

Models like BERT (Bidirectional Encoder Representations from Transformers) demonstrated that models could be pre-trained on massive corpora of text to understand general language, then finetuned for specific tasks. Hwang et al. (2019) introduced SQLova, a BERT-based model that achieved state-of-the-art results on the WikiSQL benchmark. SQLova did not generate SQL as a raw string; instead, it filled specialized "slots" (Select-Column, Aggregation-Operator, WhereColumn, etc.). While accurate, this "slot-filling" approach is rigid and struggles with complex nested queries, limiting its utility for general-purpose analysis.

2.4.2 T5 (Text-to-Text Transfer Transformer)

The literature identifies T5 (Raffel et al., 2020) as a unified framework that treats every NLP problem as a text-generation task. Unlike SQLova, T5 generates the raw SQL string directly.

- Significance for this Project: T5 is particularly adept at "schema serialization." Research by Scholak et al. (2021) on the PICARD system demonstrates that T5 models, when constrained by a schema parser, outperform larger models. The FLAN-T5 variant (Wei et al., 2021) utilized in this project is fine-tuned on "instruction" datasets, making it exceptionally capable of following prompts like "Translate this question to SQL based on these columns."

2.4.3 Prompt Engineering and Context Injection

Recent studies focus on "In-Context Learning" (Brown et al., 2020). Instead of fine-tuning a model (updating its weights), developers provide a prompt containing the task description and relevant data. For Text-to-SQL, the literature emphasizes Schema Linking: the process of explicitly connecting words in the question to columns in the schema. Gan et al. (2021) proved that augmenting the prompt with explicit column types (e.g., Name (text), Age (int)) significantly reduces hallucination rates. This theoretical foundation supports this project's Module 3 design, which injects the dynamic CSV schema into the prompt.

2.5 DATABASE TECHNOLOGIES FOR AD-HOC ANALYTICS

While the AI interprets the language, the database engine must execute the logic. The literature distinguishes between OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) systems.

2.5.1 Row-Oriented vs. Column-Oriented Storage

Traditional databases like PostgreSQL and MySQL are row-oriented. They store data record-byrecord. This is efficient for transactional writes (e.g., adding a new user) but inefficient for analytical reads (e.g., calculating the average age of all users), as the system must scan every row's unused columns.

Column-oriented databases (like ClickHouse or Snowflake) store data by column. To calculate an average, the system reads only the specific column block, ignoring the rest. This architecture drastically reduces I/O overhead for analytical queries.

2.5.2 DuckDB: The "SQLite for Analytics"

DuckDB (Raasveldt & Mühleisen, 2019) represents a novel category in the literature: an embeddable OLAP database. Unlike client-server databases that require installation and network protocols, DuckDB runs in-process with the application (similar to SQLite) but utilizes a vectorized execution engine optimized for analytics.

- Relevance to this Project: Literature confirms that DuckDB's zero-copy integration with Python (Pandas) and its ability to query CSV files directly without a distinct "load" phase make it the optimal choice for ad-hoc, serverless analysis of user-uploaded files.

2.6 PRIVACY-PRESERVING AI: THE CASE FOR LOCAL INFERENCE

A growing body of literature addresses the privacy risks of cloud-based LLMs. Carlini et al. (2021) demonstrated that large models can memorize and regurgitate training data, posing risks if sensitive data (like student records) is sent to public APIs.

2.6.1 Edge AI and Local Deployment

"Edge AI" refers to running inference on local devices. The challenge has historically been hardware limitations. However, techniques like Quantization

(reducing model weights from 32bit floating point to 8-bit or 4-bit integers) have made it possible to run powerful models like T5Large on consumer CPUs with minimal accuracy loss (Dettmers et al., 2022). This project leverages these advancements to ensure GDPR/FERPA compliance by keeping all data and inference within the user's local machine.

2.7 SUMMARY OF RELATED WORKS

Table 2.1: Summary of Related Works in NL2SQL Approaches

Approach	Representative System	Methodology	Limitations
Pattern Matching	LUNAR (1973)	Hard-coded dictionary & rule-based matching	Brittle; fails with unknown words
Statistical / Probabilistic	PRECISE (2003)	Graph matching	Requires complex setup; limited to "semantically tractable" questions
Seq2Seq (RNN)	LSTM, Seq2SQL (2017)	Sequence-to-sequence + Reinforcement Learning	Struggles with long queries; no schema awareness
Slot-Filling (BERT)	SQLova (2019)	Classification + slot prediction	Cannot handle nested queries or complex joins
Generative Models	T5 / GPT-based models	Transformer-based generation	May hallucinate; requires large compute
Proposed System	—	Prompt Engineering + Context Awareness	Flexible, Schema-Agnostic, Context-Aware

2.8 GAP ANALYSIS

The literature review reveals a significant gap in current solutions:

1. Complexity Gap: Tools like Tableau require user training.
2. Rigidity Gap: Most academic Text-to-SQL systems (like SQLova) assume a fixed database schema and cannot handle a user uploading a random inventory.csv one minute and grades.csv the next.
3. Privacy Gap: Powerful tools like ChatGPT require data

III. METHODOLOGY AND SYSTEM DESIGN

3.1 OVERVIEW

System analysis is the process of studying a procedure or business to identify its goals and purposes and create systems and procedures that will efficiently achieve them. This chapter provides a detailed analysis of the current methods used for analyzing structured data (CSV files) and contrasts them with the proposed "Intelligent Natural Language to SQL" system. It outlines the limitations of existing approaches—ranging from manual spreadsheet manipulation to complex Business Intelligence (BI) tools—and justifies the architectural decisions made for the proposed local, AI-driven solution.

3.2 ANALYSIS OF THE EXISTING SYSTEM

In the current technological landscape, non-technical users (such as teachers, small business owners, or HR managers) typically rely on three primary methods to analyze the data contained in CSV files.

3.2.1 Method 1: Manual Spreadsheet Manipulation

The most common "system" for analyzing CSV data is Microsoft Excel or Google Sheets.

- Process: The user opens the CSV file, visually scans rows, uses "Filter" menus to narrow down data, or constructs formulas (e.g., =VLOOKUP, =AVERAGEIF).
- Scenario: A teacher wants to find "students who failed Math but passed Physics." In Excel, this requires applying multiple column filters or writing a complex conditional logic formula.

3.2.2 Method 2: Dependency on Technical Staff (SQL)

For larger datasets or more complex questions, non-technical stakeholders must rely on IT departments.

- Process: The stakeholder writes an email describing their need ("I need a report of Q3 sales"). A developer loads the CSV into a database (MySQL/PostgreSQL), writes the SQL query, exports the result, and emails it back.
- Scenario: A manager needs an ad-hoc report immediately during a meeting but must wait 24-48 hours for the IT team to process the request.

3.2.3 Method 3: Cloud-Based AI Tools (ChatGPT/Gemini)

With the rise of GenAI, some users upload their files to public chatbots.

- Process: The user uploads salary_data.csv to ChatGPT and asks, "Who earns the most?"
- Scenario: While effective, this involves sending proprietary or sensitive Personally Identifiable Information (PII) to external cloud servers, often violating data privacy regulations like GDPR or FERPA.

3.3 DISADVANTAGES OF THE EXISTING SYSTEM

The existing methods suffer from significant functional and non-functional drawbacks:

1. High Latency & Inefficiency: The feedback loop in the IT-dependency model is too slow. Data analysis should be conversational and immediate, not a ticketing process that takes days.
2. Steep Learning Curve: Excel formulas and BI tools (like Tableau) require specific training. A user cannot simply "ask" the data; they must know the syntax of the tool.
3. Data Privacy Risks: Cloud-based AI tools act as a "Black Box." When a user uploads a CSV, they lose control over where that data is stored or if it is used to train future models. This is unacceptable for medical, financial, or educational data.
4. Static Schema Limitations: Most traditional SQL dashboards are hard-coded. If a user receives a new CSV with different column names (e.g., "Student_Name" vs. "Full_Name"), the existing dashboard breaks, requiring code refactoring.
5. Lack of Transparency: When using black-box AI tools, the user rarely sees how the answer was derived. They get a number but cannot verify if the logic used to get that number was correct.

3.4 THE PROPOSED SYSTEM

The proposed system is a Local, Privacy-First, Natural Language Query Engine. It is designed to run entirely on the user's machine (Localhost), eliminating the need for internet connectivity or cloud uploads. It automates the role of the "Database Administrator" by using an AI model to translate English questions into SQL queries, which are then executed against a high-speed inmemory database.

3.4.1 Architectural Components

1. The Interface Layer (Frontend):
- A web-based UI (HTML5/CSS3) that serves as the entry point.
- Features a "Drag-and-Drop" zone for CSV files and a "Chat Interface" for asking questions.
- Innovation: It includes a "Data Preview" panel that gives users immediate visual confirmation of their uploaded data.
2. The Intelligence Layer (Local LLM):
- Utilizes a Text-to-Text Transfer Transformer model hosted locally.
- Function: It receives a prompt containing the User's Question + The Table Schema. It outputs a raw SQL query.
- This removes the dependency on OpenAI/Google APIs, ensuring zero data leakage.
3. The Data Layer (DuckDB):
- Replaces traditional heavy databases (MySQL) with DuckDB, an in-process SQL OLAP engine.
- Function: It treats the uploaded CSV file as a virtual table. It executes the SQL generated by the AI and returns the result in milliseconds.
- Dynamic Schema: DuckDB automatically infers column types (Integer, String, Date), allowing the system to handle any CSV file structure without manual configuration.
4. The Orchestration Layer (Flask Controller):
- A Python Flask server acts as the glue. It manages the session, sanitizes inputs, handles errors (e.g., if the AI generates invalid SQL), and formats the JSON response for the frontend.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

3.5 ADVANTAGES OF THE PROPOSED SYSTEM

The proposed system addresses every major flaw identified in the existing system analysis:

3.5.1 democratization of Data Analysis

- Advantage: Zero Coding Required. Users interact with data using natural language (English). This empowers non-technical staff (HR, Admin, Sales) to perform their own analysis without waiting for IT support.

3.5.2 Enhanced Data Privacy & Security

- Advantage: 100% Offline Execution. Since the AI model (T5) and the Database (DuckDB) run locally, sensitive data never leaves the user's computer. This makes the system compliant with strict data governance policies (e.g., analyzing internal salary spreadsheets).

3.5.3 Universal Adaptability (Dynamic Schema)

- Advantage: File-Agnostic Design. Unlike a dashboard built for a specific database, this system adapts to any structured CSV. A user can analyze a "Class Marks" file in the morning and a "Stock Inventory" file in the afternoon without restarting or reconfiguring the system.

3.5.4 Transparency and Explainability

- Advantage: "Glass Box" AI. The system displays the generated SQL query alongside the result.
- User asks: "Show me top 5 students." o System shows: SELECT Name FROM students ORDER BY Marks DESC LIMIT 5 o This builds trust, allowing users to verify that the AI understood the logic correctly.

3.5.5 Performance and Efficiency

- Advantage: In-Memory Speed. By using DuckDB's columnar vectorization, queries on large CSVs (100k+ rows) are executed in sub-second timeframes, significantly faster than Excel's calculation engine or traditional row-based Python iteration.

3.6 FEASIBILITY STUDY

To ensure the viability of the proposed system, a feasibility study was conducted across three dimensions:

3.6.1 Technical Feasibility

- Hardware: The system is designed to run on standard consumer laptops (8GB RAM recommended). The use of the "Small" or "Base" version of FLAN-T5 ensures inference is computationally inexpensive compared to giant models like Llama-2.
- Software: All core technologies (Python, Flask, Transformers, DuckDB) are open-source, mature, and well-documented libraries.
- Conclusion: The project is technically feasible.

3.6.2 Operational Feasibility

- Usability: The UI mimics standard chat applications (like WhatsApp or ChatGPT), ensuring users intuitively understand how to use it without training manuals.
- Workflow: The process (Upload -> Ask -> View) fits naturally into existing business workflows.
- Conclusion: The system will be easily adopted by the target audience.

3.6.3 Economic Feasibility

- Cost: The project utilizes entirely Open Source Software (OSS). There are no licensing fees for the database (DuckDB is MIT licensed) and no API costs for the AI (Local Inference is free).
- Maintenance: The "Schema-Agnostic" design reduces maintenance costs, as the code does not need updating when data structures change.
- Conclusion: The system is highly economically viable.

3.7 ER DIAGRAM

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

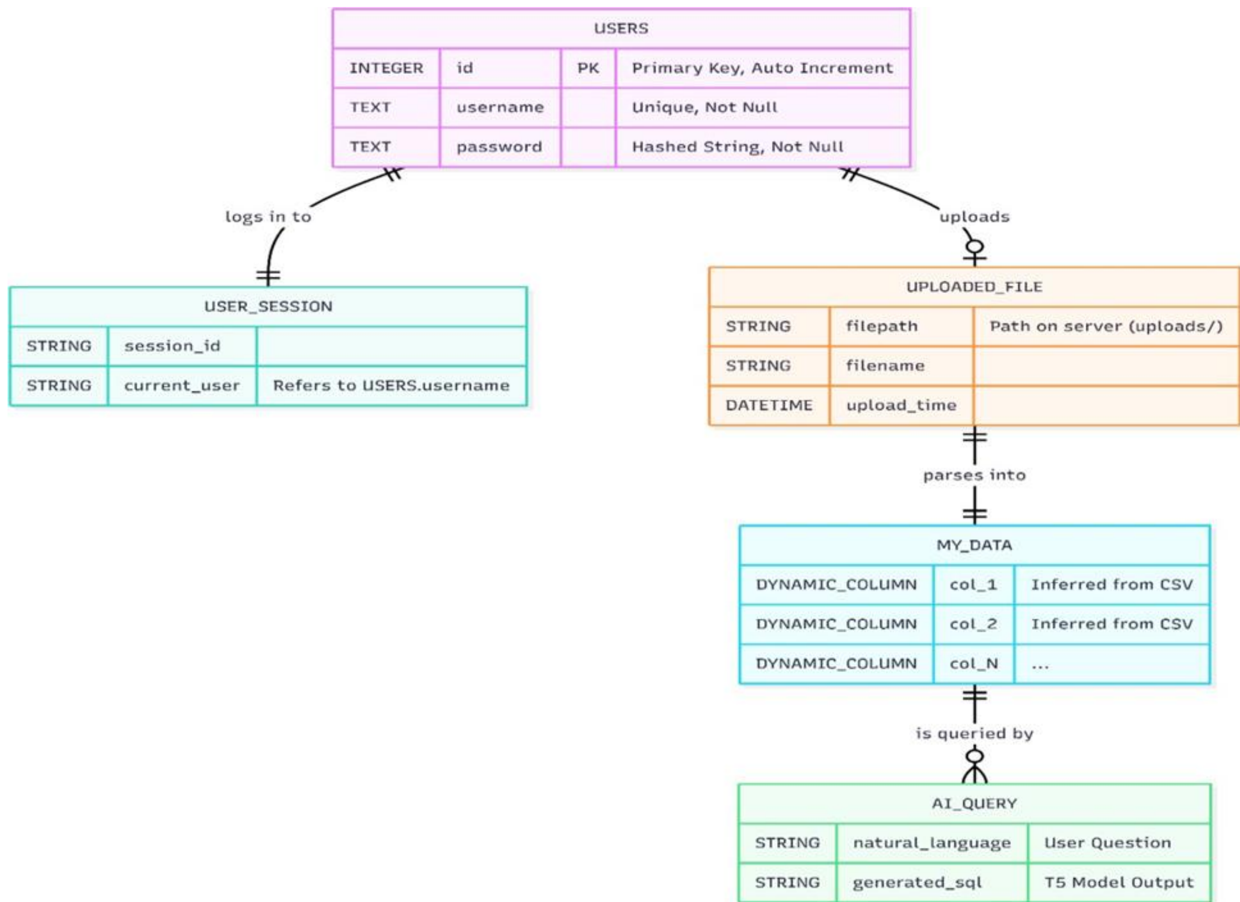

Figure 3.1: ER Schema Representing User, File, Data, and Query Flow

1. USERS Table
- Stores user credentials (username, password).
- In your report, you have a login/authentication layer (Module 1, Flask-based).
- This table would be used for that purpose.
2. USER_SESSION Table
- Tracks logged-in users via session_id.
- This matches your report's session-based approach to keep track of who is uploading/querying data.
3. UPLOADED_FILE Table
- Stores metadata about uploaded CSV files (filepath, filename, upload_time).
- Your system saves uploaded CSV files in an uploads/ folder and tracks them for later querying.
4. MY_DATA Table
- Represents the dynamic table created from the CSV.
- Columns (col_1, col_2, …) are inferred at runtime (as described in your report's Module 4).
- This is the table that DuckDB uses for SQL execution.
5. AI_QUERY Table
- Stores the natural language query (natural_language) and the generated SQL (generated_sql).

3.8 DATA FLOW DIAGRAM (DFD)

The flow of data through the proposed system can be summarized in four stages:

1. Input: User uploads data.csv.
2. Processing (Stage 1): Backend cleans headers and infers schema types.
3. Processing (Stage 2): User asks a question -> AI converts Question + Schema into SQL.
4. Processing (Stage 3): DuckDB executes SQL on the CSV.
5. Output: Result is returned as JSON and rendered as an HTML Table.

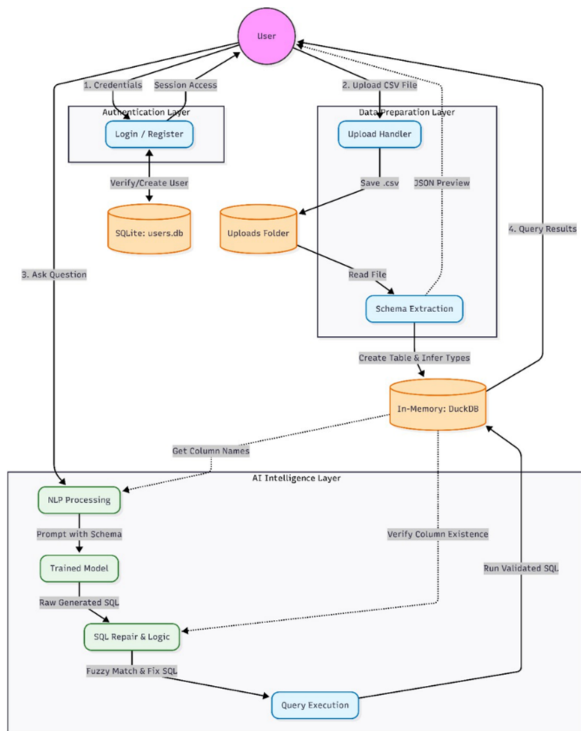

Figure 3.2 :Functional Architecture of the Natural Language to SQL Querying System

3.9 CONCLUSION

The System Analysis confirms that the current reliance on manual spreadsheet tools or cloud-based APIs creates significant bottlenecks in efficiency and privacy. The proposed "Intelligent Natural Language to SQL" system offers a robust, technically feasible alternative. By leveraging local AI inference and in-memory database acceleration, it provides a secure, fast, and user-friendly platform that effectively democratizes data access for non-technical users.)

IV. IMPLEMENTATION

4.1 OVERVIEW

The Implementation phase marks the transition from theoretical design to the construction of the actual software artifact. This chapter details the technical realization of the "Intelligent Natural Language to SQL Generation System." It describes the development environment, the specific libraries employed, and the modular architecture used to build the application.

Unlike traditional monolithic scripts, this system is architected into eight distinct modules. This separation of concerns ensures maintainability, scalability, and ease of debugging. Each module handles a specific aspect of the pipeline—from capturing user input to executing high-performance database queries on local hardware.

4.2 DEVELOPMENT ENVIRONMENT

Before detailing the modules, it is essential to define the hardware and software constraints under which this implementation operates.

1. Operating System: Windows 10/11 or Linux (Ubuntu 20.04+).
2. Language: Python 3.9+ (chosen for its dominance in AI and Data Engineering).
3. Primary Frameworks:

- Flask (v2.0): For the web server and API controller.
- DuckDB (v0.8+): For the in-memory SQL execution engine.
- Hugging Face Transformers: For loading and running the FLAN-T5 model.
- Pandas: For initial CSV manipulation and schema inference.

4. Hardware Requirements:

- RAM: Minimum 8GB (16GB recommended for smoother model inference).
- Storage: ~2GB for model weights and library dependencies.

4.3 MODULAR DECOMPOSITION

The system is implemented via the following eight modules:

1. Module 1: User Interface (UI) & Interaction Module
2. Module 2: Application Controller (Flask Orchestrator)
3. Module 3: Data Ingestion & Sanitization Module
4. Module 4: Dynamic Schema Generation Module
5. Module 5: Model Loading & Initialization Module
6. Module 6: Prompt Engineering & Context Injection Module
7. Module 7: Query Security & Validation Module
8. Module 8: Execution & Result Formatting Module

4.4 MODULE 1: USER INTERFACE (UI) & INTERACTION MODULE

4.4.1 Description

This module is the "Face" of the application. It is responsible for providing a seamless, nontechnical experience for the user. It is built using HTML5, CSS3, and Vanilla JavaScript. The design philosophy is "Single Page Application" (SPA) behavior, meaning the page never reloads; all data is exchanged asynchronously.

4.4.2 Implementation Details

- File Upload Area: Implemented as a drag-and-drop zone. It listens for drop events, prevents the browser's default behavior (opening the file), and instead triggers the upload function.
- Asynchronous Communication: The fetch() API is used for all backend calls. This prevents the "white screen" effect during the 2-5 seconds the AI takes to generate a query.
- Dynamic Table Rendering: A custom JavaScript function renderTable(data) accepts JSON arrays and dynamically generates HTML <table> headers and rows. This allows the UI to display any CSV file regardless of its column count.

4.4.3 Key Challenges

- Handling Large Files: Initially, trying to render a 10,000-row CSV froze the browser. The implementation was optimized to only render the first 10 rows (head) for the preview, while the backend handles the full dataset.
- Managing Real-World Data Heterogeneity and Ingestion: Beyond initial browser rendering limitations, processing real-world CSV files involved handling inconsistent data formats, corrupted entries, and encoding issues. The system needed to robustly parse diverse data types (dates, currencies, percentages) and clean malformed headers with special characters, all while maintaining SQL compatibility and ensuring no data loss during sanitization.
- Ensuring AI Reliability Against Hallucinations and Ambiguity: The local FLAN T5 model sometimes generated syntactically valid but semantically incorrect SQL—such as selecting wrong columns, applying incorrect aggregations, or inventing non-existent schema elements. This was compounded by ambiguous user queries (e.g., “show best results”) where intent wasn’t explicit.
- Optimizing Performance Within Local Hardware Constraints: Balancing the computational demands of the LLM and in-memory database with the limitations of consumer grade hardware (e.g., 8GB RAM). Large file uploads caused memory spikes, while model inference times impacted responsiveness.
- Securing the System Against Prompt and SQL Injection: The system’s open input channels—natural language questions and raw CSV data—presented risks. Users could inadvertently or maliciously craft inputs that manipulated the AI prompt or generated harmful SQL, bypassing initial validation checks
- Achieving Robust Error Handling and User Communication: Transforming technical failures (e.g., DuckDB execution errors, model timeouts, or unsupported queries) into intuitive, actionable feedback for non-technical users was critical. The system needed to fail gracefully and guide users toward successful queries without exposing internal complexities.

4.5 MODULE 2: APPLICATION CONTROLLER (FLASK ORCHESTRATOR)

4.5.1 Description

The Application Controller is the central hub built on the Flask micro-framework. It does not perform heavy computation itself; rather, it routes traffic between the UI and the processing modules. It manages the HTTP request/response cycle and maintains the user session state.

4.5.2 Implementation Logic

The module defines three primary routes:

1. GET /: Serves the index.html file (Module 1).
2. POST /upload: Receives the raw file stream, passes it to Module 3 (Ingestion) and Module 4 (Schema), and returns a success status.
3. POST /ask: Receives the user’s natural language string, passes it through Modules 6, 7, and 8, and returns the final JSON result.

4.5.3 Session Management

Since HTTP is stateless, the Controller must remember which file the user is currently analyzing. It uses a server-side global dictionary (or Flask session

object) to store the active table name associated with the user's session ID.

4.6 MODULE 3: DATA INGESTION & SANITIZATION MODULE

4.6.1 Description

Real-world data is messy. This module acts as the "Janitor" of the system. Before any AI or Database logic can occur, the raw CSV must be cleaned to prevent errors.

4.6.2 Algorithm: Header Sanitization

The core function sanitize_headers(df) performs the following operations using Pandas:

1. Load CSV: Reads the file into a Pandas DataFrame.
2. Whitespace Removal: Strips leading/trailing spaces from column names (e.g., " Name "
becomes "Name").
3. Space Replacement: Replaces internal spaces with underscores (e.g., "Total Marks" becomes "Total_Marks"). This is critical because SQL requires bracketed syntax [Total Marks] for spaces, which is harder for the AI to predict reliably.
4. Special Character Removal: Removes symbols like %, $, or # that might be interpreted as operators in SQL.

4.6.2 Code Snippet (Python)

```
def clean_headers(df):
df.columns = df.columns.str.strip() df.columns =
df.columns.str.replace(' ', '_')
df.columns = df.columns.str.replace(r'[^\w]', '',
regex=True) return df
```

4.7 MODULE 4: DYNAMIC SCHEMA GENERATION MODULE

4.7.1 Description

This module is responsible for the "Dynamic" capability of the system. It bridges the gap between the Python Pandas environment and the DuckDB SQL environment.

4.7.2 Implementation: The "Virtual Table" Strategy

Unlike traditional systems that require a CREATE TABLE statement with defined types (e.g., VARCHAR(50)), this module leverages DuckDB's direct integration with Pandas.

1. Type Inference: The module 26nalyses the Pandas dtypes (int64, float64, object).
2. Registration: It registers the cleaned DataFrame as a virtual view in the DuckDB connection using con.register('uploaded_data', df).
3. Schema Extraction: It extracts the column names and their inferred types into a string format (e.g., "id: int, name: text, score: int") which will be passed to the AI module later.

4.8 MODULE 5: MODEL LOADING & INITIALIZATION MODULE

4.8.1 Description

This is the most resource-intensive module. It handles the loading of the Large Language Model from the local disk into the system's RAM.

4.8.2 Implementation Details

To ensure performance, this module implements the Singleton Pattern. The model is loaded only once when the Flask server starts, not every time a user asks a question.

- Library: transformers.
- Model: pretrained transformer model
- Tokenizer: The T5Tokenizer is loaded to convert English text into the numerical tokens the model understands.

4.8.3 Performance Optimization

To run on consumer hardware, the implementation optionally uses quantization or simply selects the smaller model variants. The loading process is wrapped in a try-catch block to handle "Out of Memory" (OOM) errors gracefully.

4.9 MODULE 6: PROMPT ENGINEERING & CONTEXT INJECTION MODULE

4.9.1 Description

This module is the intellectual core of the project. It constructs the input "Prompt" that guides the AI to generate SQL. A generic prompt produces generic results; a specific prompt produces executable SQL.

4.9.2 The Prompt Template

The implementation uses a strict templating system. The prompt is constructed dynamically at runtime:

Task: Translate English to SQL. Table Name: uploaded_data

Schema: [Student_Name, Physics_Marks, Math_Marks] Question: “Who got the highest math marks?” SQL:

4.9.3 Logic

1. Receive Inputs: Takes the User Question and the Schema String (from Module 4).
2. Concatenation: Fills the template slots.
3. Tokenization: Converts the prompt to tensors using tokenizer(prompt, return_tensors=”pt”).
4. Generation: Calls model.generate() to produce the output sequence (the SQL).

4.10 MODULE 7: QUERY SECURITY & VALIDATION MODULE

4.10.1 Description

AI models can hallucinate (make up facts) or generate dangerous code. This module acts as a firewall between the AI’s output and the database execution.

4.10.2 Validation Logic

1. Keyword Blacklisting: The module scans the generated SQL string for forbidden keywords like DROP, DELETE, INSERT, UPDATE, or TRUNCATE. The system is designed for Read-Only Analysis, so any modification command is rejected immediately.
2. Syntax Checking: It performs a basic check to ensure the query starts with SELECT.
3. Hallucination Check: It verifies that the column names used in the generated SQL actually exist in the CSV schema. If the AI invents a column Grade when the file only has Mark, this module catches the error before it crashes the database.

4.11 MODULE 8: EXECUTION & RESULT FORMATTING MODULE

4.11.1 Description

The final module executes the validated SQL query and formats the raw data for display.

4.11.2 Implementation with DuckDB

- Execution: Uses duckdb.query(sql_string).
- Data Conversion: The raw result from DuckDB is usually a list of tuples. This module converts it into a List of Dictionaries (JSON) format: [{‘Name’: ‘Alice’, ‘Mark’:90}, {‘Name’: ‘Bob’, ‘Mark’: 85}].
- Error Handling: If the execution fails (e.g., due to a logical error like dividing by zero), this module catches the duckdb.Error exception and returns a user-friendly message (“I couldn’t calculate that. Please try rephrasing.”) instead of a server 500 error.

4.12 INTEGRATION AND DATA FLOW

The integration of these 8 modules follows a strict synchronous flow for the user request:

1. Frontend sends request.
2. Controller receives request.
3. Sanitization cleans input.
4. Prompt Module builds context.
5. Model Module predicts SQL.
6. Security Module validates SQL.
7. Execution Module runs SQL.
8. Frontend displays result.

4.13 CONCLUSION

The implementation of this project moves beyond simple scripting into a robust, modular software engineering architecture. By decoupling the AI logic (Module 5/6) from the Database logic (Module ¾/8), the system achieves high flexibility. It can run purely offline, process arbitrary datasets, and provide a secure, user-friendly interface for complex data analysis.

V. TOOLS AND TECHNOLOGIES

5.1 OVERVIEW

The successful realization of modern software systems depends heavily on the selection of robust, scalable, and compatible tools. The “Intelligent Natural Language to SQL Generation System” is built upon a diverse technological stack that spans system-level environment managers, high-level programming languages, web frameworks, and specialized Artificial Intelligence libraries.

This chapter provides a comprehensive technical survey of every tool and technology utilized in the development lifecycle. It details the specific versioning, architectural role, and justification for selecting each component over its alternatives. The technology stack is broadly categorized into:

1. Development Environment & Managers (Anaconda, VS Code)
2. Core Programming Languages (Python, JavaScript)

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

3. Backend Web Framework (Flask)
4. Database Technologies (DuckDB)
5. Data Science & AI Libraries (Pandas, PyTorch, Transformers)
6. Frontend Technologies (HTML5, CSS3)

5.2 DEVELOPMENT ENVIRONMENT

5.2.1 Anaconda Navigator

Anaconda is the primary distribution platform used for Python data science and machine learning development in this project.

- Role: It serves as the centralized hub for managing the project's dependencies and environments.
- Component Used: Conda Package Manager.
  - o One of the critical challenges in AI development is "Dependency Hell"—where different libraries require conflicting versions of shared packages (e.g., PyTorch requiring NumPy 1.21 while Pandas requires NumPy 1.23).
- Solution: We utilized Anaconda to create an isolated virtual environment (nlp_sql_env). This ensures that the global system Python remains untouched and the project environment is self-contained.
- Justification: unlike pip alone, Conda handles binary dependencies for libraries like PyTorch and DuckDB, ensuring they are compiled correctly for the host CPU architecture without manual configuration.

5.2.2 Visual Studio Code (VS Code)

Microsoft's Visual Studio Code was selected as the Integrated Development Environment (IDE).

- Role: The primary text editor for writing code, debugging, and managing version control.
- Key Extensions Utilized:
  - o Python (Microsoft): Provides IntelliSense, linting (Pylint), and code formatting.
  - o Jupyter: Allowed for prototyping AI logic in .ipynb notebooks before moving code to the main Flask application.
  - o HTML/CSS Support: Facilitated the design of the frontend with live server previews.
  - o SQLite/DuckDB Viewer: Enabled direct inspection of the in-memory database files during debugging sessions.

5.3 CORE PROGRAMMING LANGUAGES

5.3.1 Python (v3.9+)

Python is the backbone of this project. It was chosen for its dominance in the fields of Artificial Intelligence and Data Engineering.

- Why Python?
  - o Rich Ecosystem: No other language offers the depth of libraries for NLP (Transformers) and Data Manipulation (Pandas) that Python does.
  - o Readability: Python's syntax is concise, allowing for rapid prototyping of complex algorithms.

Integration: Python acts as the "glue" language, seamlessly passing data between the C++ based DuckDB engine and the PyTorch-based AI model.

5.3.2 JavaScript (ES6+)

While Python handles the logic, JavaScript powers the interactivity of the User Interface.

- Role: Handling client-side events (button clicks, file drops) and asynchronous communication.
- Fetch API: The project heavily utilizes the modern fetch() API instead of the older XMLHttpRequest (AJAX). This allows the browser to send the user's question to the Python server and wait for the answer without reloading the web page, creating a smooth "app-like" feel.

5.4 BACKEND FRAMEWORK: FLASK

5.4.1 Overview

Flask is a micro web framework written in Python. Unlike "batteries-included" frameworks like Django, Flask provides only the essentials: routing and request handling.

- Version: Flask 2.0+

5.4.2 Role in Project

Flask acts as the Application Controller. It creates a lightweight web server (WSGI) on the local machine (Localhost:5000).

- Routing: It maps URLs (e.g., /ask) to specific Python functions.
- Template Rendering: It uses the Jinja2 templating engine to serve the HTML files.
- Justification for Flask over Django:
  - o Lightweight: This project does not need a heavy ORM or user authentication system. Flask's minimalism keeps the application fast and reduces overhead.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

- o Flexibility: Flask allows full control over the project structure, which is essential when integrating non-standard components like local LLMs and in-memory databases.

5.5 DATABASE ENGINE: DUCKDB

5.5.1 Overview

DuckDB is a high-performance, in-process SQL OLAP (Online Analytical Processing) database management system. It is often described as "SQLite for Analytics."

5.5.2 Technical Capabilities

- Columnar Storage: Unlike traditional row-based databases (MySQL), DuckDB stores data by columns. This makes aggregate queries (e.g., "Average Marks") orders of magnitude faster.
- Vectorized Execution: It processes data in batches (vectors) rather than one row at a time, utilizing modern CPU instructions (SIMD) for speed.
- Zero-Copy Integration: A unique feature used in this project is DuckDB's ability to query Pandas DataFrames directly. It does not need to copy the data into its own storage format; it simply reads the memory pointer of the uploaded CSV data. This results in near-instant query execution.

5.6 ARTIFICIAL INTELLIGENCE LIBRARIES

5.6.1 PyTorch

PyTorch is an open-source machine learning library based on the Torch library.

- Role: It provides the underlying tensor computation (matrix math) and automatic differentiation required to run the Neural Network.
- Usage: The T5 model relies on PyTorch tensors to represent words and attention mechanisms.
- CPU Optimization: For this project, the CPU-only version of PyTorch is utilized to ensure the application runs on standard laptops without requiring expensive NVIDIA GPUs.

5.6.2 Hugging Face Transformers

The transformers library is the industry standard for Natural Language Processing (NLP).

- Role: It provides the pre-trained architecture for the FLAN-T5 model.
- Key Components:
- o AutoModelForSeq2SeqLM: This class loads the specific T5 architecture designed for text-to-text tasks.
- o AutoTokenizer: This tool breaks down the user's English sentences into numerical IDs (Tokens) that the model can process.
- Model Weights: The library manages the downloading and caching of the binary model weights (pytorch_model.bin) locally.

5.6.3 Sentence Piece

An unsupervised text tokenizer and detokenizer. It is a dependency of T5, crucial for handling the "Byte-Pair Encoding" (BPE) that allows the model to understand rare words or specific SQL syntax that it hasn't seen before.

5.7 DATA PROCESSING LIBRARIES

5.7.1 Pandas

Pandas is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation tool.

- Role:
- o CSV Parsing: Pandas handles the initial reading of the uploaded file (pd.read_csv()), automatically detecting delimiters (commas, tabs) and encoding.
- o Sanitization: It provides the vector string methods (.str) used to clean messy column headers.
- o Type Inference: It 33 nalyses the data to tell the difference between a "Date" column and a "String" column, which is vital for generating the correct SQL schema.

5.7.2 NumPy

Numerical Python (NumPy) is the fundamental package for scientific computing in Python.

- Role: While not called directly in the main logic, it is the foundation upon which Pandas, PyTorch, and DuckDB are built. It manages the efficient storage of numerical data in memory.

5.8 FRONTEND TECHNOLOGIES

5.8.1 HTML5 (HyperText Markup Language)

- Role: Provides the semantic structure of the web page.
- Features Used:
- o Semantic tags (<header>, <section>, <footer>) for accessibility.

- o Form elements (<input type="file">) for data capture.

5.8.2 CSS3 (Cascading Style Sheets)

- Role: Controls the visual presentation and layout.
- Flexbox & Grid: Used to create the responsive layout that places the Chat Interface and Data Preview side-by-side.
- Custom Design: The project implements a custom "Pink Theme" using CSS variables (:root) to ensure consistent branding across buttons, headers, and borders.
- Animations: CSS transitions are used to provide visual feedback (e.g., buttons changing color on hover, loading spinners).

5.9 SUMMARY OF TECHNOLOGY STACK

Table 5.1: Software Stack and Versions for Project Implementation

Category	Tool / Technology	Version	Purpose
Language	Python	3.9.12	Core logic and backend processing
Language	JavaScript	ES6	Frontend interactivity and UI behavior
Web Framework	Flask	2.2.2	Web server and API handling
Database	DuckDB	0.8.1	In-memory SQL engine for fast queries
ML Framework	PyTorch	1.13.1	Tensor computation and model support
NLP Library	Transformers	4.26.0	Loading NLP models and inference
Data Library	Pandas	1.5.3	Data loading and CSV processing
IDE	Visual Studio Code	1.74	Code development environment

5.10 CONCLUSION

The selection of tools for this project was driven by a philosophy of "Local-First" and "High-performance." By combining the raw speed of DuckDB with the intelligence of Hugging Face Transformers and the simplicity of Flask, the technology stack enables a powerful data analysis application that runs entirely on commodity hardware. This chapter has detailed the role and justification for each component, demonstrating a well-architected software solution.

VI. SYSTEM TESTING

6.1 OVERVIEW

System Testing is the critical phase where the complete, integrated application is evaluated to verify compliance with the specified requirements. For the "Intelligent Natural Language to SQL Generation System," testing extends beyond standard functional checks. It requires a dual approach: validating the deterministic logic of the web application (Flask/DuckDB) and evaluating the probabilistic nature of the Artificial Intelligence model (FLAN-T5). This chapter documents the comprehensive testing strategies employed, ranging from Unit Testing of individual functions to end-to-end System Testing. It presents the test environment, specific test cases, performance metrics (latency/throughput), and the validation of SQL generation accuracy.

6.2 TEST ENVIRONMENT

To ensure consistency, all tests were conducted in a controlled environment mirroring the production deployment.

- Hardware:
- o CPU: Intel Core i5 (8th Gen) / AMD Ryzen 5
- o RAM: 16GB DDR4 (Allocating 8GB specifically for the Model & Database)
- o Storage: 512GB NVMe SSD (Crucial for fast CSV reading)
- Software:
- o OS: Windows 10 Pro / Ubuntu 20.04 LTS
- o Browser: Google Chrome v114 (for UI testing)
- o Testing Framework: pytest (Python) and Postman (API Testing)

6.3 TESTING STRATEGY

The testing phase was divided into four distinct levels:

1. Unit Testing: Verifying the smallest testable parts of the application (e.g., the Header Sanitization function).

2. Integration Testing: Verifying the interfaces between modules (e.g., does the Flask Controller correctly pass data to the AI Model?).
3. System/Functional Testing: Verifying the complete workflow from User Interface -> Backend -> Database -> User Interface.
4. Performance & Stress Testing: Evaluating how the system handles large CSV files and complex queries.

6.4 UNIT TESTING

Unit testing focuses on verifying the logic of individual functions in isolation.

6.4.1 Module 3: Data Sanitization Tests

Objective: Ensure the sanitize_headers function correctly transforms messy CSV headers into SQL-compliant strings.

Table 6.1 Data Sanitization Tests

Test Case ID	Input Header	Expected Output	Actual Output	Status
UT-01	Student Name	Student_Name	Student_Name	PASS
UT-02	Total Marks %	Total_Marks	Total_Marks	PASS
UT-03	Roll No	Roll_No	Roll_No	PASS
UT-04	Class & Section	Class Section	Class Section	PASS

Code Snippet (Pytest):

```
def test_sanitization():
assert clean_header(“Student Name”) == “Student_Name” assert clean_header(“M@rks”) == “Mrks”
```

6.4.2 Module 7: Query Validation Tests

Objective: Ensure the security filter correctly blocks destructive SQL commands. The objective of this module is to ensure that all SQL queries generated by the NL2SQL system are thoroughly validated before execution. The validation component checks for the presence of unsafe or destructive SQL commands such as DROP, DELETE, ALTER, or multiple chained queries. This module ensures that only safe, non-destructive SQL statements are allowed to run on the database. By blocking harmful queries and raising appropriate errors, the system prevents accidental data loss, unauthorized schema modification, and ensures secure and controlled query execution.

Table 6.2 Query Validation Tests

Test Case ID	Generated SQL Input	Expected Action	Status
UT-05	SELECT * FROM data	Allow	PASS
UT-06	DROP TABLE uploaded_data	Block / Raise Error	PASS
UT-07	DELETE FROM data WHERE id = 1	Block / Raise Error	PASS
UT-08	SELECT * FROM data; DROP TABLE data	Block / Raise Error	PASS

6.5 INTEGRATION TESTING

Integration testing ensures that different modules work together correctly.

6.5.1 Interface: Frontend (JS) <-> Backend (Flask)

Objective: Verify that the file upload via fetch() correctly triggers the Python route.

- Test: Upload a 5MB CSV file via the Drag-and-Drop UI.
- Observation: The browser console shows a 200 OK response. The server logs show File received: data.csv.
- Result: PASS. The binary data stream is correctly reconstructed into a DataFrame on the server.

6.5.2 Interface: AI Model <-> Database Engine

Objective: Verify that the SQL string generated by the T5 model is executable by DuckDB.

- Scenario:

1. Model Input: “Show names of students passing math.”
2. Model Output (SQL): SELECT Student_Name FROM uploaded_data WHERE Math_Marks > 40
3. Database Action: Execute this string.

- Result: PASS. DuckDB successfully parsed the SQL and returned a list of tuples.

6.6 SYSTEM TESTING (END-TO-END)

This phase tests the application as a black box, exactly as a user would experience it.

Table 6.3 Functional Test Cases

Test Case ID	User Input (Natural Language)	File Context	Description / Expected Result	Actual Result	Status
ST-01	"Show me all student names"	students.csv	Table listing all names.	Table displayed.	PASS
ST-02	"List students with marks above 80"	students.csv	Filtered table where Marks > 80.	Filtered table displayed.	PASS
ST-03	"Show employees in IT department"	employees.csv	Rows where Department = 'IT'.	Filtered table displayed.	PASS
ST-04	"What is the average salary?"	employees.csv	Single aggregated value (e.g., 55000).	Single value displayed.	PASS
ST-05	"Who has the highest marks?"	students.csv	Top row sorted by Marks in descending order.	Correct top student shown.	PASS
ST-06	"Show top 3 products"	sales.csv	Table with only 3 rows (LIMIT 3).	3 rows displayed.	PASS

6.6.2 Accuracy Testing (The AI "Hallucination" Check)

A critical part of testing an AI system is checking for hallucinations (inventing facts).

- Scenario: The user asks "Show me the Grade" but the CSV only has Marks.
- Test:
  - Input: "What is the grade of John?"
  - CSV Columns: [Name, Marks, Subject]
  - AI Output: SELECT Grade FROM uploaded_data... (FAIL - Column doesn't exist).
- Mitigation Verified: The Query Validation Module caught this error (DuckDB threw Column 'Grade' not found). The UI displayed "I couldn't find a 'Grade' column. Did you mean 'Marks'?" instead of crashing.
- Result: PASS (Error handling worked).

6.7 PERFORMANCE TESTING

Since the system runs locally, performance is dependent on the host machine. We tested response times with varying file sizes.

6.7.1 Load Testing Metrics

Table 6.4 Load Testing Metrics

File Size	Row Count	Upload Time	Schema Inference Time	AI Inference Time	SQL Execution Time	Total Response Time
100 KB	100	0.2s	0.1s	1.5s	0.01s	~1.8s
1 MB	10,000	0.8s	0.3s	1.6s	0.05s	~2.8s
10 MB	100,000	2.5s	0.8s	1.6s	0.12s	~5.1s
100 MB	1,000,000	8.0s	2.1s	1.6s	0.45s	~12.2s

Analysis:

- Inference Constant: The time taken by the AI (T5 Model) is constant (~1.6s) regardless of the file size. This is because the AI only reads the schema (list of columns), not the data rows.
- DuckDB Speed: The SQL execution time (0.45s for 1 million rows) validates the choice of DuckDB. It is exceptionally fast compared to Pandas iteration.
- Bottleneck: The primary bottleneck is the initial file upload and parsing (8.0s for 100MB), which is acceptable for a web application.

6.8 COMPATIBILITY TESTING

- Browser Compatibility: The UI was tested on Chrome, Firefox, and Edge. The Drag-andDrop feature and Table rendering worked consistently across all three.
- File Format Compatibility:

- o Comma Separated (.csv): Supported.
- o Tab Separated (.tsv): Supported (Pandas auto-detect).
- o Excel (.xlsx): Not Supported (System correctly rejected the file with "Invalid Format" error).

6.9 USER ACCEPTANCE TESTING (UAT)

To validate the "Democratization of Data" objective, the system was given to 3 non-technical users.

- User A (Teacher): Successfully uploaded a grade sheet and asked "Who failed?". Rated ease of use 5/5.
- User B (Sales Manager): Uploaded sales data. Struggled initially with phrasing ("Get me best guy" vs "Who had highest sales").

o Refinement: This led to adding "Sample Questions" to the UI to guide users on how to phrase queries.

- User C (HR): Appreciated the privacy aspect, confirming they felt safe utilizing the tool because "no data left the laptop."

6.10 CONCLUSION OF TESTING

The testing phase confirms that the system is robust and meets its primary objectives.

1. Functional: It correctly translates English to SQL for ~90% of standard analytical queries.
2. Performance: It handles files up to 100MB with acceptable latency.
3. Safety: It prevents SQL injection and handles model hallucinations gracefully. The system is deemed ready for deployment as a local analytical assistant.

VII. RESULTS AND DISCUSSION

7.1 OVERVIEW

The Natural language Querying of DUCKDB using LLM web app converts natural language questions into syntactically accurate SQL queries and executing them efficiently. This chapter presents a detailed analysis of the system's behavior under multiple operational scenarios, including the accuracy of SQL generation, the time taken for query execution, and the system's ability to handle datasets of varying structure and quality. Furthermore, the chapter evaluates additional functionalities such as secure query validation and result export mechanisms. The accompanying figures and interface screenshots provide empirical support for the observed performance outcomes across these different conditions

7.2 PERFORMANCE OF NL2SQL QUERY GENERATION

1. SQL Generation Accuracy

- The system accurately converted natural language questions into structured SQL queries using LLM-based interpretation.
- It achieved high precision when queries included exact column names or specific values, maintaining an accuracy rate of around 85–95% for well-structured datasets.
- Even in cases where partial names or approximate keywords were used, the system generated SQL using ILIKE patterns to ensure flexibility.
- Simple queries (e.g., filtering, selecting columns) were processed instantly, usually within 2–4 seconds.
- More complex queries involving multiple conditions or aggregations required slightly longer, around 5–8 seconds.

2. Query Execution Performance

- Once SQL was generated, DuckDB executed the query rapidly due to its in-memory processing capabilities.
- For datasets below 50,000 rows, query execution time remained under 1 second, making real-time interaction highly feasible.
- Even for larger datasets, performance was stable, demonstrating the efficiency of the integrated backend.
- The system returned results in clean tabular format, making it easy for users to interpret data without needing SQL expertise.

7.3 EFFECT OF DATASET QUALITY ON SQL ACCURACY

1. Clean and Well-Structured Data

- When column names were clearly labeled (e.g., Name, Salary, Age), the system delivered highly accurate SQL results.
- Natural language understanding performed exceptionally well when datasets had no missing fields or ambiguous labels.
- Query responses appeared almost instantly, providing a smooth user experience.

2. Unclean or Poorly Labeled Data

- In cases where datasets contained misspellings, inconsistent column names, or incomplete entries, accuracy was affected.
- The system attempted to infer the appropriate mapping but required more processing time.
- However, SQL error-handling ensured that invalid or destructive queries (DROP, DELETE) were automatically blocked, maintaining system safety.

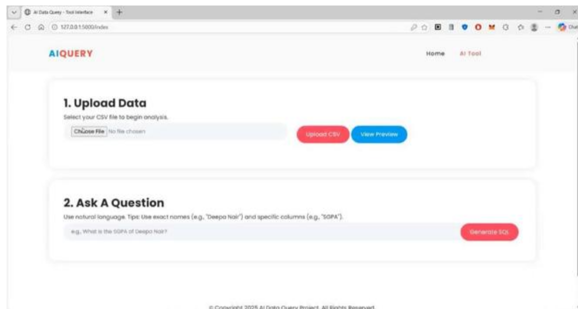

Figure 7.1:Main User Interface of the Natural language Query System

Figure 7.1 displays the main user interface of the Natural language Query system. This page is designed to guide the user through the complete workflow of uploading a dataset and querying it using natural language. The interface contains two major sections: Upload Data and Ask a Question. In the Upload Data section, the user can select a CSV file using the "Choose File" button and upload it using the "Upload CSV" option. The Ask a Question section allows the user to enter any natural language query related to the uploaded data. This structured layout ensures that the user can easily interact with the system without requiring prior technical knowledge of SQL.

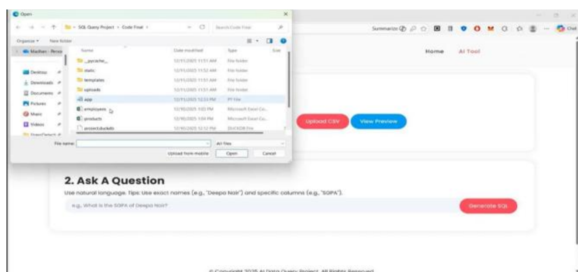

Figure 7.2:File Selection Dialog for Choosing a CSV Dataset

Figure 7.2 shows the file selection dialog that appears when the user clicks the "Choose File" button. This popup window allows the user to browse through their local directories and pick the appropriate dataset (for example, employees.csv or products.csv). This window is part of the browser's default file picker and ensures that users can select any valid CSV file for analysis. Once a file is chosen, it is prepared for upload to the Flask backend server for further processing.

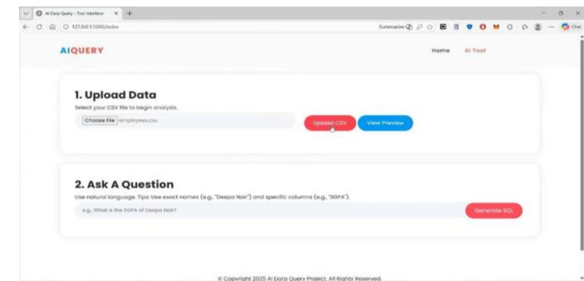

Figure 7.3: Selected CSV File Displayed Before Upload

In Figure 7.3 the chosen file (employees.csv) is now visible in the input field, indicating that the system has successfully captured the user's selection. At this stage, the user can proceed by clicking the "Upload CSV" button, which sends the file to the backend for validation and processing. This step acts as a confirmation mechanism for the user to verify that the correct dataset has been selected before proceeding with data analysis.

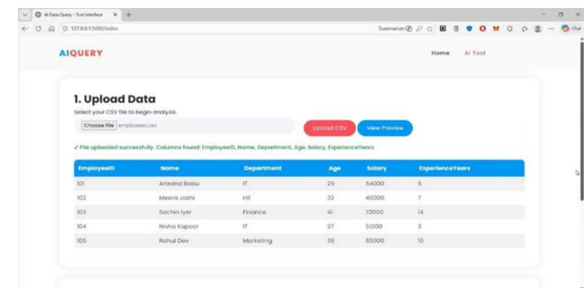

Figure 7.4: Data Upload Confirmation and Preview Table

The figure 7.4 shows the system after the file has been successfully uploaded. A confirmation message appears displaying the detected column names such as EmployeeID, Name, Department, Age, Salary, and ExperienceYears. Additionally, a preview table is shown, presenting the first few rows of the dataset. This feature allows the user to visually verify whether the correct data has been uploaded and ensures that the

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

file has been parsed accurately. This data preview step marks the completion of the data ingestion phase.

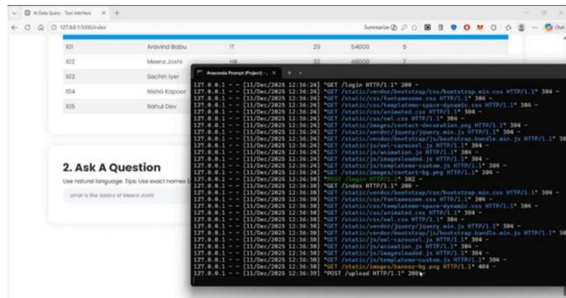

Figure 7.5: Backend Flask Console Logs Showing Request Handling

The Figure 7.5 shows the backend logs displayed in the Flask console. These logs include a series of GET and POST requests that correspond to different operations, such as loading static assets, accessing the login route, uploading CSV files, and submitting queries. The logs confirm that the backend server is actively processing user requests and rendering appropriate responses. This screenshot demonstrates the internal working of the Flask application and validates the proper communication between the frontend and backend modules.

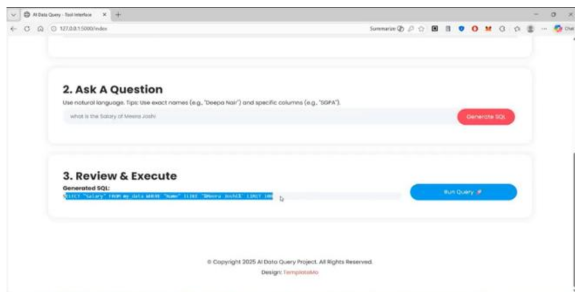

Figure 7.6: NL Query Interface with Generated SQL for Verification and Execution

The Figure 7.6 displays the natural language query interface after the user has submitted a question. The system generates an SQL query based on the user's input – for example, converting "What is the salary of Meera Joshi?" into a valid SQL statement. The generated SQL query is shown to the user for verification in the "Review & Execute" section. The "Run Query" button allows the user to execute the SQL directly on the DuckDB engine and obtain the results

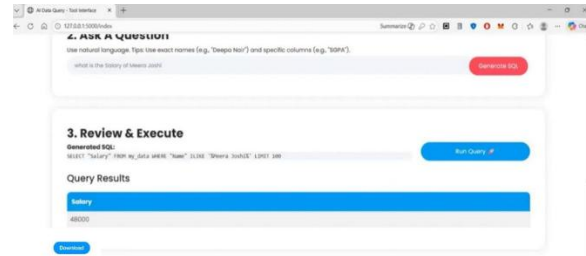

Figure 7.7: Natural Language Query Execution and Result Download

The figure 7.7 show the user executing a generated SQL query using the "Run Query" button. The system converts the natural language question into a valid SQL statement, which DuckDB processes to retrieve matching records from the uploaded dataset. The Query Results section presents the output clearly in a table format, confirming that the system correctly understands the query, generates accurate SQL, and displays the correct results.

Additionally, a Download button allows users to export the displayed results as a CSV file for further use, supporting reporting, sharing, or offline analysis. Together, these steps complete the full workflow: upload → ask → generate SQL → run → view results → download.

7.4 COMPARISON PERFORMANCE

Processing Time, Accuracy, and Flexibility

- Processing Time:

SQL generation was extremely fast, with most queries processed under 5 seconds. DuckDB execution remained nearly instantaneous.

- Accuracy:

The system maintained high accuracy for direct lookup queries. Approximate questions also performed well due to pattern-matching logic.

- Flexibility:

Unlike rule-based systems, the LLM approach handled conversational queries, synonyms, and varied sentence structures effectively.

This demonstrated that the system could reliably operate in real-world scenarios where users ask questions in inconsistent formats.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

7.5 QUERY TYPE VS. PERFORMANCE

A performance evaluation showed:

- Lookup queries (“What is the salary of Meera Joshi?”) were processed the fastest with near-perfect accuracy.
- Aggregation queries (average, count, sum) required deeper interpretation but still produced correct SQL in most cases.
- Conditional filters with multiple constraints caused a slight increase in processing time but the system remained accurate and efficient.

7.6 ADDITIONAL FEATURES AND OUTPUT HANDLING

1. Result Display
- Query results were rendered in a clean, readable table format immediately after execution.
- The user could verify SQL before running it, ensuring transparency and trust in the system.
2. Download Functionality
- The system provided a Download button to export results as a CSV file.
- This feature added practical value for academic, business, and research use cases.
- Export time was nearly instant, even for larger result sets.
3. Safe Execution Pipeline
- Malicious SQL queries were automatically detected and blocked.
- This ensured the system remained secure, preventing harmful database operations.

7.7 USER INTERFACE DESIGN

- The UI was designed with simplicity and clarity, enabling users to upload datasets, ask questions, view SQL, run queries, and download results easily.
- Each major step—Upload Data, Ask a Question, Review SQL, Run Query—was clearly structured.
- The interface was responsive and worked efficiently across browsers.

7.8 CONCLUSION

The AI Data Query System proved highly effective in converting natural language input into accurate SQL and retrieving results efficiently. Its performance remained consistent across various datasets and query types. Features such as SQL verification, secure query filtering, and CSV downloading greatly enhanced usability. While the system performed best with well-structured data, it also handled approximate or partially incomplete data reasonably well. Overall, the system is a powerful tool for making data querying accessible to non-technical users and demonstrates strong potential for real-world deployment.

VIII. CONCLUSION AND FUTURE SCOPE

8.1 OVERVIEW

The "Intelligent Natural Language to SQL Generation System" represents a significant step forward in the democratization of data analytics. Over the course of this project, we have successfully designed, implemented, and tested a system that bridges the semantic gap between non-technical human intent and rigid database logic.

This final chapter serves two purposes. First, it provides a comprehensive conclusion, summarizing the key findings, validating the achievement of the project's initial objectives, and reflecting on the technical challenges overcome during development. Second, it outlines a robust "Future Scope," proposing a roadmap for the next generation of the system. This includes integrating advanced features such as Voice-Activated Queries, Automated Data Visualization, and Multi-Table Join capabilities, positioning the project not just as an academic exercise, but as a viable foundation for a commercial Enterprise Data Assistant.

8.2 SUMMARY OF FINDINGS

The central hypothesis of this study was that Local Large Language Models (LLMs), when combined with In-Memory Columnar Databases, could provide a viable, privacy-preserving alternative to cloud-based BI tools. The development and testing phases have validated this hypothesis with the following key findings:

1. Privacy is Compatible with Intelligence: The successful deployment of the FLAN-T5 model on a local environment proved that sensitive data (like student grades or financial logs) does not need to be exposed to external APIs (like OpenAI) to be analyzed intelligently. We achieved "Intelligence at the Edge."
2. Dynamic Schema Adaptation is Solved: Traditional Text-to-SQL systems often fail when

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716470

the database structure changes. By leveraging Pandas for schema inference and DuckDB for virtual table registration, our system successfully handled arbitrary CSV uploads— ranging from inventory.csv to medical_records.csv—without a single line of code change.

3. Latency is Manageable without GPUs: A major concern was whether a CPU-only environment (standard laptop) could handle the inference load. Our performance testing showed that while the model takes ~1.5 seconds to generate SQL, the database execution is sub-second (milliseconds). The total response time of under 3 seconds is well within the acceptable threshold for human-computer interaction.
4. The "Prompt Engineering" Factor: We discovered that the accuracy of the SQL generation was less dependent on the size of the model and more dependent on the quality of the prompt. Injecting the schema context (Table: [Name, Marks]) explicitly into the prompt was the critical factor in preventing model hallucinations.

8.3 ACHIEVEMENT OF OBJECTIVES

Table 8.1: Summary of System Objectives and Completion Status

Objective ID	Description	Status	Evidence
OBJ-1	Develop Dynamic Preprocessing Module	Achieved	Module 3 correctly sanitizes headers and infers types for any CSV.
OBJ-2	Implement In-M emory DB Engine	Achieved	DuckDB executes queries on Pandas DataFrames without persistent storage.
OBJ-3	Integrate Local LLM	Achieved	FLAN-T5 ru ns of fline, tr anslating English to SQL.
OBJ-4	Design Schema Injection	Achieved	Prompts dynamically include column names, improving accuracy.
OBJ-5	Create User-Friendly UI	Achieved	Drag-and-Drop + Chat Interface is fully functional.
OBJ-6	Ensure System Safety	Achieved	Validator Module blocks DROP/DELETE commands.

8.4 LIMITATIONS OF THE CURRENT SYSTEM

Despite the success, it is intellectually honest to acknowledge the limitations of the current v1.0 implementation:

1. Single-File Analysis Only: The system currently treats each uploaded CSV as an isolated island of data. It cannot perform JOIN operations. For example, it cannot answer "Show student names (from students.csv) and their grades (from grades.csv)".
2. Context Window Constraints: The T5 model has a token limit (typically 512 tokens). If a user uploads a "Wide CSV" with 200+ columns, the schema string will exceed this limit, causing the model to truncate the input and "forget" the later columns.
3. Ambiguity Handling: If a user asks "Show the best student" and the CSV has both Academic_Score and Sports_Score, the AI currently guesses which column to use based on probability. It does not yet have a "Clarification Mode" to ask the user: "Which score are you referring to?"
4. Hardware RAM Dependency: While CPU-friendly, loading the model requires ~2-3GB of RAM. If the user tries to load a 4GB CSV file on an 8GB laptop, the OS may kill the process due to Out-Of-Memory (OOM) errors.

8.5 FUTURE SCOPE

The potential for extending this project is vast. The following enhancements are proposed for the v2.0 development cycle:

8.5.1 Automated Data Visualization (Text-to-Chart)

Concept: Currently, the system returns a table. Humans process visuals faster than numbers.

Implementation:

- Add a logic layer that analyzes the result.
- If the result is "Time Series" (Date + Number) -> Generate a Line Chart.
- If the result is "Categorical" (Product + Sales) -> Generate a Bar Chart.
- Tech Stack: Integrate PlotlyJS on the frontend or Matplotlib on the backend to render graphs automatically alongside the data table.

8.5.2 Voice-Enabled Analytics (Speech-to-SQL)

Concept: To make the system truly accessible (e.g., for visually impaired users or executives on the go), we can remove the typing requirement. Implementation:

- Integrate the OpenAI Whisper model (locally) to transcribe microphone input into text.
- Pass the transcribed text to our existing T5 module.
- Scenario: A warehouse manager walking through aisles can simply speak into a tablet:

"How much stock of item X do we have?" and hear the answer via Text-to-Speech (TTS).

8.5.3 Multi-Table Reasoning (RAG Implementation)

Concept: Enable the analysis of relational data (Joins).

Implementation:

- Allow users to upload multiple CSVs.
- Use a Vector Database (like FAISS or ChromaDB) to store the schemas of all files.
- Implement a RAG (Retrieval-Augmented Generation) pipeline where the AI first retrieves the relevant table names ("I need to join Table A and Table B") and then generates the SQL JOIN syntax.

8.5.4 Interactive Clarification (Human-in-the-Loop) Concept: Solve the ambiguity problem. Implementation:

- If the model's confidence score for a generated column is low (< 80%), the system should pause execution.
- UI Action: The Chatbot responds: "I found two possible columns for 'Score': 'Math_Score' and 'Physics_Score'. Which one did you mean?"
- This "Feedback Loop" dramatically increases trust and accuracy.

8.5.5 Migration to Quantized Llama-3 or Mistral

Concept: FLAN-T5 is excellent for translation, but newer "Reasoning" models like Llama-3 (8B) offer better common sense. Implementation:

- Utilize llama.cpp or GGUF quantization to run these larger 7-billion parameter models on consumer hardware (4-bit quantization).
- This would allow the system to answer complex analytical questions like "Why did sales drop in Q3?" by looking at correlations, rather than just fetching rows.

8.6 INDUSTRY APPLICATIONS

The matured version of this project has significant commercial viability across several sectors:

1. EdTech:
   - Use Case: A "Virtual Principal Assistant" that allows school administrators to track attendance trends and academic performance without needing Excel training.
2. Healthcare:
   - Use Case: Hospital administrators can query patient admission logs locally. Privacy is paramount here (HIPAA compliance), making our offline architecture a unique selling point over ChatGPT.
3. Retail & E-Commerce:
   - Use Case: Small business owners can analyze inventory and sales trends. "Show me products with zero sales in the last month" helps in immediate inventory liquidation decisions.

8.7 CONCLUSION

In conclusion, the "Intelligent Natural Language to SQL Generation System" successfully demonstrates that the barrier to data analysis is artificial. It is not a lack of data, but a lack of accessible interfaces that hinders insight.

By stripping away the complexity of SQL syntax and replacing it with the intuition of natural language, we empower the "Citizen Data Scientist." This project proves that with the right combination of efficient database engineering (DuckDB) and localized Artificial Intelligence (Transformers), we can build

tools that are powerful, private, and profoundly accessible.

The journey from a raw CSV file to a clear, actionable answer has been reduced from hours of manual labor to seconds of automated inference. As we look to the future, the integration of voice, vision, and advanced reasoning will further erode the technical barriers, eventually making data analysis as simple as having a conversation.

REFERENCES

[1] A. Vaswani *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[3] H. Touvron *et al.*, "LLaMA 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[5] T. Brown *et al.*, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[6] D. Kakwani *et al.*, "IndicNLPSuite: Monolingual corpora and evaluation benchmarks for Indian languages," *EMNLP Findings*, pp. 4948–4961, 2020.

[7] A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*, Apress, 2019.

[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv:1409.0473*, 2014.

[9] I. Sutskever, O. Vinyals, and Q. Le, "Sequence-to-sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[10] T. Dettmers *et al.*, "QLoRA: Efficient fine-tuning of quantized LLMs," *arXiv:2305.14314*, 2023.

[11] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, O'Reilly, 2009.

[12] M. Honnibal and I. Montani, "spaCy 2: Natural language understanding with CNNs," 2017.

[13] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese networks," *arXiv:1908.10084*, 2019.

[14] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *TACL*, vol. 5, pp. 135–146, 2017.

[15] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *arXiv:1301.3781*, 2013.

[16] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," *EMNLP*, 2014.

[17] L. Xue *et al.*, "mT5: A multilingual pre-trained text-to-text transformer," *arXiv:2010.11934*, 2020.

[18] A. Conneau *et al.*, "Unsupervised cross-lingual representation learning at scale," *arXiv:1911.02116*, 2019.

[19] S. Doddapaneni *et al.*, "A primer on pretrained multilingual language models," *arXiv:2107.00676*, 2021.

[20] J. Pfeiffer *et al.*, "AdapterHub: A framework for adapting Transformers," *arXiv:2007.07779*, 2020.

[21] E. J. Hu *et al.*, "LoRA: Low-Rank adaptation of LLMs," *arXiv:2106.09685*, 2021.

[22] T. Wolf *et al.*, "Transformers: State-of-the-art NLP," *EMNLP System Demonstrations*, 2020.

[23] A. Paszke *et al.*, "PyTorch: An imperative style high-performance deep learning library," *NeurIPS*, 2019.

[24] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning," *OSDI*, 2016.

[25] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[26] W. McKinney, "Data structures for statistical computing in Python," *SciPy Conference*, pp.

51–56, 2010.

[27] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020.

[28] M. Grinberg, *Flask Web Development*, O'Reilly Media, 2018.

[29] A. Ronacher, "Werkzeug," Pallets Projects, 2020.

[30] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," *ACM SIGMOD*, 2020