SQL Learning Assistant using RAG and LLM

1. Executive Summary

1.1 Project Overview

The SQL Learning Assistant is an intelligent chatbot designed to transform natural language questions into optimized SQL queries. The system leverages multiple generative AI techniques to deliver accurate, contextually-aware, and well-explained SQL statements for users ranging from beginners to experienced developers.

The application serves as both a learning tool and a productivity assistant, helping users understand SQL concepts while generating production-ready queries. By combining retrieval-based learning with generative AI, the system provides contextually relevant examples alongside newly generated queries.

Target Users:

- SQL beginners learning database querying
- Data analysts needing quick query prototypes
- Developers seeking SQL syntax assistance
- Students studying database management

1.2 Problem Statement

Learning SQL presents significant challenges for beginners:

- **Complex syntax:** SQL has strict grammar rules that vary across database systems (MySQL, PostgreSQL, SQLite, etc.)
- **Translation difficulty:** Converting business questions into query logic requires understanding of tables, relationships, and SQL operations
- **Lack of feedback:** Beginners often don't know why their queries fail or how to improve them
- **Limited examples:** Finding relevant, high-quality examples for specific use cases is time-consuming
- **No explanations:** Most SQL generators provide queries without explaining the logic behind them

Current Solutions and Their Limitations:

Existing Solution	Limitation
Google Search	Generic results, no personalization
ChatGPT/Claude	No domain-specific training, no similar examples
SQL Tutorials	Static content, not interactive
Stack Overflow	Requires manual searching, outdated answers

1.3 Solution Approach

Our solution implements a multi-stage AI pipeline combining four core techniques:

Component	Technology	Role
RAG	ChromaDB + LangChain	Retrieves similar examples from 80,000+ SQL pairs
Fine-Tuned LLM	TinyLlama + LoRA	Generates domain-specific SQL queries
Prompt Engineering	Custom Templates	Manages context, handles edge cases
Synthetic Data	5 Augmentation Methods	Expands training data diversity

LLM Enhancement	Google Gemini API	Refines queries and generates explanations

How It Works (User Perspective):

1. User enters a natural language question (e.g., "Find all employees earning more than $50,000")
2. System retrieves similar examples from the knowledge base
3. Fine-tuned model generates an initial SQL query
4. Gemini API refines the query and adds an explanation
5. User receives optimized SQL with a beginner-friendly explanation

Key Differentiators:

- **Context-Aware:** Uses RAG to provide relevant examples, not just generic responses
- **Domain-Optimized:** Fine-tuned specifically for SQL generation tasks
- **Explainable:** Every query comes with a natural language explanation
- **Reliable:** Fallback systems ensure consistent availability
- **Free to Use:** Deployed on HuggingFace Spaces with no cost to users

1.4 Key Achievements

Metric	Value
Knowledge Base Size	80,000+ SQL question-answer pairs
Core Components Implemented	4 of 5 (RAG, Fine-tuning, Prompts, Synthetic Data)
Augmentation Techniques	5 methods (synonym, insertion, swap, structure, case)
API Fallback Support	3 API keys, 2 model options
Deployment	HuggingFace Spaces + GitHub Pages
Training Data Expansion	3x augmentation factor
Chunking Strategies	4 methods for knowledge base organization

1.5 Project Scope

In Scope:

- Natural language to SQL conversion
- SELECT, INSERT, UPDATE, DELETE query generation
- Query explanation in plain English
- Similar example retrieval
- Web-based interactive interface

Out of Scope:

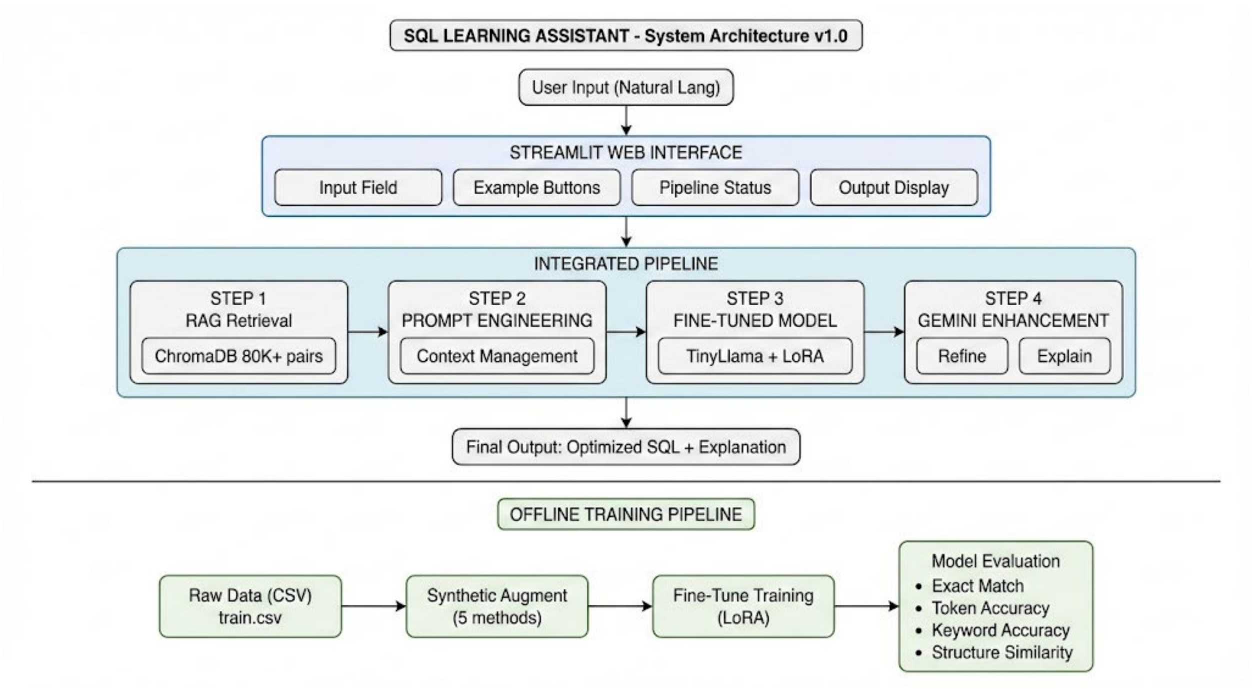
- Direct database connection and query execution
- Real-time query validation against live schemas
- Support for database-specific stored procedures
- Multi-language support (English only)

1.6 Live Demo Links

Resource	URL
Web Application	https://huggingface.co/spaces/moheesh/sql-learning-assistant
Project Page	https://moheesh.github.io/Prompt_to_SQL_using_RAG_LLM
GitHub Repository	https://github.com/moheesh/Prompt_to_SQL_using_RAG_LLM

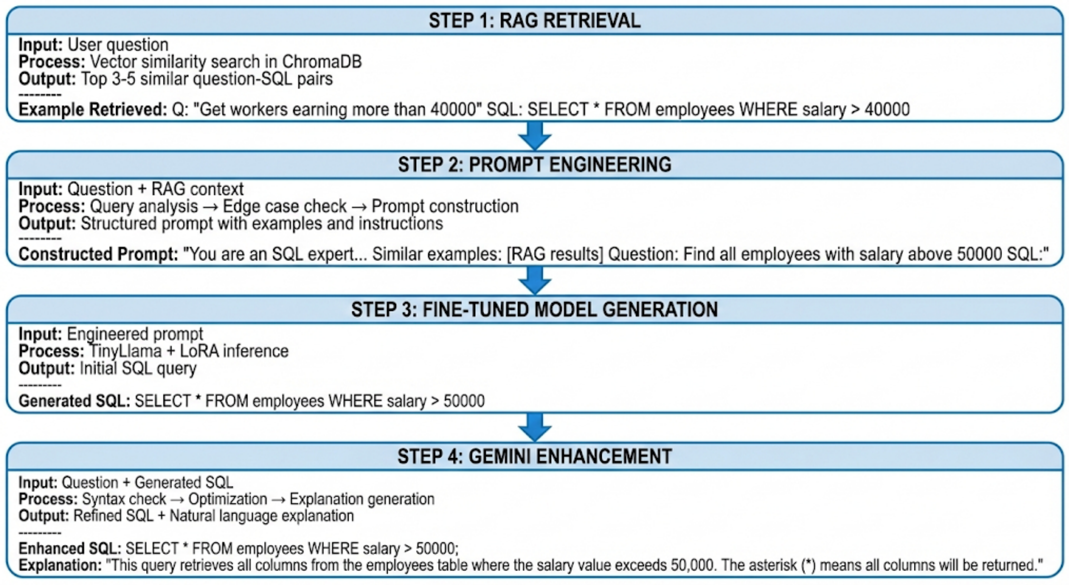
2. System Architecture

2.1 High-Level Architecture Diagram



2.2 Component Interaction Flow

The pipeline processes each user query through four sequential stages:



STEP 1: RAG RETRIEVAL

Input: User question
Process: Vector similarity search in ChromaDB
Output: Top 3-5 similar question-SQL pairs

Example Retrieved: Q: "Get workers earning more than 40000" SQL: SELECT * FROM employees WHERE salary > 40000

STEP 2: PROMPT ENGINEERING

Input: Question + RAG context
Process: Query analysis → Edge case check → Prompt construction
Output: Structured prompt with examples and instructions

Constructed Prompt: "You are an SQL expert... Similar examples: [RAG results] Question: Find all employees with salary above 50000 SQL:"

STEP 3: FINE-TUNED MODEL GENERATION

Input: Engineered prompt
Process: TinyLlama + LoRA inference
Output: Initial SQL query

Generated SQL: SELECT * FROM employees WHERE salary > 50000

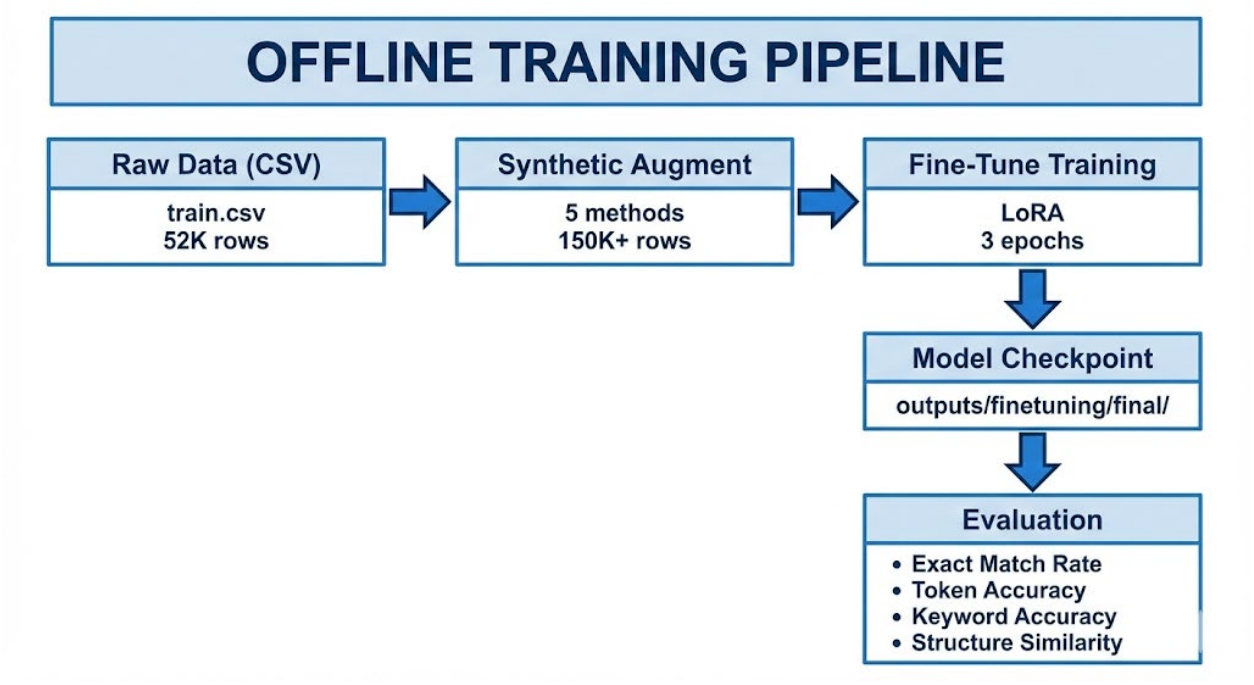
STEP 4: GEMINI ENHANCEMENT

Input: Question + Generated SQL
Process: Syntax check → Optimization → Explanation generation
Output: Refined SQL + Natural language explanation

Enhanced SQL: SELECT * FROM employees WHERE salary > 50000;
Explanation: "This query retrieves all columns from the employees table where the salary value exceeds 50,000. The asterisk (*) means all columns will be returned."

2.3 Offline Training Pipeline

In addition to the inference pipeline, the system includes an offline training pipeline:



2.4 Technology Stack

Layer	Technology	Version	Purpose
Frontend	Streamlit	1.28+	Interactive web interface
Base LLM	TinyLlama	1.1B-Chat-v1.0	Foundation language model
Fine-tuning	LoRA (PEFT)	Latest	Parameter-efficient training
Vector Database	ChromaDB	Latest	Similarity search storage
Embeddings	all-MiniLM-L6-v2	-	Text to vector conversion
Enhancement	Google Gemini API	2.5-flash	Query refinement & explanation
Orchestration	LangChain	Latest	RAG pipeline management
ML Framework	PyTorch	2.0+	Model training & inference
Data Processing	Pandas	Latest	Dataset manipulation
Deployment	HuggingFace Spaces	-	Cloud hosting

2.5 Data Flow Summary

Stage	Input	Processing	Output
User Interface	Natural language question	Validation, formatting	Clean query string
RAG Retrieval	Query string	Embedding → Search → Rank	Top K similar examples
Prompt Building	Query + Examples	Intent analysis, template fill	Structured prompt
Model Inference	Prompt	Token generation	Raw SQL query
Enhancement	Query + SQL	Refinement, explanation	Final SQL + explanation

2.6 Deployment Architecture

Local Development:

```
Local Machine
├── Python 3.10+
├── Model: outputs/finetuning/checkpoints/final/
├── ChromaDB: chromadb_data/
├── Data: data/*.csv
└── Streamlit: localhost:8501
```

Cloud Deployment (HuggingFace Spaces):

```
HuggingFace Infrastructure
├── Spaces: Streamlit application runtime
├── Model Hub: moheesh/sql-tinyllama-lora
├── Dataset Hub: moheesh/sql-chromadb
├── Secrets: GEMINI_API_KEY, HF_TOKEN
└── Auto-download on cold start
```

2.7 Project Requirement Compliance

This section verifies that all INFO7375 Final Project requirements have been satisfied.

Core Components Implementation (Minimum 2 Required)

#	Component	Implementation File
1	Prompt Engineering	`prompts/prompt_builder.py`, `prompts/system_prompts.py`
2	Fine-Tuning	`finetuning/train.py`, `finetuning/evaluate.py`, `finetuning/inference.py`
3	Retrieval-Augmented Generation (RAG)	`rag/retriever.py`, `rag/knowledge_base.py`, `rag/embeddings.py`
4	Synthetic Data Generation	`synthetic/generate_data.py`, `synthetic/synonyms.py`

Components Implemented: 4 of 5

Submission Requirements

Requirement	Deliverable
GitHub Repository	https://github.com/moheesh/Prompt_to_SQL_using_RAG_LLM
Complete source code	All modules in `pipeline/`, `finetuning/`, `rag/`, `prompts/`, `synthetic/`
Documentation	`README.md`
Setup instructions	`README.md` - Installation section
Testing scripts	`test_*()` functions in each module
Example outputs	`outputs/` directory
Fine-tuning datasets	`data/train.csv`, `data/validation.csv`, `data/test.csv`
Knowledge base	`chromadb_data/`

Requirement	Deliverable
PDF Documentation	This document
System architecture diagram	Section 2.1
Implementation details	Section 3
Performance metrics	Section 5

Challenges and solutions	Section 6
Future improvements	Section 8
Ethical considerations	Section 7

Requirement	Deliverable
Video Demonstration	Submitted separately
Live system demo	Included
Key features explanation	Included
Technical architecture overview	Included
Results and performance analysis	Included
Lessons learned	Included

Requirement	Deliverable
Web Page	https://moheesh.github.io/Prompt_to_SQL_using_RAG_LLM
Project showcase	`docs/index.html`
Interactive demo link	Links to HuggingFace Spaces
Key features	Features section
GitHub link	Navigation and hero section
Documentation links	Documentation section

3. Core Components Implementation

3.1 Retrieval-Augmented Generation (RAG)

Purpose: Retrieve relevant SQL examples from a knowledge base to provide context for query generation.

Files: `rag/knowledge_base.py`, `rag/retriever.py`, `rag/embeddings.py`

Knowledge Base Construction

The knowledge base contains 80,000+ question-SQL pairs stored in ChromaDB. Text embeddings are generated using the `all-MiniLM-L6-v2` model from Sentence Transformers, producing 384-dimensional vectors for similarity search.

Four chunking strategies are applied during indexing:

1. **SQL Clause Extraction** (`chunk_by_sql_clauses`) - Identifies clauses such as SELECT, FROM, WHERE, GROUP BY, and ORDER BY to understand query structure.
2. **Complexity Classification** (`chunk_by_complexity`) - Assigns a complexity level (simple, intermediate, complex) based on presence of JOINs, subqueries, aggregations, and nested conditions.
3. **Keyword Extraction** (`extract_sql_keywords`) - Extracts SQL operations including JOIN types, aggregate functions (COUNT, SUM, AVG), and modifiers.
4. **Size Categorization** (`calculate_chunk_size`) - Classifies question and SQL length as short, medium, or long for filtering purposes.

Retrieval Process

The retriever converts the user query into an embedding vector, performs similarity search in ChromaDB, and applies re-ranking based on word overlap between the query and retrieved questions. Results can be filtered by minimum similarity threshold or complexity level before returning the top-K matches.

3.2 Fine-Tuning

Purpose: Train a domain-specific model for SQL generation using parameter-efficient methods.

Files: `finetuning/prepare_data.py`, `finetuning/train.py`, `finetuning/evaluate.py`, `finetuning/inference.py`

Model and Training Setup

The base model is TinyLlama-1.1B-Chat-v1.0, fine-tuned using LoRA (Low-Rank Adaptation) for parameter efficiency. LoRA configuration uses rank 16, alpha 32, dropout 0.1, targeting the attention projection layers (q_proj, v_proj, k_proj, o_proj).

Training runs for 3 epochs with batch size 8, learning rate 2e-4, and FP16 precision. Maximum sequence length is 256 tokens with 100 warmup steps and gradient accumulation of 2.

Data Format

Training examples follow a structured format:

```
### Question:
Find all employees with salary greater than 50000

### SQL:
SELECT * FROM employees WHERE salary > 50000
```

Outputs

The fine-tuned model checkpoint is saved to `outputs/finetuning/checkpoints/final/`. Training statistics and evaluation results are stored as JSON files in the same directory structure.

3.3 Prompt Engineering

Purpose: Construct optimized prompts with context management and edge case handling.

Files: `prompts/prompt_builder.py`, `prompts/system_prompts.py`

System Prompts

Four specialized system prompts are defined based on query type:

- **Simple** - Basic SELECT, FROM, WHERE operations
- **Complex** - JOINs, subqueries, multiple conditions
- **Aggregation** - COUNT, SUM, AVG with GROUP BY and HAVING
- **Modification** - INSERT, UPDATE, DELETE with safety considerations

Context Management

The `ConversationContext` class maintains conversation history (up to 5 turns), current database schema, and user preferences. This context is injected into prompts to enable multi-turn interactions and schema-aware generation.

Query Intent Analysis

The `analyze_query_intent()` function examines user questions to detect:

- Query type (simple, complex, aggregation, modification)
- SQL keywords mentioned
- Question length and complexity indicators

Edge Case Handling

The `detect_edge_cases()` function identifies problematic inputs:

Edge Case	Detection Criteria
Too Short	Question length < 5 characters
Too Vague	Generic words with insufficient context
Multiple Questions	More than one question mark
Contains SQL	SQL keywords detected in input
Dangerous Operation	DROP, TRUNCATE, DELETE ALL patterns
Non-SQL Related	Greetings, off-topic queries

Each edge case triggers an appropriate response or clarification request rather than attempting generation.

3.4 Synthetic Data Generation

Purpose: Expand training data through augmentation while maintaining quality and diversity.

Files: `synthetic/generate_data.py`, `synthetic/synonyms.py`

Augmentation Techniques

Five augmentation methods are applied to original training data:

1. **Synonym Replacement** (40% probability) - Replaces words with synonyms from a curated dictionary covering SQL verbs, comparisons, entities, and domain terms.
2. **Random Insertion** (15% probability) - Inserts contextual words such as "also", "specifically", "exactly" at random positions.
3. **Random Swap** (10% probability) - Swaps adjacent words to introduce variation.
4. **Structure Variation** - Adds prefixes ("Can you", "Please", "Show me") and suffixes ("?", "please") to questions.
5. **Case Variation** - Applies different capitalization patterns.

Quality Controls

Generated data passes through multiple quality filters:

- Minimum question length of 10 characters
- Maximum question length of 500 characters
- Minimum diversity score of 0.1 (Jaccard distance from original)
- Duplicate removal using MD5 hash of normalized text
- Character validation requiring alphabetic content

Privacy Protection

The `anonymize()` function removes sensitive patterns before processing:

- Email addresses replaced with `[EMAIL]`
- Phone numbers replaced with `[PHONE]`
- Social Security Numbers replaced with `[SSN]`

Output

Synthetic data is saved to `data/synthetic.csv` with augmentation statistics and diversity metrics logged to `outputs/synthetic/stats/`.

4. Technical Implementation Details

4.1 Project Structure

The project follows a modular architecture with separate directories for each core component:

- **pipeline/** - Main orchestration logic combining all components
- **finetuning/** - Data preparation, model training, evaluation, and inference
- **rag/** - Embeddings, knowledge base construction, and retrieval
- **prompts/** - Prompt templates and context management
- **synthetic/** - Data augmentation and synonym dictionary
- **data/** - Training, validation, and test datasets
- **outputs/** - Generated checkpoints, results, and logs
- **chromadb_data/** - Vector database storage
- **docs/** - GitHub Pages landing page

Root level files include the Streamlit application (`app.py`), configuration manager (`config.py`), and dependency list (`requirements.txt`).

4.2 Configuration Management

All configuration is centralized in `config.py` with environment variables loaded from a `.env` file.

Required Variables:

- Gemini API keys (primary and fallbacks)
- Gemini model name
- HuggingFace repository IDs for model and ChromaDB
- HuggingFace authentication token

Source Resolution: The system automatically determines resource locations. For the model, it checks local checkpoint first, then HuggingFace Hub, and falls back to the base TinyLlama model. For ChromaDB, it checks local storage first, then downloads from HuggingFace, or builds from raw CSV data if neither exists.

4.3 Integrated Pipeline

The `IntegratedPipeline` class orchestrates all components into a unified execution flow. It accepts a natural language question and returns the final SQL with explanation.

The Gemini client implements automatic failover by cycling through multiple API keys and model options when rate limits are encountered, ensuring consistent availability.

Pipeline output includes the final SQL, explanation, and detailed step-by-step results from RAG retrieval, prompt construction, model generation, and Gemini enhancement.

4.4 Web Application

The Streamlit application provides an interactive chat interface with the following features:

- Eager loading of all components at startup for faster response times
- Pre-defined example buttons for common query types
- Real-time pipeline status during processing
- Expandable sections showing RAG results and intermediate outputs
- Session-based chat history

The sidebar displays component status and pipeline flow visualization.

4.5 Deployment

Local Development: Clone the repository, create a virtual environment, install dependencies, configure environment variables, build the knowledge base, and run with Streamlit.

Cloud Deployment: The application runs on HuggingFace Spaces. The fine-tuned model is hosted on HuggingFace Model Hub, and ChromaDB files are hosted on HuggingFace Dataset Hub. API keys are stored as Spaces secrets. Resources are automatically downloaded on cold start.

5. Performance Metrics & Evaluation

5.1 Evaluation Methodology

Model evaluation is performed using the `finetuning/evaluate.py` module against a held-out test set. Four metrics assess different aspects of SQL generation quality:

- **Exact Match** - Whether the predicted SQL is identical to expected output after normalization
- **Token Accuracy** - Word overlap between prediction and expected SQL as a percentage
- **Keyword Accuracy** - Presence of SQL keywords (SELECT, WHERE, JOIN, GROUP BY, etc.) from expected query
- **Structure Similarity** - Jaccard similarity of clause structure between predicted and expected queries

5.2 Model Evaluation Results

Metric	Score
Samples Evaluated	50
Exact Match Rate	0.00%
Token Accuracy	47.21%
Keyword Accuracy	91.33%
Structure Similarity	91.07%

The 0% exact match rate is expected for a small model like TinyLlama, as exact string matching is extremely strict. However, the high keyword accuracy (91.33%) and structure similarity (91.07%) indicate the model correctly identifies SQL operations and query structure. The token accuracy of 47.21% reflects partial word overlap, accounting for variations in table names, column names, and formatting.

5.3 Knowledge Base Statistics

Metric	Value
Total Documents	80,654
Collection Name	sql_knowledge
Embedding Model	all-MiniLM-L6-v2

Data Sources:

Source	Documents
Training Set	56,355
Validation Set	8,421
Test Set	15,878

Complexity Distribution:

Level	Count
Simple	80,396
Intermediate	258
Complex	0

The dataset is heavily skewed toward simple queries, which is typical for SQL learning datasets focused on foundational concepts.

5.4 Synthetic Data Statistics

Metric	Original	Synthetic
Samples	52,527	142,639
Avg Question Length	11.64 words	14.75 words
Min Length	3 words	3 words
Max Length	44 words	49 words
Unique Words	50,846	60,734

Augmentation Results:

Metric	Value
Augmentation Factor	2.72x
Average Diversity Score	0.283
Minimum Diversity	0.103
Maximum Diversity	0.800

The synthetic data pipeline increased the dataset size by 2.72x while maintaining an average diversity score of 0.283, indicating meaningful variation from original samples.

5.5 Visualizations

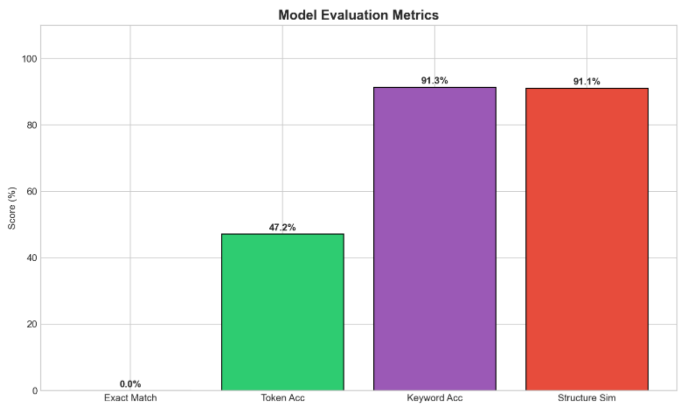
The following visualizations are generated during evaluation and stored in the outputs directory:

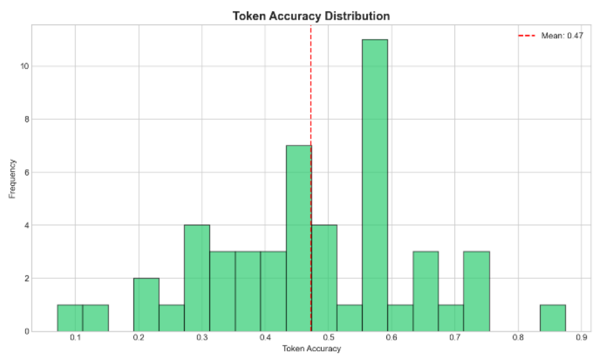
Model Evaluation:

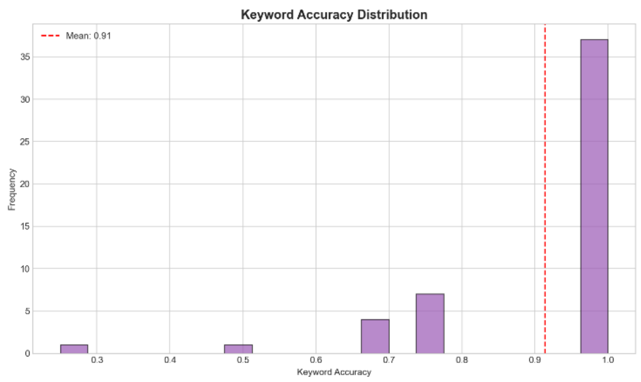
- `01_metrics_overview.png` - Bar chart comparing all four evaluation metrics
- `02_token_accuracy_dist.png` - Histogram showing token accuracy distribution across samples
- `03_keyword_accuracy_dist.png` - Histogram showing keyword accuracy distribution
- `04_training_loss.png` - Training loss over epochs

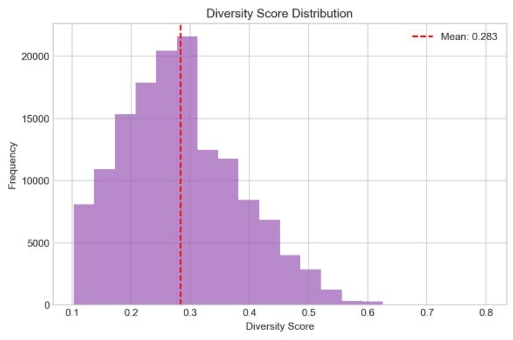
Synthetic Data:

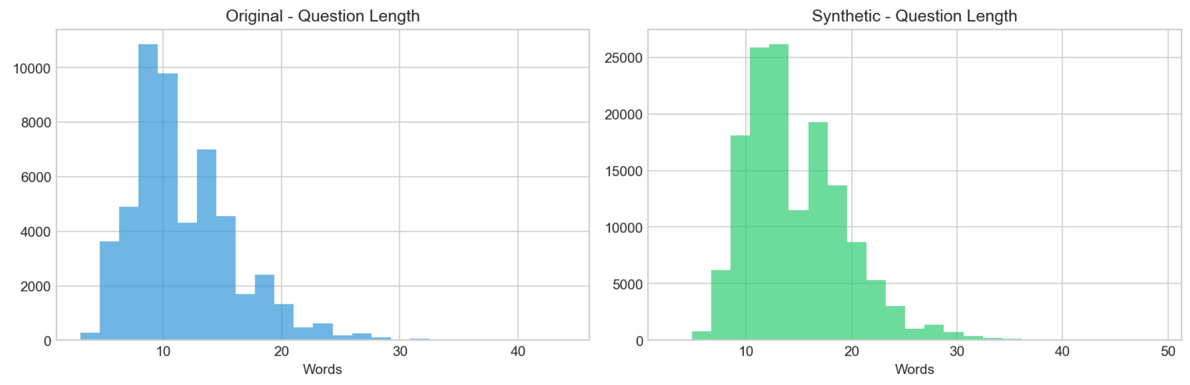
- `01_size_comparison.png` - Comparison of original vs synthetic dataset sizes
- `02_length_distribution.png` - Question length distribution for both datasets
- `03_diversity_distribution.png` - Distribution of diversity scores across synthetic samples











6. Challenges and Solutions

6.1 Technical Challenges

API Rate Limiting - Gemini API rate limits caused failures during high usage. Implemented a multi-key fallback system that automatically rotates through API keys and models.

Model Size - Full model checkpoints were too large for cloud deployment. Used LoRA fine-tuning to reduce checkpoint size and hosted on HuggingFace Hub.

ChromaDB Persistence - Vector store did not persist in cloud containers. Uploaded pre-built files to HuggingFace Dataset Hub with automatic download on startup.

Cold Start Latency - Slow initial load times due to sequential loading. Implemented eager loading with Streamlit caching.

Memory Constraints - Out-of-memory errors on CPU inference. Used memory-efficient loading and float32 precision.

6.2 Data Challenges

Limited Variety - Training data had repetitive patterns. Applied synthetic augmentation with five techniques, expanding the dataset by 2.72x.

Duplicates - Near-duplicate examples skewed training. Implemented MD5 hash-based deduplication.

Quality Issues - Some examples contained noise. Added filters for length, diversity, and character validation.

6.3 User Experience Challenges

Ambiguous Queries - Vague inputs produced poor results. Added edge case detection with clarification prompts.

No Explanations - Users did not understand generated SQL. Added Gemini-powered natural language explanations.

Error Opacity - Failures lacked clear feedback. Added real-time pipeline status display.

7. Ethical Considerations

The system implements privacy protection by anonymizing training data through the `anonymize()` function, which replaces email addresses, phone numbers, and Social Security Numbers with placeholder tokens. No personally identifiable information is stored in the knowledge base. Bias is mitigated through diverse training data sources and synthetic augmentation that reduces over-representation of specific query patterns.

Misuse prevention is built into the edge case handler, which detects dangerous operations like DROP, TRUNCATE, and DELETE ALL, issuing warnings instead of generating destructive queries. The system maintains transparency by allowing users to view full pipeline execution details and providing natural language explanations for every generated query. Known limitations include optimization for standard SQL only, potential need for manual refinement of complex queries, and lack of live schema validation.

8. Future Improvements

Short-term improvements include adding SQL dialect support (MySQL, PostgreSQL, SQLite), allowing users to upload custom database schemas for context-aware generation, and implementing syntax validation before output.

Medium-term goals involve fine-tuning a larger model such as Mistral 7B for improved accuracy, enabling multi-turn query building for complex queries, and adding a visual query builder interface.

Long-term vision includes direct database connectivity for query execution, learning from user feedback to improve over time, and supporting non-English natural language inputs.

9. Conclusion

The SQL Learning Assistant demonstrates successful integration of multiple generative AI techniques to address real-world challenges in SQL education. By combining RAG retrieval, fine-tuned language models, prompt engineering, and synthetic data generation, the system delivers accurate and explainable SQL queries from natural language input.

Key achievements include a knowledge base of 80,654 indexed documents, 91.33% keyword accuracy and 91.07% structure similarity in model evaluation, and 2.72x dataset expansion through synthetic augmentation. The multi-stage pipeline with Gemini enhancement provides both query refinement and beginner-friendly explanations.

The project fulfills all core requirements by implementing four of five specified components with comprehensive documentation, a deployed web application, and measurable evaluation results.

10. References

Technologies:

- TinyLlama - Zhang, P., et al. (2024). TinyLlama: An Open-Source Small Language Model
- LoRA - Hu, E., et al. (2021). LoRA: Low-Rank Adaptation of Large Language Models
- ChromaDB - The AI-native open-source embedding database
- Sentence Transformers - Reimers, N., & Gurevych, I. (2019). Sentence-BERT
- LangChain - Framework for building applications with LLMs
- Streamlit - The fastest way to build data apps

Datasets:

- Spider Dataset - Yu, T., et al. (2018). Spider: A Large-Scale Human-Labeled Dataset
- WikiSQL - Zhong, V., et al. (2017). Seq2SQL: Generating Structured Queries

Project Links:

- GitHub Repository: https://github.com/moheesh/Prompt_to_SQL_using_RAG_LLM
- Live Demo: https://huggingface.co/spaces/moheesh/sql-learning-assistant
- Project Page: https://moheesh.github.io/Prompt_to_SQL_using_RAG_LLM