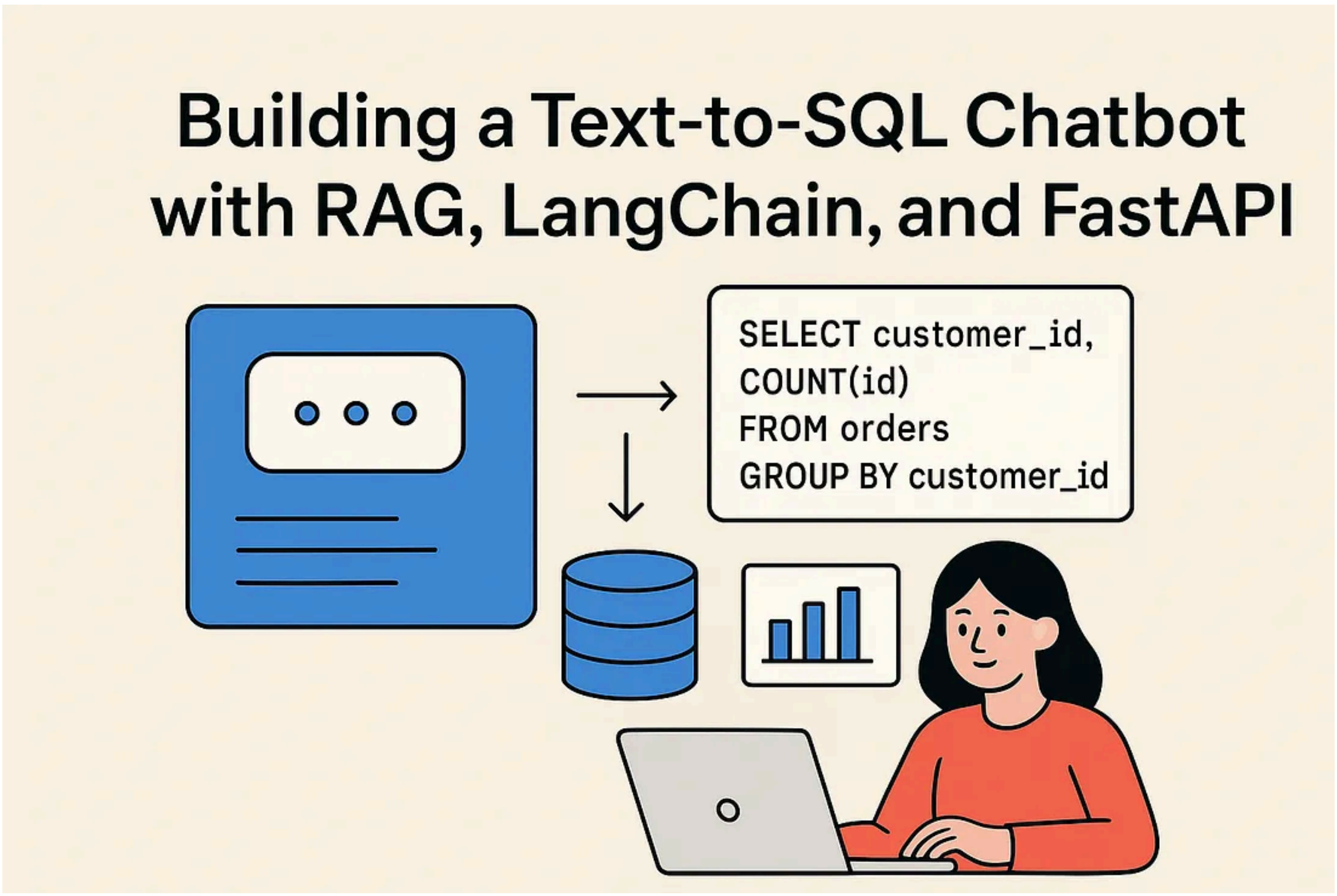
Building a Text-to-SQL Chatbot with RAG, LangChain, and FastAPI
SELECT customer_id,
COUNT(id)
FROM orders
GROUP BY customer_id

Member-only story

Building a Text-to-SQL Chatbot with RAG, LangChain, FastAPI And Streamlit

19 min read · 5 days ago

Dharmendra Pratap Singh   Follow

Listen   Share   More

In this project, I built an AI-powered chatbot that converts natural language questions into SQL queries and retrieves answers directly from a real SQLite database. Using LangChain, Hugging Face embeddings, and a Chroma vector store, this app demonstrates how to connect unstructured user input with structured data through a Retrieval-Augmented Generation (RAG) workflow — complete with a FastAPI backend and Streamlit UI.

Introduction: Why Text-to-SQL?

Imagine this:
You're in a meeting, and your manager suddenly asks,

> "Can we see all customers who joined last month?"

You glance at your SQL editor… and realize you'll have to craft a query, check table names, maybe even debug a missing JOIN. Meanwhile, someone else just *asks* an AI chatbot the same question — and instantly gets the results, neatly formatted.

That's the magic of **Text-to-SQL** — turning natural language into database queries.

The Problem

SQL (Structured Query Language) is the backbone of data analytics and engineering.
It's powerful, flexible, and precise — but it's not *accessible* to everyone.
Most business users, analysts, and even some developers find SQL syntax intimidating or slow for quick insights.

In most teams, this creates a gap:

- **Data is available**, but not easily *askable*.
- **Insights exist**, but they're locked behind SQL expertise.

The Solution: Text-to-SQL

Text-to-SQL bridges that gap.
It allows anyone — technical or not — to ask questions like:

> "What were our top 5 selling products last quarter?"

and get back the answer directly from your database, with the AI doing all the translation work behind the scenes.

Behind the curtain, a Text-to-SQL system typically does four key things:

1. **Retrieve** relevant schema and context (so it knows what tables exist).

2. **Generate** a valid SQL query from your natural-language question.
3. **Validate and execute** the SQL safely against the database.
4. **Return** the results in a human-friendly format.

Understanding the RAG Approach

So far, we've seen how Text-to-SQL lets users ask natural questions and get database results — but how does the AI *actually* know your tables, columns, and relationships?

The answer lies in a technique called **RAG**, or **Retrieval-Augmented Generation**.

What is RAG?

At its core, **RAG (Retrieval-Augmented Generation)** is a hybrid AI approach that combines **retrieval** and **generation**:

1. **Retrieval** — The system fetches relevant information (in this case, database schema, table names, and relationships).
2. **Generation** — The LLM uses that retrieved context to generate an accurate and grounded output (the SQL query).

You can think of RAG as giving your LLM a "cheat sheet" — the exact schema it needs to see before it starts writing SQL.

Why Not Just Use a Plain LLM?

If you simply ask a large language model (LLM) like GPT or Claude:

> "Show me all customers who joined last month,"

it might produce something like:

```
SELECT * FROM users WHERE signup_date >= '2025-09-01';
```

Looks fine — until you realize your actual database doesn't have a table called `users`. Maybe it's `customer_data` or `crm_clients`.

That's the problem: **LLMs hallucinate** when they don't have enough context.
They guess table names, miss JOIN relationships, or use the wrong column names —

because, by default, they don't know your database schema.

How RAG Solves This

RAG fixes hallucination by grounding the model in *real, retrieved knowledge*.

Here's what happens inside a Text-to-SQL RAG loop:

1. **Retrieve Schema Context**
   The system searches a vector database or in-memory index of your schema (table names, column descriptions, relationships).

2. **Generate SQL**
   The LLM takes both your natural language question *and* the retrieved schema as input, crafting a valid SQL query.

3. **Execute & Return Results**
   The query is validated, run against your actual database, and results are returned to the user.

This ensures that every SQL query is backed by **schema-aware reasoning**, not just model "guesswork."

The RAG Loop in Action
A simple flow diagram with the following loop:

```
[User Question]
      ↓
[Retrieve Schema Info]
      ↓
[Generate SQL (LLM)]
      ↓
[Validate + Execute]
      ↓
[Return Results]
      ↺
```

Each loop ensures the model has *up-to-date* and *accurate* schema knowledge before it generates queries — reducing hallucinations and increasing reliability.

System Architecture

Now that we understand why Text-to-SQL relies on the **RAG (Retrieval-Augmented Generation)** approach, let's look under the hood.

To make this concrete, imagine you're building a chatbot that can answer natural language questions about your company's customer database — all the way from *"Who joined last month?"* to *"What's our average order value this quarter?"*

Here's the big picture of how the system is structured 👇

1. SQLite Database — The Structured Data Source

Every Text-to-SQL system starts with a **data source**.
For simplicity, we'll use **SQLite**, a lightweight, file-based database — perfect for prototyping and testing.

This is where your data actually lives: tables like `customers`, `orders`, and `products`.

When the user asks a question, our goal is to translate that question into a valid SQL query that runs against this database.

2. Embedding Layer — Turning Schema into Vectors

Before the model can generate SQL, it needs to *understand* the database structure — table names, column names, and their meanings.

We achieve this using **embeddings:** numerical representations of text that capture semantic meaning.
With **Hugging Face Embeddings** (like `all-MiniLM-L6-v2`), we convert schema metadata into vectors:

```
import os
import sqlite3
import hashlib
from tqdm import tqdm
from dotenv import load_dotenv
from langchain_community.embeddings.huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import Chroma

# Load environment variables
load_dotenv()
SQLITE_PATH = os.getenv("SQLITE_PATH", "sample_db/sample.db")
CHROMA_DIR = os.getenv("CHROMA_DIR", "./chroma_persist")
EMBED_MODEL = os.getenv("EMBED_MODEL", "sentence-transformers/all-MiniLM-L6-v2"

# ✅ Initialize HuggingFace embedding model
```

```python
embeddings = HuggingFaceEmbeddings(model_name=EMBED_MODEL)

# ✅ Create / Load Chroma vector store
vectorstore = Chroma(
    collection_name="sqlite_docs",
    persist_directory=CHROMA_DIR,
    embedding_function=embeddings
)

def row_hash(values):
    """Generate unique hash for a row."""
    return hashlib.sha256("|".join(map(str, values)).encode()).hexdigest()

def row_to_text(table, cols, row):
    """Convert SQLite row into a readable text chunk."""
    return f"Table: {table}\n" + "\n".join([f"{c}: {v}" for c, v in zip(cols, r

def index_table(conn, table):
    """Index a single table into the vector store."""
    cur = conn.cursor()
    cur.execute(f"PRAGMA table_info({table});")
    cols = [c[1] for c in cur.fetchall()]
    cur.execute(f"SELECT {', '.join(cols)} FROM {table}")
    rows = cur.fetchall()

    docs, ids, metas = [], [], []
    for r in rows:
        txt = row_to_text(table, cols, r)
        pk = str(r[0])
        hid = row_hash(r)
        ids.append(f"{table}:{pk}")
        docs.append(txt)
        metas.append({"table": table, "pk": pk, "hash": hid})

    # Add to Chroma vector store
    vectorstore.add_texts(texts=docs, metadatas=metas, ids=ids)

def main():
    """Main indexing pipeline."""
    conn = sqlite3.connect(SQLITE_PATH)
    cur = conn.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table' AND name NOT
    tables = [t[0] for t in cur.fetchall()]

    for t in tqdm(tables, desc="Indexing tables"):
        index_table(conn, t)

    conn.close()
    print("Indexing complete and persisted in Chroma.")

if __name__ == "__main__":
    main()
```

Each table and column is now represented as a vector in high-dimensional space — enabling smart retrieval later on.

3. Vector Store (Chroma) — Finding Relevant Schema

Next, we store these embeddings in a **vector database** — in this case, **Chroma**.

When a user asks a question, we:

1. Embed the question in the same vector space.
2. Search the Chroma store for the *closest* schema elements.

For example, if the user asks *"Which users signed up recently?"*, the retriever might pull schema items like `customers.signup_date` and `customers.name`.

This ensures the model only sees relevant schema context — grounding its SQL generation in reality.

4. LangChain Components — The Brains of the System

LangChain ties everything together.
It provides modular components for retrieval, reasoning, and SQL generation:

- **Retrievers**: Pull relevant schema chunks from Chroma.
- **Chains**: Define the sequence of steps — retrieve → generate → validate → execute.
- **LLMs**: Generate SQL queries conditioned on both the user's question and retrieved schema.

Example snippet:

```
from typing import TypedDict, List
from langchain_community.vectorstores import Chroma
from langchain_core.prompts import PromptTemplate
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_community.embeddings.huggingface import HuggingFaceEmbeddings
from dotenv import load_dotenv
import os

load_dotenv()
```

```python
# state type
class RAGState(TypedDict, total=False):
    question: str
    retrieved_docs: List[str]
    generated_sql: str
    validated_sql: str
    sql_result: List[dict]
    messages: List[dict]


# init vectorstore & models
CHROMA_DIR = os.getenv("CHROMA_DIR", "./chroma_persist")
EMBED_MODEL = os.getenv("EMBED_MODEL", "text-embedding-3-small")
LLM_MODEL = os.getenv("LLM_MODEL", "gpt-4o-mini")
TOP_K = int(os.getenv("TOP_K", "8"))

embeddings = HuggingFaceEmbeddings(model_name=EMBED_MODEL)
vectordb = Chroma(persist_directory=CHROMA_DIR, embedding_function=embeddings,
retriever = vectordb.as_retriever(search_kwargs={"k": TOP_K})

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)

sql_prompt = PromptTemplate.from_template("""
        You are a SQL generator. Based on the following context, generate a SIN
        Context:
        {context}

        Question:
        {question}

        Return only the SQL SELECT statement.
        """)


async def retriever_node(state: RAGState) -> RAGState:
    docs = await retriever.ainvoke(state["question"])
    state["retrieved_docs"] = [d.page_content for d in docs]
    return state



import re
from typing import Any

async def sql_generator_node(state: RAGState) -> RAGState:
    """
    Generate SQL from the retrieved documents and user question.
    Cleans LLM output, removes markdown/code fences, and ensures only SELECT st
    """
    # 1. Combine retrieved documents
    context = "\n\n".join(state.get("retrieved_docs", []))
```

```
        # 2. Format the prompt
        prompt_text = sql_prompt.format(context=context, question=state["question"]

        # 3. Call the LLM asynchronously
        out = await llm.ainvoke(prompt_text)

        # 4. Extract text content if output is an AIMessage or ChatResult
        if hasattr(out, "content"):
            out = out.content
        out = str(out).strip()

        # 5. Remove code fences ``` or ```sql and any leading/trailing whitespace
        out = re.sub(r"```(?:sql)?\n?", "", out, flags=re.IGNORECASE).replace("```"

        # 6. Ensure the SQL starts with SELECT (case-insensitive)
        match = re.search(r"(select\b.*)", out, flags=re.IGNORECASE | re.DOTALL)
        if match:
            out = match.group(1).strip()
        else:
            # fallback if no SELECT found
            out = ""

        # 7. Optional: remove trailing semicolon if present
        out = out.rstrip(";").strip()

        # 8. Save cleaned SQL back to state
        state["generated_sql"] = out
        return state
```

LangChain acts as the **orchestrator**, ensuring that every question goes through the full RAG loop seamlessly.

5. FastAPI Backend — Serving the Pipeline

To make this system interactive, we wrap everything inside a **FastAPI backend.**

FastAPI provides a simple, high-performance way to expose the RAG pipeline as an API endpoint.

When a user sends a POST request with a question, the API:

1. Retrieves schema info.
2. Generates and validates SQL.
3. Executes the query.
4. Returns the formatted results.

This layer acts as the “engine room” for your Text-to-SQL chatbot.

```python
import os
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from dotenv import load_dotenv
from server.langgraph_nodes import retriever_node, sql_generator_node
from server.sql_validator import validate_sql
from server.executor import execute_sql
from server.utils import allowed_tables_from_db

load_dotenv()
SQLITE_PATH = os.getenv("SQLITE_PATH", "sample_db/sample.db")
ALLOWED_TABLES = allowed_tables_from_db(SQLITE_PATH)

app = FastAPI(title="RAG Text->SQL API")

class QueryRequest(BaseModel):
    question: str
    show_sql: bool = True

@app.post("/query")
async def query(req: QueryRequest):
    state = {"question": req.question, "messages": []}
    # 1. retrieve
    state = await retriever_node(state)
    # 2. generate SQL
    state = await sql_generator_node(state)
    sql = state["generated_sql"]
    ok, reason = validate_sql(sql, ALLOWED_TABLES)
    if not ok:
        raise HTTPException(status_code=400, detail=f"SQL validation failed: {r
    # 3. execute
    cols, rows = execute_sql(sql)
    # format result rows
    result = [dict(zip(cols, r)) for r in rows]
    return {"sql": sql if req.show_sql else None, "cols": cols, "rows": result}
```

6. Streamlit UI — The Chat Interface

Finally, we build a **Streamlit front-end** — a lightweight web interface for user interaction.

Users can type questions, view generated SQL (optional), and see results in real time.

Streamlit’s simplicity makes it perfect for prototyping AI tools:
a few lines of Python, and you have an interactive dashboard to talk to your data.

Putting It All Together
An end-to-end architecture diagram showing:

```
User (Streamlit UI)
        ↓
FastAPI Backend
        ↓
LangChain RAG Pipeline
   ├── Retriever (Chroma Vector DB)
   ├── Embedding Layer (Hugging Face)
   ├── LLM (SQL Generator)
        ↓
SQLite Database (Execute SQL)
        ↓
Results → Back to UI
```

This flow captures the full life cycle of a query — from plain English to executed SQL to readable results.

Check out the full code on GitHub and start experimenting yourself:
[GitHub Repo Link]

Step-by-Step Implementation

Setup & Environment

- Install dependencies:

```
pip install -r requirements.txt
```

SQLite Database Setup

- Explain the schema (e.g., `customers`, `orders`).

```
os.makedirs("sample_db", exist_ok=True)
DB = "sample_db/sample.db"
conn = sqlite3.connect(DB)
cur = conn.cursor()

cur.execute("""
CREATE TABLE IF NOT EXISTS customers (
  id INTEGER PRIMARY KEY,
  name TEXT,
  email TEXT,
  created_at TEXT
);
""")

cur.execute("""
CREATE TABLE IF NOT EXISTS orders (
  id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  total_amount REAL,
  status TEXT,
  created_at TEXT,
  notes TEXT,
  FOREIGN KEY(customer_id) REFERENCES customers(id)
);
""")
# seed data
customers = [
    (1, "Alice Johnson", "alice@example.com", "2024-12-01"),
    (2, "Bob Lee", "bob@example.com", "2024-12-05"),
    (3, "Carol Singh", "carol@example.com", "2024-12-10"),
    (4, "David Kim", "david.kim@example.com", "2024-12-12"),
.......
]

orders = [
    (1, 1, 120.50, "completed", "2025-01-03", "First order"),
    (2, 1, 15.00, "pending", "2025-01-07", "Gift wrap"),
    (3, 2, 250.00, "completed", "2025-02-10", "Bulk order"),
..........
]

cur.executemany("INSERT OR REPLACE INTO customers VALUES (?,?,?,?)", customers)
cur.executemany("INSERT OR REPLACE INTO orders VALUES (?,?,?,?,?,?)", orders)
conn.commit()
conn.close()
```

Run below command to create and insert data into sqlite database

```
python sample_db/create_sample_db.py
```

Embedding and Indexing the Schema

Before our AI can generate accurate SQL, it needs to *understand* the structure of our database — what tables exist, what columns they contain, and what kind of data they hold.

This is where **embeddings** and **semantic indexing** come into play.

What Are Embeddings?

An **embedding** is a numerical representation of text — a vector in high-dimensional space that captures semantic meaning.
For example, the phrases

> "customer name"
> and
> "client full name"
> will end up with vectors close together in that space because they *mean roughly the same thing*.

By embedding every **table name**, **column name**, and even sample data, our model can later **retrieve** relevant context when the user asks a question — e.g., "Show me recent signups" will semantically match columns like `signup_date`, `created_at`, or `join_date`.

Tools We'll Use

- **Hugging Face Embeddings** — to convert text into vectors.

- **Chroma Vector Store** — to store and retrieve those embeddings efficiently.
- **SQLite** — as the actual database we're indexing.

Here's the code indexing script 👇

```
# ✅ Initialize HuggingFace embedding model
embeddings = HuggingFaceEmbeddings(model_name=EMBED_MODEL)

# ✅ Create / Load Chroma vector store
vectorstore = Chroma(
    collection_name="sqlite_docs",
    persist_directory=CHROMA_DIR,
    embedding_function=embeddings
)

def row_hash(values):
    """Generate unique hash for a row."""
    return hashlib.sha256("|".join(map(str, values)).encode()).hexdigest()

def row_to_text(table, cols, row):
    """Convert SQLite row into a readable text chunk."""
    return f"Table: {table}\n" + "\n".join([f"{c}: {v}" for c, v in zip(cols, r

def index_table(conn, table):
    """Index a single table into the vector store."""
    cur = conn.cursor()
    cur.execute(f"PRAGMA table_info({table});")
    cols = [c[1] for c in cur.fetchall()]
    cur.execute(f"SELECT {', '.join(cols)} FROM {table}")
    rows = cur.fetchall()

    docs, ids, metas = [], [], []
    for r in rows:
        txt = row_to_text(table, cols, r)
        pk = str(r[0])
        hid = row_hash(r)
        ids.append(f"{table}:{pk}")
        docs.append(txt)
        metas.append({"table": table, "pk": pk, "hash": hid})

    # Add to Chroma vector store
    vectorstore.add_texts(texts=docs, metadatas=metas, ids=ids)

def main():
    """Main indexing pipeline."""
    conn = sqlite3.connect(SQLITE_PATH)
    cur = conn.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table' AND name NOT
    tables = [t[0] for t in cur.fetchall()]

    for t in tqdm(tables, desc="Indexing tables"):
        index_table(conn, t)

    conn.close()
    print("Indexing complete and persisted in Chroma.")
```

To generate embeddings and save into vector DB run below command

```
python ingestion/index_sqlite.py
```

Step-by-Step Explanation

1. **Extract Schema and Data**
   For each table, we grab column names and a few rows. Each row becomes a small "document" describing what that table contains.

2. **Convert Text to Vectors**
   Using `HuggingFaceEmbeddings`, every text chunk (like `"Table: customers\nname: John\nsignup_date: 2025-09-10"`) is converted into a numerical vector.

3. **Store Vectors in Chroma**
   These embeddings are stored in a **Chroma vector store**, where each vector is linked to its metadata — such as table name, primary key, and a hash ID.

4. **Enable Semantic Search**
   Later, when a user asks a question, we embed their query in the same vector space and use Chroma to *find the most relevant chunks* of schema or data.

This retrieval step gives the LLM precise context — helping it write accurate, grounded SQL queries.

Why Indexing Matters

Without embeddings, your model is flying blind — it doesn't *know* what columns exist or what data types they hold.
By embedding and indexing your schema, you create a **semantic memory** that the model can reference dynamically, ensuring it always generates valid and context-aware SQL.

```
SQLite Database
   ↓
Extract Tables + Columns
   ↓
Convert to Embeddings (Hugging Face)
   ↓
Store in Chroma Vector DB
   ↓
Semantic Search During Query Time
```

RAG Query Flow

We've built our schema index. Now it's time to put it to work.
The **RAG Query Flow** is where the magic happens — the system takes a user question, retrieves relevant schema context, generates SQL, executes it on SQLite, and returns results.

Let's break down each stage.

Step 1: Retrieve Relevant Schema

When the user asks a question — say,

> "Show me all customers who joined last month"

the system embeds this question into the same vector space as your database schema (thanks to the embeddings you created earlier).
Using the **Chroma retriever**, it fetches the most relevant schema chunks (e.g., tables and columns related to `customers`, `signup_date`, etc.).

```
async def retriever_node(state: RAGState) -> RAGState:
    docs = await retriever.ainvoke(state["question"])
    state["retrieved_docs"] = [d.page_content for d in docs]
    return state
```

Now, the model has *real schema awareness* — a cheat sheet of what exists in your database.

Step 2: Generate SQL via LLM

Next, we pass the question **and** the retrieved schema to an LLM (e.g., OpenAI, Mistral, or Gemini).

The model uses a prompt designed to produce *only* a valid, read-only `SELECT` query.

```
sql_prompt = PromptTemplate.from_template("""
You are a SQL generator. Based on the following context, generate a SINGLE READ

Context:
{context}

Question:
{question}

Return only the SQL SELECT statement.
""")
```

The LLM response is cleaned and validated to ensure it outputs *only* the SQL statement:

```
async def sql_generator_node(state: RAGState) -> RAGState:
    context = "\n\n".join(state.get("retrieved_docs", []))
    prompt_text = sql_prompt.format(context=context, question=state["question"]
    out = await llm.ainvoke(prompt_text)

    out = str(getattr(out, "content", out)).strip()
    out = re.sub(r"```(?:sql)?\n?", "", out, flags=re.I).replace("```", "").str

    match = re.search(r"(select\b.*)", out, flags=re.I | re.DOTALL)
    if match:
        out = match.group(1).rstrip(";").strip()

    state["generated_sql"] = out
    return state
```

At this point, your chatbot can take *English in* and produce *SQL out.*

Step 3: Validate the SQL (Safety First)

Before execution, we verify the query. This step protects your database and ensures correctness.

We use the `sqlglot` parser to:

- Reject any non-`SELECT` statements (no `DELETE`, `UPDATE`, etc.).
- Check that only allowed tables are referenced.
- Parse syntax safely.

```
def validate_sql(sql: str, allowed_tables: Set[str]) -> Tuple[bool, str]:
    if ";" in sql:
        return False, "semicolon or multiple statements not allowed"
    for kw in DISALLOWED:
        if f" {kw} " in f" {sql.lower()} ":
            return False, f"disallowed keyword: {kw}"

    try:
        parsed = sqlglot.parse_one(sql, read="sqlite")
    except Exception as e:
        return False, f"sql parse error: {e}"

    if parsed.key.lower() not in ALLOWED_STATEMENTS:
        return False, "only SELECT statements allowed"

    tables = extract_tables(sql)
    if not tables.issubset(allowed_tables):
        return False, f"disallowed tables used: {tables - allowed_tables}"
    return True, "ok"
```

This layer keeps your system *safe and read-only.*

Step 4: Execute on SQLite

Once validated, the SQL runs against the SQLite database.
We add a safeguard to limit results and prevent heavy queries.

```
def execute_sql(sql: str, row_limit: int = 1000, timeout=5.0) -> Tuple[List[str
    sql = enforce_limit(sql, row_limit)
    conn = open_ro_conn()
    conn.execute(f"PRAGMA busy_timeout = {int(timeout*1000)};")
    cur = conn.cursor()
    cur.execute(sql)
    cols = [c[0] for c in cur.description] if cur.description else []
    rows = cur.fetchmany(row_limit)
    conn.close()
    return cols, rows
```

The results are then packaged into JSON and sent back to the user.

FastAPI Backend

With the RAG pipeline built and tested, the next step is to **expose it as an API** so users and front-end clients can send queries, get SQL results, and interact in real time.

This is where **FastAPI** shines — it's fast, type-safe, async-ready, and perfect for serving AI-powered workflows.

Why FastAPI?

FastAPI provides:

- **Speed** — async I/O and automatic docs (Swagger / ReDoc)
- **Integration** — easy to connect with LangChain, Chroma, or local models
- **Validation** — Pydantic ensures incoming requests are clean
- **Scalability** — deploy anywhere: Docker, serverless, or on-prem

In short: it's the ideal wrapper around your RAG Text-to-SQL logic.

API Design Overview

We'll expose a single endpoint:

```
POST /query
```

Request body:

```json
{
  "question": "Show me all customers who joined last month",
  "show_sql": true
}
```

Response:

```
{
  "sql": "SELECT * FROM customers WHERE join_date >= '2025-09-01'",
  "cols": ["id", "name", "join_date"],
  "rows": [
    {"id": 1, "name": "Alice", "join_date": "2025-09-05"},
    {"id": 2, "name": "Bob", "join_date": "2025-09-12"}
  ]
}
```

Full Code:

```python
import os
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from dotenv import load_dotenv

# import the core RAG components
from server.langgraph_nodes import retriever_node, sql_generator_node
from server.sql_validator import validate_sql
from server.executor import execute_sql
from server.utils import allowed_tables_from_db

# Load environment variables
load_dotenv()
SQLITE_PATH = os.getenv("SQLITE_PATH", "sample_db/sample.db")
ALLOWED_TABLES = allowed_tables_from_db(SQLITE_PATH)

app = FastAPI(title="RAG Text->SQL API")

class QueryRequest(BaseModel):
    question: str
    show_sql: bool = True

@app.post("/query")
async def query(req: QueryRequest):
    # Initialize conversation state
    state = {"question": req.question, "messages": []}

    # 1️⃣ Retrieve relevant schema info
    state = await retriever_node(state)

    # 2️⃣ Generate SQL query using LLM
    state = await sql_generator_node(state)
    sql = state["generated_sql"]

    # 3️⃣ Validate SQL
    ok, reason = validate_sql(sql, ALLOWED_TABLES)
    if not ok:
        raise HTTPException(status_code=400, detail=f"SQL validation failed: {r

    # 4️⃣ Execute SQL safely
    cols, rows = execute_sql(sql)
    result = [dict(zip(cols, r)) for r in rows]

    # 5️⃣ Return JSON response
    return {
        "sql": sql if req.show_sql else None,
        "cols": cols,
        "rows": result
    }
```

Step-by-Step Flow

1. **Receive the Question**
   The user sends a natural-language question via POST.
2. **Retrieve Schema Context**
   `retriever_node` pulls the most relevant tables and columns from Chroma.
3. **Generate SQL**
   `sql_generator_node` uses the retrieved context and the question to produce a `SELECT` statement.
4. **Validate for Safety**
   `validate_sql()` checks that the query is well-formed and uses only allowed tables.
5. **Execute & Return**
   `execute_sql()` runs the query on SQLite (read-only mode) and returns formatted results.

Why This Design Works

- **Modular:** each function (retriever, generator, validator, executor) can be swapped out or extended.
- **Secure:** only `SELECT` queries are allowed, and tables are whitelisted.
- **Async:** the whole pipeline can run concurrently, scaling easily with user load.
- **Deployable:** wrap in Docker, deploy to AWS, GCP, or Hugging Face Spaces in minutes.

```
[User / Streamlit UI]
        ↓ (POST /query)
    FastAPI Backend
  ├── retriever_node()  →  Chroma Vector DB
  ├── sql_generator_node()  →  LLM (OpenAI / Gemini / Mistral)
  ├── validate_sql()  →  SQLGlot Parser
  └── execute_sql()  →  SQLite Database
        ↓
     JSON Response
```

This clean modular setup makes your backend robust, transparent, and production-ready.

Streamlit Frontend

Now that your FastAPI backend is up and running, let's make it **interactive**.
The goal: build a simple web interface where anyone can type a natural-language question, see the generated SQL, and view the live query results — all in seconds.

Full Code:

```
import streamlit as st
import requests

API_URL = "http://localhost:8000/query"

st.set_page_config(page_title="RAG Text→SQL Demo", layout="centered")
st.title("RAG Text → SQL (SQLite replica)")

with st.form("query_form"):
    question = st.text_input(
        "Ask a natural language question (about the DB)",
        value="Show total orders per customer"
    )
    show_sql = st.checkbox("Show generated SQL", value=True)
    submitted = st.form_submit_button("Submit")

if submitted:
    # Ensure types are correct
    question = str(question).strip()
    show_sql = bool(show_sql)

    if not question:
        st.warning("Please enter a question.")
    else:
        payload = {"question": question, "show_sql": show_sql}
        with st.spinner("Querying..."):
            try:
                resp = requests.post(API_URL, json=payload, timeout=60)
                resp.raise_for_status()
                data = resp.json()

                if show_sql:
                    st.subheader("🧠 Generated SQL")
                    st.code(data.get("sql", ""), language="sql")

                st.subheader("📊 Query Results")
                rows = data.get("rows", [])
                if rows:
                    st.dataframe(rows)
                else:
                    st.info("No rows returned for this query.")

            except requests.exceptions.HTTPError as http_err:
                st.error(f"HTTP error occurred: {http_err} - {resp.text}")
            except requests.exceptions.ConnectionError:
                st.error("Could not connect to the API. Make sure FastAPI is ru
            except requests.exceptions.Timeout:
                st.error("Request timed out. Try again later.")
            except Exception as e:
                st.error(f"Unexpected error: {e}")
```

How It Works

1. **User Input:**
   The user types a question into the Streamlit text box.

2. **Form Submission:**
   Streamlit sends the question (and the `show_sql` flag) as a JSON payload to the FastAPI endpoint (`/query`).

3. **Backend Processing:**
   The FastAPI app runs the full RAG pipeline:

- Retrieve relevant schema
- Generate SQL with the LLM
- Validate and execute on SQLite

4. Results Display:
Streamlit receives the JSON response and:

- Displays the generated SQL (if `show_sql=True`)
- Shows the result rows as an interactive table

```
+------------------------------------------------------------+
|  🧠 RAG Text → SQL (SQLite Replica)                         |
|  -------------------------------------------------------    |
|  Ask a natural language question:                           |
|  [ Show total orders per customer               ] [Submit] |
|                                                             |
|  🧠 Generated SQL                                             |
|  SELECT customer_id, COUNT(*) AS total_orders               |
|  FROM orders GROUP BY customer_id                           |
|                                                             |
|  📊 Query Results                                            |
|  ┌──────────────┬───────────────┐                            |
|  │ customer_id  │ total_orders  │                            |
|  ├──────────────┼───────────────┤                            |
|  │ 1            │ 12            │                            |
|  │ 2            │ 8             │                            |
|  └──────────────┴───────────────┘                            |
+------------------------------------------------------------+
```

This gives users an instant feedback loop — from *plain English* to *SQL to data visualization.*

Run the app using below commands:

```
uvicorn main:app --reload
streamlit run app.py
```

Demo:

Question: Show total orders per customer

Question: Total revenue from completed orders

Next Steps: Incremental Embeddings & Scalable Updates

By now, you've built a complete **RAG-powered Text-to-SQL pipeline** — from schema embeddings and retrieval to SQL generation, validation, and execution.
But in real-world scenarios, **databases evolve constantly**. Tables change, new records are inserted, and column definitions shift over time.

So how do we keep our embedding index — and therefore our RAG system — up to date?

Let's explore some next steps to make your project *production-ready*.

1. Handling Dynamic Databases

In our prototype, we indexed the database *once* at startup.
That works great for static datasets, but production systems need **incremental embedding updates** whenever:

- A table structure changes (new or dropped columns).
- New rows are added that change context meaningfully.
- Schema documentation or metadata evolves.

Instead of re-indexing everything, you can **embed only what changed**.

Incremental Indexing Strategy:

- **Track updates** via timestamps or row hashes.
- Compare against your Chroma metadata (e.g., stored hash IDs).
- Re-embed only modified rows or new tables.
- Persist embeddings using Chroma's incremental `add_texts()` API.

This approach minimizes cost, time, and redundancy — essential when dealing with millions of rows.

2. Automate with Background Jobs

Re-indexing or updating embeddings shouldn't block user queries.
You can offload these operations to **background workers** using:

- **Celery** (with Redis/RabbitMQ) — for distributed task management.
- **FastAPI BackgroundTasks** — for lightweight async updates.

Example using FastAPI's built-in background jobs:

```
from fastapi import BackgroundTasks

@app.post("/update_embeddings")
async def update_embeddings(background_tasks: BackgroundTasks):
    background_tasks.add_task(reindex_changed_tables)
    return {"status": "update scheduled"}
```

This allows your main API (`/query`) to remain responsive while indexing happens in the background.

3. Future Enhancements

Here are a few directions for further innovation:

- **Fine-tune the SQL generator** on your company's query logs.
- **Add caching** for frequently asked questions.
- **Integrate charts** into Streamlit to visualize query results dynamically.
- **Switch to larger vector DBs** (like Pinecone or Qdrant) for massive datasets.
- **Add feedback loops** — let users correct SQL errors and retrain the model.

Each improvement moves your system closer to a **self-learning data assistant** that understands your evolving schema and business logic.

Key Learnings

You've just built something powerful — a complete **RAG-based Text-to-SQL system** that turns plain English into actionable database insights. Let's recap what you accomplished:

1. Mastered RAG for Text-to-SQL

You learned how **Retrieval-Augmented Generation (RAG)** bridges the gap between **natural language** and **structured database schemas**, dramatically improving accuracy and reducing hallucinations in SQL generation.

- ✅ Embedded database schema and sample rows for context.
- ✅ Retrieved the most relevant chunks before query generation.

- ✅ Combined retrieval + LLM reasoning for reliable SQL output.

2. Built a Modular, Scalable Architecture

You structured your system like a production-ready AI service using **LangChain** and **FastAPI:**

- **SQLite** as your structured data source.
- **Hugging Face embeddings + Chroma** for semantic search.
- **LangChain** for orchestrating retrievers, prompts, and models.
- **FastAPI** as the lightweight, asynchronous backend.
- **Streamlit** for a clean, user-friendly interface.

Each layer is modular — meaning you can swap components (like models or databases) without breaking the rest of the pipeline.

3. Deployed an Interactive Chat Interface

Finally, you wrapped everything in a **Streamlit frontend**, letting users:

- Type natural-language questions.
- See generated SQL queries.
- Instantly view results in a dynamic table.

You turned what was once a developer-only task (writing SQL) into a **natural conversation with data.**

Final Thoughts

This project highlights a powerful shift in how we interact with data: **AI is democratizing database access**, breaking down barriers so *anyone* can ask complex questions without knowing SQL.

By combining retrieval-augmented generation with modern embeddings and conversational interfaces, we're creating tools that put data directly into people's hands — making insights faster, easier, and more accessible than ever before.

I'd love to hear your feedback, ideas, and experiences building your own Text-to-SQL systems.

Check out the full code on GitHub and start experimenting yourself:
[GitHub Repo Link]

Let's build the future of data conversations together! 🚀

Langchain AI Rags Sql Fastapi

Follow

Written by Dharmendra Pratap Singh

102 followers · 14 following

Hi, I'm Dharmendra Pratap Singh — I've spent 11 years building software and now I'm focused on LLMs, LangChain, and the promise of Generative AI.

Responses (5)

Bgerby

What are your thoughts?

Isaac
1 day ago

the design is applicable to simplified prototyping and basic use cases. It faces limitations when dealing with complex SQL logic (e.g., multi-table joins, window functions, CTEs) and non-intuitive table schemas (e.g., ambiguous naming, hidden relationships)

-- Reply

JBarti he
8 hours ago (edited)

Love this!

Really simply explains how to do RAG.

A lot of resources tend to overcomplicate it with much theory.

This story is great. It builds an interesting scenario, presents a solution, and explains it really well.

Kudos for the good work.

-- Reply

Ganesan J
13 hours ago

It is really informative concept. But I try this concept is my small application contain 50+table, it is working fine.

But same architecture for more than 500+ I face some difficult like user context and validate and time consuming etc..

-- 1 reply Reply

See all responses