Project Overview: SQL AI Query Bot & EDA Agent

This project is a dual-purpose AI application that allows users to:

- **Chat with a SQL Database:** Convert natural language questions into SQL queries.
- **Analyze CSV Files (EDA):** Upload datasets and have an AI write Python code to analyze and plot data.

It is built with a FastAPI backend and a React frontend, using Google Gemini as the intelligence layer.-----📂 **Project Structure & File Guide**

Here is a detailed breakdown of the files and folders:

1. Root Directory

File/Folder	Description
`README.md`	The instruction manual. It tells you how to set up the project (install dependencies, set up database, run servers) and what the features are.
`.gitignore`	Tells Git which files to ignore (not track). It lists things like `venv` (virtual environment), `.env` (secrets), `__pycache__` (compiled python files), and `node_modules` (frontend dependencies).
`data.csv`	A sample dataset (Bank Marketing data) provided to test the "EDA Agent" feature. You can upload this to the app to ask questions like "Plot the age distribution".

2. Backend (`/backend`)

This contains all the Python code for the server and AI logic.

Category	File/Folder	Description
Configuration & Setup	`.env`	Stores secrets and configuration: `GOOGLE_API_KEY`, `DB_...` (Database connection

		details). *Note: This file is ignored by git.*
	`requirements.txt`	Lists all the Python libraries the project needs: `fastapi`, `uvicorn`, `langchain...`, `sqlalchemy`, `psycopg2`, `pandas`, `matplotlib`, `seaborn`.
Database Scripts	`schema.sql`	The blueprint for the database. Creates `chat_sessions` and `chat_messages` tables.
	`run_init_db.py`	A helper script to setup the database. Runs logic to create tables and seed initial data.
	`debug_db.py`	A diagnostic script to check if your database connection works and lists the tables it finds.
Tests	`test_safety.py`	Tests the "Safety Railings". Ensures malicious queries (like "Drop the chat_sessions table") are blocked.
	`test_sessions.py`	Tests the API endpoints for creating and listing chat sessions to ensure history works.
	`test_upload.py`	Tests the CSV upload functionality.

3. Application Logic (`/backend/app`)

This is where the actual application code lives.

File	Description

`main.py`	**The Entry Point.** Sets up the FastAPI application. Defines API endpoints: `POST /api/chat`, `POST /api/eda_chat`, `POST /api/upload_csv`, `GET/POST /api/sessions`. Mounts the `workspace/plots` directory.
`agent.py`	**The SQL Agent (3-Stage Pipeline). Planner Stage:** Writes a SQL Plan (JSON). **Executor Stage:** Runs SQL (with safety checks). **Responder Stage:** Writes a natural language answer.
`eda_agent.py`	**The EDA (Exploratory Data Analysis) Agent. Planner:** Writes Python code (pandas/matplotlib). **Executor:** Runs the Python code in a sandboxed directory. **Responder:** Explains the results.
`database.py`	**The Toolbox for database operations.** Includes functions like `get_database_url()`, `init_db()`, `create_session()`, and `add_message()`.

4. Zeno (`/zeno`)

File	Description
`generate_extra_graphs.py`	A script used to generate "Research Paper" style graphs (e.g., Latency Distribution, Metric Comparison) for documentation or reporting.

5. Frontend (`/frontend`)

Description
Web interface built with React and Vite. Talks to the backend API to send messages and display results/tables/charts.

SQL Execution Flow Document

This document details the step-by-step code execution for three common SQL Agent scenarios.

Core File: `backend/app/agent.py`
Entry Point: `backend/app/main.py` -> `chat()` endpoint-----1. Scenario: Create or Delete a Table

User Prompt: "Create a table named 'employees' with columns id and name"Execution Flow

Stage	File/Function	Action/Description
API Request	`backend/app/main.p…`	Receives the POST request, extracts `message` ("Create a table..."), connects to DB, and calls `get_agent_response`.
Agent Initialization	`backend/app/agent.…`	Initializes `ChatGoogleGenerativeAI` (LLM) and `SQLDatabase` (db connection). Calls `get_schema_info` to understand the current state of the DB.
Stage 1: The Planner (Generating SQL)	`backend/app/agent.…`	Constructs a big text prompt ("You are a SQL Expert...", User Goal, Database Schema). Sends this to Google Gemini models.
	Generated Output (Inte…	`json\n{\n "plan_description": "Create employees table",\n "queries": ["CREATE TABLE employees (id SERIAL PRIMARY KEY, name VARCHAR(100))"]\n}`
Stage 2: The Executor (Running SQL)	`backend/app/agent.…`	Loops through queries. Includes a **Safety Check** for protected tables (`chat_sessions`, `chat_messages`). Executes: `db.run(query)`.

	Result	`"Success"` (or an empty string indicating the command ran).
Stage 3: The Responder (Explaining Results)	`backend/app/agent.…`	Constructs a prompt with the User Question and Execution Log. Asks Gemini to explain what happened.
	Result	"I have successfully created the 'employees' table with columns id and name."

-----2. Scenario: Reading the DB (Listing Tables)

User Prompt: "What tables are in the database?"Execution Flow

Stage	File/Function	Action/Description
Context Loading (The "Reading" Part)	`backend/app/agent.py`: `get_schema_info(db)`	**Before every request**, the agent inspects the database. Calls `db.get_usable_table_names()` and formats it into a string (e.g., `Table: students\nTable: chat_sessions...`). This schema is injected into the Planner's prompt.
Stage 1: The Planner	`planner_stage(...)`	The prompt includes the schema info. The LLM generates a metadata query (Option A) or, less commonly, answers directly (Option B).
	Generated Output (Option A)	The LLM generates a metadata query: `SELECT table_name FROM information_schema.tables WHERE table_schema = 'public'`.

Stage 2 & 3: Executor & Responder	`executor_stage` / `responder_stage`	Executor runs the `SELECT` query. Responder reads the list of tables returned (e.g., `[('students',), ('employees',)]`) and lists them for the user.

-----3. Scenario: Modify/Query Existing Table

User Prompt: "Add a student named 'Alice' to the students table"Execution Flow

Stage	File/Function	Action/Description
Stage 1: The Planner	`planner_stage(...)`	The `schema_info` in the prompt indicates the `students` table columns (e.g., `name`, `class`, `section`, `marks`). The LLM infers missing values or generates a specific `INSERT`.
	Generated Output	`json\n{\n "queries": ["INSERT INTO students (name) VALUES ('Alice')"]\n}`
Stage 2: The Executor	`executor_stage(db, plan)`	Executes `db.run("INSERT INTO students...")`. The new row is added to the database.
Stage 3: The Responder	`responder_stage(...)`	Receives the execution log showing "Success". Generates text: "Added Alice to the students table."

-----Summary of Files & Key Functions

Functionality	File	Key Functions
API Endpoint	`backend/app/mai…`	`chat()`

Agent Controller	`backend/app/age…`	`get_agent_response()`
Database Inspection	`backend/app/age…`	`get_schema_info()`
Thinking (AI)	`backend/app/age…`	`planner_stage()`
Action (SQL)	`backend/app/age…`	`executor_stage()`
Response (AI)	`backend/app/age…`	`responder_stage()`
DB Connection	`backend/app/dat…`	`get_database_url()`, `SQLDatabase` (LangChain)

Note on "Files Generated"

- **SQL Agent:** This flow **does not generate any files** on your disk. It simply reads/writes to the PostgreSQL database.
- **EDA Agent (different flow):** If you were using the EDA Agent (for CSV analysis), it *would* generate Python files and PNG plots in `backend/workspace/plots`. This is handled by `eda_agent.py`, not the SQL flow described above.

EDA (Exploratory Data Analysis) Execution Flow

This document details the step-by-step code execution for the EDA Agent (CSV analysis).-----**Core File:** `backend/app/eda\_agent.py`

Entry Point: `backend/app/main.py` -> `eda\_chat()` endpoint-----1. Scenario: Uploading a CSV File

User Action: Drag and drop a CSV file (e.g., `data.csv`) in the UI.

Step	File/Function	Action	Result/Location
API Request	`backend/app/…`	Receives file stream. Generates a unique UUID filename (e.g., `1234-abcd.csv`).	-
File Generation (Saving)	-	Server writes actual file content to disk.	`backend/workspace/uploads/1234-abcd.csv`
Preview Generation	-	Reads the first 5 rows using pandas.	Returns JSON with filename (UUID), columns, and preview data to the frontend.

-----2. Scenario: Analyzing Data (Plotting/Querying)

User Prompt: "Plot the distribution of age as a histogram"

Step	File/Function	Action
API Request	`backend/app/main.py`: `eda\_chat(request: EdaChatRequest)`	Payload contains message, filename (UUID from step 1), and history. Calls `get\_eda\_response(...)`.
Data Loading	`backend/app/eda\_agent.py`: `get\_eda\_response(...)`	Locates file at `workspace/uploads/<filename>`. Loads into a Pandas DataFrame (`df = pd.read\_csv(...)`). Captures `df.info()` and `df.head()` to create a "Dataframe Info" summary for the LLM.

Stage 1: The Planner (Generating Python Code)

File/Function	Action	Generated Output (Internal)

backend/app/eda_agent.py: planner_stage(...)	Constructs a prompt for Gemini: "You are a Python Data Analysis Expert... Dataframe Info: [columns, types]... User Goal: Plot age...". Asks Gemini to write Python Code.	python\nimport matplotlib.pyplot as plt\nimport seaborn as sns\nplt.figure(figsize=(10, 6))\nsns.histplot(df['age'], kde=True)\nplt.title('Age Distribution')\nplt.savefig('age_hist.png') # Crucial: Saves the plot to disk!\nplt.clf()\nprint("Plotted the distribution of age.")\n

Stage 2: The Executor (Running Python & Creating Files)

File/Function	Action	Result/Location
backend/app/eda_agent.py: executor_stage(code, df)	Creates a secure Temp…	-
File Generation	-	The executed code generates PNG files (e.g., age_hist.png) in the temp directory.
Processing Results	-	Captures stdout (any print() output). Scans the temp directory for any .png files created.

Moving Files	-	Renames images to unique UUIDs (e.g., `plot\_xyz123.png`) and moves them.
Result Update	-	Updates the result object with the web-accessible path.

Stage 3: The Responder (Explaining Results)

File/Function	Action	Result
`backend/app/eda\_agent.py`: `responder\_stage(...)`	Takes the `stdout` and the fact that plots were generated. Asks Gemini to write a friendly response.	"I have generated a histogram showing the distribution of age. You can see the plot below."

-----Summary of Files & Functions

Functionality	File	Key Functions	Generated Files (Where?)
API Endpoint	`backend/app/…`	`upload\_csv`, `eda\_chat`	Uploads: backe…
Agent Controller	`backend/app/…`	`get\_eda\_response`	-
Code Generation	`backend/app/…`	`planner\_stage`	(None, code is i…
Code Execution	`backend/app/…`	`executor\_stage`	Plots: backend/…
Explanation	`backend/app/…`	`responder\_stage`	-

-----Key Differences from SQL Agent

Feature	SQL Agent	EDA Agent
Language	Generates SQL	Generates Python
State	Relies on the external Database state	Loads a CSV file into memory for every request
Artifacts	Creates data (rows/tables)	Creates files (PNG plots) on the server disk

LangChain Usage Analysis

LangChain is used as the orchestration framework to connect the LLM (Google Gemini) with your data (SQL Database and CSV files).

Here is exactly where and how it is used in your codebase:1. Google Gemini Integration (The "Brain")

Component: `ChatGoogleGenerativeAI`

Package: `langchain_google_genai`

This is the wrapper that allows LangChain to talk to Google's Gemini models.

Used in:

- `backend/app/agent.py` (SQL Agent)
  # Line 165
- llm = ChatGoogleGenerativeAI(
- model=os.getenv("GEMINI_MODEL", "gemini-1.5-flash"),
- google_api_key=google_api_key,
- temperature=0
-)
- `backend/app/eda_agent.py` (EDA Agent)
  # Line 170
- llm = ChatGoogleGenerativeAI(
- model=os.getenv("GEMINI_MODEL", "gemini-1.5-flash"),
- google_api_key=google_api_key,
- temperature=0
-)

Purpose: To send prompts to Gemini and get text responses back. The `temperature=0` setting ensures the AI is deterministic (less creative, more precise), which is cleaner for generating code/SQL.2. SQL Database Tools (The "connector")

Component: `SQLDatabase`

Package: `langchain_community.utilities`

This provides a unified way to interact with SQL databases, abstracting away the underlying driver (PostgreSQL vs MySQL etc).

Used in:

- `backend/app/agent.py`
  # Line 174
- db = SQLDatabase(engine)

Key Functions Used:

- `db.get_usable_table_names()`: Fetches list of tables (Step 1 of execution).
- `db.get_table_info()`: Dumps the `CREATE TABLE` schema text (Step 1 of execution).
- `db.run(query)`: Executes the SQL query generated by the AI (Step 2 of execution).

3. Prompt Management (Implicit)

While you aren't using LangChain's `PromptTemplate` class explicitly (you are using f-strings), you are following the LangChain pattern of "Chains" manually.

Manual Chaining in `agent.py`:

1. **Context Construction:** Getting `schema_info` from the `SQLDatabase` tool.
2. **Prompt 1 (Planner):** Sending Schema + User Query to LLM -> Get SQL.
3. **Action:** Executing SQL.
4. **Prompt 2 (Responder):** Sending SQL Results + User Query to LLM -> Get Answer.

Summary Table

File	LangChain Component	Purpose
`backend/app/agent.…`	`ChatGoogleGenerati…`	The Intelligence: Generates SQL and text answers.

`backend/app/agent.…`	`SQLDatabase`	The Tool: Reads schema metadata and runs queries.
`backend/app/eda_ag…`	`ChatGoogleGenerati…`	The Intelligence: Generates Python code for CSV analysis.

Note on `requirements.txt`: You have `langchain-experimental` installed but it acts as an unused dependency in the current code. It is often used for `create_pandas_dataframe_agent`, but your project implements a custom EDA agent (`backend/app/eda_agent.py`) instead of using the out-of-the-box LangChain one. This gives you more control over the plotting and file saving mechanism.