SQL QUERY SYSTEM DOCUMENTATION

Conversational Data Analytics System with Multi-Turn Context & Visualization

Created By – Nuhan Gunasekara

Table of Contents

##

Deploying System on New Workspace

(Environment Setup)

##

System Requirements

- **OS**: Windows 10/11, Linux (Ubuntu 20.04+), macOS 11+
- **RAM**: 8 GB minimum, 16 GB+ recommended
- **Storage**: 10 GB for models + data
- **CPU**: Modern multi-core processor
- **GPU**: Optional (speeds up Ollama if NVIDIA with CUDA)

References

- **Ollama Documentation**: https://ollama.com/docs
- **LangChain Documentation**: https://python.langchain.com/docs
- **ChromaDB Documentation**: https://docs.trychroma.com/
- **Gradio Documentation**: https://www.gradio.app/docs
- **Plotly Documentation**: https://plotly.com/python/

1. Install Python 3.10+

Download from python.org or use winget

```bash
winget install Python.Python.3.10

# Verify Installation

python --version

# Expected: Python 3.10.x or higher
```

2. Install Ollama

- Download installer from: https://ollama.com/download/windows
- Run `OllamaSetup.exe`
- Follow installation wizard
- Ollama will start automatically

To check ollama is exists

```*bash*
*Test-Path "C:\Program Files\Ollama\ollama.exe"*
```

Verify Ollama

```*bash*
*ollama –version*
```

Test Ollama API

```*bash*
*curl -s http://localhost:11434/api/tags*
```

Troubleshooting

If Ollama connection fails

Invoke-WebRequest -Uri "http://localhost:11434/api/tags" -UseBasicParsing | Select-Object -ExpandProperty Content } catch { Write-Host "Ollama not running or not accessible" }

Check if model is properly loaded

curl http://localhost:11434/api/tags

Restart Ollama if needed

taskkill /f /im ollama.exe

ollama serve

#ollama serve for start the ollama service

3. Create Virtual Environment and Install Python Dependencies

Create .venv

```bash
#Create
python -m venv .venv
#Activate
.\.venv\Scripts\Activate.ps1
```

Install requirements.txt – first, copy requirements.txt file into the project folder

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

4. Start Ollama Service

If it is not running already

```bash
# Start in background
ollama serve

# Verify server
curl http://localhost:11434/api/tags
```

5. Download Ollama model

```bash
# Main LLM model (7B parameters, ~4.7 GB)
ollama pull qwen2.5:7b

# Embedding model (~275 MB)
ollama pull nomic-embed-text

# Verify models
ollama list
```

Expected output:

NAME	ID	SIZE	MODIFIED
qwen2.5:7b	abc123def456	4.7 GB	2 hours ago
nomic-embed-text	ghi789jkl012	275 MB	2 hours ago

Glossary

- **CTE**: Common Table Expression (SQL WITH clause)
- **ChromaDB**: Vector database for semantic search
- **Embeddings**: Numerical representations of text (768-dim vectors)
- **Intent**: User's goal (NEW_QUERY, RE_VISUALIZE, TRANSFORM, etc.)
- **LLM**: Large Language Model (qwen2.5:7b)
- **Ollama**: Local LLM server
- **Plotly**: Interactive charting library
- **Pydantic**: Data validation using Python type hints
- **Semantic Search**: Finding relevant data by meaning, not exact keywords
- **Temperature**: LLM randomness parameter (0.0-1.0)
- **Vector Database**: Database optimized for similarity search

##

Initializing Databases

(SQL Database and Vector Database for Metadata)

##

1. Converting Excel and CSV Files to a SQL Database.

I created ***excel_to_db.py*** script as a smart data ingestion tool that converts raw CSV and Excel files into a structured SQLite database for ease purpose.

SQLite was chosen for this architecture because it requires zero configuration. We don't need to install or configure a database server; the database is just a file that lives alongside the project.

Unlike standard importers that blindly treat everything as text or guess types poorly, this script uses a LLM to:

1. Understand the data → It analyzes column names and sample values to determine the true business meaning.

2. Infer correct types → It distinguishes between `INTEGER`, `REAL` (float), `DATE`, and `TEXT` with high accuracy.

3. Generate schema → It creates a proper SQL table with constraints (e.g., `NOT NULL`).

4. Clean data → It sanitizes column names into database-friendly `snake_case`.

5. Handle Multi-Sheet Workbooks → It can process an entire Excel workbook, creating a separate table for each sheet.

Prerequisites

- Ollama running locally (`ollama serve`)
- Model pulled : `qwen2.5:7b` (or other specified model)
- Python environment with requirements installed (`pandas`, `openpyxl`, `langchain`, etc.)

analyze_existing_db.py

```
├── pandas
├── sqlite3
├── langchain_ollama (ChatOllama)
├── langchain_core (PromptTemplate, StrOutputParser)
├── pydantic (BaseModel, Field)
└── langchain_core.output_parsers (PydanticOutputParser)
```

QueryAgent_Ollama.py

```
│
├── pandas
├── sqlite3
├── plotly (express, graph_objects)
├── langchain_ollama (ChatOllama)
├── langchain_core (PromptTemplate, StrOutputParser)
├── langchain_community (SQLDatabase, QuerySQLDatabaseTool)
├── langchain_classic (create_sql_query_chain)
└── pydantic (BaseModel, Field)
```

main_query_agent.py

```
│
├── argparse
├── json
├── os
├── QueryAgent_Ollama (QueryAgent)
└── plotly.graph_objects (for viz export)
```

Command Line Arguments

Argument	Default	Description
input_file	**Required**	Path to the source CSV or Excel file (e.g., `sales.xlsx`)
--db	"database_name.db"	Path where the SQLite database will be created
--model	"qwen2.5:7b"	The LLM used for analyzing data types
--sheets	"None"	Optional list of specific sheet names to process (Excel only)

Example Commands

Basic CSV Run

```bash
python excel_to_db.py sales_data.csv
```

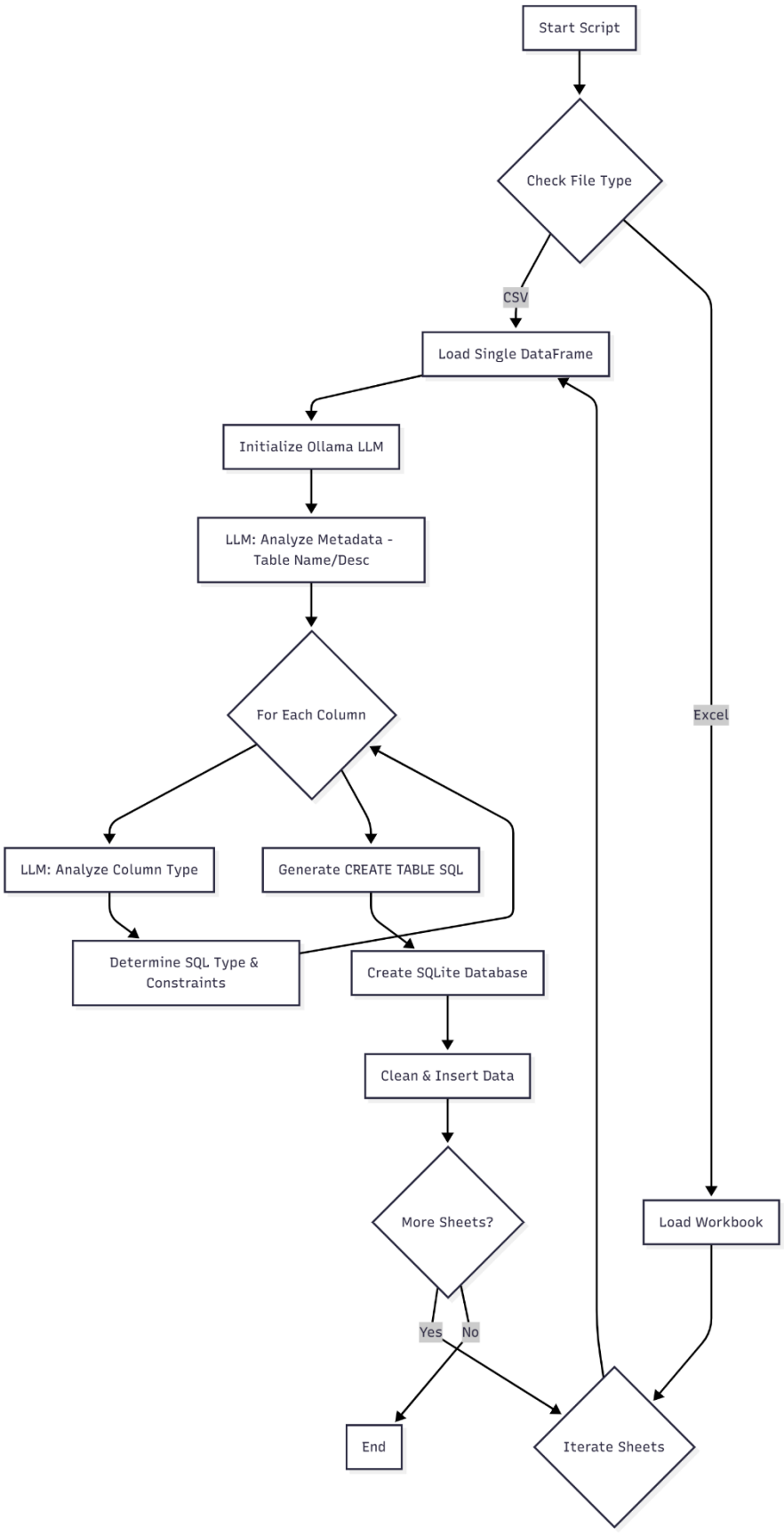
Excel Workbook (All Sheets)

```bash
python excel_to_db.py financial_report.xlsx --db finance.db
```

Excel Workbook (Specific Sheets)

```bash
python excel_to_db.py financial_report.xlsx --sheets "Q1 2024" "Q2 2024"
```

excel-to-db.py



Class & Method Reference

Class: *`CSVtoDatabaseConverter`*

`__init__(model, db_path)`

Initializes the LLM connection and sets up the output database path.

`convert(input_file, sheets=None)`

The main orchestration method.

1. Detects file type (CSV vs Excel).
2. Iterates through sheets (if Excel) or processes single file (if CSV).
3. Calls *`_process_dataframe`* for each dataset.

`analyze_metadata(df, source_label)`

Input: DataFrame and sheet name/filename.

Process: Asks LLM to suggest a table name, description, and category based on the first 5 rows.

Output: Dictionary with table metadata.

`analyze_column(column_name, column_data)`

Input: Column name and sample values.

Process: Asks LLM to determine the best SQL type (`INTEGER`, `REAL`, `DATE`, `TEXT`) and constraints.

Fallback: Uses *`_auto_detect_type`* (Pandas logic) if LLM fails.

`_process_dataframe(df, source_label)`

Handles the pipeline for a single table:

1. Calls `analyze_metadata` for table-level info.
2. Iterates columns and calls `analyze_column`.
3. Calls `create_database` to finalize the table creation.

`create_database(df, metadata, column_analysis)`

1. Generates a `CREATE TABLE` statement using the inferred types.
2. Sanitizes column names (e.g., "Order Date" -> `order_date`).
3. Performs type conversion on the DataFrame (e.g., string dates to datetime objects).
4. Inserts data into SQLite using `to_sql`.

Data Structures (Pydantic Models)

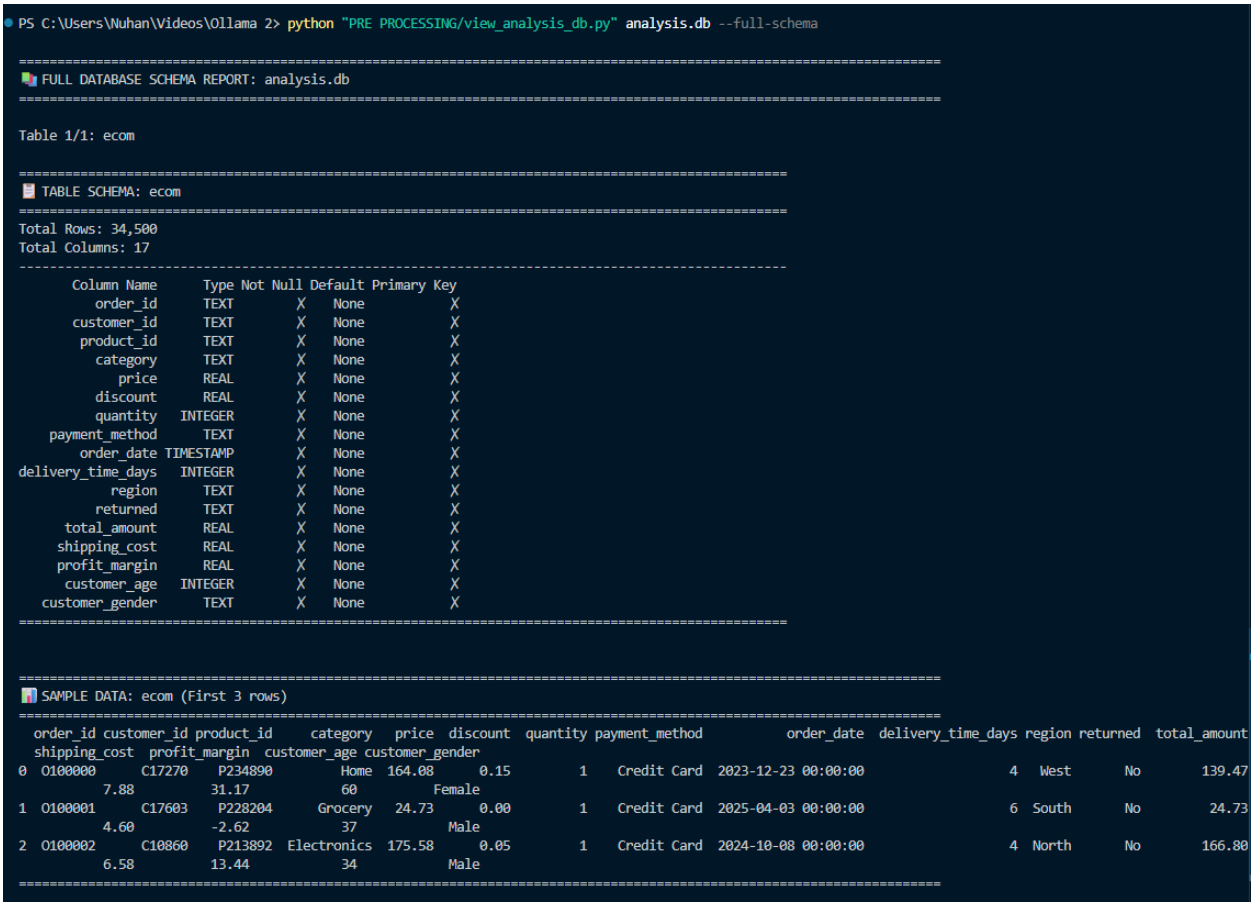
`MetadataResponse`

* `table_name`: Suggested snake_case name.
* `description`: Business description.
* `category`: Data category (Sales, HR, etc.).

`DataTypeResponse`

* `sql_type`: `TEXT`, `INTEGER`, `REAL`, `DATE`, `BOOLEAN`.
* `constraints`: List of SQL constraints (e.g., `NOT NULL`).
* `is_nullable`: Boolean indicating if NULLs are allowed.

Using view_analysis_db.py you can get the overview of created database.



```
PS C:\Users\Nuhan\Videos\Ollama 2> python "PRE PROCESSING/view_analysis_db.py" analysis.db --full-schema

================================================================================================================
FULL DATABASE SCHEMA REPORT: analysis.db
================================================================================================================

Table 1/1: ecom

================================================================================
TABLE SCHEMA: ecom
================================================================================
Total Rows: 34,500
Total Columns: 17
--------------------------------------------------------------------------------
       Column Name      Type Not Null Default Primary Key
          order_id      TEXT        X    None           X
       customer_id      TEXT        X    None           X
        product_id      TEXT        X    None           X
          category      TEXT        X    None           X
             price      REAL        X    None           X
          discount      REAL        X    None           X
          quantity   INTEGER        X    None           X
    payment_method      TEXT        X    None           X
        order_date TIMESTAMP        X    None           X
delivery_time_days   INTEGER        X    None           X
            region      TEXT        X    None           X
          returned      TEXT        X    None           X
      total_amount      REAL        X    None           X
     shipping_cost      REAL        X    None           X
     profit_margin      REAL        X    None           X
      customer_age   INTEGER        X    None           X
   customer_gender      TEXT        X    None           X
================================================================================


================================================================================================================
SAMPLE DATA: ecom (First 3 rows)
================================================================================================================
  order_id customer_id product_id     category   price  discount  quantity payment_method          order_date  delivery_time_days region returned  total_amount
  shipping_cost  profit_margin  customer_age customer_gender
0  O100000      C17270    P234890         Home  164.08      0.15         1    Credit Card  2023-12-23 00:00:00                   4   West       No        139.47
          7.88         31.17           60          Female
1  O100001      C17603    P228204      Grocery   24.73      0.00         1    Credit Card  2025-04-03 00:00:00                   6  South       No         24.73
          4.60         -2.62           37            Male
2  O100002      C10860    P213892  Electronics  175.58      0.05         1    Credit Card  2024-10-08 00:00:00                   4  North       No        166.80
          6.58         13.44           34            Male
================================================================================================================
```

2. Analyzing Database to Extract Metadata

The ***analyze_existing_db.py*** script is the foundation of the semantic intelligence layer for the Conversational Data Analytics System. It performs a one-time ingestion and understanding process on an existing SQLite database.

Instead of just reading schema names (which might be cryptic like `tbl_01`), this script uses LLM to analyze representative samples of the actual data, infer business meaning, and store this knowledge in a Vector Database (ChromaDB).

Smart SQL sampling means even very large tables can be profiled without loading them fully into memory, while still giving the main application enough semantic context to answer natural-language queries like "Show me revenue" even if the table is named `transactions`.

Prerequisites

- Ollama running locally (`*ollama serve*`)

- Models pulled: `qwen2.5:7b` (logic) and `*nomic-embed-text*` (embeddings)

Command Line Arguments

Argument	**Default**	**Description**
source_db	**Required**	Path to the source SQLite database file
--vector-db	"chroma_db_768dim"	Directory where the Vector Database will be created
--model	"qwen2.5:7b"	The LLM used for analyzing data context
--embedding-model	"nomic-embed-text"	The model used for generating vector embeddings
--ollama-url	http://localhost:11434	URL of the local Ollama server
--temperature	"0.1"	Creativity of the LLM
--chunk-size	“10000”	How many rows to pull per SQL chunk when needed
--sample-size	“1000”	Number of random rows to sample per table for LLM context

Class & Method Reference

Class: *`ExistingDatabaseAnalyzer`*

This is the main controller class that orchestrates the entire process.

`__init__(...)`

Initializes connections to SQLite, Ollama, and ChromaDB. Sets up Pydantic parsers for structured LLM output and exposes `chunk_size` / `sample_size` knobs so you can control how much data is retrieved per table.

`setup_vector_database()`

Creates or loads two collections in ChromaDB:

1. *`table_metadata`*: Stores table-level info (description, category).
2. *`column_metadata`*: Stores column-level info (types, constraints, meaning).

`analyze_all_tables()`

The main entry point. Iterates through every table in the source database and calls *`analyze_table()`*.

`analyze_table(table_name)`

Orchestrates the analysis for a single table without loading it fully:

1. Uses SQL *(`COUNT(*)`, `PRAGMA table_info`)* to gather row counts and column names.
2. Pulls a random sample of up to *`sample_size`* rows for LLM context.
3. Calls *`_analyze_table_metadata`* with the lightweight sample plus the true row count.

4. Iterates through columns and calls `*_analyze_column_efficient*` (SQL-based stats + minimal sampling) for specific details.

5. Returns a dictionary with full analysis results.

`_analyze_table_metadata(table_name, df, total_rows)`

Input: Table name, sampled rows (DataFrame of up to `sample_size` records), and the true row count.

Process: Sends a prompt to the LLM asking it to describe the business purpose, suggest a primary key, and categorize the data based on the sample while respecting the full table size.

Output: `MetadataResponse` object (description, category, etc.).

`_analyze_column_efficient(conn, table_name, column_name, total_rows)`

Input: SQLite connection, column name, true table row count.

Process: Runs SQL aggregations *(`COUNT(DISTINCT)`, `COUNT` NULLs)* plus a tiny random sample (≤20 values) to brief the LLM. Only 100 values are fetched for fallback type detection, so no column is fully materialized.

Fallback: If LLM parsing fails, `*_auto_detect_type*` inspects the small sample to infer SQL/Python types.

`save_analysis(analysis)`

1. Constructs a rich text document for the table and each column containing all learned metadata.

2. Calls `_get_embedding()` to convert these text documents into 768-dimensional vectors.

3. Upserts (Update/Insert) these vectors and metadata into ChromaDB.

`_analyze_column(...)`

Legacy helper retained for contingency debugging. The efficient SQL path is used by default.

`_get_embedding(text)`

Sends text to Ollama's `/api/embeddings` endpoint using `nomic-embed-text`. Returns a list of floats (the vector).

Data Structures (Pydantic Models)

The script uses Pydantic to enforce structured output from the LLM.

`*MetadataResponse*` (Table Level)

* `table_name`: Name of the table.
* `description`: LLM-generated description of content.
* `suggested_primary_key`: Best guess for ID column.
* `category`: e.g., "Sales", "HR", "Inventory".
* `business_context`: How this data fits into a business.
* `data_quality_notes`: Observations about nulls or formatting.

`*DataTypeResponse*` (Column Level)

* `sql_type`: Recommended SQL type (INTEGER, TEXT, etc.).
* `python_type`: Equivalent Python type.
* `description`: Description of the column.
* `business_meaning`: Semantic meaning (e.g., "Revenue in USD").
* `constraints`: e.g., "NOT NULL", "UNIQUE".

* `is_nullable`: Boolean.

* `suggested_index`: Boolean.

Output Artifacts

After running, the script creates a directory (default: `chroma_db_768dim`) containing:

* `chroma.sqlite3`: The vector database index.

* UUID folders: Binary vector data.

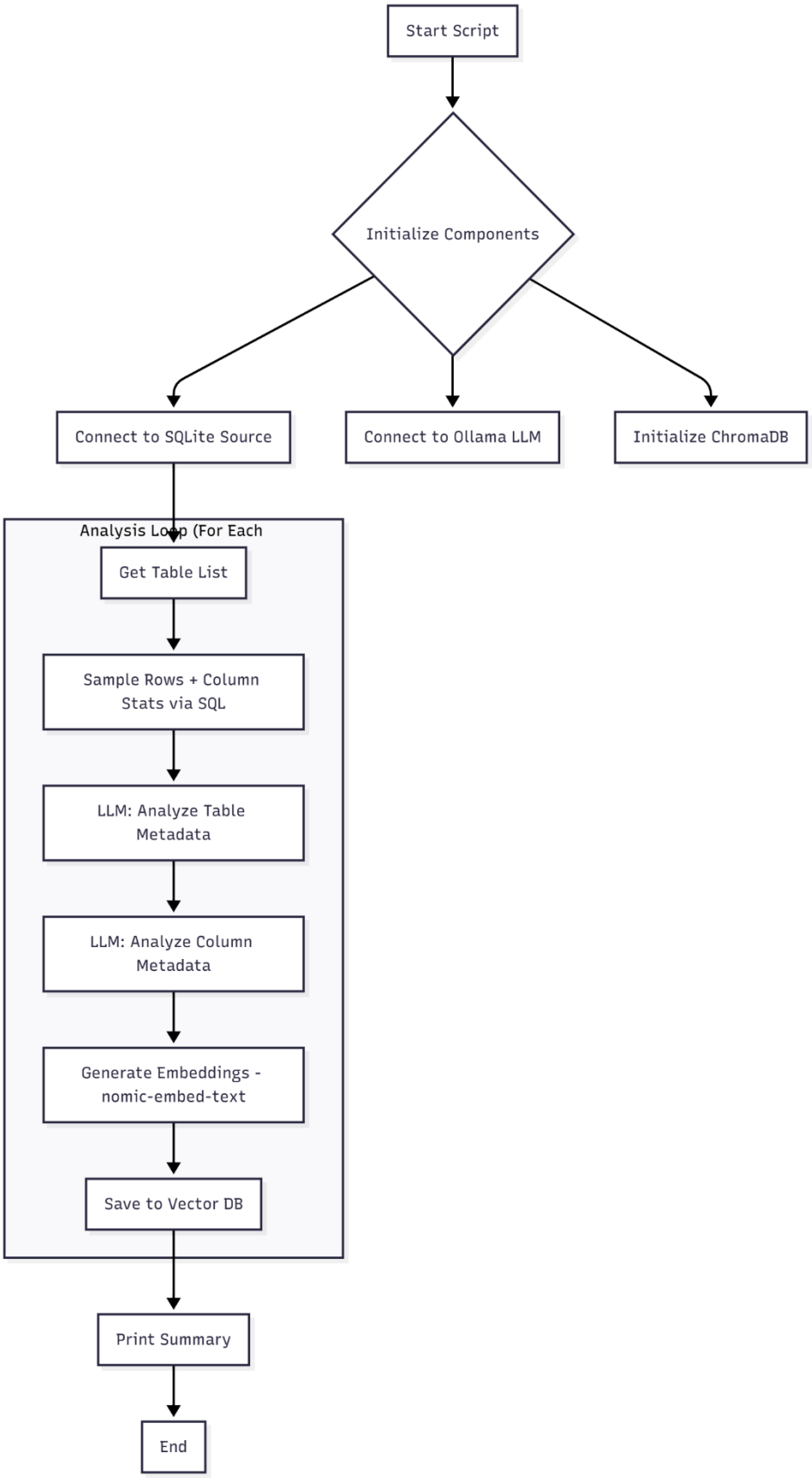
This directory is required by the main application (`app_gradio_enhanced.py`) to function.

Error Handling

* LLM Failures: If the LLM returns malformed JSON, `_parse_with_fallback` attempts to clean the string using Regex.

* Analysis Failures: If column analysis fails completely, it falls back to Pandas type detection so the process doesn't crash.

* Connection Errors: Logs errors if Ollama or SQLite cannot be reached.


Start Script
Initialize Components
Connect to SQLite Source
Connect to Ollama LLM
Initialize ChromaDB
Analysis Loop (For Each
Get Table List
Sample Rows + Column Stats via SQL
LLM: Analyze Table Metadata
LLM: Analyze Column Metadata
Generate Embeddings - nomic-embed-text
Save to Vector DB
Print Summary
End

Using ***view_metadata.py*** we can see the content of created vector database.

Table Metadata

```
============================================================
TABLE METADATA:
============================================================

[Table 1]
ID: table_ecom
Document:
Table: ecom
Description: The ecom table contains detailed information about each order placed on an e-commerce platform, including customer details, product information, payment methods, and delivery
times. This data is crucial for sales analysis, financial reporting, and understanding customer behavior.
Category: sales
Business Context: This dataset is used to analyze sales performance, customer preferences, payment method effectiveness, and overall profitability of the e-commerce platform. It helps in m
aking data-driven decisions for inventory management, marketing strategies, and financial planning.
Primary Key: order_id
Data Quality: The 'discount' column has a value (0.15) that seems unusually high as a discount rate; it might be better to check if this is an error or represents some special promotion., 
The 'profit_margin' column contains negative values, which could indicate errors in the calculation or data entry issues., The 'customer_age' and 'customer_gender' columns have valid age r
anges but may benefit from additional validation checks for consistency.
Row Count: 34500
Column Count: 17

Metadata: {
  "business_context": "This dataset is used to analyze sales performance, customer preferences, payment method effectiveness, and overall profitability of the e-commerce platform. It helps
 in making data-driven decisions for inventory management, marketing strategies, and financial planning.",
  "row_count": 34500,
  "column_count": 17,
  "description": "The ecom table contains detailed information about each order placed on an e-commerce platform, including customer details, product information, payment methods, and deli
very times. This data is crucial for sales analysis, financial reporting, and understanding customer behavior.",
  "category": "sales",
  "table_name": "ecom",
  "data_quality_notes": "[\"The 'discount' column has a value (0.15) that seems unusually high as a discount rate; it might be better to check if this is an error or represents some specia
l promotion.\", \"The 'profit_margin' column contains negative values, which could indicate errors in the calculation or data entry issues.\", \"The 'customer_age' and 'customer_gender' co
lumns have valid age ranges but may benefit from additional validation checks for consistency.\"]",
  "suggested_primary_key": "order_id"
}
```

Column Metadata

```
============================================================
COLUMN METADATA:
============================================================

[Column 1]
ID: column_ecom_order_id
Document:
Table: ecom
Column: order_id
Type: TEXT (str)
Description: A unique identifier for each order in the ecom table.
Business Meaning: This column contains a unique alphanumeric code assigned to each order, which helps in tracking individual transactions and managing order data.
Constraints: UNIQUE, NOT NULL
Nullable: False
Unique Values: 34500
Null Count: 0

Metadata: {
  "is_nullable": "False",
  "sql_type": "TEXT",
  "description": "A unique identifier for each order in the ecom table.",
  "constraints": "[\"UNIQUE\", \"NOT NULL\"]",
  "column_name": "order_id",
  "unique_count": 34500,
  "null_count": 0,
  "business_meaning": "This column contains a unique alphanumeric code assigned to each order, which helps in tracking individual transactions and managing order data.",
  "python_type": "str",
  "suggested_index": "True",
  "table_name": "ecom"
}

[Column 2]
ID: column_ecom_customer_id
Document:
Table: ecom
Column: customer_id
Type: TEXT (str)
Description: A unique identifier for each customer in the ecom table.
Business Meaning: This column contains a unique identifier assigned to each customer, likely used to track and manage individual user interactions with the e-commerce platform.
Constraints: UNIQUE, NOT NULL
Nullable: False
Unique Values: 7903
Null Count: 0

Metadata: {
  "constraints": "[\"UNIQUE\", \"NOT NULL\"]",
  "description": "A unique identifier for each customer in the ecom table.",
  "table_name": "ecom",
  "suggested_index": "True",
  "is_nullable": "False",
  "column_name": "customer_id",
  "sql_type": "TEXT",
  "unique_count": 7903,
```

Database Schema

metadata.db

```
├── table_metadata
│   ├── table_name (PK)
│   ├── description
│   ├── category
│   ├── business_context
│   ├── suggested_primary_key
│   ├── data_quality_notes (JSON)
│   ├── row_count
│   ├── column_count
│   └── analyzed_at
│
└── column_metadata
    ├── id (PK)
    ├── table_name (FK)
    ├── column_name
    ├── sql_type
    ├── python_type
    ├── description
    ├── business_meaning
    ├── constraints (JSON)
    ├── is_nullable
    ├── suggested_index
    ├── unique_count
    ├── null_count
    └── analyzed_at
```

##

Configuration and Parameters

##

1. QueryAgentEnhanced Configuration

Parameter	Type	Default	Description
source_db_path	str	Required	Path to SQLite database
vector_db_path	str	./chroma_db_768dim	Path to ChromaDB directory
llm_model	str	qwen2.5:7b	Ollama model name
conversation_state	ConversationState	None	Existing state or creates new
max_context_messages	int	10	Max messages in LLM context
max_data_contexts	int	20	Max data contexts to retain
temperature	float	0.1	LLM temperature (0.0-1.0)
ollama_base_url	str	http://localhost:11434	Ollama server URL
embedding_model	str	nomic-embed-text	Embedding model (768-dim)

Temperature Impact:

- `0.0`: Deterministic, same output for same input

- `0.1`: Slightly varied, good for SQL generation (current)

- `0.5`: Moderately creative

- `1.0`: Maximum creativity/randomness

2. Ollama LLM Configuration

```*python*

*self.llm = ChatOllama(*

*model=llm_model,*

*base_url=ollama_base_url,*

*temperature=temperature,*

*num_ctx=4096, # Context window size (tokens)*
```

num_predict=1024, # Max tokens to generate

repeat_penalty=1.1, # Penalize repetition (>1.0 reduces repetition)

timeout=120 # Request timeout in seconds)

```

num_ctx (Context Window):

- 4096 tokens = ~16,000 characters

- Includes: system prompt + conversation history + question

- If exceeded, oldest messages dropped

- qwen2.5:7b supports up to 128K, but 4096 is optimal for speed

num_predict:

- Maximum tokens in response

- SQL queries typically <200 tokens

- Answers typically 100-500 tokens

- 1024 provides buffer for complex responses
```

##

System Deployment

(Query Agent + Conversation Manager + Gradio UI)

##

Purpose

This system is a conversational data analytics platform that enables natural language querying of SQLite databases with:

- Multi-turn conversation support with full context retention

- Intelligent SQL generation using LLM (Large Language Models)

- Automatic visualization recommendation and generation

- Vector database-powered metadata retrieval for semantic search

- Persistent conversation history with ability to restore sessions- Interactive web interface using Gradio

Key Capabilities

1. Natural Language to SQL: Convert user questions into executable SQL queries

2. Contextual Awareness: Remember previous queries and reference them in follow-up questions

3. Smart Visualization: Automatically detect when to visualize and choose appropriate chart type

4. Data Transformation: Filter, sort, and manipulate previous results without re-querying

5. Conversation Persistence: Save and restore complete conversation sessions

6. Metadata-Driven Intelligence: Use LLM-analyzed metadata for better SQL generation

Technology Stack

- **Language**: Python 3.10+

- **LLM Framework**: LangChain + Ollama (qwen2.5:7b model)

- **Vector Database**: ChromaDB (with nomic-embed-text embeddings, 768-dim)

- **SQL Database**: SQLite

- **Visualization**: Plotly (interactive charts)

- **Web Interface**: Gradio - **Data Processing**: Pandas

1. Prepare the Databases

Option A: Use Existing SQLite Database

Copy and paste the existing SQLite database and Vector Database (metadata) in to project folder.

```
Ollama /
├── app_gradio_enhanced.py          # Main Gradio UI application
├── QueryAgent_Ollama_Enhanced.py   # Core query processing logic
├── conversation_manager.py         # Conversation state management
├── requirements.txt                # Python dependencies
│
├── PRE PROCESSING/
│   ├── analyze_existing_db.py      # Database analyzer
│   ├── csv_to_db.py                # CSV to SQLite converter
│
├── conversations/                  # Saved conversation files
│   ├── <uuid>.json
│
├── chroma_db_768dim/               # Vector database
│   ├── chroma.sqlite3
│   ├── <collection-id>/
│
└── analysis.db                     # User's SQLite database
```

Option B: From existing database or by creating SQLite database, create vector database as previously mentioned.

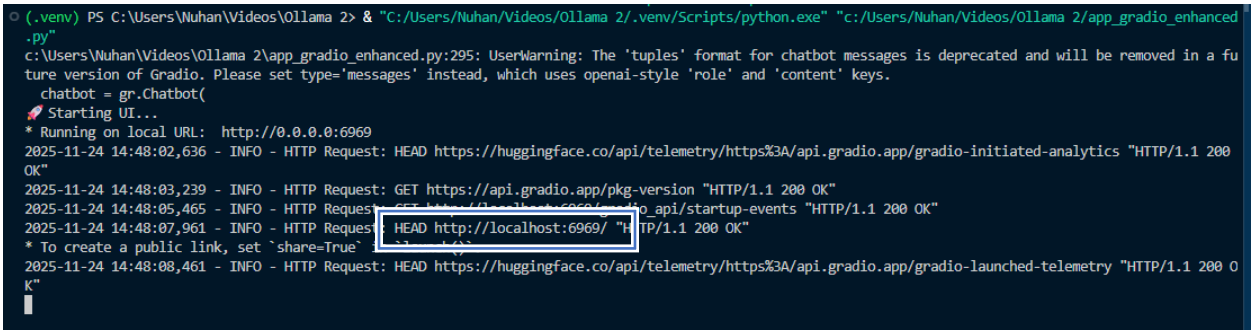
2. Launch Application

```bash

python app_gradio_enhanced.py

```

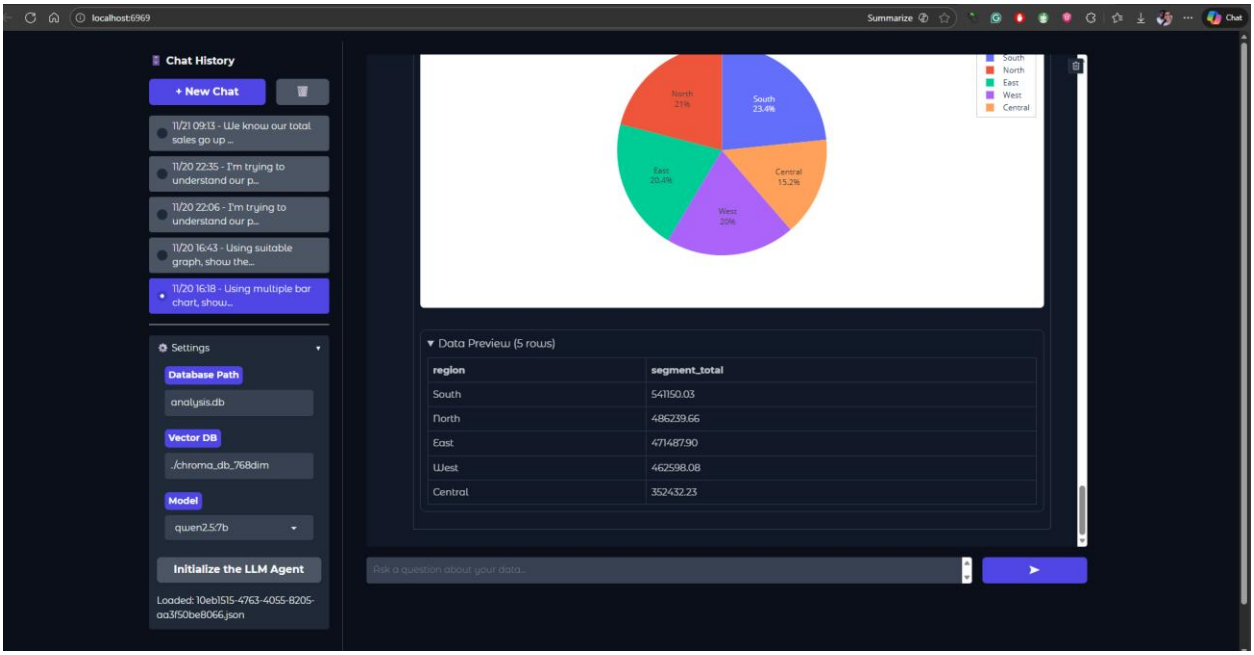
app_gradio_enhanced.py is the script created to run the UI. Gradio Library is a library which can create simple UI.

To start the system you just need to run the app_gradio_enhanced script and then select the link of the local host which is `http://localhost:6969`

You can change this later.



After clicking the link, the system will run on your browser as below.



3. Initialize Agent in UI

If database files are in project folder, just Click the Initialize the LLM Agent button. Otherwise do the following order.

1. Enter database path: `analysis.db`
2. Enter vector DB path: `./chroma_db_768dim`
3. Select model: `qwen2.5:7b`
4. Click "Initialize the LLM Agent"
5. Wait for "☑ Agent Ready"

END-TO-END USER GUIDE - GETTING FINAL ANSWERS

Step 1: Start a New Session

1. Open the web interface at `http://localhost:6969`.

2. Click **"+ New Chat"** in the sidebar.

3. Ensure the agent is initialized (check "☑ Agent Ready").

Step 2: Ask Your First Question

Goal: Get a high-level overview.

Input: "What tables do we have in the database?"

Expected Output: A list of tables with descriptions (e.g., Sales, Products, Customers).

Step 3: Drill Down into Data

Goal: Analyze specific metrics.

Input: "Show me the total sales by product category for 2024."

Expected Output:

- A text summary of sales figures.

- A data table showing categories and amounts.

- (Optional) A bar chart if the system auto-detects the need.

Step 4: Visualize the Results

Goal: Create a chart if one wasn't generated automatically.

Input: "Visualize this as a pie chart."

Expected Output: An interactive pie chart showing the distribution of sales by category.

Step 5: Refine the Analysis

Goal: Filter or modify the data.

Input: "Filter for only Electronics and Home categories."

Expected Output: Updated text, table, and chart reflecting only the selected categories.

Step 6: Save and Share

1. The conversation is automatically saved.

2. You can return to it later via the sidebar history.

3. To share, you can export the conversation JSON file from the `conversations/` folder.

##

Query Agent Enhanced Class

##

Pydantic Models

QuestionIntent (Enum)

```python
class QuestionIntent(str, Enum):
    NEW_QUERY = "new_query"       # Fresh database query
    RE_VISUALIZE = "re_visualize"  # New chart for same data
    TRANSFORM = "transform"       # Modify existing data
    COMBINE = "combine"           # Merge multiple datasets
    COMPARE = "compare"           # Side-by-side comparison
    CLARIFY = "clarify"           # Explain previous result
```

IntentAnalysis

```python
class IntentAnalysis(BaseModel):
    intent: str                   # One of QuestionIntent values
    references_previous: bool     # True if mentions "that", "those", etc.
    referenced_concepts: List[str]  # ["sales", "categories", "profit"]
    needs_context: bool           # True if requires conversation history
    confidence: float             # 0.0-1.0 confidence score
```

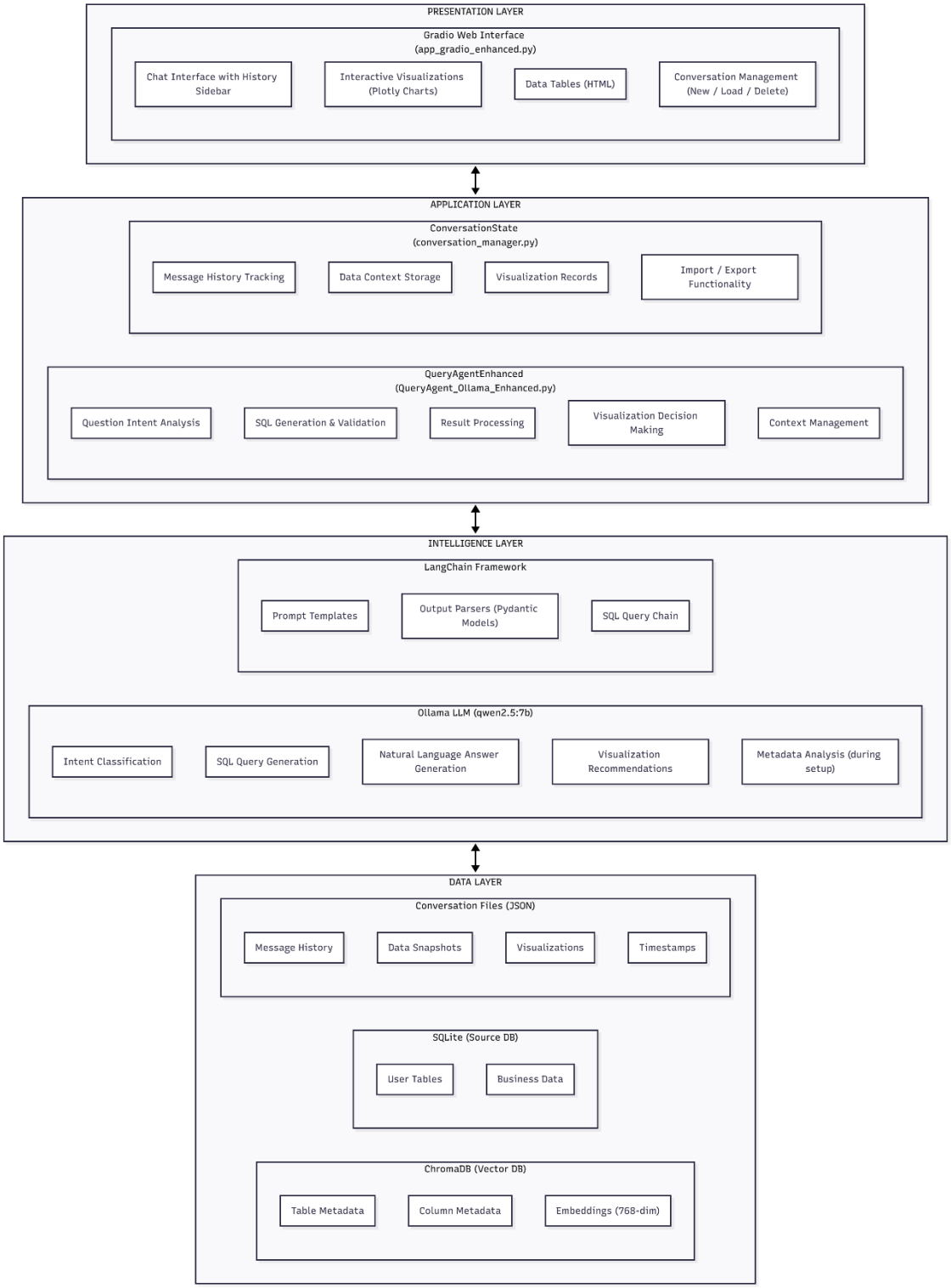
VisualizationResponse

```python
class VisualizationResponse(BaseModel):
    should_visualize: bool
    chart_types: List[str]  # ["bar", "line"] - alternatives
    primary_chart: str      # "bar" - recommended type
    x_axis: Optional[str]   # "category"
    y_axis: Optional[str]   # "sales_amount"
    color_by: Optional[str] # "region"
    title: str              # "Sales by Category"
    visualization_rationale: str  # Why this viz is appropriate
```

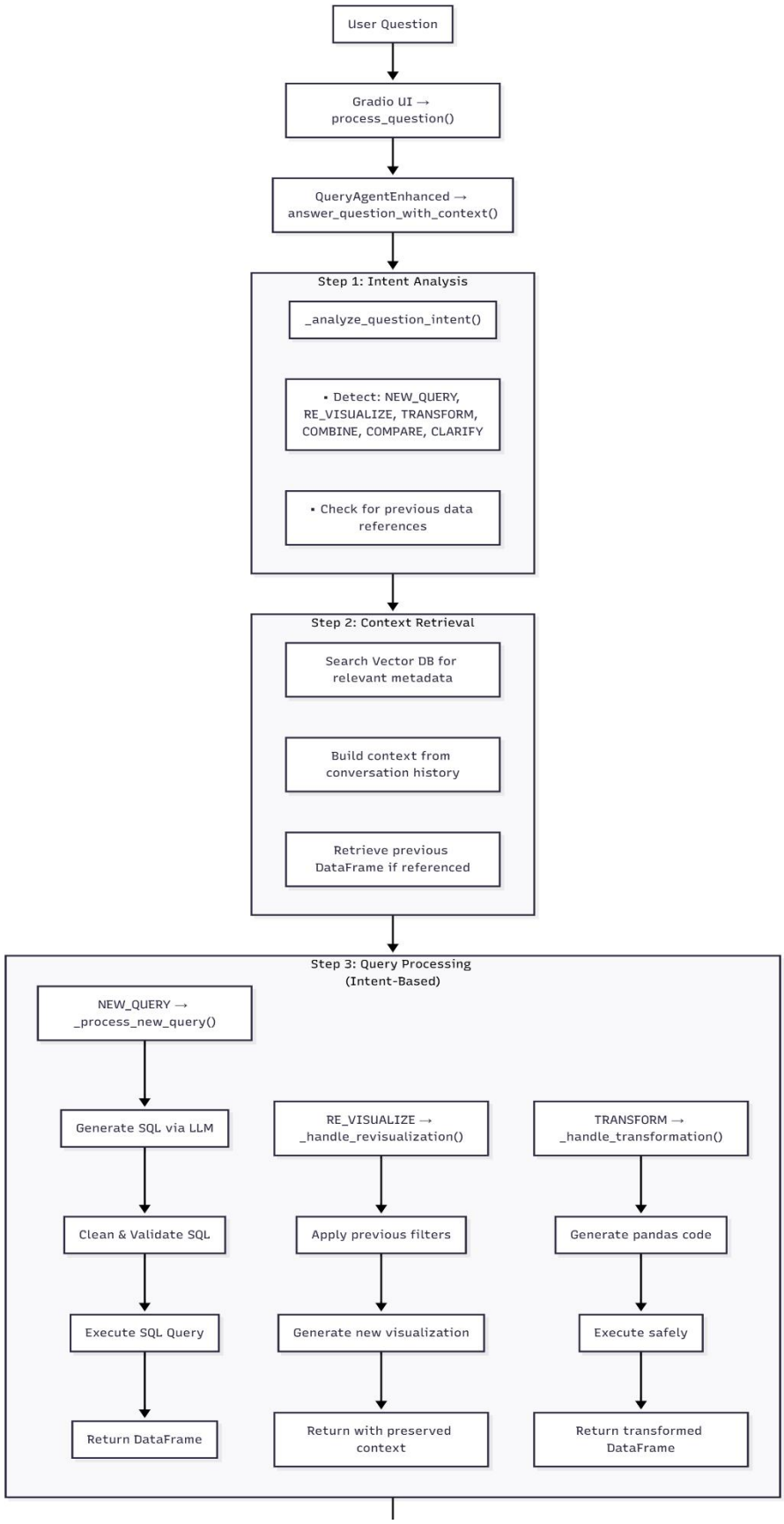
DataTypeResponse

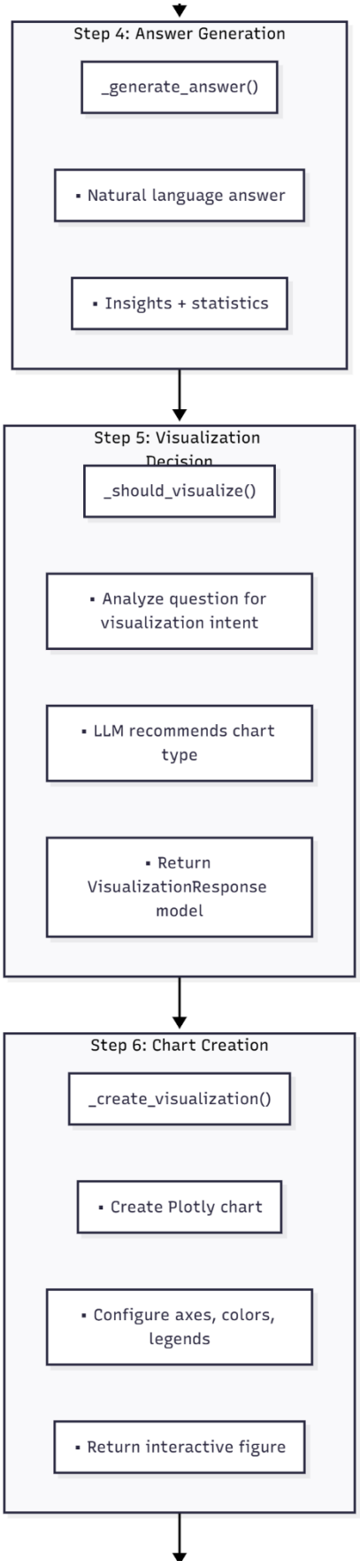
```python
class DataTypeResponse(BaseModel):
    sql_type: str          # TEXT, INTEGER, REAL, DATE, BOOLEAN
    python_type: str       # str, int, float, datetime, bool
    description: str
    business_meaning: str
    constraints: List[str]  # ["NOT NULL", "UNIQUE"]
    is_nullable: bool
    suggested_index: bool
```

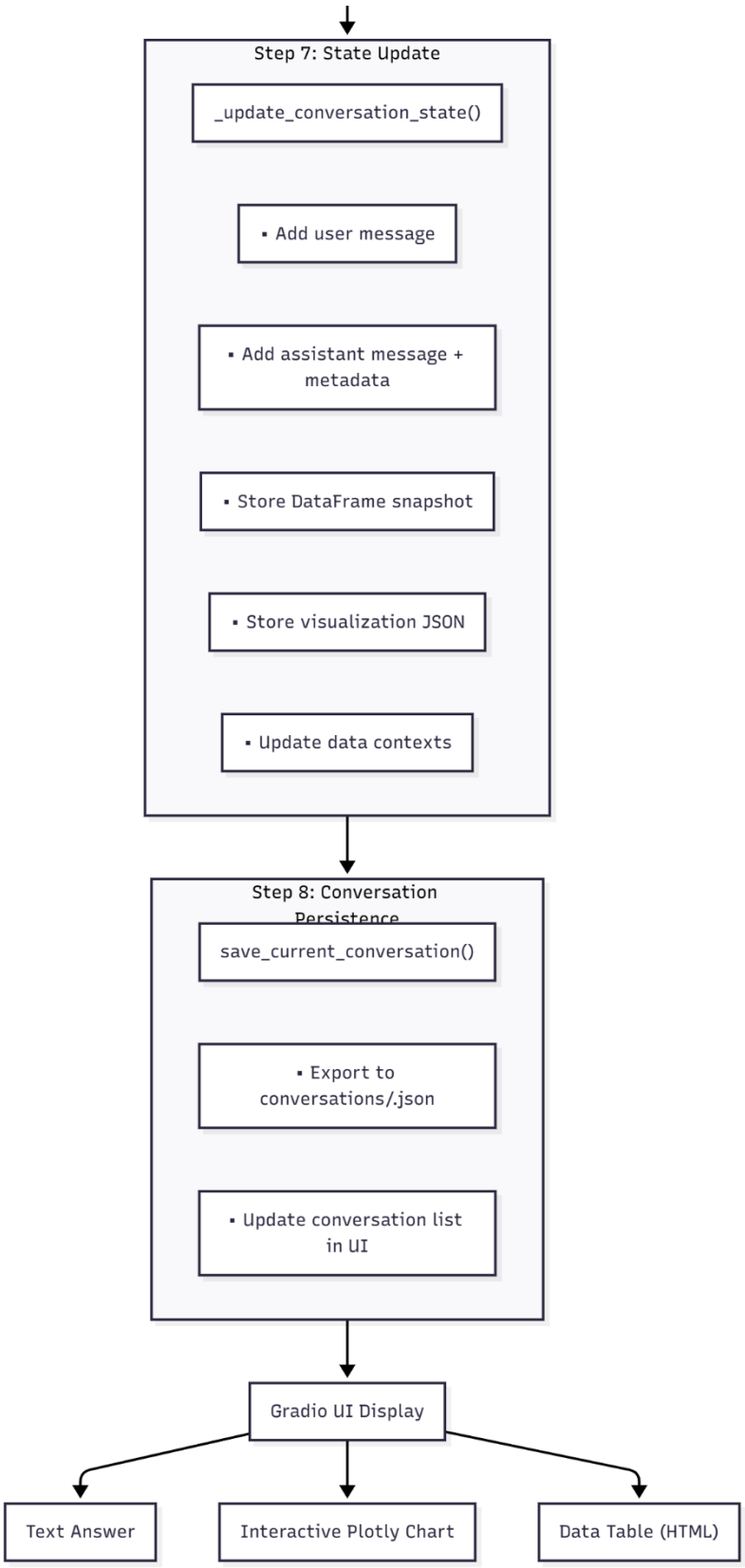
1. System Layers



2. Component Interaction Flow




Step 4: Answer Generation
_generate_answer()
▪ Natural language answer
▪ Insights + statistics
Step 5: Visualization Decision
_should_visualize()
▪ Analyze question for visualization intent
▪ LLM recommends chart type
▪ Return VisualizationResponse model
Step 6: Chart Creation
_create_visualization()
▪ Create Plotly chart
▪ Configure axes, colors, legends
▪ Return interactive figure


Step 7: State Update
_update_conversation_state()
▪ Add user message
▪ Add assistant message + metadata
▪ Store DataFrame snapshot
▪ Store visualization JSON
▪ Update data contexts
Step 8: Conversation Persistence
save_current_conversation()
▪ Export to conversations/.json
▪ Update conversation list in UI
Gradio UI Display
Text Answer
Interactive Plotly Chart
Data Table (HTML)

3. Core Components Explanation

Constructor Parameters

```python
QueryAgentEnhanced(
    source_db_path: str,                          # Path to SQLite database
    vector_db_path: str = "./chroma_db_768dim",   # Path to ChromaDB
    llm_model: str = "qwen2.5:7b",                # Ollama model name
    conversation_state: Optional[ConversationState] = None,
    max_context_messages: int = 10,               # Max messages to keep in context
    max_data_contexts: int = 20,                  # Max data contexts to retain
    temperature: float = 0.1,                     # LLM temperature (0.0-1.0)
    ollama_base_url: str = "http://localhost:11434",
    embedding_model: str = "nomic-embed-text"     # 768-dim embeddings
)
```

Database Connection

```python
self.conn = sqlite3.connect(source_db_path, check_same_thread=False)
self.db = SQLDatabase.from_uri(f"sqlite:///{source_db_path}")
```

- Creates persistent SQLite connection
- Initializes LangChain SQLDatabase wrapper
- Allows thread-safe operations

Vector Database Setup

```python
self._setup_vector_database()
```

- Connects to ChromaDB at specified path
- Retrieves `table_metadata` and `column_metadata` collections
- Loads all metadata into memory for fast access

LLM Initialization

```python
self.llm = ChatOllama(
    model=llm_model,
    base_url=ollama_base_url,
    temperature=temperature,
    num_ctx=4096,          # Context window size
    num_predict=1024,      # Max tokens to generate
    repeat_penalty=1.1,    # Prevent repetition
    timeout=120            # Request timeout
)
```

Chain Creation

```python
self.query_chain = create_sql_query_chain(self.llm, self.db)
```

- Creates LangChain SQL query generation chain
- Automatically includes schema information

Parser Setup

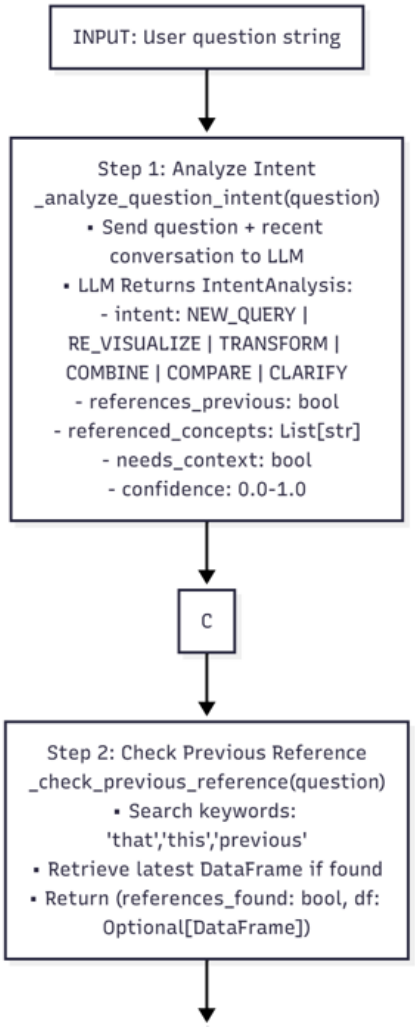
```python
self.viz_parser = PydanticOutputParser(pydantic_object=VisualizationResponse)
```

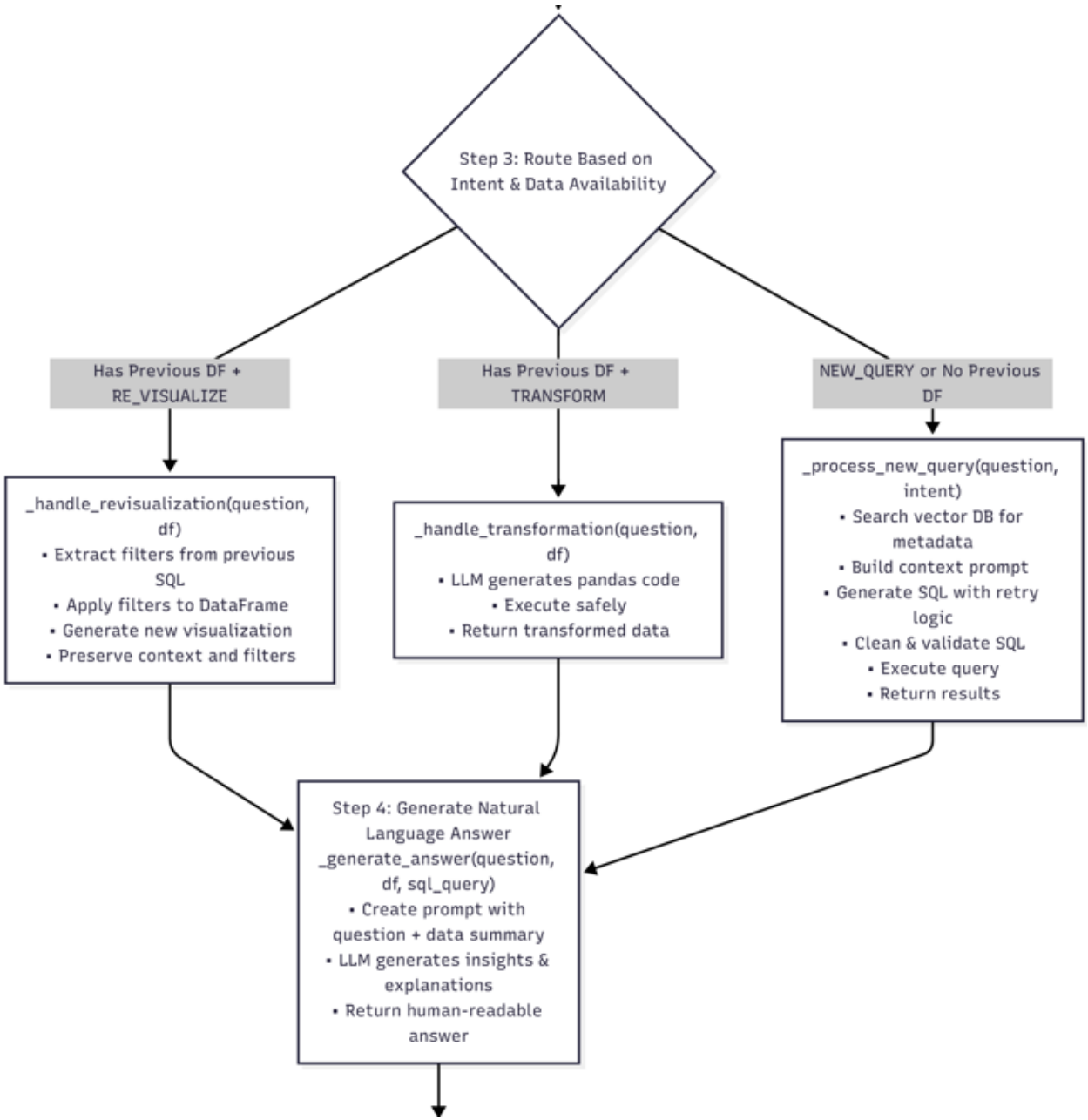
- Initializes structured output parsers
- Ensures LLM responses match expected format

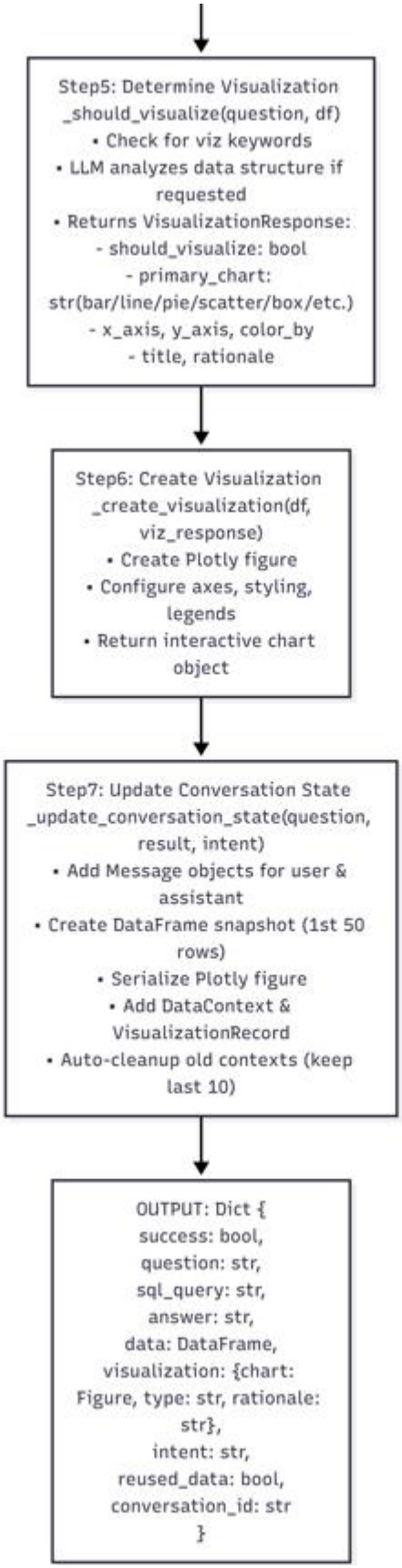
Context Awareness

`answer_question_with_context(question: str, reuse_data: bool = False) -> Dict[str, Any]`

Purpose: Main entry point for processing user questions with full context awareness.




Step 3: Route Based on Intent & Data Availability
Has Previous DF + RE_VISUALIZE
Has Previous DF + TRANSFORM
NEW_QUERY or No Previous DF
_handle_revisualization(question, df)
• Extract filters from previous SQL
• Apply filters to DataFrame
• Generate new visualization
• Preserve context and filters
_handle_transformation(question, df)
• LLM generates pandas code
• Execute safely
• Return transformed data
_process_new_query(question, intent)
• Search vector DB for metadata
• Build context prompt
• Generate SQL with retry logic
• Clean & validate SQL
• Execute query
• Return results
Step 4: Generate Natural Language Answer
_generate_answer(question, df, sql_query)
• Create prompt with question + data summary
• LLM generates insights & explanations
• Return human-readable answer


Step5: Determine Visualization
_should_visualize(question, df)
• Check for viz keywords
• LLM analyzes data structure if requested
• Returns VisualizationResponse:
- should_visualize: bool
- primary_chart: str(bar/line/pie/scatter/box/etc.)
- x_axis, y_axis, color_by
- title, rationale
Step6: Create Visualization
_create_visualization(df, viz_response)
• Create Plotly figure
• Configure axes, styling, legends
• Return interactive chart object
Step7: Update Conversation State
_update_conversation_state(question, result, intent)
• Add Message objects for user & assistant
• Create DataFrame snapshot (1st 50 rows)
• Serialize Plotly figure
• Add DataContext & VisualizationRecord
• Auto-cleanup old contexts (keep last 10)
OUTPUT: Dict {
success: bool,
question: str,
sql_query: str,
answer: str,
data: DataFrame,
visualization: {chart: Figure, type: str, rationale: str},
intent: str,
reused_data: bool,
conversation_id: str
}

`_analyze_question_intent(question: str) -> IntentAnalysis`

Purpose: Classify user's question to determine processing strategy.

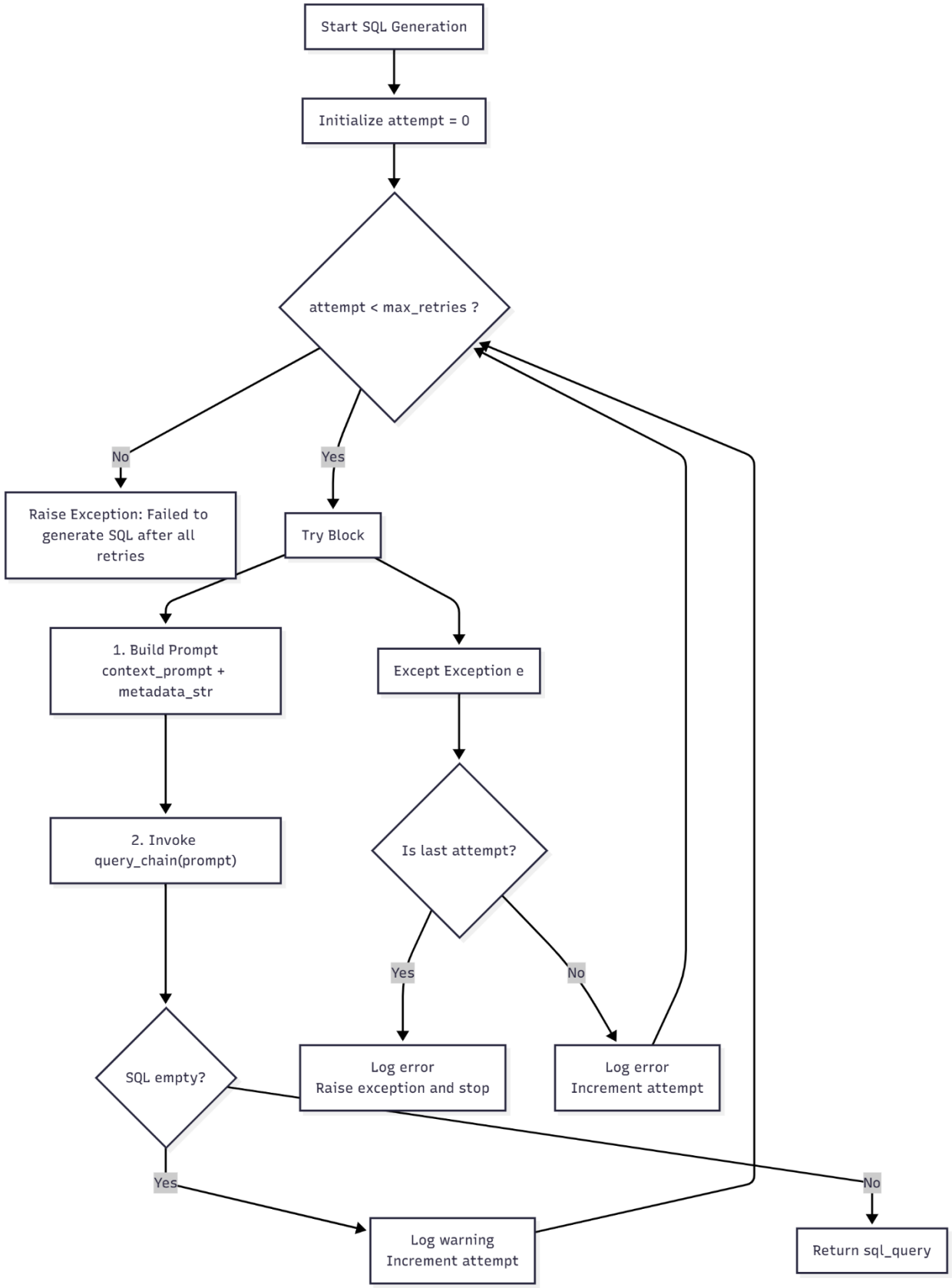
Output Example

```python
IntentAnalysis(
   intent="re_visualize",
   references_previous=True,
   referenced_concepts=["profit margin", "categories"],
   needs_context=True,
   confidence=0.9)
```

`_generate_sql_with_retry(question: str, context_prompt: str, metadata_str: str, max_retries: int = 2) -> str`

Purpose: Generate SQL with automatic retry on failure.

Flowchart used for this function as below.


Start SQL Generation
Initialize attempt = 0
attempt < max_retries ?
No
Yes
Raise Exception: Failed to generate SQL after all retries
Try Block
1. Build Prompt context_prompt + metadata_str
2. Invoke query_chain(prompt)
SQL empty?
Yes
Log warning Increment attempt
No
Return sql_query
Except Exception e
Is last attempt?
Yes
Log error Raise exception and stop
No
Log error Increment attempt

SQL Query Cleaning `_clean_sql(query: str) -> str`

Purpose: Comprehensive SQL cleaning and sanitization.

Input: Raw LLM SQL output (may contain markdown, explanations)

Transformations:

1. Remove markdown fences: ` ```sql ... ``` `
2. Strip "SQLQuery:" prefixes
3. Extract last SQL statement marker
4. Remove numbered steps before SELECT
5. Remove explanatory prose
6. Extract SELECT/WITH keywords
7. Balance quotes
8. Collapse whitespace

Example:

Raw LLM SQL output (may contain markdown, explanations)

```python
# Input
raw_sql = """
    Here's the SQL query:
    ```sql
    SELECT * FROM sales
    WHERE amount > 100
    ORDER BY date DESC
    ```
    This query returns sales over $100.
"""
```

Output: Clean, executable SQL

*clean_sql = "SELECT * FROM sales WHERE amount > 100 ORDER BY date DESC"*

`_validate_and_fix_tables(sql_query: str) -> str`

Purpose: Fix invalid table/column references

Validation Steps:

1. Load valid tables from SQLite
2. Extract CTE (Common Table Expression) names
3. Build column-to-table mapping
4. Analyze query aliases
5. Detect invalid table references
6. Fix or remove invalid JOINs
7. Replace invalid column references
8. Clean up broken clauses

Example Fix:

```*sql*
*-- BEFORE (Invalid table "invalid_table")*
*SELECT s.region, SUM(invalid_table.amount) as total*
*FROM sales s*
*LEFT JOIN invalid_table ON s.id = invalid_table.sale_id*
*GROUP BY s.region*

*-- AFTER (Fixed)*
*SELECT s.region, SUM(s.amount) as total*
*FROM sales s*
*GROUP BY s.region*
```

1. Remove Markdown Fences

```python
q = re.sub(r"```\w*", "", q)
```

2. Extract Last SQL Marker

```python
marker_iter = list(re.finditer(r"(?i)(sqlquery|sql query|sql|query)\s*:", q))
```

3. Remove Explanatory Text

```python
q = re.sub(r"(?i)^.*?(?:here's?\s+(?:the\s+)?(?:sql\s+)?query[:\s]+)", "", q)
```

4. Extract SELECT/WITH

```python
cte_match = re.search(r"(?i)\bwith\s+[A-Za-z_][\w]*\s+as", q)
select_match = re.search(r"(?i)\bselect\b", q)
```

5. Balance Quotes

```python
single_quote_count = query.count("'")
```

6. Fix Common Syntax Issues

- Adjacent quoted strings: `AS "Total" "Sales"` → `AS "Total Sales"`

- Unquoted aliases with spaces
- Missing closing quotes in GROUP BY/ORDER BY

7. Collapse Whitespace

```python
q = re.sub(r"\s+", " ", q).strip()
```

`_validate_and_fix_tables(sql_query: str) -> str`

Purpose: Validate table/column references and fix invalid ones.

1. Get Valid Tables
   - Query SQLite master table
   - Extract CTE names from query
   - Combine into valid_tables set

2. Build Column Map

   FOR each table IN valid_tables:
      - Get columns via PRAGMA table_info
      - Map column_name.lower() → (table, original_column_name)

3. Analyze Query Aliases

   _analyze_query_aliases(sql_query)
   - Extract FROM/JOIN aliases
   - Track: valid_aliases, base_to_alias, alias_to_table

4. Find Invalid JOIN Tables
   - Scan for JOIN patterns
   - Identify tables not in valid_tables

- Mark as invalid_tables

5. Fix Invalid References

IF invalid_tables exist:

- Remove JOIN clauses with invalid tables
- Replace invalid_table.column with valid_table.column
- Fix aggregate functions: SUM(invalid.col) → SUM(valid.col)
- Cleanup broken clauses (empty WHERE, trailing commas)

6. Fix Window Functions

- Detect SUM(alias.column) OVER ()
- Verify alias and column exist in subquery outputs
- Replace with correct column references

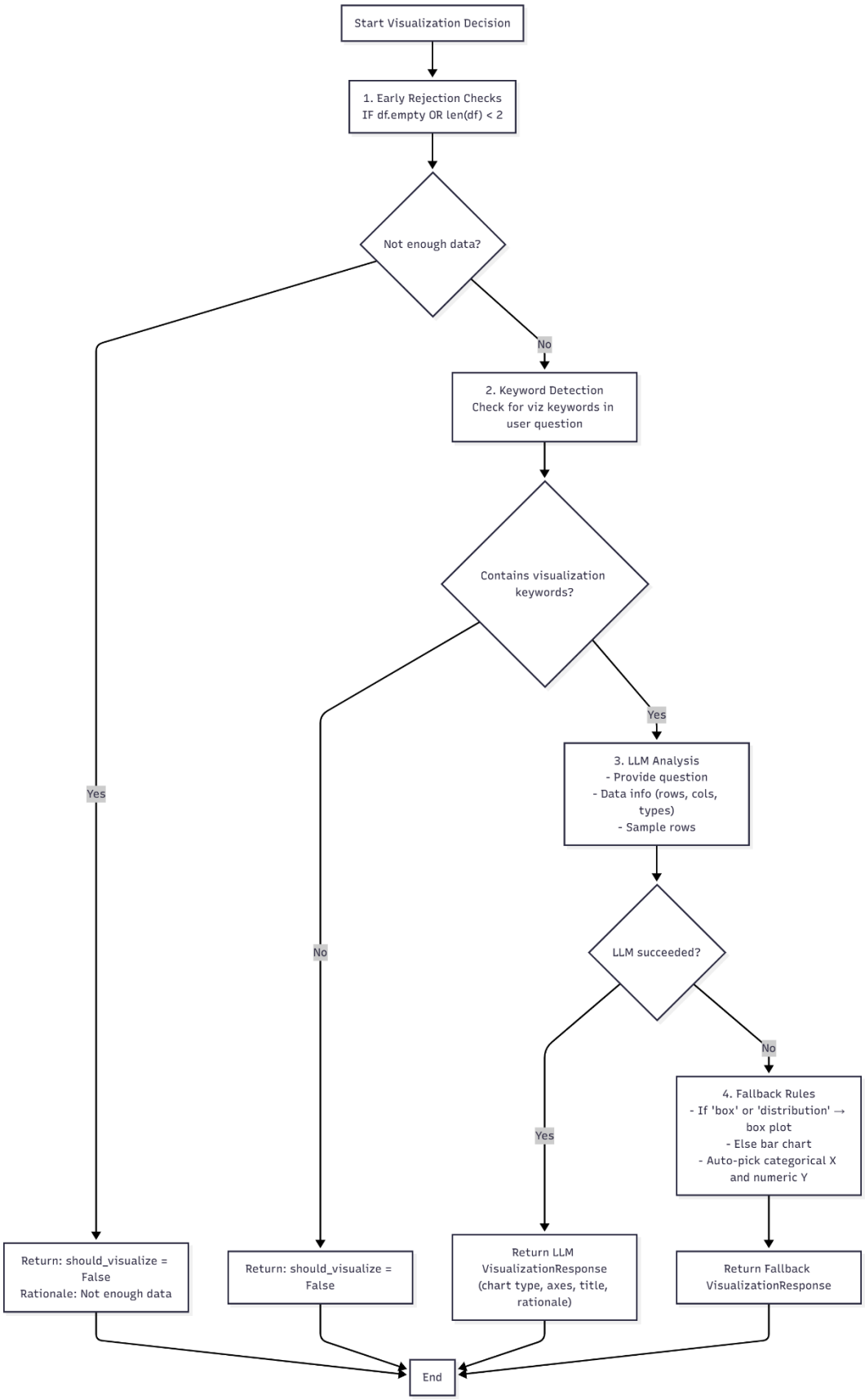
7. Replace Unknown Aliases

- Find all table.column references
- If table not in valid_aliases:
  - Look up in base_to_alias mapping
  - Replace with known alias

RETURN: Fixed SQL query

` _should_visualize(question: str, df: pd.DataFrame) -> VisualizationResponse `

Purpose: Determine if visualization is appropriate and recommend type.



`_create_visualization(df: pd.DataFrame, viz_response: VisualizationResponse) -> go.Figure`

Purpose: Create Plotly visualization based on recommendations.

Chart Types Supported

1. Bar Chart
2. Multiple Bar Chart (Grouped)
3. Line Chart
4. Pie Chart
5. Scatter Plot
6. Histogram
7. Box Plot

Chart type Selection

4. Bar Chart

   When: Comparing values across categories

   Indicators:

   - Question contains: "compare", "each", "by category"
   - Data has: 1 categorical + 1 numeric column
   - Row count: 2-100 optimal

   Example Question: "Show sales by region"

5. Line Chart

` When: Showing trends over time

   Indicators:

   - Question contains: "trend", "over time", "change"
   - Data has: date/time column + numeric column
   - Rows ordered by date

   Example Question: "How have sales changed over months?"

6. Pie Chart

When: Showing part-to-whole relationships

Indicators:

- Question contains: "percentage", "proportion", "share"
- Data has: categorical + 1 numeric (usually sum/count)
- Limited categories (<10 for readability)

Example Question: "What's the percentage of sales by category?"

Special Handling:

- Auto-limit to top 10 if more categories exist
- Explode largest slice (optional)

7. Box Plot

When: Analyzing distributions and outliers

Indicators:

- Question contains: "distribution", "range", "outliers"
- Data has: categorical grouping + numeric values
- Multiple values per category (≥4 for box)

Example Question: "Show profit margin distribution by category"

Requirements:

- Each category must have ≥4 data points
- Falls back to bar chart if insufficient data

8. Scatter Plot

When: Exploring relationships between two numeric variables

Indicators:

- Question contains: "relationship", "correlation"
- Data has: 2+ numeric columns

Example Question: "Is there a relationship between price and quantity?"

9. Histogram

When: Showing frequency distribution of a single variable

Indicators:

- Question contains: "frequency", "distribution"
- Data has: 1 numeric column

Example Question: "Show the distribution of order amounts"

##

Conversation State Class

##

1. Data Classes

Message

```python
@dataclass
class Message:
    role: str            # "user" or "assistant"
    content: str         # Message text
    timestamp: datetime    # When message was created
    sql_query: Optional[str] = None          # Generated SQL (assistant only)
    dataframe_snapshot: Optional[Dict] = None # Data metadata
    visualization: Optional[str] = None      # Chart type
    figure_json: Optional[str] = None        # Plotly figure as JSON
    metadata: Dict = field(default_factory=dict)
```

DataFrame Snapshot Structure

```python
    "columns": ["col1", "col2", "col3"],
    "row_count": 1000,
    "sample": {
        "col1": {"0": "val1", "1": "val2", ...},  # First 50 rows
        "col2": {"0": 10, "1": 20, ...},
        "col3": {"0": "A", "1": "B", ...}
```

Data Context

```python
@dataclass
class DataContext:
    query: str            # SQL query executed
    columns: List[str]    # Column names in result
    row_count: int        # Number of rows returned
    sample_data: Dict     # First 5 rows as dict
    timestamp: datetime   # When context was created
```

Visualization Record

```python
@dataclass
class VisualizationRecord:
    question: str         # User's question
    chart_type: str       # Type of chart created
    data_summary: str     # Brief description of data
    timestamp: datetime   # When viz was created
```

2. Core Methods

`add_message(message: Message)`

Purpose: Add a message to conversation history.

Usage Pattern:

```python
# User message
conversation_state.add_message(Message(
    role="user",
    content="What are the top 5 sales?",
    metadata={"intent": "new_query"}
))

# Assistant message
conversation_state.add_message(Message(
    role="assistant",
    content="Here are the top 5 sales...",
    sql_query="SELECT * FROM sales ORDER BY amount DESC LIMIT 5",
    dataframe_snapshot={
        "columns": ["id", "amount", "date"],
        "row_count": 5,
        "sample": {...}
    },
    visualization="bar",
    figure_json='{"data": [...], "layout": {...}}',
    metadata={"intent": "new_query", "success": True}
))
```

`add_data_context(context: DataContext)`

Purpose: Store query result metadata.

```python
    # Keep only last 10 contexts to manage memory
    if len(self.data_contexts) > 10:
        self.data_contexts = self.data_contexts[-10:]
```

Auto-Cleanup: Automatically limits to last 10 contexts to prevent memory bloat.

`get_recent_messages(n: int = 5) -> List[Message]`

Purpose: Retrieve last N messages for context building.

```python
def get_recent_messages(self, n: int = 5) -> List[Message]:
    return self.messages[-n:] if len(self.messages) >= n else self.messages
```

Usage: Used by `_analyze_question_intent()` and `_build_context_prompt()` to provide conversation history to LLM.

`export_conversation() -> Dict`

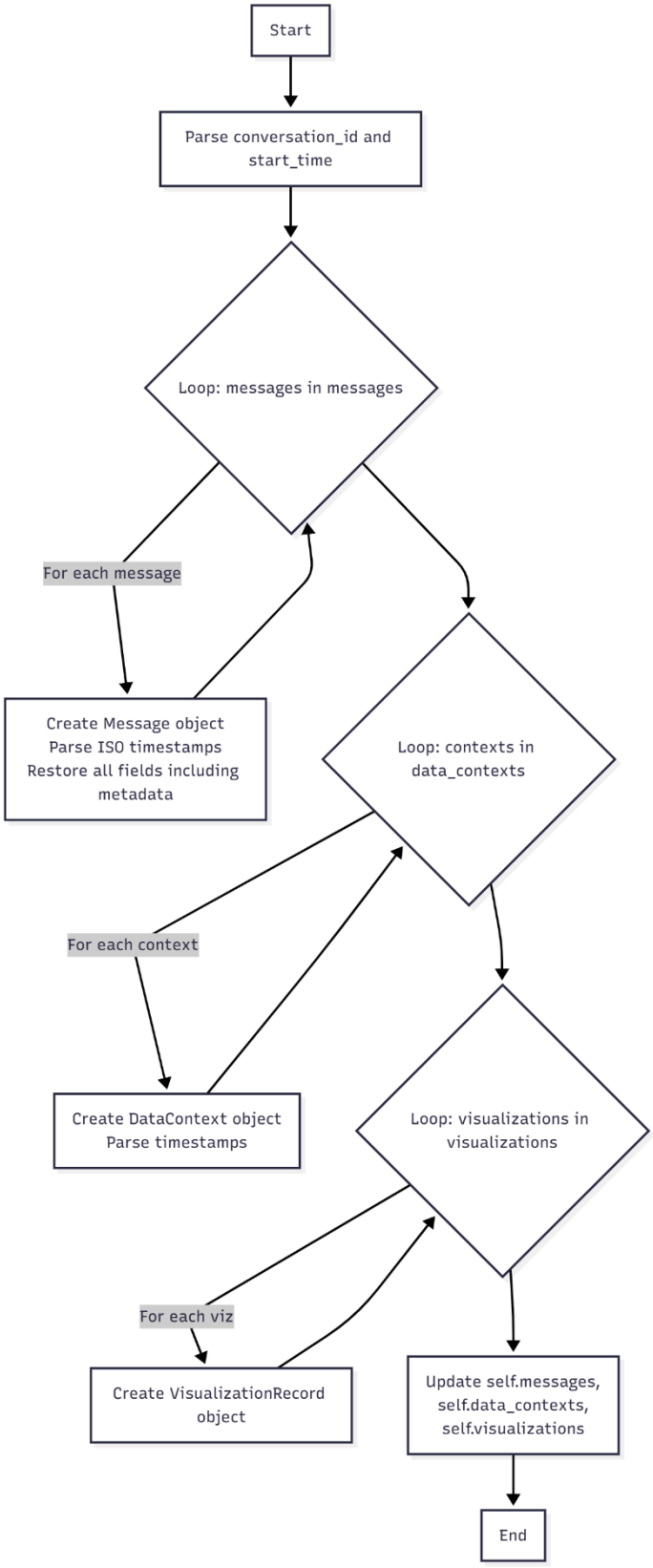
Purpose: Serialize conversation state for persistence.

```python
def export_conversation(self) -> Dict:
    return {
        "conversation_id": self.conversation_id,
        "start_time": self.start_time.isoformat(),
        "message_count": len(self.messages),
```

`import_conversation(data: Dict)`

Purpose: Restore conversation state from JSON.

These two classes are created for further implementations.



##

Gradio UI

##

Simple Web interface for user interaction

Global State Management

```python
# Global variables (module-level)
agent: Optional[QueryAgentEnhanced] = None
conversation_state: Optional[ConversationState] = None
CONVERSATIONS_DIR = "conversations"
```

Why Global: Gradio functions are stateless; global state persists across invocations.

1. Key Functions

`initialize_agent(db_path: str, vector_db: str, model: str) -> Tuple[str, gr.update]`

Purpose: Initialize the QueryAgentEnhanced with user-specified parameters.

Steps:

1. Validate database file exists
2. Create new Conversation State
3. Initialize Query Agent Enhanced with parameters
4. Load conversation list from disk
5. Return status message and updated history dropdown

Returns:

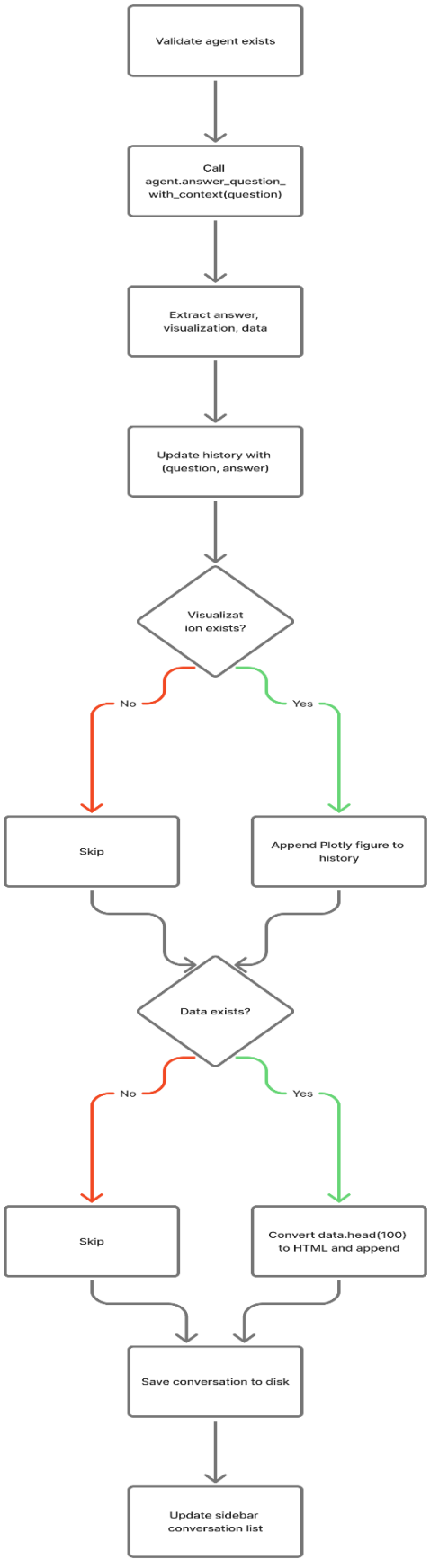
- Status message: "☑ Agent Ready" or error
- Gradio update for history dropdown

`process_question(question: str, history: List) -> Tuple[List, gr.update]`

Purpose: Process user question and update chat history.

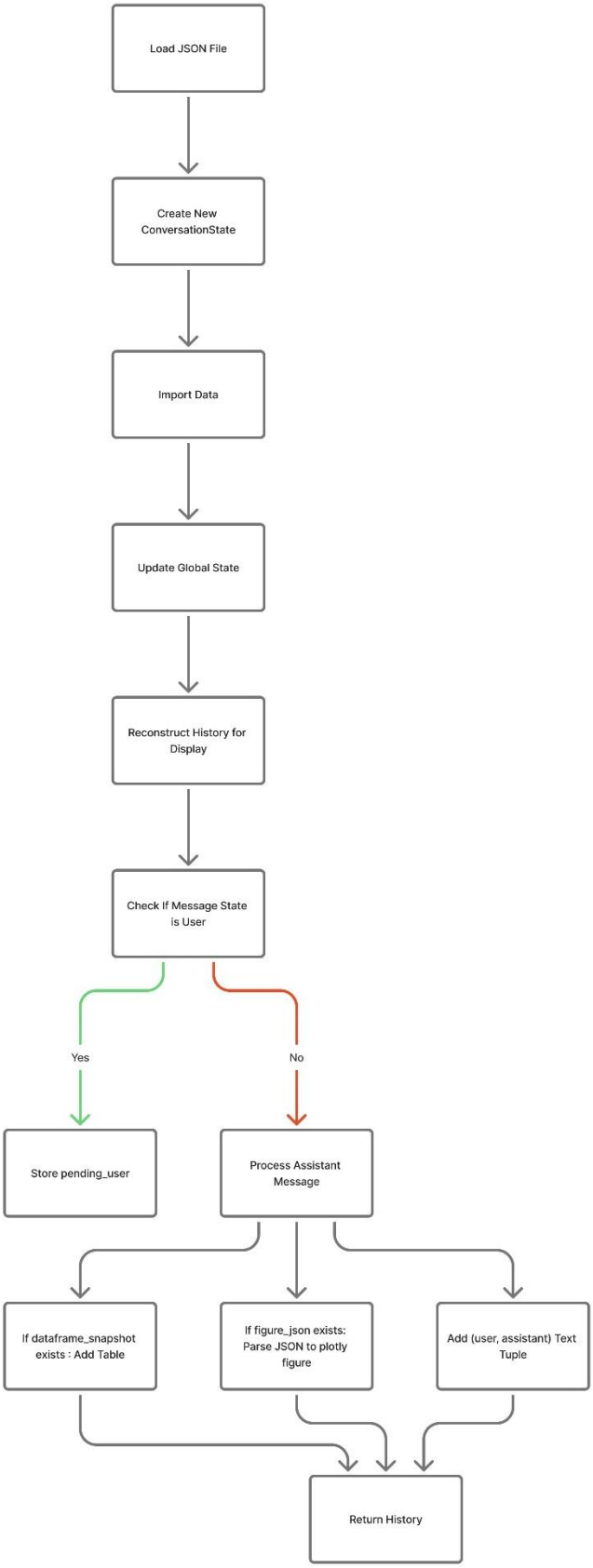
History Format:

```python
("User question 1", "Assistant answer 1"),
(None, gr.Plot(...)),                    # Chart
(None, "<div>HTML Table</div>"),         # Data table
("User question 2", "Assistant answer 2"),
...
```



`load_conversation(filename: str) -> Tuple[List, str]`

Purpose: Load and reconstruct a saved conversation.



- Plotly figures stored as JSON strings are deserialized

- DataFrames reconstructed from sample data (only first 50 rows saved)

- Order is preserved: user → assistant → chart → table

`save_current_conversation()`

Purpose: Persist current conversation to disk.

Trigger Points:

- After every successful question processing

- Automatic (no user action required)

`get_conversation_list() -> List[Tuple[str, str]]`

Purpose: Get list of saved conversations for sidebar dropdown.

Output Example:

```python

[

    ("11/20 22:50 - I'm trying to understand ou...", "2e6a2e96-1a3f-43f8-95cb-2a1debc1135a.json"),

    ("11/19 15:30 - Show me top 5 sales...", "df8d33d4-744a-4f66-8330-d03ce7f17844.json"),

]

```

2. UI Layout

.sidebar — Provides a padded, full-height vertical panel with a right border and vertical scrolling for sidebar content.

.chat-area — Adds padding and makes the main chat column full-height with its own vertical scroll.

.data-table — Ensures the table fills available width, collapses borders, and uses a slightly smaller font for compact display.

.data-table th — Styles table headers with a secondary background, padding, left-aligned text, and a pronounced bottom border.

.data-table td — Gives table cells consistent padding and a subtle bottom border for row separation.

.table-container — Wraps tables in a horizontally scrollable box with a border, rounded corners, and inner padding.

##

Data Flow & Workflows

##

1. First-Time Setup (Database Analysis)

Script: *`analyze_existing_db.py`*

Purpose: One-time analysis to create vector database with metadata

Step 01: User Runs Analyzer

$ python analyze_existing_db.py analysis.db

Step 2: Initialize Connections

• Connect to source SQLite database

• Initialize Ollama LLM (qwen2.5:7b)

• Create/connect to ChromaDB

• Setup prompt templates

Step 3: Discover Tables

get_table_names()

• Query sqlite_master for table names

• Exclude system tables (sqlite_%)

• Return list: ["sales", "products", "customers"]

Step 4: For Each Table - Analyze Metadata

_analyze_table_metadata(table_name, df)

LLM Input:

- Table Name
- Column list
- First 5 rows sample
- Row Count

LLM Output:


```
{
  "table_name": "sales",
  "description": "Transaction records for product sales with timestamps and amounts",
  "category": "financial",
  "business_context": "Core revenue tracking system",
  "suggested_primary_key": "transaction_id",
  "data_quality_notes": [
    "Some null values in customer_id",
    "Date range: 2023-01-01 to 2024-12-31"]
}
```


Step 5: For Each Column – Analyze Details

_analyze_column(table_name, column_name, column_data)

LLM Input:

- Table and Column names
- Sample values (first 10)
- Unique Count
- Null Count

LLM Output (DataTypeResponse):


```
{
  "sql_type": "REAL",
  "python_type": "float",
  "description": "Transaction amount in USD",
  "business_meaning": "Revenue generated from sale",
  "constraints": ["NOT NULL", "CHECK > 0"],
  "is_nullable": false,
  "suggested_index": true
}
```

Step 6: Generate Embeddings

_get_embedding(text)

For Table:

- Text block from table metadata
- POST to Ollama /api/embeddings
- Model: nomic-embed-text
- Returns: 768-dim embedding vector

For Each Column:

- Text block built from column metadata
- Generate 768-dim embedding

Step 7: Store in ChromaDB

save_analysis(analysis)

Table Collection Entry:

- ID: table_sales
- Document: full table description text
- Embedding: 768-dim vector
- Metadata: { table_name, description, category, ... }

Column Collection Entry:

- ID: column_sales_amount
- Document: full column description text
- Embedding: 768-dim vector
- Metadata: { table_name, column_name, sql_type, ... }

Step 8: Repeat for All Tables

- Process next table
- Analyze metadata
- Analyze columns
- Generate embeddings
- Store in ChromaDB
- Log progress

Step 9: Analysis Complete

print_summary()

ANALYSIS SUMMARY

Tables Analyzed: 3

Total Columns: 45

Vector Database: ./chroma_db_768dim

Output: ChromaDB at `./chroma_db_768dim/` with:

- `table_metadata` collection

- `column_metadata` collection

- All embeddings and metadata

ChromaDB Vector Database

Collections

1. table_metadata

- Stores: Table-level semantic information

- ID format: `table_{table_name}`

- Embedding: 768-dimensional vector from nomic-embed-text

Metadata Schema:

```*python*

*{*

*"table_name": str,*

*"description": str,*

*"category": str, # sales, financial, educational, etc.*

*"business_context": str,*

*"suggested_primary_key": str,*

*"data_quality_notes": str (JSON array),*
```

"row_count": int,

"column_count": int

}

```

2. column_metadata

- Stores: Column-level semantic information
- ID format: `column_{table_name}_{column_name}`
- Embedding: 768-dimensional vector

**Metadata Schema**:

```*python*

{

"table_name": str,

"column_name": str,

"sql_type": str, # TEXT, INTEGER, REAL, DATE, BOOLEAN

"python_type": str, # str, int, float, datetime, bool

"description": str,

"business_meaning": str,

"constraints": str (JSON array),

"is_nullable": str (bool as string),

"suggested_index": str (bool as string),

"unique_count": int,

"null_count": int

}

```
```

Vector Search Mechanism

Query Processing

```python
# 1. User asks: "Show me sales data"

# 2. Generate embedding for question
query_embedding = _get_embedding("Show me sales data")
# Result: [0.234, -0.567, ...] (768 numbers)

# 3. Query ChromaDB
results = table_collection.query(
    query_embeddings=[query_embedding],
    n_results=5  # Top 5 most relevant tables)

# 4. ChromaDB returns ranked results
{  "ids": [["table_sales", "table_transactions", ...]],
    "distances": [[0.12, 0.34, ...]],  # Lower = more similar
    "metadatas": [[
        {"table_name": "sales", "description": "..."},
        {"table_name": "transactions", "description": "..."} ]],
    "documents": [[
        "Table: sales\nDescription: ...",
        "Table: transactions\nDescription: ..." ]]}

# 5. Format metadata for LLM context
metadata_str = "\n\n".join(results["documents"][0])
```

Similarity Calculation

Cosine similarity between embeddings

- Score 0.0-1.0 (ChromaDB shows as distance, lower is better)
- Semantic matching: "revenue" finds "sales", "income", "earnings"

Embedding Model

Nomic Embed Text

Specifications:

- Dimensions: 768
- Context length: 8192 tokens
- Hosted: Ollama local server
- Endpoint: `POST http://localhost:11434/api/embeddings`

Request/Response:

```python
# Request
{
   "model": "nomic-embed-text",
   "prompt": "Show me sales data"
}

# Response
{
   "embedding": [0.234, -0.567, 0.123, ..., 0.891]  # 768 floats
}
```

Why 768 dimensions:

- Balance between accuracy and performance
- Standard BERT-base size
- Sufficient for capturing semantic meaning
- Efficient storage (~3KB per embedding)

2. Query Processing (Multi-Turn)

Scenario: New Query (No Context)
User: "What are the top 5 sales by amount?"

UI: *process_question()* receives user input

agent.answer_question_with_context(question)

Step 1: Intent Analysis

Input to LLM:

- Question: "What are the top 5 sales by amount?"
- Recent conversation [ ]

LLM Response:

{
  "intent": "new_query",
  "references_previous": false,
  "referenced_concepts": ["sales", "amount", "top 5"],
  "needs_context": false,
  "confidence": 0.95
}

Step 2: Metadata Retrieval

_retrieve_metadata(question, top_k=5)

- Generate embedding for question:

Example: [0.234, -0.567, 0.123, ...] (768 dims)

- Query ChromaDB table_collection:

Returned similar tables ranked by semantic similarity:

Result: ["sales", "transactions", "revenue"]

- Query ChromaDB column_collection:

Relevant columns: ["amount", "total_sales", "quantity", "price"]

- Format metadata string:

Table: sales

Description: Transaction records...

Columns: amount (REAL), date (DATE)

Step 3: SQL Generation

_generate_sql_with_retry()

LLM Input:

- Database schema (from SQLDatabase)
- Metadata context (from ChromaDB)
- Question: "What are the top 5 sales by amount?"

LLM Output:

```*sql*

*SELECT * FROM sales*

*ORDER BY amount DESC*

*LIMIT 5*

```

Step 4: SQL Cleaning and Validation

_clean_sql() + _validate_and_fix_tables()

Cleaning actions:

- Remove markdown ``` fences
- Extract SQL (SELECT Statement)
- Balance Quotes
- Repair syntax

Validation steps:

- Check Table exists: sales → yes
- Check Column exists: amount → yes
- Validate joins (If any)

Step 5: Query Execution

pd.read_sql_query(sql, conn)

Result DataFrame (example):

transaction_id	amountdate	category	
TXN-1234	1536.17	2024-03-15	Electronics
TXN-5678	1112.25	2024-02-28	Home

Step 6: Answer Generation

_generate_answer(question, df, sql)

LLM prompt:

- Question
- Data summary -- "Found 5 rows with columns..."
- Sample rows (first 10 rows)

LLM Response:

The top 5 sales by amount are:

1. $1,536.17 from Electronics (TXN-1234)

…

5. $246.47 from Beauty (TXN-7890) The highest transaction was in Electronics.

Step 7: Visualization Decision

_should_visualize(question, df)

Check keywords: "show", "draw", "graph" → No viz keywords

Result: should_visualize = False (User didn't explicitly request visualization)

Step 8: State Update

_update_conversation_state()

Add user message:

role: user

content: "What are the top 5 sales?"

metadata: { "intent": "new_query" }

Add assistant message:

role: assistant

content: "The top 5 sales..."

*sql_query: "SELECT * FROM sales..."*

dataframe_snapshot:

columns: [transaction id, …]

row_count: 5

sample: { first 50 rows as dict}

visualization: None

metadata: { "success": true }

Add data context:

*query: "SELECT * FROM sales..."*

columns: ["transaction_id", "amount", ...]

row_count: 5

Step 9: UI Display

process_question() returns updated history.

Gradio UI shows:

- User message
- Assistant answer
- Data table with 5 rows

Step 10: Conversation Persistence

save_current_conversation()

Saved file example: conversations/a1b2c3d4-uuid.json

Contents: full conversation state as JSON.

Scenario: Follow-Up Query with Visualization

User: Show me those sales as a bar chart

Step 1: Intent Analysis

Input to LLM:

- Question: "Show me those sales as a bar chart"
- Recent conversation:

"- user: What are the top 5 sales by amount?

assistant: The top 5 sales are..."

LLM Response:

{

"intent": "re_visualize",

"references_previous": true,

"referenced_concepts": ["those sales", "bar chart"],

"needs_context": true,

"confidence": 0.92

}

Step 2: Retrieve Previous Data

_check_previous_reference()

Keywords detected: "those" → Search *conversation_state* for latest Data Frame

Found: DataFrame from previous query (5 rows)

transaction_id | amount | date | category

TXN-1234 | 1536.17 | 2024-03-15 | Electronics

TXN-5678 | 1112.25 | 2024-02-28 | Home

Step 3: Route to Re-visualization Handler

_handle_revisualization(question, df)

- Extract filters from previous SQL:

  Previous SQL:

  *SELECT * FROM sales ORDER BY amount DESC LIMIT 5*

  Filters found: None

- Apply filters to DataFrame: N/A

- Generate visualization recommendation:

  _should_visualize(question, df)

Step 4: Visualization Decision

Detect keywords: "bar chart"

LLM Prompt includes:

- Question
- Data info (5 rows, 4 columns)
- Numeric columns: [amount]
- Sample data

LLM Response:

"should_visualize": true,

"chart_types": ["bar"],

"primary_chart": "bar",

"x_axis": "category",

"y_axis": "amount",

"color_by": "category",

"title": "Top 5 Sales by Amount",

"visualization_rationale":

"Bar chart effectively shows comparison of amounts across categories"

Step 5: Create Plotly Chart

_create_visualization(df, viz_response)

Result: Interactive Plotly Figure object

Step 6: Generate Answer

_generate_answer(question, filtered_df, previous_sql)

LLM Response:

> "I've created a bar chart showing the top 5 sales by amount. Electronics leads with $1,536.17, followed by Home at $1,112.25.The visualization clearly shows the distribution of high-valuetransactions across different categories."

Step 7: State Update

_update_conversation_state()

Add User Message:

role="user"

content="Show me those sales as a bar chart"

metadata={"intent": "re_visualize"})

Add Assistant Message:

role="assistant",

content="I've created a bar chart...",

*sql_query="SELECT * FROM sales...",*

dataframe_snapshot={...},

visualization="bar",

figure_json="{...}",

metadata={"success": True, "reused_data": True}

Add Visualization Record:

question="Show me those sales as a bar chart",

chart_type="bar",

data_summary="5 rows"

Step 8: UI Display

Gradio Chat Display shows:

- User message
- Assistant text summary
- Interactive Plotly Bar Chart:

  X-axis: Category, Y-axis: Amount

  Bars colored by category

  Hover tooltips

- Data Table with the same 5 rows.

Scenario: Data Transformation

User: Filter only Electronics category

Step 1: Intent Analysis

Transform

Step 2: Retrieve Previous Data Frame

5 rows

Step 3: Generate Transformation Code

_handle_transformation(question, df)

LLM Prompt:

- Dataframe columns: [transaction_id, amount, ...]
- User request: "Filter only Electronics category"
- Instructions: Generate pandas code using 'df' var

LLM Response:

df[df['category'] == 'Electronics']

Step 4: Execute Transformation Safely

local_vars = {"df": df.copy(), "pd": pd}

exec(code, {"__builtins__": {}}, local_vars)

transformed_df = local_vars["df"]

Result:

	transaction_id	amount	date	category
0	TXN-1234	1536.17	2024-03-15	Electronics

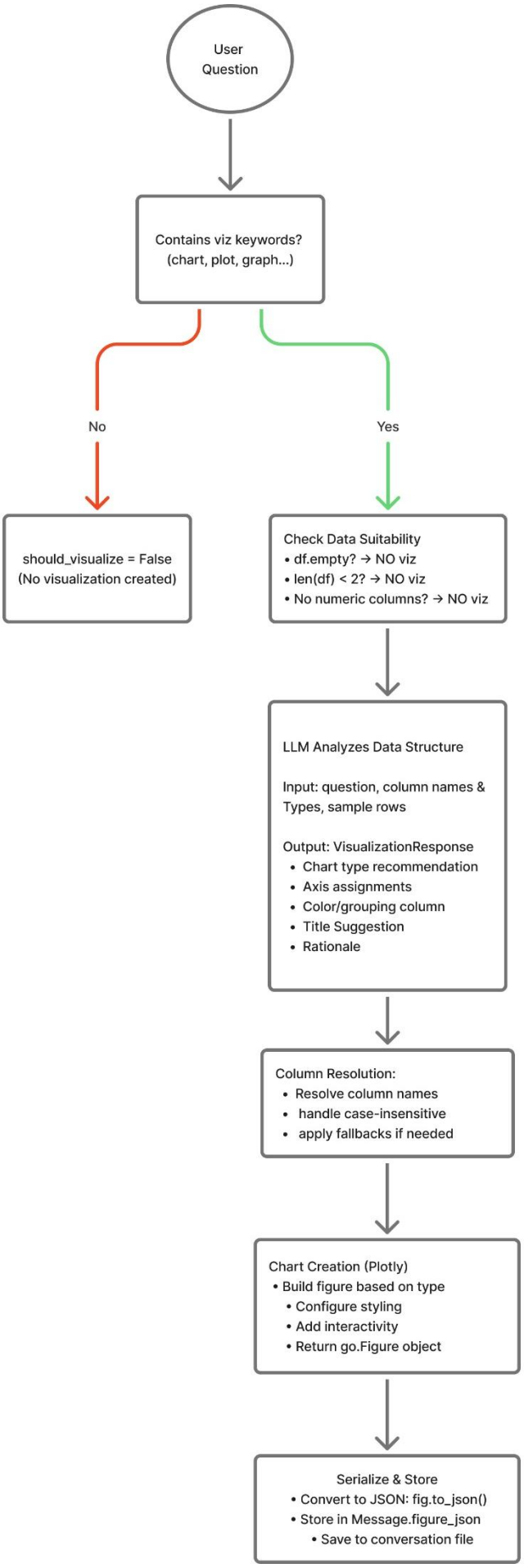
Step 5: Generate Answer

I've filtered the data to show only Electronics category. Found 1 transaction with amount $1,536.17.

Step 6: State Update & UI Display

(Similar pattern as previous scenarios)

Visualization Pipeline



Visualization Preservation

Plotly figures are complex objects that can't be directly serialized to JSON. So I use `fig.to_json()` and `go.Figure(json.loads(...))`

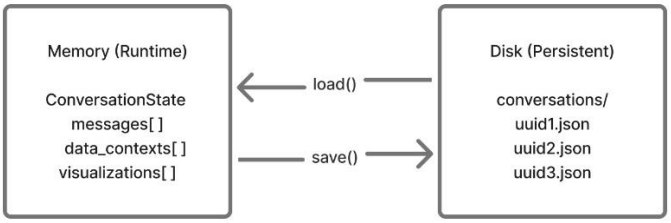
```*python*

*# Save*

*figure_json = fig.to_json()*

*message.figure_json = figure_json*

*# Load*

*fig = go.Figure(json.loads(message.figure_json))*

*history.append((None, gr.Plot(value=fig)))*

```

JSON Structure (abbreviated):

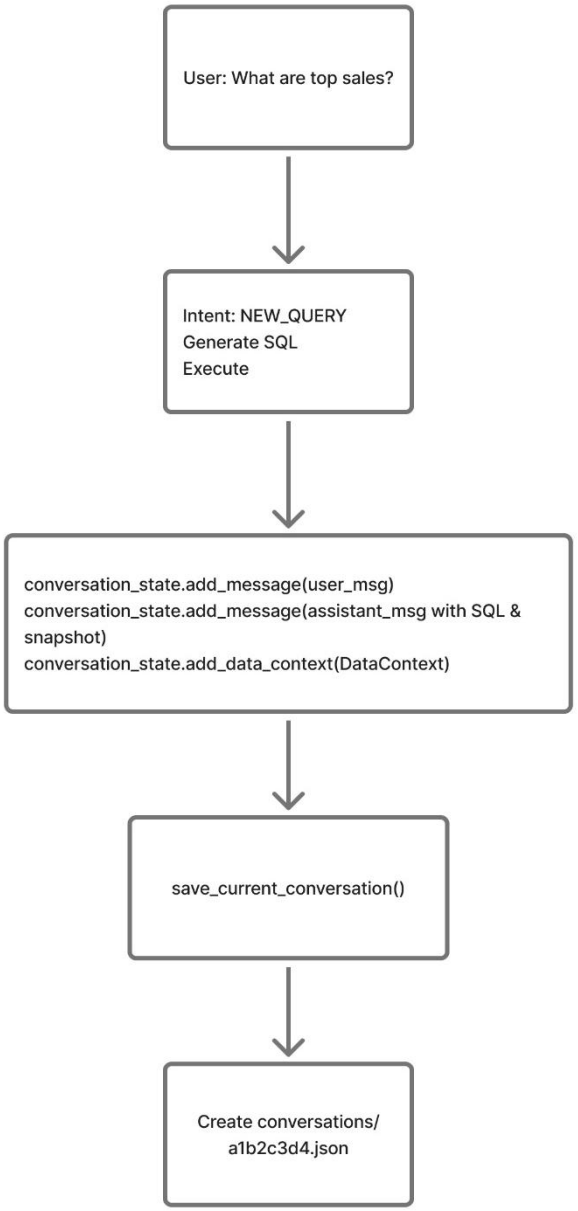
```json
{
  "data": [
                {
                  "type": "bar",
                  "x": ["Cat1", "Cat2", "Cat3"],
                  "y": [10, 20, 15],
                  "marker": {"color": "#636efa"}
                }
        ],
  "layout": {
                "title": {"text": "My Chart"},
                "xaxis": {"title": {"text": "Category"}},
                "yaxis": {"title": {"text": "Value"}},
                "template": "plotly"
            }
}
```

3. Conversation Management System

State Persistence Architecture

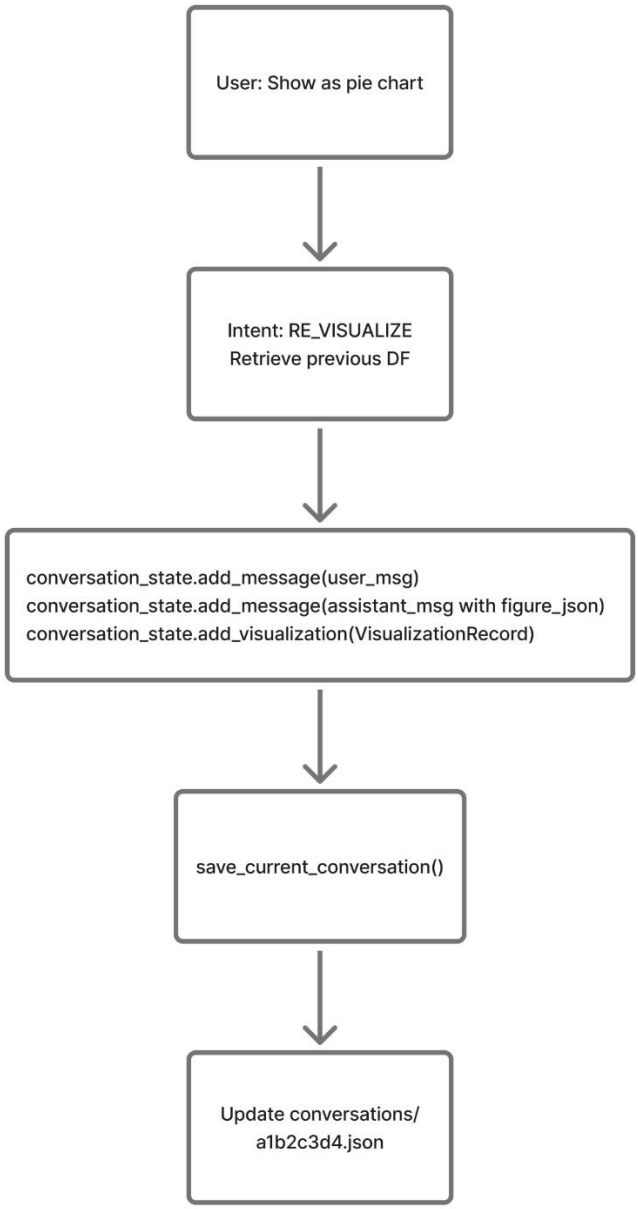


Message Flow Timeline



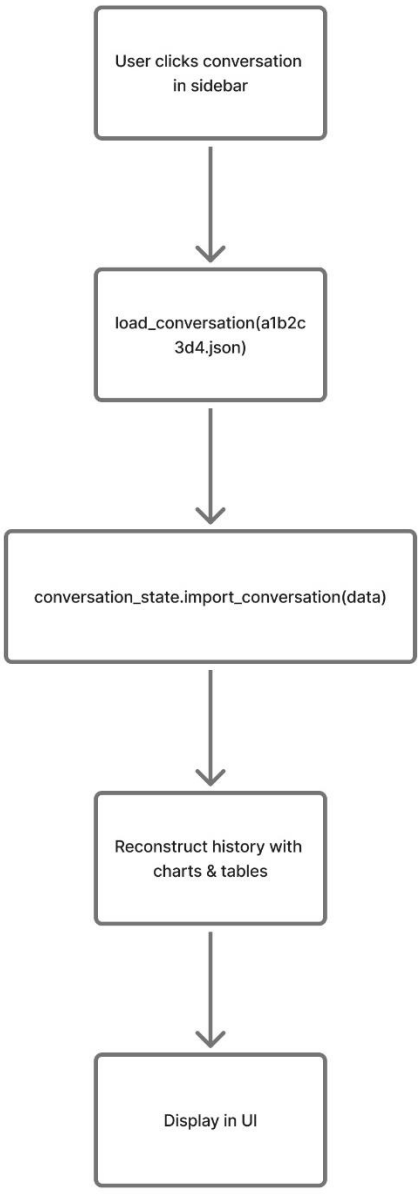
Time → T0

First Query & Save



T1 → After Few Minutes

Re-visualization and Update

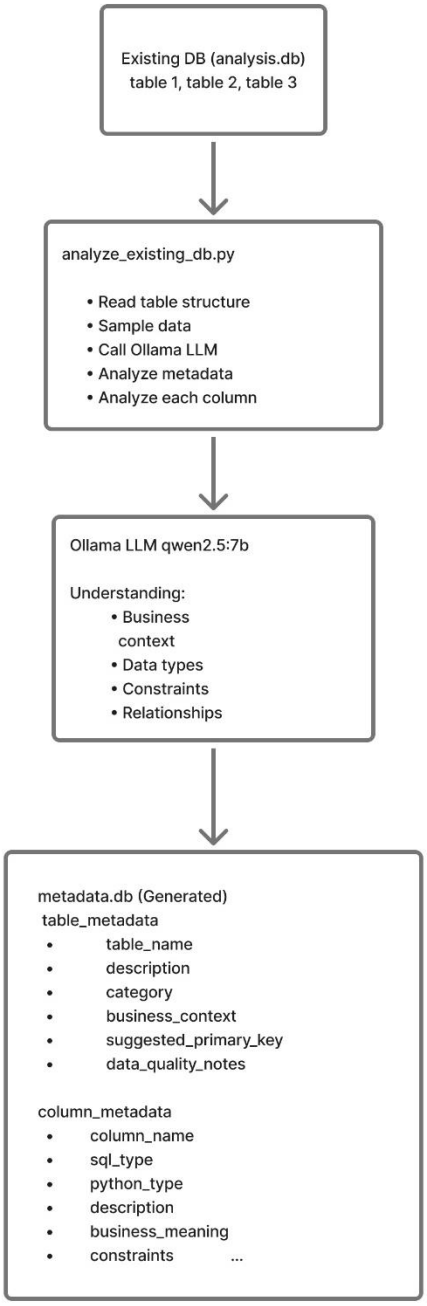


T2 → Next Day

Loading Previous Conversation

Memory Management Strategy

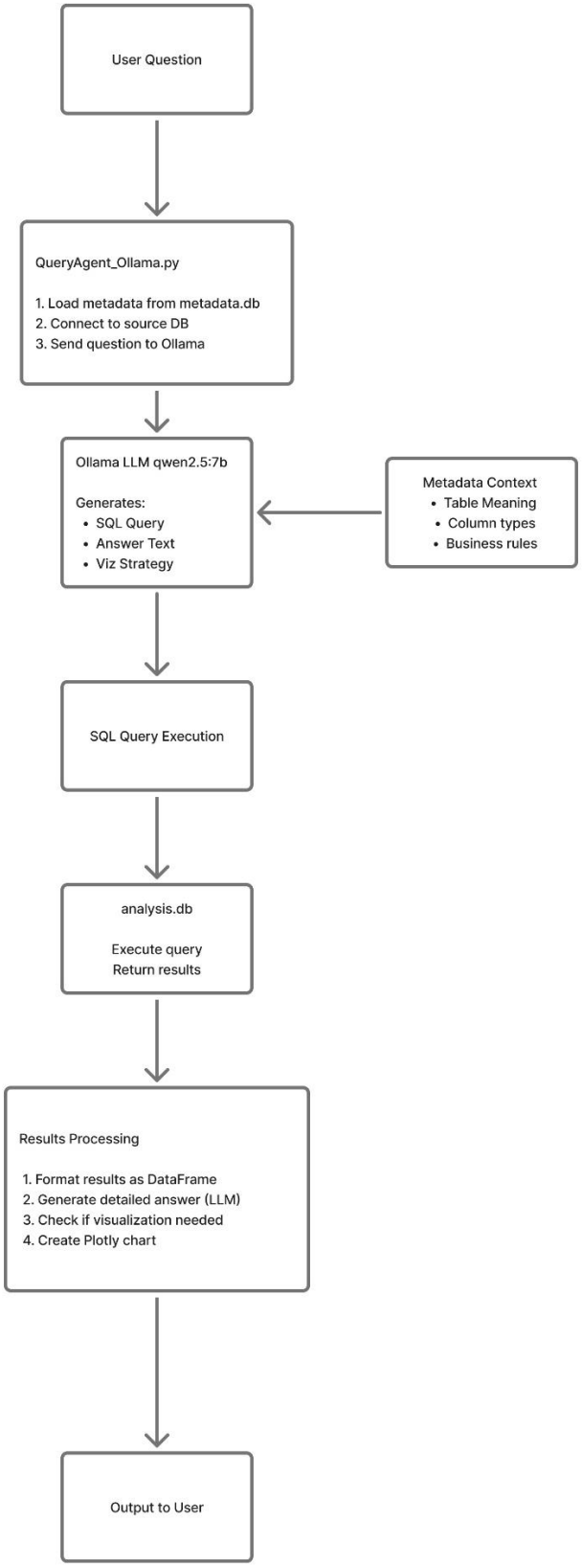
Long conversations can consume excessive memory. Therefore, I implement multi-level cleanup.



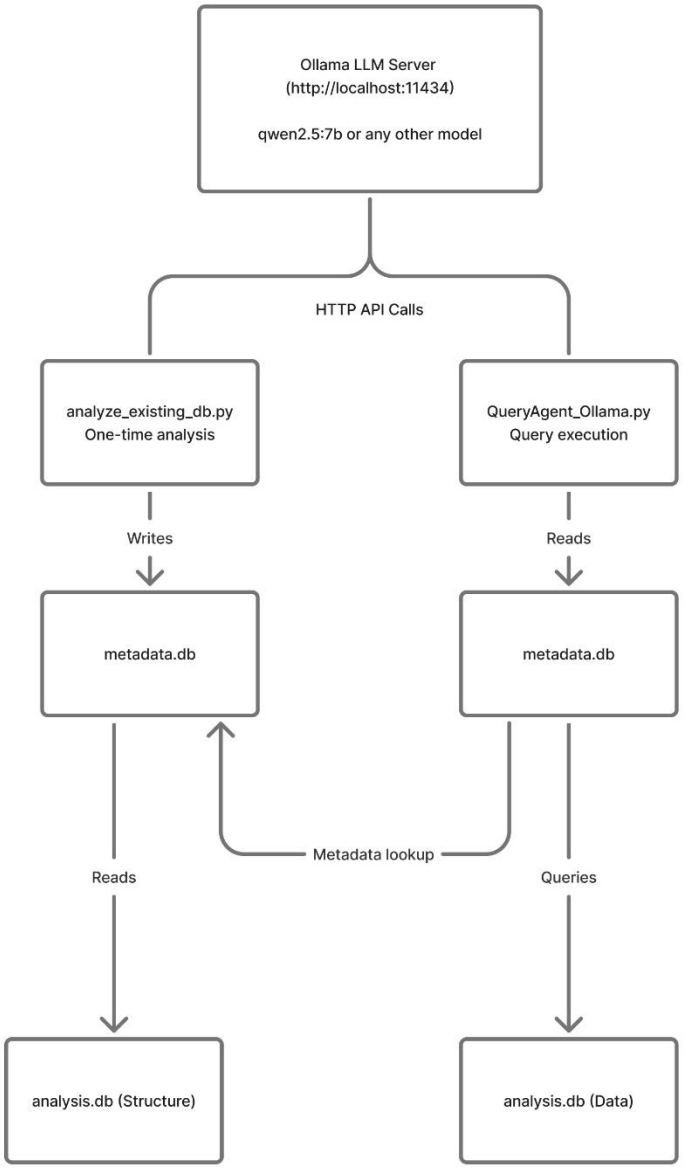
One time Process

Re run Only If the Schema Changed

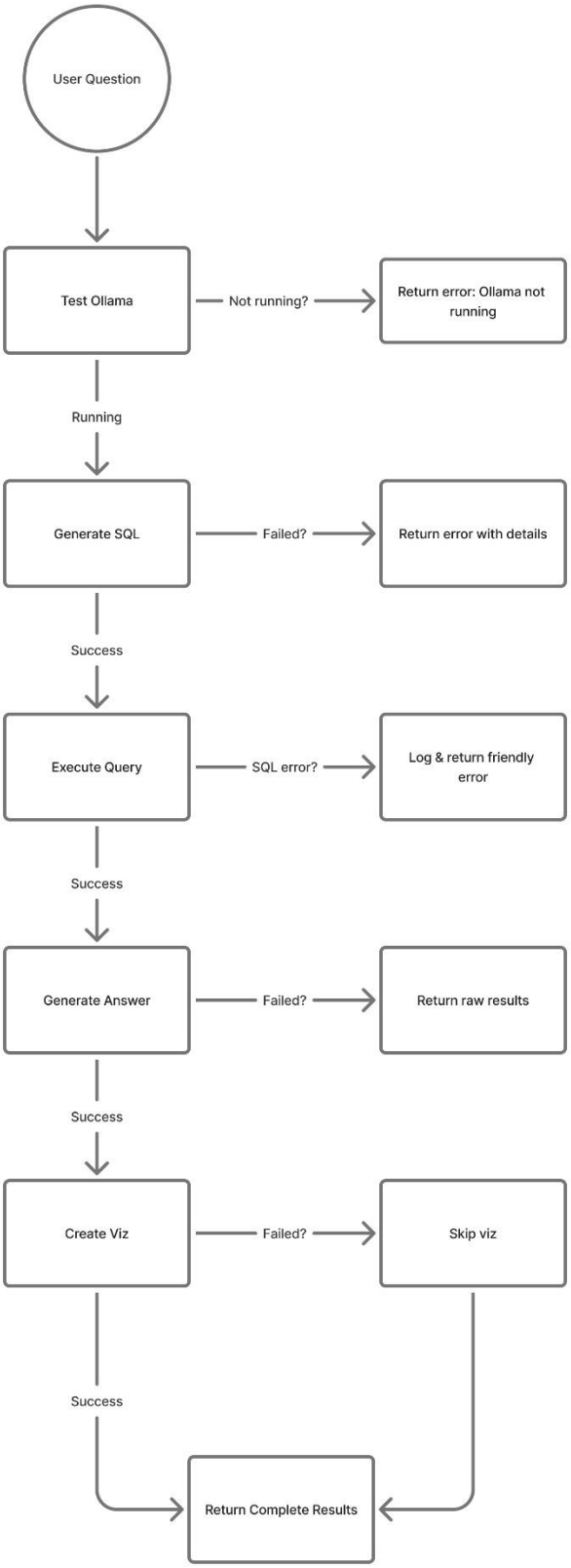
Phase 2


User Question
QueryAgent_Ollama.py
1. Load metadata from metadata.db
2. Connect to source DB
3. Send question to Ollama
Ollama LLM qwen2.5:7b
Generates:
• SQL Query
• Answer Text
• Viz Strategy
Metadata Context
• Table Meaning
• Column types
• Business rules
SQL Query Execution
analysis.db
Execute query
Return results
Results Processing
1. Format results as DataFrame
2. Generate detailed answer (LLM)
3. Check if visualization needed
4. Create Plotly chart
Output to User

Component Interaction



Error Handling Flow



##

Data Structures and Examples

##

1. Conversation JSON File Structure

```json
{
  "conversation_id": "2e6a2e96-1a3f-43f8-95cb-2a1debc1135a",
  "start_time": "2025-11-20T22:35:32.992155",
  "message_count": 4,
  "messages": [
    {
      "role": "user",
      "content": "What are the top 5 sales?",
      "timestamp": "2025-11-20T22:35:40.123456",
      "sql_query": null,
      "dataframe_snapshot": null,
      "visualization": null,
      "figure_json": null,
      "metadata": {
        "intent": "new_query"
      }
    },
    {
      "role": "assistant",
      "content": "The top 5 sales by amount are:\n1. $1,536.17 from Electronics (TXN-1234)\n2.
$1,112.25 from Home (TXN-5678)\n...",
      "timestamp": "2025-11-20T22:35:45.789012",
      "sql_query": "SELECT * FROM sales ORDER BY amount DESC LIMIT 5",
      "dataframe_snapshot": {
        "columns": ["transaction_id", "amount", "date", "category"],
        "row_count": 5,
        "sample": {
          "transaction_id": {
            "0": "TXN-1234",
            "1": "TXN-5678",
            "2": "TXN-9012",
            "3": "TXN-3456",
            "4": "TXN-7890"
          },
          "amount": {
            "0": 1536.17,
            "1": 1112.25,
            "2": 765.28,
            "3": 508.85,
            "4": 246.47
          },
          "date": {
            "0": "2024-03-15",
            "1": "2024-02-28",
```

```
            "2": "2024-01-12",
            "3": "2024-04-03",
            "4": "2024-03-22"
          },
          "category": {
            "0": "Electronics",
            "1": "Home",
            "2": "Sports",
            "3": "Fashion",
            "4": "Beauty"
          }
        }
      },
      "visualization": null,
      "figure_json": null,
      "metadata": {
        "intent": "new_query",
        "success": true
      }
    },
    {
      "role": "user",
      "content": "Show as bar chart",
      "timestamp": "2025-11-20T22:36:00.111222",
      "sql_query": null,
      "dataframe_snapshot": null,
      "visualization": null,
      "figure_json": null,
      "metadata": {
        "intent": "re_visualize"
      }
    },
    {
      "role": "assistant",
      "content": "I've created a bar chart showing the top 5 sales by amount...",
      "timestamp": "2025-11-20T22:36:05.333444",
      "sql_query": "SELECT * FROM sales ORDER BY amount DESC LIMIT 5",
      "dataframe_snapshot": {
        "columns": ["transaction_id", "amount", "date", "category"],
        "row_count": 5,
        "sample": {...}
      },
      "visualization": "bar",
      "figure_json":
"{\"data\":[{\"type\":\"bar\",\"x\":[\"Electronics\",\"Home\",\"Sports\",\"Fashion\",\"Beauty\
"],\"y\":[1536.17,1112.25,765.28,508.85,246.47],\"marker\":{\"color\":\"#636efa\"}}],\"layout\
":{\"title\":{\"text\":\"Top 5 Sales by
Amount\"},\"xaxis\":{\"title\":{\"text\":\"Category\"}},\"yaxis\":{\"title\":{\"text\":\"Amoun
t\"}}}}",
      "metadata": {
        "intent": "re_visualize",
        "success": true
      }
    }
```

```
  ],
  "data_contexts": [
    {
      "query": "SELECT * FROM sales ORDER BY amount DESC LIMIT 5",
      "columns": ["transaction_id", "amount", "date", "category"],
      "row_count": 5,
      "sample_data": {
        "transaction_id": {"0": "TXN-1234", "1": "TXN-5678", ...},
        "amount": {"0": 1536.17, "1": 1112.25, ...},
        "date": {"0": "2024-03-15", "1": "2024-02-28", ...},
        "category": {"0": "Electronics", "1": "Home", ...}
      },
      "timestamp": "2025-11-20T22:35:45.789012"
    }
  ],
  "visualizations": [
    {
      "question": "Show as bar chart",
      "chart_type": "bar",
      "data_summary": "5 rows",
      "timestamp": "2025-11-20T22:36:05.333444"
    }
  ]
}
```

2. Answer Examples

Example 1: Simple Query Answer

```

Question: "What are the total sales by region?"

Answer: "The total sales by region are as follows:

- North: $245,678.90 (32% of total)

- South: $198,432.15 (26% of total)

- East: $176,543.22 (23% of total)

- West: $142,109.73 (19% of total)

The North region leads with the highest sales, accounting for nearly one-third of all revenue. Total across all regions: $762,764.00."
```

SQL: SELECT region, SUM(amount) as total_sales

FROM sales

GROUP BY region

ORDER BY total_sales DESC

```

**Example 2: Comparative Analysis Answer**

```

Question: "Compare Q1 vs Q2 sales"

Answer: "Comparing Q1 and Q2 sales performance:

Q1 (Jan-Mar): $1,234,567.89

Q2 (Apr-Jun): $1,456,234.12

Key findings:

• Q2 sales increased by $221,666.23 (+18% growth)

• Average daily sales: Q1 = $13,720, Q2 = $16,003

• Peak month: June with $523,145.67

• Growth driven primarily by Electronics (+45%) and Home categories (+32%)

The upward trend suggests strong market momentum heading into the second half of the year."

SQL: SELECT

CASE

WHEN CAST(SUBSTR(date, 6, 2) AS INTEGER) <= 3 THEN 'Q1'

WHEN CAST(SUBSTR(date, 6, 2) AS INTEGER) <= 6 THEN 'Q2'

```
    END as quarter,
    SUM(amount) as total_sales,
    COUNT(*) as transaction_count
FROM sales
WHERE quarter IN ('Q1', 'Q2')
GROUP BY quarter
```

Example 3: Distribution Analysis Answer

```
Question: "Show profit margin distribution by category using box plot"

Answer: "The profit margin distribution reveals interesting patterns across categories:

• Electronics: Widest range (-2.95 to 1536.17), median 55.72
  - High variability indicates diverse product mix
  - Outliers on high end (premium items)

• Home: Second-highest median (47.86), range -3.55 to 1112.25
  - Consistent profitability with occasional high-value items

• Grocery: Notable concern - negative median (-2.26)
  - Range: -6.20 to 26.47
  - May require pricing strategy review

• Fashion, Beauty, Sports: Moderate ranges with positive medians
  - Relatively stable profitability
```

• Toys: Lowest median (7.93), narrow range

- Consistent but low margins

The box plot visualization shows quartiles, outliers, and overall spread for each category, making it easy to identify which segments have the most variable or stable profit margins."

Visualization: Box plot with category on X-axis, profit_margin on Y-axis

```
```

##

Best Practices and Troubleshooting

##

1. Best Practices for Getting Good Answers

1. Be Specific

❌ "Show me data"

☑ "Show me the top 10 sales transactions by amount"

❌ "What about last month?"

☑ "What were the total sales for March 2024?"

2. Use Visualization Keywords

☑ "Show me sales by category as a bar chart"

☑ "Visualize the trend"

☑ "Create a pie chart of distribution"

☑ "Plot the correlation"

3. Build on Previous Questions

Q1: "What are the total sales by region?"

Q2: "Show those as a map" # References Q1 results

Q3: "Which region grew the most?" # Still in context

Q4: "Show me the top customers in that region" # Still connected

4. Provide Context When Needed

☑ "Show me sales for Q1 2024" # Clear time period

☑ "Compare Electronics vs Home categories" # Clear comparison

☑ "Filter transactions above $1000" # Clear threshold

5. Ask for Explanations

Q: "Why is Electronics the best category?"

Q: "What factors contributed to July's peak?"

Q: "Explain the correlation between price and quantity"

2. Troubleshooting Common Issues

Issue 1: Agent not responding

Symptoms: Spinning indicator, no answer

Causes:

1. Ollama not running

2. Model not loaded

3. Database connection lost

Solutions:

```powershell
# Check Ollama status
curl http://localhost:11434/api/tags
# Restart Ollama if needed
ollama serve
# Reload page and reinitialize agent
```

Issue 2: SQL execution error

Symptoms: Error message in chat

Causes:

1. Invalid table/column reference

2. Syntax error in generated SQL

3. Database locked

Solutions:

- Automatic: System retries with error feedback to LLM

- Manual: Rephrase question with explicit table/column names

- Example: Instead of "Show sales", try "Show data from sales_transactions table"

Issue 3: No visualization created

Symptoms: Answer but no chart

Causes:

1. No viz keywords in question

2. Data not suitable for visualization

3. Chart creation failed

Solutions:

Be explicit about visualization

❌ "Show me sales data"

☑ "Show me sales data as a bar chart"

Or ask for viz after getting data

Q: "Show me sales by category"

[Get table]

Q: "Visualize that as a bar chart"

[Get chart]

Issue 4: Can't find previous data

Symptoms: "Show me those" doesn't work

Causes:

1. Started new conversation (data not persisted across chats)

2. Too many turns (data contexts auto-cleaned after 10)

Solutions:

- Use same conversation

- Re-run original query if needed

- Loaded conversations restore all data

3. Error Categories & Responses

1. Ollama Connection Errors

Scenario: Ollama server not running

```*python*

*Error: Cannot connect to Ollama: Connection refused*

*Recovery:*

*1. Check Ollama status: `ollama serve`*

*2. Verify port: Default 11434*

*3. Check firewall rules*

*4. Restart Ollama service*

*User Message: "✗ Ollama is not running. Please ensure Ollama is installed and running with: ollama serve"*

```

Detection:

```*python*

*def test_ollama_connection(self) -> bool:*

```

2. SQL Execution Errors

Scenario: Generated SQL is invalid

```*python*

*Error: no such column: invalid_column*

*Recovery Strategy:*

*1. Log error with full SQL query*

*2. Extract error details (column name, table reference)*

*3. Build error feedback prompt for LLM*

*4. Regenerate SQL with error context*

*5. Retry execution (max 1 retry)*

*Prompt Example:*

*"Previous SQL attempt:*

*SELECT invalid_column FROM sales*

*SQLite error: no such column: invalid_column*

*Available columns: [transaction_id, amount, date, category]*

*Rewrite the SQL to use valid column names."*

```

3. LLM Parsing Errors

Scenario: LLM returns malformed JSON

```python
Error: JSON parsing failed: Expecting ',' delimiter

Example Bad Response:
  "intent": "new_query",
  "confidence": 0.9,  # Missing closing brace
```

Recovery:

```python
def _parse_with_fallback(self, response_text, parser, fallback_data):
```

4. Visualization Errors

Scenario: Cannot create chart

```python
Error: KeyError: 'invalid_column'
```

Recovery:

1. Log error with visualization config

2. Attempt column name resolution

3. If fails, skip visualization

4. Return results without chart

5. Inform user in answer text

User Message: "I couldn't create the visualization due to a column mismatch, but here are the query results..."

```

**5. Conversation Load Errors**

Scenario: Corrupted conversation file

```*python*

Error: JSON decode error in conversation file

Recovery:

1. Log error with filename

2. Attempt partial recovery:

- Load valid messages only

- Skip corrupted visualizations

3. Display warning to user

4. Allow continued use with recovered data

User Message: "⚠️ Some data couldn't be loaded from this conversation, but I've recovered what I could."

```

**Logging Strategy**

Log Levels:

- DEBUG: SQL queries, LLM prompts, detailed flow
- INFO: Major operations, user actions, success messages
- WARNING: Fallback usage, retries, non-critical issues
- ERROR: Failures, exceptions, invalid states

Example Logs:

```

2025-11-20 22:35:40 INFO: Detected intent: new_query (confidence: 0.95)

*2025-11-20 22:35:42 DEBUG: Generated SQL: SELECT * FROM sales ORDER BY amount DESC LIMIT 5*

2025-11-20 22:35:43 INFO: Query executed successfully, returned 5 rows

2025-11-20 22:35:45 WARNING: Pydantic parsing failed: JSONDecodeError, using fallback

2025-11-20 22:35:46 INFO: Created bar chart visualization

2025-11-20 22:35:47 INFO: Conversation saved: a1b2c3d4-uuid.json

```
```

##

END OF DOCUMENTATION

##

For questions or support, refer to the code comments or contact me on nuhang.ou@mobitel.lk .

Last Updated: December 05, 2025

Document Author: Nuhan Maleesha Gunasekara