**Self Service Visualizer and Chatbot Documentation**

[**Problem Statement** 2](#_Toc184234534)

[**Architecture Overview** 2](#_Toc184234535)

[**1. Start Node** 3](#_Toc184234536)

[**2. Setup SQL DB Connection** 3](#_Toc184234537)

[**3. Fetch SQL DB/Table Schema** 3](#_Toc184234538)

[**4. Instruction** 3](#_Toc184234539)

[**5. Prompt** 4](#_Toc184234540)

[**6. LLM Component - SQL Query Generation** 4](#_Toc184234541)

[**7. Execute** 4](#_Toc184234542)

[**8. Transaction Complete** 4](#_Toc184234543)

[**9. LLM Component - Visualization Code Generation** 4](#_Toc184234544)

[**10. Query Output** 5](#_Toc184234545)

[**11. Streamlit UI** 5](#_Toc184234546)

[**12. Final Output** 5](#_Toc184234547)

[**13. Stop Node** 5](#_Toc184234548)

[**Process Flow** 6](#_Toc184234549)

[**Tech Stack Used** 6](#_Toc184234550)

[**Installation & Configuration** 6](#_Toc184234551)

[**Application Structure** 6](#_Toc184234552)

[**Tab 1: Data Visualizer** 7](#_Toc184234553)

[**Tab 2: Chatbot** 8](#_Toc184234554)

[**Database Connection** 10](#_Toc184234555)

[**API Integration & Error Handling** 11](#_Toc184234556)

[**Database Schema** 16](#_Toc184234557)

[**Streamlit UI - Data Visualizer & Chatbot** 16](#_Toc184234558)

[**DATA VISUALIZER** 17](#_Toc184234559)

[CHATBOT 20](#_Toc184234560)

[**Closing the connection** 22](#_Toc184234561)

## 

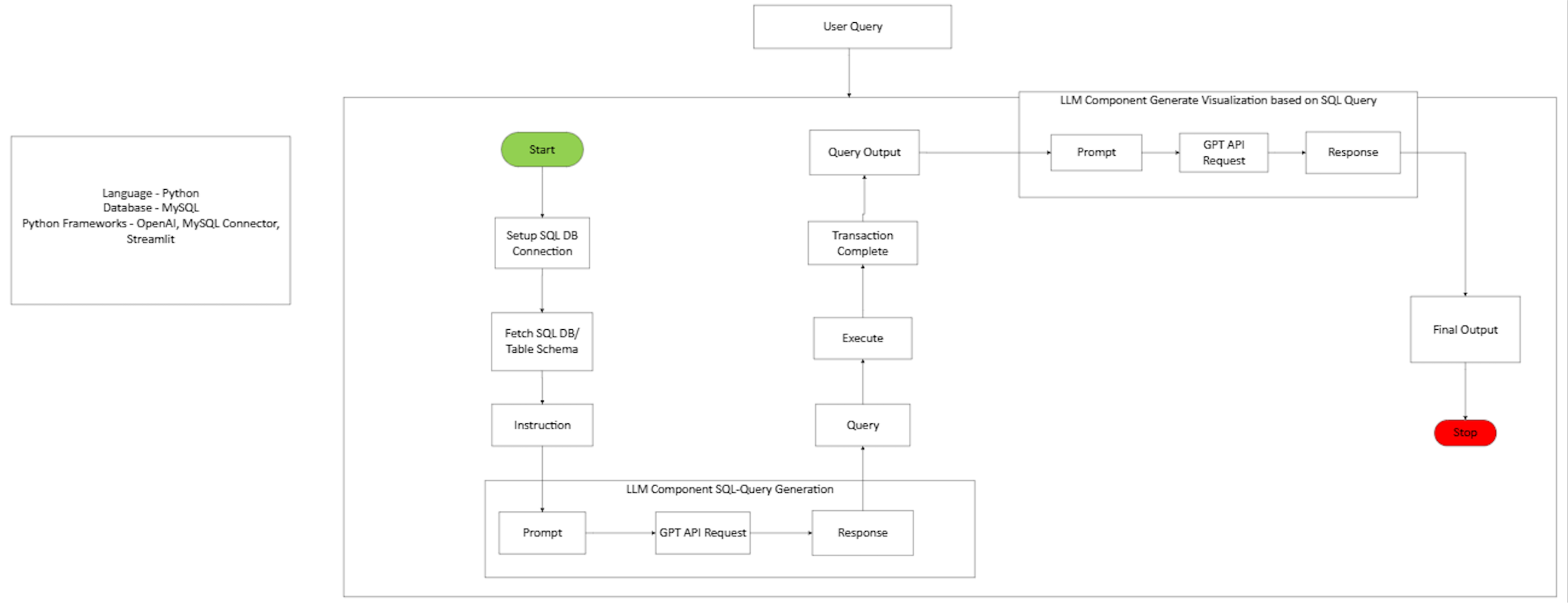
## **Problem Statement**

In this project, the objective is to create a Streamlit-powered LLM app that answers the queries of the users by allowing them to give the prompt, based on the policy dataset, which is stored in an Azure SQL Database. It is a visualizer-chatbot interface that was developed as a part of the DataSonic project.

This Streamlit-based application is basically a query-based visualizer and chatbot designed to facilitate data visualization and interaction through a chatbot interface. It integrates with OpenAI's API for generating SQL queries and Python visualization code and connects to an Azure Synapse SQL database for data retrieval.

1. **Chatbot** - Returns and displays any possible numeric output along with the text explanation of the same based on the prompt given by the user from the database schema.
2. **Visualizer** - Returns and displays the most appropriate visualization or simply based on the prompt given by the user from the database schema.

## **Architecture Overview**

****

The architecture of the system involves multiple interconnected components that enable seamless SQL query generation, execution, and visualization. Below is a detailed breakdown of all components involved in the system:

#### **1. Start Node**

* **Purpose:** This is the entry point of the system where the application initializes.
* **Action:** It triggers the setup process for database connectivity and prepares the environment for schema fetching and query generation.

#### **2. Setup SQL DB Connection**

* **Purpose:** Establishes a connection to the SQL database using credentials.
* **Details:**
  + **Library Used:** pyodbc
  + **Configuration:** Database server details, credentials, and ODBC driver specifications are used.
  + **Error Handling:** Ensures connection errors are captured and displayed to the user.

#### **3. Fetch SQL DB/Table Schema**

* **Purpose:** Retrieves metadata from the SQL database to facilitate query generation and visualization.
* **Details:**
  + **Presentation Schema:** Only schema details relevant to the presentation layer are fetched for Data Visualizer tasks.
  + **Entire Schema:** Complete database schema is retrieved for chatbot functionality.
  + **Output Format:** JSON representation of the schema, including tables, views, columns, and data types.

#### **4. Instruction**

* **Purpose:** Forms the system-level instructions provided to the GPT API.
* **Details:**
  + **Data Visualizer Instructions:** Focus on generating optimized SQL queries restricted to the presentation schema.
  + **Chatbot Instructions:** Generate broader SQL queries while adhering to database constraints.
* **Key Feature:** Instructions ensure that the AI aligns generated queries with user requirements and database capabilities.

#### **5. Prompt**

* **Purpose:** Combines user input and schema details into a structured prompt for the GPT API.
* **Details:**
  + User queries are contextualized with database schema information.
  + Prompts are dynamically generated to meet user needs (e.g., SQL generation, data visualization).

#### **6. LLM Component - SQL Query Generation**

* **Purpose:** Uses GPT API to generate SQL queries based on the given prompt.
* **Process:**
  + Prompt and instructions are sent to GPT API.
  + The API returns the SQL query aligned with the schema and user intent.
* **Error Handling:** If the API fails or rate limits occur, appropriate error messages are shown to the user.

#### **7. Execute**

* **Purpose:** Executes the generated SQL query on the connected database.
* **Details:**
  + Executes SQL queries using the pyodbc cursor.
  + Fetches results in a structured format (e.g., rows of data).
* **Error Handling:** If SQL execution fails (e.g., syntax errors, missing tables), errors are captured and displayed.

#### **8. Transaction Complete**

* **Purpose:** Indicates successful query execution and data retrieval.
* **Details:**
  + The fetched data is formatted for further processing (e.g., visualization or chatbot responses).

#### **9. LLM Component - Visualization Code Generation**

* **Purpose:** Uses GPT API to generate Python code for visualizing the fetched data.
* **Details:**
  + Libraries Supported: Altair, Plotly, Matplotlib.
  + Input: Data and user instructions for visualization (e.g., bar chart, pie chart).
* **Execution:** The generated code is executed dynamically using Python's exec().

#### **10. Query Output**

* **Purpose:** Processes and formats the data retrieved from the SQL query for visualization or chatbot interpretation.
* **Details:**
  + Output is either shown as a chart/graph or as a natural language response (for chatbot queries).

#### **11. Streamlit UI**

* **Tabs:**
  + **Data Visualizer Tab:**
    - User provides queries to fetch and visualize data.
    - Generated SQL query results are plotted using visual libraries.
  + **Chatbot Tab:**
    - User queries are interpreted, and SQL query results are translated into natural language.
* **Dynamic Interaction:** Users can provide inputs in text areas and receive outputs in real time.

#### **12. Final Output**

* **Purpose:** Presents the results (charts, graphs, or textual responses) to the user.
* **Details:**
  + Visualizations are displayed directly in the Streamlit app.
  + Chatbot responses are rendered as text for user clarity.

#### 

#### **13. Stop Node**

* **Purpose:** Marks the end of the application lifecycle.
* **Action:** Closes all database connections and releases resources.

## **Process Flow**

* The user provides input via the Streamlit UI (either in the Data Visualizer or Chatbot tab).
* The system establishes a database connection and fetches schema details.
* User input and schema details are combined to form a user prompt for the GPT API.
* SQL queries and visualization code are generated and executed.
* Results are presented as visualizations or natural language responses.
* On application exit, all connections are closed to prevent resource leakage.

## **Tech Stack Used**

The following is the list of the tech stacks that have been used in the project:

* OpenAI GPT 4o
* Streamlit 1.24.0
* Python 3.13
* Pyodbc - ODBC Driver 17
* Plotly 5.24.1
* Matplotlib 3.9
* Altair 5.5.0

## **Installation & Configuration**

To install the necessary dependencies, run. Ensure you have an api\_key.txt file in the root directory containing your OpenAI API key. This key is essential for making API calls to OpenAI's services.

|  |
| --- |
| pip install streamlit openai pyodbc altair plotly matplotlib |

## **Application Structure**

The application files are structured as follows:

* **api\_call:** Function to interact with OpenAI's API.
* **Database Connection:** Establishes connection to the Azure Synapse SQL database.
* **Schema Fetching:** Retrieves schema information for the presentation layer and the entire database.
* **Streamlit UI:** Sets up the user interface with two tabs: Data Visualizer and Chat-Bot.

The application structure is built around two core functionalities: Data Visualizer and Chatbot, each driven by distinct system prompts that define their behavior. Below is an explanation of the structural differences and how the system prompts are tailored for each part.

### **Tab 1: Data Visualizer**

**Purpose**

The Data Visualizer generates SQL queries based on user input and schema information restricted to the presentation layer. After executing the query, it generates visualizations based on the query results.

**Key Features**

* **Schema Restriction:** Queries are restricted to the presentation schema.
* **Visualization-Focused:** Results are visualized using Python libraries like Plotly, Altair, and Matplotlib.
* **Dual Prompts:** Two system prompts are used:
  + To generate SQL queries.
  + To generate visualization code.

**First** **System Prompt for Data Visualizer (SQL Query Generation)**

|  |
| --- |
| system\_prompt = "You are an expert SQL query generation assistant. Your task is to generate SQL queries using only data from the \*\*presentation\*\* layer, even though you have access to the entire database schema. Always reference tables and views explicitly with the 'presentation' schema prefix (e.g., presentation.vw\_policy). The query must be optimized, select only relevant columns, include necessary filters or conditions (e.g., `WHERE`, `GROUP BY`, or `ORDER BY`), and be ready for execution without modification. Provide only the SQL query without explanations, comments, or formatting like Markdown. Always align the query with the user's intent, and restrict your queries to the \*\*presentation\*\* layer, using the format presentation.<table\_or\_view\_name>." |

**Second** **System Prompt for Data Visualizer (Visualization Code Generation)**

|  |
| --- |
| system\_prompt = """ You are a skilled Python code generation assistant. Generate Python code to visualize data using libraries like Plotly, Altair, or Matplotlib based on the user's request. Do not include any explanations, comments, or formatting such as ```python or code blocks. Provide only the executable Python code without any extra characters or Markdown-like formatting. """ |

**Explanation**

* **SQL Query Prompt:**
  + Ensures that the generated query aligns strictly with the presentation schema.
  + Generates optimized SQL queries without including unnecessary elements like explanations or formatting.
* **Visualization Code Prompt:**
  + Dynamically generates Python code for visualizing the query result.
  + Ensures compatibility with common visualization libraries.
  + The generated code is executed using Python's exec() to render the required visualization.
* **Process Flow:**
  + The first prompt generates the SQL query.
  + The query is executed, and results are fetched.
  + The second prompt generates Python code to visualize the results.
  + The visualization is displayed on the Streamlit interface.

### **Tab 2: Chatbot**

**Purpose**

The Chatbot generates SQL queries using the entire database schema and provides natural language responses based on query results. This functionality caters to user queries that require textual insights instead of visual representations.

**Key Features**

* Broader schema access (entire database).
* Focuses on natural language explanations of query results.
* Utilizes dual prompts:
  + For SQL query generation.
  + For generating natural language responses..

**First System Prompt for Chatbot (SQL Query Generation)**

|  |
| --- |
| system\_prompt = "You are an expert SQL query generation assistant. Your task is to generate SQL queries using only data from the \*\*presentation\*\* layer, even though you have access to the entire database schema. Always reference tables and views explicitly with the 'presentation' schema prefix (e.g., presentation.vw\_policy). The query must be optimized, select only relevant columns, include necessary filters or conditions (e.g., `WHERE`, `GROUP BY`, or `ORDER BY`), and be ready for execution without modification. Provide only the SQL query without explanations, comments, or formatting like Markdown. Always align the query with the user's intent, and restrict your queries to the \*\*presentation\*\* layer, using the format presentation.<table\_or\_view\_name>." |

**Second System Prompt for Chatbot (Natural Language Interpretation)**

|  |
| --- |
| system\_prompt = f""" You are a data wizard that can answer questions based on user input and data from a database. You have access to the entire conversation, including the user's question and the SQL query generated to retrieve the data. You also have the result of the query execution. Here are the details: - User's question: {user\_input} - SQL query generated: {sql\_query} - Result of the query: {result\_str} Use this data to answer the user's question in clear, natural language, providing only the answer with no extra information. """ |

**Comparison of Prompts**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Data Visualizer** | **Chatbot** |
| **Schema Restriction** | Restricted to presentation schema | Uses the entire database schema |
| **Output Format** | SQL query + Visualization Code | SQL query + Natural Language Response |
| **Primary Purpose** | Generate queries for data visualization | Generate queries and interpret results conversationally |
| **Dual Prompts** | 1. SQL query generation  2. Visualization code generation | 1. SQL query generation  2. Natural language output |

## **Database Connection**

The application connects to the Azure Synapse SQL database using pyodbc. The connection parameters are specified in the code and include the server, database, username, and password.

* Server: adro-data-sonic-ondemand.sql.azuresynapse.net
* Database: Datasonic\_Workflow
* Username: sqladminuser
* Password: root@123

|  |
| --- |
| # ESTABLISHING THE SQL SERVER DATABASE CONNECTION-----------------  server = 'adro-data-sonic-ondemand.sql.azuresynapse.net' database = 'Datasonic\_Workflow' username = 'sqladminuser' password = 'root@123'  conn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};SERVER=' +  server + ';DATABASE=' + database + ';UID=' + username + ';PWD=' + password) cursor = conn.cursor() |

## **API Integration & Error Handling**

**API Integration**

Generates SQL queries and visualization code using OpenAI GPT API.

**Process:**

* **API Key Management:** The key is stored in api\_key.txt and loaded dynamically.
* **Prompt Structure:**
  + **SQL Query Generation:** Prompts include schema details and user input for generating efficient SQL.
  + **Visualization Code Generation:** Prompts request executable Python visualization code.
* **API Request:** Uses the OpenAI library to send prompts and retrieve responses.
* **Response Handling:** Extracts SQL queries or Python code for execution.

**Error Handling**

Ensures application stability by managing common API issue like Rate Limit Error (HTTP 429), and displays a message like “Rate Limit Error Occurred: Please try again.”

Following is the code for API Integration & Error Handling:

|  |
| --- |
| # Function for API CALL--------------------  def api\_call(system, user):  # Fetch API Key  with open('api\_key.txt', 'r') as file:  api\_key = file.read().strip()   # Create the OpenAI client  client = OpenAI(api\_key=api\_key)   try:  # Make the API call  completion = client.chat.completions.create(  model="gpt-4o",  messages=[  {"role": "system", "content": system},  {"role": "user", "content": user},  ]  )   # Return the content of the response  return completion.choices[0].message.content  except Exception as e:  # Convert the exception to a string for inspection  error\_message = str(e)   # Check if the error message contains HTTP 429 (Rate Limit Error)  if 'HTTP 429' in error\_message or 'rate\_limit\_exceeded' in error\_message.lower():  st.error(  "Rate Limit Error occurred. Please try again with another prompt.")  else:  st.error(f"An error occurred: {error\_message}")  return None |

**Schema Fetching for Presentation Layer**

The application fetches schema information for both the presentation layer and the entire database. This information is used to generate SQL queries and visualize data.

Presentation Layer Schema Fetching:

* Fetches only the schema named 'presentation'.
* Retrieves tables and views within the 'presentation' schema.
* Collects column names, data types, and lengths for each table and view.

Following is the code for fetching presentation schema:

|  |
| --- |
| # SCHEMA FETCHING FOR PRESENTATION LAYER ---------------------------------------------------------- presentation\_schema\_dict = {}  # Fetch only the presentation schema cursor.execute(  "SELECT SCHEMA\_NAME FROM INFORMATION\_SCHEMA.SCHEMATA WHERE SCHEMA\_NAME = 'presentation'") schema = cursor.fetchone() if schema:  schema\_name = schema[0]   presentation\_schema\_dict[schema\_name] = {"tables": {}, "views": {}}   cursor.execute(  f"SELECT TABLE\_NAME, TABLE\_TYPE FROM INFORMATION\_SCHEMA.TABLES WHERE TABLE\_SCHEMA = '{schema\_name}'")  tables\_views = cursor.fetchall()   for table\_view in tables\_views:  table\_view\_name = table\_view[0]  table\_view\_type = table\_view[1]   cursor.execute(f"""  SELECT COLUMN\_NAME, DATA\_TYPE, COALESCE(CHARACTER\_MAXIMUM\_LENGTH, '') AS LENGTH  FROM INFORMATION\_SCHEMA.COLUMNS  WHERE TABLE\_NAME = '{table\_view\_name}' AND TABLE\_SCHEMA = '{schema\_name}'  """)  columns = cursor.fetchall()   columns\_list = [  f"{col[0]}:{col[1]}{f'({col[2]})' if col[2] else ''}" for col in columns]   if table\_view\_type == 'BASE TABLE':  presentation\_schema\_dict[schema\_name]["tables"][table\_view\_name] = columns\_list  elif table\_view\_type == 'VIEW':  presentation\_schema\_dict[schema\_name]["views"][table\_view\_name] = columns\_list  schema\_str\_presentation = json.dumps(presentation\_schema\_dict, indent=2) |

**Schema Fetching of the Entire Database**

Entire Database Schema Fetching:

* Fetches all schemas in the database.
* Retrieves tables and views for each schema.
* Collects column names, data types, and lengths for each table and view.

Following is the code for fetching entire database:

|  |
| --- |
| # SCHEMA FETCHING OF THE ENTIRE DATABASE schema\_entire\_db = {}  cursor.execute("SELECT SCHEMA\_NAME FROM INFORMATION\_SCHEMA.SCHEMATA") schemas = cursor.fetchall()  for schema in schemas:  schema\_name = schema[0]   schema\_entire\_db[schema\_name] = {"tables": {}, "views": {}}   cursor.execute(  f"SELECT TABLE\_NAME, TABLE\_TYPE FROM INFORMATION\_SCHEMA.TABLES WHERE TABLE\_SCHEMA = '{schema\_name}'")  tables\_views = cursor.fetchall()   for table\_view in tables\_views:  table\_view\_name = table\_view[0]  table\_view\_type = table\_view[1]   cursor.execute(f"""  SELECT COLUMN\_NAME, DATA\_TYPE, COALESCE(CHARACTER\_MAXIMUM\_LENGTH, '') AS LENGTH  FROM INFORMATION\_SCHEMA.COLUMNS  WHERE TABLE\_NAME = '{table\_view\_name}' AND TABLE\_SCHEMA = '{schema\_name}'  """)  columns = cursor.fetchall()   columns\_list = [  f"{col[0]}:{col[1]}{f'({col[2]})' if col[2] else ''}" for col in columns]   if table\_view\_type == 'BASE TABLE':  schema\_entire\_db[schema\_name]["tables"][table\_view\_name] = columns\_list  elif table\_view\_type == 'VIEW':  schema\_entire\_db[schema\_name]["views"][table\_view\_name] = columns\_list  schema\_str\_entire\_db = json.dumps(schema\_entire\_db, indent=2) |

## 

## **Database Schema**

* The database\_schema.txt file will be generated after fetching the schema. It contains detailed schema information for various tables and views in the database.
* The database comprises multiple schemas, including Presentation Schema, that contains views like vw\_claim, vw\_policy, vw\_person\_details, and vw\_vehicle, and other Schemas that Includes raw, confirmed, and system-defined schemas like dbo.

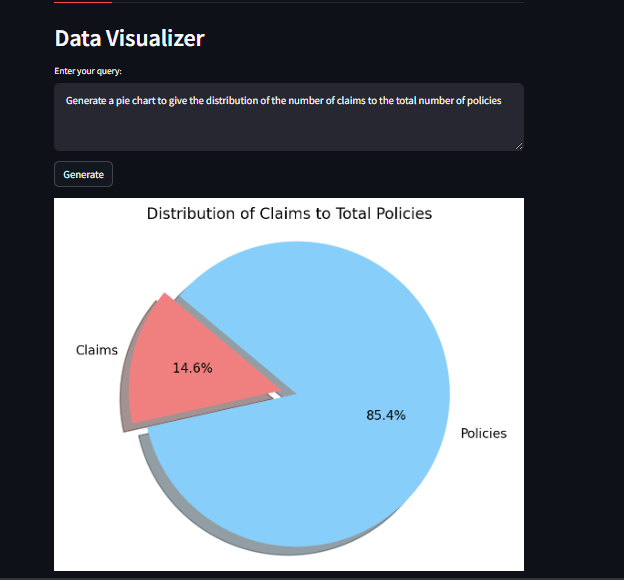
## 

## **Streamlit UI - Data Visualizer & Chatbot**

The Streamlit UI is divided into two tabs:

1. **Data Visualizer:** Allows users to input queries and generate visualizations.
2. **Chat-Bot:** Provides a chat interface for user interaction.

### **Data Visualizer**

****

* In Data Visualizer, unlike chatbot, we’re using the presentation layer schema.
* The application generates SQL queries using OpenAI's API.
* The results are visualized using Python libraries like Plotly, Altair, or Matplotlib.

**Data Visualizer Tab**

The Data Visualizer Tab is responsible for generating SQL queries and visualizations based on user input. It is designed to handle structured queries that retrieve data from the presentation schema of the database and generate interactive plots or charts.

**Step-by-Step Workflow**

1. **User Input:**

|  |
| --- |
| user\_input = st.text\_area("Enter your query:", "", key="header\_tab1") generate\_button = st.button("Generate", key="button\_tab1") |

* **Purpose:** Provides a text area for the user to input queries or requests for data visualization.
* **Interaction:** When the user clicks the "Generate" button, the application starts the SQL generation process

.

1. **SQL Query Generation:**

|  |
| --- |
| system\_prompt = "You are an expert SQL query generation assistant. ..." user\_prompt = user\_input + "The schema for the presentation layer is: " + schema\_str\_presentation sql\_query = api\_call(system=system\_prompt, user=user\_prompt) |

* **System Prompt:** Directs the GPT API to generate a SQL query specific to the presentation schema.
* **User Prompt:** Combines the user query and presentation schema details into a single structured request.
* **API Call:** Sends the prompt to the GPT API and retrieves the SQL query.

1. **Executing the SQL Query:**

|  |
| --- |
| if sql\_query is not None:  cursor.execute(sql\_query)  result = cursor.fetchall() |

* **Execution:** The generated SQL query is executed against the database using the pyodbc cursor.
* **Result:** The query output is fetched as a list of rows.

1. **Visualization Code Generation:**

|  |
| --- |
| result\_str = str(result) system\_prompt = """ You are a skilled Python code generation assistant. Generate Python code to visualize data using libraries like Plotly, Altair, or Matplotlib. ... """ user\_prompt = user\_input + "The data you need to visualize is as follows: " + result\_str visualization\_code = api\_call(system=system\_prompt, user=user\_prompt) |

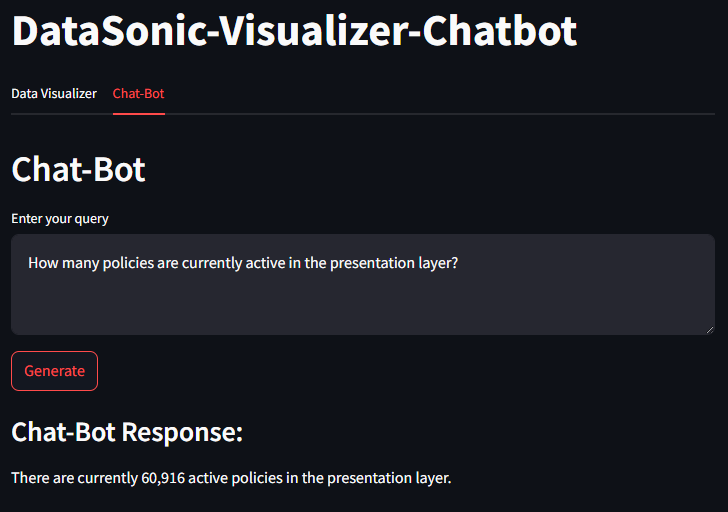
* System Prompt: Instruct the GPT API to generate Python code for visualizing the query results.
* User Prompt: Includes the user query and the query output (formatted as a string).

1. **Visualization Execution:**

|  |
| --- |
| if visualization\_code is not None:  plot\_area = st.empty()  exec(visualization\_code) |

* **Execution:** The Python code generated by the GPT API is executed dynamically using exec().
* **Visualization Display:** The resulting visualization (e.g., bar chart, pie chart) is rendered in the Streamlit UI.

### **CHATBOT**

****

* Users can interact with the chatbot to ask questions and get responses in the natural language.
* In the Chatbot, unlike visualizer, we’re using the entire database schema.
* The chatbot generates SQL queries and retrieves data to answer user questions.

**Chatbot Tab**

The Chatbot Tab provides conversational interaction, allowing users to ask questions about the data. It generates SQL queries using the entire database schema and interprets the query results into natural language responses.

**Step-by-Step Workflow**

1. **User Input:**

|  |
| --- |
| user\_input = st.text\_area("Enter your query", "", key="header\_tab2") generate\_button = st.button("Generate", key="button\_tab2") |

* **Purpose:** Offers a text area for the user to input natural language queries about the data.
* **Interaction:** Clicking the "Generate" button triggers the SQL generation and natural language interpretation process.

1. **SQL Query Generation:**

|  |
| --- |
| system\_prompt = "You are an expert SQL query generation assistant. ..." user\_prompt = user\_input + "The entire schema present in the database is as follows: " + schema\_str\_entire\_db sql\_query = api\_call(system=system\_prompt, user=user\_prompt) |

* **System Prompt:** Directs the GPT API to generate SQL queries using the entire database schema.
* **User Prompt:** Combines the user's natural language query and schema details.
* **API Call:** Sends the prompt to the GPT API and retrieves the SQL query.

1. **Executing the SQL Query:**

|  |
| --- |
| if sql\_query is not None:  cursor.execute(sql\_query)  result = cursor.fetchall() |

* **Execution:** The generated SQL query is executed against the database.
* **Result:** The query output is retrieved as a list of rows.

1. **Natural Language Interpretation:**

|  |
| --- |
| system\_prompt = f""" You are a data wizard that can answer questions based on user input and data from a database. ... """ user\_prompt = user\_input chatbot\_response = api\_call(system=system\_prompt, user=user\_prompt) |

* **System Prompt:** Instructs the GPT API to interpret the query results and provide a natural language explanation.
* **User Prompt:** Includes the user’s question and the query result.

1. **Displaying the Response:**

|  |
| --- |
| st.subheader("Chat-Bot Response:") st.write(chatbot\_response) |

* **Output:** Displays the natural language response from the GPT API in the Streamlit UI.

## **Closing the connection**

Ensure the database connection is closed when the app is closed:

|  |
| --- |
| # Close the connection when the app is closed if 'connection' in locals() and conn.is\_connected():  conn.close()  cursor.close() |