AI-Powered Chatbot for Conversational SQL with Python

This project is an AI-powered chatbot that enables natural language interaction with SQL databases. Users can ask questions in plain English and receive answers derived from the underlying SQL tables, making data exploration and querying accessible—even for those with no knowledge of SQL.

The system leverages powerful natural language processing (NLP) to translate user queries into SQL commands, executes them securely against the database, and returns human-readable responses. This helps businesses, analysts, and developers interact with their data intuitively.

Features Natural Language to SQL: Converts user queries from conversational English to SQL.

Secure Database Access: Safely interacts with SQL databases (like MySQL, PostgreSQL, or SQLite).

Interactive Chat Interface: Users chat in real time and receive immediate answers.

Customizable: Easily connect any SQL database or extend NLP capabilities.

Error Handling: Graceful handling of invalid queries or database errors.

Key Libraries Used

Purpose Library Language Model (NLP)

OpenAI (or Hugging Face Transformers, LangChain, or similar) SQL Parsing & Execution

SQLAlchemy (or sqlite3, psycopg2, etc.) Chat Interface (Web)

Streamlit, Gradio, or Flask Environment Management

python-dotenv (to manage secrets/keys) Data Processing

Pandas (optional; for formatting results) Prompt Engineering / Chains LangChain (if used for chaining models/tools)

Let's first create 100 unique customer names using faker

```
!pip install faker
import sqlite3
from faker import Faker
import random
from datetime import timedelta

from faker import Faker

# Initialize Faker with Indian English locale
fake = Faker('en_IN')

unique_names = set()

# Generate 100 unique Indian names
while len(unique_names) < 100:
    name = fake.name()
    unique_names.add(name)

# Convert to a sorted list for neat printing
unique_names_list = sorted(unique_names)

# Print only the first 10 names
for i, name in enumerate(unique_names_list[:10], start=1):
    print(f"{i}. {name}")
```

```
Collecting faker
  Downloading faker-37.5.3-py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: tzdata in /usr/local/lib/python3.11/dist-packages (from faker) (20
Downloading faker-37.5.3-py3-none-any.whl (1.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 17.5 MB/s eta 0:00:00
Installing collected packages: faker
Successfully installed faker-37.5.3
1. Aarnav Devan
2. Abhiram Bala
3. Arin Deol
4. Balhaar Bera
5. Balvan Toor
6. Barkha Gour
```

```
7. Barkha Kamdar
8. Bhanumati Ghosh
9. Bimala Ben
10. Bina Mohan
```

The next task is to create generative data for an ecommerce company.For this project I am creating only one column, more star schema columns can be created or any other database can be used to run this AI assisted chatbot.We will insert more common columns in this database in the next code.

```
import random
from faker import Faker
from datetime import timedelta

# Initialize Faker with Indian English locale
fake = Faker('en_IN')

# Assume unique_names_list already exists from previous step, e.g.:
# unique_names_list = [...]  # 100 unique Indian English names

# Example for demonstration only - remove this if you have your own list
# unique_names_list = sorted(set([fake.name() for _ in range(100)]))
# create a list for products, later we will randomly insert values for each customer
products = [
    'Smartphone', 'Laptop', 'Bluetooth Headphone', 'Smartwatch',
    'TV', 'Refrigerator', 'Microwave Oven', 'Washing Machine',
    'Air Conditioner', 'Tablet', 'Camera', 'Gaming Console', 'Speaker'
]
# add any method of your choice
payment_methods = ['UPI', 'Credit Card', 'Debit Card', 'COD', 'Net Banking', 'Wallet']
# add any method of your choice
order_statuses = ['Delivered', 'In Transit', 'Cancelled', 'Returned', 'Processing']
# add any method of your choice
gst_rates = [5.0, 12.0, 18.0, 28.0]
# add any method of your choice
couriers = ['Delhivery', 'FedEx', 'Shadowfax', 'Bluedart', 'Ecom Express', 'DHL']

customer_orders = []
#idx is a variable that holds the current index number of the loop iteration.
for idx, customer_name in enumerate(unique_names_list, start=1):
    product = random.choice(products)
    price = round(random.uniform(500, 60000), 2)
    gst = random.choice(gst_rates)
    quantity = random.randint(1, 5)#max quantity set to 5
    total_price = round((price * quantity) + (price * quantity * gst / 100), 2)
    payment_method = random.choice(payment_methods)
    order_date_obj = fake.date_between(start_date='-2y', end_date='today')
    delivery_days = random.randint(2, 10)
    delivery_date_obj = order_date_obj + timedelta(days=delivery_days)
    order_status = random.choice(order_statuses)
    returnable = random.choice([True, False])
    city = fake.city()
    state = fake.state()
    address = f"{fake.street_address()}, {city}, {state}, {fake.postcode()}"
    courier = random.choice(couriers)

    customer_orders.append({
        'customer_id': idx,
        'customer_name': customer_name,
        'product': product,
        'price': price,
        'gst': gst,
        'quantity': quantity,
        'total_price': total_price,
        'payment_method': payment_method,
        'order_date': order_date_obj.strftime('%Y-%m-%d'),#upper case Y for full year li
        'delivery_date': delivery_date_obj.strftime('%Y-%m-%d'),
        'order_status': order_status,
        'returnable': returnable,
        'city': city,
        'state': state,
        'address': address,
        'courier': courier
    })

# Print 10 sample records
```

```
for record in customer_orders[:10]:
    print(record)
```

```
{'customer_id': 1, 'customer_name': 'Aarnav Devan', 'product': 'Washing Machine', 'price': 54364.
{'customer_id': 2, 'customer_name': 'Abhiram Bala', 'product': 'Gaming Console', 'price': 43194.4
{'customer_id': 3, 'customer_name': 'Arin Deol', 'product': 'Tablet', 'price': 54803.9, 'gst': 12
{'customer_id': 4, 'customer_name': 'Balhaar Bera', 'product': 'Air Conditioner', 'price': 18902.
{'customer_id': 5, 'customer_name': 'Balvan Toor', 'product': 'Smartwatch', 'price': 4948.36, 'gs
{'customer_id': 6, 'customer_name': 'Barkha Gour', 'product': 'TV', 'price': 46630.91, 'gst': 28.
{'customer_id': 7, 'customer_name': 'Barkha Kamdar', 'product': 'Speaker', 'price': 48279.66, 'gs
{'customer_id': 8, 'customer_name': 'Bhanumati Ghosh', 'product': 'Tablet', 'price': 20471.3, 'gs
{'customer_id': 9, 'customer_name': 'Bimala Ben', 'product': 'Smartphone', 'price': 12876.87, 'gs
{'customer_id': 10, 'customer_name': 'Bina Mohan', 'product': 'Microwave Oven', 'price': 58503.92
```

The next process is to simple create a sql datatabse and store all the columns into the database

```
import sqlite3

# Connect to (or create) SQLite database named 'ecommerce_data.db'
connection = sqlite3.connect('ecommerce_data.db')
cursor = connection.cursor()

# Create table 'orders' with all relevant columns
cursor.execute('''
CREATE TABLE IF NOT EXISTS orders (
    customer_id INTEGER PRIMARY KEY,
    customer_name TEXT UNIQUE,
    product TEXT,
    price REAL,
    gst REAL,
    quantity INTEGER,
    total_price REAL,
    payment_method TEXT,
    order_date TEXT,
    delivery_date TEXT,
    order_status TEXT,
    returnable BOOLEAN,
    city TEXT,
    state TEXT,
    address TEXT,
    courier TEXT
)
''')

# Prepare list of tuples for insertion (to match the table structure)
records_to_insert = [
    (
        order['customer_id'],
        order['customer_name'],
        order['product'],
        order['price'],
        order['gst'],
        order['quantity'],
        order['total_price'],
        order['payment_method'],
        order['order_date'],
        order['delivery_date'],
        order['order_status'],
        int(order['returnable']),  # SQLite does not have a native BOOLEAN type; 0/1 use
        order['city'],
        order['state'],
        order['address'],
        order['courier']
    )
    for order in customer_orders
]

# Insert data into the table
cursor.executemany('''
INSERT OR IGNORE INTO orders (
    customer_id, customer_name, product, price, gst, quantity, total_price,
    payment_method, order_date, delivery_date, order_status, returnable,
    city, state, address, courier
) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
''', records_to_insert)

# Commit and close connection
```

```
connection.commit()
connection.close()

print("Data saved successfully into 'ecommerce_data.db' SQLite database.")
```

```
Data saved successfully into 'ecommerce_data.db' SQLite database.
```

What are these ? symbols (placeholders)? Each ? is a parameter placeholder used in SQL statements when working with SQLite in Python.

They mark positions where actual data values will be safely inserted at runtime.

The number of ? matches the number of columns listed inside INSERT INTO orders(...).

Example: For the first tuple in records_to_insert, SQLite replaces the first ? with the first value, the second ? with the second value, and so on.

Summary: The ? in the VALUES clause are placeholders indicating where actual data values will be inserted securely and efficiently when the statement runs.

Summary:

Used Faker('en_IN') to generate English names typical to India (like "Rahul Kumar", "Priya Singh").

All necessary columns are included.

Random but realistic data for each column.

Addresses generated by faker (may include line breaks replaced by commas).

Unique customer names ensured.

```
Start coding or generate with AI.
```

```
!pip install langchain-community
```

```
Collecting langchain-community
  Downloading langchain_community-0.3.27-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: langchain-core<1.0.0,>=0.3.66 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: langchain<1.0.0,>=0.3.26 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: SQLAlchemy<3,>=1.4 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: requests<3,>=2 in /usr/local/lib/python3.11/dist-packages (from 1
Requirement already satisfied: PyYAML>=5.3 in /usr/local/lib/python3.11/dist-packages (from lang
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: tenacity!=8.4.0,<10,>=8.1.0 in /usr/local/lib/python3.11/dist-pac
Collecting dataclasses-json<0.7,>=0.5.7 (from langchain-community)
  Downloading dataclasses_json-0.6.7-py3-none-any.whl.metadata (25 kB)
Collecting pydantic-settings<3.0.0,>=2.4.0 (from langchain-community)
  Downloading pydantic_settings-2.10.1-py3-none-any.whl.metadata (3.4 kB)
Requirement already satisfied: langsmith>=0.1.125 in /usr/local/lib/python3.11/dist-packages (fr
Collecting httpx-sse<1.0.0,>=0.4.0 (from langchain-community)
  Downloading httpx_sse-0.4.1-py3-none-any.whl.metadata (9.4 kB)
Requirement already satisfied: numpy>=1.26.2 in /usr/local/lib/python3.11/dist-packages (from la
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: aiosignal>=1.4.0 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-packages (from ai
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11/dist-packages (fro
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11/dist-packages (fro
Collecting marshmallow<4.0.0,>=3.18.0 (from dataclasses-json<0.7,>=0.5.7->langchain-community)
  Downloading marshmallow-3.26.1-py3-none-any.whl.metadata (7.3 kB)
Collecting typing-inspect<1,>=0.4.0 (from dataclasses-json<0.7,>=0.5.7->langchain-community)
  Downloading typing_inspect-0.9.0-py3-none-any.whl.metadata (1.5 kB)
Requirement already satisfied: langchain-text-splitters<1.0.0,>=0.3.9 in /usr/local/lib/python3
Requirement already satisfied: pydantic<3.0.0,>=2.7.4 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: jsonpatch<2.0,>=1.33 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: typing-extensions>=4.7 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: packaging>=23.2 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: orjson<4.0.0,>=3.9.14 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: requests-toolbelt<2.0.0,>=1.0.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: zstandard<0.24.0,>=0.23.0 in /usr/local/lib/python3.11/dist-packa
Collecting python-dotenv>=0.21.0 (from pydantic-settings<3.0.0,>=2.4.0->langchain-community)
  Downloading python_dotenv-1.1.1-py3-none-any.whl.metadata (24 kB)
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from req
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: greenlet>=1 in /usr/local/lib/python3.11/dist-packages (from SQLA
Requirement already satisfied: anyio in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.11/dist-packages (from ht
Requirement already satisfied: h11>=0.16 in /usr/local/lib/python3.11/dist-packages (from httpcc
```

```
Requirement already satisfied: jsonpointer>=1.9 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.11/dist-packages
Collecting mypy-extensions>=0.3.0 (from typing-inspect<1,>=0.4.0->dataclasses-json<0.7,>=0.5.7-
  Downloading mypy_extensions-1.1.0-py3-none-any.whl.metadata (1.1 kB)
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.11/dist-packages (from any
Downloading langchain_community-0.3.27-py3-none-any.whl (2.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.5/2.5 MB 32.6 MB/s eta 0:00:00
Downloading dataclasses_json-0.6.7-py3-none-any.whl (28 kB)
```

How Gemini Translates Text to SQL Queries in Your Project

In your project, Google Gemini acts as a large language model (LLM) “brain” that receives a prompt combining two things:

The schema of your orders table in the ecommerce database (all column names and types).

The user’s question written in plain English.

How it works:

When a user enters a question (like “What is the total GST collected for all orders delivered in 2025?”), your Python code builds a structured prompt for Gemini: It provides the schema and asks Gemini to “translate this question into a precise and safe SQL SELECT query, outputting only the SQL.”

Gemini analyzes the schema and the natural language question using its pre-trained knowledge of both English and SQL.

It generates an appropriate SQL query as text

The code presents this SQL query to the user, who can then copy and execute it on their own database for the actual data results.

In short: Gemini uses advanced natural language and code understanding to “map” your English request to valid SQL that precisely matches your table structure—making database analysis easy for anyone, regardless of SQL expertise.

```
!pip install google-generativeai


import google.generativeai as genai

# Configure Gemini API key (replace with your valid key)
genai.configure(api_key="[redacted]")# use your own API key f

# Specify the Gemini model to use (must be exact)
MODEL_NAME = "models/gemini-2.5-pro"

# Your ecommerce database schema as a string, it is important to provide database schema a
TABLE_SCHEMA = """
CREATE TABLE orders (
    customer_id INTEGER PRIMARY KEY,
    customer_name TEXT,
    product TEXT,
    price REAL,
    gst REAL,
    quantity INTEGER,
    total_price REAL,
    payment_method TEXT,
    order_date TEXT,
    delivery_date TEXT,
    order_status TEXT,
    returnable BOOLEAN,
    city TEXT,
    state TEXT,
    address TEXT,
    courier TEXT
);
"""

def generate_sql_from_text(natural_language_question):
    prompt = (# prompt=This creates a long string (the “prompt”) by joining several lines
        f"Given the table schema:\n{TABLE_SCHEMA}\n\n" # Gives the model your actual table
        f"You are an expert SQL assistant.\n\n" #Tells Gemini to act as an expert in SQL,
        f"Translate the following natural language question into a precise and safe SQL SE
        f"Output only the SQL query without explanations.\n\n"#Instructs Gemini to print c
        f"Question: {natural_language_question}\nSQL:" #Inserts the specific user question
    )
```

```
    model = genai.GenerativeModel(MODEL_NAME)
    response = model.generate_content(prompt)
    return response.text.strip()#Takes the returned text(the SQL code),removes extra white

def main():
    print("Welcome to the Text-to-SQL query generator for your ecommerce database.")
    print("Type your question in natural language to get the corresponding SQL query.")
    print("Type 'exit' or 'quit' to stop.\n")

    while True:
        user_input = input("Your question: ").strip()
        if user_input.lower() in ["exit", "quit"]:
            print("Exiting.")
            break
        try:
            sql_query = generate_sql_from_text(user_input)
            print("\nGenerated SQL (copy this query and run it on your ecommerce db):\n")
            print(sql_query)
            print("\n" + "-"*60 + "\n")
        except Exception as e:
            print(f"An error occurred while generating SQL: {e}")

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

```
Requirement already satisfied: google-generativeai in /usr/local/lib/python3.11/dist-packages (0.8
Requirement already satisfied: google-ai-generativelanguage==0.6.15 in /usr/local/lib/python3.11/d
Requirement already satisfied: google-api-core in /usr/local/lib/python3.11/dist-packages (from go
Requirement already satisfied: google-api-python-client in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: google-auth>=2.15.0 in /usr/local/lib/python3.11/dist-packages (fro
Requirement already satisfied: protobuf in /usr/local/lib/python3.11/dist-packages (from google-ge
Requirement already satisfied: pydantic in /usr/local/lib/python3.11/dist-packages (from google-ge
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from google-genera
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: proto-plus<2.0.0dev,>=1.22.3 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: googleapis-common-protos<2.0.0,>=1.56.2 in /usr/local/lib/python3.1
Requirement already satisfied: requests<3.0.0,>=2.18.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.11/dist-packages (from goog
Requirement already satisfied: httplib2<1.0.0,>=0.19.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: google-auth-httplib2<1.0.0,>=0.2.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: uritemplate<5,>=3.0.1 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: grpcio<2.0.0,>=1.33.2 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: grpcio-status<2.0.0,>=1.33.2 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: pyparsing!=3.0.0,!=3.0.1,!=3.0.2,!=3.0.3,<4,>=2.4.2 in /usr/local/l
Requirement already satisfied: pyasn1<0.7.0,>=0.6.1 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from reque
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from
Welcome to the Text-to-SQL query generator for your ecommerce database.
Type your question in natural language to get the corresponding SQL query.
Type 'exit' or 'quit' to stop.

Your question: What is the average delivery time (difference between delivery_date and order_date)

Generated SQL (copy this query and run it on your ecommerce db):

```sql
SELECT AVG(julianday(delivery_date) - julianday(order_date)) FROM orders;
```

Your question: What is the total GST collected for all orders delivered in 2025?

Generated SQL (copy this query and run it on your ecommerce db):

```sql
SELECT SUM(gst) FROM orders WHERE STRFTIME('%Y', delivery_date) = '2025'
```

Your question: List the customers who have placed more than 3 orders.
An error occurred while generating SQL: Invalid operation: The `response.text` quick accessor requ
Your question: List the customers who have placed more than 3 orders.

Generated SQL (copy this query and run it on your ecommerce db):

```sql
SELECT customer_name
FROM orders
GROUP BY customer_name
HAVING COUNT(*) > 3;
```

Your question: Show total sales grouped by city.

Generated SQL (copy this query and run it on your ecommerce db):

```sql
SELECT city, SUM(total_price) AS total_sales
FROM orders
GROUP BY city;
```

Your question: Which cities have more than 10 orders?
An error occurred while generating SQL: Invalid operation: The `response.text` quick accessor requ
Your question: Which cities have more than 10 orders?

Generated SQL (copy this query and run it on your ecommerce db):

```sql
SELECT city FROM orders GROUP BY city HAVING COUNT(*) > 10;
```

Your question: exit
```

```
Exiting.
```

```
while True:
    user_input = input("Your question: ").strip()
    if user_input.lower() in ["exit", "quit"]:
        print("Exiting.")
        break
try:
        sql_query = generate_sql_from_text(user_input)
        print("\nGenerated SQL (copy this query and run it on your ecommerce db):\n")
        print(sql_query)
        print("\n" + "-"*60 + "\n")
    except Exception as e:
        print(f"An error occurred while generating SQL: {e}")
```

code explanation:

What does this code do? It's like a friendly robot that sits and waits for you to ask questions. When you type a question, it listens and thinks really hard, then writes a special SQL sentence (a code sentence for talking to a database) that answers your question. If you type "exit" or "quit," the robot says "Goodbye!" and stops listening.

Step-by-step Explanation

1. It keeps asking you for questions python while True: user_input = input("Your question: ").strip() The code keeps looping (like a merry-go-round) and asks you, "Your question: "

input() waits for whatever you type.

.strip() just erases any extra spaces at the beginning or end.

2. If you want to stop python if user_input.lower() in ["exit", "quit"]:

   ```
   print("Exiting.")
   break
   ```

   If you type exit or quit (even in capital letters), the robot says "Exiting." and jumps off the merry-go-round —it stops the loop.

3. If you ask a question... python try:

   ```
   sql_query = generate_sql_from_text(user_input)
   ```

   The robot tries to make a smart SQL code sentence from what you asked, using the function generate_sql_from_text.

4. It shows you the answer python

   ```
   print("\nGenerated SQL (copy this query and run it on your ecommerce db):\n")
   print(sql_query)
   print("\n" + "-"*60 + "\n")
   ```

   It tells you "Here's your SQL query!" then prints out the SQL code, so you can copy it.

It draws a long line to make things neat.

5. If the robot gets confused python except Exception as e:

   ```
   print(f"An error occurred while generating SQL: {e}")
   ```

   If the robot makes a mistake or has a problem, it won't crash.

Instead, it says "Oops, I had an error!" and tells you what went wrong, so you're not left in the dark.

In Short: The code waits for your question.

It turns your words into SQL code.

If you want to stop, type "exit" or "quit".

If something goes wrong, it tells you—very politely!

It's like a friendly helper that understands your questions and writes computer code for you!

```
Start coding or generate with AI.
```

Here are some sample natural language questions you can use with your text-to-SQL tool for your ecommerce database:

Sales & Product Insights Which product has the highest total sales?

What are the top 5 most sold products?

How many units were sold for the product 'Smartphone'?

List all products that were delivered in Mumbai.

Which product has the highest average price?

Customers & Orders How many unique customers are there?

Show all customers from Delhi who used Credit Card as payment.

List the customers who have placed more than 3 orders.

How many orders were returned?

Which customers have returnable orders in Bangalore?

Revenue & Transactions What is the total revenue for July 2024?

Show total sales grouped by city.

What is the average order value (total_price) per customer?

Count the number of orders that used UPI as payment.

What is the total GST collected for all orders delivered in 2025?

Delivery & Logistics Which courier has delivered the maximum number of orders?

List all orders that are 'In Transit'.

How many orders are in 'Processing' status?

What is the average delivery time (difference between delivery_date and order_date)?

Miscellaneous List all orders with price greater than 20,000 INR.

Count the number of orders made in each state.

Which cities have more than 10 orders?

Show all orders that are not returnable.

Feel free to ask any of these, or modify them as needed for your business use case!

Summary: Using Gemini for Text-to-SQL on a Star Schema Database

When using Gemini to translate natural language to SQL on a star schema database (common in data warehousing and analytics), the process is similar to what you did for a single-table case—but with richer relationships:

How does it work?

A star schema has a central "fact" table (e.g., sales) surrounded by several "dimension" tables (e.g., products, customers, dates, stores). Each dimension table links to the fact table via a key.

You provide Gemini with the full schema: list all fact and dimension tables and specify how they are joined, including primary and foreign keys.