LangChain SQL AI-Agent for AdventureWorks Sales Schema

## **Overview**

The SQL AI Agent is an intelligent system that translates natural language queries into optimized SQL queries for Microsoft SQL Server. It utilizes Google Gemini (gemini-2.0-flash-thinking-exp) as the LLM, sqlglot for SQL validation, and fuzzywuzzy for approximate text matching to extract relevant schema components dynamically. The agent ensures efficient query generation, validation, and execution, with built-in logging for monitoring.

## **Tools Used**

* Python
* Django
* LangChain
* Fuzzywuzzy
* MSSQL
* Gemini API
* Docker (Containerization)
* Ngrok

## **Key Components**

### **LangChain**

* + A **prompt template (sql\_prompt)** is defined, instructing Gemini to generate an SQL query based on user input and the database schema.
  + The sql\_prompt is **piped** (|) into the Gemini model (llm), creating a **LangChain chain** (sql\_chain).
  + run\_sql\_agent() calls this chain with the **user query** and **database schema** as input.

#### Components Used

* PromptTemplate:  
   Defines structured instructions for the AI to generate SQL queries.
* ChatGoogleGenerativeAI:  
   Connects to the **Gemini API** for processing prompts.
* **Pipelining (|)**:  
   LangChain allows us to **chain** a PromptTemplate with an AI model, making SQL generation **direct and modular**.

### **Database Connection**

The agent connects to an SQL Server database using SQLAlchemy. The connection string includes authentication credentials and an ODBC driver specification:

DATABASE\_URL = "mssql+pyodbc://django\_user:1234@host.docker.internal:1433/AdventureWorks?driver=ODBC+Driver+17+for+SQL+Server&TrustServerCertificate=yes"

The connection is tested at startup to ensure availability.

### **Schema Management**

To minimize redundant file access, the schema is loaded into memory and cached globally:

def get\_schema():

global schema\_cache

if schema\_cache is None:

with schema\_lock:

if schema\_cache is None:

with open("D:\sql\_agent\AdventureWorks\_schema.json", "r") as f:

schema\_cache = json.load(f)

return schema\_cache

### **Synonym Expansion**

A predefined synonym mapping helps improve query understanding by replacing words with their database equivalents:

SYNONYM\_MAP = {

"region": "territory",

"orders": "salesorder",

"customer": "person",

}

The function expand\_query\_with\_synonyms() applies these replacements dynamically.

### **Schema Filtering using Fuzzy Matching**

The fuzzywuzzy library is used to identify relevant tables and columns based on the user query in order to filter the schema since it becomes too big to feed to the LLM.

Each table and column name is compared against the query, and those with a match score of 40% or higher are considered relevant:

if table\_match\_score >= 40 or any(score >= 40 for score in column\_match\_scores.values()):

relevant\_tables[table\_name] = columns

Foreign keys between selected tables are also retained for join optimization.

### **LLM-Powered SQL Generation**

The agent uses Google Gemini (gemini-2.0-flash-thinking-exp) to generate SQL queries. The LLM is initialized with:

llm = ChatGoogleGenerativeAI(

model="gemini-2.0-flash-thinking-exp",

temperature=0.1, top\_p=0.1, top\_k=5, verbose=True,

api\_key="YOUR\_API\_KEY"

)

The LLM is prompted with a structured template that includes filtered schema information and query constraints.

### **Query Cleaning and Validation**

Generated queries are cleaned to remove markdown artifacts and non-SQL characters:

query = re.sub(r"sql\\n(.\*?)\\n", r"\1", raw\_query, flags=re.DOTALL)

query = query.replace("`", "").strip()

Validation is performed using sqlglot, which checks for syntax and schema compatibility:

sqlglot.parse\_one(query, dialect="tsql")

### **Query Execution**

Validated SQL queries are executed using SQLAlchemy:

with engine.connect() as connection:

result = connection.execute(text(query))

rows = result.fetchall()

Results are converted to a **JSON-serializable list of dictionaries**:  
column\_names = result.keys()

processed\_rows = [

{

col: (

float(value) if isinstance(value, Decimal) else

value.isoformat() if isinstance(value, datetime) else

value

)

for col, value in zip(column\_names, row)

}

for row in rows

]

* This ensures that:
  + **Decimal values** are converted to **floats**.
  + **Datetime values** are converted to **ISO format**.

### **Logging and Error Handling**

Logs are recorded in sql\_agent.log

Errors in query execution are captured and logged for debugging:

except Exception as e:

logging.error(f"❌ Query execution error: {e}")

## **End-to-End Flow**

1. **User** enters a question (e.g., *"Find top 5 High-Value Customers (Orders > $50,000)"*).
2. **Django** sends it to run\_sql\_agent()which orchestrates the process:
   1. Extract relevant schema based on the user query.
   2. Generate SQL using the LLM.
      1. run\_sql\_agent(user\_query) calls Gemini through LangChain.
      2. Gemini returns an **SQL query**.
3. The query is **cleaned** and **formatted**.
4. Validate SQL before execution.
5. Execute the query through **SQLAlchemy** and return results.
   1. execute\_query(query) sends the SQL query to SQL Server.
   2. If successful, results are **processed** and **returned**.
6. **Django renders results** in query.html.
   1. Results are stored in **session storage**:  
       request.session["query\_result"] = result
   2. The data is passed to query.html and displayed in the UI.

## 

## **Flow Diagram:**

## 

## **Deployment**

The SQL AI Agent can be deployed using Docker and accessed via Ngrok.

**Running the System in Docker**

1. **Build the Docker image:**

docker build -t sql-ai-agent .

1. **Run the Docker container:**  
    docker run -p 8000:8000 sql-ai-agent
2. **Expose Django API using Ngrok:**  
    ngrok http 8000

**Running Ngrok Separately**

If you want Ngrok to run outside of Docker:

1. **Start Django normally:**

python manage.py runserver

1. **Run Ngrok manually:**

ngrok http 8000

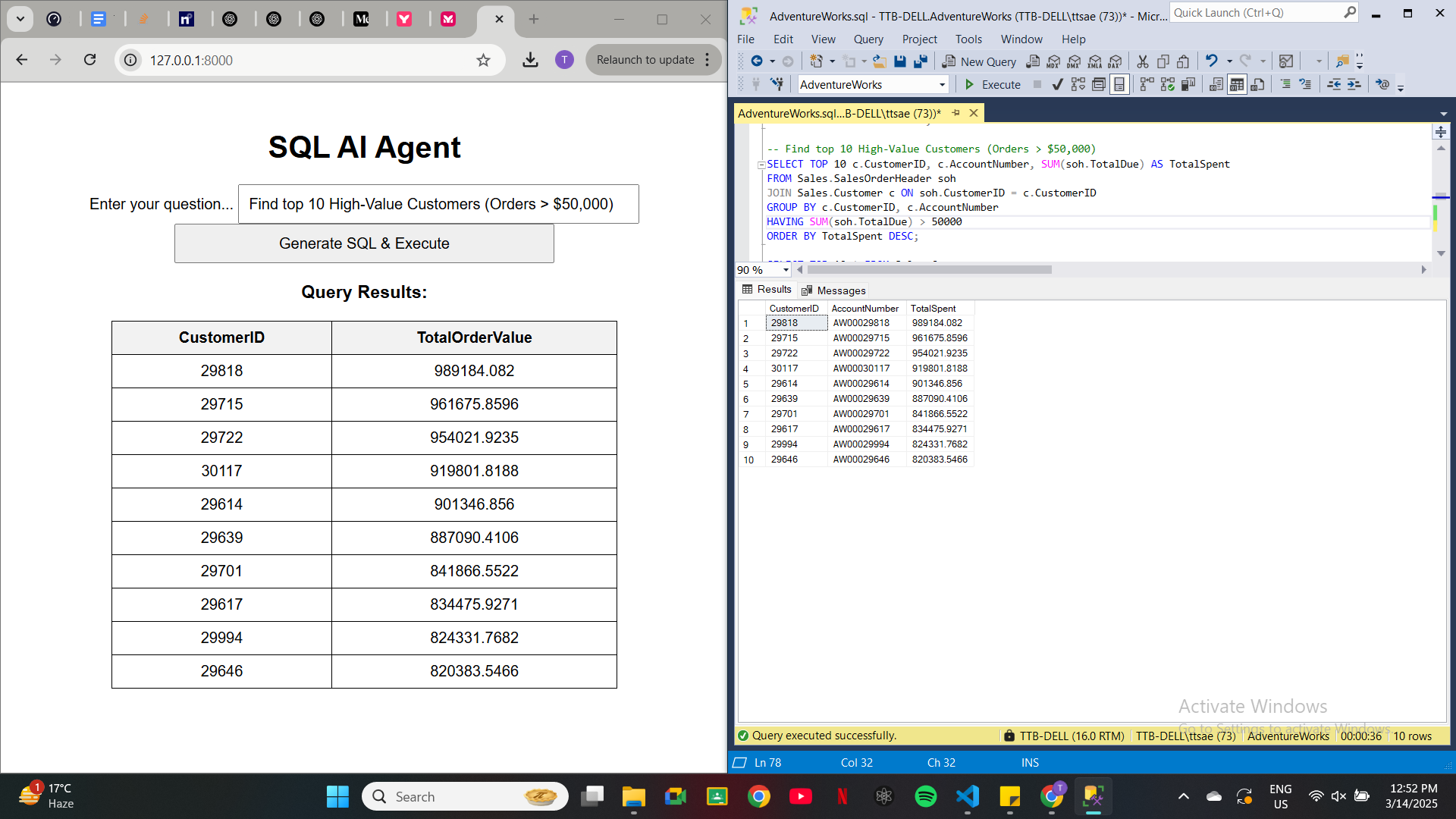
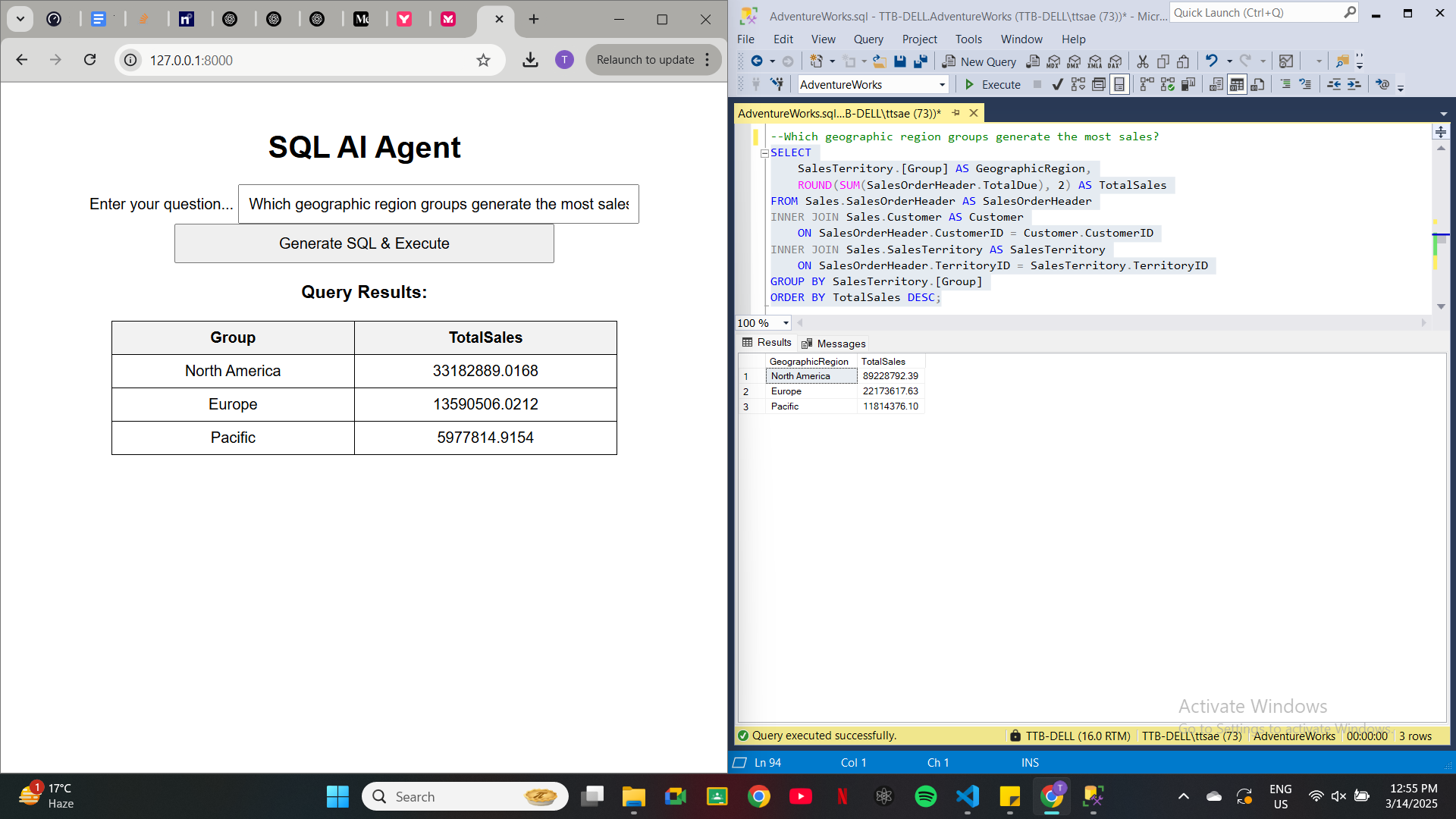
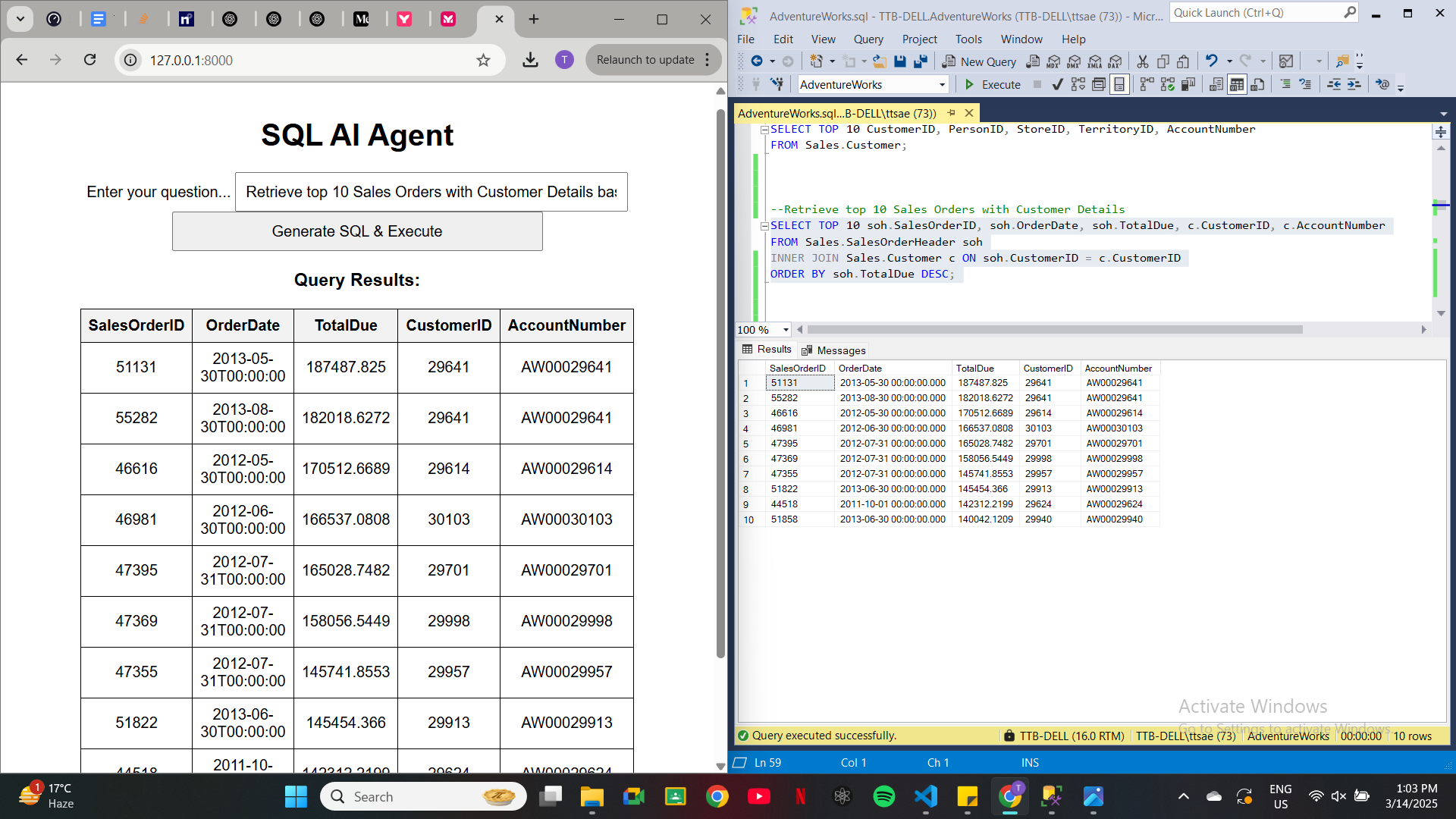
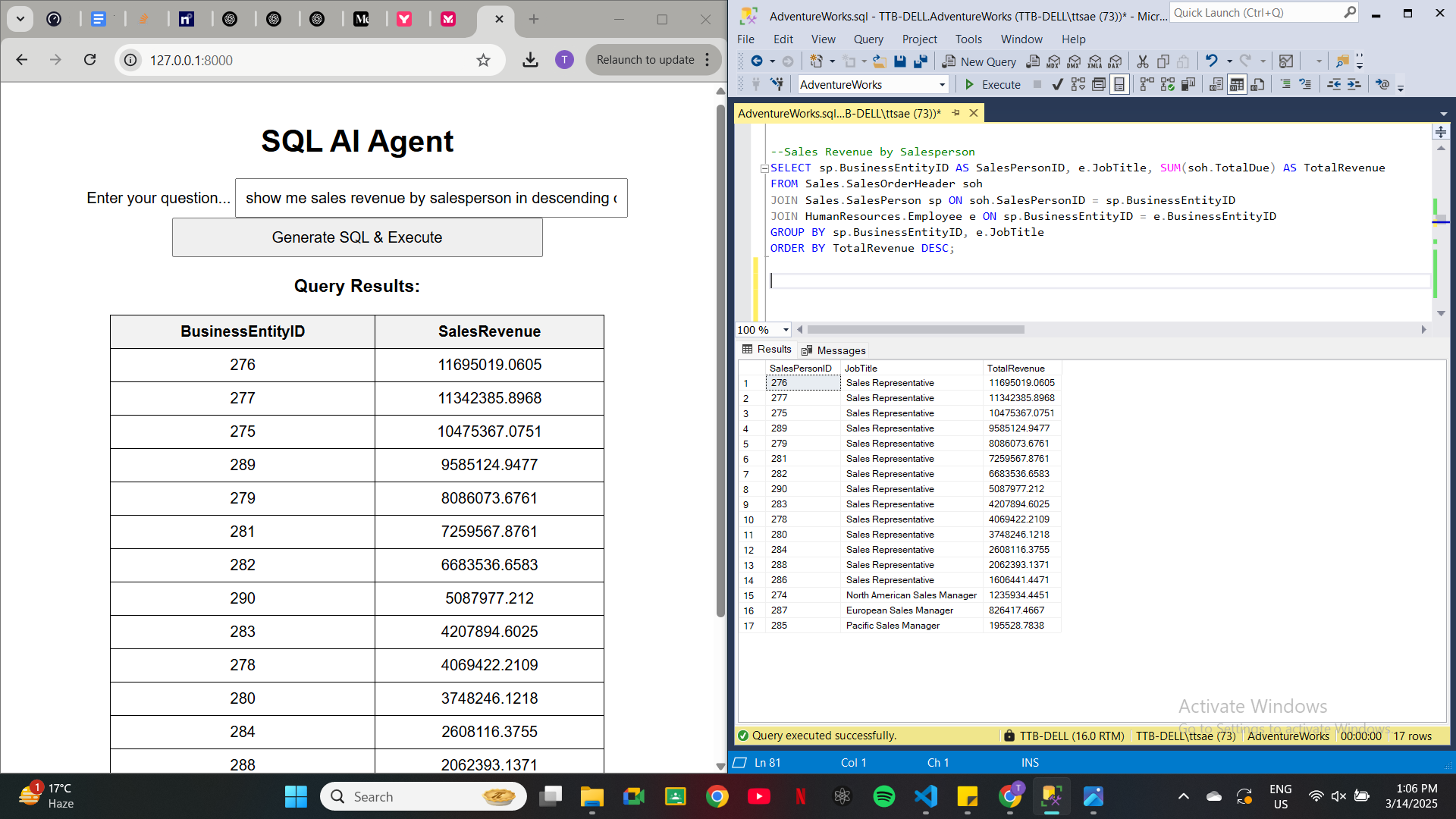
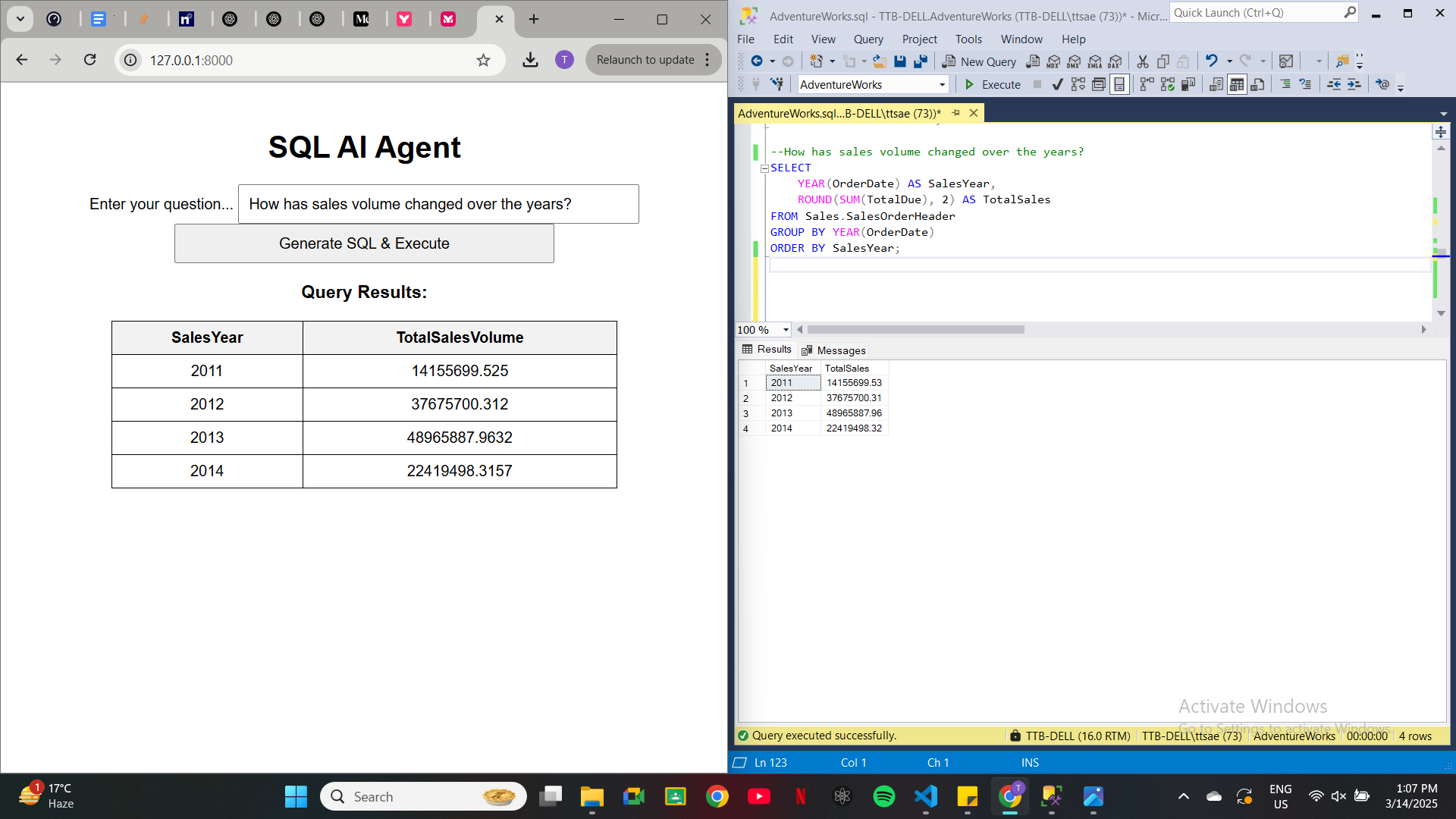
This ensures that only Django is exposed via Ngrok without interfering with Docker.

## **Schema:**

| **Table Name** | **Column Names** |
| --- | --- |
| CountryRegionCurrency | CountryRegionCode, CurrencyCode, ModifiedDate |
| CreditCard | CreditCardID, CardType, CardNumber, ExpMonth, ExpYear, ModifiedDate |
| Currency | CurrencyCode, Name, ModifiedDate |
| CurrencyRate | CurrencyRateID, CurrencyRateDate, FromCurrencyCode, ToCurrencyCode, AverageRate, EndOfDayRate, ModifiedDate |
| Customer | CustomerID, PersonID, StoreID, TerritoryID, AccountNumber, rowguid, ModifiedDate |
| PersonCreditCard | BusinessEntityID, CreditCardID, ModifiedDate |
| SalesOrderDetail | SalesOrderID, SalesOrderDetailID, CarrierTrackingNumber, OrderQty, ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount, LineTotal, rowguid, ModifiedDate |
| SalesOrderHeader | SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNumber, PurchaseOrderNumber, AccountNumber, CustomerID, SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID, ShipMethodID, CreditCardID, CreditCardApprovalCode, CurrencyRateID, SubTotal, TaxAmt, Freight, TotalDue, Comment, rowguid, ModifiedDate |
| SalesOrderHeaderSalesReason | SalesOrderID, SalesReasonID, ModifiedDate |
| SalesPerson | BusinessEntityID, TerritoryID, SalesQuota, Bonus, CommissionPct, SalesYTD, SalesLastYear, rowguid, ModifiedDate |
| SalesPersonQuotaHistory | BusinessEntityID, QuotaDate, SalesQuota, rowguid, ModifiedDate |
| SalesReason | SalesReasonID, Name, ReasonType, ModifiedDate |
| SalesTaxRate | SalesTaxRateID, StateProvinceID, TaxType, TaxRate, Name, rowguid, ModifiedDate |
| SalesTerritory | TerritoryID, Name, CountryRegionCode, Group, SalesYTD, SalesLastYear, CostYTD, CostLastYear, rowguid, ModifiedDate |
| SalesTerritoryHistory | BusinessEntityID, TerritoryID, StartDate, EndDate, rowguid, ModifiedDate |
| ShoppingCartItem | ShoppingCartItemID, ShoppingCartID, Quantity, ProductID, DateCreated, ModifiedDate |
| SpecialOffer | SpecialOfferID, Description, DiscountPct, Type, Category, StartDate, EndDate, MinQty, MaxQty, rowguid, ModifiedDate |
| SpecialOfferProduct | SpecialOfferID, ProductID, rowguid, ModifiedDate |
| Store | BusinessEntityID, Name, SalesPersonID, Demographics, rowguid, ModifiedDate |

## **Cross Validation**

Based on some sql queries from: <https://medium.com/@Splendor001/analyzing-sales-and-customer-data-in-adventureworks2019-using-sql-999c47bc8d80>



## 

## 

## **Conclusion**

This SQL AI Agent leverages LLMs, schema filtering via fuzzy matching, and SQL validation to efficiently generate and execute queries. It optimizes query generation and execution while ensuring safety and correctness through validation mechanisms, decreasing LLM overhead much more than done in previous versions.

## 