ChatGPT

# Architecture

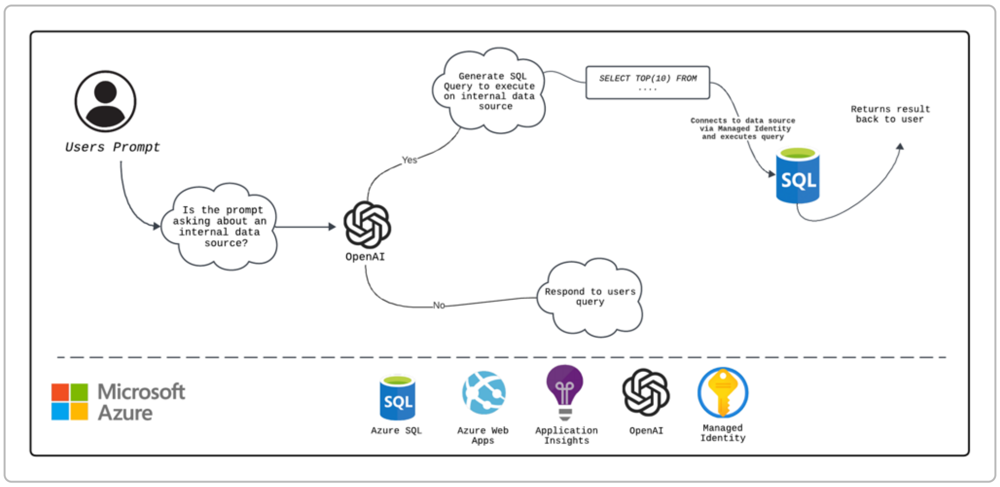
Users Prompt
Is the prompt asking about an internal data source?
OpenAI
Yes
Generate SQL Query to execute on internal data source
SELECT TOP(10) FROM ....
Connects to data source via Managed Identity and executes query
SQL
Returns result back to user
No
Respond to users query
Microsoft Azure
Azure SQL
Azure Web Apps
Application Insights
OpenAI
Managed Identity

Figure: Example architecture of the agentic text-to-SQL system. A FastAPI service accepts the user's natural-language query. The system constructs a prompt that includes the database schema and a few sample rows [1], then calls an LLM (via OpenAI's function-calling API) to generate an SQL query. If the LLM returns an error message, we abort with a friendly fallback; if it returns SQL, we perform a safety check (e.g. allow only `SELECT` statements [2]) and then execute it on PostgreSQL. The resulting data (or any error) is passed to a second LLM call to produce a human-readable summary. Overall the flow follows proven patterns (e.g. an Azure demo "generates a SQL query … executes that query, and relates it back to the user" [3]). In practice, this means:

- **Receive & Authenticate:** FastAPI endpoint (e.g. `/query`) takes the user question, validates an auth token, and fetches that user's credentials (OpenAI API key and Postgres login) from MongoDB.
- **Prompt Construction:** The service introspects the Postgres schema and top-N sample rows to build a system prompt. For example, as in the Azure sample, "the schema information of tables [is] injected as part of the prompt" [1].
- **LLM Function Call:** The prompt (with role=user content) is sent to `gpt-4o` (or similar) with a declared function like `execute_query(query: string)`. The model returns either an SQL query string (as JSON via `function_call`) or an error note.
- **SQL Safety Check:** Before execution we verify the SQL is read-only. For example, we enforce it *starts* with `SELECT` and contains no dangerous clauses (no `DROP`/`DELETE` etc). This enforces least-privilege as recommended by OWASP (an agent "should only need read access… it should not have … insert/update/delete" [4]). The Azure demo similarly "enforce[s] only read queries to the database" [2].
- **Execute & Summarize:** If the SQL passes checks, it is executed on PostgreSQL (using a read-only DB user). The query results are then summarized: we call the LLM again (or the same LLM in a second step) with a prompt like *"Here are the results of your query: [data]. Give a concise answer."* This produces a natural-language answer.

- **Error Handling:** At any failure (LLM returns an error, safety check fails, SQL exec fails), we log the issue and return a safe fallback response.

**Performance note:** For reduced latency/cost, one can consider open-source LLMs for generation or summarization. Recent benchmarks test models like Meta's LLaMA 3 (8B/70B) and Mistral 7B on SQL tasks [5]. These open models can run on local hardware and offer speedups, though GPT-4/Gemini may still be more accurate.

## Component Code Snippets

Below are illustrative code excerpts for key components. (Error handling, logging, and security checks are simplified.)

```
# FastAPI endpoint with JWT auth dependency
@app.post("/query")
async def query_endpoint(q: QueryModel, token: str = Depends(oauth2_scheme)):
    # Authenticate user and fetch their credentials from MongoDB
    user_id = get_user_from_token(token)
    creds = await get_user_config(user_id)  # returns API keys, DB creds
    logger.info(f"Query from {user_id}: {q.text}")

    # Build system prompt: include schema and sample data
    schema_text = introspect_schema(creds.postgres)
    preview_text = fetch_sample_rows(creds.postgres)
    system_prompt = f"Schema:\n{schema_text}\nSample data:\n{preview_text}"

    # Call LLM with function schema for SQL generation
    response = openai.ChatCompletion.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user",   "content": q.text}
        ],
        functions=[{
            "name": "execute_query",
            "description": "Generate a SQL query from the user question (read-
only)",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {"type": "string"}
                },
                "required": ["query"]
            }
        }],
        function_call="execute_query"
    )
    result_msg = response.choices[0].message
    if "function_call" in result_msg:
        data = json.loads(result_msg.function_call.arguments)
        sql_query = data["query"]
    else:
        return {"answer": "Sorry, I could not create a query."}
```

```
# SQL safety check (enforce read-only SELECT)
if not sql_query.strip().upper().startswith("SELECT"):
    logger.warning(f"Rejected non-SELECT query: {sql_query}")
    return {"answer": "Only read-only queries are allowed."}
```

```
# Execute the SQL on PostgreSQL (using a connection from creds)
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine(f"postgresql://{creds.pg_user}:{creds.pg_password}@"
                       f"{creds.pg_host}:{creds.pg_port}/{creds.pg_db}")
try:
    df = pd.read_sql_query(sql_query, engine)  # Fetch results into DataFrame
except Exception as e:
    logger.error(f"SQL execution error: {e}")
    return {"answer": "Error executing the query."}
```

```
# Summarize results with LLM
summary_resp = openai.ChatCompletion.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": f"Summarize the following results:
{df.head().to_json()}" }],
    temperature=0.0
)
answer = summary_resp.choices[0].message.content
return {"answer": answer}
```

Each snippet above uses FastAPI for the API layer, OpenAI's function-calling API for generation, `sqlalchemy`/`pandas` to query Postgres, and Python's `logging`. We attach a logger to track each step. The final response dictionary can be returned as JSON by FastAPI.

## Agent Strategy

Rather than trusting a one-shot query, we embed an **agent loop** that can auto-correct SQL. For example, LangChain's SQL Agent toolkit can handle errors by retrying queries: it "recovers from errors by running a generated query, catching the traceback and regenerating it correctly" [6]. In practice, this means after a failed execution (e.g. wrong column name or type error), we feed the error message back to the LLM to refine the SQL and try again (up to a few retries). Similarly, Hugging Face's SmolAgents cookbook demonstrates using a SQL tool where the agent "critically inspects outputs and decide[s] if the query needs to be changed" [7].

*Example using a SmolAgent-style tool* (pseudocode):

```python
from smolagents import SQLAgent, tool

@tool
def sql_tool(query: str) -> str:
    """Execute the SQL query on the DB and return results or error text."""
    try:
        result = run_query(query)  # custom function to execute and fetch
        return str(result)
    except Exception as e:
        return f"ERROR: {e}"

agent = SQLAgent(llm=ChatOpenAI(model_name="gpt-4o", temperature=0),
                 tools=[sql_tool])
agent_response = agent.run("What is the average price by category?")
```

In this setup, the agent has a *tool* `sql_tool` to run queries. If the generated SQL fails, the agent can see the "ERROR: …" output and prompt the LLM to revise the query. This iterative loop (often guided by few-shot prompts or chain-of-thought) ensures the final SQL is valid and schema-compliant.

## Security Design

To secure the system and prevent SQL injection or abuse:

- **Read-Only Enforcement:** We *only* allow SELECT queries. Any SQL that doesn't start with `SELECT` (or contains forbidden keywords like `DROP`, `DELETE`, `INSERT`) is rejected [2]. The database user we use has *no* write permissions, consistent with OWASP's guidance that an extension should "only need read access to a table; it should not have…insert/update/delete" [4]. Even if the LLM hallucinated a malicious statement, the DB role prevents damage.
- **Query Sanitization:** Before running the SQL, the code does a regex or AST check to ensure it's a single read-only statement. We may strip out extra whitespace, ban semicolons (to prevent stacked queries), and double-check there's no "; DROP" etc.
- **Authentication & Secrets:** All endpoints require an authenticated JWT (using OAuth2 password flow with hashing, as in FastAPI's security tutorial [8]). Database credentials and OpenAI API keys are never hard-coded; they come from the user's profile in MongoDB. Service-level secrets (like MongoDB URI) are passed via environment variables or a secrets manager, not in code.
- **Logging & Monitoring:** We use Python's `logging` to record each query and system action. An audit log (with timestamps and user IDs) makes it possible to review any suspicious inputs. We can also implement rate limiting or anomaly detection to spot injection attempts.

- **Secondary Checks:** Optionally, we can run the LLM's moderation or content filtering on the user's natural-language input to block obvious malicious prompts (e.g., "delete table"). However, the main defenses are the DB role and the safety-check code.

By combining strict query filtering, least-privilege DB access [4], and careful handling of LLM output, we guard against SQL injection and data leaks.

## MongoDB Integration

User-specific credentials are stored in a MongoDB collection (e.g. `user_configs`). Each document follows a schema like:

```
{
  "customerID": "user123",
  "openai_api_key": "sk-…",
  "postgres": {
    "host": "db.example.com",
    "port": 5432,
    "db": "mydb",
    "user": "readonly_user",
    "password": "secret"
  }
}
```

At login, FastAPI verifies the user's username/password and returns a JWT containing `customerID` (using the OAuth2/JWT pattern described in FastAPI's docs [8]). In the `/query` endpoint, a dependency reads the token to get `customerID`. Then we do something like:

```
client = AsyncIOMotorClient(MONGO_URI)
config = await client.appdb.user_configs.find_one({"customerID": customer_id})
if not config:
    raise HTTPException(404, "User config not found")
creds = ConfigModel(**config)  # Pydantic model for fields
```

Now `creds.openai_api_key` and `creds.postgres` can be used to configure the OpenAI client and PostgreSQL connection dynamically for that user. In this way, different customers (or environments) can use different LLM accounts or databases. The MongoDB access itself should be secured (e.g. use a connection string with credentials in environment vars).

## Dockerization Steps

We containerize the system for deployment:

- **FastAPI App Dockerfile:** Use a slim Python base. For example:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

(This follows FastAPI's recommended Docker setup using an official Python image [9].)

- **PostgreSQL Service:** In `docker-compose.yml`, use the official `postgres:15` image. Set environment variables like `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB`. Mount a Docker volume for data persistence. (Docker Hub provides a trusted PostgreSQL image [10].)

- **MongoDB Service:** Similarly, use the official `mongo:6` image with `MONGO_INITDB_ROOT_USERNAME`, `MONGO_INITDB_ROOT_PASSWORD`. Mount a volume for the database files.

- **Networking and Config:** Define a user network so the FastAPI app can reach `postgres` and `mongo` by service name. Pass secrets/config via a `.env` file or Docker secrets (for production). For example, set `OPENAI_API_KEY`, `MONGO_URI`, and any other global vars in the environment.

- **Observability:** Configure the container to log to stdout/stderr so Docker (or Kubernetes) captures it. Python's `logging` is already used in code for this. Optionally, front the FastAPI container with an NGINX proxy or Traefik for TLS termination and load balancing.

By using Docker Compose, the stack is simple to bring up. All components (FastAPI, Postgres, Mongo) run as separate containers. Environment variables and volumes are declared in `docker-compose.yml`. This isolates services and makes the system easy to deploy and scale.

**References:** The above design is grounded in modern practice. For example, Azure's sample text-to-SQL app embeds table schemas in prompts and enforces read-only queries [1] [2]. LangChain's SQL agents explicitly retry on errors [6]. OWASP guidance on LLM integrations stresses minimal permissions [4]. FastAPI's official docs cover JWT auth [8] and Docker deployment [10] [9]. Together these form the blueprint for a robust, secure, high-performance text2SQL API.

---

[1] [2] [3] GitHub - Azure-Samples/function-call-dynamic-query-demo: FastAPI app that takes a users prompt, transforms it to a SQL query and fetches the result from an Azure SQL Database through an OpenAI function call.
https://github.com/Azure-Samples/function-call-dynamic-query-demo

[4] OWASP Top 10 for LLM Applications 2025 - WorldTech IT
https://wtit.com/blog/2025/04/17/owasp-top-10-for-llm-applications-2025/

[5] Which LLM writes the best analytical SQL?
https://www.tinybird.co/blog-posts/which-llm-writes-the-best-sql

6 Agents | LangChain
https://python.langchain.com/v0.1/docs/use_cases/sql/agents/

7 Agent for text-to-SQL with automatic error correction - Hugging Face Open-Source AI Cookbook
https://huggingface.co/learn/cookbook/en/agent_text_to_sql

8 OAuth2 with Password (and hashing), Bearer with JWT tokens - FastAPI
https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/

9 10 FastAPI in Containers - Docker - FastAPI
https://fastapi.tiangolo.com/deployment/docker/