“Building My Own Local Finance Q&A Agent: A Project-Based Guide for Learning Stocks, SQL, and AI”

As a Computer Science graduate student, I’ve been curious about how AI can help us understand financial data better. So I decided to build a personal Finance Q&A Agent — entirely local — using open-source tools like PostgreSQL, LangChain, CrewAI, and Python. This project helps me dive into financial data, build hands-on ML agent workflows, and understand how stock trends play out over time.

**🧱 1. Why Dockerize PostgreSQL?**

To keep the setup portable and reproducible, I containerized both PostgreSQL and pgAdmin using Docker. This gives me:

* An isolated database environment that won’t clash with other services.
* Persistent volumes to store stock data.
* The ability to share my setup using just a docker-compose.yml file.

**⚙️ 2. Setting Up Docker Compose for Postgres + pgAdmin**

I created a docker-compose.yml file with two services:

* A Postgres container running on port 5433.
* A pgAdmin GUI running on port 5050 for easy DB access.

We exposed the right ports, defined a default user, and created a database called mag7\_data for our stock records.

**🔐 3. Connecting to Postgres via pgAdmin**

Once the containers were running, I connected to Postgres using pgAdmin at localhost:5050. I registered a new server with:

* Host: localhost
* Port: 5433
* DB: mag7\_data
* User: finance\_user
* Password: secret

**🧽 4. Cleaning Historical Stock Data for the MAG7**

I wrote a Python script called clean\_stock\_data.py that:

* Parses raw CSV files for each MAG7 stock (GOOG, AAPL, MSFT, etc.).
* Cleans up date formats, percentage signs, and volume abbreviations (e.g., 645.62M → 645620000).
* Adds a ticker column and reorders the data to match our Postgres schema.
* Appends each cleaned stock file to a master file called mag7\_cleaned.csv.

**📊 5. Verified Schema for Finance Stock Table**

* I designed a mag7 table with proper types for finance data:
* sql
* CopyEdit
* CREATE TABLE mag7 (
* id SERIAL PRIMARY KEY,
* ticker VARCHAR(10),
* date DATE,
* open NUMERIC(10, 4),
* high NUMERIC(10, 4),
* low NUMERIC(10, 4),
* close NUMERIC(10, 4),
* volume BIGINT,
* change\_percent NUMERIC(6, 2)
* );
* This structure ensures price precision and reliable storage of trading volume and percentage change over time.

**📥 Importing CSV into Postgres: Methods I Explored**

There are multiple ways to get a CSV into Postgres. Here's what I learned:

**🖥️ 1. Using pgAdmin’s Import GUI (Beginner-Friendly)**

* Right-click the mag7 table → *Import/Export Data...*
* Point to mag7\_cleaned.csv
* Choose format = CSV, delimiter = ,, and check **Header**
* Make sure to **exclude the id column**, which is auto-generated
* Hit **Import**

This is the **easiest method**, especially if you're using pgAdmin regularly.

**🐚 2. Using \copy from the psql CLI (Scripting-Friendly)**

* psql -h localhost -p 5433 -U finance\_user -d mag7\_data
* Then inside the prompt:

\copy mag7(ticker, date, open, high, low, close, volume, change\_percent)

FROM '/Users/username/Desktop/finance\_agent/mag7\_cleaned.csv'

DELIMITER ',' CSV HEADER;

**🧠 Key Takeaway**

If you're running Postgres in Docker, always use **pgAdmin’s Import UI** or \copy from psql unless you’ve explicitly mounted your local file system into the container.

**🧭 Next Steps: Building Your Finance Agent**

Here’s a high-level roadmap from where you are now:

**✅ Step 1: Connect Python to Your Postgres DB**

Use psycopg2 or SQLAlchemy to query your mag7 table:

* Test out a few SQL queries manually
* Example: get all rows for NVDA in 2023

This validates DB connectivity and query structure.

**🤖 Step 2: Build Your Local Finance Agent with CrewAI + LangChain**

This is where the real fun begins.

**Agent Components:**

* 🧠 **LangChain**: Handles natural language prompts → SQL conversion
* 🔗 **CrewAI**: Orchestrates multi-agent workflows
* 🗃️ **PostgreSQL (via Python)**: Responds to SQL queries
* 🔍 Optional: Use **Text2SQL** or **semantic embeddings** for more advanced parsing

You’ll likely need:

* A **LangChain tool or agent** that can query Postgres
* A **CrewAI agent** with role like “Stock Research Analyst”

Example question your agent will answer:

*"What was the highest monthly return for Apple in 2023?"*

**🛠️ Step 3: Build an Interface (Optional but Cool)**

* Start with a simple **Streamlit dashboard** for:
  + Asking questions
  + Displaying charts or table results
* Later, integrate this into your existing **finance search app** if you want

**🐍 Connecting Python to Dockerized PostgreSQL**

* I used psycopg2 to connect my Python code to the Postgres container running on localhost:5433.
* Credentials like database name, user, and password were stored securely in a .env file and loaded using python-dotenv.
* A simple test script ran a SELECT COUNT(\*) FROM mag7 to confirm the connection and row count.
* This step validated that the data was accessible for querying and downstream agent integration.

**🧪 [Optional] Creating a Python Virtual Environment**

* To isolate project-specific dependencies, I created a virtual environment using:

python -m venv venv

source venv/bin/activate # Mac/Linux

 Installed required libraries (psycopg2-binary, pandas, python-dotenv) inside the environment.

 Used pip freeze > requirements.txt to save all dependencies.

 This makes it easy for others to recreate the same setup using:

pip install -r requirements.txt

While LangChain's ReAct-based agents can answer questions using Postgres, they often repeat steps like listing tables or inspecting schemas. To reduce this latency and looping:

- Switch to smaller/faster models like `mistral`

- Manually inject schema knowledge

- Prune toolset to only what's needed

- Consider using LangChain's newer LangGraph framework for production agents

**✅ What I’ve Accomplished So Far**

1. **Infrastructure Setup**
   * Dockerized PostgreSQL and pgAdmin
   * Imported cleaned CSVs for MAG7 stocks into a table mag7
   * Connected Python to Postgres via psycopg2
2. **LangChain SQL Agent Integration**
   * Built a LangChain agent using SQLDatabaseChain from langchain-experimental
   * Connected it to my local database and ran test queries
3. **LLM Benchmarking Framework**
   * Created a Python script to benchmark multiple local models
   * Measured **latency**, **response quality**, and **SQL accuracy**
   * Used terminal coloring (termcolor) for clean visual output
4. **Models Tested via Ollama**
   * ✅ **phi3**: Returned valid SQL, clean response
   * ✅ **neural-chat**: Returned a final answer, minimal noise
   * ✅ **codellama**: Generated syntactically valid SQL
   * ❌ **llama3**, **mistral**, **gemma**: Failed due to formatting issues or SQL syntax errors

**🧠 Key Learnings**

* Not all LLMs behave well out of the box for SQL — **prompt design and formatting control are critical**
* Local models like **phi3 and neural-chat** are both fast and effective for structured Q&A tasks
* Model performance varies not just by speed, but **by how cleanly they separate logic from explanation**

**📌 Next Steps**

* Replace deprecated .run() with .invoke() in LangChain
* Tune prompts to avoid markdown/code block noise
* Possibly introduce token tracking and factual accuracy comparison
* Extend benchmarks to multi-question suites