AI Assignment — Retail Analytics Copilot (DSPy + LangGraph)

Overview

Build a local, free AI agent that answers retail analytics questions by combining:

- RAG over local docs (docs/)
- SQL over a local SQLite DB (Northwind)

Produce typed, auditable answers with citations.

Use DSPy to optimize at least one component.

No paid APIs or external calls at inference time.

Estimated time: 2–3 focused hours

Runs on: normal PC (CPU ok), 16GB RAM recommended

Local model constraint: Phi-3.5-mini-instruct via Ollama (or llama.cpp GGUF)

Data & Documents (download + create)

Database (free, tiny, open-source)

Use the Northwind sample database in SQLite format.

Download (one-liner):

```
mkdir -p data
curl -L -o data/northwind.sqlite \

https://raw.githubusercontent.com/jpwhite3/northwind-SQLite3/main/dist/northwind.db
```

(Optional) Create lowercase compatibility views (lets you write simpler SQL):

```
sqlite3 data/northwind.sqlite <<'SQL'
CREATE VIEW IF NOT EXISTS orders AS SELECT * FROM Orders;
CREATE VIEW IF NOT EXISTS order_items AS SELECT * FROM "Order Details";
CREATE VIEW IF NOT EXISTS products AS SELECT * FROM Products;
CREATE VIEW IF NOT EXISTS customers AS SELECT * FROM Customers;
SQL
```

Tables you'll use (canonical Northwind names):

- `Orders(OrderID, CustomerID, EmployeeID, OrderDate, …)`
- `"Order Details"(OrderID, ProductID, UnitPrice, Quantity, Discount)`
- `Products(ProductID, ProductName, SupplierID, CategoryID, UnitPrice, …)`
- `Customers(CustomerID, CompanyName, Country, …)`
- (You may use `Categories`, `Suppliers` as needed.)

Document Corpus (create these four files in `docs/`)

Create a local docs corpus to power RAG:

`docs/marketing_calendar.md`

```
# Northwind Marketing Calendar (1997)

## Summer Beverages 1997
- Dates: 1997-06-01 to 1997-06-30
- Notes: Focus on Beverages and Condiments.

## Winter Classics 1997
- Dates: 1997-12-01 to 1997-12-31
- Notes: Push Dairy Products and Confections for holiday gifting.
```

`docs/kpi_definitions.md`

```
# KPI Definitions
## Average Order Value (AOV)
```

```
- AOV = SUM(UnitPrice * Quantity * (1 - Discount)) / COUNT(DISTINCT OrderID)

## Gross Margin
- GM = SUM((UnitPrice - CostOfGoods) * Quantity * (1 - Discount))
- If cost is missing, approximate with category-level average (document your
approach).
```

docs/catalog.md

```
# Catalog Snapshot
- Categories include Beverages, Condiments, Confections, Dairy Products,
Grains/Cereals, Meat/Poultry, Produce, Seafood.
- Products map to categories as in the Northwind DB.
```

docs/product_policy.md

```
# Returns & Policy
- Perishables (Produce, Seafood, Dairy): 3–7 days.
- Beverages unopened: 14 days; opened: no returns.
- Non-perishables: 30 days.
```

Project Skeleton (create these paths/files)

```
your_project/
├─ agent/
│  ├─ graph_hybrid.py           # your LangGraph (≥6 nodes + repair loop)
│  ├─ dspy_signatures.py        # DSPy Signatures/Modules
(Router/NL→SQL/Synth)
│  ├─ rag/retrieval.py          # TF-IDF or simple retriever (chunking +
search)
│  └─ tools/sqlite_tool.py      # DB access + schema introspection
├─ data/
│  └─ northwind.sqlite          # downloaded DB
├─ docs/
│  ├─ marketing_calendar.md
│  ├─ kpi_definitions.md
│  ├─ catalog.md
│  └─ product_policy.md
├─ sample_questions_hybrid_eval.jsonl
├─ run_agent_hybrid.py          # main entrypoint (CLI contract below)
└─ requirements.txt
```

Provide this exact eval file (`sample_questions_hybrid_eval.jsonl`) with 6 items (don't change ids; you may add more for your own testing):

```
{"id":"rag_policy_beverages_return_days","question":"According to the product policy, what is the return window (days) for unopened Beverages? Return an integer.","format_hint":"int"}
{"id":"hybrid_top_category_qty_summer_1997","question":"During 'Summer Beverages 1997' as defined in the marketing calendar, which product category had the highest total quantity sold? Return {category:str, quantity:int}.","format_hint":"{category:str, quantity:int}"}
{"id":"hybrid_aov_winter_1997","question":"Using the AOV definition from the KPI docs, what was the Average Order Value during 'Winter Classics 1997'? Return a float rounded to 2 decimals.","format_hint":"float"}
{"id":"sql_top3_products_by_revenue_alltime","question":"Top 3 products by total revenue all-time. Revenue uses Order Details: SUM(UnitPrice*Quantity*(1-Discount)). Return list[{product:str, revenue:float}].","format_hint":"list[{product:str, revenue:float}]"}
{"id":"hybrid_revenue_beverages_summer_1997","question":"Total revenue from the 'Beverages' category during 'Summer Beverages 1997' dates. Return a float rounded to 2 decimals.","format_hint":"float"}
{"id":"hybrid_best_customer_margin_1997","question":"Per the KPI definition of gross margin, who was the top customer by gross margin in 1997? Assume CostOfGoods is approximated by 70% of UnitPrice if not available. Return {customer:str, margin:float}.","format_hint":"{customer:str, margin:float}"}
```

You may assume CostOfGoods ≈ `0.7 * UnitPrice` if no cost field exists in Northwind; document any variant in your README.

What to Build

Role

You are the Retail Analytics Copilot for Northwind. Use docs to extract constraints (dates, KPIs, categories). When numbers are required, generate SQL against the local SQLite schema. Always return the exact `format_hint` and include citations to both DB tables and doc chunk IDs. If SQL fails or output shape is wrong, repair up to 2 times.

LangGraph (≥6 nodes, stateful)

Implement a hybrid agent with at least these nodes:

1. **Router** (DSPy classifier encouraged): `rag | sql | hybrid`
2. **Retriever**: top-k doc chunks + scores (include chunk IDs)
3. **Planner**: extract constraints (date ranges, KPI formula, categories/entities)
4. **NL→SQL** (DSPy): generate SQLite queries using live schema (PRAGMA)
5. **Executor**: run SQL; capture `columns`, `rows`, `error`
6. **Synthesizer** (DSPy): produce typed answer matching `format_hint`, with citations
7. **Repair loop (required)**: on SQL error or invalid output/citations, revise up to 2x
8. **Checkpointer/trace**: keep a replayable event log (file/console)

> You have design freedom (extra critics, rerankers, or validators). Keep it local and deterministic.

DSPy Requirement (optimize one module)

Use a DSPy optimizer (e.g., MIPROv2, BootstrapFewShot, Teleprompter) to improve at least one of:

- **Router** (classification accuracy),
- **NL→SQL** (valid-SQL rate or exec success),
- **Synthesizer** (exact format adherence / citation completeness).

Show a before/after metric on a tiny train split (e.g., 20–40 examples or handcrafted). Keep budgets small and local.

CLI (do not change flags)

We will run exactly:

```
python run_agent_hybrid.py \
  --batch sample_questions_hybrid_eval.jsonl \
  --out outputs_hybrid.jsonl
```

Each line in `outputs_hybrid.jsonl` must follow the Output Contract.

Output Contract (per question)

```
{
  "id": "...",
  "final_answer": <matches format_hint>,
  "sql": "<last executed SQL or empty if RAG-only>",
  "confidence": 0.0,
  "explanation": "<= 2 sentences>",
  "citations": [
    "Orders",
    "Order Details",
    "Products",
    "Customers",
    "kpi_definitions::chunk2",
    "marketing_calendar::chunk0"
  ]
}
```

- **`final_answer`**: must match the input `format_hint` exactly (e.g., `int`, `float`, object, or list of objects). Floats graded with ±0.01 tolerance.
- **`citations`**: include every DB table actually used and every doc chunk ID you relied on (e.g., `marketing_calendar::chunk0`).

Acceptance Criteria & Scoring

- **Correctness (40%)** — values match expected (±0.01 for floats), and types match `format_hint`.
- **DSPy impact (20%)** — a measurable improvement on the chosen module (brief table/notes).
- **Resilience (20%)** — a repair/validation loop that actually helps (e.g., raises valid-SQL or format-adherence rate).
- **Clarity (20%)** — readable code, short README, sensible confidence, proper citations & trace.

Implementation Hints

Retrieval

- You can implement TF-IDF (no downloads) or BM25 (e.g., `rank-bm25`) over paragraph-level chunks.
- Keep chunks small; store `id`, `content`, `source` (filename), `score`.

SQL

- Prefer `Orders` + `"Order Details"` + `Products` joins.
- Revenue: `SUM(UnitPrice * Quantity * (1 - Discount))` from `"Order Details"`.
- If needed, map categories via `Categories` join through `Products.CategoryID`.

Confidence

- Heuristics are fine: combine retrieval score coverage + SQL success + non-empty rows; down-weight when repaired.

Deliverables

1. **Code** in `agent/` (your graph, DSPy modules, retriever, tools).
2. **README.md** (short), containing:
    - 2–4 bullets: your graph design
    - Which DSPy module you optimized + metric delta (before → after)
    - Any trade-offs or assumptions (e.g., CostOfGoods approximation)
3. **`outputs_hybrid.jsonl`** generated by the CLI above.

Please share the Github link.

Setup

requirements.txt (example, pin or tweak as you like):

```
dspy-ai>=2.4.0
langgraph>=0.1.0
langchain-core>=0.2.0
pydantic>=2.0.0
click>=8.1.7
rich>=13.7.0
numpy>=1.26.0
pandas>=2.2.0
scikit-learn>=1.3.0
rank-bm25>=0.2.2  # optional
```

Local model (choose one):

```
# Ollama (recommended)
# Install from https://ollama.com, then:
ollama pull phi3.5:3.8b-mini-instruct-q4_K_M
```

(llama.cpp + GGUF of Phi-3.5-mini-instruct is also acceptable.)

Notes & Constraints

- No external network calls at inference. Download the DB once; keep everything local.
- Keep prompts compact (≤1k tokens total is plenty).
- Bound repair to ≤2 iterations.

Good luck — build something you'd be happy to ship locally!