AI Assignment — Retail Analytics Copilot (DSPy + LangGraph) — Technical Task Report

Author: ***Yousef Roshdy Mohamed Fathalla Abdou***

Date: ***27/11/2025***

---

1. Overview

This project implements a **hybrid agent** capable of answering analytical questions over the Northwind dataset by combining:

1. **SQL grounding** (for database-answerable questions), and
2. **RAG (Retrieval-Augmented Generation)** (for policy and document-based answers).

The system follows the assignment specification:

- Accept a batch of questions in JSONL format {id, question, format_hint}
- Decide dynamically whether each question requires:
  - SQL-only
  - RAG-only
  - Hybrid (retrieval + SQL + synthesis)
- Produce outputs matching the exact contract:

```
{
 "id": "...",
 "final_answer": <matches format_hint>,
 "sql": "<last executed SQL>",
 "confidence": float,
 "explanation": "... <= 2 sentences",
 "citations": [...]
}
```

The execution pipeline uses **LangGraph** for control flow, **DSPy modules** for robust LLM prompting, and **Ollama + Phi-3.5 (3.8B Q4_K_M)** for local inference.

---

2. System Architecture

Graph Design

The system is structured as a LangGraph pipeline consisting of:

- **Router Module**
  Determines route: "sql", "rag", or "hybrid".
- **Retriever Module**
  Retrieves relevant text chunks from:
  - Product policy documents
  - Marketing calendar
  - KPI definitions
  - Catalog Snapshot
- **NL2SQL Module**
  Generates SQL based on:
  - Question
  - Retrieved context (if hybrid)
  - Schema hints
- **SQL Execution Module**
  Executes SQL against northwind.db and returns rows or errors.
- **Synthesizer Module**
  Combines:
  - SQL rows
  - Retrieved text chunks
  - Question + format hint
    Produces final answer + explanation + citations.
- **Repair Loop**
  If SQL or synthesis fails, the system retries with:
  - Max retries: **2 attempts**
  - If still failing → exit gracefully with partial output.

---

3. Documentation Retrieval (RAG)

TF-IDF Retriever

Implemented in: `agent/rag/retrieval.py`

Features:

- Loads `.md` files under `./docs`
- Split each document into paragraph-level chunks
- Compute TF-IDF embeddings
- Provides `retrieve_topk(query, k)` returning:
    - text chunk
    - document source
    - relevance score
    - chunk identifier

---

4. SQL Execution Layer

SQLite Tool

Implemented in: `agent/tools/sqlite_tool.py`

Functions:

- `execute_sql(sql: str) -> List[Dict[str,Any]]`
- Safe SQL execution (no writes)
- Converts results into row dictionaries
- Reflects schema for the NL2SQL module

---

5. DSPy Modules

All signatures are defined in `agent/dspy_signatures.py`.

5.1 RouterModule

- Inputs: {question, format_hint}
- Output: route ∈ {"rag", "sql", "hybrid"}

5.2 NL2SQLModule

- Converts natural language to SQL queries
- Conditioned on:
    - question
    - schema description
    - retrieved chunks
    - JSON-encoded constraints

5.3 SynthModule

- Inputs:
    - question
    - retrieved_chunks
    - sql_result
    - constraints_json
- Outputs:
    - final_answer (typed)
    - explanation (≤2 sentences)
    - citations
    - confidence

The synthesizer performs:

- Format coercion
- Citation generation

---

6. LangGraph Execution

Implemented in `graph_hybrid.py`.

Graph nodes:

- router — Uses RouterModule to select a route
- retriever — Gets TF-IDF top-k chunks
- nl2sql — Generates SQL query via DSPy
- sql_exec — Executes SQL, captures errors
- synthesizer — Produces final answer
- repair — Applies retry logic

---

7. Repair Loop (Error Handling)

The system includes a bounded repair loop:

Error Type	Action
SQL syntax error	Regenerate SQL
Wrong output format	Ask SynthModule to rewrite
Missing citation	Retry synthesis

If still failing → exit gracefully with partial output. Max retries: 2 attempts

---

8. Logging & Tracing

Every node logs a JSON snapshot via `log_step("node_name", state)`.

Logged fields include:

- task id
- route
- attempts
- sql_query
- sql_error
- synth_error

This results in a full trace of the agent's reasoning and retry behavior.

---

9. Local Model Integration (Ollama + Phi-3.5)

- **Ollama** for local model serving
- **Phi-3.5** Q4_K_M quantization for fast CPU inference

---

10. Limitations and Tradeoffs

The system works end-to-end but has notable constraints:

Model Limitations

- Phi-3.5 Mini is not strong at schema-aware SQL generation.
- Frequent hallucinations of tables, columns, and JOIN paths.
- Weak multi-step reasoning for hybrid tasks.

RAG Limitations

- TF-IDF is lexical only, missing semantic matches.
- Irrelevant chunks sometimes contaminate NL→SQL reasoning.

---

11. Suggested Improvement

To make SQL generation robust and reduce hallucinations:

1. **Use a Specialized SQL Model or Fine-tune Phi-3.5**
    - Fine-tune on Northwind text-to-SQL examples.
    - Or use a chat-to-database model designed for SQL reliability.
2. **Replace TF-IDF With an Embedding Retriever**
   MiniLM or BGE-small would reduce irrelevant retrieval and improve consistency.
3. **BERT-based SQL Task Classifier**
   Predicts the report type (e.g., revenue ranking, AOV, margin analysis).
   Guides NL→SQL into template-safe generation.
4. **Entity Tagging (NER)**
   Extracts date, category, metric, group-by field, date window, etc.
   Provides structured context for SQL generation.

---

12. Conclusion

This project successfully implements a complete hybrid analytics agent using RAG + SQL, DSPy modules, LangGraph routing, and a structured synthesis pipeline. The system operates fully locally via Ollama. While the current model and retriever introduce SQL reliability issues, the architecture is modular and ready for enhancements such as BERT-based classifier, entity extraction, semantic retrieval, and stronger SQL-capable models.