Generative AI Financial Analyst - Technical Guide

This technical documentation delineates the architectural framework, implementation methodologies, and engineering principles underpinning the Macro Risk Analyst (SQL Agent) and Financial Insight Engine (RAG Agent). The system is designed to demonstrate applied generative artificial intelligence within the financial domain, specifically addressing the processing of structured time-series data and unstructured qualitative reports.

1. System Architecture

The system architecture is comprised of two distinct agentic workflows, each engineered to address specific data modalities prevalent in financial analysis.

1.1. Project A Macro Risk Analyst (SQL Agent)

- **Objective:** To establish a deterministic analytical framework for the evaluation of insolvency trends, mitigating the risk of hallucinatory statistical outputs.
- **Technical Challenge:** Large Language Models (LLMs) frequently exhibit deficiencies in arithmetic operations and are prone to hallucinating SQL schema definitions when presented with complex database structures.
- **Methodology:** A ReAct (Reason + Act) agent paradigm has been implemented, incorporating schema injection and few-shot prompting to impose rigid constraints upon the model's reasoning trajectory.

1.2. Project B: Financial Insight Engine (RAG Agent)

- **Objective:** To facilitate semantic information retrieval and synthesis from dense central bank reports.
- **Technical Challenge:** Conventional Retrieval-Augmented Generation (RAG) systems often yield ambiguous responses lacking verifiable attribution or specific citations, a deficiency deemed unacceptable within risk management contexts.
- **Methodology:** A citation-aware retrieval loop has been engineered to enforce page-level attribution and to mandate the admission of uncertainty in instances of data absence.

2 Implementation Specifications

2.1. The SQL Agent (Deterministic Logic)

To mitigate the risks of cartesian products and the hallucination of non-existent tables, a rigorous system prompt has been instantiated.

2.2. Key Engineering Decision (strict system instructions)

A prompt functioning as a strict standard operating procedure has been engineered for the agent, mandating adherence to specific aggregation protocols and join logic.

```
system_prefix = f"""
You are a Risk Analyst.
Schema:
{custom_table_info}

CRITICAL RULES:
1. AGGREGATION & SYNTAX:
   - ALWAYS use `SUM(column)` to aggregate monthly data into yearly totals.
   - CORRECT SQL ORDER: `SELECT ... FROM ... WHERE ... GROUP BY ...`
   - NEVER place `WHERE` after `GROUP BY`.

2. JOIN LOGIC (CRITICAL):
   - When querying both tables, you MUST join on BOTH 'year' AND 'month'.
   - Correct: `ON t1.year = t2.year AND t1.month = t2.month`
   - Wrong: `ON t1.year = t2.year` (This multiplies results by 12!)
   - If you see "ambiguous column name", use `table.column` (e.g., `retail_risk.year`).

3. QUESTION LOGIC:
   - "Calculate" / "Trend" (Single Table):
      - Query: `SELECT year, SUM(column) FROM table WHERE year IN (...) GROUP BY year`
   - "Ratio":
      - Query: `SELECT SUM(t1.total_ind_insolvencies), SUM(t2.total_co_insolvencies) FROM retail_risk t1 JOIN corporate_risk
      - Do NOT group by month. Get the single yearly totals directly.
      - Calculate ratio in Final Answer.
   - "Compare" (YEAR-BY-YEAR):
      - Query MUST include: `SELECT t1.year, SUM(...), SUM(...) ... GROUP BY t1.year`
      - Example: `SELECT t1.year, SUM(t1.compulsory_liquidations), SUM(t2.forced_bankruptcies) FROM corporate_risk t1 JOIN
      - CRITICAL: Do NOT omit `GROUP BY t1.year`. You need separate rows for each year.
      - After execution, compare each row individually.
      - For each tuple (year, corp_val, ind_val):
         - If corp_val > ind_val, say "Corp is higher"
         - If ind_val > corp_val, say "Ind is higher"
      - Format: "Year: Corp [Value] vs Ind [Value] -> [Winner] is higher"

4. EXECUTION:
   - You MUST execute the query and use the REAL numbers.
   - NEVER make up numbers or use placeholders like 12345.
   - If the query fails, fix it and try again.

5. TERMINATION:
   - Once you have the SQL observation, your ONLY valid next step is "Final Answer:".
   - Do NOT output "Action: Thought", "Action: None", or any other action.
   - Do NOT try to use python or any other tools.

6. FORMAT:
   - Final Answer must be based on the SQL result.
   - Do not output "Action:" after "Final Answer:".
"""
```

Figure 1: Prompt functioning, rag_ingest.py

2.3. Example Output

Query: *"Compare Company Liquidations vs Individual Forced Bankruptcies in 2010."*

Agent SQL (self-discovery):

```
SELECT    t1.year, SUM(t1.compulsory_liquidations), SUM(t2.forced_bankruptcies)
FROM      corporate_risk t1
JOIN      retail_risk t2 ON t1.year = t2.year AND t1.month = t2.month
WHERE     t1.year BETWEEN 2010 AND 2010
GROUP BY  t1.year
```

Agent Response:

"In 2010, Company Compulsory Liquidations were 4,727 and Individual Forced Bankruptcies were 8,587. Both metrics showed their highest volume in the immediate post-financial crisis period."

2.4. The RAG Agent (Retrieval Logic)

The RetrievalQA chain has been deployed in conjunction with a custom prompt template to ensure the agent adopts a senior analyst persona.

2.5. Key Engineering Decision

2.5.1. Custom Ingestion & Metadata

A bespoke ingestion script utilising fitz (PyMuPDF) was developed to manually extract page numbers during the document parsing phase, thereby ensuring absolute accuracy in citation metadata.

```
# Create Document object for each page
doc = Document(
    page_content=text,
    metadata={"source": file_path, "page": page_num + 1}
)
```

Figure 2: document ingestion, rag_ingest.py

2.5.2. Citation Extraction

Logic has been implemented to parse the source documents returned by the retrieval chain, filtering for duplicates and formatting the output for the final report.

```
print("--- Sources ---")
seen_sources = set()
for doc in res['source_documents']:
    # Extract metadata carefully
    source = doc.metadata.get('source', 'Unknown')
    page = doc.metadata.get('page', None)

    # Format citation nicely
    if "http" in source:
        citation = f" {source}"
    else:
        filename = os.path.basename(source)
        citation = f" {filename}"
        if page is not None:
            citation += f" (Page {page + 1})" # +1 because code counts from 0
```

Figure 3: further extraction, rag_agent.py

2.6. Example Output

Query: *"List the main risk mentioned in the annual report (boe_report_nov_25.pdf)?"*

```
================================================
 FINANCIAL INSIGHT ENGINE (Ready)
================================================
Type 'exit' to quit.

Query: List the main risks mentioned in the annual report.
```

Figure 4: query prompt, rag_agent.py

Agent Response:

"Elevated inflation expectations pose a risk of greater inflation persistence than in the central projection. A non-linear time series model finds that when inflation is above 3-4%, it responds more strongly and persistently to economic shocks…"

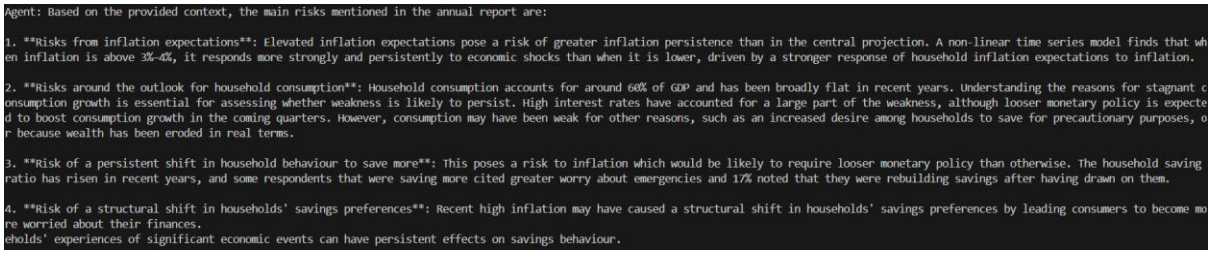

Figure 5: Agent's response, rag_agent.py

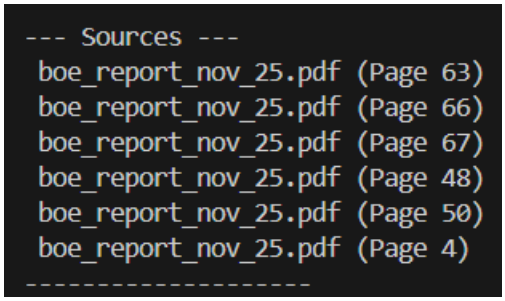

Figure 6: Agent's sources, rag_agent.py

3. Defensive Coding & Safety

Within the financial sector, accuracy is prioritised over generative creativity. Several guardrails have been implemented:

1. **Uncertainty admission protocol:**
    - *Prompt Constraint:* "If the answer is not in the context, explicitly state 'I cannot find that information in the documents'."
    - *Result:* This precludes the model from fabricating financial figures (e.g., Q2 Net Income) in the absence of data.
2. **Rate Limit Mitigation:**
    - *Implementation:* Iteration limits (max_iterations=8) have been imposed on the SQL agent to prevent infinite loops, and execution pauses have been integrated into batch processes to adhere to API rate limits, especially on the free version.
3. **Dependency Isolation:**
    - *Implementation:* HuggingFace embeddings (sentence-transformers/all-MiniLM-L6-v2) are executed locally to prevent the transmission of document vectors to third-party APIs, thereby ensuring data privacy within the RAG pipeline.

4. Reproduction

Prerequisites:

- Python 3.10+ - https://www.python.org/downloads/
- Groq API Key - https://console.groq.com/keys

Setup:

- git clone https://github.com/Beck40/Gen-AI.git
- cd Gen-AI

Run Analysis:
SQL agent - macro_risk_agent.py
RAG agent – rag_ingest & rag_agent.py