

Banking Insights Using a Text2SQL Agentic AI System

Agentic AI System to Answer Retail Banking Questions

Overview

In this capstone project, you will act as an AI engineer on a retail banking analytics problem. Build a **Text2SQL AI agent** that converts natural-language questions into **safe, read-only SQLite** queries over a compact, banking database (6 tables) and returns concise, grounded answers.

You will enable insights across customers, accounts, merchants, branches, and disputes while enforcing strict guardrails (no writes, minimal columns, default row limits).

Tasks

- **Agent:** Implement an AI agent **using one of the following frameworks - LangGraph, AutoGen, or CrewAI**, that can leverage tools to list tables, inspect relevant schemas, validate SQL, execute read-only queries, and summarize results clearly. **The LLM powering your Agent should be OpenAI GPT-4.1-mini**
- **Evaluation:** Run the agent on **10 predefined test banking questions** (e.g., merchant spend, balances, branch activity, disputes) and you will be evaluated based on your agent's answers to those queries as well as your code for the agentic workflow

Banking Insights Text2SQL Challenge

The retail banking sector increasingly relies on data to drive customer insights, risk controls, and operational decisions. Turning ad-hoc business questions into accurate, auditable answers requires consistent access to transaction data, balances, and dispute records. However, self-serve analytics is hard when users must hand-craft SQL across multiple tables and join paths.

Traditional dashboards and rule-based query templates struggle with evolving questions and complex joins. Modern **agentic Text2SQL** systems can ground questions in schema context, synthesize correct **read-only SQLite** queries, and return concise, trustworthy answers.

In this capstone project, you will design and implement a Text2SQL agent using a realistic, INR-denominated banking dataset (SQLite, 6 tables: **Customer**, **Branch**, **Account**, **Merchant**, **Transactions**, **Dispute** with **AmountINR**). You will:

- **Translate natural-language questions** into syntactically correct, **read-only** SQLite that targets only the required columns and rows.
- **Enforce guardrails**: no writes (INSERT/UPDATE/DELETE/DDL), no **SELECT ***, and SQL syntax checking before execution.
- **Execute queries and summarize results** as clear, grounded answers with minimal tables and brief explanations.
- **Evaluate your agent on 10 predefined banking questions** (merchant spend, balances, branch activity, disputes, inactivity, ticket size) and verify against provided ground truths.
- **Iterate on agent strategy** (e.g., schema probing, few-shot examples, ReAct planning, error-aware retries) to improve robustness and accuracy.

This project applies an end-to-end **agentic Natural Language (NL)→SQL workflow** on structured banking data to solve practical analytics tasks without manual SQL writing.

Learning Objectives

This capstone will help you apply agentic NL→SQL techniques to a practical banking analytics problem.

By the end, you will be able to:

- Build an end-to-end **Text2SQL agent** over SQLite using either **LangGraph or AutoGen or CrewAI**
- Orchestrate safe tool use: **list tables → inspect schema → validate SQL → execute query (read-only)**.
- Translate natural-language questions into **minimal, qualified SELECTs** (no **SELECT ***)
- Enforce guardrails: **no writes** (no INSERT/UPDATE/DELETE/DDL), only required columns, and conservative filters.
- Write and execute correct **SQL queries** using **your AI Agent**
- Summarize results into **concise, grounded answers** (small tabular snippets + brief explanation; no SQL in final message).
- **Evaluate your agent on 2 predefined banking questions** and compare outputs to ground-truth answers

- **Generate agent responses on 10 predefined banking test questions** and create your submission CSV file

(we will evaluate your code and how your agent performs on the 10 test questions).

- Prepare submission artifacts:
 - `code.ipynb` — your agent implementation and tests
 - `submission.csv` — answers for the 10 test questions (`query`, `response`)

Solution Requirements & Specifications

Database Overview

This capstone uses a compact **SQLite** database tailored for **retail banking Text2SQL**. All monetary values are stored directly in **Indian Rupees (INR)** via the `AmountINR` column. The agent must generate **read-only** SQL (no writes), select only necessary columns.

- **Entities (6):** `Customer`, `Branch`, `Account`, `Merchant`, `Transactions`, `Dispute`
- **Common use cases:** merchant analytics, balances and activity, channel/ticket-size analysis, dispute reporting, customer inactivity

Datasets

File name	Description
<code>banking_insights.db</code>	SQLite DB. Six tables with realistic transactional data; <code>Transactions</code> has <code>AmountINR</code> .
<code>sample_queries_with_responses.csv</code>	CSV file with 2 sample natural language queries and their corresponding expected results from a working Text2SQL Agent
<code>test_queries.csv</code>	CSV file with 10 test questions which you need to run on your Text2SQL Agent to submit responses

Schema Summary — Entities & Relationships

- **Customer** → Person who owns accounts; location attributes used for city/region rollups.
- **Branch** → Physical branch that services accounts.
- **Account** → Deposit account (Checking/Savings) tied to a **Customer** and a **Branch**.
- **Merchant** → Counterparty for POS/online spend; includes generic entries (Employer Payroll, Bank Transfer In/Out, ATM Withdrawal, Bank Fees).
- **Transactions** → All money movement with **AmountINR**, **TxnType** (Credit/Debit), **Channel**, and **MerchantID** on every row.
- **Dispute** → Case record linked to a specific transaction.

Join paths (typical):

- Customer activity/balances: **Customer** → **Account** → **Transactions**
- Merchant analytics: **Transactions** → **Merchant**
- Branch rollups: **Account** → **Branch**
- Disputes: **Dispute** → **Transactions** → **Merchant**

Database Schema Diagram:

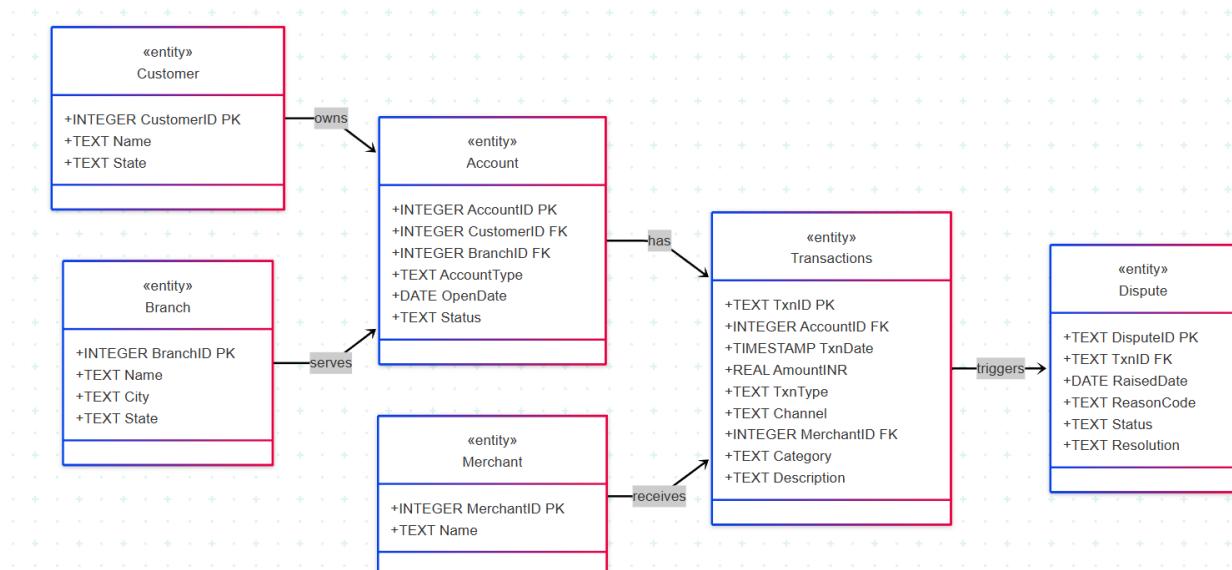


Table Details

1) Customer Schema

Column	Type	Description
CustomerID	INTEGER	PK. Unique customer identifier.
Name	TEXT	Full name.
State	TEXT	State/UT code (e.g., MH, KA, DL).

Purpose: Customer dimension for ownership, segmentation, and geo rollups.

Customer (example)

CustomerID	Name	State
1	Advait Das	DL
2	Kabir Gupta	MH
3	Diya Bose	KA

2) Branch Schema

Column	Type	Description
BranchID	INTEGER	PK. Unique branch identifier.
Name	TEXT	Branch name (e.g., "Mumbai Fort").
City	TEXT	City where the branch is located.
State	TEXT	State/UT code.

Purpose: Branch dimension for operational rollups.

Branch (example)

BranchID	Name	City	State
1	Mumbai Fort	Mumbai	MH
2	Bengaluru MG Road	Bengaluru	KA
3	Delhi CP	New Delhi	DL

3) Account Schema

Column	Type	Description
AccountID	INTEGER	PK. Unique account identifier.
CustomerID	INTEGER	FK → Customer.CustomerID. Account owner.
BranchID	INTEGER	FK → Branch.BranchID. Servicing branch.
AccountType	TEXT	Account category (e.g., Checking, Savings).
OpenDate	DATE	Date the account was opened.
Status	TEXT	Current status (e.g., Active, Closed).

Purpose: Account grain for balances, activity, and branch/cust joins.

Account (example)

AccountID	CustomerID	BranchID	AccountType	OpenDate	Status
1001	1	1	Checking	2023-11-02	Active
1002	1	1	Savings	2024-05-12	Active

1003	2	3	Checking	2022-08-19	Active
------	---	---	----------	------------	--------

4) Merchant Schema

Column	Type	Description
MerchantID	INTEGER	PK. Unique merchant/counterparty identifier.
Name	TEXT	Merchant or generic label (e.g., Reliance Retail, IRCTC, Employer Payroll, Bank Transfer Out).

Purpose: Counterparty dimension for spend analysis and dispute attribution.

Merchant (example)

MerchantID	Name
1	Reliance Retail
2	Amazon India
3	Flipkart

5) Transactions Schema

Column	Type	Description
TxnID	TEXT	PK. Unique transaction id.
AccountID	INTEGER	FK → Account.AccountID. Account impacted.
TxnDate	TIMESTAMP	Transaction timestamp.

AmountINR	REAL	Amount in rupees (INR).
TxnType	TEXT	Credit (inflow) or Debit (outflow).
Channel	TEXT	Channel (e.g., ATM, POS, Online, Branch).
MerchantID	INTEGER	FK → Merchant.MerchantID. Present for all rows (generic merchant used for salary/fees/transfers).
Category	TEXT	Category label (e.g., Groceries, Travel, Dining, Electronics, Utilities, Fees, Transfer, etc.).
Description	TEXT	Optional description.

Purpose: Fact table for all money movement; basis for balances, spend, and ticket-size analytics.

Transactions (example)

TxnID	AccountID	TxnDate	AmountINR	TxnType	Channel	MerchantID	Category
T9AB12XKQ	1001	2025-08-21 13:05:00	1,499.00	Debit	POS	7	Dining
T7QW33PLH	1001	2025-08-18 09:12:00	65,000.00	Debit	Online	3	Electronics
T5LM90TRS	1002	2025-08-01 10:00:00	120,000.00	Credit	Online	9	Salary

6) Dispute Schema

Column	Type	Description

DisputeID	TEXT	PK. Unique dispute case id.
TxnID	TEXT	FK → Transactions.TxnID. Disputed transaction.
RaisedDate	DATE	Date dispute was raised.
ReasonCode	TEXT	Reason (e.g., Fraudulent, Duplicate, NotAsDescribed, RefundNotReceived).
Status	TEXT	Current status (e.g., Open, Resolved, Rejected).
Resolution	TEXT	Optional resolution notes (for non-Open cases).

Purpose: Case management for disputes related to specific transactions.

Dispute (example)

DisputeID	TxnID	RaisedDate	ReasonCode	Status	Resolution
D001	T7QW33PLH	2025-08-22	NotAsDescribed	Open	
D002	T9AB12XKQ	2025-08-23	Fraudulent	Resolved	Refund processed

Core System Architecture (Compulsory)

The **Banking Insights Text2SQL** solution is a **single ReAct agent** that uses **four tools** to turn natural-language questions into **read-only SQLite** queries and produce concise, grounded answers.

Tools to Implement:

You may **custom-implement** these tools or use **built-in** ones from providers like **LangChain** (e.g., `sql_db_list_tables`, `sql_db_schema`, `sql_db_query_checker`, `sql_db_query`) or equivalents from other frameworks.

1. List Tables — `sql_db_list_tables()`

Purpose: Discover available tables at the start of each query.

I/O: No input → returns list of table names.

Notes: Do not assume names; confirm **Transactions** (plural).

2. Inspect Schema — `sql_db_schema(table_names)`

Purpose: Pull columns/types for only the **relevant** tables.

I/O: List of table names → structured schema text/JSON.

Notes: Pay attention to canonical columns: **AmountINR**, **TxnType**, **MerchantID**,

AccountID.

3. SQL Query Checker — `sql_db_query_checker(sql)`

Purpose: Validate the SQL **before** execution (syntax + policy).

Enforces:

- **Read-only** (`SELECT` only; no DDL/DML)
- **No `SELECT *`** (only needed columns)
- Explicit join keys; no unbounded cross-joins
- Correct table/column names (e.g., `Transactions`, `AmountINR`)

4. Execute Query — `sql_db_query(sql)`

Purpose: Run the **checked** SQL query on the database and return result rows.

Notes: Return actual data only (no fabrication). Empty results are allowed.

Required Tool Use Sequence (ReAct)

1. **List tables** → 2) **Inspect relevant schemas** → 3) **Draft SQL query** → 4) **Run query checker** → 5) **Execute Query** → 6) **Summarize results \ Generate response.**

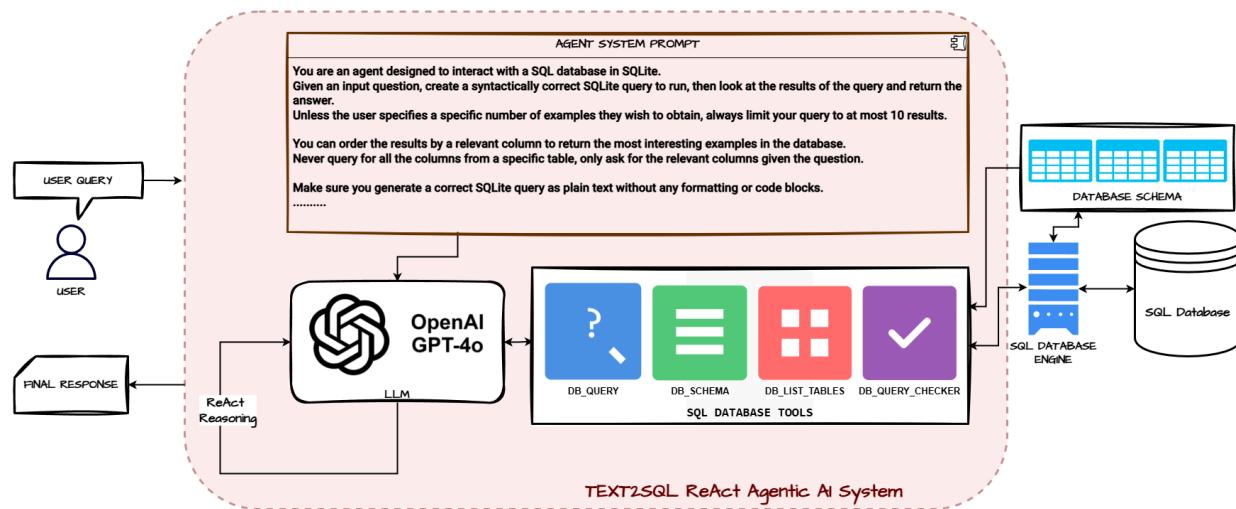
This order needs to be enforced by your agent & system prompt.

Query Rules (need to enforce in your system prompt)

- **Read-only:** Never generate `INSERT/UPDATE/DELETE/DDL`.
- **Minimal columns:** Never use `SELECT *`; qualify only what's needed.
- **Mention Table Schema Details to help the Agent**

- **Few-shot examples if necessary**
 - **Qualified joins:** Use explicit `JOIN ... ON ...` with the correct keys; avoid cartesian products.
 - **No SQL in the final answer:** The agent returns a short explanation + a small table (if helpful), not the SQL text.
-

System Architecture



Output Format (user-facing)

- **Answer:** Natural language insight as a short paragraph of text.
 - **Optional table or bullet points for results:** ≤10 rows, only relevant columns.
 - **No SQL shown** to the user.
-

Implementation Options

- **Use one of the following Agent Framework based on your choice - LangGraph \ AutoGen \ CrewAI**
 - **Use OpenAI GPT-4.1-mini as the LLM powering your Agent**
-

Optional Agent Enhancements

Optionally, you can enhance your existing Text2SQL Agent by adding the following capabilities (*you can try any one or both as an optional exercise*):

1. MCP Client-Server Architecture for Text2SQL

- **Create an MCP Server using FastMCP:**
 - Host all SQL-related tools (`list_tables`, `get_schema`, `query_checker`, `execute_query`) and the SQLite database inside an MCP server.
 - Deploy the server on a local port using `mcp.run(transport="streamable-http")`.
- **Build a separate Client Agent:**
 - Implement the Text2SQL agent in a client file (e.g., `client_agent.py`).
 - Use the `MultiServerMCPClient` (from `langchain-mcp-adapters`) to connect to the MCP server. (**or any other frameworks of your choice**)
 - Dynamically **discover available SQL tools** exposed by the server.
 - Run experiments by sending natural language queries through the client agent.
- **Testing:**
 - Try out a few representative queries (from either validation and test sets).
 - Verify that the client agent can seamlessly use the MCP tools via the server.
- **Reference Material:**
 - You may refer to a [sample notebook](#) demonstrating an end-to-end MCP Client-Server setup for agentic AI systems (provided separately).
 - This example shows how to structure servers, register tools, and build a client agent that consumes multiple MCP services.
- **Submission:**
 - You can deploy and test it on your own server or on Google Colab itself to make sure its working
 - Add the relevant code in an Optional Enhancements section at the end of the notebook

2. Multi-Agent System for Text2SQL + Visualization

- **Objective:** Extend the single Text2SQL agent into a **multi-agent system**
- **Agents Involved:**
 - **Text2SQL Analyst Agent**
 - Same as your current Text2SQL agent.
 - Handles all natural language → SQL queries.
 - Fetches structured answers (tables, rows, aggregates).

- **Visualization Agent**
 - A coding agent with access to Python execution tools.
 - Can generate plots, charts, and graphs using `matplotlib`, `pandas`, or other visualization libs.
 - Input: results from the Analyst Agent.
 - Output: visual representation (e.g., bar charts, line plots).
 - **Framework Options:**
 - You can implement the multi-agent system using **any one of the following**:
 - **LangGraph**
 - **AutoGen**
 - **CrewAI**
 - **Testing:**
 - Try at least **2-3 queries** with visualization requests (sample examples below):
 - “Show me the total deposits by branch in a chart.”
 - “Visualize the top 10 customers by debit spend.”
 - Compare outputs with and without visualization.
 - **Submission:**
 - Add the relevant code under the **Optional Enhancements** section at the end of the notebook.
-

Submission Guidelines

Your final submission must include two key artifacts:

1. **Jupyter Notebook** (`code.ipynb`) containing the complete implementation of your Banking Text2SQL Agent.
2. **CSV file** (`submission.csv`) containing the agent's generated responses to the 10 test queries.

These files will serve as the **primary artifacts for evaluation**.

1. Jupyter Notebook (`code.ipynb`)

Your notebook should be the central implementation file and must include:

- **Clear Structure with Headings & Explanations**
 - Use Markdown headings to separate sections.
 - Include short explanations of what each section does.
 - Add code comments for clarity.

- **Logical Flow**
 - Each part of the notebook should flow into the next, demonstrating how it contributes to the overall Text2SQL system.

- **Required Components:**

SNo	Component	Description
1	Database Setup	Load the provided SQLite database (<code>banking_insights.db</code>) and confirm connectivity. Show table names and a few sample rows for validation.
2	Tool Definitions & Implementations	Define and set up the SQL tools: <ul style="list-style-type: none"> • <code>sql_db_list_tables</code> • <code>sql_db_schema</code> • <code>sql_db_query_checker</code> • <code>sql_db_query</code>
3	System Instruction Prompt	Define the system-level instructions for your Text2SQL agent. Must include: <ul style="list-style-type: none"> • Available tools • Workflow sequence (list tables → schema → query checker → query) • Guardrails (no writes, no <code>SELECT *</code>, • Output format rules (plain answer, optional table) <i>Check the Query Rules section above for more details.</i>
4	LLM Integration	Use the chosen LLM (e.g., <code>gpt-4.1-mini</code>)

5	Agent Creation	Construct the ReAct-style agent using one of the following frameworks of your choice. <ul style="list-style-type: none">• Option A: LangGraph• Option B: AutoGen• Option C: CrewAI
6	Agent Experimentation & Validation	Run the agent on the provided two validation queries (<code>sample_queries_with_responses.csv</code>). Inspect results, compare with expected answers, and refine the agent as needed.
7	Testing on Hidden Queries	Run your finalized agent on the provided ten test queries (<code>test_queries.csv</code>), capture outputs, and save as <code>submission.csv</code> .

- **Optional Components:**

These enhancements are **not mandatory** but can be attempted to extend the functionality of your Text2SQL Agent. You may choose to implement **any one or both**, based on your interest.

SNo	Component	Description
1	MCP Client–Server Architecture	Create an MCP Server from scratch using FastMCP that hosts all SQL tools (<code>list_tables</code> , <code>get_schema</code> , <code>query_checker</code> , <code>execute_query</code>) along with the database. Deploy the server locally or in Colab. Build a separate Client Agent that connects to the MCP server, dynamically pulls in the SQL tools, and runs test queries through the client. Submission: You can deploy and test this on your own server or on Google Colab to ensure it works. Add the relevant code in an Optional Enhancements section at the end of the notebook.

2	<p>Supervisor Multi-Agent System</p> <p>Implement a Supervisor-based multi-agent architecture with two specialized agents:</p> <ul style="list-style-type: none"> • Text2SQL Analyst Agent – your original Text2SQL agent for generating SQL queries and retrieving results. • Visualization Agent – a coding agent (e.g., Python REPL) for producing charts/plots from query results. • Supervisor Agent – Optional (to coordinate between these 2 agents). <p>You may implement this using LangGraph, AutoGen, or CrewAI, based on your preference.</p> <p><i>Submission:</i> Test with at least 2–3 queries that explicitly request visualizations (e.g., “Show me a chart of top 5 merchants by spend”). Add the relevant code in the Optional Enhancements section of the notebook.</p>
---	--

2. Final Output CSV File (`submission.csv`)

This file will be used for final evaluation. It must follow the **exact format** below:

- **Filename:** `submission.csv`
- **Columns (in exact order):**
 1. `query` → The natural language query from `test_queries.csv` (do not change the queries).
 2. `response` → The final output from your agent.

Format Requirements

- `response` must be a **short, user-friendly text string** that directly answers the question.
- Avoid showing SQL code in the output.
- Tables or bullet lists may be included in the text where relevant.
- Each response should follow the structure enforced in your **system instruction prompt**.

Submission File Creation Workflow

- Load the `test_queries.csv` file with 10 test questions

```
1 test_df = pd.read_csv('test_queries.csv')
2 test_df
```

	query	response
0	Count active accounts by account type.	NaN
1	List the top 3 spending categories by total tr...	NaN
2	Top 3 customers who spent the most on dining i...	NaN
3	Current balance for the top 5 accounts (all ti...	NaN
4	Give me the total deposits by branch	NaN

- Generate answers for all the 10 test questions and store it in the `response` column

```
1 test_queries = test_df['query'].tolist()
2 # call_agent -> example function which uses your agent
3 # takes in a query and generates the agent response for it
4 test_responses = [call_agent(query)
5 | | | | | for query in test_queries]
```

```
1 pd.set_option('display.max_colwidth', 30)
2 test_df['response'] = test_responses
3 test_df
```

	query	response
0	Count active accounts by a...	There are 12 active Checki...
1	List the top 3 spending ca...	The top 3 spending categor...
2	Top 3 customers who spent ...	The top 3 customers who sp...
3	Current balance for the to	The top 5 accounts by curr

- Store the dataframe as `submission.csv`.

```
1 test_df.to_csv('submission.csv',
2 | | | | | index=False)
```

Submission File Sample Output Format

query	response
Which branch has the highest number of customers?	The Mumbai Fort branch has the highest number of customers, with
Top 3 merchants by total debit spend in 2025	1. Reliance Retail → ₹20.6M 2. Amazon India → ₹19.0M 3.
...	...

3. Submission Deliverables

Your final submission should include:

1. **code.ipynb** – complete, cleanly structured notebook.
 2. **submission.csv** – responses generated by the agent on the 10 test queries.
-

How Your Submission Will Be Evaluated

Your submission will be evaluated on the following three major dimensions:

1. **Agentic Workflow Design & Implementation**
2. **Agentic AI Task Performance** (based on the 10 test queries)
3. **Solution Code Quality**

A **3-point rubric** will be applied to each dimension, for a **maximum score of 9 points**.

Rubric

1. Agentic Workflow Design & Implementation

Evaluates the overall architecture and design of the **Text2SQL agent**, including tool configuration, planning logic, execution order, and response formatting.

Grade	Definition
Needs Improvement (1 point)	The workflow is incomplete or has major flaws. Tools are misused or missing. Steps are disordered or skipped (e.g., query execution without schema validation). Prompts are overly generic and responses lack structure.
Satisfactory (2 points)	A functional end-to-end ReAct Text2SQL workflow is implemented. Tools are integrated and used correctly, though orchestration or prompts may be basic. Responses are relevant but not always consistently formatted.
Excellent (3 points)	A robust, modular, and well-orchestrated ReAct Text2SQL workflow is implemented with complete tool usage (<code>list_tables</code> → <code>schema</code> → <code>query_checker</code> → <code>execute_query</code> → <code>summarize</code>). Prompts are detailed and effective. Final responses are structured, consistent, and aligned with project requirements.

2. Agentic AI Task Performance (10 Test Queries)

Evaluates how accurately the agent answers the **10 test queries** submitted in `submission.csv`.

Grade	Definition
Needs Improvement (1 point)	≤ 4 correct responses. Frequent SQL errors, irrelevant outputs, or hallucinations. Many answers incomplete or incorrect.
Satisfactory (2 points)	5–7 correct responses. Most answers relevant and factually grounded, though some may lack completeness or contain minor errors. SQL execution works for most cases.

Excellent (3 points)	8–10 correct responses. Outputs are accurate, clear, and well-structured. SQL queries are correct and grounded in the database. No hallucinations or irrelevant information.
---------------------------------	---

3. Solution Code Quality

Evaluates the quality, clarity, and maintainability of the submitted notebook (`code.ipynb`).

Grade	Definition
Needs Improvement (1 point)	Code is poorly structured, hard to follow, and lacks documentation. Error handling is absent. Notebook is disorganized.
Satisfactory (2 points)	Code is reasonably structured, functional, and includes some documentation. Notebook is readable but may lack polish.
Excellent (3 points)	Code is clean, modular, and well-structured. Comprehensive documentation and comments explain each step. Notebook has clear headings, markdown, and consistent naming. Demonstrates best practices in readability and reproducibility.

Scoring & Grades

Total Score	Grade	Pass/Fail	Description
8–9 points	Excellent	Pass	Outstanding performance across all dimensions. Demonstrates strong understanding of Agentic AI.
6–7 points	Satisfactory	Pass	Good overall performance with a solid grasp of key concepts, though some areas may need refinement.
3–5 points	Needs Improvement	Fail	Basic or inconsistent performance. Key aspects of the system require further development.

Note on Optional Components

The **Optional Components** (e.g., MCP Client–Server Architecture, Supervisor Multi-Agent System with Visualization) are **not required for grading**.

However, you are encouraged to attempt them:

- They provide valuable experience with advanced Agentic AI concepts.
 - While they will not affect your core score, strong implementations will help you tackle future use-cases with ease.
-

Submission Checklist

Before submitting, verify:

- Notebook (`code.ipynb`) runs **end-to-end** without errors.
 - All **required components** are implemented.
 - **System prompt** enforces guardrails.
 - `submission.csv` has exactly **2 columns** (`query`, `response`) and matches the required format.
 - Responses are **concise, correct, and grounded** in the DB.
 - No SQL queries are exposed in final user-facing answers.
-