Banking AI Assistant — Architecture (Detailed)

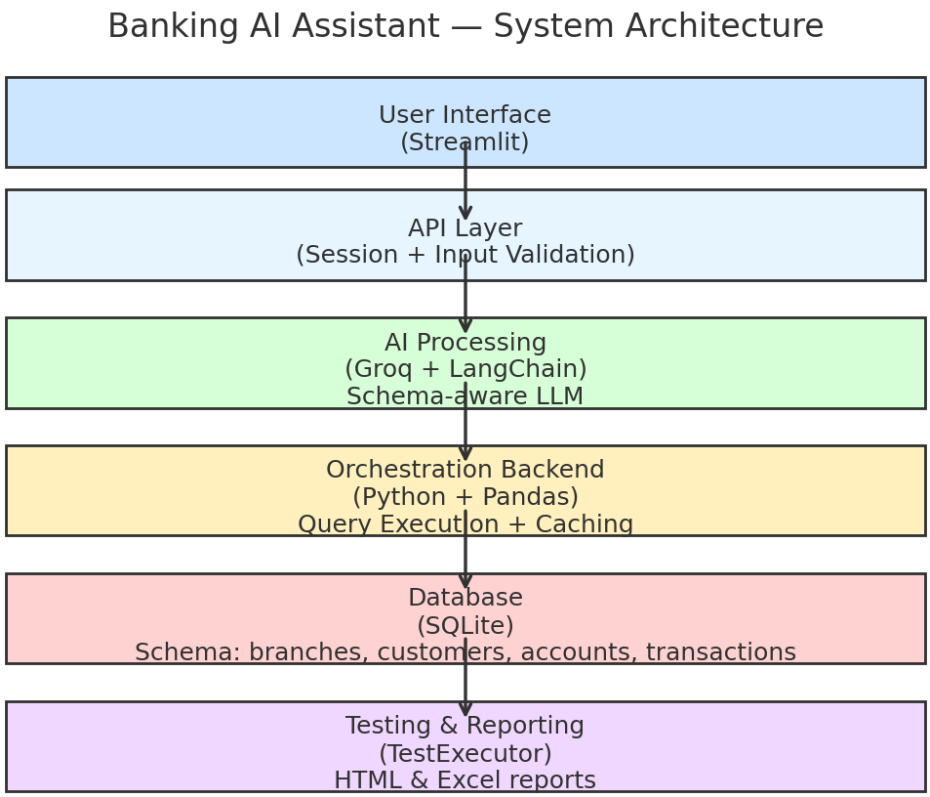
Table of Contents

1. System Overview

The Banking AI Assistant enables non-technical users to query a bank's database using natural language. It combines a Streamlit frontend, a schema-aware LLM via Groq and LangChain, and a Python backend that executes read-only SQL against an SQLite dataset. Design goals: usability, security, auditability, and easy migration to more scalable systems.

2. System Architecture

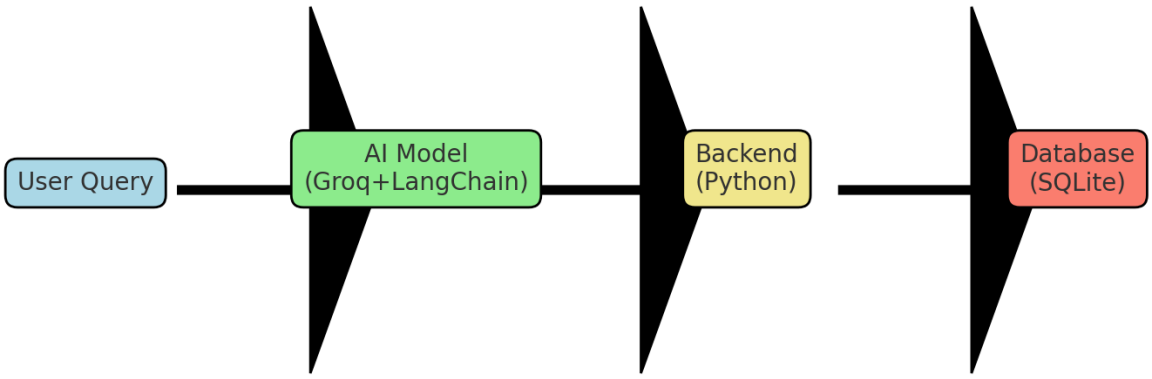
High-level layered architecture (visual):



Explanation: The frontend handles user interaction and session state. The AI Processing layer receives the database schema and conversation history to produce schema-constrained SQL. The Backend validates, executes, caches results, and formats responses. The Database stores transactional and master data. The Testing & Reporting layer runs automated test cases and produces detailed reports for audit and QA.

3. End-to-End Data Flow

Steps in the request-response cycle:



Detailed flow:
1) User enters a natural language query in the Streamlit chat.
2) The frontend constructs a system prompt that includes the DB schema and conversation history, and sends it to the AI layer.
3) The LLM returns either a SQL statement or a single CLARIFICATION question.
4) If SQL is returned, the backend sanitizes and executes the query against SQLite and returns the results to the UI. If clarification is requested, the frontend prompts the user and continues the chain.

4. Component Architecture

Frontend (Streamlit)

- Interactive chat UI using st.chat_input() and st.chat_message().
- Session state (st.session_state) stores conversation and results history.
- Result visualization via st.dataframe() and st.code().
- User-friendly error messages and example queries in the sidebar.

AI Processing (Groq + LangChain)

- Uses model llama-3.1-8b-instant for NL→SQL conversion.
- System prompt includes the full DB schema to prevent hallucinations.
- Clarification handling: model may return 'CLARIFICATION: ...' prompting the frontend to ask follow-up.

Backend (Python + Pandas)

- run_query() connects to SQLite and returns a pandas.DataFrame.
- Caching with st.cache_data to avoid re-reading schema.
- Orchestration of LLM calls via a lightweight workflow (langgraph.StateGraph).

Database (SQLite)

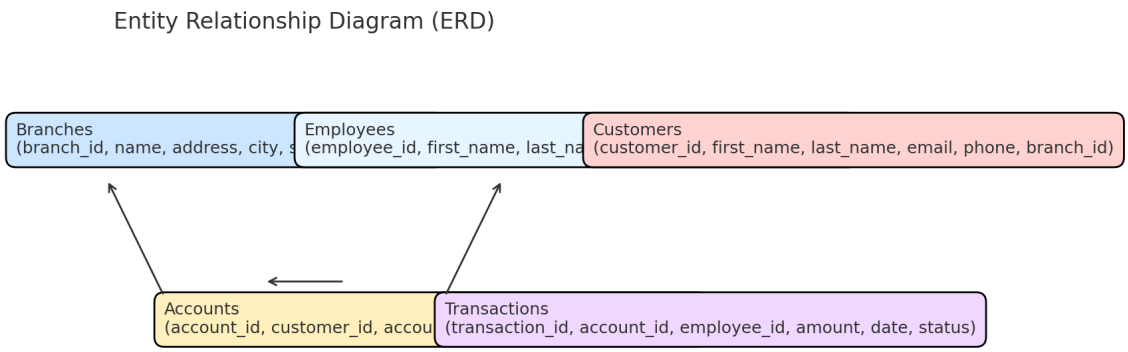
- Tables: branches, customers, employees, accounts, transactions.
- Designed for referential integrity and temporal queries.

Testing & Reporting (TestExecutor)

- Automated suite (35+ cases) that validates NL->SQL correctness and execution.
- Generates HTML and Excel reports with summary metrics and detailed rows.

5. Database Architecture (ERD)

Visual representation of entities and relationships:



Notes:
- branches is the central hub for branch-specific data.
- employees and customers reference branches via branch_id.
- accounts reference customers; transactions reference accounts and employees.
- All operations are read-only from the application to keep data safe.

6. Code Walkthrough & Key Snippets

I reviewed your main app and test executor to extract representative snippets and explain how they fit together.

6.1 Main app (core pieces)

1. System prompt used to constrain the LLM and embed the DB schema (excerpt):

```
    You are a highly intelligent SQLite expert. Your task is to convert a
user's natural language question into a valid SQLite query.

    **INSTRUCTIONS:**
    1. If the request is ambiguous and you truly cannot generate SQL, ask ONE
clarification question (prefixed with `CLARIFICATION:`).
    2. Otherwise, make reasonable assumptions and generate the best possible
SQL query directly.
```

```
    3. NEVER invent columns or tables — only use what is explicitly in the
schema.
    4. Use JOINs correctly (e.g., to get a customer's city, JOIN `customers`
with `branches`).
    5. Return ONLY the raw SQL query (or a clarification if absolutely
needed). No explanations, no markdown.

    **Schema:**
    ```sql
    {db_schema}
    ```
    """

    # LangGraph state
    class State(dict):
        messages: list

    workflow = StateGraph(State)

    def call_llm(state: State):
        response = llm.invoke([SystemMessage(content=system_prompt)] +
state["messages"])
        return {"messages": state["messages"] + [response]}

    workflow.add_node("llm", call_llm)
    workflow.set_entry_point("llm")
    workflow.add_edge("llm", END)

    agent_executor = workflow.compile()
```

2. run_agent() uses a langgraph StateGraph to stream model responses and return the final content. Key implementation (excerpt):

```
    def run_agent(messages):
        events = agent_executor.stream({"messages": messages},
stream_mode="values")
        response = None
        for event in events:
            response = event["messages"][-1].content
        return response

# --- UI ---

st.set_page_config(page_title="Banking AI Assistant", layout="wide")

with st.sidebar:
    st.title("💳 Banking AI Assistant")
    st.info("This app uses AI to answer your questions about the bank's
database. It remembers the conversation context and asks for clarification if
needed.")
    if st.button("Clear Chat History"):
        st.session_state.messages = []
        st.session_state.results = []   # also clear results
```

```
        st.rerun()
    st.markdown("---")
    st.header("Example Conversation")
    st.markdown("1. **You:** Show me recent transactions.")
    st.markdown("2. **AI:** CLARIFICATION: What timeframe do you consider
'recent'?")
    st.markdown("3. **You:** In the last 7 days.")

if "messages" not in st.session_state:
    st.session_state.messages = []

if "results" not in st.session_state:
    st.session_state.results = []   # store query results persistently

st.header("Query the Banking Database")

# Display past messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get new user input
if user_input := st.chat_input("Ask a question about the database..."):
```

3. Database access helper run_query() (excerpt):

```
def run_query(query):
    """Connects to the DB and runs the given SQL query."""
    db_full_path = os.path.join(os.path.dirname(__file__), '..', '..',
DB_PATH)
    conn = sqlite3.connect(db_full_path)
    df = pd.read_sql_query(query, conn)
    conn.close()
    return df

# --- Agent Setup ---

if not GROQ_API_KEY:
    st.error("❌ No Groq API key found. Please add it to the .env file in the
'code' directory.")
else:
    llm = ChatGroq(model="llama-3.1-8b-instant", api_key=GROQ_API_KEY)

    db_schema = get_schema()

    system_prompt = f"""
    You are a highly intelligent SQLite expert. Your task is to convert a
user's natural language question into a valid SQLite query.

    **INSTRUCTIONS:**
    1. If the request is ambiguous and you truly cannot generate SQL, ask ONE
clarification question (prefixed with `CLARIFICATION:`).
    2. Otherwise, make reasonable assumptions and generate the best possible
SQL query directly.
```

```
    3. NEVER invent columns or tables — only use what is explicitly in the
schema.
    4. Use JOINs correctly (e.g., to get a customer's city, JOIN `customers`
with `branches`).
    5. Return ONLY the raw SQL query (or a clarification if absolutely
needed). No explanations, no markdown.

    **Schema:**
    ```sql
    {db_schema}
```

6.2 Test Executor (automation & reporting)

4. TestExecutor initializes a dedicated ChatGroq instance, loads schema, and runs a suite of NL queries producing HTML/Excel reports.

```
    def generate_sql(self, natural_query: str):
        """Convert natural language to SQL using AI"""
        try:
            messages = [
                SystemMessage(content=self.system_prompt),
                HumanMessage(content=natural_query)
            ]
            response = self.llm.invoke(messages)
            return response.content
        except Exception as e:
            return f"Error generating SQL: {str(e)}"

    def execute_test_case(self, test_id: str, query: str) -> TestResult:
        """Execute a single test case"""
        start_time = datetime.datetime.now()

        # Generate SQL
        ai_response = self.generate_sql(query)

        # Check if clarification is requested
        if ai_response.startswith("CLARIFICATION:"):
            end_time = datetime.datetime.now()
            execution_time = (end_time - start_time).total_seconds()
            return TestResult(
                test_id=test_id,
                query=query,
                generated_sql=ai_response,
                execution_status="CLARIFICATION_REQUESTED",
                execution_time=execution_time,
                result_count=0,

    def execute_test_case(self, test_id: str, query: str) -> TestResult:
        """Execute a single test case"""
        start_time = datetime.datetime.now()

        # Generate SQL
        ai_response = self.generate_sql(query)

        # Check if clarification is requested
```
```

```
        if ai_response.startswith("CLARIFICATION:"):
            end_time = datetime.datetime.now()
            execution_time = (end_time - start_time).total_seconds()
            return TestResult(
                test_id=test_id,
                query=query,
                generated_sql=ai_response,
                execution_status="CLARIFICATION_REQUESTED",
                execution_time=execution_time,
                result_count=0,
                clarification_requested=True
            )

        # Extract SQL from response
        match = re.search(r'SELECT .*', ai_response, re.IGNORECASE |
re.DOTALL)
        sql = match.group(0).strip() if match else ai_response.strip()
        if sql.endswith(';'):
            sql = sql[:-1]

        # Execute SQL
        result_df, error = self.run_query(sql)
        end_time = datetime.datetime.now()
        execution_time = (end_time - start_time).total_seconds()

        if error:
            return TestResult(
                test_id=test_id,
                query=query,
                generated_sql=sql,
                execution_status="FAILED",
                execution_time=execution_time,
                result_count=0,
                error_message=error
            )
        else:
            return TestResult(
                test_id=test_id,
                query=query,
                generated_sql=sql,
                execution_status="PASSED",
                execution_time=execution_time,
                result_count=len(result_df) if result_df is not None else 0
            )

class TestReportGenerator:
    def __init__(self, test_results: List[TestResult]):
        self.test_results = test_results
        self.timestamp = datetime.datetime.now()

    def generate_summary_stats(self) -> Dict[str, Any]:
        """Generate summary statistics"""
        total_tests = len(self.test_results)
        passed = len([r for r in self.test_results if r.execution_status ==
"PASSED"])
```

```
        failed = len([r for r in self.test_results if r.execution_status ==
"FAILED"])
        clarification = len([r for r in self.test_results if
r.execution_status == "CLARIFICATION_REQUESTED"])

        avg_execution_time = sum(r.execution_time for r in self.test_results)
/ total_tests if total_tests > 0 else 0

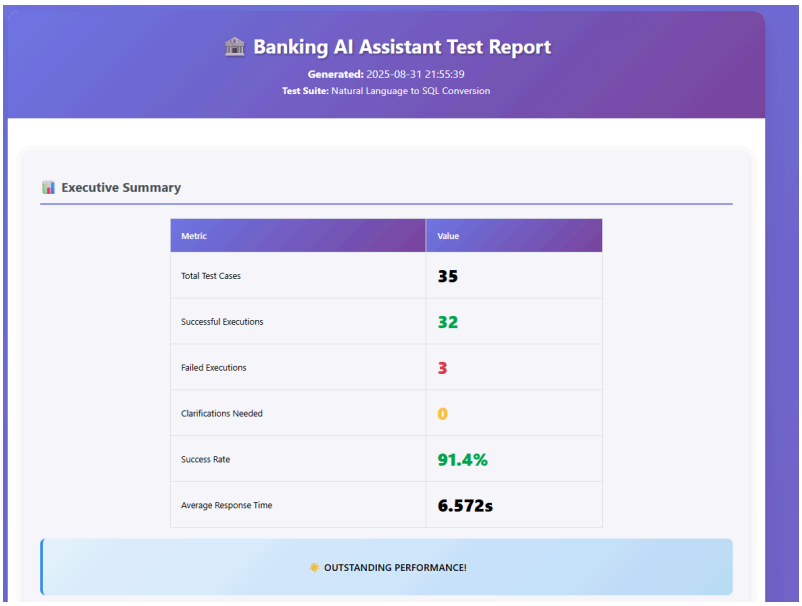
        return {
            "total_tests": total_tests,
            "passed": passed,
            "failed": failed,
            "clarification_requested": clarification,
            "pass_rate": (passed / total_tests * 100) if total_tests > 0 else
0,
            "average_execution_time": avg_execution_time
        }

    def generate_detailed_report(self) -> pd.DataFrame:
        """Generate detailed test results DataFrame"""
        data = []
        for result in self.test_results:
            data.append({
```

7. Testing & Reporting

Highlights of TestReportGenerator:

- - Produces an executive summary with pass/fail counts and average execution time.
- - Generates a polished HTML report and an Excel workbook with multiple sheets for passed/failed tests.
- - Important for auditing model behavior and tracking regressions after prompt changes.



8. Deployment, Security, and Future Roadmap

Deployment notes: run the Streamlit app with `streamlit run code/src/main.py`. Ensure .env contains GROQ_API_KEY and DB_PATH. (More detailed run instructions are in Appendix.)

Security notes: API keys in .env, read-only DB access, schema-aware prompts to avoid hallucinated columns, sanitized user-visible errors.

Roadmap: Replace SQLite with PostgreSQL, containerize AI & backend, implement RBAC and audit logging, add real-time dashboards.

9. Appendix

Quick run checklist:

5. 1. Create .env with GROQ_API_KEY and DB_PATH.
6. 2. Ensure banking_system.db is at the path defined in DB_PATH.
7. 3. Install dependencies: pip install -r requirements.txt
8. 4. Run: streamlit run code/src/app/main.py