provide the Design decisions & trade-offs and Known limitations using below readme file

Multi-agent Chatbot for Data analytics, RAG and Conversation

Features
* Multi-agent chatbot using LangGraph containing a Supervisor agent and three expert agents (Data analyst, RAG and Conversation)
* Reusable subgraph to create multi-agent system
* Expert agent has access of tools such as local python executor and knowledge vector DB
* Data Cleaning and Normalization from csv/txt/xls/json file
* Data Ingestion on Sqlite (local) and Postgres (server based)
* LLM reasoning (OpenAI)
* Streamlit frontend

📸 Architecture Overview
Architecture |
:-------------------------:
<img src="assets/architecture.png" style="display: block; margin-left: auto; margin-right: auto; max-width: 75%; height: auto;" >

<!-- Multi-Agent Swarm |
:-------------------------:
<img src="assets/multi-agent-swarm.png" style="display: block; margin-left: auto; margin-right: auto; max-width: 75%; height: auto;" > -->

Frontend Interface |
:-------------------------:
<img src="assets/frontend.png" style="display: block; margin-left: auto; margin-right: auto; max-width: 75%; height: auto;" >

Agents task:
* Supervisor agent → Routing between expert agents, data_analyst and conversation
* Data analyst agent → Answer queries related to data by Generate SQL query from natural language, execute SQL query on local python executer as tool and recursively fix the problem
* RAG agent → Question related of pdf documents using

knowledge base from chromaDB
* Conversation agent → Casual and friendly conversation

📁 Folder Structure

```
Multi_agent_Chatbot/
│
├── assests/                    # Images & docs
├── data/                       # I/O data folder
│   ├── in
│   ├── temp
├── src/
│   ├── agents/                 # Agents modules for
│   ├── configs/                # Configurations fil
│   ├── tools/                  # Tools modules
│   ├── utils/                  # Utility functions
│   ├── chatbot.py              # Streamlit Frontend
├── test/                       # Notebooks for test
├── pyproject.toml              # project file
└── README.md                   # Readme file
```

Data ingestion scripts [csv | txt | json | xls | xlsx | pdf]

```bash
    # For ingesting [csv | txt | json | xls | xl
    uv run python src/utils/data_ingest_sqlite.p

    # For ingesting pdf file to vector db
    uv run python src/utils/data_ingest_vectordb
```

Application usage

1. Create a `.env` environment variables as per below:-

```bash
    # LLM
    LLM_PROVIDER = "openai"
    MODEL_NAME = "gpt-4o-mini"
    EMBEDDING_MODEL = "text-embedding-3-small"
    OPENAI_API_KEY = "openai_api"

    # SQlite DB
    DB_PATH="data/temp/ingested.db"

    # (Optional) Langsmith for tracking and obse
    LANGSMITH_TRACING=true
    LANGSMITH_ENDPOINT=https://api.smith.langcha
```

```
LANGSMITH_API_KEY=API_GOES_HERE
LANGSMITH_PROJECT="Project name"
```

2. Install [uv package manager](https://docs.astral.sh/uv/getting-started/installation/)

3. Prepare environment

```bash
uv sync
```

4. Start Frontend (Streamlit) using below command

```bash
uv run streamlit run src/chatbot.py
```

5. Open Frontend (Streamlit) → http://localhost:8501

6. (Optional) using CLI mode

```bash
uv run python src/agents/supervisor.py
```

Observability using [LangSmith](https://smith.langchain.com/)
* Set the environment variables in `.env` file as mentioned above.
* Open the `https://smith.langchain.com` to track the chatbot performance

TODOs
1. [ ] Storing conversation threads
2. [ ] Cloud deployment
3. [ ] Test cases

Limitations
* Sometimes data_analyst agent is not able to answer complicated query due to multi-steps workflow includes SQL query generation and execution on local python executor tool
* Frontend does not render code and normal text due to formatting inconsistences from token streaming
* Locally served small LLM (8B parameters) can show its limitations due to small size
* Supervisor agent struggle with routing decision between `data_analyst` and `rag` for closely related queries

Design Decision and Tradeoffs

Below is a **clean, professional, and interview-ready** version of **Design Decisions & Trade-offs** and **Known Limitations**, written to fit naturally into your existing README without repeating earlier sections.

You can **paste this directly** under the corresponding headings.

Design Decisions & Trade-offs

1. LangGraph-based Multi-Agent Architecture

Decision:
The system is implemented using LangGraph with a **Supervisor agent** orchestrating multiple **expert agents** (Data Analyst, RAG, Conversation), each encapsulated as a reusable subgraph.

Trade-off:

- ✅ Clear separation of responsibilities and extensibility
- ✅ Fine-grained control over agent routing and tool access
- ❌ Increased orchestration complexity compared to a single-agent design
- ❌ Requires careful handling of state propagation and streaming across subgraphs

This design was chosen to reflect real-world agentic systems where different reasoning patterns and toolsets are required.

2. Supervisor-driven Intent Routing

Decision:
All user queries are first processed by a Supervisor agent that decides which expert agent should handle the request.

Trade-off:

- ✅ Centralized control and easier debugging of routing logic
- ✅ Enables future extensions (e.g., additional agents)
- ❌ Routing quality depends heavily on LLM reasoning
- ❌ Closely related intents (e.g., RAG vs data analysis) can cause misclassification

This approach mirrors production agent routers but introduces probabilistic behavior.

3. Tool-based Data Analysis via SQL & Python Execution

Decision:
The Data Analyst agent answers structured data questions by generating SQL queries and executing them via a **local Python executor** (SQLite/Postgres).

Trade-off:

- ✅ Guarantees answers are grounded in provided data
- ✅ Avoids hallucination from LLM internal knowledge
- ❌ Multi-step reasoning (generate → execute → debug) can fail on complex queries
- ❌ Execution safety requires restricted environments and careful validation

This was preferred over pure RAG for tabular data due to accuracy and transparency.

4. Local-first Storage (SQLite + Optional Postgres)

Decision:
SQLite is used as the default local database, with optional Postgres support for server-based deployment.

Trade-off:

- ✅ Easy local setup and fast iteration
- ✅ No external dependencies required for basic usage
- ❌ Not suitable for high-concurrency or large-scale datasets
- ❌ Schema inference is best-effort and may require manual tuning

This balances simplicity with extensibility.

5. Streaming at Supervisor Level with `subgraphs=True`

Decision:
Token-level streaming is implemented at the **parent (supervisor) graph level** using `subgraphs=True`.

Trade-off:

- ✅ Enables real-time streaming from subgraph agents
- ✅ Preserves a unified user experience across agents
- ❌ Requires filtering of routing tokens and final message chunks
- ❌ Streaming markdown (especially code blocks) introduces UI complexity

This approach follows LangGraph’s intended design for multi-agent streaming.

6. Streamlit-based Frontend

Decision:
Streamlit was chosen for rapid prototyping and ease of interaction.

Trade-off:

- ✅ Fast development and minimal boilerplate
- ✅ Built-in support for chat-style UIs
- ❌ Limited control over incremental markdown rendering

- ❌ Code blocks and text streaming require careful handling

Streamlit is suitable for demos and internal tools but not ideal for fine-grained UI control.

7. LLM Provider Abstraction (OpenAI / Ollama)

Decision:

The system supports both cloud-based (OpenAI) and local (Ollama) LLMs via configuration.

Trade-off:

- ✅ Flexibility in deployment environments
- ✅ Cost control and offline experimentation
- ❌ Smaller local models show reduced reasoning quality
- ❌ Behavioral differences across providers require tuning

This abstraction allows experimentation across model sizes and infrastructures.

Known Limitations

1. **Complex Analytical Queries**
   Multi-step analytical questions (e.g., nested comparisons, windowed aggregations) may fail due to cascading errors in SQL generation and execution.
2. **Supervisor Routing Ambiguity**
   The Supervisor agent can struggle to distinguish between closely related intents (e.g., RAG vs structured data queries), especially when user queries are underspecified.
3. **Streaming + Markdown Rendering Issues**
   Token-by-token streaming of markdown, especially code fences, can cause formatting issues in the frontend. This is a known limitation of incremental markdown rendering.
4. **Model Capacity Constraints**
   When using locally hosted small LLMs (≈8B parameters), reasoning depth, tool usage accuracy, and routing reliability are reduced compared to larger models.
5. **Schema Inference Is Heuristic-based**
   Data cleaning and normalization infer types automatically, which may not always match domain-specific expectations for ambiguous columns.
6. **No Persistent Conversation Storage**
   Conversation threads are currently session-bound and not persisted across restarts.
7. **Not Production-hardened**
   The system prioritizes clarity and extensibility over production concerns such as authentication, concurrency control, and resource isolation.