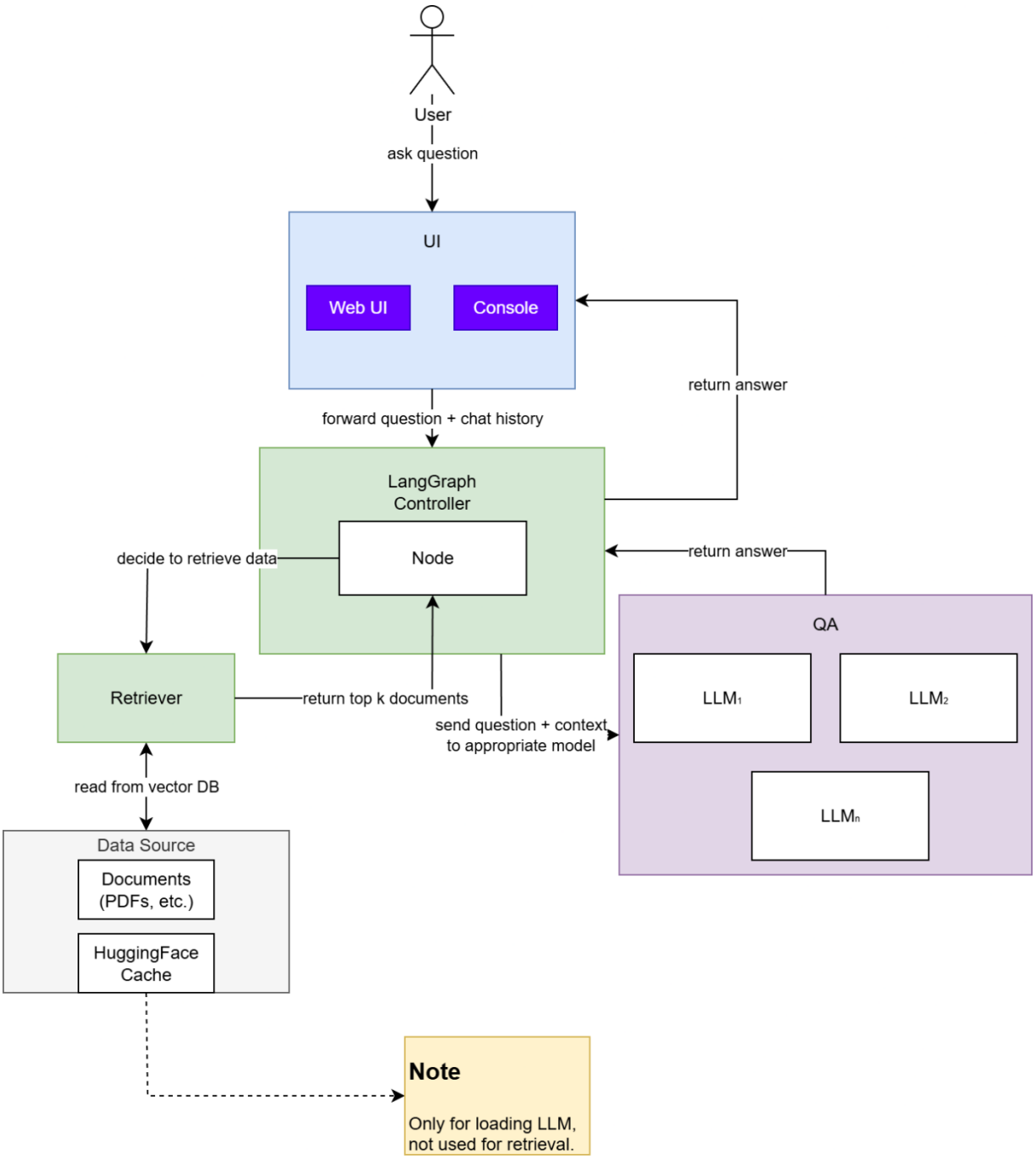
Agentic RAG Chatbot

Project Overview



Gábor Sárosi, 2025

Components:

UI: Interchangeable component, a chat view between User and Application. Collects and forwards questions of User, shows answers.

LangGraph Controller: Manages workflow. Decides if it needs extra context from Data Source, orchestrates gathered information. Forwards question and extra context to the appropriate large language model.

Retriever: After splitting documents into chunks, collects relevant information for controller.
Note: This project doesn't use web search to retrieve extra information. Production systems must do so.

QA: Collection of specialized large language models depending on what type of task the user wants to accomplish. Returns final answer to controller node.
Note: For demonstration purposes, the project uses one model (Meta Llama 3.2, 3B-Instruct) only.

Data Source: Persistence layer for documents, extra knowledge. Knowledge must be gathered before first use.

Key ideas and decisions:

- The app is thinking in nodes. The reason behind it is space-based architecture. Classic, monolithic approaches wouldn't be able to handle many user requests at the same time. Applications like these need distributed systems with dynamically growing infrastructure. Technologies like Kubernetes, cloud solutions would be must in production
- The project uses Meta LLama 3.2 3B-Instruct locally. A modern app in uses many, specialized LLMs (agents) in the cloud, depending on the type of task the user wants to accomplish.

Known bottlenecks:

- The app is currently working in a monolithic approach, made only for demonstration purposes. This includes all the potential issues discussed above.
- The app uses HuggingFace's Hub to download a Llama. Creating vector DB and first initialization of the model takes long time. Potential fix is discussed below.
- Loading Meta Llama, loading ChromaDB with knowledge takes long for the first time.
- Query time increases as we put more knowledge into ChromaDB.
- No parallelization is implemented in this project.
- No caching of conversations in this project due to performance limits.

Gábor Sárosi, 2025

- ChromaDB was stored on a HDD during development. Using other storages can improve query times massively.

Potential improvements:

- **Add another node which analyzes relevancy and groundness of retrieved data.**
- Space-based architecture
- Use remote API calls for LLM, for example https://groq.com/. API calls to Groq would speed up requests to be even faster than regular (like Gemini, DeepSeek) calls by seconds. For development purposes, Groq is free to use, user only needs an API key.
- Hosting an own LLM 24/7 on different piece of hardware. For obvious reasons this is not possible at the moment.
- More sophisticated controller decision logic
- Bigger vector database with more knowledge

Pipeline Overview:

- The user asks a question, which is sent to the UI (either Web or Console).
- The UI forwards the question and chat history to the LangGraph Controller.
- The controller decides whether to retrieve additional data from the Data Source via the Retriever.
- If needed, the Retriever reads from the vector database and returns documents to the LangGraph.
- The controller then sends the question and any retrieved context to the appropriate LLM.
- After processing, the LLM returns the final answer back to the controller, which then forwards it to the UI for display to the user. Testing strategies, Performance measures:

Technology Stack:

- **Large Language Model**: **Meta Llama 3.2 3B-Instruct**. Performance was a key factor during planning. Llama's model can run on any modern mid-range GPU, as it requires approximately 6GB of RAM to run locally.
- **Vector Database: ChromaDB** is a lightweight, open-source vector store. Its Python-native nature and ease of setup make it ideal for rapid prototyping and local development, as demonstrated in this project. It efficiently stores document embeddings and performs similarity searches, a core function of the RAG retrieval process. The use of a persistent directory ensures that the vector store is cached, significantly reducing startup time for subsequent runs.

- **Embedding Model: HuggingFace's all-MiniLM-L6-v2.** Sourced from the HuggingFace Hub, this model is a robust and widely-used choice for generating document embeddings. It transforms text into high-dimensional vectors, enabling the semantic search capabilities of the vector database.
- The project was configured to run on the GPU (**CUDA**), leveraging hardware acceleration to expedite the embedding process.
- **Orchestration Framework: LangGraph** is a framework built on top of **LangChain** for creating multi-actor applications. Unlike simple linear chains, LangGraph allows for the creation of cyclic graphs and conditional routing, enabling complex decision-making logic. In this architecture, a Controller node utilizes LangGraph to determine if a retrieval step is necessary based on the user's query, providing a foundation for more sophisticated, adaptive RAG pipelines in the future.
- The project utilizes a **subset** of entries from a **Wikipedia dataset**. This give a general, broad database knowledge to the model. The documents are processed using RecursiveCharacterTextSplitter with a chunk size of 1000 tokens and an overlap of 200 tokens. This strategy ensures that relevant contextual information is preserved across document boundaries while keeping individual chunks small enough to be efficiently processed by the LLM. Subset is used to save space and increase performance for demo.
- **HuggingFace** was chosen because it is the main community for modern machine learning. It provides broad choice of LLMs, provides great support for developing new AI apps.

Performance Analysis:

For the demo Python's `time` package is used. This is used in the Notebook showcase (though some IDEs show execution time in the bottom left corner). `Time` measures execution time and does not provide complete overview. Logs from `line_profiler`, `psutil` and `memory_profiler` would provide much better, accurate and broad measurements even on multi thread, but this would make the code lot more complex. The app has different goals.

Hardware used during implementation:

CPU: AMD Ryzen 7 3700X

RAM: 32GB DDR4 3200MHz

GPU: Nvidia GeForce RTX 4070 12GB

Storage: Caching was done on a SATA SSD, ChromaDB was stored on HDD.