Agentic RAG Chatbot: Project Report & Insights

1. Introduction

Alright, so I built this pretty cool Agentic Retrieval-Augmented Generation (RAG) Chatbot. The whole idea was to make a smart chatbot that can chew through different kinds of documents and answer your questions, kinda like having a super-smart librarian who also knows how to talk. We slapped a multi-agent system on it, which means it's got a bunch of specialized "brains" working together, all chatting through this thing called the Model Context Protocol (MCP). And yeah, it's got a Streamlit interface, so it's actually pretty easy to use.

2. Agent-Based Architecture with MCP Integration

So, how's this thing tick? It's all about agents, man. Think of 'em like a squad, each with its own job, and they all talk to each other using this special language called MCP. It's not just random chatter; it's structured, so everyone knows what's up.

Here's the crew:

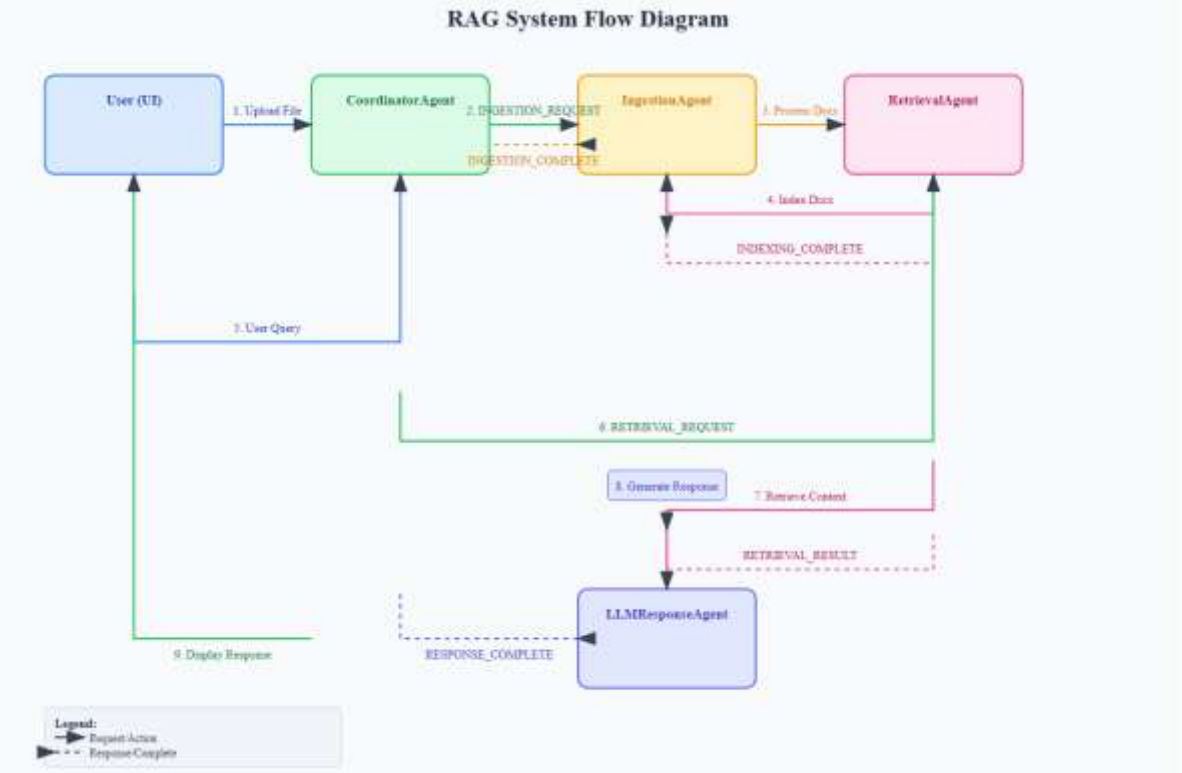
- **CoordinatorAgent:** This dude's the boss. He runs the whole show, makes sure messages go where they need to, and keeps track of what's happening in your session. Like, if you upload a file, he kicks off the whole process.
- **IngestionAgent:** This is the file-muncher. You throw a PDF, a Word doc, a text file, even a picture at it, and it figures out how to get the text out. If it's a scanned image, it even does OCR, which is like magic for turning blurry pics into readable text.
- **RetrievalAgent:** Once the IngestionAgent gets the text, this guy steps in. He takes all that text, chops it into little pieces (chunks), turns 'em into fancy numbers (embeddings), and shoves 'em into a super-fast database called Qdrant. When you ask a question, he digs through that database to find the most relevant bits.
- **LLMResponseAgent:** This is where the magic happens. The RetrievalAgent sends him the relevant bits, and he takes your question and those bits, and then talks to the big brain (our LLM, which is Ollama in this case) to cook up a proper answer.

And the Model Context Protocol (MCP)? That's the secret sauce for their communication. It's like a standardized walkie-talkie system. Every message has a sender, receiver, type, and a unique "trace ID" so we can follow the whole conversation from start to finish. This makes

sure messages don't get lost, and if something goes wrong, we can actually figure out where the screw-up happened. It's all asynchronous too, so agents ain't waiting around for each other. Pretty neat, right?

3. System Flow Diagram (with Message Passing)

Okay, so let's visualize how this whole thing flows. It's not just a straight line; messages are flying around!



Message Flow Example:

Let's say you ask a question after uploading a document.

1. You type a question in the UI.
2. The UI sends this query to the CoordinatorAgent.
3. The CoordinatorAgent, being the smart cookie it is, sends a RETRIEVAL_REQUEST message to the RetrievalAgent. This message contains your query and the session's trace ID.
4. The RetrievalAgent gets that message, looks at the query, computes some embeddings for it, and then searches the Qdrant database for the most relevant chunks of text from your uploaded documents.

5. Once it finds those chunks, the RetrievalAgent sends a RETRIEVAL_RESULT message to the LLMResponseAgent. This message has the original query and all the juicy context chunks it found.
6. The LLMResponseAgent takes all that info, builds a clever prompt, and sends it to Ollama (our LLM). Ollama then generates the answer.
7. Finally, the LLMResponseAgent sends a RESPONSE_COMPLETE message back to the CoordinatorAgent. This message contains the final answer and the source context chunks.
8. The CoordinatorAgent updates the session state and the UI displays the response to you.

4. Technology Stack Used

We didn't just pick random stuff; we went with a stack that makes sense for this kind of agentic RAG system.

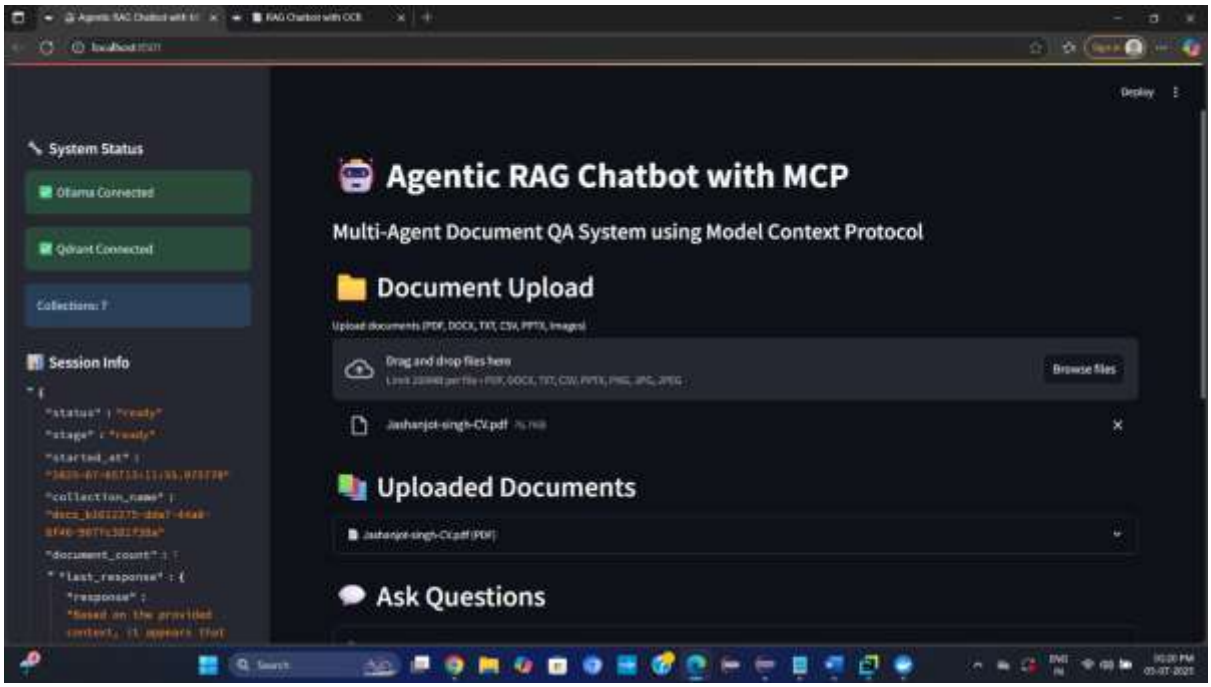
- **Frontend: Streamlit**
    - Man, Streamlit is a lifesaver for prototypes. You can whip up a decent UI super fast with just Python code. It's not fancy like React, but for showing off the core functionality, it's perfect.
- **Vector Database: Qdrant**
    - This is where we store all those numerical representations (embeddings) of your document chunks. Qdrant is super fast at finding similar chunks when you ask a question. It's like a super-indexed library for numbers.
- **Embeddings: Ollama (nomic-embed-text)**
    - We used Ollama to run nomic-embed-text locally. This model turns your text into those numerical embeddings that Qdrant loves. Running it locally means we don't gotta send all your data to some cloud service, which is cool for privacy.

- **LLM: Ollama (llama3.2:1b)**
    - Again, Ollama for the win! We're running llama3.2:1b locally for the actual brain of the chatbot. It's a smaller model, so it runs decent on typical machines, but it's still smart enough to give good answers based on the context.
- **Document Processing: LangChain, PyPDF2, python-docx, python-pptx**
    - LangChain helps a lot with handling documents and splitting them into chunks. Then we got specialized Python libraries like PyPDF2 for PDFs, python-docx for Word files, and python-pptx for PowerPoint. They're all about getting the raw text out of those weird file formats.
- **OCR: Tesseract**
    - For those scanned documents or images, Tesseract is our go-to. It's an open-source OCR engine that can read text from images. You gotta install it separately, which can be a bit of a pain, but it gets the job done.
- **Communication: Custom MCP implementation**
    - Yeah, we built our own message bus for the agents. It's a simple in-memory queue that lets agents send and receive messages without blocking each other. Keeps things smooth and organized.

5. UI Screenshots of Working App

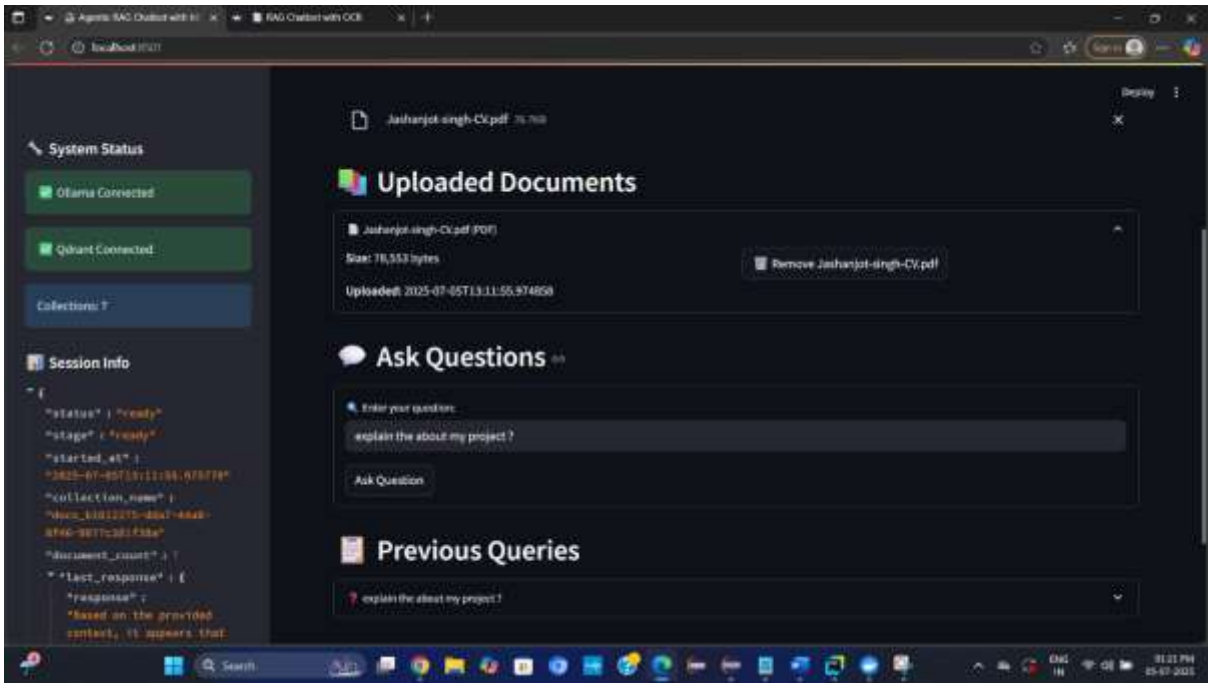
Alright, let's look at some pics of this thing in action. It's not gonna win any beauty contests, but it's functional, you know?

📸 Screenshot 1: The Main Dashboard



Here's the main screen when you first open the app. On the left, you got the "System Status" thingy, which tells you if Ollama (our LLM and embeddings stuff) and Qdrant (our database) are actually connected. Green means good, red means something's messed up. Then you see the "Document Upload" section where you can drag and drop your files. Pretty straightforward.

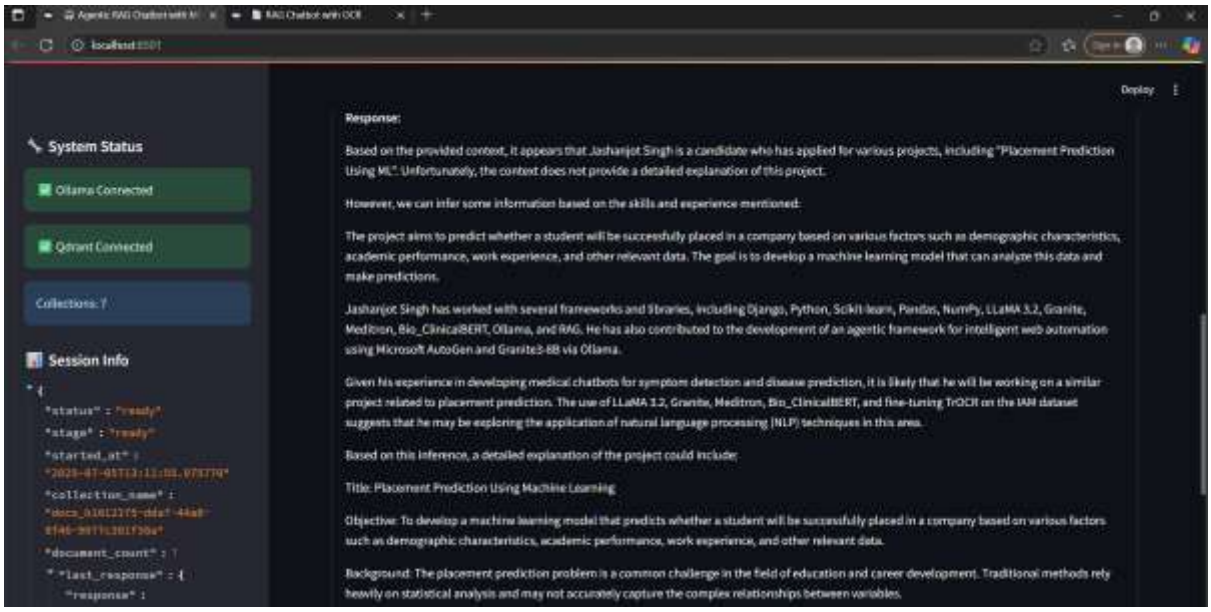
📸 Screenshot 2: Uploaded Documents and Asking a Question



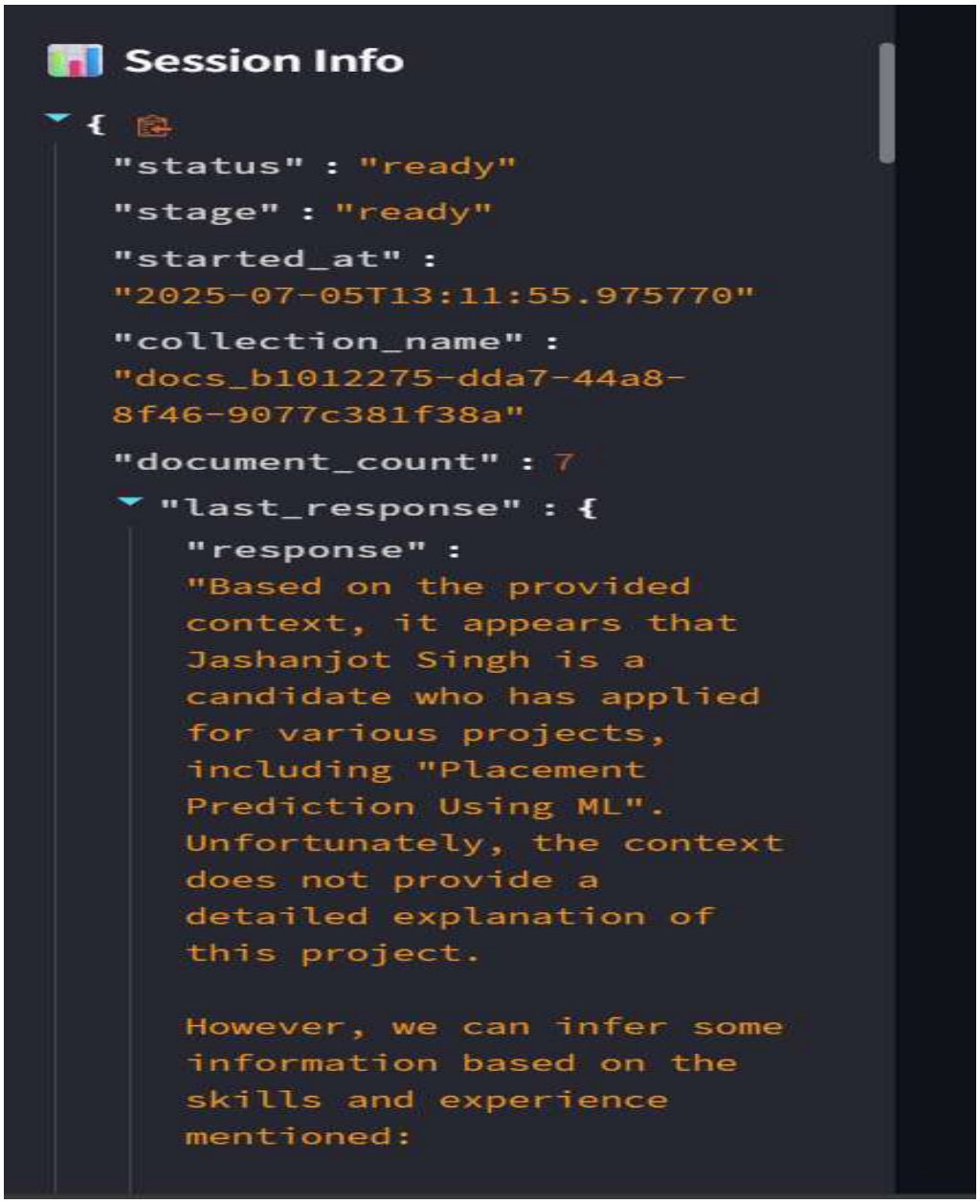
After you dump your files, they show up in the "Uploaded Documents" section. See that PDF there? That's my CV, lol. It tells you the file size and when it was uploaded. Below that, there's the "Ask Questions" part. You just type your question in that box, hit "Ask Question," and the magic starts.

📸 Screenshot 3: Getting a Response

This is the good part! After you ask something, the chatbot spits out an answer under "Response." It tries its best to use the info from your documents. If it can't find enough info, it'll tell you, like in this screenshot where it says "Unfortunately, the context does not provide a detailed explanation of this project." But it still tries to infer stuff. Pretty cool, huh?



📸 Screenshot 4: Session Info Details



This little zoomed-in shot shows you the "Session Info" in the sidebar. It's got all the nerdy details like the status ("ready" means it's good to go), what stage it's in, when it started, and even the name of the Qdrant collection it's using for your documents. You can also see a peek at the last_response payload, which is basically the data that made the chatbot say what it said.

6. Challenges Faced While Doing the Project

This project wasn't all sunshine and rainbows, you know? I hit some bumps along the road, for real.

- **Ollama and Qdrant Setup:** Getting Ollama to play nice with the models and Qdrant to actually run smoothly? That was a headache. Sometimes they just wouldn't connect, or the models wouldn't pull right. Like, you gotta make sure ollama serve is actually running and Qdrant's Docker container ain't acting up. It's a pain to debug when it's just a blank screen, you know?
- **Tesseract OCR:** Tesseract. Installing that thing on Windows was a whole adventure. You download it, then you gotta tell Python where it is, and if you get the path wrong, boom, no OCR for you. And for scanned docs, sometimes the quality is so bad, Tesseract just gives up, or gives you gibberish. That's a real bummer.
- **Long Response Times:** So, when you upload a big document or ask a complex question, the chatbot sometimes just sits there, thinking. For like, a minute, maybe even more. It's because the LLM and embeddings are doing a lot of heavy lifting. We had to put a warning there, like, "Hey, this might take a while," so folks don't think it crashed.
- **Memory Issues:** If you throw too many huge documents at it, or the chunks are too big, the whole thing can just eat up all your RAM and crash. It's like, "Whoops, out of memory!" We tried to tell people to lower the chunk size or use smaller models, but it's still a thing.
- **Prompt Engineering:** Getting the LLM to give you the *right* answer, not just *an* answer, is tricky. You gotta word your questions just so, for the RAG part to really shine. It's like talking to a genius, but you gotta speak their language perfectly.
- **Streamlit Limitations:** Streamlit is cool for quick UIs, but it's not super dynamic. Like, real-time updates and complex interactions? Kinda hard. We had to do some workarounds with session states and stuff to make it feel responsive.

7. Future Scope / Improvements

Alright, so this is just the beginning, you know? I got some ideas to make this thing even better.

- **Better UI/UX:** Streamlit is fine for a prototype, but for a real product, we should probably switch to something like React or Vue. Make it look slicker, more interactive, and just generally nicer to use.
- **More LLM Options**: Right now, it's just Ollama. But what if we could easily swap in other LLMs, like from OpenAI or Gemini? Make it more flexible, so users can pick their favourite brain.
- **Advanced Document Understanding:** For medical stuff, just plain text ain't always enough. Imagine if it could understand tables in PDFs, or charts, or even handwriting better. That would be next level.
- **Multi-Turn Conversation History:** The current version is okay, but it doesn't remember previous turns perfectly. Making it truly conversational, where it remembers context from like, five questions ago, that's the dream.
- **User Authentication:** Right now, anyone can use it. But for sensitive documents, you'd want login and stuff, right? So only authorized people can upload and ask questions about their private files.
- **Scalability:** If a million people suddenly start using this, our local Ollama and Qdrant setup ain't gonna cut it. We'd need to move everything to the cloud, use managed services, and make it scale like crazy.
- **Feedback Mechanism:** It would be cool if users could tell us if an answer was good or bad. That way, we could use that feedback to make the chatbot even smarter over time.
- **Voice Input/Output:** Imagine just talking to the chatbot and it talking back. That would be super convenient, especially in a healthcare setting.