Tatiyana Fridge

HON 402 Thesis

Dr. Pokorny

December 9, 2025

Using RAG to Build a Prototype University Conversational Assistant

Everyday responsibilities can quickly become too much to handle, and that's where AI can be very helpful. AI saves time, cuts down on wasted energy, and makes things simplistic for users. Advising season should be a simple streamlined experience for students seeking help. However, professors often manage full teaching loads, busy schedules, and advising duties all at once. When these demands pile up, gaps appear and mistakes are more likely to occur. Another concern is that not every professor has the background needed to fully understand specific student concerns, especially when those concerns could be outside the professor's area of expertise.

Research Question: Can a retrieval-augmented generation (RAG) system improve informational access for university students by providing accurate and context-specific answers compared to traditional static resources?

Introduction:

The rise of AI chatbots in higher education reflects a shift from cautious experimentation to proactive campus-wide integration. Institutions like the University of Michigan, UC Irvine, and Arizona State University have embedded chatbots into academic workflows, learning management systems, and even into mobile assistants that are offering personalized tutoring, course guidance, and administrative support (Gao, 2025; Hennick, 2025). 37% of universities provide institution-wide chatbot access, while 14% have developed homegrown systems (Hennick, 2025). Research suggests that course-specific bots outperform general-purpose models, supporting critical thinking and enhancing student engagement, while some students argue that they prefer AI-based emotional support for certain tasks over human interaction (Gao, 2025).

Inspired by these trends, McKendree Connect, a university-specific Retrieval-Augmented Generation (RAG) prototype, was developed to support McKendree's curriculum catalog. The system combines large language models, a vector database, secure login, and custom prompts to function as a campus-tailored academic advisor for majors, minors, and courses. Development was informed by technical resources and tutorials on FAISS (FAISS Team, 2021), LangChain (LangChain, 2022; Rabbitmetrics, 2023), Streamlit (Streamlit Team, 2023), Hugging Face (Hugging Face Team, 2021), and Google Gemini (Google DeepMind, 2023), as well as educational videos on RAG and prompt engineering strategies (IBM Technology, 2023; Tina Huang, 2024). These resources provided guidance on vector search, prompt chaining, orchestration, and front-end implementation, shaping the design and functionality of the prototype created.

Although still in early stages, McKendree Connect illustrates the potential of AI systems to provide personalized support, streamline academic workflows, and enhance the student experience while highlighting the technical depth required to implement such tools effectively (Gao, 2025; Hennick, 2025).

Background:

The McKendree Connect is a Retrieval-Augmented Generation (RAG) system, which means it uses its stored information along with a model that can interpret questions and generate accurate responses. The process begins with ingestion, where the catalog text files containing course descriptions and program requirements are added into the system so they can be processed and used later. After the files are stored, they go through chunking, which breaks the catalog text into smaller sections the model can work with more effectively. Each course becomes its own chunk, and each program's requirements are separated into individual sections based on the major or minor making the information easier to retrieve. Metadata is then added to every chunk, which is essentially a label that tells the system what each chunk contains. In McKendree Connect, these labels include details such as department, course title, and other related information. This structure allows the retriever to quickly locate the most relevant information instead of just searching through unrelated text. By going through these steps, the system becomes more accurate and easier to manage.

Once the text is chunked and labeled, the system moves on to embedding, which is the process of turning each text chunk into a vector or a mathematical representation that the machine can understand and compare. This step uses a sentence transformer, a specialized tool that creates embeddings by capturing the meaning behind the text which is otherwise known as context. In this system, the all-MiniLM-L6-v2 model is used as the encoder, which reads the text and turns it into numbers that represent its meaning. This is where the "real magic" of the RAG system happens here. Normal words and sentences from the text are transformed into arrays that the computer can work with mathematically. These embeddings are then stored in a vector database called ChromaDB, which keeps all the vectors organized and easy to search in one location. For understanding McKendree Connect maintains two separate knowledge bases, one for courses and one for programs.

When a student submits a question through the Streamlit chat interface, the code uses query routing to figure out where the question should go. The code looks for patterns in the question: if it mentions a specific course code or asks about classes, it sends the question to the course retriever. If it asks about major requirements or program details, it goes to the program retriever. Questions that need both types of information trigger a hybrid approach. The retriever is the part of the RAG system that searches the vector database and fetches the most relevant chunks of information based on the student's query. Within that process, it uses similarity search to find content that is closest in meaning while a setting called top_k is used to control how many documents the AI considers for the response. This helps prevent the model from being overwhelmed by too many possibilities. McKendree Connect retrievers also use guardrails, which are boundaries built into the system to make sure responses stay within the intended limits. Specifically, it checks for exact course code matches first to prevent the system from making up courses that don't exist. Thus, preventing hallucination. Once the most relevant chunks are found, the system moves to augmentation, where these chunks and their metadata are added to the prompt given to the language model.

Now we have the augmented prompt, which includes the student's question, the retrieved course or program information, and relevant metadata. The augmented prompt is then sent to the Large Language Model (LLM). LLM is a type of AI pretrained on massive amounts of text so it can understand human language and respond in a natural conversational way. In McKendree Connect, the LLM Llama 3.2:3b runs locally through Ollama. Using an LLM in a RAG system means the model doesn't have to be built from scratch but can instead be customized to handle specific tasks in this case the project is being developed to answer questions about McKendree's academic catalog. The LLM understands language through a Natural Language Processing (NLP), which lets it translate human language into something the computer can process and respond to in a way that feels natural for a user, such as yourself to read.

To summarize, the LLM acts as the generator by taking the top-ranked chunks from the retriever to generate the final answer displayed to the student. To make sure the responses are accurate and reliable, the system uses a common practice called prompt engineering, which is the process of designing the instructions or questions given to the model. In McKendree Connect, prompt engineering guides the LLM to answer only based on the provided context, cite sources with specific course codes and program names, format responses clearly, and admit when information isn't available instead of making things up. This form of prompting is essentially the "instructions" that tell the AI exactly how to generate the response.

McKendree Connect uses a temperature parameter on the LLM model to control how creative or predictable its responses are. A lower temperature makes the model more focused and cautious causing it to stick closely to the facts found, while a higher temperature allows it to be more open and creative in its responses. In this chatbot, the temperature is set to 0.0, the lowest possible setting, which makes the model highly "deterministic" and ensures it relies strictly on the information retrieved from the catalog rather than generating variations. This parameter setting along with limiting top_k to about 3 to 5 documents and using carefully designed prompts is essentially what helps prevent hallucinations from occurring. Hallucinations happen when the model makes up information that isn't in the data. It might sound confident, but the answer can be incorrect or completely unrelated.

Something to consider about AI is its ability for potential bias, which occurs when the AI reflects patterns or opinions in its training data even if they aren't necessarily accurate or fair. Since the LLM was pretrained on internet text some of these patterns could appear in its responses. To address this, in every response from McKendree Connect it includes a disclaimer reminding students to verify information with their academic advisor ensuring that human oversight is part of the process. Another common concept for chatbot is its use of caching to store frequently used data or results in memory so it doesn't have to reprocess them each time. The vector databases are loaded once at startup and then kept in memory from there which makes repeated similarity searches faster than new ones. Session management by Supabase are also using caching to save user conversations to allow students to continue previous chats without retrieving the same information again.

System Architecture and Methods:

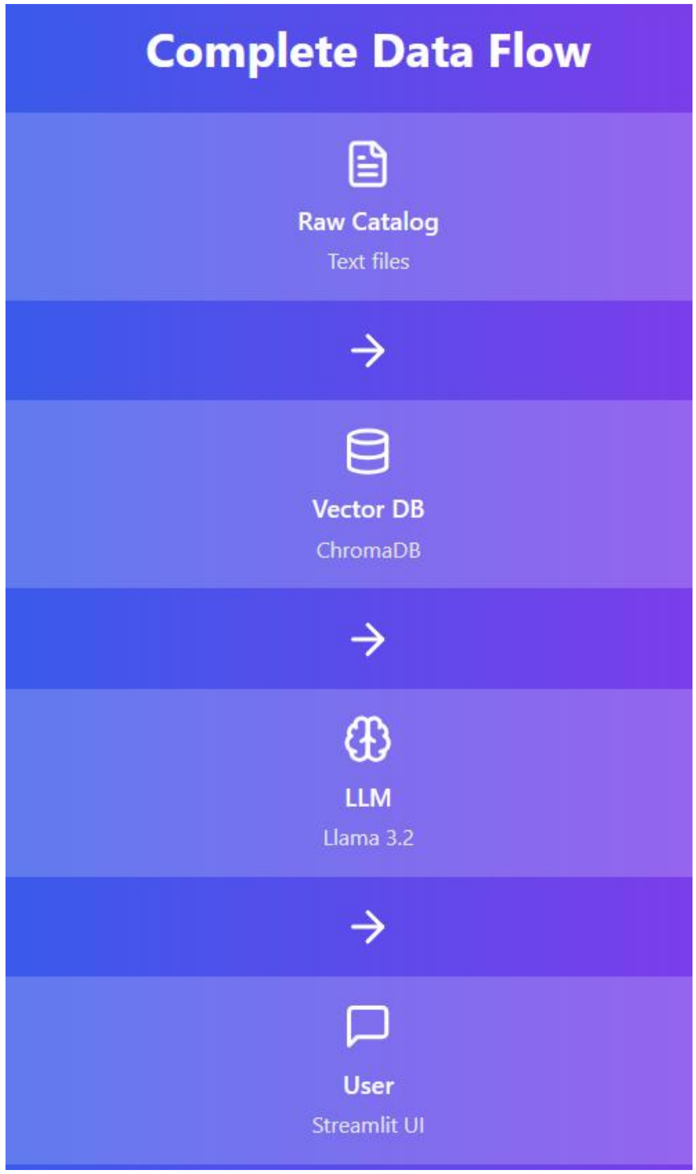
Building McKendree Connect involved significant trial and error to arrive at the system's current prototype. Each tool was selected to address a specific problem encountered during development. For example, earlier attempts with ChatGPT hit usage limits, and Google Gemini and Gemma 2B presented access restrictions or struggled with grounded prompting. Ultimately, the system uses Ollama running Llama 3.2:3b for local LLM execution. This model handles academic queries effectively, follows grounding instructions, and can manage more complex prompts without relying on cloud APIs. SentenceTransformers generate embeddings for the course and program data, which are stored in ChromaDB for semantic retrieval. Supabase manages user authentication, session history, and database storage, while Streamlit provides a interactive interface for students. Basic translation support is provided via the Translate API, though English remains the most reliable option for structured catalog queries. This line up of components is designed for modest hardware and low-cost execution making it accessible to schools with limited IT resources.

The current RAG pipeline processes student queries through five stages. Students first login, with Supabase handling authentication and session management. The student submits a question through the Streamlit interface, which is converted into an embedding using SentenceTransformers. ChromaDB retrieves the most relevant document chunks based on the embedding. These chunks, along with their metadata, are combined with the original question to create a structured prompt. The Llama 3.2:3b model that is running locally via Ollama generates a response from this prompt. Finally, the answer is displayed in the Streamlit interface, optionally passed through the Translate API for basic language support, and saved in Supabase to maintain conversation history. That is completing the full cycle from the question to the response in McKendree Connect.

Figure 1.1: summary of the tools

Component	Tool	Justification
UI Framework	Streamlit	Enables rapid development of interactive applications without frontend complexity; ideal for prototyping conversational interfaces
Embedding Model	SentenceTransformers	Open-source, CPU-efficient, produces high-quality semantic embeddings without requiring fine-tuning
Vector Database	ChromaDB	Runs locally with no cloud costs, efficient on modest hardware, ensures data privacy
Language Model	Llama 3.2:3b (Ollama)	Provides local LLM execution, eliminating API costs and external dependencies while maintaining complete control
Authentication	Supabase	Managed database with built-in authentication and PostgreSQL backend for user profiles and chat history

Figure 1.2: simplified data flow pipeline



The entire system is implemented in Python 3.11. The undergraduate course descriptions are stored in a .txt file, while program requirements for majors and minors are stored in a separate catalog.txt file. The run_parser.py script orchestrates the parsing process by calling the course_parser.py and program_parser.py classes, which convert the raw text files into structured JSON documents enriched with metadata such as course codes, departments, and credit hours. These JSON documents are then processed by run_loader.py, which calls the files course_loaders.py and program_loaders.py to generate vector embeddings that are stored in their respective ChromaDB databases for retrieval. From there, the retriever and responder logic handle user queries by searching the vector databases for relevant chunks and passing them to the LLM to generate answers.

To manage and streamline the workflow, the system uses several Python libraries. LangChain facilitates the construction of the RAG pipeline and manages interaction between the embeddings, retriever, and language model. Python-dotenv allows secure loading of environment variables. Pydantic is used for data validation. uuid handles unique identifiers for user sessions. Pypdf and pdfplumber assist with PDF processing if needed. Langdetect provides basic multilingual support and PyTorch enables running the local LLM for generation.

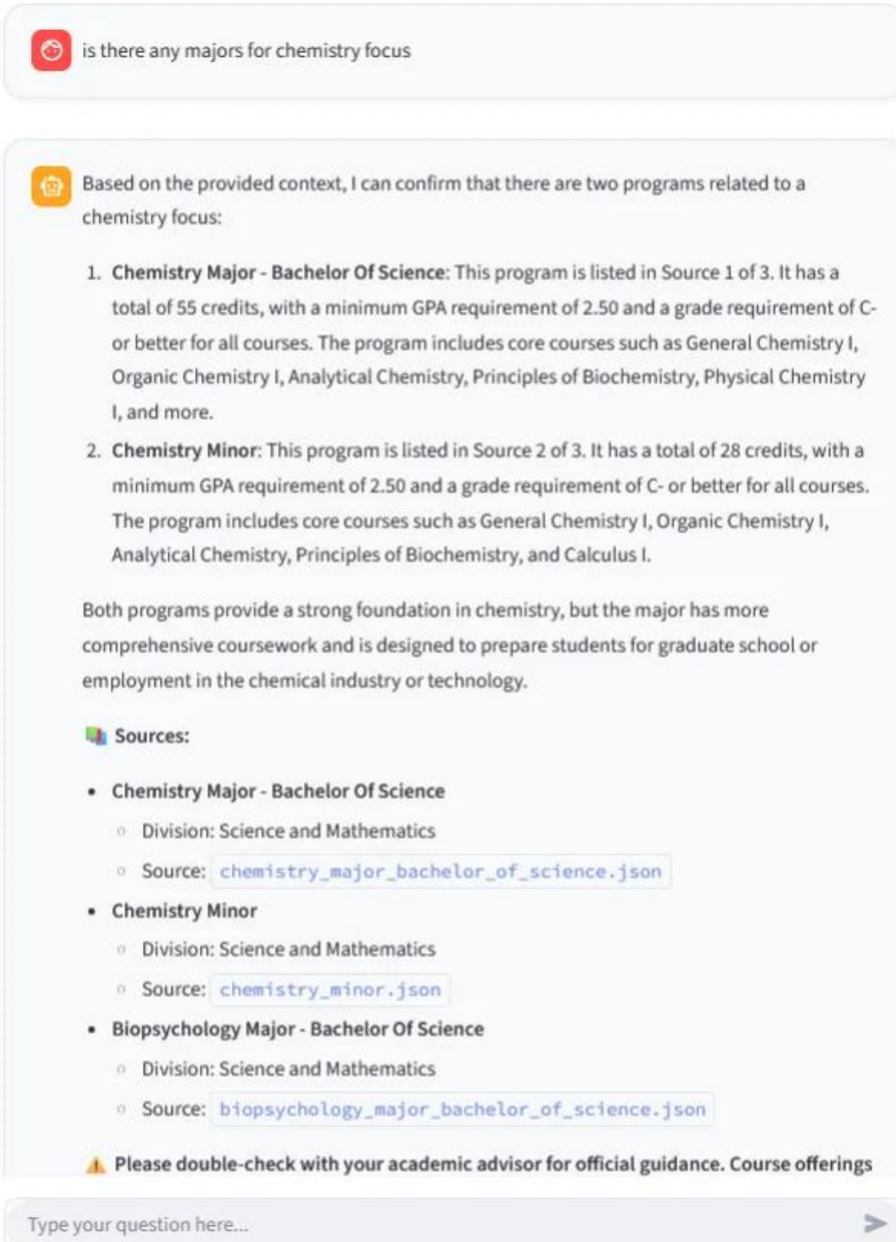
Privacy and accessibility were considered in the design of this prototype by including some basic multilingual support and a clean interface. However, this system does not fully comply with FERPA (Family Educational Rights and Privacy Act) or ADA (Americans with Disabilities Act) requirements, and readers should be aware that additional measures would be needed for production if used in an educational setting.

Evaluation:

To preface, McKendree Connect is not yet a fully developed prototype. Due to time constraints and computational limitations needed to reduce latency in response time and retrieval, the system currently includes only a select group of majors and minors from the catalog. However, every course description listed in McKendree's undergrad 2025 to 2026 academic catalog has been added to the database. As a result, some evaluations may appear unusual, but they are generated solely from the existing data within McKendree Connect's ChromaDB. For clarification, the majors included in this version are biopsychology, chemistry, computer information systems, computer science, cybersecurity, management information systems, and the minors included are applied artificial intelligence, chemistry, clinical and counseling psychology, computer information systems, and computer science.

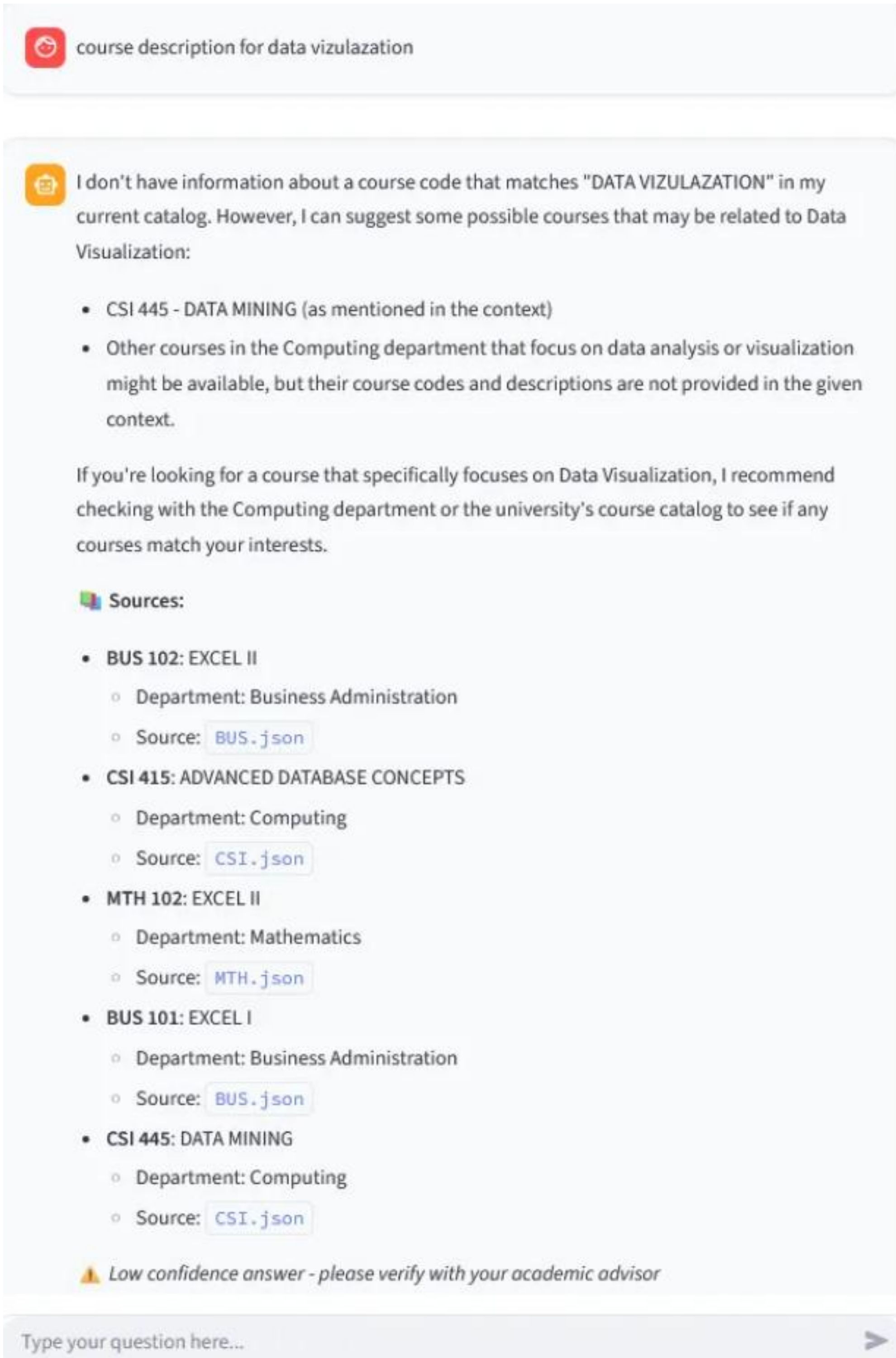
The McKendree Connect demonstrates strong performance across a range of query types, though the system also shows clear patterns in where it excels and where improvement is still needed. Figure 2.1 shows McKendree Connect handling a broad program-level question ("Is there any majors for chemistry focus?"). It correctly identifies both the Chemistry Major and Chemistry Minor and provides details such as credit hour requirements, GPA minimums, and core courses. This is one of the system's best-case scenarios: the query routing recognizes it as a program-related question, the retriever locates the correct JSON documents, and the generator turns them into a structured answer with proper citations. Everything functions exactly as the RAG pipeline intends.

Figure 2.1:



However, McKendree Connect struggles with misspellings and non-standard terms. Figure 2.2 demonstrates this with the query "course description for data vizulazation." Because of the spelling error, the system cannot find an exact course and refuses to guess which is part of the guardrails. Instead, it offers related courses like CSI 445 (Data Mining) or CSI 415 (Advanced Database Concepts). This shows that the system prioritizes accuracy over flexibility. While it's good that it avoids hallucination especially since temperature is set to a lower number so students who don't know the exact spelling may find the experience frustrating. The "Low confidence answer" warning does help manage expectations, but it also shows that the embedding-based similarity search does not handle misspellings as well as it could.

Figure 2.2:



Figures 2.3 and 2.4 show how McKendree Connect handles follow-up questions. When asked about the Applied Artificial Intelligence Minor, the chatbot initially gives the core courses. When the user asks for “more information,” it provides program details, credit requirements, GPA minimums, and learning outcomes. This shows that the system can handle multi-turn conversation as long as the user directly asks for clarification. It also continues to cite its sources clearly such as applied_artificial_intelligence_minor.json which helps maintain transparency about where the information came from.

Figure 2.3:

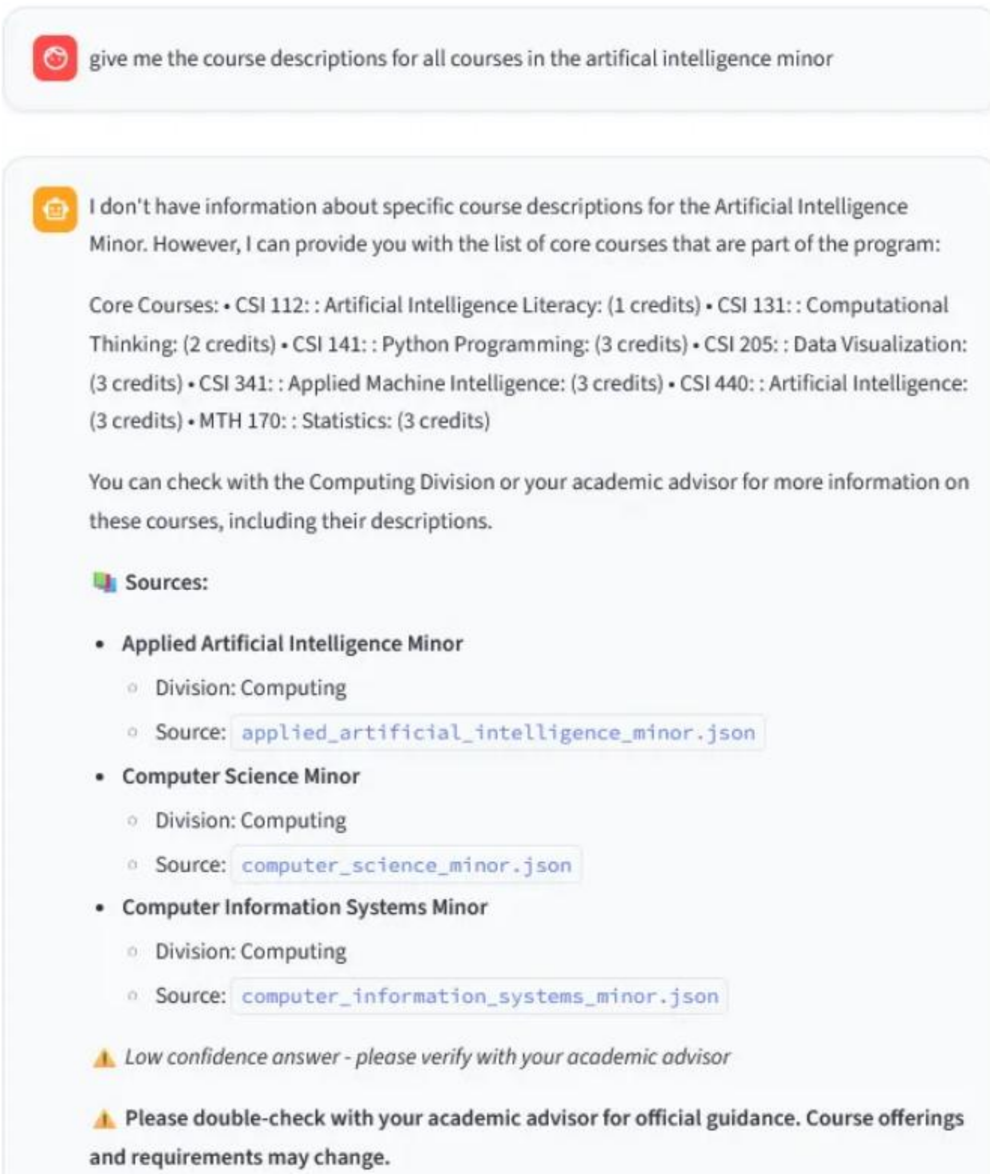
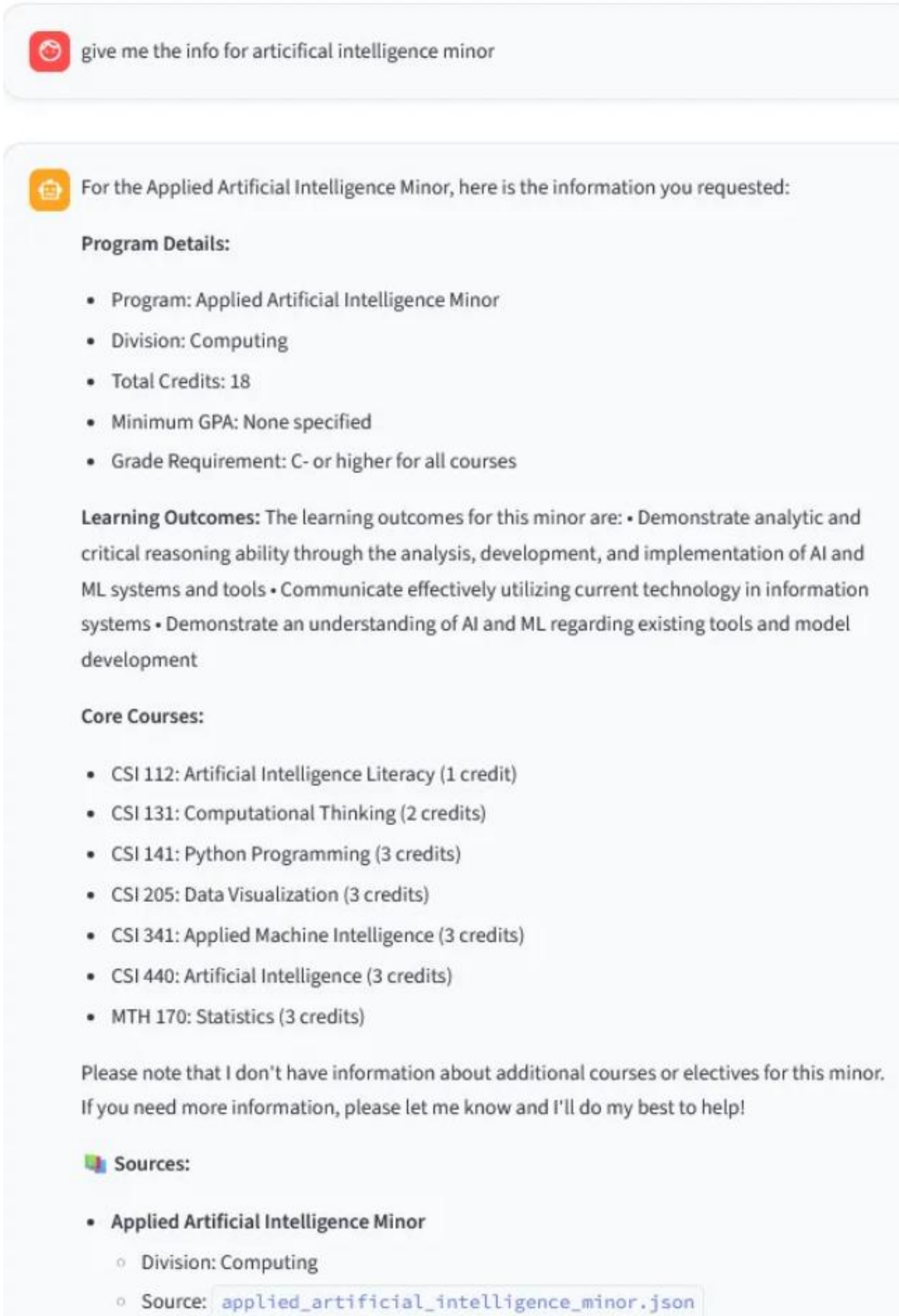


Figure 2.4:



This shows that the metadata filtering is working, even though the system only contains a smaller subset of the full catalog due to processing limitations. Figure 2.5 also highlights strong performance. When asked, “What are courses that have programming?”, the system retrieves programming-related courses across several programs and groups them clearly. Even when "programming" isn't explicitly stated in every course title, the semantic search still connects the concepts. Figure 2.6 shows the special case rule for upper-level CSI courses. Instead of using the LLM, McKendree Connect directly filters and returns all CSI courses numbered 300 or above giving 27 courses in total.

Figure 2.5:

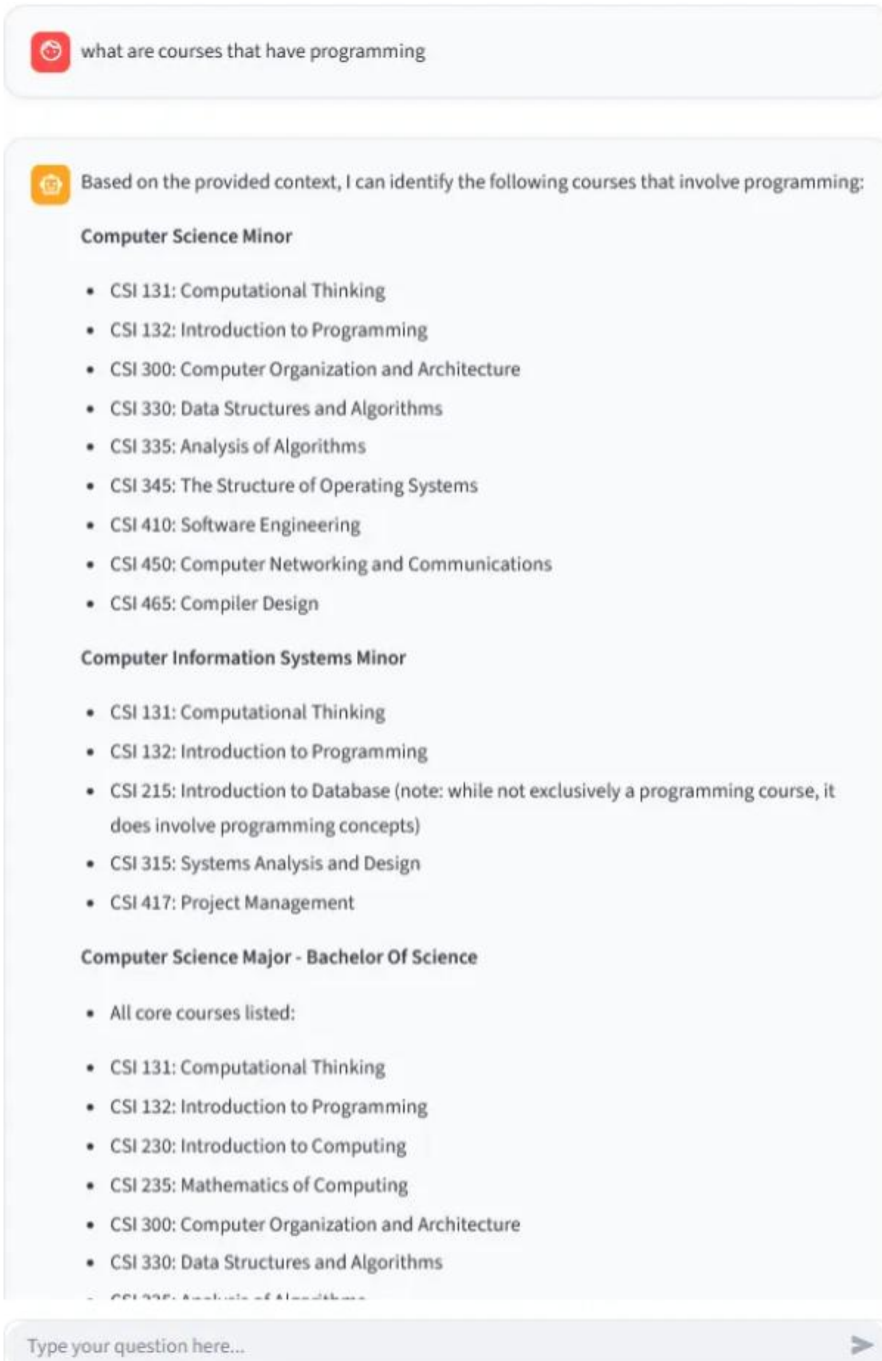


Figure 2.6:

upper-level courses for CSI

Here are the upper-level (300+) CSI courses:

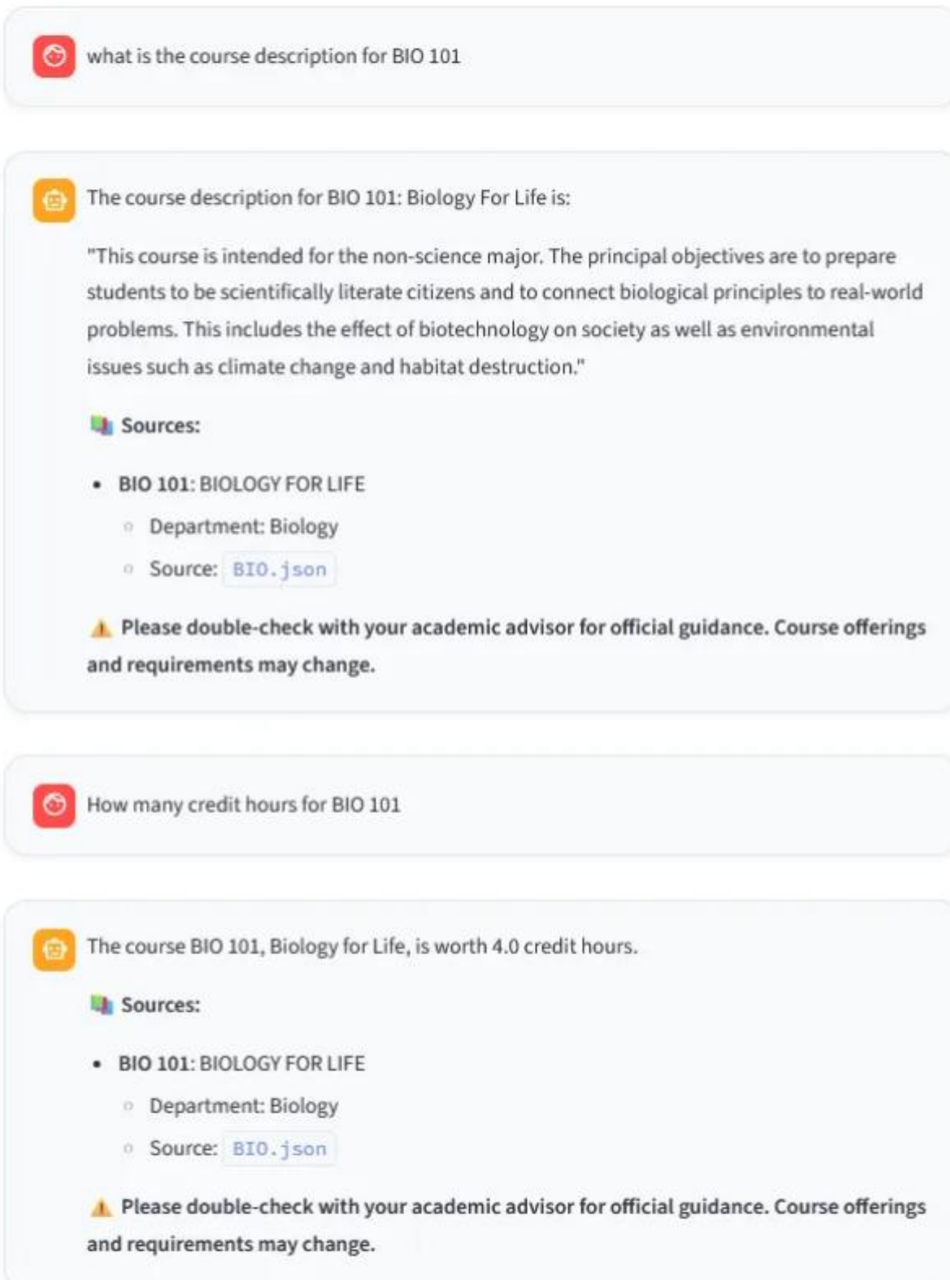
- **CSI 300**: COMPUTER ORGANIZATION AND ARCHITECTURE ([3.0] credits)
- **CSI 337**: INFORMATION SECURITY ([3.0] credits)
- **CSI 345**: THE STRUCTURE OF OPERATING SYSTEMS ([3.0] credits)
- **CSI 357**: ETHICAL HACKING ([3.0] credits)
- **CSI 369**: SOCIAL, LEGAL AND ETHICAL ISSUES OF COMPUTING (W) ([3.0] credits)
- **CSI 380**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 381**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 382**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 383**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 384**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 385**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 386**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 387**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 388**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 389**: SPECIAL TOPICS IN COMPUTING AND INFORMATION SYSTEMS ([3.0] credits)
- **CSI 410**: SOFTWARE ENGINEERING (W) ([3.0] credits)
- **CSI 415**: ADVANCED DATABASE CONCEPTS ([3.0] credits)
- **CSI 417**: PROJECT MANAGEMENT ([3.0] credits)
- **CSI 420**: THEORY OF PROGRAMMING LANGUAGES ([3.0] credits)
- **CSI 430**: COMPUTER GRAPHICS AND USER INTERFACE DESIGN ([3.0] credits)
- **CSI 450**: COMPUTER NETWORKING AND COMMUNICATIONS ([3.0] credits)
- **CSI 465**: COMPILER DESIGN ([3.0] credits)
- **CSI 467**: DIGITAL FORENSICS ([3.0] credits)
- **CSI 470**: INTERNSHIP IN COMPUTING AND INFORMATION SCIENCE ([3.0, 8.0] credits)
- **CSI 480**: INDEPENDENT STUDY IN COMPUTING AND INFORMATION SCIENCE ([1.0, 3.0] credits)
- **CSI 497**: SENIOR SEMINAR I ([1.0] credits)
- **CSI 498**: SENIOR SEMINAR II ([2.0] credits)

Total upper-level courses found: 27

Type your question here...

Finally, Figure 2.7 shows McKendree Connect at its most reliable: answering simple, direct questions like “What is the course description for BIO 101?” or “How many credit hours for BIO 101?” With exact course codes, retrieval is clean and precise. The system pulls the correct chunk, formats the answer, and cites the source without adding anything unnecessary. These examples show that the RAG setup works very well when the user knows the exact course or program name and the information exists in the database.

Figure 2.7:



Across all results, a clear pattern emerges as the system performs best with exact codes, standard phrasing, or well-defined catalog terms. It becomes less reliable with misspellings, vague queries, or questions that require piecing together information from many sources. These results also show that expanding the database to the full academic catalog, adding fuzzy matching for misspelled course names, and improving synonym handling (e.g. "coding" to "programming") would significantly improve usability. Another improvement would be prompting the system to ask clarifying questions when it detects ambiguity, instead of giving a low-confidence warning by default. Even with its limitations, the system consistently avoids hallucinations, maintains proper citations, and stays grounded in the catalog data. These results show that the core RAG architecture works as intended, especially for a prototype built on modest hardware and local processing.

Discussion:

This prototype addresses a common challenge in smaller universities. Essential academic information is often scattered across PDFs, outdated webpages, and disconnected platforms. Although advising support exists through faculty and the Academic Success Center, it is not always accessible during peak periods or after hours. A retrieval-augmented chatbot helps fill this gap by providing consistent on-demand answers to routine questions and reduces the strain on academic advisors. One of the system's most significant strengths is cost efficiency. Commercial academic chatbots are expensive and typically require high-end infrastructures while this prototype uses open-source tools, runs locally, and operates on modest hardware making it more realistic for institutions with limited budgets. Because the system is trained on verified catalog data it also avoids the common issue with general-purpose AI models such as ChatGPT or Google Search. Those traditional general-purpose AI cannot answer school-specific questions accurately since they lack access to internal documents.

Despite these strengths the prototype also has several limitations. Response times are slower without GPU acceleration which may not meet user expectations shaped by commercial AI systems. Performance also varies depending on how a question is phrased as stated before broad or unusually worded queries can lead to incomplete or incorrect retrieval. The system handles straightforward questions reliably but pulling information from multiple documents or working with inconsistent catalog formatting introduces errors. Early iterations also demonstrated the risk of hallucination such as inventing nonexistent majors or requirements when insufficient data was retrieved. The knowledge base is static, meaning catalog updates require manual re-indexing which introduces maintenance challenges and the possibility of outdated responses if updates are not applied effectively.

Overall, this thesis project demonstrates the complexity of integrating and building a system within a RAG pipeline. Several unexpected setbacks required adaptation throughout development, ultimately improving understanding of the process. A major challenge was the inconsistent structure of the academic catalog, which made parsing and extracting information accurately more difficult. The overhead of tokenizing large amounts of text was initially underestimated. At one point, the instructions added had overloaded the model and exceeded GPU limits. Debugging proved difficult because much of the underlying pipeline does depend on the pre-trained components that cannot be modified thus making it harder to isolate implementation errors from model behavior. As a prototype, the system demonstrates the feasibility of a low-cost academic assistant but also lacks the robustness, scalability, and refinement of a production-level platform like ChatGPT. The project highlights how design decisions such as chunk size, metadata structure, preprocessing, indexing methods, and guardrails have major effects on accuracy and overall reliability. Despite hardware limitations and being developed by a single contributor the work provides insight into the value of an institution-specific AI assistant and the technical depth required to build them effectively.

Future Work:

While this prototype demonstrates that a RAG-based academic assistant can support student advising additional features are required before it could be deployed at a university level. A production-ready system would need automated catalog updates, removing the reliance on manual re-indexing when courses or policies change. It should also personalize responses based on a student's major, academic progress, and degree plan while remaining fully FERPA-compliant. Improvements in speed and more robust handling of complex or multi-part questions, and deeper integration with institutional systems such as the academic calendar, advising appointments, or degree tools would significantly enhance reliability and usability for the average user. Future versions could also include expanded accessibility features such as screen-reader compatibility, adjustable text sizes, and more comprehensive multilingual support. Although these capabilities extend beyond the scope of what one developer can build independently in this timeframe the architecture in this prototype provides a foundation that can be expanded to incorporate these changes.

Conclusion:

Access to reliable academic information shouldn't be a privilege, it should be the baseline for every student. McKendree Connect demonstrates that even smaller universities can build intelligent AI systems without massive budgets. The system uses SentenceTransformers to generate embeddings of catalog data which are stored in ChromaDB for retrieval. Queries submitted through the Streamlit interface are routed to the appropriate retriever and responses are generated by the Llama 3.2:3b model. Supabase manages authentication, session history, and data storage. While LangChain and other Python libraries orchestrate the RAG pipeline to create an interconnected system. This project also provided an opportunity to apply skills learned from my major's curriculum. Similarly, it has also improved my research abilities and has allowed me to learn how to leverage available tools and forums to build something from scratch. While it's not perfect, it proves what's possible. Therefore, AI can meaningfully support students and make academic information more available for all students. The real takeaway is that with thoughtful design and effort AI can help shape a future where students have the proper resources they need to succeed at universities.

References

FAISS Team. (2021, March 5). *FAISS.* https://ai.facebook.com/tools/faiss/
Description:
FAISS is a highly efficient library developed by Facebook AI Research for similarity search in high-dimensional spaces. It supports exact and approximate search algorithms and can scale to millions of vectors. FAISS is essential for indexing and retrieving documents efficiently in a RAG system, allowing rapid retrieval of relevant information based on query embeddings.

Streamlit Team. (2023, February 14). *Streamlit.* https://streamlit.io/
Description:
Streamlit is a Python library for building interactive web applications for machine learning models. It allows developers to create UI components such as text inputs, buttons, and real-time data visualizations. Streamlit is ideal for building a user-friendly interface for a RAG system, enabling users to run queries and see results instantly.

LangChain. (2022, July 20). *LangChain.* https://www.langchain.com/
Description:
LangChain is an open-source framework that connects large language models (LLMs) with external data sources, such as vector databases. It provides tools for prompt templates, chains, and embedding integration. LangChain simplifies RAG system development by linking retrieval and generation components efficiently.

Hugging Face Team. (2021, September 8). *Hugging Face.* https://huggingface.co
Description:
Hugging Face offers a library of pre-trained NLP models for tasks including text generation and embeddings. Its models can be integrated into RAG systems to enhance retrieval and generation quality, allowing faster development and more accurate responses.

Google DeepMind. (2023, December 6). *DeepMind Gemini.* https://deepmind.google/technologies/gemini/
Description:
Gemini is a multimodal language model capable of complex reasoning and generating human-like text. Its integration into RAG systems supports contextually accurate responses based on retrieved documents, improving the generative component of the system.

Rabbitmetrics. (2023, April 13). *LangChain Explained in 13 Minutes | QuickStart Tutorial for Beginners* [Video]. YouTube. https://www.youtube.com/watch?v=aywZrzNaKjs
Description:
This tutorial provides a beginner-friendly overview of LangChain and its application in RAG systems. It covers LLM wrappers, chains, embeddings, vector stores, and agent-based actions, demonstrating how to retrieve data and generate responses efficiently.

IBM Technology. (2023, August 23). *What is Retrieval-Augmented Generation (RAG)?* [Video]. YouTube. https://www.youtube.com/watch?v=T-D1OfcDW1M
Description:
This video introduces the RAG framework, explaining how it addresses LLM limitations like outdated information and lack of source validation. It covers retrieval mechanisms, grounding responses in authoritative sources, and ensuring accuracy and transparency in AI outputs.

Tina Huang. (2024, December 9). *Google's 9 Hour AI Prompt Engineering Course in 20 Minutes* [Video]. YouTube. https://www.youtube.com/watch?v=p09yRj47kNM
Description:
This tutorial condenses Google's AI prompt engineering course, covering prompt design, chaining, and agent-based approaches. It provides strategies for iterative improvement, accuracy, and bias mitigation in RAG systems.

Gao, Y. (2025, July 14). *The rise of chatbots in higher education: Transforming teaching, learning, and student support.* The Ohio State University. https://ascode.osu.edu/news/rise-chatbots-higher-education-transforming-teaching-learning-and-student-support
Description:
This article examines AI chatbot adoption in higher education, including instructional support, administrative integration, and pedagogical strategies. It highlights ethical considerations, bias mitigation, and the role of human oversight, providing context for developing campus-specific RAG systems.

Hennick, C. (2025, May 13). *How three universities developed their chatbots.* EdTech: Higher Education. https://edtechmagazine.com/higher/article/2025/05/how-three-universities-developed-their-chatbots
Description:
This article provides case studies of AI chatbot implementation at the University of Michigan, UC Irvine, and Arizona State University. It details technical strategies, adoption metrics, and RAG use to reduce hallucinations. Insights inform best practices for building a university-specific RAG prototype.