**Generative AI and Learning: Using Retrieval-Augmented Generation (RAG) for C++ Tutoring**

**Ben Carpenter**

Computer Science Department, Quinsigamond Community College

**James Stevens**

Computer Science Department, Quinsigamond Community College

**Abstract**

By utilizing advancements from the field of AI, we aim to develop a tutoring app that answers the questions of intro-level C++ programming students, while creating a framework to expand into other academic areas. A Retrieval-Augmented Generation (RAG) approach will provide targeted feedback, dynamically generating context-specific responses by interfacing with a locally hosted AI.

Our goal is to minimize the risk of oversaturated or hallucinated responses. By processing incoming questions from users and communicating with the AI model, a RAG method ensures that the answers generated by the backend are both concise and relevant to the search. The front end will allow students to easily input their questions and view immediate feedback, ultimately enhancing the interactive learning experience. The functionality will be custom-built, while the graphic design from a previous personal project will be used. Combined, these systems will create a user-friendly application that provides students with clear and direct responses to their specific queries.

Overall, the project aims to create a scalable and efficient tutoring tool. Not only would this tool improve students’ understanding of C++ programming concepts, but it could be integrated into other educational platforms. Unlike the VS Code extension Continue or other similar tools, our project is designed as a two-part web application for easy integration into current LMSs (Learning Management Systems). The desired outcome is an engaging, adaptive, and accessible learning environment that can be extended to additional subjects and environments in the future.

**Tabel of Contents**

**Chapter 1**

Introduction 4

**Chapter 2**

Related Works 5

**Chapter 3**

Approach and Methodology

3.1 Problem Statement 9

3.2 Purpose 10

3.3 Solutions and Implementation

3.3.1 Preparing the Hardware 10

3.3.2 Understanding the Software 12

3.3.3 Teaching with Algorithms 13

3.3.4 Putting Everything Together and Testing 16

3.3.5 Analyzing the Results 17

3.3.6 Continuous Improvement 19

**Chapter 4**

Future Works 20

**Chapter 5**

Conclusions 22

**References 24**

**CHAPTER 1**

**Introduction**

Our objective for this project is the design and implementation of an Artificial Intelligence based tutor app that effectively addresses beginner-level C++ programming questions for students in the introductory computer science course. We chose C++ as the focus of our app because it is a foundational programming language in the software engineering program. It is important for a beginner to understand the more complex syntax of C++ before they can effectively learn topics such as pointers, memory management, data structures, and algorithms. The building blocks of these data types are laid in an introductory course, but the learning curve for C++ is steeper than higher-level languages, such as Python. These advanced concepts are more tedious and harder to grasp in the beginning, therefore an accurate and effective tutor app will enable a beginning student to absorb the new material more effectively.

The frontend is simply a means of accessing the backend. Written in Hypertext Markup Language, it provides a means of interaction with the function and features of the backend. It is the backend that executes the commands and returns the results.

The backend, implemented in Python, will serve as the engine of the application. It will process incoming questions from users and communicate with the Llama 3.2 model via the REST API, ensuring that the answers generated are both concise and relevant. This design minimizes the risk of oversaturated or generic responses, providing students with clear and direct explanations tailored to their queries.

By using a Retrieval-Augmented Generation approach to providing targeted feedback, our goal is to develop a basic, but effective AI framework that facilitates learning. Instead of relying on a pre-stored database of answers, the system will dynamically generate context-specific responses by interfacing with a local Llama3 AI model through Ollama’s local REST API.

We chose Ollama because… By using Llama3…

Retrieval-Augmented Generation is the cornerstone of our project. It is a generative AI process that combines the power of Large Language Models with a curated supply of information. Retrieval-Augmented Generation supplies the AI with contextual information from a data store that is related to a user query. It then uses this context to inform and guide the generative process, giving the AI a stronger knowledge of the topic at hand. (Lu, Yiu, 2.3) (citing references format?)

This approach also solves many of the outstanding issues that exist with generative AI. One of generative AI’s largest and most well-known flaws is that of hallucination, where the AI fabricates information with no factual basis. In a recent study, the authors found that out of 5000 ChatGPT responses, 19.5% contained hallucinations. (Li et al) Retrieval-Augmented Generation “has been used to improve code generation and summarization, enhance text-to-image generation, and perform more advanced slot filling, among other use cases.” (Can Small Language Models With Retrieval-Augmented Generation Replace Large Language Models). By implementing Retrieval-Augmented Generation, we are able to restrict the data pool, thus limiting the AI’s answers from straying into hallucination and solving the problem of hallucination with the use of our tutor.

(*beginning of introduction section)*

The value of our project is in its ability to provide an introductory computer science student with a course-specific learning tool. The app is not intended to replace textbooks or teachers, but to help support currently established methods of education. By focusing on specific material as the basis for our tutor app’s responses, we can add support to the development of problem-solving skills for these students, enabling a stronger knowledge of the material.

Online education already exists, though the online materials traditionally used to learn introductory computer science have their limitations. For example, Python Tutor helps students visualize runtime data structure changes during program execution. Visual Algo helps students visualize algorithms through animation. These are good tools and help provide insight into programming, data structures, and algorithms, but they are not always helpful with introductory topics. The issue is accessibility. Since uninitiated students will often lack an understanding of basic concepts, traditional online resources may not always be effective for them. These tools may not offer the flexibility to offer the best examples early on in a student’s coding education. Some newer students can struggle finding pertinent information without a clear overview of the problem.

By using Retrieval-Augmented Generation, the program will dynamically adjust to each unique question, providing students with a personalized response to each question and empowering them with answers that will help build a broader understanding. Because Retrieval-Augmented Generation is capable of drawing on current class materials, it can also focus in on a course-specific information set. By sourcing information from traditional educational resources, such as a textbook, the tutor remains consistent informationally with classroom materials. This feature allows the app to deliver this information in a more engaging, personal way for each student. By using relevant coding examples, it can offer an efficient method in reaching inexperienced students with supportive information, allowing them to better digest the textbook information.

This app can also help build confidence by approaching the information in an unthreatening way. Eliminating the fear of being judged by a tutor or faculty member, students are free to explore answers to their questions in an effective and comfortable environment. By freely pursuing basic questions, students can build their understanding and confidence to ask more precise questions of an instructor. This serves to lower barriers for new students and facilitate quicker, more stress-free progress and eventual mastery of the basics of C++ programming. As students better understand the material, they are more likely to continue in the degree path. (Li…)

*(beginning of related works section)*

This is significant, because Computer Science is currently facing obstacles. There is a growing reliance on software in all aspects of modern society, requiring more programmers. This causes more students to consider software careers. This increased interest drives the record undergraduate enrollment in Computer Science that many schools are seeing. The problem is that these schools are facing both a lack of qualified faculty and varied curriculum challenges. The combined force of these factors is stressing Computer Science education. As Ma, Martins, and Lopes pointed out, “Providing individualized support to many students in introductory courses, especially regarding mastery of complex material, has been challenging.” A strategic use of AI could further the educational reach of the faculty that now exist, reducing the need for an instructor’s direct involvement in simpler questions. (Ma…)

Without a working knowledge of computer concepts, many beginning computer science students need to be able to learn and review the intro. Ma, Martins, and Lopes, instructors at the University of California – Irvine, conducted a study of AI tutors within the context of computer science education. They looked at a pool of 455 students at the University of California – Irvine. They deployed five RAGMan tutor apps to assist the students with their supplemental homework assignments. These tutor apps were designed to give guidance, not solutions. In this way, the students developed experience by participating in a more practical process, ultimately finding their own answers. (Ma…)

Their research suggested that, “AI tutors can positively impact student success and provide important help, especially to students who would be struggling in challenging courses.” (Ma…) They concluded that the increase in the number of students continuing through the degree path, when using the RAGMan tutors, was considered statistically significant. Furthermore, the student feedback was very positive, demonstrating a positive user experience. User satisfaction helps to ensure a broader use of these tools. (Ma…)

Creating a virtual personal assistant for computer science students is very promising based on the results of such research. Our tutor app seeks to provide a pressure free, efficient, and personalized tutor experience for introductory students that is able to draw specifically on trusted course materials. If we continue to prioritize the feedback and interactions of the students, we can further enhance these learning tools, making them more effective and user friendly.

Our tutor app has the advantage of accessibility. This would benefit students financially, as personal tutors can be very expensive. Most students cannot afford to pay a human tutor $50-$200 per hour for guidance. Also, for students enrolled in schools with high class populations, it can be hard to get access to tutoring help from other students or faculty. It could also provide students with active, accurate support outside a tutor’s or professor’s available hours. Though both affordability and accessibility, the app would be a great supplement to traditional teaching resources such as textbooks and class lectures. The app would make extended support possible as students begin to establish their basic skills.

As a learning tool, it would also be cost-efficient for educational institutions to implement. By using Ollama, a locally hosted AI, running the Llama 3.2 model, we were able to reduce the costs often associated with generative AI. Most large language models cost per token. These operating costs accumulate with each use. However, Ollama allows you to run a variety of models locally. After the initial cost of setup, this limits the continuing costs of operating to just maintenance and electricity.

This also reduces the environmental impact of AI. A recent article by ???... shows the difference in the impact of cloud AI versus that of a locally run iteration…

*(beginning of solutions/implementation section)*

Our app is made with two main modules, the front-end and the back-end. The front-end handles the displaying and storing of messages and user input. This is the interactive portion accessed by the end user. The back-end serves as the workhorse of the app. It handles the context information, storage, and lookup, as well as response generation. By separating it into two pieces, administrators are able to integrate the function of the app into their existing platforms or software.



***Figure 1*** – This is an image showing the front-end of our project as we designed it.

For our front-end (see figure 1), we opted for a user-interface built with Hypertext Markup Language, Cascading Style Sheets, and JavaScript. This allows us to build a robust interface that can either be used on a website, or hosted by the local user. We built the frontend with several core features. These include Messaging the AI for help, showing the user’s chat history, remembering inputs from the user’s last session, and chapter selection. Another feature is the running of user-entered code, allowing the testing of your code within the browser.

For the messages, we built a display box that follows similar design language to most cell-phone texting apps. This will make it more natural for people to use / gives it a feel that will be familiar to a wide range of people. It has a box beneath the display where a question can be typed or a message sent. It also has a button to clear the user’s message history. This is the primary means of interaction with the tutor application.

This chat area not only takes the input, but its core functions include remembering the user’s code, question, and chat history, passing that information to the back-end with every new question. This allows the tutor to have a chain-of-thought, remembering a user’s initial question and responding to normal human language by referencing previously discussed information.

At the top is a drop-down box that allows users to select the chapter of the textbook that they are working in. This not only permits the user to specify a particular section in the course, but it also aids the generation process by limiting the search parameters to the information within that section of the text. By limiting the textual references, the bot is able to work leaner, faster, and more accurately.

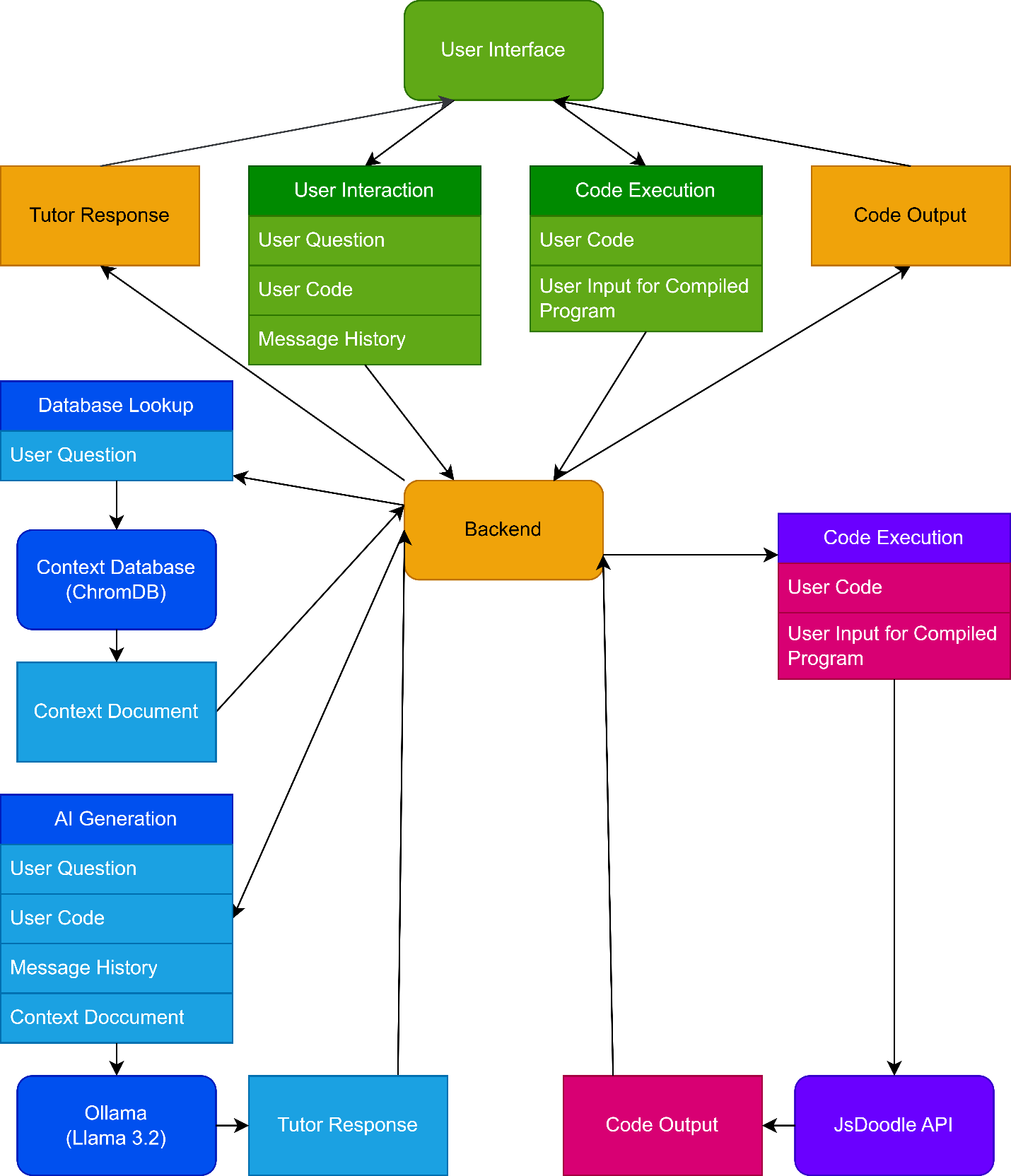
To the left of the message area is a code box where C++ code can be entered. Once entered, the front-end will color the code similar to a typical Integrated Development Environment. Doing this provides a familiar experience for people that are used to working in a normal Integrated Development Environment. We were able to achieve this by using a JavaScript library called highlight.js. This library is used to parse the information from the text input field and put it into a colorized and stylized format, displaying it behind that field in real time. The code being edited is actually invisible text inside the text input field overlaying the display element. To further the continuity, we overrode the default tab action to instead place four spaces, similar to Visual Studio Code (A Popular Integrated Development Environment).

Below this we have a user input area. This area is for the commands that a user would typically type into the terminal during the running of the program. We included this because the code is not compiled and run in the browser. (This feature would require web assembly. More about that later). Instead, we decided to use an Application Programming Interface to compile and run the program.

Once the user’s question, code, and chat history are submitted, it is sent to the back-end for processing. The back-end was originally designed in C++, but we ran into issues while attempting to implement our vector database library. Due to the time constraints of the project we decided to migrate to Python because of its more robust selection of Artificial Intelligence libraries.

The back-end has three main components. The first handles the loading of all context information into the database, the second processes the user's questions, and the third is responsible for compiling and running the user’s code. The overview of how all the data is processed is in Figure 2.

When initially started, the back-end gathers the data. This context data is all stored in text documents, which are placed in folders according to chapter. Once gathered, the data is then sent to Ollama. Ollama then uses Llama 3.2 to process the context documents into embeds (can probably grab a reference and explain what embeds are here). For our context documents, we used chapters one through six of the ZyBooks textbook, CSC 108: Computer Science I [[[APA-format title]]]. Once Ollama has generated the embeds, the back-end stores them in a ChromaDB database, along with their corresponding context documents and titles. We chose ChromaDB due to its high functionality and ease of integration into python. Finally, the back-end starts the Flask server (a webserver library for python) and begins listening for questions from the front-end. Once all of this is complete, the back-end is ready to answer questions.

 ***Figure 2*** – The Chart above show what information is sent where.

When the question is received from the front-end, it is packaged with extra information. Amongst other items, this extra information includes the chat history, user’s code, and chapter. The first task for the backend is to separate out the information. The app parses the text information into JSON, slicing it into individual elements. It achieves this by using the native JSON parsing tools in python. The user’s question is then processed into an embed. This is then used to search the ChromaDB database.

ChromaDB efficiently identifies and retrieves the most relevant chunks to the query. This is what creates the accuracy and contextual precision needed to help the student. This remedies the typical downfall of generative AI simply generating an answer from metadata without concise context. Again, the retrieval process builds the foundation for an optimal educational experience for the student. In this way, it is able to create a user-friendly environment for beginners. Once ChromaDB returns a document, the generation process can begin.

The generation passes several bits of preliminary information to Ollama, running the Llama 3.2 model. We use the LangChain library to accomplish this, simplifying the formatting of the requests to Ollama. The app first passes the following prompt as a system message “You are a Tutor for CSC108 - Intro to C++. You are answering questions about C++ coding. Use the following pieces of context. to answer the question at the end. If there are No relevant documents found, ask for clarification instead of answering. If you don't know the answer, just say that you don't know, don't try to make up an answer. If it is a vague question, ask for more information. Whenever possible use the Socratic method.” The app then passes the context document as a system message. At this point, it passes the chat history, originally received from the front end. With all the preliminary information sent, it is time to begin the generation by passing the user’s question.

Once the generation is done, Ollama returns the response to our back-end. The back-end then adds the document name (citing chapter and section) to the end of the message as a source. This informs the student where to look in the textbook for further information regarding this topic. This process facilitates accurate and sourced answers, to better help the student. The completed message is then sent back to the front-end, which displays it to the user, completing the interaction.

The final component of the back-end is to handle processing the user’s code and return the results. This is achieved by, once again, parsing the JSON information sent by the front end. At this point, it differs by sending a request to the JDoodle Application Programming Interface. This request contains the user’s code and desired program input. JDoodle then compiles and runs the user's code, returning a response with the program's output. Once our back-end receives the response, it sends that information to the front-end to be displayed to the user.

*(beginning of the section about the modularity of the backend)*

One of our goals was to create our tutor app to be flexible, allowing it to be used for other courses and subjects in the future. We designed our back-end to be easily integrated with other subjects matter. By avoiding the hard-coding of specific rules for C++ education, it allows the app to be used in other subjects by simply swapping out the context information. This can be achieved by swapping out the .txt documents with those containing the desired subject material By configuring our back-end to read the context from these .txt documents and store it in a ChromaDB database, it can use any material that a tutor or instructor provides the system. In this way, our tutor app can be used for any subject or course, maintaining a consistent and efficient experience for the students.

The biggest constraint with Results-Augmented Generation is that text documents need to be small in size to be properly searched. This prohibits the use of a single file containing an entire textbook. We solved this issue by separating the data into separate files, each including only one chapter section. (i.e 6.1, 6.2, etc.))

The embedding process remains uniform regardless of the context information. This means that no changes are required in code. Code modification would only be required if more than 6 chapters were needed.

Because our back-end is made as an Application Program Interface, it allows anyone to use it with their own front-end.

**Furthermore, this modularity offers significant advantages in terms of scalability. Upgrades to the embedding model or improvements in the vector database can be implemented centrally, benefiting all courses that use the system. This ability to serve multiple subjects with the same core engine reduces redundancy and enhances the overall reliability of the tutoring system. (check this paragraph – too good)**

The more we can measure the performance of the application, the more we can improve upon its functionality. One of the main ways to do that is to efficiently get detailed feedback from the students.

*(beginning of carbon efficiency & privacy benefits of local LLM)*

As Large Language Models power more applications their environmental impact has come under examination. Cloud providers commonly allocate multiple Graphics Cards to satisfy service-level objectives for latency and throughput. However, for typical tutoring workloads which consist of short prompts this strategy can backfire. This approach substantially increases carbon emissions.

The LLMCO2: Advancing Accurate Carbon Footprint Prediction for LLM Inferences paper quantifies this effect. Fu, Chen, Zhou, Li, and Jiangdemonstrate that, using a Bloom-7b1 inference with a 64-token prompt and batch size of 1, adding GPUs actually raises total carbon footprint. (**All-reduce**. Given the huge memory demands of LLMs and the limited capacity of individual GPUs, multiple GPUs connected via PCIe or NVLink (Patel et al., [2024](https://arxiv.org/html/2410.02950v1#bib.bib25)) are crucial for LLM inferences. Tensor parallelism (Aminabadi et al., [2022](https://arxiv.org/html/2410.02950v1#bib.bib3)) splits tensors across GPUs and replicates all layers, providing a significant speedup over other parallelism strategies. To support tensor parallelism, two all-reduce kernels are incorporated into each transformer layer. An all-reduce kernel (Hidayetoglu et al., [2024](https://arxiv.org/html/2410.02950v1#bib.bib11)) consists of a reduce-scatter operation followed by an all-gather operation, as shown in Figure [3](https://arxiv.org/html/2410.02950v1#S2.F3). For instance, a 4×4 matrix, evenly distributed across four GPUs (each holding a column), undergoes reduce-scatter, where each row is assembled and summed on one GPU, followed by all-gather, where the summed values are shared across all GPUs.)The all-reduce??? communication required for tensor parallelism across multiple devices introduces latency and extra energy use, outweighing any per-GPU efficiency gains(Zhenxiao Fu). Indeed, although larger batches (e.g., batch size 4 with 1K tokens) can benefit from two or four GPUs by spreading computation, small-scale queries common in interactive tutoring see per-GPU carbon overhead climb steeply as device count increases.

In contrast, hosting the Large Language Model locally on a single GPU avoids these cross-device costs entirely. Without networked communication between GPUs, inference remains streamlined: the model loads once, processes the prompt, and returns a response, minimizing idle cycles and interconnect traffic. This configuration aligns directly with instructional use cases, where students issue brief, focused questions rather than large batch inference jobs.

Furthermore, SPROUT’s “generation directives” approach demonstrates that local inference can cut carbon emissions even more—by over 40%—by controlling output verbosity based on regional grid carbon intensity and prompt requirements. (Baolin Li) Together, these findings indicate that for scenario-specific workloads like a tutoring assistant, a self-hosted LLM on a single GPU not only preserves data privacy but also achieves substantially lower per-inference carbon emissions than default cloud deployments. Educators and institutions aiming for sustainable AI should therefore consider local hosting of appropriately sized models as a greener alternative to multi-GPU cloud inference.

*“The performance benefits of cloud based LLMs may come at a cost of privacy. Privacy risks in LLMs arise from their inherent capacity to process and generate text based on extensive and diverse training datasets. These models, like GPT-3, may inadvertently capture and reproduce sensitive information that exists in training data, potentially posing privacy concerns during the text generation process. Issues such as unintentional data memorization, data leakage, and the potential disclosure of confidential information or PII are key challenges.” (Das)*

In a setting like our tutoring app, utilizing a locally hosted Large Language Model basically restricts any outside access to the sensitive data of the users. It also prevents malicious code being supplied to the users query thus preventing various types of software and hardware attacks.(I have no idea)

*(Beginning of future research section)*

Throughout our project we ran into several things that would be good extensions of our project, but were out of scope due to time constraints. We would like to include more context from the code by creating a web assembly and live terminal for running the code, that way we can collect a log of the run program as context. This would also allow us to see the errors directly, making troubleshooting code with the tutor easier and more accurate.

Since web assembly is essentially compiling code into an executable for a browser, you would need to take the source code for a C++ compiler, G++ for example, and compile it with a web assembly compiler. Once you did that you would have to figure out how to take the code, save it as a file in the web assembly file structure, compile it using the compiler, and then finally run it, while being able to access its input and output live. This would likely mean you needed to also compile a terminal application in web assembly.

Another area of possible research would be to use CORAG instead of RAG, allowing for multiple context searches instead of one. This would allow things like the C++ online refrence to be used in addition to the textbook, as well as other sources. This would make the tutor even more accurate than it is currently.

More work into making it more Socratic.

More testing with different models would be a good area of reserch as well. Since we only used Llama 3.2 testing to see if there is a better model for the question answering would be good. Also you could test and see if splitting the embeds into a unique embed focuse model wouuld incresease accuracy.

The cost of premium AI services of outpaces the speed benefits. We found Llama to be an excellent when weighing speed, flexibility, and cost.

**MY OUTLINE**

Approach

Why choices (RAG, LLM, OLlama, Llama)

The advantage of our app is that it is set up to be able to use any model avalible for Ollama. THis means that you can use a bunch of different models, and with the Results-Agumented Generation, increase their accuracy. This allows for you to reach near comercial qualtiy, but at a very reduced cost.

Another study by Wang and Ramon **(Quantitative Evaluation of Using Large Language Models and Retrieval-Augmented Generation in Computer Science Education)** quantified the performance of different AI models and how effective they were by a cost-effective analysis for instructors. "Implementing RAG enhances the ability of LLMs to answer context-specific questions accurately. This improvement is particularly noticeable in models with integrated course materials and pre-answered question databases and allows open-source models to close some of the gap with GPT-4."

Moreover, they found that advanced model Large Language Models did outperform open-source models in Q&A tasks. However, the performance gap wasn’t significant enough to justify the cost-benefit of using locally hosted open-source models. The needs of the instructor/students should be considered in this regard.

Though we took a unique path, a significant inspiration was Ragman

UC-I Study (Ma…)

Thus, avoiding hallucinations. Also, system prompts allow it to be guided against responding with hallucinations if the model doesn’t know the answer or doesn’t have enough information. This is an important part of the process as well.

**FUTURE IMPROVEMENTS**

Six chapter limit

**PROBLEM WORDS**

Course/class

AI

RAG Man/RagMan

App/Application/Tutor App

Info

Backend/back-end

# References

Baolin Li, Y. J. (2024, November). *SPROUT: Green Generative AI with Carbon-Efficient LLM Inference, aclanthology.org*. Retrieved April 22, 2025

Das, A. W. (2024, November 14). *Security and Privacy Challenges of Large Language Models: A Survey* . Retrieved April 22, 2025

Zhenxiao Fu, F. C. (2024, October 3). *LLMCO2: Advancing Accurate Carbon Footprint Prediction for LLM Inferences*. Retrieved from arxiv.org: https://arxiv.org/html/2410.02950v1#:~:text=Throughout%20its%20lifecycle%2C%20a%20large%20language%20model,GPU%20types%20and%20quantities%20to%20meet%20diverse