Generative AI and Learning:

Using Retrieval-Augmented Generation (RAG) for C++ Tutoring

Ben Carpenter and James Stevens

Computer Science Department, Quinsigamond Community College

https://github.com/NobleWolf42/CSC108-Tutor

Abstract

By utilizing advancements from the field of AI, we aim to develop a tutoring application 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 back-end 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 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

Our objective for this project is the design and implementation of an Artificial Intelligence (AI) based tutor application that effectively addresses beginner-level C++ (C Plus Plus) programming questions for students in the introductory computer science course. We chose C++ as the focus of our application 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. An accurate and effective tutor application will help enable a beginning student to absorb the new material more effectively.

We designed our app in two parts, the front-end and the back-end. Written in Hypertext Markup Language (HTML), the front-end provides a way for end users to interact with the function and features of the back-end. The front-end is simply a means of accessing the back-end. It is the back-end that powers the application, executing the commands and returning the results.

The back-end, implemented in Python, serves as the engine of the application. It processes incoming questions that users submit via the front-end and communicates with the Llama 3.2 Large Language Model (LLM) via a Representational State Transfer Application Programming Interface (REST API). In this way, we ensure 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, uniquely tailored to their queries.

By using a Retrieval-Augmented Generation approach to providing targeted feedback, our goal is to develop a basic, but effective Artificial Intelligence 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 Llama 3.2 Large Language Model through Ollama's local REST API.

Retrieval-Augmented Generation is the cornerstone of our project. It is a generative Artificial Intelligence process that combines the power of Large Language Models (LLMs) with a curated supply of information. Retrieval-Augmented Generation supplies the designated Large Language Model with contextual information from a data store. The supplied information is specifically related to a user's query. The Large Language Model then uses this contextual information to inform and guide the generative process, giving the Artificial Intelligence a stronger knowledge of the topic at hand. (Liu et al., 2024)

Chapter 2

Related Works

2.1 Computer Science Education Challenges

Computer science is presently facing growth pains. As the world experiences a growing reliance on software in all aspects of modern society, the demand for programmers increases. This encourages more students to consider software careers, increasing interest in the field. As a result, many schools are seeing record undergraduate enrollment in computer science. The problem with this growth is a lack of qualified faculty available to instruct the students gained from this increase. The combined force of these factors is beginning to stress computer science education. As Ma, Martins, and Lopes — instructors at the University of California: Irvine — point out, “Providing individualized support to many students in introductory courses, especially regarding mastery of complex material, has been challenging.” A strategic use of Artificial Intelligence could further the educational reach of the faculty that now exist, reducing the need for an instructor’s direct involvement in simpler questions. (Ma et al., 2024)

Without a working knowledge of computer concepts, many beginning computer science students need help to be able to learn and review the introductory information. Ma, Martins, and Lopes conducted a study consisting of 455 student participants. They investigated the use of Artificial Intelligence tutors within the context of computer science education. The study introduced five RAGMan tutor applications to assist the students with their supplemental homework assignments. These tutor applications were designed to give guidance, rather than solutions. Using these applications, the students developed experience by participating in a more practical process, ultimately finding their own answers. (Ma et al., 2024)

Their research suggested that, “AI [Artificial Intelligence] tutors can positively impact student success and provide important help, especially to students who would be struggling in challenging courses.” (Ma et al., 2024) They concluded that there was a statistically significant increase in the number of students that continued through the degree path after introducing the RAGMan tutors. They also found the student feedback to be very positive, demonstrating a positive user experience. This is significant, because greater user satisfaction will help to ensure a broader use of these tools, further impacting student success. (Ma et al., 2024)

Creating a virtual personal assistant for computer science students is very promising based on the results of such research. Our tutor application seeks to provide a pressure-free, efficient, and personalized tutor experience for introductory students. This is done by drawing specifically on trusted course materials. By continually prioritizing the feedback and interactions of the students, we can further enhance these learning tools, making them more effective and user friendly.

2.2 Cost Effectiveness

Our tutor application also 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. In addition, students enrolled in schools with high class populations can find it difficult to get access to tutoring help from other students or faculty. Our application could also provide students with active, accurate support outside a tutor’s or professor’s available hours. This would make extended support possible as students begin to establish their basic skills. Both affordable and accessible, the application would be a great supplement to traditional teaching resources, such as textbooks and class lectures.

When designing our application, we kept cost in mind. In the study, “Quantitative Evaluation of Using Large Language Models and Retrieval-Augmented Generation in Computer Science Education,” Wang and Lawrence (2025) determined the cost effectiveness of different Large Language Models. Their analysis quantified both closed-source and open-source models, measuring cost-to-performance in an educational setting. The authors found that closed-source Large Language Models outperformed their open-source counterparts when answering questions. However, the performance gap was not considered significant enough to justify the extra cost of using remotely hosted, closed-source models. Therefore, they determined that the cost of premium Artificial Intelligence services outpaced the speed benefits.

According to the study, "Implementing RAG [Results-Augmented Generation] enhances the ability of LLMs [Large Language Models] 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" (Wang & Lawrence, 2025). After weighing the speed, flexibility, and cost of each model, we decided on a combination of a locally hosted Llama 3.2 model and Results-Augmented Generation.

As a learning tool, our application would be a cost-efficient choice for educational institutions to implement. By running the Llama 3.2 model on Ollama — a free, locally hosted Artificial Intelligence — we were able to reduce the costs often associated with generative Artificial Intelligence. Most large language models cost per token, increasing operating costs with each use. Ollama, however, 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.

2.3 Environmental Impact and Privacy Concerns

As Large Language Models begin to power more applications, their environmental impact has come under increased examination. Cloud providers commonly allocate multiple Graphics Processing Units (GPUs) to satisfy service-level objectives for latency and throughput. However, for typical tutoring workloads, which consist of short prompts, this strategy can backfire by substantially increasing carbon emissions.

The *LLMCO$_2$: Advancing Accurate Carbon Footprint Prediction for LLM Inferences* paper quantifies this effect, measuring the total rise on the actual carbon footprint when adding successive GPUs to process questions with artificial intelligence. The authors used a Bloom-7b1 inference, with a 64-token prompt and batch size of one, to simulate a short question submitted to the specific model. This determines the effect of adding GPUs in a context similar to ours. Running this prompt with GPU configurations of one, two, and four, they found that adding GPUs increased carbon emissions exponentially. This is due to the communication overhead required between GPUs to stay in sync. As the inference size increases, this overhead decreases. By the time the researchers reached batch sizes of four with 1,000-token prompts, the carbon savings favored multiple GPUs. They concluded that one GPU is better for smaller prompts, while larger prompts are (up to a point) better suited to multiple GPUs. (Fu et al., 2024)

Since tutor questions are smaller in nature, one GPU in a locally hosted configuration is more economically friendly for our purposes. This is because tutoring most often involves brief, focused questions rather than large batch jobs. By keeping the GPU numbers small, the idle cycles and interconnect traffic are kept to a minimum, keeping carbon emissions down.

Another down side of using a cloud-based, Large Language Model is privacy concerns. “The performance benefits of cloud based LLMs [Large Language Models] 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

[Personal Identifying Information] are key challenges" (Das et al., 2025). These privacy risks can be offset by hosting the Large Language Model locally, as it prevents end-user data being sent to a third-party.

Together, these findings indicate that, for scenario-specific workloads like a tutoring assistant, a self-hosted Large Language Model 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 Artificial Intelligence should therefore consider local hosting of appropriately sized models as a greener alternative to multi-GPU cloud inference.

2.4 Accuracy

When looking to implement an Artificial Intelligence tutor, accuracy is of paramount importance. The biggest threat to this accuracy is Artificial Intelligence's propensity for hallucination. Hallucination is when an Artificial Intelligence 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., 2023). 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" (Liu et al., 2024). Using Retrieval-Augmented Generation allows us to overcome this important hurdle.

Chapter 3

Problem Statement

Accurate answers are crucial for a tutor. Therefore, the hallucination issue is the most important problem to overcome. By implementing Retrieval-Augmented Generation, we are able to restrict the data pool, thus limiting the Artificial Intelligence's answers from straying into hallucination. This should virtually eliminate the possibility of hallucination with the use of our app.

With accuracy issues resolved, our project is able to provide an introductory computer science student with a course-specific learning tool, adding value to students, while potentially reducing teacher workload. The application is not intended to replace textbooks or teachers, but to supplement and support currently established methods of education. By focusing on specific material as the basis for our tutor application'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 in computer science, 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; Jupyter Notebook and Zoho also function similarly, albeit at varying levels of complexity. 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 be flexible enough 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, our 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 course 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 other classroom materials. This feature gives the application the ability to deliver this information in a more engaging and personal way for each student. By using relevant coding examples, it can offer a more efficient method in reaching inexperienced students with supportive information, allowing them to better digest the textbook information.

This application can also help build a student’s 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, facilitate quicker and more stress-free progress, and promotes eventual mastery of the basics of C++ programming. As students better understand the material, they will gain confidence, making it more likely that they continue in the degree path. (Ma et al., 2024)

Chapter 4

Approach and Methodology

4.1 Application Overview

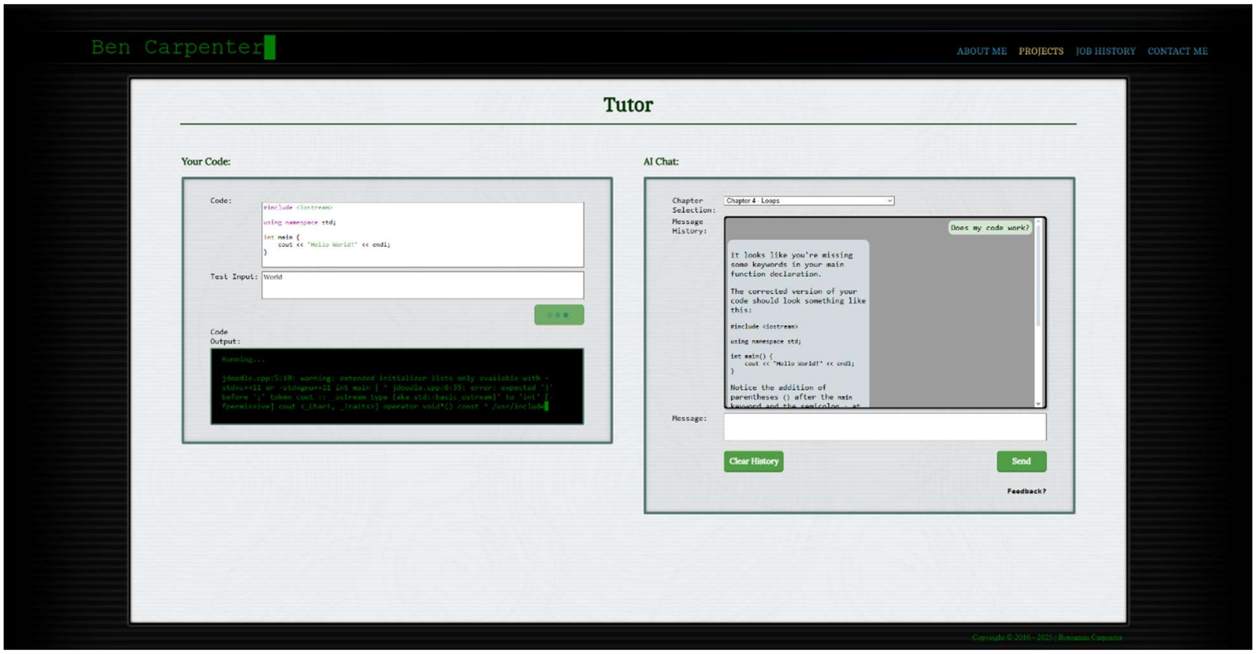
Our application is made with two main modules, the front-end and the back-end. The front-end handles displaying and storing messages as well as user input. This is the interactive portion accessed by the end user. The back-end serves as the workhorse of the application. It handles the context information, storage, and lookup, as well as communication with the Artificial Intelligence.

We chose Ollama for our Artificial Intelligence engine because of its ability to host a variety of open-source Large Language Models. It allows the in-use model to be easily swapped for another, without the necessity of modifying the commands sent to the engine. Once we settled on using Ollama, we decided to use the Llama 3.2 model because of its size, efficiency, and its ability to be run on our hardware. Lastly, we built our backend as an API, allowing administrators the option of integrating our application into new and existing platforms, software, and services.

4.2 Front-End

For our front-end (see figure 1), we opted for a user-interface built with Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS). 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 front-end with several core features. These include messaging the Artificial Intelligence 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 applications. Our intention is to create a more intuitive user experience, shortening the learning curve for many users. This will make it more natural for people to use. 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 messaging interface is the primary means of interaction with the tutor application.



Code:

```
#include <iostream>

using namespace std;

int main {
    cout << "Hello World!" << endl;
}
```

Test Input: World

Code Output:

```
Running...

jdoodle.cpp:5:10: warning: extended initializer lists only available with -
std=c++11 or -std=gnu++11 int main { ^ jdoodle.cpp:6:35: error: expected '}'
before ';' token cout ::__ostream_type {aka std::basic_ostream}' to 'int' [-
fpermissive] cout <_CharT, _Traits>] operator void*() const ^ /usr/include
```

Chapter Selection: Chapter 4 - Loops

Message History:

Does my code work?

It looks like you're missing some keywords in your main function declaration.

The corrected version of your code should look something like this:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello World!" << endl;
}
```

Notice the addition of parentheses () after the main keyword and the semicolon ; at

Message:

Figure 1 *– This image shows the front-end of our project.*

This chat area not only takes the input, but incudes several core functions. These include remembering input (the user's code, question, and chat history) and passing that information to the back-end alongside each new question. This allows the tutor to have chain-of-thought, remembering a user's conversation and responding to normal human language by referencing previously discussed information.

At the top of the chat area 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 (IDE). Doing this provides a familiar experience for people who are used to working in that setting. 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 in real time. The way we do this is with two layered elements. The bottom element contains the formatted text, while the top element containing the user input is transparent. To further the functional continuity, we overrode the default tab action to instead place four spaces, similar to Visual Studio Code — a popular IDE.

Below this we have a user input area. This area accepts the commands typically typed into the terminal during the running of the program. This is necessary because the code is not compiled and run inside the browser. Compiling the code in the browser would require web assembly (more about this in the Future Works section). Instead, we decided to use an API to compile and run the user’s code.

Through these combined elements, the front-end provides user access to the program. The information entered into these elements makes up the user’s query. Upon submission, the front-end begins to gather the required information. Once the information is gathered, it is sent to the back-end for processing. At this point, the front-end enters a standby mode, awaiting a response from the back-end. This response will be displayed upon receipt.

4.3 Back-End

4.3.1 Overview

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. These libraries enabled us to make a robust back-end within the allotted time.

The Python 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 this data is processed is shown in Figure 2.

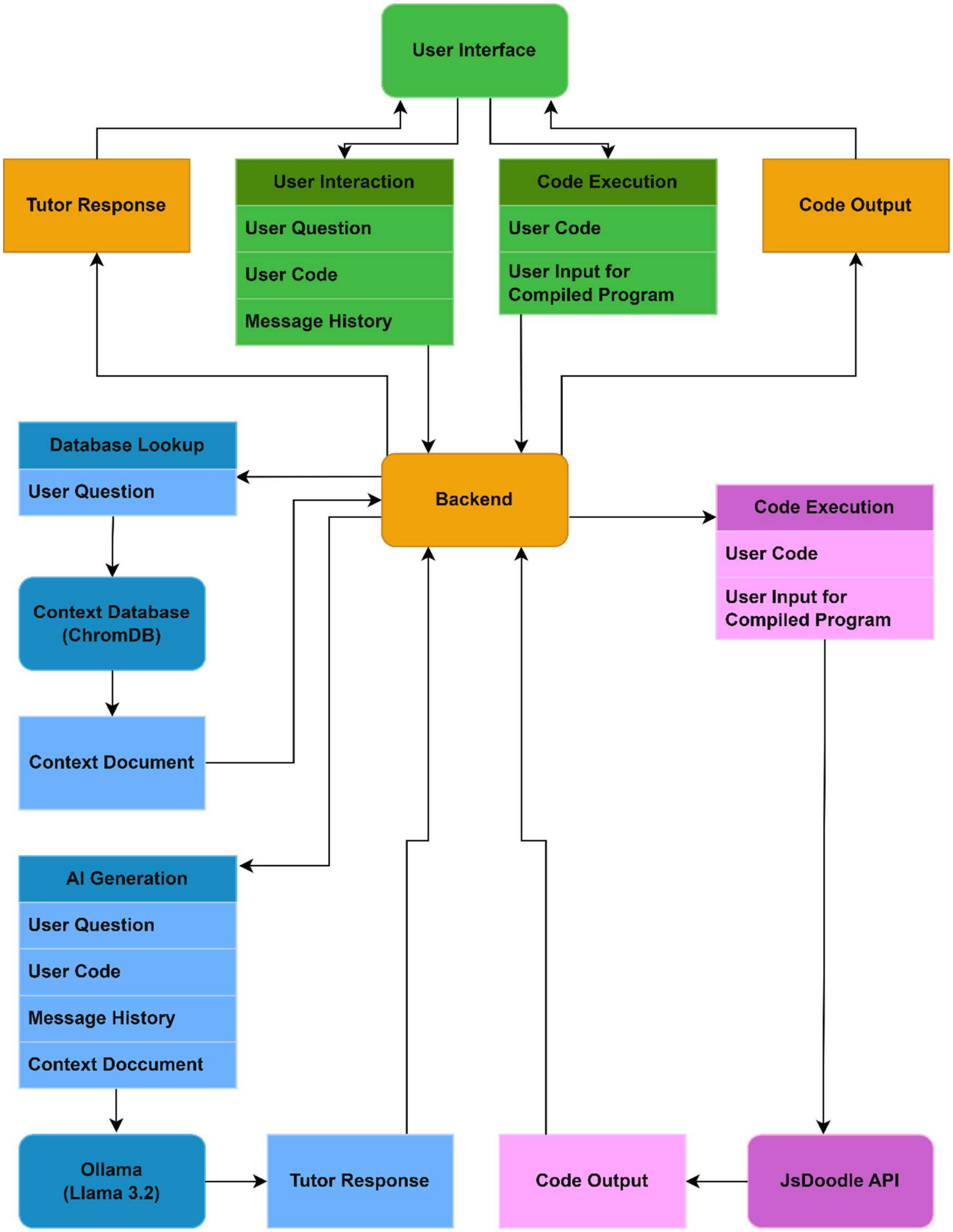


Figure 2 *– The Chart above illustrates the flow of information through the application. The green represents the front-end, while the orange is the back-end. The blue represents the answer generation process and the pink represents the code compilation and execution process.*

When initially started, the back-end gathers the context data. This context data is stored in text documents, which are placed in folders according to chapter. Once gathered, the data is sent to Ollama, which then uses Llama 3.2 to process the context documents into embeds. Embeds are lists of numbers that represent information. These lists better facilitate the searching of the information.

For our context documents, we used chapters one through six of the ZyBooks textbook, “CSC 108: Computer Science I”. 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 because of 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.

When a question is received from the front-end, it comes in a package that contains extra information. Amongst other items, this extra information includes the chat history, user’s code, and chapter. The first task for the back-end is to separate out the information. The application parses the text information into JSON (JavaScript Object Notation), 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 embed is then used to search the ChromaDB database.

ChromaDB efficiently identifies and retrieves the chunks of data that are most relevant to the query. This is what creates the accuracy and contextual precision needed to help the student. This remedies the typical downfall of generative Artificial Intelligence 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 process 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 application 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 application 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 handles the processing of the user’s code and returns 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 API. 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.

4.3.2 Modularity

One of our goals was to create our tutor application 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 subject matter. By avoiding the hard-coding of specific rules for C++ education, it allows the application to be used in other subjects by simply switching 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 application 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. Section 1.2, 2.3, 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 (more about this in the Future Works section).

This modularity also offers significant advantages in terms of scalability. Upgrades to the embedding process and improvements related to the vector database can be implemented centrally, benefiting all courses using the system. The ability to serve multiple subjects with a central core engine also reduces both redundancy and maintenance overhead for the tutoring system.

The more we can measure the performance of the application, the more we can improve upon its functionality. This can be achieved by receiving detailed feedback from the students. Since the goal is to create a tool to improve a new student's experience, any measure of the application's effectiveness would require feedback from the end users. By regularly gathering this information, administrators would be able to gauge current efficiency and decide on future needs. The application could then be modified to meet those needs.

Chapter 5

Future Works

During the development of our project, we encountered several points of research that would make good extensions of our project. Unfortunately, time constraints prevented us from pursuing them. One possibility would be to include more context from the code by creating a web assembly and live terminal for running the code, thus making it possible to 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 both easier and more accurate.

Since web assembly is essentially compiling code into an executable for a browser, one would need to take the source code for a C++ compiler, G++ for example, and compile it with a web assembly compiler. After compiling the C++ compiler, the user's code would somehow have to be saved as a file in the web assembly file structure. Once this was accomplished, the user's code would have to be compiled using the compiler and finally run, all while maintaining live access to its input and output. To run all of this would likely require the compilation of a terminal application in web assembly.

Another area of possible research would be to explore the replacement of Retrieval Augmented Generation with Chain-of-Retrieval Augmented Generation. Instead of being limited to a single context search the initial context of the question would be further refined with successive context searches. This would allow the textbook context to be used to search within other sources, such as the C++ online reference, further improving the tutor's accuracy.

One feature that we would like to see explored is the application of a Socratic approach within the tutor's responses. By responding with prompts, rather than answers, the tutor will help students better internalize the information. This would further strengthen the tutor's educational potential.

Additionally, removing the six-chapter limit on the back-end context information is an improvement that could open up broader application. This task should be relatively simple. It would require changing the manual process for loading the documents into one that is programmatic, thus improving the ease at which the information could be swapped out.

Of course, another useful area of research would be to test other Large Language Models. We chose Llama 3.2 due to its advantages within our context and constraints, however, other models should also be tested to verify our decision. Other testing could involve separating the embed processing from the main model, moving their generation to an embed focused model.

Chapter 6

Conclusion

This project set out to create a tutor application that could be used in a variety of settings for many different subjects and courses. We wanted to make it flexible, allowing it to be used with any current or future front-end with minimal issues. In order for it to be useful to students, it needed to be accurate, avoiding hallucinations. We also included the option to run it locally, if desired, to increase privacy, affordability, and eco-friendliness.

In this regard, our project was a success. Our tutor application is capable of answering introductory C++ questions with context and accuracy, including citations where the answer was found, allowing the student direct access to further explore the material. We were able to accomplish this in a package that is easy to set up and run on most computers. Its back-end is capable of receiving questions from any front-end that is formatted according to the documentation.

By using Ollama, we provide the option to run the application either locally or using a cloud-based implementation. It can also accommodate any choice of Large Language Model. This allows administrators to customize the setup according to each individual infrastructure. We felt this was an important step in widening the compatibility of our application.

In the end, our project was able to accomplish the goals we set for it. Still, future research is needed to make it even more efficient and accurate. Other Large Language Models should be tested to determine the optimum model for the tutor. The educational potential could be improved by making answers more Socratic in nature. Before making it a core part of a classroom, the back-end should be made more user friendly for administrators. These improvements will strengthen the application as it better fulfills its academic purposes.

As demand grows in the field of computer science, so does the demand for a computer science education. While Artificial Intelligence tutoring applications cannot meet that demand by themselves, such applications can help augment traditional education methods. This could help facilitate greater student success and lessen the demand on teachers. Our hope is to see Artificial Intelligence responsibly used to help provide students with a better learning experience.

References

Das, B. C., Amini, M. H., & Wu, Y. (2025, Febuary 10). Security and Privacy Challenges of Large Language Models: A Survey. *ACM Computing Surveys, Volume 57, Issue 6*. Retrieved April 22, 2025, from https://arxiv.org/abs/2402.00888

Fu, Z., Chen, F., Zhou, S., Li, H., & Jiang, L. (2024, October 3). *LLMCO2: Advancing Accurate Carbon Footprint Prediction for LLM Inferences*. arxiv.org. https://arxiv.org/abs/2410.02950

Li, J., Cheng, X., Zhao, X., Nie, J.-Y., & Wen, J.-R. (2023, October 07). *HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models*. https://openreview.net. https://openreview.net/forum?id=bxsrykzSnq

Liu, S., Yu, Z., Huang, F., Bulbulia, Y., Bergen, A., & Liut, M. (2024). Can Small Language Models With Retrieval-Augmented Generation Replace Large Language Models When Learning Computer Science? *ITiCSE 2024: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1.* Milan, Italy: Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3706599.3720240

Ma, I., Martins, A. K., & Lopes, C. V. (2024). Integrating AI Tutors in a Programming Course. *SIGCSE Virtual 2024: Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1.* Virtual Event, NC, USA: Association for Computing Machinery. https://arxiv.org/abs/2407.15718

Wang, K. S., & Lawrence, R. (2025). Quantitative Evaluation of Using Large Language Models and Retrieval-Augmented Generation in Computer Science Education. *SIGCSETS 2025:*

Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1. Pittsburgh, PA, USA: Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3641554.3701917