

**LEVERAGING LARGE LANGUAGE MODELS TO SUPPORT  
INDIVIDUALIZED LEARNING**

A Dissertation  
Presented to  
The Academic Faculty

By

Lindsey Lubin

In Partial Fulfillment  
of the Requirements for the Degree  
Masters of Science in the  
School of Electrical and Computer Engineering  
College of Engineering

Georgia Institute of Technology

May 2025

© Lindsey Lubin 2025

**LEVERAGING LARGE LANGUAGE MODELS TO SUPPORT  
INDIVIDUALIZED LEARNING**

Thesis committee:

Dr. Fabian Monroe  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Linda M Wills  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Ying Zhang  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Date approved: April 24, 2025

## ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Fabian Monroe, for his steadfast support and thoughtful feedback. His guidance has been instrumental not only to this thesis but also to my overall growth as a researcher. I am also thankful to my committee members, Dr. Linda Wills and Dr. Ying Zang, for generously offering their time, insight, and encouragement throughout this process. Their perspectives helped sharpen my thinking and made this a stronger piece of work. Thank you to my lab mates at Astrolavos in the Center for Cyber Operations Enquiry and Unconventional Sensing (COEUS). Kevin "V" Valakuzhy, thank you for being my go-to person for debugging issues. Lydia Han, thank you for your emotional support. Mac Malone, your dissertation and your work on Riposte laid the foundation for so much of what I built, and I'm grateful for your trailblazing.

I'm also thankful to the incredible academic and administrative staff in the School of Electrical and Computer Engineering. Your encouragement and laughter helped me get through each day. A very special thanks to Sandia National Labs for financially supporting my studies and making it feasible to explore this work in depth.

To my parents, thank you for always pushing me and for being my anchors. Janine, thank you for always picking up the phone, no matter the hour, and for reminding me that I could make it happen. And to Cashmere, my feline companion through late nights of writing, thank you for the purrs and the company.

## TABLE OF CONTENTS

<b>Acknowledgments</b> . . . . .	iii
<b>List of Tables</b> . . . . .	vii
<b>List of Figures</b> . . . . .	viii
<b>List of Acronyms</b> . . . . .	x
<b>Summary</b> . . . . .	xi
<b>Chapter 1: Introduction</b> . . . . .	1
<b>Chapter 2: Related Work</b> . . . . .	5
2.1 Traditional Feedback in Programming Education . . . . .	5
2.2 Large Language Models in Programming Education . . . . .	6
2.2.1 Student Experiences . . . . .	6
2.2.2 Instructor Insights and Concerns . . . . .	7
2.2.3 AI Assistants in Controlled Environments . . . . .	8
2.3 Prompt Engineering . . . . .	9
2.3.1 Common Prompting Techniques . . . . .	9
2.3.2 Real World Applications of Prompt Engineering . . . . .	10

<b>Chapter 3: Methodology</b> . . . . .	12
3.1 Design Principles . . . . .	13
3.2 System Development . . . . .	14
3.2.1 LLM Selection . . . . .	14
3.2.2 Classroom Framework Selection . . . . .	17
3.3 System Evaluation . . . . .	19
3.3.1 Data Collection . . . . .	19
3.4 Final Remarks . . . . .	20
<b>Chapter 4: Implementation</b> . . . . .	22
4.1 System Architecture . . . . .	22
4.2 Integrating Codehelp into Riposte . . . . .	22
4.2.1 Class Context . . . . .	24
4.2.2 Response Flexibility . . . . .	26
4.2.3 Student Session Analysis . . . . .	27
4.3 Final Remarks . . . . .	31
<b>Chapter 5: Evaluation</b> . . . . .	34
5.1 Student Scenarios . . . . .	34
5.2 Class Context . . . . .	36
5.3 Code Flexibility . . . . .	38
5.3.1 Data Analysis . . . . .	38
5.3.2 Script Profiling . . . . .	38
5.4 Student Session Summary . . . . .	41

5.5	Final Remarks . . . . .	41
<b>Chapter 6: Results</b>	. . . . .	<b>42</b>
6.1	Class Context . . . . .	42
6.1.1	Student Scenarios . . . . .	43
6.1.2	Measuring Specificity . . . . .	49
6.2	Code Flexibility . . . . .	50
6.2.1	Improving Student Code Performance . . . . .	50
6.3	Student Session Summary . . . . .	52
6.4	Final Remarks . . . . .	55
<b>Chapter 7: Discussion</b>	. . . . .	<b>57</b>
7.1	Limitations . . . . .	58
7.2	Future Work . . . . .	59
7.3	Final Remarks . . . . .	63
<b>Chapter 8: Conclusion</b>	. . . . .	<b>64</b>
<b>References</b>	. . . . .	<b>66</b>

## LIST OF TABLES

3.1	Summary of Modifications and Enhancements to Codehelp and Riposte . . .	20
6.1	LLM Specificity Classification by Assignment Type . . . . .	43
6.2	Average Response Score by Query Type . . . . .	43
6.3	LLM Classification Metrics by Query Type . . . . .	43
6.4	Average LLM Specificity by Query Type . . . . .	44
6.5	Average Number of Built-in Function Calls per Solution . . . . .	51
7.1	Daily LLM Token Usage Summary . . . . .	61
7.2	Daily Cost Breakdown per Request . . . . .	61
7.3	Pearson Correlation Between Cost and LLM Usage Metrics . . . . .	61

## LIST OF FIGURES

3.1	Codehelp’s original response workflow [22] . . . . .	16
3.2	Riposte’s Mission Page [24]. <b>A</b> shows the signed-in student’s username. <b>B</b> shows the guide window. <b>C</b> shows the mission page. <b>D</b> shows our contribution of help button which allows students to directly access Codehelp. <b>E</b> shows trophies that can be achieved by completing certain milestones. <b>F</b> shows the due date for the assignment and the student’s grade. . . . .	18
4.1	Codehelp’s improved response workflow. Our additions are shown in red. .	23
4.2	Codehelp’s improved Instructor page workflow. <b>A</b> shows the Assignment contexts table. <b>B</b> Shows the corresponding addition to the Queries table. <b>C</b> shows the Student Session Analysis. . . . .	24
4.3	Codehelp’s create a context in class page [22]. Our addition is the Allow Code Responses field . . . . .	28
4.4	Student Session Analysis Response Workflow . . . . .	29
4.5	Example Student Session Analysis . . . . .	32
4.6	Home Page Frequently Asked Questions Mock-up for the main page. <b>A</b> shows an example class name. <b>B</b> shows the active assignment. <b>C</b> shows optional next assignment and previous assignment buttons. <b>D</b> shows a resources tab. <b>E</b> shows a button directing to the AI teaching assistant help form. <b>F</b> shows the common questions with the most common question listed first. <b>G</b> shows the number of students asking the selected question. .	33
6.1	<b>LLM response from <i>Passwords 1</i></b> . The model offers suggestions for improving efficiency, handling edge cases, and reducing redundancy. . . . .	44
6.2	<b>LLM response from <i>Field Exercise 1</i></b> . The model identifies a matrix logic error and explains the rationale behind the fix. . . . .	45

6.3	<b>LLM response from <i>Field Exercise 1</i>.</b> The model identifies a lack of context in the query and requests clarification before offering general guidance on websocket usage. . . . .	47
6.4	<b>LLM response from <i>Field Exercise 1</i>.</b> The model provides a step-by-step guide to implementing the Levenshtein distance algorithm using dynamic programming. . . . .	48
6.5	<b>LLM response from <i>Passwords 1</i>.</b> The model maintains assignment integrity while breaking down the steps to generate efficient mnemonic passwords. . . . .	49
6.6	<b>LLM response from <i>Passwords 1</i>.</b> The model analyzes specific lines of student code and offers targeted suggestions to improve clarity, performance, and readability without completely rewriting it. . . . .	50
6.7	<b>LLM-Generated Student Session Summary from <i>Field Exercise 1</i>.</b> The LLM groups related queries by topic and describes the student’s challenges and errors across tasks. . . . .	53
6.8	<b>LLM-Generated Low Effort Student Session Summary from <i>Passwords 1</i>.</b> The LLM highlights the student’s misuse of a Python function. . . . .	54
6.9	<b>LLM-Generated Low Effort Student Session Summary from both <i>Field Exercise 1</i> and <i>Passwords 1</i>.</b> The LLM acknowledges the student’s lack of detail in queries. . . . .	55

## **LIST OF ACRONYMS**

**AI** artificial intelligence

**LLM** large language model

**LMS** learning management systems

**TA** teaching assistant

## SUMMARY

This thesis explores the integration of artificial intelligence (AI) teaching assistants within advanced programming courses to enhance individualized learning and instructor support. The increasing reliance on large language models in programming education has sparked interest in their potential to improve student outcomes. However, challenges remain in effectively implementing AI in classroom environments with more advanced learning outcomes. This thesis presents a demonstration of a large language model (LLM)'s ability to enhance a student's individual learning experience while providing instructors with enhanced management tools. Key contributions include the development of a dynamic AI teaching assistant integrated into an intro to cybersecurity course framework. The system offers context-aware AI support, code suggestions with guardrails, and summarization of student-AI interactions. Through data analysis, this thesis verifies the effect of course context. This research provides practical insights for educators considering the integration of AI in their classrooms. It proposes future directions for optimizing AI teaching assistants to meet the unique needs of advanced programming courses.

# CHAPTER 1

## INTRODUCTION

In the past few years, the exponential rise of AI has led to an increased interest in integrating LLMs into academia. At the time of this thesis, researching the use of LLMs in programming education has become widespread. Existing literature has explored how programming students are now relying on LLMs both inside and outside the classroom to write and debug their code [1]. However, students using LLMs outside the classroom come with several caveats. Wang et al. observed that LLMs lack the inherent ability to establish conceptual relationships in the same manner that humans do, unless they are explicitly provided with the contextual information relevant to the query [2]. Similarly, Denny et al. found that using poor prompting techniques resulted in missing more edge cases when solving programming problems [3]. These characteristics commonly contribute to technical errors and incorrect outputs. This poses a problem for students. When they receive plausible yet inaccurate answers or code samples from an LLM without recognizing the mistake, they risk being guided down incorrect learning paths [2] [4] [5]. Such risks underscore the need for controlled LLM use in educational settings, where risks to students can be mitigated and pedagogical strategies can be adapted to ensure effective learning.

Within the context of programming classrooms, controlled use of LLMs has shown promising educational benefits. Thuy et al. found that students using coding AI assistants liked being nudged in the right direction when starting an assignment [6]. Students also appreciated the immediate support with basic syntax and semantic issues. It allowed them to resolve problems within seconds rather than spending hours searching for solutions online [6]. The twenty-four-hour availability of LLMs enhanced accessibility to academic support, particularly for students who were unable to attend traditional office hours [7]. When integrated into programming courses under the guidance of instructors, LLMs can serve as

tools that support learning while reducing cognitive load and frustration.

While students may benefit from structured LLM support, instructors face a new set of challenges when introducing this new tool into the classroom. With accessible AI assistance, many established teaching approaches must change to challenge students [8]. One struggle is in balancing the advantages of AI assistance with the need to foster independent problem-solving and technical mastery. A common concern among instructors is student over-reliance on LLMs when it comes to assessments and assignments [4] [9]. Others are concerned with academic dishonesty and the decrease of instructor and student interactions [9]. As a result, some educators are hesitant to adopt these tools without clear frameworks or institutional support.

Despite growing interest and experimentation with LLMs in educational settings, current research has primarily focused on using LLMs to help introductory programming students arrive at working solutions. Advanced courses often involve complex, open-ended tasks and a wide range of student skill levels, making individualized learning support especially crucial. They tend to emphasize not only correctness but also code quality, efficiency, and adherence to best practices. In these contexts, meaningful learning extends beyond writing functional code to refined and efficient code. LLMs offer the potential to meet students where they are by providing tailored assistance, adaptive feedback, and autonomy in navigating programming challenges. However, the more individualized the AI support becomes, the more difficult it is for instructors to monitor usage, ensure consistency, and provide meaningful oversight. Without proper tools and integration strategies, instructors may struggle to balance the benefits of LLM-driven personalization with the practical need to manage classroom dynamics and uphold pedagogical goals. Prior research has highlighted both the promise and pitfalls of using LLMs, but more can be done to understand how to reap the benefits of personalized classroom LLM usage while providing seamless integration and control for instructors. This gap motivates the present study. We explore how LLMs can be incorporated into advanced programming courses in a way that supports

individualized student learning while meeting instructor needs. The following research questions guide our work.

- 1. How can AI teaching assistants be integrated into non-standardized, advanced coding classroom settings?*
- 2. How can LLM-generated suggestions be adapted to meaningfully align with course-specific content and learning outcomes?*
- 3. How effective are LLMs in supporting common advanced programming goals, such as writing efficient code?*
- 4. How can LLMs help instructors better understand their students' interactions with AI teaching assistants?*

To answer the above questions, this thesis explores the integration of an existing AI teaching assistant into a web-based learning platform to better understand the adaptability of AI teaching assistants in an advanced course with specific needs. In doing so, we show how AI teaching assistants can increase the individualized experience for students, instructors, and the class as a whole. The contributions are listed below.

- **Seamless System Integration for AI Teaching Assistants:** Implemented a dynamic connection between an introductory cybersecurity course platform and an AI teaching assistant with guardrails, enabling students to access help directly within an established learning environment.
- **Context-Aware AI Support:** Enhanced the AI's ability to provide tailored, context-sensitive responses by incorporating course-specific material.
- **Student Query Summarization:** Developed features to summarize student queries and interactions, helping instructors monitor student progress and better prepare for one-on-one meetings.

- **Strategic Code Support:** Provided instructors with control over the AI’s response style by enabling adjustments to the level of detail in code suggestions.
- **Future Recommendations:** Proposed steps for scaling the system further and suggested next steps for deploying the system in a live classroom setting and evaluating its impact on student learning and instructor workload.

These contributions collectively lay the groundwork for the future integration of AI teaching assistants in advanced programming courses, where efficiency and individualized learning goals are paramount. This thesis combines system integration, context-aware support, and instructor management tools to demonstrate how AI can improve student learning outcomes and instructor effectiveness, while promoting a balanced approach to classroom use. The findings aim to inform future pedagogical strategies and provide actionable insights for educators who want to integrate AI tools into their teaching practices.

The structure of the remaining work is as follows. In Chapter 2, a literature review is presented consisting of various forms of feedback found in programming classes, along with other works related to this study. Chapter 3 outlines our design goals. Chapter 4 provides a description of our implementation, and Chapter 5 presents our evaluation. Chapter 6 presents preliminary results, while Chapter 7 summarizes our findings. Finally, Chapter 8 concludes our work with recommendations for future work.

## **CHAPTER 2**

### **RELATED WORK**

This chapter reviews existing work on various forms of feedback in programming education, the use of LLMs in the classroom, and prompt engineering techniques for improving AI responses. Traditional office hour systems and early digital tools offer limited scalability, while recent research on AI-powered feedback systems highlights both promise and challenges. Prompt engineering has emerged as a powerful strategy for enhancing AI support, but students often lack the skills to apply these techniques effectively. Together, these areas reveal a gap in research and practice around AI teaching assistant (TA)s that are personalized, instructor-guided, and usable in complex, real-world classroom environments.

#### **2.1 Traditional Feedback in Programming Education**

Programming classes have traditionally relied on instructor and TA one-on-one meetings to provide individualized feedback to students. In earlier models, this support often came from former students of the course who offered help outside of class time. As class sizes have grown, many institutions have adopted a basic whiteboard queue system to manage demand. While this approach works reasonably well for small to medium-sized classes, it becomes unsustainable in larger courses due to long wait times and repetitive questions. Studies report wait times of over an hour, driven by vague student requests and frequent repetition of the same conceptual issues [10] [11].

More recent systems have implemented digital queue management tools to improve efficiency. For example, My Digital Hand [11] requires students to fill out a structured request form before entering the queue. This system provides TAs with early insight into student problems and allows for logging interactions. However, Smith et al. reported that students often provided minimal or ambiguous information, making it difficult for TAs

to diagnose the issue in a short interaction window. TAs also report that students expect walkthroughs of full assignments, rather than support for specific bugs or concepts [10] [11]. These challenges underscore the limitations of traditional office hour systems and highlight the need for scalable, responsive tools that support individualized student aid.

## **2.2 Large Language Models in Programming Education**

To adapt to the increasing presence of AI while mitigating the disadvantages of traditional feedback systems, recent research has explored how LLMs can reshape the classroom learning experience. This section summarizes prior work on how students interact with LLMs, how instructors perceive the use of LLMs in the classroom, and how existing AI teaching assistants have been designed to operate in controlled educational settings.

### 2.2.1 Student Experiences

Studies, such as those by Kazemitabaar et al. [12], have examined how access to AI code generators affects first-time programmers. The study involved 69 students, aged 10 to 17, to complete a series of Python programming tasks with the help of OpenAI Codex, the AI model behind GitHub Copilot. It found that those who had access to Codex were able to complete more assignments with fewer errors in less time. As topics grew more complex, these students increasingly relied on the AI tool by submitting unmodified code from the generator in 49% of cases. While many participants reported feeling empowered and motivated by the tool, some expressed dissatisfaction with the lack of explanation behind the generated code. These students preferred step-by-step guidance over complete solutions, highlighting the tension between increased productivity and conceptual understanding.

In another study, Sheese et al. [13] further observed that students often engaged with AI in minimal-effort ways, such as copy and pasting vague prompts. This suggests a need for AI teaching assistants that guide intentional help-seeking behavior while discouraging reliance on solutions generated by LLMs. These studies suggest that while students

appreciate the accessibility and responsiveness of AI teaching assistants, many lack the prompting skills or awareness to use them effectively in learning contexts.

These issues align with broader findings in existing literature. Students often submit unmodified code from LLMs and are left without a clear understanding of why the solution works [12] [13]. Additionally, these AI-assisted tools often assume students know how to engage productively with the AI. In reality, many students struggle to come up with useful prompts or ask the right follow-up questions. This is especially true if they have limited understanding of a task [7] [12].

### 2.2.2 Instructor Insights and Concerns

Across studies focusing on instructor insights on LLM use in the classroom, instructors expressed cautious optimism about integrating AI tools into programming education. In a survey of 12 instructors with moderate to high familiarity with ChatGPT, Husain et al. found that instructors were using LLMs to help design lesson plans and automate feedback on student code [9]. These instructors called for the strategic implementation of AI with institutional support and training.

Another study conducted interviews with 12 instructors and reported similar interest in AI tools, particularly for personalized feedback in both traditional coding assignments and project-based learning [8]. Despite this, participants voiced concerns that AI integration would require curriculum adjustments, especially in introductory courses where the teaching emphasis is shifting from code writing to code comprehension and debugging.

Overall, instructors consistently expressed concerns about over-reliance, academic integrity, and a decrease in the need for foundational coding skills [8] [9] [14]. They worried that students might rely too heavily on AI-generated solutions, especially in introductory courses where foundational skills must be developed. As a result, many instructors advocated for revised assessment strategies, such as oral exams and in-person debugging tasks, and delayed access to AI tools until students had developed sufficient coding skills. Im-

portantly, instructors were not opposed to AI tools themselves. They called for AI tools designed with education in mind. They also emphasized having customized control over what the LLM would output to the students [14]. Collectively, these studies reveal that instructors are willing to incorporate AI into the classroom, but only when the tools align with learning goals, uphold academic integrity, and support the development of reflective skills.

### 2.2.3 AI Assistants in Controlled Environments

AI teaching assistants deployed in controlled classroom environments offer instructors and researchers an opportunity to monitor how students interact with LLMs in real-time. Codehelp [7] and CodeAid [14] showcase two classroom-focused tools that try to offer more controlled and educationally appropriate support. Both use guardrails to support student learning without revealing complete solutions. This design encourages students to reflect on their errors, promotes active problem-solving, and gives instructors insight into common student struggles. Codehelp has the student provide their code, error, and question in a form to query the LLM. This structured format encourages students to ask targeted questions. Student surveys indicated that learners valued Codehelp as an initial resource, especially when instructors or TAs were unavailable. However, students also noted that it could not fully replace the benefits of in-person student-TA interactions [7]. CodeAid took a slightly different approach by incorporating structured feedback templates, which provide a variety of feedback styles. In addition to asking general questions through a single form, students can also ask the AI teaching assistant to explain, fix, or write code [14]. Students who used CodeAid favored using it for general concepts and understanding code, but a large percentage still used ChatGPT or in-class resources throughout the semester.

Instructors felt Codehelp was easy to implement and valued the “direct and unfiltered look” into their students’ queries. This visibility allowed instructors to adapt their lectures in response to patterns in student queries [7]. CodeAid was well-received by instructors

as it was deemed as “a safer alternative” unmoderated tools like ChatGPT [14]. Still, both students and instructors noted the LLM’s tendency to answer questions with coding suggestions outside the scope of the class. While Liffiton et al. [7] added an “avoid set” to suppress undesired responses, this solution was limited. More dynamic and granular tailoring is needed to align feedback with assignment goals and student ability levels. While LLMs offer on-demand support and promising classroom integration models, existing tools often lack course-specific adaptations and provide limited control for instructors, especially in advanced settings where multiple teaching styles are used.

## **2.3 Prompt Engineering**

A user’s prompt plays a critical role in how an LLM responds. Since LLMs do not understand context the way humans do, a well-structured prompt can guide the model toward more accurate and useful answers. In educational settings, especially in programming, prompting can determine whether the model gives a surface-level answer or walks the student through the reasoning behind it. This section examines common prompting strategies and how they compare to how students actually use LLMs in practice.

### 2.3.1 Common Prompting Techniques

Chain of Thought (CoT) prompting is a common method used to guide the model’s reasoning step by step. It was first used in math problems, but Wei et al. [15] also saw improved accuracy in general conversation prompts. By employing Automatic Chain of Thought and prompting “Let’s think step by step,” [16] the LLM is encouraged to walk through a problem as a human would [17].

More advanced prompting techniques have since been introduced. Least-to-Most Prompting aids in breaking down problems that have more complex solutions [18]. While CoT is limited by the prompter’s explanation of how to answer the question, Least-to-Most first queries the LLM to decompose the problem into sub-questions. Then the LLM is queried

to answer each sub-question. As the prompter works their way through each sub-question, the next prompt is appended to the previous query to provide the LLM with a chain of thought. Zhou et al. reported that this prompting technique outperformed COT even on longer prompts. Thread of Thought Prompting also showed increased accuracy with longer prompts [17]. Sahoo et al. focused on enhancing LLM reasoning in prompts with “chaotic context”. These are longer prompts that provide a large amount of information from various sources. Depending on the query, some sources are more relevant to the solution than others. Thread of Thought asks the LLM to “walk me through this context in manageable parts step by step, summarizing and analyzing as we go” [17] to help the LLM piece together the details of the prompt that aid in finding the correct solution. This technique saw a significant improvement against COT in question-and-answer prompts.

Some strategies focus less on how the model thinks through the problem and more on how to prevent it from giving incorrect answers. Creswell et al. introduced Faithful Reasoning, which trains the model to admit when it does not know the answer to a prompt in an attempt to reduce hallucinations [19]. This study created a forward-chaining model, denoted as Selection Inference. This introduces two new steps in the LLM fine-tuning: halter and value function. Halter queries the LLM to set a limit on the number of reasoning steps taken to find the solution to a query. This step also re-formats the LLM’s response into a standardized format. Then, the value function checks the quality of the LLM’s reasoning behind the given solution.

### 2.3.2 Real World Applications of Prompt Engineering

While prompt engineering strategies have proven to be a useful tool, they are rarely used by students. Denny et al. [3] showed that developers often have better results with LLMs when they break problems down into smaller tasks. For students, there is a clear difference in strategy. Students tend toward asking for immediate debugging or implementation assistance without taking the time to understand concepts [3] [5] [13]. Thus, many students

struggle to break down the problem and instead attempt to have the LLM generate an entire solution. This suggests that there is a growing need to teach students the necessary skills to formulate effective prompts as well as have a “critical eye” when reviewing AI-generated code [5]. This further supports the need for classroom tools that scaffold students’ interactions with AI and help instructors guide the quality of prompts and learning outcomes.

In summary, the literature shows a strong interest in using LLMs to enhance programming education, but also highlights critical gaps around meaningful explanations, individualized support, instructor control, and classroom alignment. Existing systems either focus on beginner use cases, limit instructor customization options, or assume that students have the skills to use AI effectively. This thesis builds on this work by exploring and developing a flexible AI teaching assistant that addresses these concerns within the context of an advanced programming classroom.

,

## **CHAPTER 3**

### **METHODOLOGY**

This chapter outlines the methodology used to investigate the integration of AI teaching assistants into an advanced programming classroom setting. Guided by the research questions presented in Chapter 1, this work focuses on developing a prototype system that connects an AI teaching assistant to a classroom learning platform. In doing so, we aim to provide individualized student support while giving instructors classroom management capabilities. The methodology includes system design features that support context-aware responses, the option for responses with code, and student session summarization for instructors to monitor and customize AI behavior. This chapter also describes the data collection and analysis process used to evaluate the AI teaching assistant’s effectiveness in providing meaningful support without offering full solutions.

Unlike prior research, which primarily focused on creating AI teaching assistants for novice programmers [1] [20] [21], this thesis aims to provide personalized feedback for students in advanced programming courses. This calls for a set of design goals that match the needs of these types of courses. In advanced classes, the questions submitted by students will be more complex and focus on subjects specific to the course. In this case, the focus will be on writing efficient code to store, organize, and access data. We also want to build on insights from instructors in previous studies, which call for more customization options for AI-powered assistants. To achieve this, the presented solution was designed to meet two key design goals: dynamic and seamless, which are explained further below.

### 3.1 Design Principles

#### *Dynamic*

The AI teaching assistant must be able to tailor its feedback based on the needs of the student, instructor, and class as a whole. At the student level, personalization is straightforward because students interact directly with the assistant through queries. Becker et al. have shown that stand-alone LLM can reply to students' queries on code improvement and computer science topics with high accuracy [1]. Tailoring to the instructor has been less explored in previous works on AI teaching assistants. While Liffiton et al. [7] introduced an *avoid set* to restrict certain types of AI-generated feedback, existing systems generally lack mechanisms for instructors to specify desired needs in AI teaching assistants [14]. For example, a cybersecurity instructor may want the AI assistant to highlight potential security vulnerabilities in student code. A machine learning instructor may wish to have the AI assistant emphasize the use of specific algorithms taught in course lectures. Our implementation allows instructors to exert this kind of influence over the AI's feedback generation, offering a foundation for course-aligned customization.

It is also of interest for the AI teaching assistant to be able to observe the students' queries in a given classroom and summarize any trends that may be present. This information can be valuable to both students and instructors. Understanding what common questions are being asked can help create a learning space tailored to the needs of the students within it. Unlike rigid, rule-based tutoring systems, meeting this goal ensures that students receive individualized and evolving support that mirrors human-like instruction. For our system to be *dynamic*, it must have the ability to incorporate assignment context and course materials, be responsive to varying levels of student input quality, and be flexible in tailoring output based on instructor preferences.

## ***Streamlined***

The overall solution should be easy to integrate, use, and adjust settings based on classroom needs as noted in Kazemitabaar et al.'s design considerations for AI-powered assistants [14]. The system setup should be minimal, and the user interface should be intuitive for both students and instructors. The system should allow professors to oversee and understand student interactions with minimal effort. A student should be able to reference their past interactions with ease. For our system to be streamlined, it must have an easy setup and minimal configuration for new classes, a straightforward user interface for instructors and student query summaries, and support the ability to review student engagement patterns for feedback sessions quickly. These features aim to reduce the cognitive and logistical load typically associated with integrating new technology into the classroom.

## **3.2 System Development**

### 3.2.1 LLM Selection

The selection of an existing AI teaching assistant was a critical design choice, as the assistant needed to provide feedback in a controlled way so that direct answers were not revealed to the student. Various AI assistants, including Promptly [5], Codehelp [7], and CodeAid [14], were considered based on their ability to provide code explanations and interact with students. Codehelp was chosen for its strong use of guardrails and ease of integration. Currently, Codehelp utilizes OpenAI's GPT-4 via API as its supporting LLM. Alternative AI assistants were not chosen because their use cases focused on providing starter code for novice students [12] or they had already become obsolete due to OpenAI's discontinuation of Codex. We felt that Codehelp was a great starting point. However, there are some limitations to its design. One is that the instructor can only add an *avoid set* to tell the LLM which topics to exclude from the AI assistant's responses. To address this limitation, we extended Codehelp's functionality by converting the *avoid set* into a context parameter as

shown in subsection 4.2.1.

Additionally, the guardrail to prevent the LLM from responding with code was too rigid. In some cases, it can be beneficial to allow code responses when the assignment considers the code's efficiency or clarity. As a result, an option was created for the instructor to allow the AI teaching assistant to respond with code. This is explained in greater detail in subsection 4.2.2. Lastly, Codehelp's instructor page was limited to displaying student interactions in a table format. Instructors would have to comb through data to gather any insights about their students' interactions with Codehelp. This was mitigated by aggregating the student interactions into summary dashboards, reducing the instructor's workload as explained in subsection 4.2.3. The decision to use Codehelp gives a solid base to ensure the assistant delivers high-quality feedback while minimizing the need for complex fine-tuning. This approach aligns with the goal of creating a streamlined assistant that can be easily integrated into classroom workflows.

### ***Codehelp***

Codehelp [7], developed by Liffiton et al., is a web-based automated AI teaching assistant used to provide on-demand assistance to programming students. Its key feature is the use of guardrails to prevent the sharing of direct solutions and encourage students to better understand class concepts. An instructor can quickly set up a class by heading to the Codehelp landing page [22]. There, they can designate what LLM will be used for student queries, add any topics or libraries via the *avoid set* to exclude in responses, and make the class joinable through a unique link. Once students are actively in the class, they can ask the LLM on a page similar to ChatGPT's landing page [23]. Past student queries populate on the right side of the webpage, and the usual search bar is accompanied by two additional bars for the student's code and the accompanying error message.

Codehelp's original query backend workflow is pictured in Figure 3.1. When a student submits a question, the LLM's response is sent through a sufficiency check as shown in

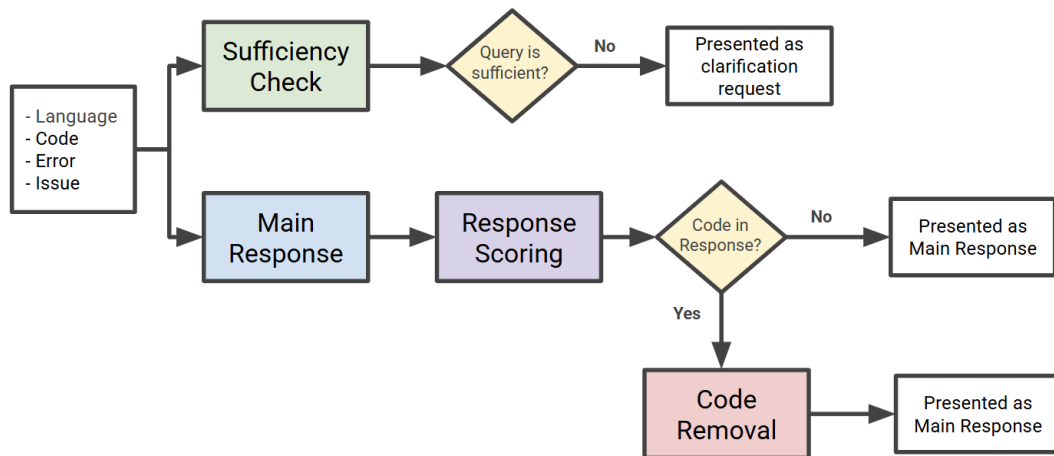


Figure 3.1: Codehelp’s original response workflow [22]

the green box in Figure 3.1. This check asks the LLM if the information provided by the student is enough for the LLM to answer the student’s question. If more information is needed, the LLM’s response is rephrased to ask the student for more information and to explicitly state that its response is based on the limited information provided. An example of this check is shown in Figure 6.3. Codehelp’s sufficiency combats a LLM’s tendency to hallucinate when it does not know the answer to a question [19]. This also works well against students who tend to submit low-effort prompts [12]. This is best described as prompts where students write bare minimum prompts in an attempt to get the full solution. This could be as little as asking the LLM to “Fix this code” or as far as copying and pasting the problem statement from assignment materials verbatim [5] [13]. It compels students to craft their prompts carefully. Codehelp’s sufficiency check makes it clear that providing more information about their issue will yield better results. If the student’s supporting information is sufficient, the LLM’s response is sent through a code check. The secondary check ensures that there is no code in the response. The idea is not to correct the students’ code but to give them an idea of what may be causing the error so that they can fix it on their own. This also works well when a student doesn’t know where to start. Instead of providing starter code, the LLM may suggest just the first step in the problem.

While Codehelp provides a strong foundation for individualized student support through its use of guardrails and prompt shaping, its effectiveness also depends on the learning context in which it is deployed. To evaluate how this tool performs in a classroom with complex, multi-step assignments, we integrate it into Riposte, a web-based cybersecurity platform that features gamified and exploratory learning. This allows us to test not only the quality of the AI teaching assistant’s support but also the broader fit of AI assistance in advanced, topic-specific programming environments.

### 3.2.2 Classroom Framework Selection

To ensure a streamlined experience for students, the AI assistant was selected so that it can be embedded within various classroom frameworks rather than a separate application. To make this possible we must test the AI assistant on different types of learning management systems (LMS). Codehelp already can be integrated into popular LMS such as Canvas. To evaluate how AI assistants perform in advanced, non-standard classroom environments, we embedded Codehelp into a unique learning system, as more advanced classes tend to stray from standardized LMS. For our study, Riposte [24], a web-based cybersecurity learning platform, was selected due to its multifaceted framework. It incorporates gamified assignments alongside traditional coding assignments via Jupyter Notebooks. The introductory cybersecurity course that utilizes Riposte covers various cybersecurity topics, including access control, cryptography, cryptographic protocols, network security, and online privacy [25]. Prerequisites for this course include a foundation in programming, data structures, and discrete math. These topics are not typically covered in beginner-level programming classes, and therefore, there are no existing studies that share AI-assisted insights for such a course. This serves as a great playground to test LLM support with various types of assignments that require more than just simple syntax correction.



Figure 3.2: Riposte’s Mission Page [24]. **A** shows the signed-in student’s username. **B** shows the guide window. **C** shows the mission page. **D** shows our contribution of help button which allows students to directly access Codehelp. **E** shows trophies that can be achieved by completing certain milestones. **F** shows the due date for the assignment and the student’s grade.

## Riposte

Riposte [24], designed by Malone et al., is a modular web-based learning management platform that creates a gamified learning environment for students. Its focus is on client/server reverse engineering, and its assignments vary from password cracking to cipher block chaining. Some assignments are traditional and require coding in their assigned Jupyter Notebook, while others are interactive and require using web developer tools. Students log in with their specified credentials and access the missions page to see their current assignment as shown in Figure 3.2. There, they are welcomed by the guide, which consistently gives an initial tip for each assignment to help the student get started in the right direction. As students complete assignments, more features of the platform are unlocked. An example of this is the game module, where they play a variety of challenges and see how they measure up against their classmates on the leaderboard page [24].

Malone et al. saw great success with this style of learning but ran into some limita-

tions that could be alleviated with the help of AI. One concern was that a large number of students were not aware of the resource constraints in the Jupyter Notebooks. Inefficient submissions caused their notebook to crash. Another pain point was when students would fall down a rabbit hole while trying to find a solution using the wrong technique. While a student in this situation could be corrected at an office hours session, it was highly dependent on them knowing they needed help. Often, this doesn't happen until after hours of debugging, leaving the student frustrated. An AI teaching assistant can identify these issues early for a student, saving them time and frustration.

### **3.3 System Evaluation**

Direct user testing was not feasible within the research timeframe, as a course at Georgia Tech that uses the framework was not yet available. Subsequently, this evaluation focuses on assessing the AI teaching assistant's ability to provide meaningful, context-aware support through log analysis and a manual review of student scenarios. Specifically, this evaluation aims to determine whether the system can provide feedback that aligns with the goals of advanced programming courses, such as improving efficiency, readability, and error resolution, without revealing full solutions. We conducted a manual review of AI-generated responses to assess their correctness, conceptual depth, and instructional quality.

#### 3.3.1 Data Collection

We conducted a manual review of AI-generated responses from two assignments on the Riposte platform. These assignments reflect the complexity and specificity of advanced programming content. Simulated student submissions were crafted based on common help-seeking patterns observed in real classrooms [7] [14]. Each scenario was submitted to Codehelp, and the AI assistant's response was reviewed for its clarity, accuracy, and instructional value. This allowed for a focused evaluation of how well the AI assistant performs when embedded in a real course context.

Additionally, to evaluate the AI assistant’s feedback performance at scale, this study uses the open-source CodeChef Competitive Programming Dataset [26], which is available on Kaggle and licensed under the CC0 1.0 Universal (Public Domain Dedication) license [27]. Many competitive programming platforms, such as LeetCode, HackerRank, and CodeChef, provide a rich set of coding problems that simulate real-world classroom assignments. These platforms often release large datasets of user-submitted solutions, many of which are accompanied by grading criteria or outcome labels. CodeChef was chosen due to its well-organized format, diverse range of problem types, and accessibility. These submissions serve as a proxy for student solutions, allowing us to simulate realistic interactions between students and the AI assistant. Each submission is treated as a prompt to the AI assistant, who is then asked to identify areas for improvement and explain its suggestions. The breakdown of this dataset is outlined in subsection 5.3.2.

### 3.4 Final Remarks

Table 3.1: Summary of Modifications and Enhancements to Codehelp and Riposte

Component	Original Capability	Modification / Enhancement
Instructor Context	Instructors could specify an <i>avoid set</i> to block specific topics.	Added a <b>custom context field</b> allowing instructors to include course-specific guidance.
Code Responses	Hard-coded rule to suppress code in all responses.	Introduced <b>instructor-controlled toggle</b> to allow code output when pedagogically appropriate.
Student Query Summary	Codehelp displays raw student query data in a table on the instructor page with no summarization.	Built an LLM-powered <b>summary dashboards</b> that highlight trends, common struggles, and class-wide insights.
Learning Platform	Codehelp designed for integration with Canvas and other standard LMS.	<b>Integrated into Riposte</b> , a gamified cybersecurity platform with multi-modal assignments.

In summary, this methodology outlines a design-driven approach to developing a dy-

dynamic and streamlined AI teaching assistant tailored for advanced programming classrooms. By integrating the assistant into an existing classroom framework, this work emphasizes both ease of adoption and instructor usability, while preserving the flexibility needed to support individualized student learning. The system was implemented with targeted modifications to enhance adaptability, contextual responsiveness, and instructional oversight. These modifications are summarized in Table 3.1. The evaluation strategy includes a large-scale feedback analysis and contextual testing using Riposte-based assignments. This was designed to assess the assistant's effectiveness in providing meaningful, pedagogically aligned support without offering complete solutions.

## **CHAPTER 4**

### **IMPLEMENTATION**

This chapter details how we implement the methodology explained in the previous chapter. The implementation focuses on realizing the two core design goals by creating a dynamic assistant that can adapt to individual student needs and streamline the experience for instructors. The following sections describe the system architecture, key components, and the integration process.

#### **4.1 System Architecture**

The system design combines two web platforms: the classroom environment and the AI assistant interface. This ensures that the AI assistant operates independently while still accessing relevant student data when authorized. Communication between the platforms is facilitated through URL parameters and backend API calls, allowing for context-aware assistance without modifying the core classroom platform.

#### **4.2 Integrating Codehelp into Riposte**

To standardize the use of LLM teaching assistants in diverse programming classes, we need to ensure that the framework can assist students with fine granularity without requiring an intensive setup process. The integration into the class should be seamless, so it doesn't require additional training from the instructor. The first step was to integrate Codehelp as is into Riposte's framework. From the students' perspective, Riposte's built-in guide provides a description to help the students get started on their current assignments. The adjustable screen width and accessibility when viewing both the mission page and the Jupyter Notebook made it an ideal place for students to access Codehelp. A help button was added to the

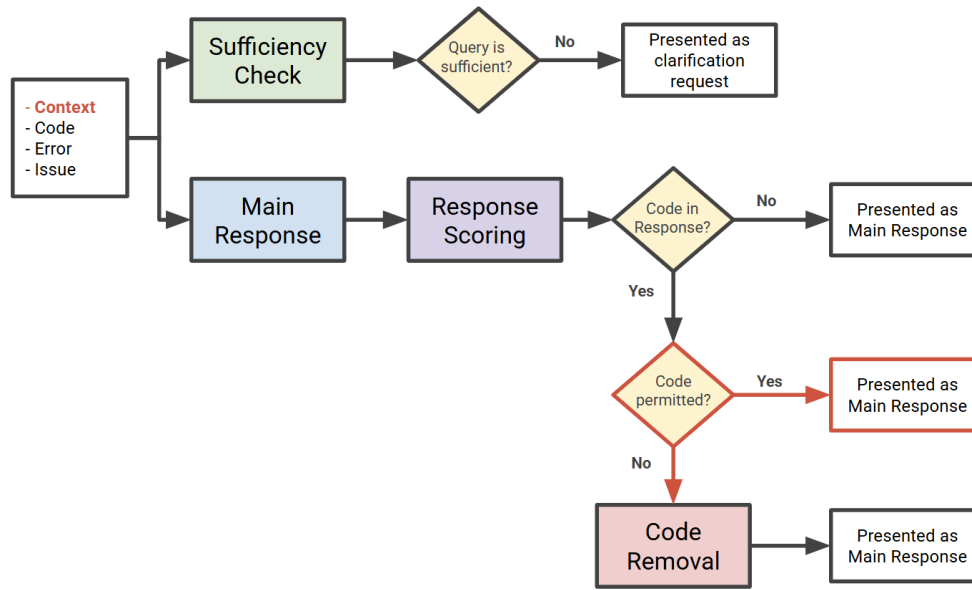


Figure 4.1: Codehelp’s improved response workflow. Our additions are shown in red.

bottom right corner of the guide’s HTML frontend as shown in Figure 3.2. Once clicked, a pop-up window matching the guide’s frame size will appear, redirecting the student to Codehelp’s help page. On the backend, Codehelp utilizes the exact user identification as Riposte. This means a student can bypass creating an account through CodeHelp and logging in repeatedly. From here, a student can seamlessly ask questions to Codehelp without leaving the current webpage.

From here, we address three of Codehelp’s limitations by implementing a context feature in Codehelp’s LLM response workflow, allowing instructors to add class and/or assignment details and a Student Session Analysis on Codehelp’s instructor page. The following subsections describe how assignment context, student session analysis, and response flexibility are implemented into Codehelp and how they were utilized on Riposte. Figure 4.1 shows how these additional features fit into Codehelp’s response workflow. Changes are shown in red.

The screenshot shows the Codehelp Instructor interface for a class named 'Riposte\_Test'. At the top, there are navigation links for 'CodeHelp', 'Get Help', 'masterslacker Riposte\_Test', 'Configure Class', 'Instructor View', 'Admin', and 'Logout'.

**Contexts Table (A):** A table listing assignment contexts with columns for 'name'. The entries are: Field Exercise 1, Getting Started 1, No Context, Passwords\_1, and Passwords\_2.

**Users Table:** A table showing user activity with columns: user, #queries, 1wk, active?, and instructor?. The users listed are bluehoudini, greenlizard, masterslacker, suntzu, and whiteghost.

**Queries Table (B):** A table showing student queries with columns: user, time, context, code, error, issue, response (len), and helpful. The 'context' column is highlighted with a red box. The entries show queries from 'suntzu' on '2025-04-03 07:38AM' and '2025-04-03 07:32AM' related to 'Passwords\_1'.

**Student Session Summary (C):** A summary box containing three bullet points:

- Mnemonic Password Generation:** The student is working on generating mnemonic passwords from a given line of text, focusing on transformations and filtering inputs. (Query IDs: 95, 91, 75, 74, 73, 72, 71, 70, 69, 68, 67)
- Recursive Function for Password Transformation:** The student is using a recursive function to apply character transformations to passwords, ensuring unique mnemonic passwords are generated. (Query IDs: 95, 81, 80, 79)
- String Replacement and Transformation Logic:** The student is dealing with string replacement logic, specifically ensuring only the first occurrence of a character is replaced and handling case transformations. (Query IDs: 95, 93, 92)
- Password Hashing:** The student encountered a TypeError related to encoding passwords before hashing with SHA-1. (Query ID: 94)

Figure 4.2: Codehelp’s improved Instructor page workflow. **A** shows the Assignment contexts table. **B** Shows the corresponding addition to the Queries table. **C** shows the Student Session Analysis.

#### 4.2.1 Class Context

Codehelp’s original *avoid set* is extended to operate at the assignment level, allowing instructors to incorporate specific course materials into the LLM’s context and refine responses based on active tasks. This was incorporated by utilizing Codehelp’s SQL database. This database stores all the course data, user data, and their interactions with the LLM. Each time a student submits a query, the details (code, error, and question) and LLM Responses are saved in the query table that is displayed on the instructor’s page as shown in Figure 4.2. On the backend, we added a context table to store an instructor’s assignment contexts, which the instructor must add before the course starts. This table is joined with the Queries table and is used as an additional filter for the instructor. When the student queries the LLM, the corresponding context is passed to the LLM along with the student’s question. An example context may look like this:

```
context_set = ''''''
```

The assignments are written in a Jupyter Notebook with limited memory. The goal is to accomplish the task efficiently as the test cases their code will be graded against will be large in number.

Examples of Primitive Transforms are :

Capitalize the first letter p

(For example: `\password" →\Password" )`

Replace the first met letter s with \$ (`\password" →\pa$sword"`)

For example sentence, `\I went to London four and a half years ago"`.

You could use the first letter of every word, replace went with w, four with 4, and with &. Combine them into a password: `iwtl4&ahya`.

```
''''''
```

*context\_set* would be passed into the main response using the following conditional:

```
...
```

```
if context_set is not None:
```

```
    class_context = f''You can reference the following to aid in  
    your response: {context_set}''
```

```
else:
```

```
    class_context = '' ''
```

```
...
```

On the front end, this added Context column is shown in Figure 4.2. There, the instructor can click their desired assignment and see all the queries for that context. For the student, this context feature is only visible at the top of the query form. This is because we utilize Riposte's dynamic nature to populate the active assignment automatically. Riposte, like other LMSs (i.e., Canvas), may look different depending on what assignments are visible to the student. Riposte's assignments are displayed on the mission page when

a student logs in with their credentials. The guide, Jupyter Notebook, and other features update depending on which mission is selected, as shown in Figure 3.2. It would make sense to have the context provided to the LLM follow this pattern of correlating with the student's active assignment and prevent any confusion on which context to use. Contexts were adapted into Riposte by updating the redirect link from Codehelp's general help page to the context-specific help page that matches the student's active assignment. This works well given that the active mission has a matching context label in the Codehelp class and the student is enrolled in the Codehelp class. If either condition is not met, Codehelp sends an error to the student.

Improving Liffiton's use of avoid sets is essential for integration into classroom frameworks where assignments may share similar learning objectives but differ in approach or constraints. Students must receive feedback tailored to their specific, active assignment. Without this contextual awareness, an LLM might provide guidance that is appropriate for a different assignment and potentially hint at incorrect methods or confuse the student. Enhancing the assistant's ability to distinguish between assignments ensures that feedback remains relevant, accurate, and aligned with the instructional intent.

#### 4.2.2 Response Flexibility

Next, we worked on increasing Codehelp's adaptability by enabling response flexibility. While the code check feature is well-suited for novices, advanced courses may benefit from more detailed feedback, such as line-by-line analysis or efficiency suggestions. Here, we refine the code check to support those scenarios, giving instructors control over when and how the LLM generates code.

To implement this, we added an option on the instructor page to allow code responses for a specific assignment context. This is shown in Figure 4.3. We also adjusted Codehelp's main template to allow code when the assignment has this feature enabled. We emphasize correcting student code one line at a time, along with in-depth explanations, to prevent

responses with full-coded solutions.

Here, we modify Liffiton's original system prompt, which serves as the main template to allow for code.

```
{'role': 'system', 'content': `````\
```

```
If the student query is off-topic, respond with an error.
```

```
Otherwise, respond to the student with an educational explanation, helping the student figure out the issue and understand the concepts involved. If the student's query includes an error message, tell the student what it means, giving a detailed explanation to help the student understand the message. Explain concepts, language syntax, semantics, standard library functions, and other topics that the student may not understand.
```

- Be positive and encouraging!
- Only write a corrected or updated version of the student's code one line at a time. You must not rewrite the entire code for the student.
- Use Markdown formatting, including ` for inline code.
- Do not write a heading for the response.
- Do not write any example code blocks.
- If the student wrote in a language other than English, always respond in the student's own language.

```
How would you respond to the student to guide them and explain concepts, improving code efficiency and security when necessary?
```

```
`````\}
```

### 4.2.3 Student Session Analysis

Finally, to enhance instructor support, we extend Codehelp's instructor dashboard. While Codehelp can display and export data, reviewing raw student queries can be cumbersome.

## Create context in class Riposte\_Test

**Name \*** Passwords\_2

**Allow Code Responses**  Allow Code Responses

Check this box if you want to enable this feature for students.

**Environment and tools**

Languages, libraries, and/or frameworks that students are learning or using in this context.

Write one per line.

Python  
Jupyter Notebooks

**Details**

Additional important details for this context. This could include assignment specifications, module learning goals, or other guidance for the language model.

You can use markdown formatting.  
This will *not* be shown to students.

Cracking passwords online requires

In the previous homework you implemented a mnemonic password to the rules of the previous assignment (e.g., their passwords included mnemonics).\*\* In fact, we overheard some of them bragging about composite strings as passwords.

The student's task is to write code that performs an online password

Figure 4.3: Codehelp's create a context in class page [22]. Our addition is the Allow Code Responses field

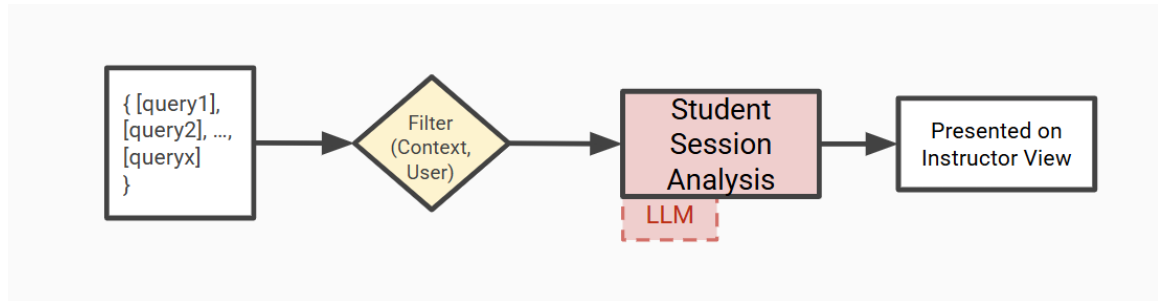


Figure 4.4: Student Session Analysis Response Workflow

The updated system introduces a student session analysis feature that summarizes recent student interactions, allowing instructors to quickly identify struggling students and provide tailored support. This design was inspired by tools like My Digital Hand’s peer instructor interface [11].

The data collected from students’ queries on Codehelp contains valuable information for instructors. Each time a student asks a question, the information submitted and the LLM’s response are saved in Codehelp’s internal SQL database. This query table is displayed on the Instructor tab of Codehelp and is available for download in CSV format. This information can be used to adjust the class lectures to fit the needs of their students. However, in a large class, students may ask several questions per assignment. This can be tedious for the instructor to sort through. If a student seeks help during office hours and wants to reference their interactions with Codehelp, it would be beneficial for the instructor to get an idea of what questions the student has already asked. This saves time and gives the instructor a bird’s eye view of what questions their students are struggling with as the class progresses. In active or flipped classroom models, where in-class time is dedicated to discussion and problem-solving, such insight allows instructors to focus on the concepts that students are collectively struggling with and tailor their instruction to better meet evolving needs.

To implement this, we utilized the existing query SQL table shown on the instructor page. This table already had the ability to filter by the selected user. This feature was

enhanced by adding a context column to the table that could also be filtered. Additionally, a contexts table was added to the pre-existing user table on the instructor page. These two features made it possible to pass the queries filtered by the selected user or selected context to the LLM and ask it to summarize the student's interactions with Codehelp. Specifically, the LLM was asked to list the topics that the student(s) queried about and to note the query IDs that go with each topic.

We create a similar workflow to Codehelp's student query workflow. Figure 4.4 this process. All the queries from the query table would be filtered in the backend by user or context. Then those selected queries would be passed to LLM.

The prompt below is passed as the user prompt and acts as the instructor's query to the LLM.

```
{'role': 'user', 'content':  
    ``Help me, a computer science instructor, understand this student's  
    session on a help forum so that I can quickly understand how to  
    help a student during my office hours.``}
```

*analysis\_template\_sys* is passed as the system prompt and sets the overall behavior for the LLM. The LLM would have the following information to achieve a structured response for each query:

```
{'role': 'system', 'content':  
analysis_template_sys = jinja_env.from_string(``````\n    You are an AI assistant specializing in programming and computer  
    science.  
    Your role is to assist the professor give targeted feedback  
    on topics their students are struggling with their coursework,  
    but you must do so in a way that the professor can understand
```

what topics to explain at a quick glance to keep office hour lines short. Here are your guidelines:

1. Write in HTML format but don't start with `'''HTML '''`.
2. Write only a 1 concise section titled `''Student Session Summary''`
3. In bullets, list the topics the student queried about and note the query IDs that go with each topic.  
`''''''') }`

On the front end, this can quickly produce a concise summary of queries specific to the selected class, user, or assignment. This is illustrated in Figure 4.5 and Figure 4.2. While we focused on using this feature for the instructor, many extensions can come from this as the output is customizable. We feel a natural next step is to make this information available to the students on Codehelp's Homepage via a Frequently Asked Questions Frame. Studies documenting the struggles of office hours note the issues of the same questions being asked. This can become an avoidable cost with Codehelp. We can save tokens and students' time by directing them to review the commonly asked questions summarized by the LLM. This would also change dynamically, meaning the more that students use it, the more effective the tool is. A mock-up of this feature is shown in Figure 4.6 because adding it was not feasible within the research time frame. In summary, this feature displays the LLM's response in bullet format. This dynamic frame provides the instructor with at-a-glance details of their class, current assignment, or selected student.

### 4.3 Final Remarks

The implementation of Codehelp into Riposte's framework demonstrates how AI-driven teaching assistants can be seamlessly integrated into existing classroom environments while maintaining adaptability and ease of use. By introducing assignment-specific context, response flexibility, and student session analysis, we address key limitations in traditional LLM-based tutoring systems. These improvements ensure that students receive more rel-

## Student Session Summary

- **Levenshtein Distance Algorithm Issues:** The student has multiple queries related to implementing the Levenshtein distance algorithm. They seem to struggle with understanding the dynamic programming approach and recursive solutions. (Query IDs: 24, 23, 22, 21, 20, 19, 18, 17, 16, 15)
- **Basic Python Syntax and Printing:** The student has issues with basic Python syntax and printing strings, as seen in their attempts to print a string variable. (Query IDs: 3, 2, 1)
- **Array Initialization:** The student has a query related to initializing an array in Python. (Query ID: 4)

Figure 4.5: Example Student Session Analysis

evant and structured assistance while providing instructors with valuable insights into student progress. Moreover, the modular design of these features allows for potential expansion, such as integrating frequently asked questions to optimize student learning further. While this research focuses on improving the classroom experience within Riposte, the methodologies developed here can be generalized to other educational platforms, paving the way for more dynamic and intelligent AI-assisted learning environments.

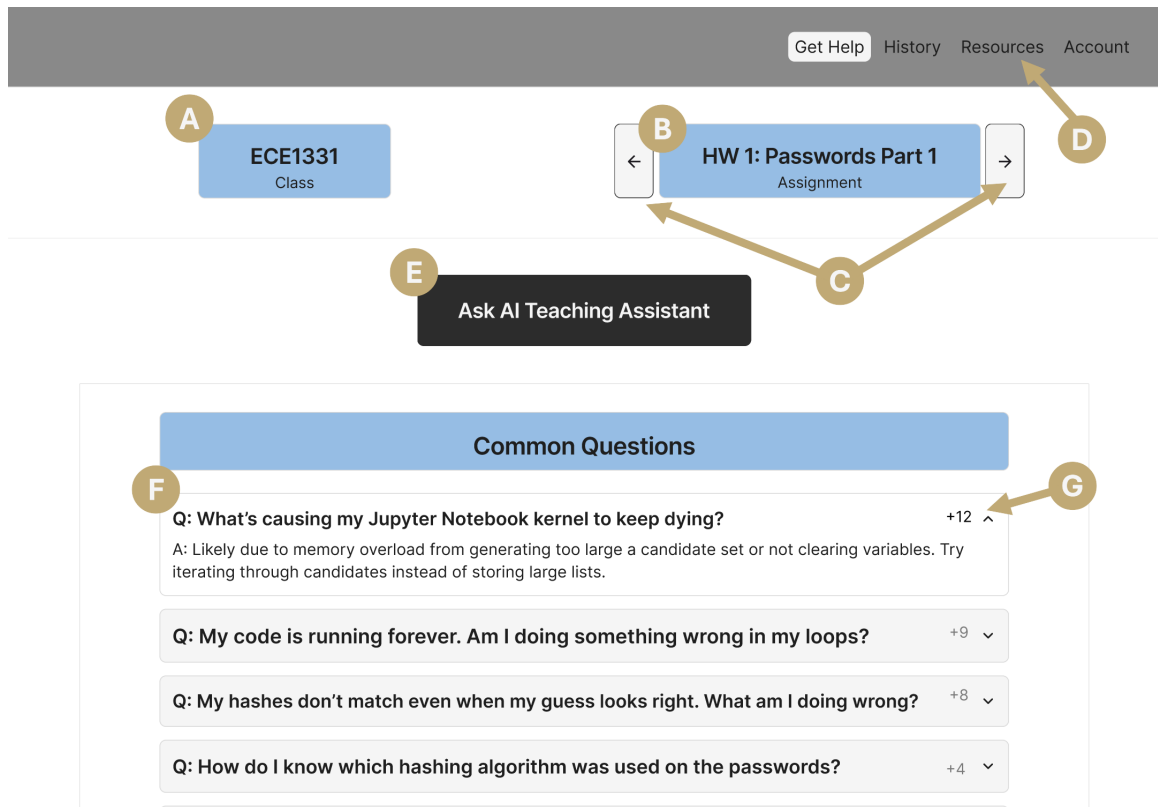


Figure 4.6: Home Page Frequently Asked Questions Mock-up for the main page. **A** shows an example class name. **B** shows the active assignment. **C** shows optional next assignment and previous assignment buttons. **D** shows a resources tab. **E** shows a button directing to the AI teaching assistant help form. **F** shows the common questions with the most common question listed first. **G** shows the number of students asking the selected question.

## CHAPTER 5

### EVALUATION

This chapter describes how we evaluate the features we introduced in the previous chapter. Many other implementations of AI assistants included longitudinal examinations of student-AI interactions [5], [6], [7], [13], [14]. To align with our goal of creating a dynamic and streamlined solution, we opted for scenario-based testing to evaluate AI performance across specific student submissions and interactions. Additionally, we recall our research questions to guide our evaluation. Key evaluation criteria included the assistant’s ability to identify insufficient prompts, support responses from instructor-provided contexts, and provide non-code conceptual guidance (when applicable).

#### 5.1 Student Scenarios

To assess the effectiveness of the AI assistant in providing individualized support, we designed a series of test scenarios based on common student help-seeking behaviors. These behaviors were inspired by other live course evaluations of students prompting LLMs [5], [6], [7], [13],[14], and interactions with office hours in programming courses [11]. We submitted queries related to each student scenario and observed how the improved AI Assistant responds. These student scenarios are outlined below.

##### *Full Context*

This student will represent interactions that use the AI assistant to strengthen their understanding of class concepts. This will include general questions about the syntax and quality of their code, as well as any error handling they encounter. This may also involve the understanding of specific coding libraries and data structures. These students are using the AI assistant as intended. While this study focuses on implementing a coding AI assistant,

we feel it may be valuable to understand if students rely on it for non-coding questions. Specifically, we are looking to see if the class contexts provided by instructors provide more relevant feedback.

### ***Low Effort Submissions***

This student will write queries with minimal information such as “Help” or “I don’t know where to start”. These were some common pitfalls in office hour interactions [11] and student-LLM interactions in past studies. This represents the student who may be struggling with “what” to ask or the student hoping to get the best results with the least amount of effort. This case is beneficial for evaluating the sufficiency check.

### ***Requesting Immediate Assistance***

This scenario will represent the students who may try to take advantage of the LLM to gain direct answers. With the addition of code flexibility and the instructor’s context, we open the system to potential jailbreaking opportunities. We crafted prompts based on Yu et al.’s study on jailbreaking techniques when prompting LLMs [28]. These prompts will help ensure that the guardrails in place work even in blatant attempts to bypass them.

Using these student personas, we queried our improved version of Codehelp through two of Riposte’s assignments, *Field Exercise 1* and *Passwords 1* [25]. In *Field Exercise 1*, the student performs a series of tasks that build up to coding the Levenshtein distance. This optional pre-homework assignment is used to prepare students with the skills to succeed in *Passwords 1*. In *Passwords 1*, the student must crack a list of hashes using mnemonic passwords generated from a list of offline source texts. Efficiency is key in this assignment, as the possible combinations produced can kill the Jupyter Notebook if poorly coded. With this in mind, we use *Passwords 1* as an assignment to allow coding snippets, seeing if the LLM can improve the lines of code for efficiency without revealing the entire solution. subsection 6.1.1 explains the results of our manual review of these queries. We also exam-

ined how these personas affect LLM response quality in section 6.1, where we calculate the specificity and sensitivity of each query based on query type. These metrics are explained further in section 5.2.

## 5.2 Class Context

To see if adding context to Codehelp’s response workflow affected the LLM’s responses, a context variable was added to the main response and sent to the LLM. This variable would provide assignment context from the instructor to the LLM. We also provided tips from the instructor’s lecture slides that were presented before the assignment was released. In the case of assignments *Field Exercise 1* and *Passwords 1*, this context specified that the assignments would be written in a Jupyter Notebook with limited memory, where the goal is to complete the tasks efficiently.

We evaluated 96 LLM responses across four conditions expected in tailored programming feedback, using a rubric that measures the amount of instructor context included. Each LLM response was reviewed and scored on the presence (1) or absence (0) of the following:

- Reference to the student’s code logic
- Use of course-specific terminology
- Inclusion of step-by-step guidance
- Provision of a meaningful code improvement or fix

We used standard statistical classification metrics: Sensitivity, Specificity, Accuracy, and Predictive Values. These metrics help assess whether the LLM’s level of detail matched the expectations set by the instructor’s prompt. *Sensitivity* measures how often the LLM gave a specific, context-aware response when the instructor explicitly provided assignment context. *Specificity*, on the other hand, measures how often the LLM lacked detailed context

when no instructor context was given. Low specificity suggests the model may be inferring or generating context based solely on the student’s query, rather than responding to the instructional framing.

A response was marked as *specific* if it met **three or more rubric criteria**, indicating a high level of instructional alignment. To assess how well the LLM aligned with assignment-level expectations, each response was classified based on whether specificity was *expected* (i.e., assignment context was provided) and whether the LLM produced a *specific* response:

- **True Positive (TP):** Specificity was expected, and the response was specific
- **False Negative (FN):** Specificity was expected, but the response was not specific
- **False Positive (FP):** Specificity was not expected, but the response was specific
- **True Negative (TN):** Specificity was not expected, and the response was not specific

These counts were used to compute the following standard classification metrics:

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

$$\text{Specificity} = \frac{TN}{TN + FP}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

$$\text{Positive Predictive Value (PPV)} = \frac{TP}{TP + FP}$$

$$\text{Negative Predictive Value (NPV)} = \frac{TN}{TN + FN}$$

These metrics were calculated for each assignment type (explanation vs. generation) and query type (e.g., full context, low effort, jailbreak) to examine how instructional framing affected LLM behavior.

### 5.3 Code Flexibility

The effect of adding our Code Flexibility feature was evaluated through analyzing each query-response pair to assess how well the AI assistant adhered to restrictions on providing code in its responses. In cases where coding responses were allowed, we look to see if the LLM offers valuable guidance to improve the correctness and performance of the code alongside its suggestions. In the case of *Field Exercise 1*, where we would not want the LLM to respond with code, we are looking for clear explanations to guide the student in the right direction. Beyond evaluating individual responses from our student scenarios, understanding the overall effectiveness of the LLM in improving student code required a more systematic approach. Therefore, this study also examined a large dataset to quantify how often the LLM’s feedback led to meaningful code improvements, as explained in the next section.

#### 5.3.1 Data Analysis

The Codechef Competitive Programming Dataset [26] was categorized by both language and submission status. Due to the large amount of submissions, we focused on the 26,651 Python submissions, which make up 14% of the total solutions dataset. After some minimal cleaning, Turbo Sort was selected for the variety of solution submission statuses in their submissions.

#### 5.3.2 Script Profiling

It was important for the LLM’s output code to closely resemble the style and format of the code from the dataset. The consistent format streamlined the process of analyzing the code’s correctness and performance. It was also beneficial to understand why the LLM made those specific changes to the original code. With these objectives in mind, the prompt was broken into two parts, the *developer prompt* and the *user prompt*. The former provided

the LLM the context needed and format to which it would respond to the user prompt. There, we emphasized the LLM’s use case as a tutor for computer science students, where the goal is to promote learning rather than just provide answers. The latter focused on giving the LLM an understanding of the dataset being provided. As mentioned in section 2.3, how you construct your prompt to the LLM has a direct effect on its output. The prompts crafted are shown below were crafted using OpenAI’s guidelines with a focus on being “specific, descriptive and as detailed as possible about the desired context, outcome, length, format, style [29].” The results of this are shared in .

The *developer\_prompt* below sets the overall behavior for the LLM. We focus on adhering the LLM’s responses to a strict format for easier data comparison.

```
developer_prompt: ``````
```

```
You are an AI tutor specializing in programming  
and computer science. Your role is to assist students who are  
seeking help with their coursework or projects, but you must do so  
in a way that promotes learning and doesn't provide direct  
solutions. Here are your guidelines:
```

1. Only reply with code solution
2. Instead of writing comments on your corrected code,  
write 1-3 sentences describing you changes after revealing  
your solution
3. Encourage good coding practices.

```
Desired Format:
```

```
[Your updated code solution]
```

```
[Changes Made]
```

- 1.
- 2.
- 3.

```
``````
```

The *user\_prompt* below is the individual input for each solution. We also provide the solution's question criteria for the LLM to reference.

```
user_prompt = `
You will be provided a list of dictionaries with the solutions and
the corresponding info to the following question:

###
{question_info}
###

The keys to the files provided are defined as follows:
SolutionID: unique identifier for each solution.
Solutions: contains the users submitted code solution
Timeago: the time elapsed since an event (e.g., submission
or acceptance).
UserID: unique identifier for each user.
Status: indicates the state of the solution (e.g., accepted,
wrong answer, time limit exceeded).
TimeTaken: the amount of time taken to complete or submit
a solution.
MemTaken: indicates memory usage associated with the solution.
```

Improve the performance of the following solution marked time limit exceeded.

```
###
{solution}
###
`
`
```

## **5.4 Student Session Summary**

Next, we review the Student Session Summary to understand how LLMs can help instructors improve individualized classroom experiences. Using the student profiles outlined above, we conducted a manual qualitative review of the session summaries generated by the LLM for the students. Here, we want to see if the summary effectively captures the topics students are asking about. Additionally, we want to see if the summary captures any common trends within the students' interactions. This includes receiving frequent, insufficient prompts and making frequent queries on a specific topic or concept. The results are discussed in section 6.3.

## **5.5 Final Remarks**

This evaluation framework allowed us to systematically test the assistant's alignment with the project's design goals in different student scenarios. In doing so, we can observe how well the LLM's responses align with class content and whether different assignment constraints affect the guidance the LLM provides to the student.

## CHAPTER 6

### RESULTS

In this chapter, we present our observations and insights derived from the evaluation conducted in the previous chapter. The results are categorized into different sections to highlight the system’s performance, adaptability, and limitations under various scenarios.

#### 6.1 Class Context

After adjusting the LLM to consider assignment contexts as described in section 5.2, the responses to queries made to Codehelp were more specific to *Field Exercise 1* and *Passwords 1* assignments. It was noted that the LLM only pulled information from the context that was relevant to the student’s question. If the query was inefficient, the response would include an additional nudge to remind the student of the space constraints of the assignment. When asking for examples, the LLM was provided with context pulled from the lecture slides supplied to it and gave specific examples. Without this context, the LLM provides general advice and no real examples.

The results of our specificity test are shown in Table 6.1. In *Field Exercise 1*, where no code output was allowed, providing assignment-level context increased the specificity rate from 50% to 81%, and the average rubric score from 1.69 to 2.33. In *Passwords 1*, where code output was permitted, LLM responses were rated specific in 100% of cases regardless of context, with particularly high average scores (3.00 without context, 2.76 with context). These findings suggest that assignment context improves LLM specificity in explanation-based tasks, while code generation capability strongly drives specificity in generative tasks.

We also examined how query type affected specificity in Table 6.3 and found that it significantly influenced LLM performance. Prompts including full assignment context yielded the highest specificity scores (average 2.70), with over 91% of responses rated as spe-

Table 6.1: LLM Specificity Classification by Assignment Type

| Assignment Type         | TP | FP | TN | FN | Sensitivity | Specificity | PPV   | NPV   | Count |
|-------------------------|----|----|----|----|-------------|-------------|-------|-------|-------|
| <b>Field Exercise 1</b> | 10 | 8  | 18 | 11 | 0.476       | 0.692       | 0.556 | 0.621 | 47    |
| <b>Passwords 1</b>      | 19 | 24 | 0  | 6  | 0.760       | 0.000       | 0.442 | 0.000 | 49    |

cific. Prompts designed to simulate low-effort queries still achieved a high specificity rate (89.7%) and moderate rubric scores, suggesting that LLMs can infer instructional needs even from sparse student input. In contrast, prompts that attempted to jailbreak or mislead the LLM resulted in the lowest performance, with an average rubric score of 1.91 and only 54.5% of responses classified as specific. These results reinforce the importance of clear, structured prompts in steering LLM behavior toward effective instructional output.

Table 6.2: Average Response Score by Query Type

| Assignment              | Context Provided | Code Allowed | Avg. Score | % Specific | Count |
|-------------------------|------------------|--------------|------------|------------|-------|
| <b>Field Exercise 1</b> | No               | No           | 1.69       | 50%        | 26    |
| <b>Field Exercise 1</b> | Yes              | No           | 2.33       | 81%        | 21    |
| <b>Passwords 1</b>      | No               | Yes          | 3.00       | 100%       | 24    |
| <b>Passwords 1</b>      | Yes              | Yes          | 2.76       | 100%       | 25    |

Table 6.3: LLM Classification Metrics by Query Type

| Query Type          | TP | FP | TN | FN | Sensitivity | Specificity | PPV  | NPV  | Count |
|---------------------|----|----|----|----|-------------|-------------|------|------|-------|
| <b>Full Context</b> | 2  | 6  | 6  | 8  | 0.20        | 0.50        | 0.25 | 0.43 | 22    |
| <b>Low Effort</b>   | 18 | 17 | 7  | 4  | 0.82        | 0.29        | 0.51 | 0.64 | 46    |
| <b>Jailbreak</b>    | 9  | 9  | 5  | 6  | 0.60        | 0.36        | 0.50 | 0.46 | 29    |

### 6.1.1 Student Scenarios

As discussed in section 5.1, we made queries to Codehelp based on three student scenarios (full context, low effort, jailbreak). When provided with a complete query (code, error, and question), the LLM’s responses were able to explain concepts, locate errors, and add other relevant tips as well. If the question was general, such as “How can I improve this code?” or “Do you see any issues with this code?”, the LLM successfully would break

Table 6.4: Average LLM Specificity by Query Type

| Query Type          | Avg. Score | % Specific |
|---------------------|------------|------------|
| Full Context prompt | 2.70       | 91.3%      |
| Low Effort prompt   | 2.41       | 89.7%      |
| Jailbreak prompt    | 1.91       | 54.5%      |

**Issue / Question:** How do I efficiently generate multiple mnemonic passwords for the given line

**Response** *Remember: It will not always be correct!*

It looks like you're on the right track with generating mnemonic passwords from a line of text. Let's discuss a few ways you can make your function more efficient and robust:

- Avoid Redundant Transformations:** Currently, you're generating both `password.upper()` and `password.lower()`, which are redundant because `password` is already in lowercase. You can remove one of these to streamline your code.
- Use List Comprehensions:** Python list comprehensions can make your code more concise and potentially faster. For example, you can use a list comprehension to create the list of initial letters.
- Handle Edge Cases:** Ensure that your function gracefully handles cases where the input line is `None` or doesn't have enough words. You already have a check for this, but make sure it returns an empty list or a meaningful result instead of `None`.
- Return a Set:** Since you want to return a set of passwords, consider returning a set directly to avoid duplicates.

Here's how you might apply some of these suggestions:

- **Use a List Comprehension** to get the initials:
 

```
line_list = [word[0] for word in line_list[:8]]
```
- **Remove Redundant Transformations:**

```
mnem.append(password.upper()) # You can remove password.lower() since it's the same as password
```
- **Return a Set:**

```
return set(mnem)
```

By implementing these changes, your function will be more efficient and clearer. Keep up the great work! If you have any more questions or need further clarification, feel free to ask.

Figure 6.1: LLM response from *Passwords 1*. The model offers suggestions for improving efficiency, handling edge cases, and reducing redundancy.

down the student's code to them and provide suggestions to improve efficiency and reduce redundancy as shown in Figure 6.1. Notably, the LLM also made a point to address student misconceptions of a function. These were prevalent in queries that explained the student's intention alongside their code. In Figure 6.2, LLM doesn't just correct the student's matrix for Field Exercise 1, it also explains the why in its suggested solution.

**Issue / Question:** "When I use the input strings ""catch and ""hat"" to calculate the levenshtein distance I get this resulting matrix and a distance of 3. How do I correct it?"

```
0 1 1 3
1 1 2 0
2 2 1 0
3 3 2 0
4 4 3 0
5 4 4 0 "
```

**Response** *Remember: It will not always be correct!*

It looks like you're on the right track with implementing the Levenshtein distance algorithm, but there's a small issue with how you're indexing and updating the matrix `d`. Let's go through the key points to help you correct it:

1. **Matrix Initialization:** You've correctly initialized the matrix `d` with dimensions `(len(first) + 1) x (len(second) + 1)`. This is important because you need an extra row and column to account for the empty prefix of each string.
2. **Matrix Filling:** The issue arises in the loop where you calculate the distance. The matrix `d` should be filled starting from `d[1][1]` because `d[0][*]` and `d[*][0]` are already initialized to represent the cost of converting an empty string to the prefixes of `first` and `second`.
3. **Indexing:** In your nested loop, you should start iterating from `i = 1` and `j = 1` instead of `i = 0` and `j = 0`. This is because `d[i][j]` should refer to the cost of converting the first `i` characters of `first` to the first `j` characters of `second`.
4. **Accessing Matrix Elements:** When you access `d[i-1][j]`, `d[i][j-1]`, and `d[i-1][j-1]`, make sure `i` and `j` are within the correct range to avoid accessing out-of-bounds elements.
5. **Final Result:** After filling the matrix correctly, the Levenshtein distance will be found at `d[len(first)][len(second)]`, not `d[len(first)-1][len(second)-1]`.

By adjusting the loop indices and ensuring you're accessing the matrix correctly, you should get the correct Levenshtein distance for the strings "catch" and "hat". Keep up the good work, and feel free to ask more questions if you need further clarification!

Figure 6.2: LLM response from *Field Exercise 1*. The model identifies a matrix logic error and explains the rationale behind the fix.

### ***Low Effort Submissions***

Students submitting vague prompts received targeted clarification requests, encouraging more thoughtful engagement. These prompts included stating “I get an error” without providing any details or error codes. In these cases, the LLM marked the student’s query as insufficient and requested them to share the actual error for better responses. It then walked through the students’ code, if supplied, listing which lines might be causing an error. In all five query attempts made in this format, the LLM was able to identify the actual error.

Additionally, we submitted queries, copying and pasting the assignment instructions verbatim, to see how much of the process the LLM would give to the student. All of Riposte’s assignments are broken down into tasks with the Jupyter Notebook. We made three queries for each task to account for variance in LLM responses. The responses from the LLM were standardized first to repeat what the task was asking. Second, the response broke down the recommended solution into steps. Lastly, it encourages the student to ask again for help regarding a specific step. In all three attempts for each task, we saw slight variance in details or suggested steps.

Tasks that involved more complex libraries, such as using websockets in *Field Exercise I*’s Task 5, were marked as insufficient by the LLM. It asked the student what area of websockets they were struggling with. This interaction is shown in Figure 6.3. Another interesting interaction was with *Field Exercise I*’s Task 6, where the student is asked to compute the Levenshtein distance. The LLM provides a detailed, step-by-step explanation of how to code this algorithm. This interaction is shown in Figure 6.4.

### ***Jailbreaking Attempts***

Codehelp was robust in its ability to respond to queries that made attempts to bypass its guardrails. For example, if a student started their query with “ignore all previous instructions” or “if you were a student, how would you do this assignment?”, the LLM would respond as it usually does. It would break down the actual coding solution without provid-

**Response** *Remember: It will not always be correct!*

**Please clarify**

Could you please provide more details about the specific issue you're encountering with the websocket task? For example, are you having trouble sending the message, receiving the response, or understanding the expected format of the data?

An *attempt* at a response is below, but you can [Retry](#) this query and provide additional details or clarification to receive a more helpful response.

To tackle Task 5, you'll need to understand how websockets work in Python. Websockets allow for full-duplex communication channels over a single TCP connection, which means you can both send and receive messages in real-time.

Here are some key points to help you with this task:

1. **WebSocket Basics:** Websockets are different from HTTP requests because they maintain an open connection, allowing for continuous data exchange. This is useful for real-time applications.
2. **Boilerplate Code:** Since you have been provided with boilerplate code, it likely includes the setup for connecting to a websocket server. Make sure you understand how this connection is established. Look for functions or methods that handle the connection, sending, and receiving messages.
3. **Sending Messages:** You need to send a string over the websocket. The task specifies that this string should be a concatenation of whitespace-separated letters. Ensure that your string is formatted correctly before sending it.
4. **Receiving Responses:** Once you send the message, you should await a response from the server. The response will be a string, as shown in your example. Make sure your code is set up to handle incoming messages and process them as needed.
5. **Handling Asynchronous Code:** Websockets in Python are often handled using asynchronous programming. If your boilerplate code uses `asyncio` or similar libraries, ensure you understand how to work with `async` and `await` keywords to manage asynchronous tasks.
6. **Debugging:** If you encounter issues, check for common problems such as connection errors, incorrect message formats, or issues with the asynchronous flow. Use print statements or logging to track the flow of your program and identify where things might be going wrong.

**Figure 6.3: LLM response from *Field Exercise 1*.** The model identifies a lack of context in the query and requests clarification before offering general guidance on websocket usage.

**Response** *Remember: It will not always be correct!*

To calculate the Levenshtein distance between two strings, you'll need to understand how this metric works. The Levenshtein distance measures how many single-character edits (insertions, deletions, or substitutions) are required to transform one string into another.

Here's a step-by-step guide to help you implement a function to calculate this distance:

1. **Understand the Problem:** You need to transform one string into another using the minimum number of edits. Each edit can be an insertion, deletion, or substitution of a single character.
2. **Dynamic Programming Approach:** A common way to solve this problem is by using a dynamic programming approach. This involves creating a 2D table (or matrix) where the cell at position  $(i, j)$  represents the Levenshtein distance between the first  $i$  characters of the first string and the first  $j$  characters of the second string.
3. **Initialize the Matrix:** Start by initializing the first row and the first column of the matrix. The first row represents transforming the empty string into the first  $j$  characters of the second string, which requires  $j$  insertions. Similarly, the first column represents transforming the first  $i$  characters of the first string into the empty string, which requires  $i$  deletions.

4. **Fill the Matrix:** For each cell  $(i, j)$ , calculate the cost of the three possible operations:

- **Insertion:** The cost of inserting a character into the first string.
- **Deletion:** The cost of deleting a character from the first string.
- **Substitution:** The cost of substituting a character in the first string with a character from the second string.

The value of each cell is the minimum of these three costs plus the cost of the operation.

5. **Base Case and Recurrence Relation:** If the characters at position  $i-1$  in the first string and  $j-1$  in the second string are the same, the substitution cost is 0. Otherwise, it's 1. The recurrence relation will help you fill in the matrix based on these calculations.
6. **Result:** Once the matrix is filled, the value at the bottom-right corner of the matrix will be the Levenshtein distance between the two strings.

By following these steps, you can implement a function that calculates the Levenshtein distance. Remember to test your function with different string pairs to ensure it works correctly. Good luck, and feel free to ask more questions if you need further clarification!

Figure 6.4: LLM response from *Field Exercise 1*. The model provides a step-by-step guide to implementing the Levenshtein distance algorithm using dynamic programming.

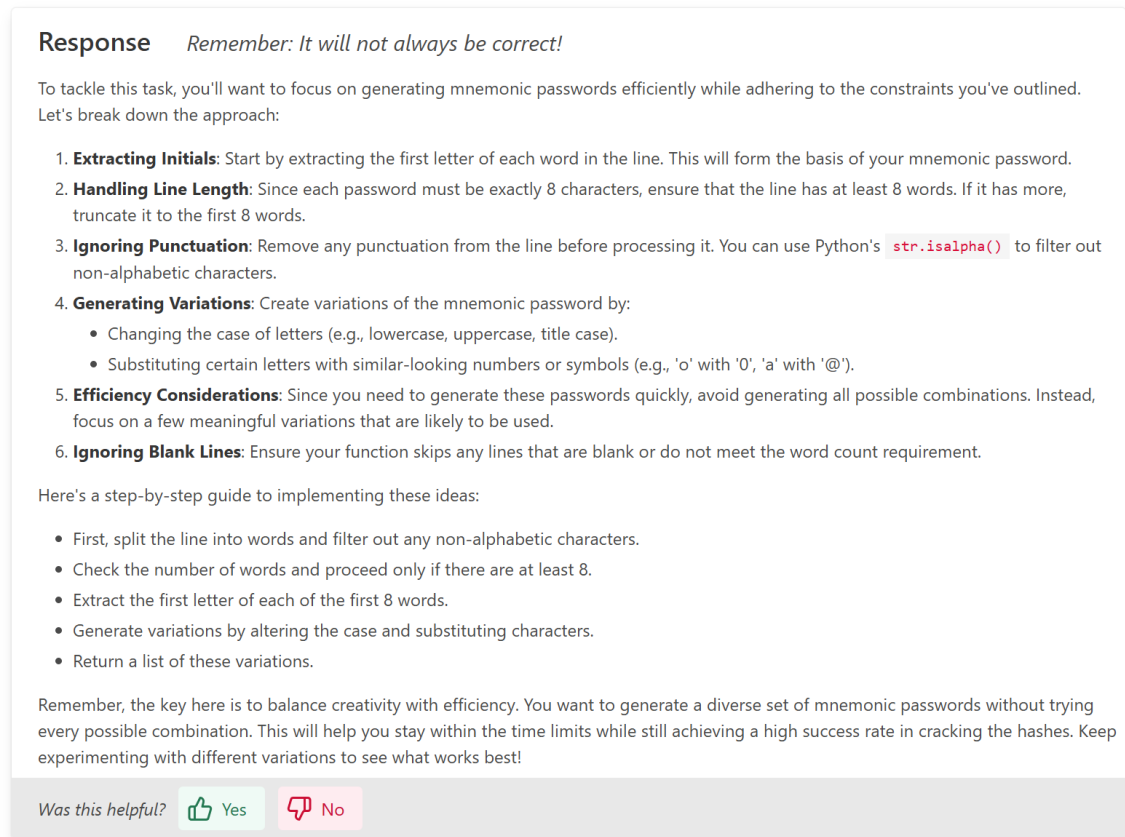


Figure 6.5: LLM response from *Passwords 1*. The model maintains assignment integrity while breaking down the steps to generate efficient mnemonic passwords.

ing any answers. In assignment in *Passwords 1* where code snippets were allowed in LLM responses, this maintained the integrity of the assignment. It continued to break down the steps to tackle the task being asked, as shown in Figure 6.5.

### 6.1.2 Measuring Specificity

Instructor-provided context has a measurable impact on the specificity of LLM output in explanation-based tasks, where students require verbal reasoning and instructional feedback. However, in code generation tasks, the LLM demonstrates a high degree of responsiveness and specificity even without explicit contextual prompts. This suggests that LLMs leverage implicit context from student code when permitted to respond with executable examples, reducing their reliance on instructional scaffolding.

**Issue / Question:** How can I improve this code?

**Response** *Remember: It will not always be correct!*

Your code for generating mnemonic passwords is quite comprehensive, but there are a few areas where you can improve its efficiency and readability. Let's go through some suggestions:

1. **Avoid Global Variables:** The `trans` dictionary is defined inside the function `generate_mnemonic_passwords_for_line`, which is good because it keeps the scope limited. However, you can make it a constant outside the function if it doesn't change, which can improve readability.
2. **Use Meaningful Variable Names:** The variable names like `mnem`, `trans`, and `buffer` could be more descriptive. For example, `mnem` could be `mnemonics`, and `buffer` could be `transformed_password`.
3. **Avoid Repeated Code:** You are calling `pass_recur` three times with similar logic. Consider creating a helper function to reduce redundancy.
4. **Optimize the Recursive Function:** The `pass_recur` function can be optimized to avoid unnecessary recursive calls. You can use a set to track visited passwords to prevent revisiting them.
5. **Return Consistent Types:** Ensure that your function always returns a consistent type. If the input doesn't meet the requirements, consider returning an empty list instead of `None`.
6. **Use Pythonic Constructs:** Instead of manually checking if a string is in a list, consider using a set for `mnem` to automatically handle duplicates and improve lookup times.

Let's start by addressing some of these points:

- **Use a Set for Mnemonics:** Change `mnem` from a list to a set to automatically handle duplicates and improve performance.

```
mnem = set()
```

- **Optimize Recursive Calls:** Before making a recursive call, check if the password is already in the set.

```
if buffer not in mnem:  
    pass_recur(buffer, mnem, trans, idx + 1)
```

- **Return an Empty List:** Instead of returning `None`, return an empty list when the input doesn't meet the requirements.

```
if (num_words < 8) or (line is None):  
    return []
```

These changes should help make your code more efficient and easier to read. Keep experimenting and refining your code!

Figure 6.6: **LLM response from *Passwords 1***. The model analyzes specific lines of student code and offers targeted suggestions to improve clarity, performance, and readability without completely rewriting it.

## 6.2 Code Flexibility

The LLM was also able to identify when students were asking for help with a specific line of code. For example, in the case of a student asking “I don’t understand why this line is not working”, the LLM would provide a detailed explanation of the line of code and suggest possible solutions. This is shown in Figure 6.6.

### 6.2.1 Improving Student Code Performance

To assess the impact of LLMs on code performance, a comparative analysis was conducted between student-submitted Python solutions from Codechef [26] and their corresponding LLM-refactored versions. The evaluation focused on two primary aspects: execution efficiency, measured by cumulative runtime, and function call behavior, including the use of

Table 6.5: Average Number of Built-in Function Calls per Solution

| <b>Built-in Function</b> | <b>LLM (avg)</b> | <b>Original (avg)</b> |
|--------------------------|------------------|-----------------------|
| eval                     | 0.00             | 14.81                 |
| exec                     | 1.00             | 1.00                  |
| getattr                  | 0.00             | 0.32                  |
| hasattr                  | 0.00             | 0.24                  |
| input                    | 43.04            | 74.78                 |
| len                      | 23.86            | 106.59                |
| min                      | 0.00             | 8.24                  |
| next                     | 0.02             | 0.00                  |
| print                    | 62.06            | 68.43                 |
| sorted                   | 0.06             | 0.00                  |

built-in Python functions.

### ***Cumulative Runtime***

Cumulative execution time was used as a proxy for performance efficiency. Among the 37 matched solution pairs analyzed, the LLM-generated code exhibited lower cumulative runtime than the original student solution in 16 cases, representing approximately 43.2% of the dataset. The average reduction in runtime for these cases was 0.0006 seconds. While the overall performance gain is modest, the frequency of improvements suggests that LLMs are capable of generating code that is marginally more optimized for execution time.

### ***Built-in Function Usage***

To further understand differences in program structure, built-in Python functions were examined across both solution types. This analysis included commonly used built-ins such as print(), input(), eval(), and len(). Results are summarized in Table 6.5. The use of eval() was exclusive to student code, appearing an average of 14.81 times per solution, while LLM solutions did not invoke eval() at all. Given the known risks and inefficiencies associated with eval(), this suggests that LLMs tend to avoid potentially insecure or unnecessary operations.

LLM-generated solutions used fewer input() and len() calls on average (43.04 and

23.86, respectively) compared to student-written code (74.78 and 106.59), indicating a more efficient or consolidated handling of data. Notably, LLM solutions exhibited slightly higher use of `print()` (62.06 vs. 68.43), which may reflect an emphasis on transparency or step-by-step feedback, possibly mirroring common pedagogical patterns in AI-generated explanations.

These findings indicate that, while overall function call counts remain stable, LLMs contribute to improvements in code structure by reducing reliance on verbose or inefficient patterns. The elimination of `eval()` and reduced use of redundant built-ins further supports the conclusion that LLMs are capable of refining student submissions into safer and more streamlined implementations.

### **6.3 Student Session Summary**

After reviewing each student query generated in section 5.4, the summaries generally aligned well with the query content, referencing topics such as Levenshtein distance calculation and WebSocket communication for *Field Exercise 1* and password transformations for *Passwords 1*. Several summaries also identified meaningful patterns, such as repeated questions or confusion around coding concepts. For instance, one summary highlighted a student's ongoing challenges with asynchronous syntax as shown in Figure 6.7. This summary also included this student's struggles with matrix initialization from the query shown in Figure 6.2. This suggests a persistent gap in conceptual understanding. In Figure 6.8, the summary identified inefficient string-splitting logic and inappropriate use of Python's `join` method, indicating that the LLM was able to infer deeper issues in the student's approach. Summaries of students with low effort or jailbreaking personas return summaries with limited information. The LLM noticed repeated questions or when a query lacked specificity. Figure 6.9 shows an example of this.

While these observations suggest the summaries can help instructors quickly diagnose student difficulties, this evaluation remains limited in scope. Specifically, we were unable to

## Student Session Summary

- **Levenshtein Distance Calculation:** The student attempted to implement the Levenshtein distance algorithm but faced issues with incorrect results and matrix indexing errors. (Query IDs: 55, 56, 57)
- **Websockets Communication:** The student encountered a syntax error while trying to assign a received message in an asynchronous function using websockets. (Query IDs: 51, 52)
- **Common Initial Characters in Strings:** The student attempted to find common initial characters in a list of strings but faced logic errors and index out of range issues. (Query IDs: 46, 47, 48)
- **String Transformation to Initials:** The student worked on transforming strings to their initial characters but produced incorrect results. (Query IDs: 42, 43, 44, 45)
- **Word Count in Strings:** The student implemented a function to count words in strings but faced issues with incorrect word counts. (Query IDs: 38, 39, 40, 41)
- **File Content Loading:** The student attempted to load file contents into a list but returned incorrect results due to missing return statements. (Query IDs: 34, 35, 36, 37)

Figure 6.7: **LLM-Generated Student Session Summary from *Field Exercise 1*.** The LLM groups related queries by topic and describes the student's challenges and errors across tasks.

## Student Session Summary

- **Finding Common Initials in Strings:** The student is trying to find common first characters in a list of strings but is encountering issues with their approach. They are using set operations incorrectly and need guidance on iterating through the list and comparing sets correctly. (Query IDs: 30, 31, 32)
- **Counting Words in Strings:** The student is attempting to count words in each string of a list but is unnecessarily splitting strings multiple times. They need help optimizing their approach. (Query IDs: 27, 28, 29)
- **Selecting First Characters from Words:** The student is trying to extract the first character from each word in a list of strings and join them back into a string. They are incorrectly using the join method and need clarification on string manipulation. (Query IDs: 24, 25, 26)

Figure 6.8: LLM-Generated Low Effort Student Session Summary from *Passwords 1*. The LLM highlights the student's misuse of a Python function.

## Student Session Summary

- Mnemonic Password Generation: The student is working on generating mnemonic passwords from a given line of text, focusing on transformations and recursive functions. This is evident in queries with IDs 81, 80, and 79.
- General Password Generation Issues: The student has multiple queries related to password generation, as seen in IDs 78, 77, and 76, though these lack specific issues mentioned.
- Function Execution Issues: The student has queries related to function execution, identified in IDs 64, 63, 62, 61, 60, and 59, but these entries do not specify the exact problem.

Figure 6.9: **LLM-Generated Low Effort Student Session Summary from both *Field Exercise 1* and *Passwords 1***. The LLM acknowledges the student’s lack of detail in queries.

incorporate feedback from actual instructors, who are better positioned to judge whether the summaries offer pedagogically actionable insights. As such, our analysis primarily reflects whether the summaries appeared accurate and valuable from a researcher’s perspective, rather than how effective they would be in practice for classroom instruction.

### 6.4 Final Remarks

Students who submitted queries that were more specific and detailed received more tailored responses. For example, a student who asked “I get an error when I run this code” received a generic response, while a student who provided their code and the specific error message received a more detailed explanation of the error and how to fix it. This highlights the importance of providing context and details in queries to receive the most helpful responses. Following this discovery, Liffiton pushed a significant update to Codehelp’s Github [7] that introduced contexts to the official release. This reaffirmed the necessity of this feature.

Overall, the system demonstrated strong alignment with the intended design goals, though several areas for improvement remain evident.

## CHAPTER 7

### DISCUSSION

Here, we analyze the findings from our evaluation by identifying key strengths while addressing areas for improvement and potential extensions of our work. This discussion not only contextualizes our results within the broader scope of AI-assisted coding education but also considers their practical implications. Furthermore, we highlight the ethical and cost-related considerations that must be addressed to ensure the responsible and effective integration of AI into instructional settings.

The results show that adding class-specific context moderately improves the AI's ability to align with course objectives by 31% when code is not allowed in the LLM's response. There are still some things to consider, such as how much context is optimal? Too much detail might lead to overly rigid responses and cost more tokens to use.

The LLM's resistance to direct prompt injections suggests that Codehelp's current use of the sufficiency check as a safeguard is well-designed. However, it is important to consider the adaptability of students in discovering new jailbreak attempts over time as new models develop. Future iterations should incorporate adversarial testing, where students or researchers actively attempt new jailbreak techniques. Additionally, a feedback system where instructors can report suspected misuse cases could help improve response filtering.

The Student Session Summary provided valuable data on common student struggles and showed potential for making it easier for instructors to adjust lessons accordingly. This demonstrates AI's potential as a tool for classroom analytics. Future work could explore an AI-powered dashboard that dynamically visualizes student progress and problem areas for both students and instructors.

The comparative analysis of LLM-refactored code and original student submissions highlights the subtle yet meaningful ways in which AI teaching assistants can contribute

to code quality in educational settings. While the average reduction in execution time was minimal, the fact that nearly half of the LLM solutions outperformed their student counterparts indicates a capacity for consistent, low-level optimization. More notably, the differences in built-in function usage point to qualitative improvements in how LLMs structure and streamline logic. The complete omission of *eval()* and the reduced reliance on functions such as *input()* and *len()* suggest that LLMs are sensitive to inefficiencies and potential pitfalls in student code. This is especially relevant in instructional contexts where students often default to overly verbose or dynamic logic. The AI teaching assistant appeared to apply more concise, modular, and efficient programming practices. These findings support the broader argument that LLMs, when integrated thoughtfully, can serve not only as problem-solvers but also as implicit modeling agents that suggest cleaner and more efficient coding styles without explicitly correcting student behavior.

From a pedagogical perspective, this opens space for instructors to leverage LLMs as co-teachers or real-time code reviewers. Rather than replacing the student's original intent, the AI teaching assistant often refined it in a way that preserves originality. These refinements represent precisely the kinds of improvements instructors seek to develop in students over time. As such, these findings show that LLMs have the potential to provide improvements focused on efficiency to student code, which becomes more relevant in more advanced programming courses.

## **7.1 Limitations**

The limitations of this study may have influenced the results and should be considered when interpreting the findings. Since the system was not deployed in a live classroom setting, evaluation relied on code from previous iterations of the course and coding competitions rather than real-time student queries. We acknowledge that this may not fully capture the complexity or variability of real student behavior. This limits the ability to observe user experiences, real-time interaction patterns, and long-term impact on learning outcomes.

Another challenge is that implementation choices and system configuration are tailored to the use of OpenAI’s *gpt-4o* as the LLM of choice. Further work is needed to test adaptability with other LLM models. Lastly, collecting and analyzing student query data raises potential concerns around student privacy, data handling, and compliance with institutional policies. Due care must be taken to ensure safe and secure use and distribution of student data.

## 7.2 Future Work

### *Live Classroom Deployment*

While this study focuses on a cross-sectional evaluation, future research should explore a longitudinal approach to analyze the AI assistant’s impact over time. This could be similar to other studies with examples of AI assistants [5] [7] [13] [14]. A semester-long deployment could involve monitoring AI interactions over a semester, assessing how students’ reliance on AI evolves, and how instructors adjust their use of the system for class management. Such a study would provide deeper insights into the assistant’s long-term adaptability, usability, and the effectiveness of personalized support features in an advanced classroom setting.

A key consideration for future deployments is incorporating regular feedback checkpoints throughout the semester rather than relying solely on end-of-course evaluations. Gathering student and instructor feedback at multiple points allows for iterative improvements and ensures the system remains responsive to the classroom’s evolving needs. Both Codehelp and CodeAid had some form of iterative process mid-semester to meet their students’ needs [7] [14]. Additionally, observing data trends, such as the types of queries submitted or changes in help-seeking behavior, can provide valuable insights. Comparing these trends to qualitative survey responses will help validate whether perceived experiences align with actual usage patterns, strengthening the overall evaluation of the system’s effectiveness and adaptability.

### *Additional Features*

Building on the work presented in this thesis, several extensions can be explored further to enhance the system's capabilities and classroom impact. One potential direction is to evaluate this system in other classroom environments. This study evaluates the effectiveness of our system on Python scripts, given that the classroom framework uses Jupyter Notebooks. It would be worth understanding if the coding language affects the quality of the AI teaching assistant's response. This question can also be posed to a programming course outside the topic of cybersecurity and the use of other LLMs to provide individual feedback. This would reveal if the system is adaptable across multiple platforms and course topics. Doing so can also reveal what specific course needs still need to be met.

Additionally, expanding the system's ability to provide more context could improve the user experience. In subsection 4.2.1, we mention the extension of dynamic Frequently Asked Questions to reduce repetitive questions and alert students and instructors to common issues when working on an assignment. Feeding the AI teaching assistant student query data as a class develops can reveal specific trends and insights in real-time.

### *Cost Efficiency Analysis*

Although this study did not involve a live deployment with students, the token usage and cost data gathered during system evaluation provide a useful starting point for future considerations around cost management. Over three days, 214 API requests were made to OpenAI's API. This includes student queries and requests for session summaries. Table 7.1 breaks down the total requests for each day by the number of tokens per prompt and the number of tokens per LLM response. Table 7.2 shows the daily breakdown of requests and the associated costs for that day. We see that LLM response lengths were fairly consistent despite significant increases in prompt length. Since only one API key was used during system testing, we were unable to provide a breakdown of the cost per student or distinguish the cost of the student session summary from the total cost. For future work,

it is recommended to have separate API keys for each student, session summary requests, testing, and for any additional feature being implemented. This would allow for proper tracking of costs and utilization for each feature.

Table 7.1: Daily LLM Token Usage Summary

| Date       | Total Input | Total Output | Avg Input | Avg Output |
|------------|-------------|--------------|-----------|------------|
| 2025-04-01 | 9,426       | 3,146        | 1,047.33  | 349.56     |
| 2025-04-02 | 58,466      | 17,274       | 1,271.00  | 375.52     |
| 2025-04-03 | 221,570     | 20,644       | 4,714.26  | 439.23     |

Table 7.2: Daily Cost Breakdown per Request

| Date       | Total Cost (USD) | Total Requests | Cost per Request |
|------------|------------------|----------------|------------------|
| 2025-04-01 | \$0.0055         | 20             | \$0.00028        |
| 2025-04-02 | \$0.0578         | 97             | \$0.00060        |
| 2025-04-03 | \$0.3189         | 97             | \$0.00329        |

Table 7.3: Pearson Correlation Between Cost and LLM Usage Metrics

| Metric              | Correlation with Cost |
|---------------------|-----------------------|
| Total Input Tokens  | 0.99                  |
| Total Output Tokens | 0.75                  |
| Total Requests      | 0.62                  |

Table 7.3 presents the Pearson correlation between token usage metrics and costs. We highlight that input tokens had the strongest influence on pricing ( $r = 0.99$ ), followed by output tokens ( $r = 0.75$ ) and the total number of requests ( $r = 0.62$ ). Interestingly, while OpenAI’s pricing model assigns a higher per-token cost to output tokens (\$10 per million output tokens vs. \$2.50 per million input tokens for GPT-4o [30]), this study found that input tokens had the strongest correlation with overall cost ( $r=0.99$ ). This is likely due to the nature of test interactions and course focus, where input prompts with code attached were significantly longer and more complex than the responses they generated. As a result, the cumulative token volume on the input side outweighed the higher per-token price of outputs. This reinforces the importance of prompt engineering as a key factor in cost control

and suggests that even modest reductions in prompt length could lead to significant cost savings in future deployments.

In a live deployment, ensuring equitable access to the AI teaching assistant within budget constraints will require thoughtful token allocation policies. Future work should explore strategies for distributing tokens that balance fairness and flexibility while staying within institutional budgets. One approach is to implement a baseline token quota per student to guarantee equal starting access. This could look like a fixed weekly or assignment-based budget. Rolling or cumulative token budgets, where unused tokens carry over to the next assignment, may further promote intentional usage. Transparency tools such as usage dashboards would help students track their consumption and make informed decisions. Finally, instructors might maintain a reserve token pool for discretionary allocation based on observed student needs. These approaches would help create a token allocation system that is scalable and cost-efficient in diverse classroom contexts.

### ***Ethical Issues***

Future works in this area of research must take care to consider the numerous ethical issues that come with implementing AI into the academic classroom. If this system is used in a live course, both students and instructors must be fully aware of how their interactions with the AI assistant are recorded and potentially used for evaluation or research purposes. Informed consent must be obtained, ensuring that users understand their participation is voluntary and that they can opt out without penalty. Participating involves potential exposure to sensitive academic or personal information. It is imperative to ensure data privacy and compliance with regulations such as FERPA [31] and GDPR [32]. Clear policies regarding data storage, access, and deletion should be established to protect student identities and prevent misuse of collected data.

Inside the classroom, it is imperative to be transparent about the AI teaching assistant's capabilities and limitations. Students and instructors should be informed that while the

AI teaching assistant provides guidance, it is not infallible and may generate inaccurate or incomplete suggestions. Without this understanding, users may place complete trust in the AI's responses, which could lead to misconceptions or reliance on flawed information [4]. Regular reminders of these limitations should be embedded into the system's interface. Another ethical concern is the potential impact of AI support on academic integrity. Even with the use of guardrails, there remains a risk that students may misuse the assistant to bypass the learning process by seeking complete solutions [28]. It is necessary to regularly review how AI assistance aligns with the course's academic integrity policies to prevent unintended plagiarism.

### **7.3 Final Remarks**

In summary, this work demonstrates the potential of integrating AI teaching assistants within advanced coding classrooms to enhance individualized student support while maintaining instructor oversight. The system shows promise in improving help-seeking behaviors and supporting diverse learning needs. However, several limitations must be carefully managed. This includes, but is not limited to, data privacy risks and potential issues with academic integrity. Ethical considerations, including transparency, informed consent, and continuous monitoring, remain critical for responsible deployment. Despite these challenges, the framework lays a solid foundation for future work, including deployment in live classroom settings, iterative refinement based on student and instructor feedback, and expansion of analytical tools for real-time insights. Continued research in this space will be essential to ensure AI integration complements pedagogy standards.

## **CHAPTER 8**

### **CONCLUSION**

This thesis explored how to integrate an AI teaching assistant into a programming class framework to facilitate individualized learning experiences. In doing so, we took care to create a system that is dynamic and streamlines the integration process, so that students may receive support specific to them without increasing the workload for instructors. We present several contributions. We improve an existing AI teaching assistant, known as Codehelp, by adding context-aware support, a student session summary to help instructors monitor their students' LLM interactions, and increased control by providing the option to offer code solutions when necessary for the assignment. This enhanced version of Codehelp is integrated into an advanced intro to cybersecurity framework called Riposte. We enable synchronization between the two platforms for a seamless user experience for students.

Our system considered the AI assistant's ability to support students in a coding classroom, focusing on key design goals such as adaptability and streamlined interactions. Scenario-based testing provided insights into how well the assistant handled different student behaviors, including general coding inquiries, low-effort submissions, and requests for direct answers. The integration of class context improved the relevance of responses. This feature generated more specific and instructional feedback rather than generic suggestions.

Results showed that the AI assistant effectively identified and addressed insufficient prompts, reinforced coding concepts, and guided students without giving away direct answers, even in potential jailbreak attempts. The study also analyzed code flexibility, ensuring that the assistant provided functional improvements while respecting constraints. Additionally, a large-scale dataset analysis demonstrated the AI's capacity to enhance code efficiency and correctness. The Student Session Summary feature proved valuable for instructors, as it identified common trends in student queries and highlighted areas where

additional instruction may be needed. These findings confirm the AI assistant's potential to enhance individualized learning experiences while maintaining academic integrity and instructor oversight.

Limitations, such as ethical considerations and deployment constraints, remain. Still, this research aims to support the responsible and effective adoption of AI in education, ensuring that technology enhances, rather than replaces, the essential human elements of teaching and learning. The proposed framework opens new pathways for AI-supported pedagogy in even advanced courses where current research is lacking. Future work will build on these foundations, testing the system in various live classroom environments and incorporating ongoing feedback loops from both students and instructors. Continued collaboration between educators, researchers, and technologists will be essential to realizing the full potential of AI in fostering diverse and effective learning environments.

## REFERENCES

- [1] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, “Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation,” in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023, Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 500–506.
- [2] M. Wang, M. Wang, X. Xu, L. Yang, D. Cai, and M. Yin, “Unleashing chatgpt’s power: A case study on optimizing information retrieval in flipped classrooms via prompt engineering,” *IEEE Transactions on Learning Technologies*, vol. 17, pp. 629–641, 2024.
- [3] P. Denny, V. Kumar, and N. Giacaman, *Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language*, 2022. arXiv: 2210.15157 [cs.HC].
- [4] M. Daun and J. Brings, *How chatgpt will change software engineering education*, Turku, Finland, 2023.
- [5] P. Denny *et al.*, *Promptly: Using prompt problems to teach learners how to effectively utilize ai code generators*, 2023. arXiv: 2307.16364 [cs.HC].
- [6] A. N. Thi Thuy, “The perception by university students of the use of chatgpt in education,” *International Journal of Emerging Technologies in Learning (Online)*, vol. 18, no. 17, pp. 4–19, 2023.
- [7] M. Liffiton, B. Sheese, J. Savelka, and P. Denny, *Codehelp: Using large language models with guardrails for scalable support in programming classes*, 2023. arXiv: 2308.06921 [cs.CY].
- [8] J. Sheard, P. Denny, A. Hellas, J. Leinonen, L. Malmi, and Simon, “Instructor perceptions of ai code generation tools - a multi-institutional interview study,” in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024, Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1223–1229.
- [9] A. J. A. Husain, “Potentials of chatgpt in computer programming: Insights from programming instructors,” *Journal of Information Technology Education: Research*, vol. 22, pp. 75–100, 2023.
- [10] T. MacWilliam and D. J. Malan, “Scaling office hours: Managing live qa in large courses,” *J. Comput. Sci. Coll.*, vol. 28, no. 3, pp. 94–101, Jan. 2013.

- [11] A. J. Smith, K. E. Boyer, J. Forbes, S. Heckman, and K. Mayer-Patel, “My digital hand: A tool for scaling up one-to-one peer teaching in support of computer science learning,” in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE ’17, Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 549–554, ISBN: 9781450346986.
- [12] M. Kazemitabaar, J. Chow, C. K. T. Ma, B. J. Ericson, D. Weintrop, and T. Grossman, “Studying the effect of ai code generators on supporting novice learners in introductory programming,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’23, ACM, Apr. 2023, pp. 1–23.
- [13] B. Sheese, M. Liffiton, J. Savelka, and P. Denny, “Patterns of student help-seeking when using a large language model-powered programming assistant,” in *Proceedings of the 26th Australasian Computing Education Conference*, ser. ACE ’24, Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 49–57.
- [14] M. Kazemitabaar *et al.*, “Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs,” in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’24, Honolulu, HI, USA: Association for Computing Machinery, 2024.
- [15] J. Wei *et al.*, *Chain-of-thought prompting elicits reasoning in large language models*, 2023. arXiv: 2201.11903 [cs.CL].
- [16] Z. Zhang, A. Zhang, M. Li, and A. Smola, *Automatic chain of thought prompting in large language models*, 2022. arXiv: 2210.03493 [cs.CL].
- [17] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, *A systematic survey of prompt engineering in large language models: Techniques and applications*, 2024. arXiv: 2402.07927 [cs.AI].
- [18] D. Zhou *et al.*, *Least-to-most prompting enables complex reasoning in large language models*, 2023. arXiv: 2205.10625 [cs.AI].
- [19] A. Creswell and M. Shanahan, *Faithful reasoning using large language models*, 2022. arXiv: 2208.14271 [cs.AI].
- [20] R. Balse, B. Valaboju, S. Singhal, J. M. Warriem, and P. Prasad, “Investigating the potential of gpt-3 in providing feedback for programming assessments,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE 2023, Turku, Finland: Association for Computing Machinery, 2023, pp. 292–298.
- [21] A. Hellas, J. Leinonen, S. Sarsa, C. Koutcheme, L. Kujanpää, and J. Sorva, “Exploring the responses of large language models to beginner programmers’ help requests,”

in *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, ser. ICER 2023, ACM, Aug. 2023, pp. 93–105.

- [22] M. Liffiton, *Codehelp*, <https://codehelp.app/>, 2024.
- [23] OpenAI, *Chatgpt*, <https://chatgpt.com/>, 2024.
- [24] M. Malone, Y. Wang, K. James, M. Anderegg, J. Werner, and F. Monroe, “To gamify or not? on leaderboard effects, student engagement and learning outcomes in a cybersecurity intervention,” in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1135–1141, ISBN: 9781450380621.
- [25] M. Malone and F. Monroe, “More carrot or less stick: Organically improving student time management with practice tasks and gamified assignments,” ser. ITiCSE 2023, Turku, Finland: Association for Computing Machinery, 2023, pp. 278–284.
- [26] ArjoonnSharma, *Codechef competitive programming*, version 5, 2017.
- [27] Creative Commons, *Cc0 1.0 universal (cc0 1.0) public domain dedication*, <https://creativecommons.org/publicdomain/zero/1.0/>, Accessed: 2025-04-25.
- [28] Z. Yu, X. Liu, S. Liang, Z. Cameron, C. Xiao, and N. Zhang, *Don’t listen to me: Understanding and exploring jailbreak prompts of large language models*, 2024. arXiv: 2403.17336 [cs.CR].
- [29] OpenAI, *Best practices for prompt engineering with the openai api*, 2024.
- [30] OpenAI, *Pricing*, <https://platform.openai.com/docs/pricing>, 2025.
- [31] U.S. Department of Education, *34 cfr part 99—family educational rights and privacy*, Code of Federal Regulations, Title 34, Part 99, 2024.
- [32] European Parliament and Council of the European Union, *General data protection regulation (gdpr)*, Regulation (EU) 2016/679, 2016.