A Novel RAG-Based Chatbot Solution to Improve Textbook Material Understanding

Anirudh Chintaluri, Vatsal Sivaratri
Dr. Gabor, Dr. Torbert
TJHSST Computer Systems Lab

2025.06.02

Abstract

Mainstream large language models answer general questions from a user, but they aren't well-suited to semantically pick out relevant text from textbooks and lengthy PDF files, nor do they make use of this information retrieval towards proven active recall methods. We introduce a novel method that combines information retrieval through semantic search with active recall methods to create a chatbot that improves understanding of textbook content and the quality of study sessions. Our implementation, which leverages a Retrieval-Augmented Generation (RAG) framework and a Rephrase-and-Respond (RaR) strategy, was evaluated across six AP-level subjects, achieving an average performance score of 0.906 out of 1.0. The chatbot demonstrated strong accuracy, reasoning, and memory retention, with particularly high performance in content areas such as History and Chemistry. These results highlight the potential of our system to provide credible academic assistance across a range of subjects and formats.

Introduction

Large Language Models (LLMs) have emerged as powerful tools in their ability to answer a wide array of questions spanning many subjects. In fact, a February 2024 survey from Pew Research Center reveals that 43% of adults under the age of 30 have reported using ChatGPT in the past [5]. However, they are severely limited in the scope of education due to their inability to effectively use and cite information from specific sources. OpenAI's GPT-3.5, for example, when asked to cite sources for research purposes, fabricated or incorrectly cited sources in 55% of cases, revealing a significant risk of misinformation when models operate without grounded retrieval [8]. Additionally, they don't inherently incorporate successful learning strategies backed by research. These methods include the Feynman Technique, flashcards, practice questions, and other active recall strategies.

In recent years, Retrieval-Augmented Generation (RAG) has gone beyond traditional LLMs in that they can access information from a database both accurately and quickly [4]. They serve as a great alternative to fine-tuning in the context of improving accuracy by referring to non-AI-generated information and citing sources behind reasoning, marking a step forward for generative AI.

Currently, studies have already been using textbook information for the purpose of serving as context behind generative chatbot retrieval. For instance, Singer et al.

introduced a RAG-based chatbot model to leverage textbook knowledge in the field of ophthalmology using PDF copies of numerous ophthalmology textbooks, while also citing its textbook sources [7]. Alawwad et al. introduced a RAG-based approach specifically for textbook retrieval from middle school-level science-based textbooks, covering biology, earth science, and physics using a dataset containing small chunks of text and images—making way for up to a 9.84% improvement in model performance in multiple-choice questions [1]. Levonian et al. looked into middle school-level math and how RAG can generate better responses than traditional LLMs; however, the RAG didn't produce optimal results when questions were too grounded in the textbook [3].

One limitation common to all three of these studies is that textbook retrieval using RAG-based chatbots is limited to a single subject, and in some cases to a single textbook. The wide variety of textbooks used throughout different settings go beyond a single textbook, even among the same subject, making it more difficult for people to have a textbook-specific chatbot available to them. Additionally, these studies focus exclusively on answering questions from the textbook, even though there are much wider applications to using a chatbot based on a textbook, such as generation of flashcards and problem set generation. Furthermore, the benchmark performance for a RAG-based model is roughly 84% based on the results of these studies, suggesting that there is room for improvement on the performance of these models [1].

Recently, advancements have been made toward RAG-based models, one of which is getting the chatbot to rephrase questions before retrieving any information from the textbook, known as the Rephrase and Respond (RaR) method. When a question is phrased one way, there is a possibility that the LLM may not understand or interpret everything correctly, and answers may vary from the true answer. By rephrasing the question multiple times, we can improve output quality and significantly reduce GPT-3.5 hallucinations by up 13.16% [2].

The purpose of this research study is to introduce an application that combines RAG with RaR strategies to improve reading retention and the quality of study sessions.

To effectively prepare a method of study, the LLM needs to understand what content is being taught. The system will ask for the user to attach a file (or directory of files), allowing for full flexibility of what information the chatbot would like to retrieve.

Methods

All backend code for this project was written in Python. The frontend of the project, involving the chatbot GUI, was written in HTML, CSS, and JavaScript. A Python Flask app connects the frontend to the backend.

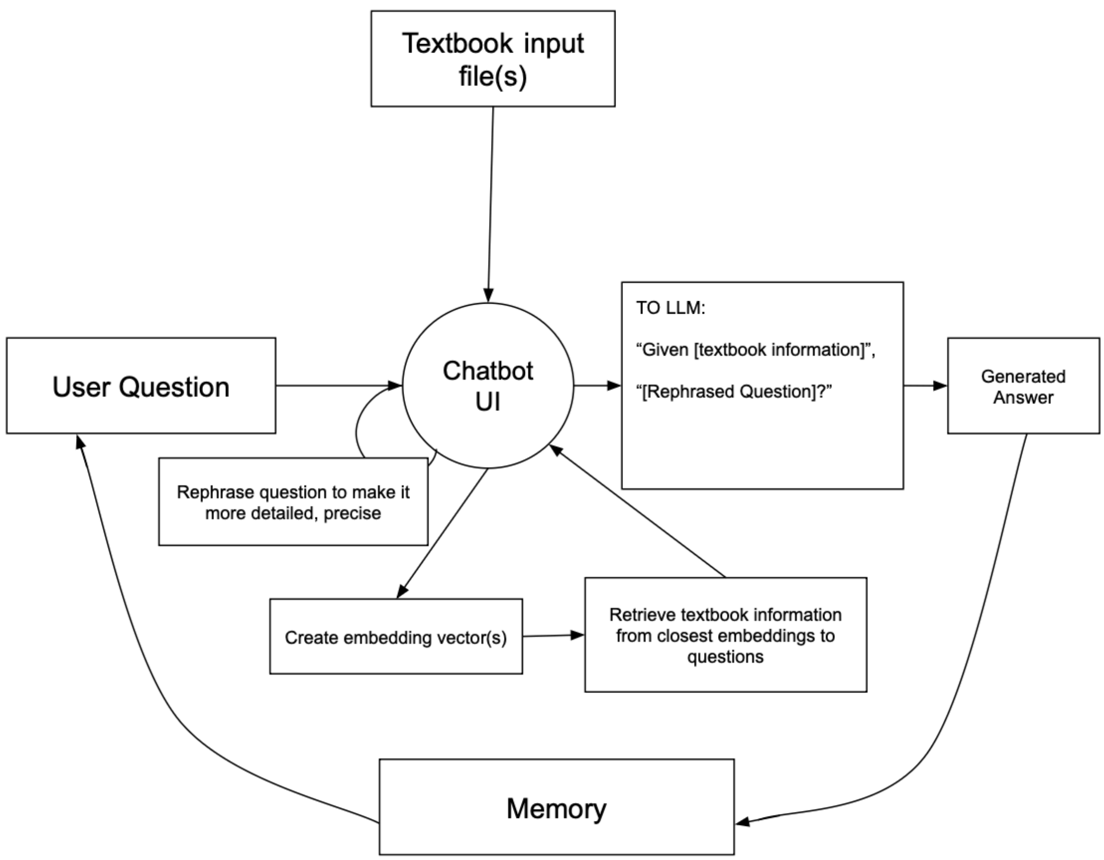

Figure 1. Diagram summarizing our custom RAG-based process based on user input.

Our implementation builds upon traditional RAG by also implementing RaR to make prompts clearer; Additionally, we incorporated flashcard generation due to them being shown to increase student participation and academic performance [6].

Our implementation also allows the user to input any textbook file and supports back-and-forth chat to interact with the chatbot in a way that improves understanding. Our custom RAG process can be summarized in Figure 1, and goes through the following steps:

1. **Extraction**: The framework for the chatbot begins by extracting text from a source of information on a web server. On this server, the user is met with a home page asking the user to input a textbook file, displayed in Figure 6. In our use case, this applies to data coming from a textbook either as a .pdf, image, or audiobook file. The Flask server uses the `POST` method to take the PDF, and allows all files to be added to the "uploads" folder in the server to enable server access.
   a. **PDF files**: Extracted text is split into chunks based on each page. PDFs used the PyMuPDF library on Python. PyMuPDF allows the code to extract text and also keep track of the page while looping through the pages of a PDF file. The data structure used to keep track of the pages and text chunks is a dictionary. Figure 2 displays the page-by-page algorithm of splitting text chunks, while Figure 3 displays the use of dictionaries and its later conversion to .csv files. The `text_tokenized` variable contains all text in each page, and if this is not supported it uses image extraction methods with

Pillow and pytesseract, explained in part (b). During this page-by-page loop, the page count goes up by one as each page number and text chunk is added to the dictionary.

```
book = pymupdf.open(filepath)
global totalpages
for i,page in enumerate(book):
    totalpages+=1
    text_tokenized = page.get_text().split()
    if not text_tokenized:
        pix = page.get_pixmap()
        image = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
        text_tokenized = pytesseract.image_to_string(image, lang='eng').split()
        if not text_tokenized: continue
    pages.append(totalpages)
    texts.append(" ".join(text_tokenized))
```

Figure 2. Algorithm for text extraction.

```
dct = {"page":pages, "text":texts}
df = pd.DataFrame(dct)
```

Figure 3. Setting up data structure of parsed text and locations with a dictionary.

b. **Images**: Much like PDF files, images are also split based on page. The key difference between image and PDF files is that images use the Pillow and pytesseract libraries to extract text that otherwise would be extracted from a PDF file. To enable input of multiple images or PDF files, all of the image files are converted to PDF files using Pillow to enable better compatibility using the code snippet shown in Figure 4. The outputted PDF file is also contained in the “uploads” folder. The server supports use of multiple .pdf or image files. If a PDF file also does not support the extraction of text, it also uses Pillow and pytesseract to extract all text in case pymupdf is unable to parse PDF text.

```
def img_to_pdf(image_paths, out_pdf):
    imgs = [Image.open(p).convert("RGB") for p in image_paths]
    imgs[0].save(out_pdf, save_all=True, append_images=imgs[1:])
```

Figure 4. Code snippet showing the image-to-PDF conversion process.

c. **Audiobooks**: Extracted text is split based on 30-second paragraphs from an .mp3 file using the Deepgram API. Deepgram is capable of extracting text from an audiobook file based on paragraphs and sentences. An example of sentences in the JSON object can be shown in Figure 5, where paragraphs have similar functionality but are much longer. We chose to base our extraction off of paragraphs, because we found sentences to be too short to be any useful both practically and semantically. Deepgram’s JSON object also contains the timestamps of the paragraphs including start and end times, allowing us to use the start timestamp for the user's reference. Since these times were given in seconds, it was necessary to convert in terms of hours, minutes, and seconds for more practical use from the user's point of view. The server supports use of one audiobook file.


```
            "end": 3170.975,
            "num_words": 108
        },
        {
            "sentences": [
                {
                    "text": "This much I understand now.",
                    "start": 3171.7551,
                    "end": 3173.7751
                },
                {
                    "text": "But the man can give no help to the boy, not in this matter nor in those that follow.",
                    "start": 3174.93,
                    "end": 3181.51
                },
                {
                    "text": "The boy moves always out of reach.",
                    "start": 3182.3699,
                    "end": 3185.8298
                },
                {
                    "text": "One afternoon, I walked a friend of mine to his house.",
                    "start": 3187.655,
                    "end": 3191.675
                },
                {
                    "text": "After he went inside, I sat on his steps for a while, then got to my feet and started toward home, wal
king fast.",
                    "start": 3192.535,
                    "end": 3201.55
                }
            ],
            "start": 3171.7551,
            "end": 3201.55,
            "num_words": 65
        },
        {
            "sentences": [
                {
```


Figure 5. Example of Deepgram text JSON, with start/end times for sentences.

When the user submits the file(s), the server will load, going through extraction and embedding conversion, before going to the chat page. This takes roughly 15 seconds for a 1000-page .pdf file, or 20 seconds for a one-hour audio file. This data, storing both the text and the location of the text, is stored into a .csv file, called "texts.csv", by concatenating the text with either the page number or the timestamp. If this .csv file already exists, it is deleted and recreated to make way for user-inputted textbook information even after it was used in a previous session.

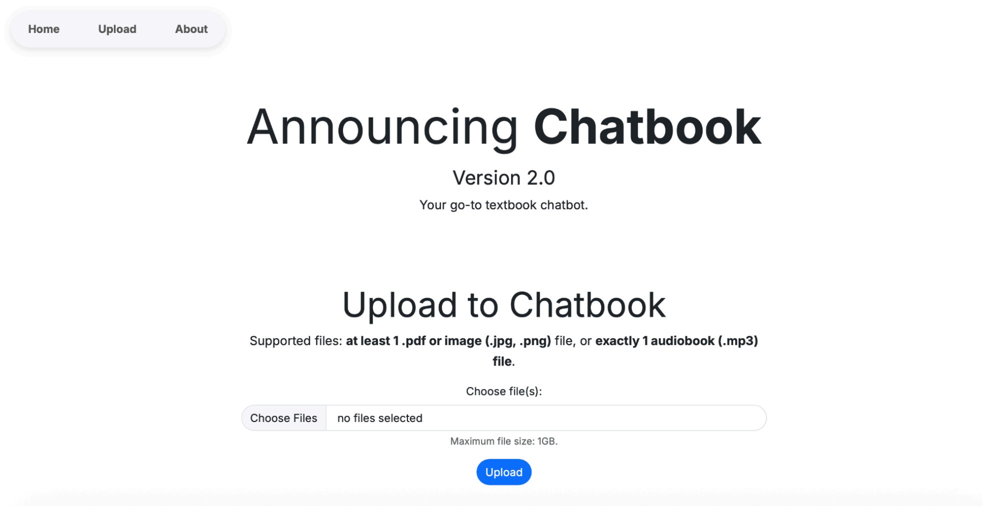

Figure 6. Home page of the web server, where the user inputs textbook file(s).

2. **Embeddings**: After storing the data into a .csv file, each text chunk from the textbook is converted to a 384-dimensional vector embedding. Using pandas once again this time for retrieval of .csv data, each chunk of text is tokenized into its words, inputted into this self-attention model, and outputs an embedding vector, with its magnitude and direction depending on the semantic definition of the words into the text. Similar text will be close together in this vector space, and would be reflected after encoding all text information from the .csv file. In our specific use case, we used a 6-layer pre-trained Sentence-BERT self-attention model, generating a vector space based on semantics. Upon completion of this step, the chat page loads, ready for the user to input a question. The chat page is displayed in Figure 7,

with a side-by-side view of both the textbook file and the chat window for easy usage.

3. **RaR Implementation**: When the user inputs a question, the chatbot rephrases the question into one that is clearer and more detailed using the OpenAI GPT-3.5 model, leaving it less prone to hallucinations by essentially asking itself the question. Still, even after the question is rephrased, the original question remains as part of the prompt because there is a risk of the rephrased question taking away information from what the original question aimed to ask. This is prompted to GPT-3.5 as a "user" prompt, as it is the user aiming to get the question rephrased. The RaR prompt is displayed in Figure 8.

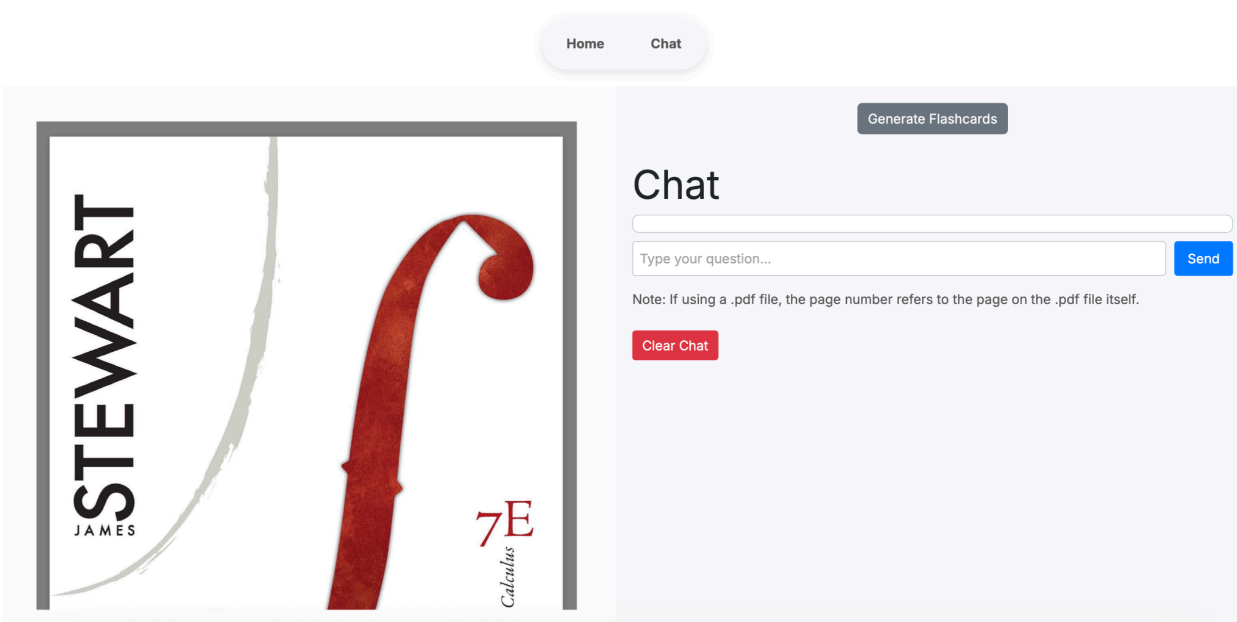

Figure 7. Visual of the chat page after the server processes text chunks and embeddings.

```
{
"role":"user",
"content": f"Rephrase the following question to be more detailed and clearer:\n\n{question}"
}
```

Figure 8. Code snippet displaying GPT-3.5 prompt used to implement RaR.

4. **Retrieval**: This rephrased question is also converted to an embedding of its own using the same SBERT embedding model, and is compared to the 5 nearest embedding vectors in the text using the cosine similarity metric. This is done by taking all of the cosine similarity values and sorting it alongside the respective embeddings. Those 5 nearest embeddings will then be selected as retrieved text for GPT-3.5's prompt. The reason behind using GPT-3.5 is not only because it is one of the most cost-effective large language models commonly used, but it is also a well-researched model that we can more clearly compare with existing literature in evaluating our chatbot.
5. **Prompting**: As this data is extracted, it is also part of the GPT-3.5 prompt to answer the user's question. Included in the prompt includes the following data from the user: original question, rephrased question, retrieved text with its sources cited by the page number or timestamp. Additional prompting information includes prompting the chatbot to use step-by-step reasoning, simple language, clear and detailed explanations, wrapping LaTeX expressions with “$$” for display equations or “$” for inline math. Figure 9 displays an example of a

question being asked, followed by the chatbot's response and citation in square brackets. This is done as part of the "system" role in GPT's API as it is meant to set the behavior of GPT-3.5 when the user asks the question.

6. **Output**: After the server loads the GPT-3.5 generated response, the user interface will display both the user's question and the chatbot's response. To ensure that the chatbot's responses are formatted in a readable format, especially with expressions in subjects including but not limited to Math and Chemistry, the MathJax library is used in JavaScript to convert LaTeX code to readable text. The reason behind using the dollar signs in the previous step was for MathJax to identify LaTeX expressions so that it can correctly process any equations, expressions, or any special characters requiring LaTeX format. The result on the HTML page can be shown with the example of the steps of u-substitution on Figure 9. This is part of the "assistant" role as it is indeed the assistant providing the generated response. At the same time, from the frontend, a series of JavaScript functions are run so that the chat box, treated as a list of chat elements, keeps adding on list items (specifically, back-and-forth interactions between the user and the chatbot) and styles them differently depending on the role contained in each message of the chatbot's history stored in the cookie. Additionally, we used fetch on our Flask server to ensure that the user

stays on the same page while the chatbot information is being processed.

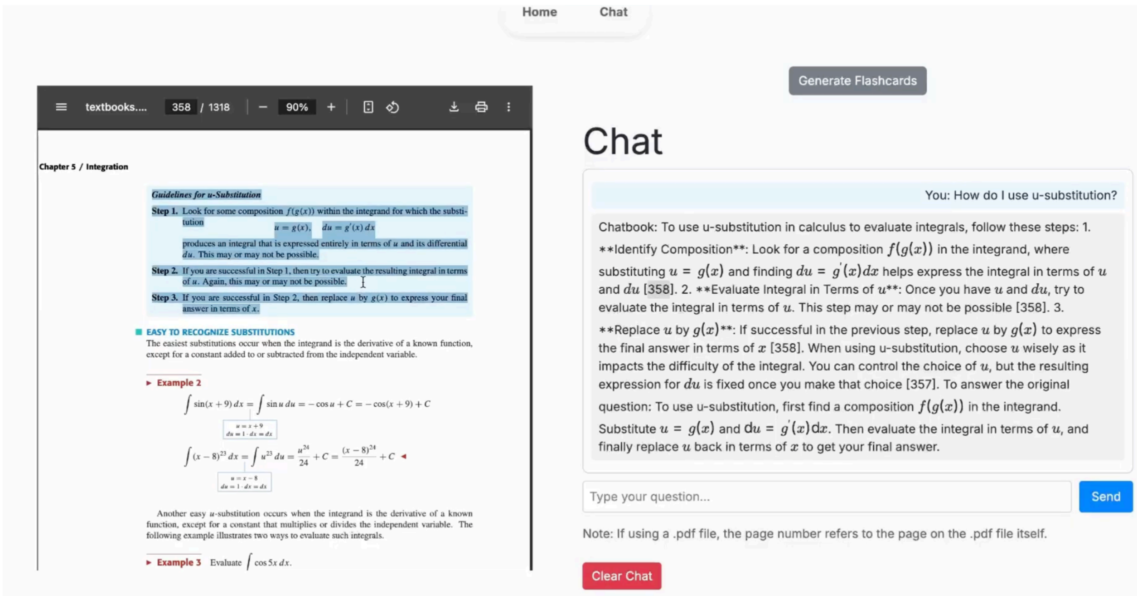

Figure 9. Chatbot output using MathJax with citations after asking the chatbot to explain u-substitution.

7. **Flashcard Generation**: Beyond the RAG framework, this chatbot also utilizes generative flashcards, enhancing recall in the user by quizzing themselves on a topic they choose. This is done by retrieving related information to the inputted topic and generating questions and answers. Figure 10 shows an example of 10 flashcards being generated on the topic of derivatives.

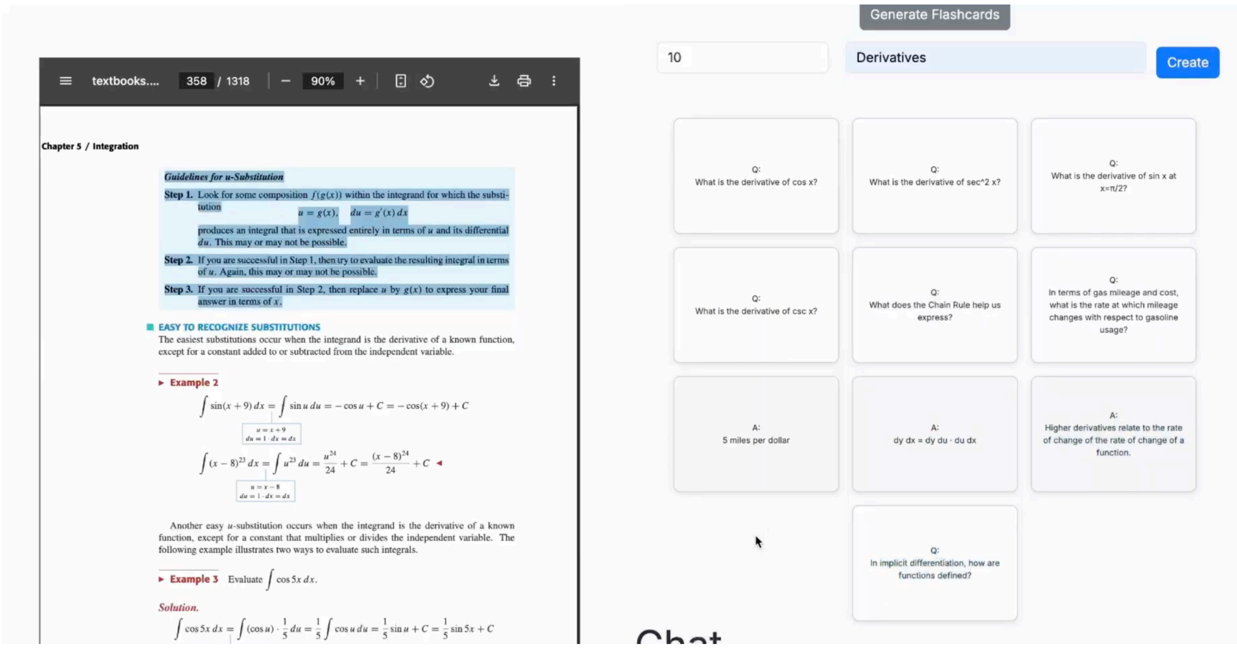

Figure 10. Demonstration of flashcard generation, with front (Q) and back (A) sides displayed upon clicking them.

8. **Memory**: We incorporated back-and-forth memory, allowing the chatbot to accommodate any special requests made by the user, including but not limited to response length, language complexity, and analogies. This was done by creating a cookie in the Flask server which contains all history information of the back-and-forth interaction between chatbot and user, including the system instructions so the chatbot remembers what it is meant to do. Specifically, this cookie is a dictionary with a "history" attribute which contains the entire history of questions and answers. Modifying this is displayed in Figure 11. When a user inputs a new question, the new question and response are simply appended to the existing memory and is stored again into the

cookies once a response is generated. Figure 12 demonstrates an example of back-and-forth chat working with an AP Calculus BC textbook, after asking the chatbot to summarize its original output in two sentences.

```python
history.append({"role":"user", "content": question})
history.append({"role":"assistant", "content": answer})
session["history"] = history
```

Figure 11. Code snippet modifying the Flask server cookie to add chatbot memory.

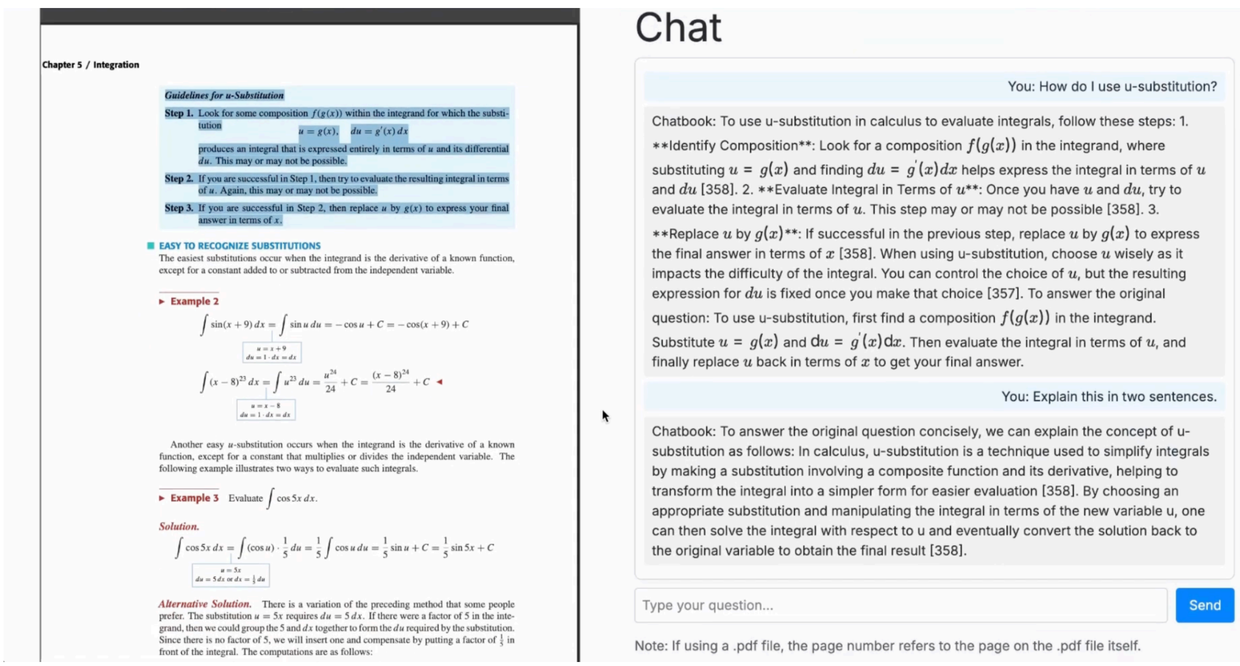

Figure 12. Demonstration of back-and-forth chat, asking the chatbot to follow-up its original response with a brief summary.

Our model was evaluated qualitatively using a combination of metrics that assess its performance in multiple criteria, namely is the information accurate and detailed, does it cite information and does it cite it correctly, and is it able to accommodate to user requests and hold a continuous conversation. The criteria go as follows, with its respective weighting in parentheses based on how important we determined the criteria to be:

1. Is the information accurate? (0.3)
2. Does it account for complexities in the facts? (0.05)
3. Does it cite its sources? (0.1)
4. If it cites, does the source reflect what is being outputted? (0.15)
5. Is it easy to understand? (0.05)
6. Does it provide the right reasoning? (0.1)
7. Does it explain applications well? (0.05)
8. Is it accommodating to special requests by the user? (0.1)
9. Does it recall past chats well enough? (0.1)

We chose to evaluate the model on 6 different subjects: AP Chemistry, AP US History, AP Physics C, AP Calculus BC, AP Computer Science, and AP Government. All of these subjects were tested at the AP level, and for AP Government we used an audiobook to test the chatbot's performance in a different format. For each subject, we asked the chatbot 10 questions, each in a different conversation. After this question, we asked a relevant follow up question, which was used to evaluate the model's conversational ability.

Results & Discussion

Results of the chatbot's performance can be summarized in Table 1. The chatbot achieved a strong overall performance with an average normalized score of 0.906 across all subjects. The highest-scoring subjects were U.S. History (0.952 normalized), Chemistry (0.921 normalized), and Government (0.903 normalized), indicating that the chatbot excels in subjects where discrete factual recall and empirical reasoning dominate. These subjects typically feature less repetitive scaffolding between concepts, allowing the RAG framework to extract and cite localized information more effectively.

Topic	Accuracy (/0.3)	Complexity (/0.05)	Citations Included (/0.1)	Correct Citation (/0.15)	Easy to Understand (/0.05)	Reasoning (/0.1)	Applications (/0.05)	Sp. Requests (/0.1)	Memory (/0.1)	Total
Chemistry	0.28	0.05	0.098	0.14	0.041	0.1	0.012	0.1	0.1	**0.921**
History	0.3	0.05	0.09	0.125	0.047	0.095	0.05	0.095	0.1	**0.952**
Physics	0.3	0.05	0.07	0.105	0.05	0.1	0.05	0.1	0.1	**0.895**
Math	0.3	0.05	0.07	0.095	0.05	0.095	0.032	0.095	0.09	**0.877**
Comp Sci	0.29	0.05	0.07	0.082	0.046	0.1	0.05	0.1	0.1	**0.888**
Gov Audio	0.28	0.04	0.1	0.138	0.05	0.1	0.045	0.08	0.07	**0.903**
Average	**0.292**	**0.048**	**0.083**	**0.114**	**0.047**	**0.098**	**0.04**	**0.095**	**0.093**	**0.906**

Table 1. Evaluation scores across subjects (/category weights).

In contrast, performance dipped slightly in Math (0.877 normalized) and Computer Science (0.888 normalized). These subjects often involve more interdependent knowledge structures, i.e. where theorems build on prior proofs or code requires multi-line context. This complexity posed challenges for accurate citation and chunk-based semantic retrieval. Specifically, CS suffered from a noticeable drop in correct citation scores, which we attribute to GPT-3.5's default bias toward code synthesis rather than source-grounded explanations.

The overall citation inclusion scored 0.083/0.1, while correct citation accuracy reached 0.114/0.15. These are substantial improvements over traditional GPT-3.5 performance. As mentioned previously, GPT-3.5 fabricated citations in over 50% of cases [8]; in contrast, our RAG-based method achieved a citation correctness score of 0.76 (normalized by weight) when citations were present. What was especially impressive about our result, though, is that we also had a citation presence score of 0.838 when normalized, demonstrating that there is a higher likelihood of correct citations relative to any citation being present.

The Government audiobook case revealed unique trends. Despite using less granular chunking (30-second intervals), it maintained a high total score. We believe that this may be from a smaller semantic space: because the source content is more limited, retrieved passages are semantically closer to queries by default. However, this advantage came at the expense of memory performance, particularly for longer conversations where timestamped context became less precise.

Conclusion

Our chatbot demonstrates strong performance when it comes to meeting our 9 criteria, especially when it comes to generating accurate information that is easy to understand and provides good reasoning. It also proved to be strong in citing its sources, a key feature. However, we noticed some limitations, especially when it came to retrieving the right information in audiobooks, as well as citing the right sources in material that involved code, formula generation, and interdependent knowledge.

In the future, studies could involve more optimized methods of chunk splitting from the original source, especially with textbook PDFs, that split based on topic rather than by page, to differentiate the various embeddings as much as possible. Additionally, future studies include using newer and more cost efficient large language models than the models of our time and using a direct comparison of our RAG implementation performance with theirs.

Reproducibility

All code used for the development of our chatbot, including instructions for usage of the chatbot, is publicly available in the following Google Drive folder:

https://tinyurl.com/chatbook-tjhsst. See the README.md file for more information on how to use the web server.

Acknowledgements

We thank Dr. Gabor and Dr. Torbert for their guidance through the TJHSST Computer Systems Lab.

References

1. Alawwad, H. A., Alhothali, A., Naseem, U., Alkhathlan, A., & Jamal, A. (2025). Enhancing textual textbook question answering with large language models and retrieval augmented generation. *Pattern Recognition*, *162*, 111332. https://doi.org/10.1016/j.patcog.2024.111332
2. Deng, Y., Zhang, W., Chen, Z., & Gu, Q. (2023). Rephrase and respond: Let large language models ask better questions for themselves. arXiv preprint arXiv:2311.04205.
3. Levonian, Z., Li, C., Zhu, W., Gade, A., Henkel, O., Postle, M. E., & Xing, W. (2023). Retrieval-augmented generation to improve math question-answering: Trade-offs between groundedness and human preference. arXiv preprint arXiv:2310.03184.
4. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., … Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), Advances in Neural Information Processing Systems (Vol. 33, pp. 9459–9474). Retrieved from https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
5. McClain, C. (2024, March 26). *Americans' use of Chatgpt is ticking up, but few trust its election information*. Pew Research Center.

https://www.pewresearch.org/short-reads/2024/03/26/americans-use-of-chatgpt-is-ticking-up-but-few-trust-its-election-information/

6. Nugroho, Y. S., Nurkamto, J., & Sulistyowati, H. (2012). IMPROVING STUDENTS' VOCABULARY MASTERY USING FLASHCARDS. English Education, 1(1).
7. Singer, M. B., Fu, J. J., Chow, J., & Teng, C. C. (2023). Development and Evaluation of Aeyeconsult: A Novel Ophthalmology Chatbot Leveraging Verified Textbook Knowledge and GPT-4. *Journal of Surgical Education.* https://doi.org/10.1016/j.jsurg.2023.11.019
8. Walters, W. H., & Wilder, E. I. (2023). Fabrication and errors in the bibliographic citations generated by chatgpt. *Scientific Reports*, *13*(1). https://doi.org/10.1038/s41598-023-41032-5