Disdoc: AI Teaching Assistant for Computer Science Courses

Brendan R. Doney

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Godmar V. Back, Chair
Margaret O. Ellis
Sally Hamouda

February 17, 2024
Blacksburg, Virginia

Keywords: Large Language Models, Generative Artificial Intelligence, AI Teaching Assistant, Computer Science Education

Disdoc: AI Teaching Assistant for Computer Science Courses

Brendan R. Doney

(ABSTRACT)

As enrollment in Computer Science grows, traditional help-seeking opportunities for students, such as office hours and forums, become less effective due to rising student-to-teaching assistant ratios. To address this issue, research has investigated large language models (LLMs) to provide individualized help to students at scale. However, prior research primarily targets introductory computing courses, does not fully connect LLMs to course material, and does not expose relevant course material to students. As a result, existing approaches do not adapt well to advanced computing courses and limit opportunities for students to develop self-sufficiency.

To address this, we present Disdoc, an LLM-based question and answer tool for students in advanced computing courses. Disdoc presents snippets of course material relevant to student questions and generates answers using an LLM. To include course-specific information in answers, we connect the LLM to all course material through retrieval-augmented generation (RAG). To ensure the RAG system retrieves the most relevant information, we organize course material into question categories.

We evaluated Disdoc in a research study on a 340-student Computer Systems class at Virginia Tech, where we tracked student reviews, activity, and exit survey responses. Students indicated that Disdoc was helpful, particularly for questions about course assignments. Usage data revealed that students strongly preferred to see LLM-generated answers and rarely clicked on outgoing links, suggesting they were satisfied with the LLM-generated answers and snippets of relevant course material.

Disdoc: AI Teaching Assistant for Computer Science Courses

Brendan R. Doney

(GENERAL AUDIENCE ABSTRACT)

As enrollment in Computer Science grows, traditional ways for students to seek help, such as office hours and forums, become less effective due to rising student-to-teaching assistant ratios. To address this issue, research has investigated large language models (LLMs), a type of AI that generates responses to text-based prompts. Past research has used LLMs to provide individualized help to students, but has primarily focused on introductory computing courses and has not fully integrated course-specific materials. As a result, existing approaches do not adapt well to advanced computing courses and limit opportunities for students to develop self-sufficiency.

To address this, we present Disdoc, an LLM-based question and answer tool for students in advanced computing courses. Disdoc presents snippets of course material relevant to student questions and generates answers using an LLM. To incorporate course-specific information in generated answers, the LLM references course materials through a process called retrieval-augmented generation (RAG). To ensure the RAG system retrieves the most relevant information, we organize course material into question categories.

We evaluated Disdoc in a research study on a 340-student Computer Systems class at Virginia Tech, where we tracked student reviews, activity, and exit survey responses. Students indicated that Disdoc was helpful, particularly for questions about course assignments. Usage data revealed that students strongly preferred to see LLM-generated answers and rarely clicked on outgoing links, suggesting they were satisfied with the LLM-generated answers and snippets of relevant course material.

Contents

List of Figures

List of Tables

List of Abbreviations

LLM Large Language Model

RAG Retrieval Augmented Generation

TA Teaching Assistant

Large Language Models are a type of deep learning model, based on the transformer architecture, trained to understand and generate text.

Retrieval Augmented Generation is a technique to improve LLM output by including domain-specific knowledge in the prompt that is relevant to the user's original prompt. It is detailed more in Section 2.2.

Teaching assistants help instructors teach their course. A teaching assistant can be an undergraduate (UTA) or graduate (GTA) student. While responsibilities vary, they typically hosting office hours and grade assignments.

Chapter 1

Introduction

Historically, office hours has been the primary help-seeking resource for students in Computer Science classes. Unfortunately, a suite of problems makes satisfying student demand for quality office hours difficult. This leaves some students without a key help-seeking resource, forcing them to turn to others or stranding them in the worst case. Simultaneously, in the digital education era, the breadth of course resources available to students in computing classes has grown so large that students cannot expect to have perfect recall or awareness over every resource that is available to them. This leaves a gap between instructors and students: while instructors publish resources to address common problems and lower demand for office hours, students are not aware of or do not remember them. Fortunately, with the use of large language models (LLMs) and retrieval augmented generation (RAG), it is now possible to create information retrieval and question and answer systems that have recall over every document in a course. So far, research has targeted only introductory computing courses and does not fully connect LLMs and students to relevant resources. As a result, existing approaches do not adapt well to advanced computing courses and limit opportunities for students to develop self-sufficiency. Given these issues, we investigate how to best use LLMs and RAG with CS3214: Computer Systems, an advanced computing class at Virginia Tech taken primarily by senior-level undergraduate students.

1.1 Traditional Help Resources

Some combination of teaching assistants (TAs), office hours, and asynchronous question and answer tools has been the primary means of help-seeking in most computing resources. But varying TA quality, extremely high student demand, and long queues mean that they do not always provide the best experience for students.

First, human teaching assistants (TAs) can be a source of poor help-seeking experiences. The level and quality of training for TAs varies greatly across universities [16]. Even when it is present, TA training often lacks training on pedagogy, instead focusing on administrative procedures [16, 53]. This low level or outright omission of training leads to TAs who are ill-equipped to teach students. Furthermore, there is evidence that even when there is TA training, it does not prevent lower self-efficacy among first-time teachers [10]. Unfortunately, with TA retention as low as 34% in some universities [31], a large number of TAs are likely to be first time teachers. This problem only compounds as growing CS enrollments [8] necessitate a larger number of potentially inexperienced and under-trained TAs.

Another large source of distress is the structure of office hours itself. In a typical office hours structure, students enter a queue and meet individually with a teaching assistant once selected. This model breaks down if there are too many more students than TAs in office hours or if TAs spend too long with individual students [73], increasing average queue lengths and wait times. In advanced classes with large enrollment sizes, the student:TA ratio is further hurt by difficulties finding TAs: the specificity of course material lowers the number of qualified TAs. While alternative TA and office hours structures [2, 9, 30], optimal queuing strategies [28], and grouping repeated problems [38, 55] can help address these issues, they cannot completely prevent long queues.

Unfortunately, many students leave long queues without being helped [27, 29, 73], hindering

student progress and causing frustration. Shortly before deadlines, queues typically get worse [47, 55], exacerbating the issue when students do not have time to spare. In Spring 2024, the queue for CS3214 reached a maximum of 111 on the due date of a major project, taking a minimum of 9 hours and 15 minutes to empty if 3 TAs took 15 minutes each per student. Even students who stay in queues make little measurable progress while waiting [27].

Lastly, office hours can unequally benefit students. Most pressing is the issue of office hours monopolization, where the top 5% of students use roughly 50% of office hours time [73]. Similarly, some students misuse time in office hours, in what some have called help abuse [59]. In CS3214, unneeded help is a particularly common form of help abuse. This often occurs because the unmanageably high queues lead students to enqueue in the morning even before they encounter an issue - with the original intention of dequeuing if they do not have an issue right before they are selected - and then attend office hours without a well-formulated question anyway. Research also shows that traditional, in-person office hours may be underutilized by students with low confidence [26]. Regardless of the cause, all of these behaviors are at the expense of students who need help.

As an alternative to office hours, research has experimented with asynchronous question and answer tools like Discord [5], Piazza [79], and other online forums [58, 83, 87]. Students show positive response to such tools [5, 87] and research has found a positive correlation between the usage of such tools and course grades [83]. Importantly, their asynchronous nature means that these tools can still service students when demand is extremely high (although with a delay), unlike office hours. Unfortunately, these tools are still reliant on teaching assistant and instructor activity to help the majority of students [79]. As a result, while asynchronous question and answer tools can help service students when office hours fall short, they still degrade under poor student:TA ratios. In the end, we need a new type of help-seeking resource robust to this issue.

1.2 Discoverability

As instructors progress through course material, students often encounter difficulties and seek help through available channels. From exchanges in these help-seeking environments, common issues emerge, prompting instructors to create and public additional material. However, as the students' need for help persists, the volume of published content can grow substantially. Without careful management, this expansion can create significant problems with discoverability and recall for students.

For example, in CS3214, we counted 375 050 words/tokens (words for prose, whitespace-separated tokens for code) across 479 distinct files provided to students throughout the course. Excluding the final exams students can optionally review, this is 216 124 words/-tokens across 434 files. For clarity, we group these files into administrative materials (e.g. syllabus), slides and source code presented in lectures, specifications for exercise and projects students complete, starter code provided to students for projects and exercises, and slides presented in help sessions for projects. The complete breakdown is shown in Table 1.1.

Unsurprisingly, CS3214 students do not demonstrate perfect recall over all 216 124 words of content when they attend office hours. For instance, many of the students who have not searched the assignment-specific FAQ pages before attending office hours say that they were not aware of the FAQ pages at all. Additionally, when attending office hours for questions about multi-threading, many students have forgotten the code examples that show how to implement common multi-threading patterns. Unfortunately, while this issue is clear in CS3214, it is difficult to determine how common it is elsewhere due to the lack of sharing of course materials in computing education [4, 21, 70].

For some students, this poor recall over course material is connected to their working habits. Namely, research has found that computing students often leave assignments until the last

Category	Words/Tokens	Files
Administrative	7 916	11
Lecture Slides	42 648	55
Lecture Code Examples	13 331	154
Exercise Specifications	12 735	28
Project Specifications	34 761	26
Starter Code	99 466	157
Help Session Slides	5 267	3
Total	216 124	434

Table 1.1: Breakdown of the word (for prose) or token (for code) count and file count for each category of resource in CS3214.

minute [22, 23] and many prefer direct solutions to their problems over guidance as deadlines approach [51]. So, under the pressure of deadlines, some students prioritize the guaranteed answer they will get in office hours over an answer they may or may not find from a thorough search of course materials. Furthermore, timeline pressure may contribute to incomplete readings of longer course materials, such as textbooks or assignment specifications.

Nevertheless, relying on the traditional help-seeking resources to correct such cases is not ideal. Regardless of the cause, students who use these resources with a question that is answered by course material make poor use of their time and degrade the resource for other students by misusing the limited time of instructors and TAs. Additionally, such questions can be repeated by multiple students independently, making office hours even less optimal. For such questions, the goal is to show students the answer or where to find it without them attending office hours at all.

Another way to address the issue is to publish less course material, but this is also unfavorable. For a course such as CS3214, where there is a large body of highly technical material and student issues to cover, a large body of course material is necessary; compacting material for such courses is extremely difficult and will require the removal of some essential information. Even material that is not intended for every student (e.g. an FAQ page) has

the potential to greatly help the subset of help-seeking students it was written for, if they are aware of it. As such, this is a discoverability problem: students need a way to easily find information relevant to their questions from a growing quantity of course material.

1.3 Large Language Model Assistance

Based on the previous sections, we have two goals for our tool: (1) it should help students discover course material relevant to their questions and (2) it should be robust to poor student:TA ratios. For a partial solution, we first turn to Large Language Models (LLMs).

A recent development in the natural language processing field, LLMs are a form of artificial intelligence that generates text in response to prompts. For question and answer applications, where LLMs generate answers to questions listed in the prompt, the key strength of LLMs is the ability to synthesize information learned from training and from the prompt in generated answers. This means that LLMs can intelligently answer student questions at a scale and speed unattainable to traditional help seeking resources. Further, because of high parameter counts and large training corpora, the LLM's base knowledge alone often suffices for simple question and answer applications. As a result, most simple question and answer applications can use an LLM to generate high quality answers with minimal or even no work.

However, there are two key issues that can limit the quality of LLM-generated answers in more complex questions and answer environments. First, LLMs cannot include information in answers that they were not provided via training or prompts. For example, they cannot discuss assignment deadlines for a particular course without additional measures. Second, LLMs occasionally hallucinate, where they generate nonsensical text or text containing incorrect information [41]. In some cases this is due to random chance, but it happens more often when LLMs are asked about information that was not contained in their training data

(due to recency or obscurity, for example). Nevertheless, LLMs enable an unparalleled level of personalized one-on-one help for Computer Science students, which addresses our second goal.

Acknowledging the potential of LLMs in providing students on-demand assistance without requiring staff, a great deal of research has explored the use of LLMs to help Computer Science students when they encounter issues in their work or understanding. Most research uses commercial LLMs, either through a custom question and answer interface [46, 50, 52] or through integrations with tools students are using [39, 77]. Unfortunately, there is limited research on fine-tuning LLMs specifically for Computer Science education [36]. Student surveys and usage indicate the value of such systems [17]. Also, to our knowledge, there is little research on using LLMs in advanced computing courses (CS3 and beyond) [46] and no research on connecting LLMs to all course content in advanced courses. Rather, most research has focused on introductory courses like CS1 and CS2. However, advanced courses still serve to benefit from LLM-based help tools. However, existing solutions targeting introductory courses are very difficult or impossible to adapt for advanced courses because of key differences.

First, there is a mismatch in raw LLM capability due to differences in course content. The majority of content from introductory computer science courses is already covered by most LLMs' training data. This means that the LLM is already capable, barring hallucinations, of answering most of the questions from a CS1 or CS2 student without any additional techniques. The same is not true of higher-level computer science classes. Rather, higher-level courses can cover concepts that are not adequately covered by an LLM's training set and may use software libraries that the LLM has little to no knowledge of. Given LLMs' known inability to generate answers including information it was not trained on or provided and their propensity to hallucinate in such scenarios, this means the quality of raw LLM-

generated answers are likely to be lower for higher-level courses.

Second, existing strategies for capturing the context behind questions, a necessary step for the LLM to generate an appropriate answer, do not scale for higher-level courses. Specifically, most rely on including the student's entire source code [39, 46, 50, 52, 77] and potentially the assignment's ground-truth solution [52] in the LLM's prompt. For higher-level courses, this may be infeasible due to context size constraints. For example, the unit tests for CS3214's web server project are 3434 lines of Python code. Even when the code does fit in the prompt, the longer prompt may lead to worse output and higher costs [43, 88] .

In addition to these adaptability issues, existing solutions do not connect LLMs, students, or both to all course material. If the LLM is not connected to all course material, it cannot accurately discuss course-specific content in its answers, becoming unreliable for a great deal of student questions. If students are not connected to relevant material, for example if an LLM generates an answer with course-specific content without disclosing its sources, we rob students of the chance to support themselves in the future and potentially foster over-reliance on the LLM. But this is the opposite of student preference, as students report they want LLM-based tools to help them develop problem-solving skills over giving answers outright, although they do not always act in line with this preference [17]. Additionally, exposing both LLMs and students to relevant information can help students identify hallucinations and provide further readings. As a result, we strongly believe LLM-based tools should connect LLMs to course material *and* expose relevant course material to students.

In summary, when used with higher-level courses, current solutions can generate lower-quality answers than with introductory courses, are not scalable in how they capture the context behind a question, and do not fully connect LLMs and students to the course material available to them. As a result, current solutions do not fully address our first goal, connecting students to relevant course material.

To fully address both of our goals and these issues with existing solutions, we developed Disdoc, a tool that connects both LLMs and students to snippets from a course's material that are relevant to the student's question. As a result, Disdoc helps students discover relevant course material (in line with our first goal) and generates answers to questions which include course-specific information like deadlines, covered content, libraries, and assignment specifics (in line with our second goal).

To accomplish this, Disdoc leverages retrieval-augmented generation (RAG) [33, 49], a technique that retrieves information from a knowledge database relevant to a user's question and supplements the LLM with this information. In a traditional RAG system, a retriever obtains text snippets relevant to a user's question (according to some similarity metric) from a previously collected corpus. These retrieved snippets then augment the user's question in the final prompt to the LLM, potentially giving the LLM information for use in generation that was not included in training. Commonly, RAG systems will convert text snippets into a vector form using an embedding model and measure relevance using a vector similarity metric. Thus, the vector and text snippet pairs from the corpus are stored in a vector database and when a question is asked, these vectors are measured against the vector form of the user's question to find relevant snippets [25]. Concurrent with our work, others have found that RAG decreases hallucinations across disciplines [56, 72, 80] and even increases the accuracy and relevance of LLM-generated answers in Computer Science education [36, 52].

In Disdoc, we use RAG with a corpus of snippets from course documents to supply the LLM with course-specific information that it would otherwise not have. For example, if a student asks why they are failing a unit test, we retrieve and include the source code for that specific unit test in the prompt. However, this traditional RAG system has a problem: when a student asks about a particular assignment, we may retrieve irrelevant snippets from other assignments. To address this, Disdoc categorizes materials either by their assignment

or as lecture material, administrative material, or sample exams. Then, when a student asks a question, they provide a category, which Disdoc uses to filter its retrieved chunks. This lowers the incidence of irrelevant sources, particularly as the course corpus grows.

Altogether, this strategy enables students to ask about and receive answers with course-specific material, accomplishing our second goal. Then, to accomplish our first goal, Disdoc also exposes snippets from the retrieval step to students. Lastly, to accommodate students' varying preferences towards LLMs, we display these retrieved snippets separately from the LLM-generated answer and make the answer optional altogether.

1.4 Contributions Made

In order to help satisfy student help-seeking demand, we created and evaluated an LLM and RAG-based tool that serves our two goals: (1) to provide a question and answer tool robust to poor student:TA ratio and (2) to help students discover course material relevant to their questions.

We make the following contributions:

- **Design of Disdoc**: We present Disdoc, a tool for connecting students to relevant snippets from course material and generating answers which include course-specific information in advanced Computer Science courses. We specifically design for easy adoption in other courses, interoperability with different LLMs, and detailed data collection.
- **An open-source implementation**: We fully open-source each component of Disdoc to facilitate use in other courses. This includes the web scraper[1], data collection API

[1] https://github.com/brdoney/cs3214-scraper

and frontend[2], RAG pipeline and Discord bot[3], and data analysis code[4].

- **Deployment**: We deployed Disdoc in CS3214, a large-scale, advanced operating systems course. This is among the last courses required for Computer Science majors at Virginia Tech, taken primarily by seniors in Computer Science.

- **Evaluation of the tool**: During deployment, we collected information on the tool's generated responses, including retrieved resources and student ratings of correctness, relevance, and helpfulness. With this data, we evaluated the tool's performance, identified trends in recommendations, and discuss whether the tool has a learning curve.

- **Analysis of student usage**: We analyzed the student usage data collected during deployment, revealing student preferences and patterns. We found a strong preference for LLM-generated answers and potentially low engagement with links to retrieved resources.

- **Survey of user base**: Following deployment, students were surveyed about their experience with Disdoc and similar LLM-based tools. Respondents were generally positive and the majority would use Disdoc more in the future, particularly for questions about course assignments.

1.5 Thesis Roadmap

In Chapter 2, we discuss the background required to understand this work. Then, we discuss the design and implementation of Disdoc in Chapter 3. We then evaluate the data collected from student interactions during deployment in CS3214 and a survey, reporting the results

[2] https://github.com/brdoney/thesis-api
[3] https://github.com/brdoney/disdoc
[4] https://github.com/brdoney/thesis-analysis

in Chapter 4. In Chapter 5, we discuss research related to this work before discussing future work in Chapter 6. Finally, we conclude our research in Chapter 7.

Chapter 2

Background

2.1 Large Language Models

Large language models stem from pre-trained language models (PLMs), a family of neural networks trained on large corpora typically scraped from the Internet [84]. They are trained to be general purpose, rather than fit for a specific language modeling task (e.g. classification or text completion). As a result, to use a PLM for a specific task, a pre-trained model is fine-tuned for the task using a task-specific dataset (for example, a dataset of text snippets and their classification) [14, 18]. This process is based on transfer learning, which supposes that we can transfer some amount of knowledge shared between tasks to lower training time and costs.

While various architectures exist, many PLMs use the transformer architecture [82]. This architecture gives models the cross-attention mechanism, which enables a model to look at other words to understand a given one (i.e. understand a word in context), while staying highly scalable using parallel programming on GPUs. BERT [18] and the early GPT family of models [64, 65] are prominent transformer-based PLMs.

Importantly, research shows that scaling language models by model size, training corpus size, or computational resources yields better performance in language modeling tasks [44]. This scaling law led to vastly larger language models, such as GPT-3 (175B parameters) [6], PaLM

(540B parameters) [11], and Llama 2 (up to 70B parameters) [81], termed large language models (LLMs). While multimodal language models which accept other forms of input exist, we will be focusing on LLMs whose prompts and responses are entirely text in this research.

Similar to PLMs, LLMs can be fine-tuned. However, because of characteristics that emerge only in LLMs, this is not always necessary. In particular, LLMs exhibit the capability to generate desirable output when given a few examples in their prompt, a process called few-shot learning [86]. Furthermore, while base LLMs have no expected prompt format, LLMs fine-tuned with a template in their prompts show better performance when asked to follow instructions without any examples at all [68, 85]. In other words, LLMs fine-tuned to follow instructions can reasonably serve as general purpose models.

As such, most modern LLMs are instruction-tuned models that expect a particular prompt. For example, Llama 2's prompt format can be seen in Listing 2.1. It includes a section for an optional system prompt (intended to instruct the LLM on its role), a section for a user's request, and various special tokens to differentiate each section.

Listing 2.1: Llama 2's prompt template.

```
<s>[INST] <<SYS>>
{{ system_prompt }}
<</SYS>>

{{ user_message }} [/INST]
```

Another side effect of the large number of parameters in LLMs is a large parametric memory: LLMs can memorize a great deal information from training in its many billions of parameters and use it later. As a result, any task that is sufficiently covered by an LLM's training data is likely to be included by an LLM's parametric memory.

Unfortunately, even facts included in an LLM's parametric memory can be misreported due to hallucinations [41]. When hallucinating, the LLM can generate nonsensical text or text that contradicts its prompt or parametric knowledge. For question and answer applications like this research, this means the LLM can output an incorrect answer despite knowing the correct one. Furthermore, LLMs answer questions with confidence unless instructed otherwise, leading to confidently incorrect answers that can be particularly confusing for the user.

LLMs understand natural language on the token level. Tokens represent the smallest unit of language in an LLM, where the finite set of all possible tokens is specific to each LLM. The LLM's tokenizer is responsible for converting sentences into sequences of tokens, namely each prompt the LLM is given. Common strategies for tokenization have emerged, particularly byte-pair encoding (proposed by Gage [24] and adapted for natural language by Sennrich et al. [69]) which is common among the GPT-3.5 [6] and Llama 2 [81] models used in this paper.

Due to the size and computational resources required to run LLMs, they are often provided in the cloud and access is provided via an API. The developer who uses the model is then billed based on the number of tokens in the prompt and the number of tokens generated. OpenAI's GPT family of models are a salient example. On the other hand, there are also open-weight and open-source LLMs, such as Meta's Llama family of models [1, 81], which are free to download, fine-tune, and deploy. Various runtimes can be used to self-host such LLMs, prominently HuggingFace's transformers[1] and Gerganov's llama.cpp[2]. Notably, while these open models are flexible, they still require expensive clusters of GPUs to run with reasonable speeds.

[1] https://github.com/huggingface/transformers
[2] https://github.com/ggerganov/llama.cpp

2.2 Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is a technique that expands the knowledge available to an LLM beyond its parametric knowledge by giving it access to external knowledge sources. It shows the capability to update an LLM's knowledge after training has finished [49], reduce hallucinations across disciplines [56, 72, 80] and increase accuracy and relevance in Computer Science education [36].

In a RAG system, a retriever obtains text snippets that are relevant to a user's question [33, 49]. To achieve this, a corpus of text is collected, typically via web scraping, then split into manageable chunks and indexed by their vector form, called an embedding. The exact embedding for each text chunk is obtained via a chosen embedding technique. When a user asks a question, the system performs a similarity search on the question's embedding using the index of chunk embeddings and a chosen similarity metric: commonly Euclidean distance $\|\vec{a} - \vec{b}\|$, dot-product similarity $\vec{a} \cdot \vec{b}$, or cosine similarity $\frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$. This search yields the text chunks most similar to the question. These retrieved chunks then augment the question in a complete prompt to provide the LLM access to the retrieved chunks when answering the question. Importantly, while not part of a traditional RAG system, the retrieved chunks can be used for more than augmentation. For instance, they can be displayed to the user as resources relevant to their question or displayed alongside the LLM's answer as citations.

For simple RAG applications, primitive embedding techniques like bag of words or term frequency-inverse document frequency (TF-IDF) may suffice. Specifically, bag of words counts the frequency of words in each document to populate the values in vectors [63] and TF-IDF combines each word's count in a document and the inverse of the word's frequency in the entire corpus to emphasize words that are significant and ignore those that are common (e.g. "as," "the," "I") [67]. Notably, these techniques do not ensure that semantically similar words

will have embeddings that are close together in the space of all possible embeddings [60]. However, for more complex RAG systems, such a property is desirable because it ensures that the retriever's vector similarity metric describes *semantic* similarity, leading it to retrieve semantically similar documents.

To ensure that semantically similar words are close together in the embedding space, neural network-based models, called embedding models, are used. While better at capturing semantic similarity than primitive embedding techniques, simple embedding models like Word2Vec [60] do not fully capture the contextual meaning of words, as individual words can only have one vector. More concretely, "bank" would produce the same embedding in "river bank" and "bank deposit," despite having a different meaning in each.

To address this issue, transformer-based [82] models are used. In the transformer architecture, a word's meaning is attended by its surrounding words, leading to a distinct vector for each contextual meaning of a word. BERT [18] and its variants are a popular base for these embedding models, leading to collections of fine-tuned BERT variants like Sentence Transformers [66]. These transformer-based embedding models are the most widely used embedding technique in current RAG systems [25].

The exact selection of embedding model and similarity metric are specific to each application. For example, because dot-product similarity is influenced by the magnitude of embeddings, it is a poor match for RAG systems where chunks vary greatly in length. Similarly, using a different similarity metric than an embedding model was trained on can lower performance [75]. Additionally, some RAG systems achieve better performance using metadata to filter out irrelevant chunks or using query optimization, where a user's question is reworded to improve suitability for retrieval before being given to the retriever [25].

Similarly, how to augment the user's question with the retrieved chunks is an open problem.

The simplest approach is to simply place the contents of the chunks in a prompt before the question, but more advanced approaches include re-ranking retrieved chunks, compressing the chunks to their essential information, and filtering irrelevant chunks [25].

In order to persist the index of chunk embeddings, a vector database can be used. Vector databases manage the storage of embeddings and implement an approximate nearest neighbors algorithm to efficiently find the chunks most similar to a given embedding. Common approximate nearest neighbors algorithms are hierarchical navigation of small words (HNSW) [57], product quantization [42], and locality-sensitive hashing [40]. Recent research shows that HNSW performs the best of current solutions [3]. Some vector databases also allow metadata to be stored along with embeddings and can be used to filter the candidates of a similarity search.

There exist two main types of vector databases: dedicated and adapted databases. Dedicated vector databases, including Chroma [12], Qdrant[3], Weaviate[4], and Pinecone[5], are built from scratch for vector search applications. On the other hand, adapted vector databases take a preexisting database technology and adapt it for vector search, such as pgvector[6]. For applications that do not need persistence or manage it themselves, there are dedicated in-memory vector indices like FAISS[7], but many dedicated vector databases also offer an in-memory mode.

[3] https://qdrant.tech/
[4] https://weaviate.io/
[5] https://www.pinecone.io/
[6] https://github.com/pgvector/pgvector
[7] https://github.com/facebookresearch/faiss

Chapter 3

Design and Implementation

This section discusses the design and implementation of Disdoc. The design of Disdoc was motivated by several goals:

1. Cleanly separate the retrieval and generation components of RAG, so relevant course material snippets can be separately exposed to students and used to generate answers.
2. Provide a user interface in students' primary means of communication in CS3214, Discord. Discord is a communication platform that supports text, voice, and video interactions as well as an API for creating interactive bots. In Discord, communication is separated into servers (e.g. a CS3214 server), which are made of topic-specific channels (e.g. announcements or exercise 1).
3. Collect detailed data on student use and perceptions during deployment, while complying with IRB requirements and offering extra credit opportunities to student regardless of study participation.
4. Support enough document types to ingest the majority of a Computer Science course's supporting materials.

The core component of Disdoc, which was developed first, is its RAG pipeline depicted in Figure 3.1. In the RAG pipeline, the retrieval system finds snippets from course material that are relevant to a student's question. These snippets are eventually displayed to the

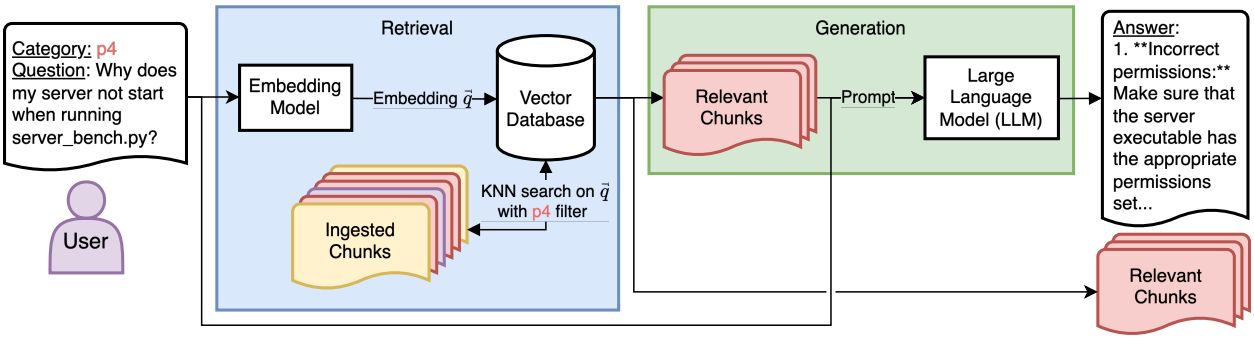


Figure 3.1: Diagram of Disdoc's RAG pipeline.

student, but are also used to generate an answer which includes course-specific information in the pipeline's generation component. More information about the RAG pipeline is given in Section 3.2.

Given the popularity of Discord in CS3214, we developed a Discord bot for students to interface with the RAG pipeline. To interact with Discord bots, Discord users send commands, each with a set of parameters, associated with the bot. The primary command for Disdoc's Discord bot is `/ask`, which accepts as parameters the student's question, question category for the RAG pipeline, and a flag describing whether the student wants to see an LLM-generated answer in addition to the relevant snippets from course material. In response to `/ask`, Disdoc's Discord bot posts a list of relevant snippets from RAG's retrieval step, along with a streamed versions of the RAG pipeline's LLM-generated answer. The Discord bot is discussed further in Section 3.5.

In order to collect detailed data on student usage and perceptions while complying with IRB requirements[1], Disdoc also has a backend, frontend, and database. The complete system diagram is displayed in Figure 3.2. For IRB requirements, all of these components integrate with CS3214's existing student authentication system and coordinate to record the consent status of students. Users must indicate their consent status, even if they decline

[1]Our study was approved by the IRB on April 5, 2024 and our protocol number is 24-275

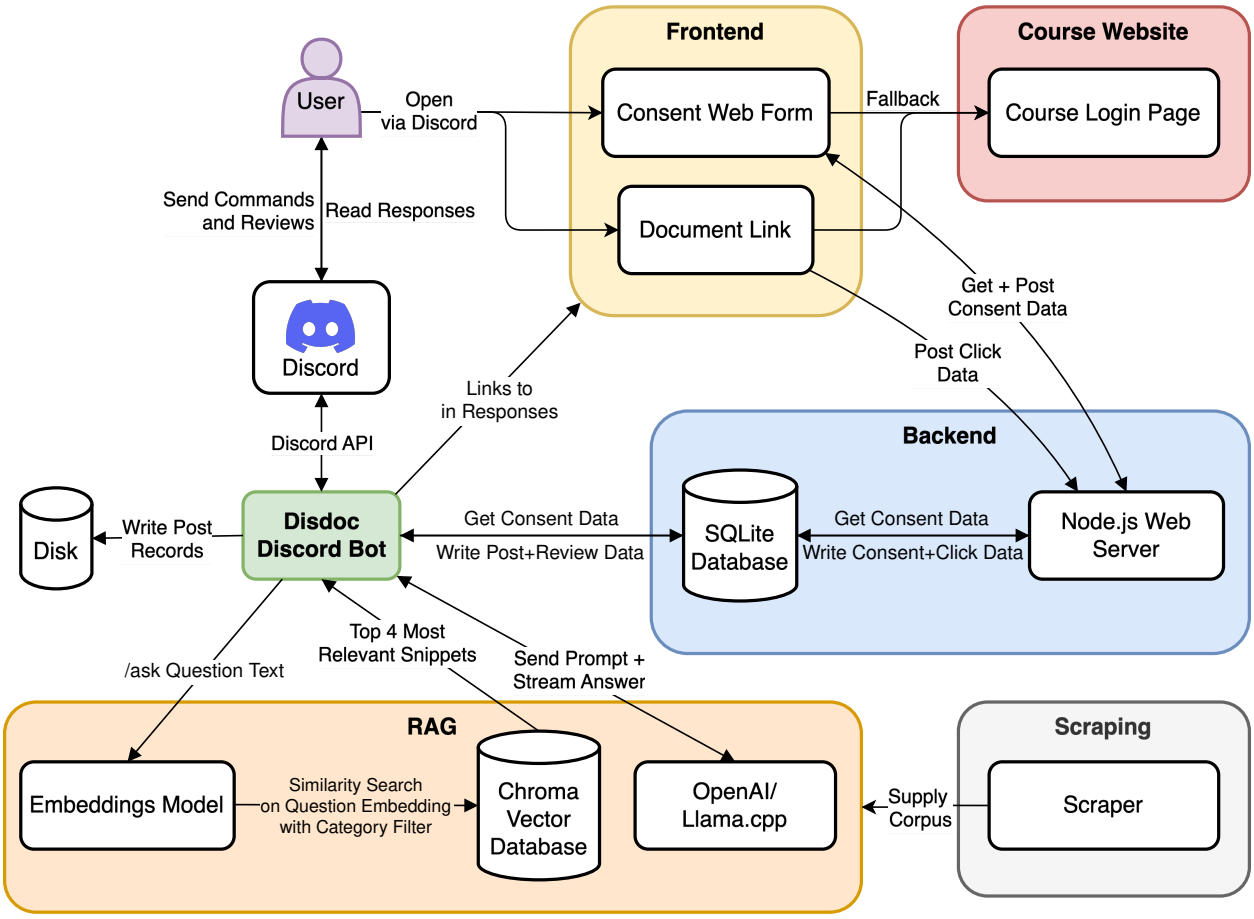


Figure 3.2: Overview of Disdoc's architecture

data collection, before they can use any of Disdoc's functionality.

Then, we collect three types of data. First, when a student asks a question with `/ask`, the Discord bot writes to disk a copy of the student's question, category, retrieved snippets, and LLM-generated answer. Second, whenever a student clicks on a link above a retrieved snippet in Discord, which lead to the snippet's full document, we record their click by leading them through our frontend's link-forwarding service. Lastly, students can leave a review on any of the tool's responses by clicking a "Start Review" button at the bottom of the response. The data from this review is saved by the Discord bot in Disdoc's database.

In summary, Disdoc consists of the following components, which are each detailed in a

corresponding section:

1. Scraping - scrapes course materials to form the course corpus.
2. RAG - ingests the course corpus, retrieves documents relevant to a student's question, and generates answers using an LLM.
3. Frontend - provides forms for study consent and a link forwarding service that tracks clicks on links to course materials.
4. Backend - API endpoints for recording consent status and user activity data (clicks, reviews, and questions).
5. Disdoc Discord Bot - connects the consent data, user activity tracking, and RAG pipeline via into a single interface that answers students questions.

3.1 Scraping

Due to the large number and rapidly changing nature of course documents for Computer Science courses such as CS3214, we automate the process of scraping documents from a variety of sources, including course websites, local manual pages, and git code repositories.

For websites, a naive approach would be to download and analyze the HTML source code of a website via an http client tool such as `curl` [76]. However, this approach is incompatible with pages (like CS3214's course website) that dynamically update their content after the initial page load via JavaScript, because `curl` and similar tools lack a JavaScript engine to execute code. As a result, such tools would give the placeholder HTML content presented until JavaScript can execute, missing some or all of the meaningful content depending on the page.

To address this, we use the puppeteer [32] library to control a headless instance of the Google Chrome browser, wait until network activity is complete (as a proxy to indicate that JavaScript activity is complete), then download the content for ingesting. For files linked on a webpage, this means we download the content of the file, and for web pages, we download the HTML content. If we downloaded a webpage, we additionally parse the downloaded HTML content for any linked webpages and queue those for further scraping. To prevent endless scraping, we limit the queued links to just those from the current website's domain. The scraping software can be easily configured to allow specific external domains, but this was not necessary for the CS3214 course. The scraper also saves any links to code repositories (e.g. GitLab and GitHub), then downloads them via `git pull`.

This approach means that every resource available by the web can be downloaded automatically, including but not limited to webpages, PDFs, code files, PowerPoint presentations, and text files. We refer to the collection of all of these resources as the course's corpus.

We also add to the corpus a subset of manual pages (manpages) from the CentOS system that students write and run code on. We include manpages from the following sections:

- 0p: overviews of the POSIX API students use
- 1 and 1p: command-line tools
- 2: system calls
- 3 and 3p: library calls
- 5: file formats and conventions (e.g. `/proc`, which students use for assignments and debugging)
- 7: miscellaneous manpages (e.g. `inode`, which students need for an assignment)

We filter out any manpages marked as deprecated or for libraries irrelevant to the CS3214 course, such as the LAPACK library for linear algebra. We have also seen that a number of CS3214 students read manpages online because of their accessibility, without checking whether they are out of date or for a different operating system than the CentOS cluster students write code for. To counter this problem, we provide students one-click online access to the correct versions of manpages via our tool. To do this, we convert the CentOS system's manpages to PDFs using the GNU roff tool [13] and host these PDFs alongside the consent and link tracking systems discussed later in Section 3.4. We then use these PDFs throughout the tool, linking to them and providing PDF-specific features, such as the annotated page previews we discuss in Section 3.5.1.

From here, we perform some filtering on the course's corpus. Image and video files are not understood by the AI models we use, so we filter them out. Binary files, zip archives, tar archives, and git-related files do not contain useful information and are also filtered out. We also filter out, via a manually written list of file and directory names, three categories of resources:

1. Resources with changing content, particularly after the AI ingests the corpus: excluding such resources helps prevent the tool from producing out of date or misleading information. Examples for CS3214 include a page with a list of open exercises (which will change as the semester progresses) and a page listing upcoming office hours (which may change week to week).

2. Irrelevant resources: excluding these resources helps prevent the tool from polluting accurate responses with irrelevant information. For example, CS3214's exclusion list includes a demo web app, because students do not need to understand or modify it in any way to complete the project it is part of. The exclusion list also includes the

course login page, since there is no useful information on it.

3. Duplicate resources: excluding resources that chiefly repeat the information of another helps improve the diversity of the resources retrieved from the vector database. For queries asking for links, this means that each link is sufficiently different; for queries asking for an AI-generated answer, this means that the AI is given more diverse information to use when synthesizing its answer. For example, we exclude the statically hosted copy of a GitLab repository for one CS3214 exercise.

While we could use this filtered corpus, it would have an important limitation: there is no way to separate resources by assignment or any other category. As a result, searching through the filtered corpus for resources related to a question about a particular assignment could yield an undesirable mix of resources about different assignments. For example, several CS3214 assignments use the `accept` system call in their instructions or code, so searching through the filtered corpus to answer a question about `accept` for a particular assignment would yield irrelevant uses of `accept` in other assignments.

To prevent this undesirable mixing, we assign a single category to each resource in the corpus. We assign resources to these categories using a combination of heuristics based on the resource's original URL and manually written file lists. For example, any resource whose URL indicates that it is contained in the `/projects/project1` directory or its sub-directories will be assigned to the `p1` category. We use the following categories for CS3214:

- A category for each exercise (`ex1`, `ex2`, etc.) - exercise instructions, exercise FAQs, and starter code if provided
- A category for each project (`p1`, `p2`, etc.) - project instructions, project FAQs, help session resources

- Midterms (`midterm`) - answer sheets for past midterms, including explanations
- Finals (`final`) - answer sheets for past final exams, including explanations
- Lecture materials (`material`) - PowerPoint slides, articles on the course website, example code, FAQ pages about course content
- Administrative materials (`admin`) - syllabus, course logistics, staff list
- Manual pages (`manpages`) - a subset of PDF version of manpages from the CentOS system that students write and run code on

We store each resource on disk in the folder with the name that corresponds to its category. We also support nested folders within each category folder to prevent file name clashing. For example, we recreate the folder structure of the path component for each webpage's URL within each category folder to prevent name clashes when multiple pages share the same name but are stored in different paths on the web. A key benefit of storing resources on disk like this is that we can easily check the correctness of categories. Additionally, if a resource is given the incorrect category, it can be be manually moved to the correct category's folder before ingestion. Once sorted into category folders, the corpus is ready for ingestion for RAG.

As a last step, we also create a JSON map from the path of the resource on disk to its URL on the web. We save these links so the Discord bot can link students to documents in the corpus when they are retrieved as part of RAG.

In order to support a different class, an instructor must take two steps:

1. Tailor or replace the existing web scraper to work for their course: For courses with custom websites, instructors can easily configure our existing web scraper using a con-

figuration file. However, our web scraper is not one size fits all; for other course deployment schemes, such as courses embedded in a learning management system, instructors must modify our web scraper, write their own, or manually collect documents.

2. Modify the heuristics to group documents into categories: the list of categories and the heuristics to sort documents into categories is necessarily course-specific. As a result, instructors must modify these for their course.

The amount of work needed for these two steps will vary by course, but the rest of the tool works with no modification by the instructor. For CS3214, we spent several days implementing a web scraper from scratch and a single day implementing the heuristics from scratch.

3.2 RAG

In order to connect students with snippets from the course corpus that are relevant to their questions and to generate answers to those questions which include course-specific information with an LLM, we use a RAG pipeline. To accommodate varying student workflows and preferences, the RAG subsystem operates in two modes:

1. Relevant resources: this retrieves and yields the top four most relevant resources from the vector database (i.e., the retrieval step of RAG).

2. Answer: this retrieves links from the vector database, as in the previous case, *and* provides them to an LLM with the original question to generate an answer (i.e., the generation step of RAG). Yields the relevant resources and the generated answer.

In other words, we complete the retrieval step of RAG regardless, but only use the generation step if the user opted in. We do not to provide an answer-only mode so that LLM-generated answers are never opaque; the Discord bot that uses the RAG pipeline will always show, and students can verify or read, the resources that the LLM used to generate its answer.

We first discuss ingestion of the course corpus in 3.2.1, then discuss retrieval in 3.2.2, and finally generation in 3.2.3.

3.2.1 Ingestion

We ingest all of the resources sorted into folders following the scraping process outlined earlier. We start by converting a resource into text according to the resource's associated loader. The loader used depends on the file type of the resource. We use a combination of custom loaders and loaders provided by the LangChain library [35], which are often derived from other libraries like Unstructured [78].

- Word documents: word document loader from Unstructured [78]. This extracts all of the text content of a given word document into one stream.
- HTML files: HTML document loader from Unstructured [78]. This extracts the full text content of the page, converting any HTML entities (e.g. `>`) as necessary.
- Markdown files: Markdown document loader from Unstructured [78]. This converts the Markdown file to HTML, then applies the Unstructured HTML loader.
- PowerPoint files: PowerPoint loader from Unstructured [78]. This extracts the full text content of each slide, but does not extract any text from images.
- PDF files: customized version of LangChain's PyMuPDFLoader [74]. It splits the PDF into pages, then extracts the text content from each page. As a result, chunks do

not span multiple pages. It also dehyphenates the end of lines that cut off mid-word, which is common in documents generated from LaTeX.

- Other files: LangChain's default text loader [35]. This has the effect of treating code files as raw text.

Although most loaders that LangChain includes from Unstructured provide the option to split file contents into multiple resources according to the file type's notion of title text and body text, we do not use this. Rather, we configured the loaders to return an uninterrupted stream of text with no semantic separation. This is because, for the CS3214 corpus, semantic separation was yielding chunks too small to be useful.

Additionally, we do not take advantage of the optical character recognition (OCR) functionality that some Unstructured loaders provide. This functionality would allow us to extract text from images embedded in documents, such as images in a PDF file. While other courses may benefit from this, we disable this functionality because the CS3214 corpus does not contain any images with substantive text content.

Now that we have a collection of each resource's complete text content, we chunk the resources with a maximum size of 500 characters and a permitted overlap of 50 characters using LangChain's `RecursiveCharacterTextSplitter` [35]. We use these constants to ensure that even models with a small context size, particularly Llama 2 with a context size of 4192 tokens, can fit the content of four chunks and a long student question in a single prompt without issue. Also, note that the number of characters rounds down to the nearest word boundary, so words are not cut off midway.

We then store the category corresponding to the chunk's source document's folder in the chunk's metadata, along with any metadata that the chunk's loader populates. At minimum, every LangChain loader saves the path to the chunk's source document in its metadata. For

chunks from PDF files, LangChain's PDF loader also saves the page of the PDF the chunk is from.

We then calculate the embeddings for every chunk using the `multi-qa-mpnet-base-cos-v1` model from Sentence Transformers [66]. We use this model because it is optimized for question and answer use, preliminary experimentation indicated that cosine similarity was the best metric for our corpus, and because it is the best performing cosine model in Sentence Transformers' published performance tests as of April 2024.

Once we have the chunks and their embeddings, we add them to the Chroma vector database [12]. This saves the chunk's content, along with its metadata. We use Chroma because of its metadata filtering features, which enables our search for relevant chunks with specific categories. This feature helps reduce the chance of the retrieval step yielding irrelevant resources from other assignments, which we find to be particularly common in a course like CS3214 due to the repetition of concepts and API calls across assignments.

In addition, we designed the ingestion process to accommodate rapid prototyping and additions or changes to course material. Most importantly, when ingestion is run repeatedly on the same vector database, new documents are ingested and old chunks are deleted if their source file has been deleted. As a result, to delete a chunk, instructors can delete the source file on disk, to change the category of a chunk, instructors can move the source file to the desired category's folder, and to update a chunk, instructors can replace the source file with the new version. This means that instructors can incrementally build on the same vector database as they release content during a semester. Instead, if an instructor wants to fully replace the vector database, ingestion is moderately quick: ingesting all 54 967 chunks across 533 documents from the CS3214 corpus takes around 10 minutes.

3.2.2 Retrieval

When a student submits a question, they do so along with a category. This category is used to filter the chunks from the vector database we can search before we perform a similarity search using Chroma's metadata filtering feature in an effort to improve chunk relevance. However, if we used the category directly, we would overly limit students. In particular, many questions about course assignments require a combination of knowledge about the assignment and knowledge about course lecture content needed to understand the assignment. To address this issue, we expand the filter to include the lecture material (`material`) and administrative material (`admin`) categories if the question's category is for a project or exercise.

Unfortunately, Chroma's metadata filtering features can become a significant bottleneck if a single collection is too large. This is because Chroma does not create indices for metadata, meaning that lookups with metadata filtering are $O(n^3)$ without any SQL optimizations, where n is the number of text chunks in the collection. The Chroma team is currently addressing this issue as of April 2024. As a result, while we do not experience a significant bottleneck with the 54 967 chunks in our final vector database spread across multiple categories, we experience a significant slowdown (approximately 1.5 minutes per search) when we include all 512 116 chunks of manpages from the system students run code on and perform a search for only resources with the `manpage` category. Therefore, we filter the manpages as outlined in Section 3.1 to limit the number of manpages we ingest to only those relevant to course material until Chroma addresses this issue.

After metadata filtering, we perform a similarity search using the same embeddings model and similarity metric for as for ingestion (`multi-qa-mpnet-base-cos-v1` and cosine). We perform this similarity search on the embedding of the student's question without any preprocessing. Because we use cosine similarity as our similarity metric, we retain the similarity

scores for each source to display as an easily understandable "percent similarity" to the user in the Discord bot frontend.

We limit the similarity search to the top four most relevant chunks. Identical to with chunking, this limit is imposed so that we can easily fit four chunks and a long student question into a single prompt, even for LLMs with smaller context sizes like Llama 2's 4192 tokens. However, the maximum number of chunks also helps lower the response time of larger models. For this reason, we keep it constant across LLMs. However, this limit is easily configurable so that more than four chunks can be used if desired.

During preliminary testing, we noticed that Chroma suffers a very poor query time on a cold start (e.g. $4.095s$ when cold vs. $0.037s$ when hot). This is because Chroma lazily loads its data structures into memory. After the first query, these data structures are already loaded and do not impair query performance. Therefore, to alleviate this issue, we perform a simple mock query during startup.

3.2.3 Generation

For questions that request an AI-generated answer, we choose between Llama 2 13 billion parameters [81] and GPT-3.5 Turbo [6] depending on a configuration parameter. At the time of testing, the current version of Llama 2 13 billion had a context size of 4096 tokens and the current version of GPT-3.5 Turbo (`gpt-3.5-turbo-0125`) had a context size of 16 385 tokens. To host the Llama model, we use the llama.cpp runtime. Additionally, we chose Llama 2 13 billion parameters over higher parameter Llama models because it was the newest and largest Llama model that ran on our available hardware. Also, we used GPT-3.5 Turbo because of cost concerns with the more expensive GPT-4 and because preliminary testing yielded a minimal difference between the responses of GPT-3.5 Turbo and GPT-4 for our

use case.

Besides settings the maximum message length to 500 tokens, we leave all parameters at the defaults provided by llama.cpp and OpenAI's Python API, respectively. Additionally, we provide our prompt template in the appendix. To prepare the prompt, we replace the `{question}` and `{content}` placeholders with the student's question and the concatenated text content of all four relevant chunks from retrieval, each separated by newlines. The prompt template contains Llama 2's default system prompt and a user prompt initially inspired by PrivateGPT's [7] default query prompt, then refined through experimentation to further prioritize information from relevant chunks.

We also designed Disdoc to easily support additional LLMs. First, we support communicating with any model that is supported by llama.cpp. As such, newer Llama models published after testing, such as Llama 3 [1], are already supported. Additionally, we support all current OpenAI models, as we wrap around their Python API. Lastly, due to our representation of LLM output as a Python generator that yields strings (tokens), it is trivial to add support for any other LLM provider that communicates via a streaming HTTP API or similar.

3.3 Frontend

The frontend is a minimal, statically compiled website built on Svelte [15]. We include routes for students to check and submit their consent status and for our system to track what links students click. The consent page is a web form that displays the student's account information and consent status (if indicated in the past) and can be seen in Figure 3.3. Students use this form to change their consent as they wish, sending an HTTP request to the backend with student's account username and updated consent status. The link click tracker sends an HTTP request to the backend containing the student's account username

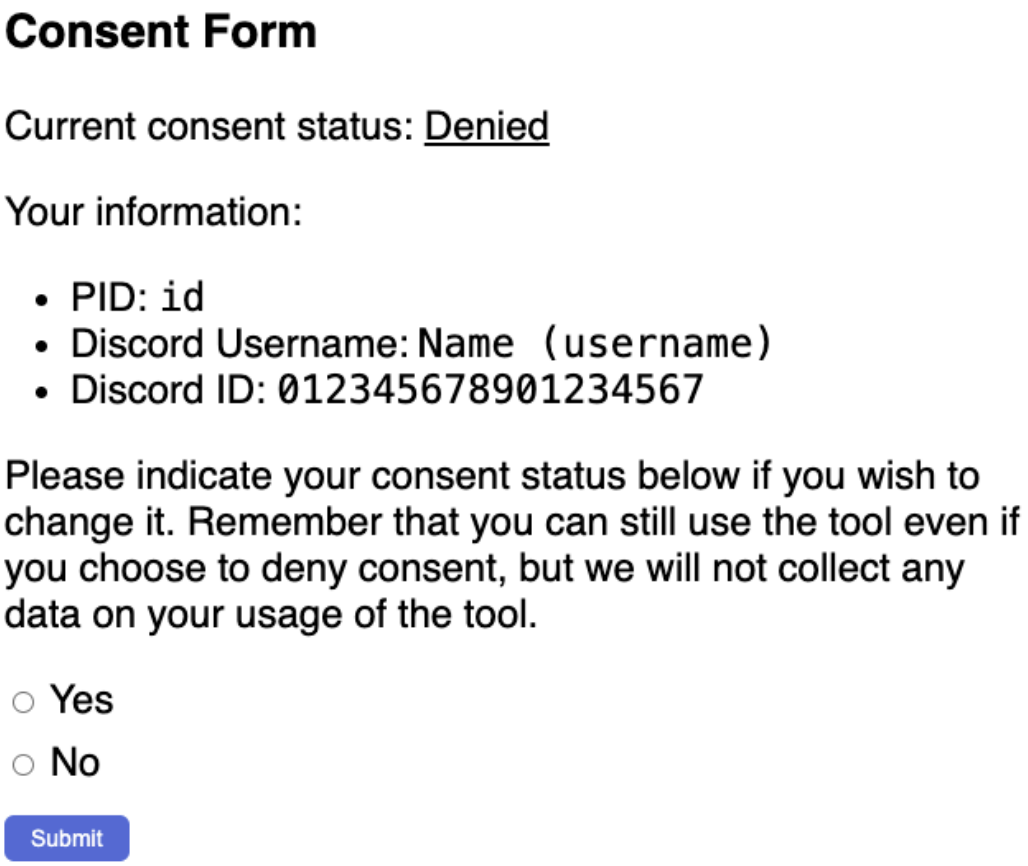


Figure 3.3: Frontend consent page for a mock user: `https://courses.cs.vt.edu/cs3214/test/consent.html?discordId=01234567890123456 7&discordName=username&discordNick=Name`

and the link the student will navigate to, then forwards the student to the link that they clicked.

To ease authentication for students and simplify our own design, we leverage the CS3214 course's existing authentication services, which itself unifies several of Virginia Tech's authentication services. Effectively, the backend is acts as a proxy for this authentication service and if a request to the backend ever reports an authentication failure, we redirect to the CS3214 course website's login page. However, it is possible to modify the authentication

process to communicate with an OAuth2 or other central authentication server rather than our backend. We discuss user management and authentication further in Section 3.4.

While authentication provides us with a student's username, both pages still need a way for data to flow from the Discord bot to the frontend. In the case of the consent page, we need information about the Discord user who is changing their consent status; for the link click tracker, we need information about the post the link was embedded in and the destination the student is navigating to. To address this need, we use URL query parameters. Specifically, the Discord bot adds query parameters for any information that needs to be transferred, then the frontend will read those URL query parameters to retrieve the information.

3.4 Backend

The backend is a minimal Node.JS [62] backend that interacts with an SQLite [37] and coordinates authentication with the CS3214 course website. Notably, this SQLite database is shared between the backend and Discord bot. In total, the backend provides the following endpoints:

1. `/consent GET/POST` gets or submits a user's consent status.
2. `/click POST` logs a click to a link provided by the Discord bot. This endpoint is `POST`ed by the frontend when the user opens the link in their web browser.
3. `/manpdfs/<file> GET` serves the PDF version of various manpages
4. Files from the frontend's static build

We protect the consent and click endpoints by authentication checks. We forward these authentication checks to the CS3214 course backend, which unifies several authentication

protocols provided by the university. The CS3214 course backend replies with the username of the authenticated user, which we use in database operations in our `POST` requests or include in our reply to the frontend in the `/consent GET` request. This forwarding is possible because the frontend, backend, course website, and course backend share the cookies used for authentication, as the cookies are valid for all paths under the same subdomain.

The schema for the database is shown in Figure 3.4. The backend writes the `users` and `clicks` tables directly, while the Discord bot writes to the rest of the tables (`posts`, `llm_reviews`, and `retrieval_reviews`). However, both the backend and Discord bot read from the `users` table.

Unfortunately, the `users` table is complicated by differences in identifying users in the backend and Discord bot. In short, the backend and Discord bot have access to disjoint information to identify users. While the frontend has the username returned by the CS3214 authentication service, the Discord bot only has the Discord ID provided by the Discord WebSocket API when a command is invoked. To address this, we associate a username, Discord ID, and consent status in the `users` table. This process is illustrated in Figure 3.3. In essence, students follow a registration process that spans the Discord bot, frontend, and backend, depicted in Figure 3.5 and detailed below:

1. The student uses the `/consent` command in the Discord server.

2. The Discord bot generates a unique link for the user to submit their consent, see Section 3.3 for details on how it is generated.

3. The user opens the link in their web browser, which loads the frontend.

4. The frontend checks authentication and retrieves the current consent value (if any) to the backend's `/consent GET` endpoint. If they are not logged in, they are redirected

to the login page.

5. The frontend updates the consent of the page to display the Discord information stored in the URL query parameters and the username information retrieved from the authentication check.

6. The user edits and submits the form.

7. Namely, the Discord bot generates a link when a student uses the `/consent` command,

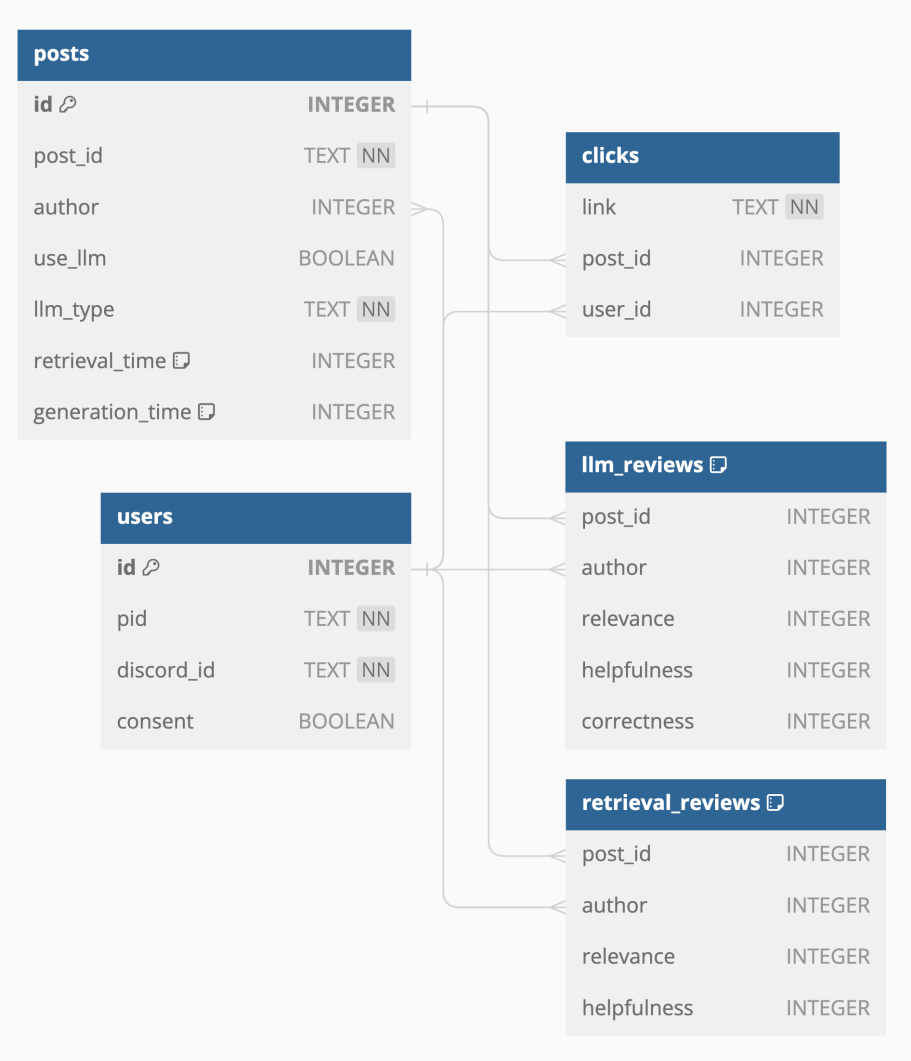


Figure 3.4: Database schema for the backend and Discord bot.

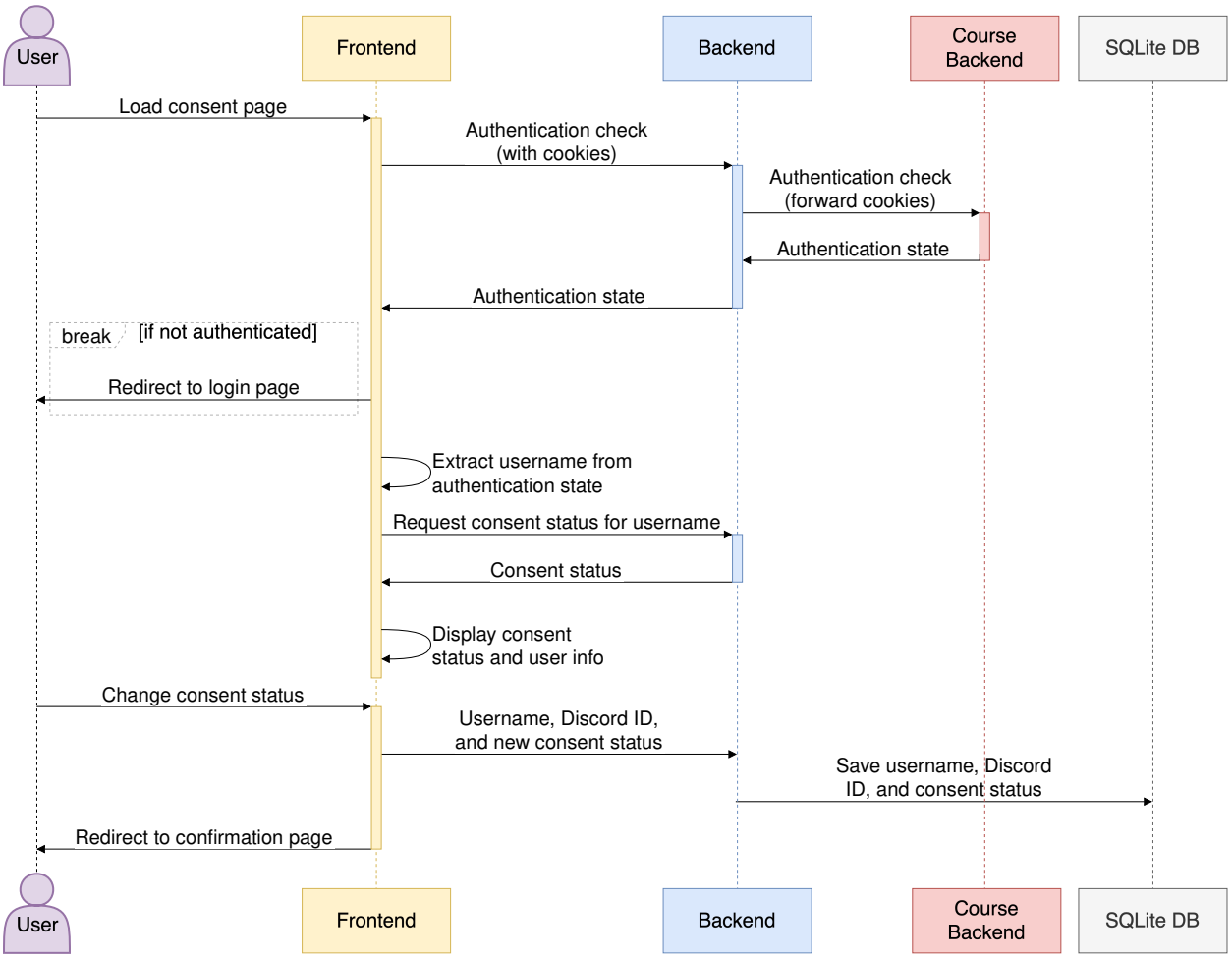


Figure 3.5: Authentication and consent process using the frontend.

the student clicks the link (which hits the backend for the frontend's files), the frontend checks the student's current consent status with the backend's `/consent GET` endpoint (which requires them to log in if they are not already), then the frontend updates the page with the information stored in the URL query parameters and authentication's response, and result of the `/consent GET` request. When the user submits the form in the frontend, the URL-encoded results is sent to the `/consent POST` endpoint and we make or update a row in the users table for the student.

The `posts` table is used to log information about posts for data collection. Every time a

student asks the Discord bot a question, the bot first retrieves the student's ID from the `users` table according to their Discord ID. Then, the bot stores the post's unique Discord ID (for manual lookup purposes, not for use as a foreign key), student's user ID, whether the student requested an LLM-generated answer, the LLM type set at the time of the question, the time taken to retrieve relevant chunks from the vector database, and the time taken to generate and send a full answer to the question from the LLM (if used).

The `clicks` table in the database is used for storing data we collect via the `/click POST` endpoint. Namely, when the endpoint is hit, the backend checks authentication and uses its result to retrieve the user's ID from the users table in the database. We then log the user's ID in the `clicks` table along with the URL of the link clicked and the ID of the database entry for the post it is from, both stored in the request's JSON payload.

The `llm_reviews` and `retrieval_reviews` tables are used to store the results of student's reviews. When the student submits a review, the Discord bot adds a row to the table that corresponds to the type of review.

We make use of SQLite's cascading delete features to delete all posts, link clicks, and reviews associated with a user when they revoke their consent.

3.5 Disdoc Discord Bot

The Discord bot unifies all of the components so that students can ask for links and AI-generated answers to their questions and get links to the consent frontend in the same messaging platform that they use for CS3214. It provides them the following functionality:

1. `/ask <use_llm> <category> <question>` command: Ask a question using documents in the given category, optionally using an LLM to generate an answer

2. `/consent` command: Displays a unique link to the consent page exclusively to the command's user. This generates a link with URL query parameters depending on the command's user, see Section 3.3 for details on how it is generated

3. `/reviewcount` command: Displays how many reviews the user has given

4. Ability to review posts using buttons and forms in the Discord client and save the results to the SQLite database on submit

5. Saves data on retrieval and generation times for students' questions

6. Commands for controlling the bot, only usable by Discord server administrators:

   (a) `editbuffer <buffer_seconds>`: Set the number of seconds the bot buffers tokens when streaming responses from the LLM before sending them to Discord via a message edit. This is set to prevent throttling by the Discord API

   (b) `llmtype <llm_type>`: Set the bot to use either Llama 2, OpenAI's GPT 3.5, or a mock LLM (generates random text) to generate answers to student's questions

   (c) `shutdown`: Shuts down the bot

We supply type annotations and choices to all commands and command arguments, so Discord provides auto-suggestions and form validation. For example, students are given a list of possible category choices for the `/ask` command and are restricted to them by their Discord client. Additionally, besides seeing other students interact with the bot, auto-complete is how students discover the Discord bot's capabilities: Discord lists the commands students can use to interact with the bot as auto-complete options when they type `/` along with their descriptions.

3.5.1 `/ask` Command

Students use the `/ask` command to ask for links and an optional LLM-generated answer for their question. The command takes the following arguments:

1. `use_llm`: Whether to generate an answer using an LLM in addition to retrieving relevant chunks (true/false)
2. `category`: Category of the question (see Section 3.1 for discussion and options)
3. `question`: Student's question

At minimum, the `/ask` command will generate a post listing the four links deemed most relevant to a student's question by our RAG subsystem. We refer to these as "retrieval" posts. A sample retrieval post is displayed in Figure 3.6. For each post, it will display the relevance of the chunk according to the cosine similarity of the question and chunk's embeddings, the name of the resource the chunk is from, a link to the complete resource, and a preview of the relevant chunk.

If the student asks for an LLM response using the `use_llm` parameter, an additional post will be generated containing a streamed copy of our RAG subsystem's answer to their question. We refer to these as "LLM" posts. An example of such a post is shown in Figure 3.7. The post starts with placeholder content, which is replaced and then updated as the LLM generates tokens. We update the post via edits, requiring us to chunk updates on a timeout to prevent the misuse of the Discord API and a resulting rate-limit. As a result, we buffer the LLM's contents and send updates in time intervals set by the `/editbuffer` command.

All posts generated with the `/ask` command are publicly visible, so students can share information, interact with other students' posts, and use Discord's search tools to search for

Figure 3.6: An excerpt from a retrieval post generated by the Discord bot in response to `/ask True p4 why does my server not start when running server_bench.py`. The blue links above each chunk preview lead to the complete resource online. The fourth chunk and start review buttons are omitted for brevity.

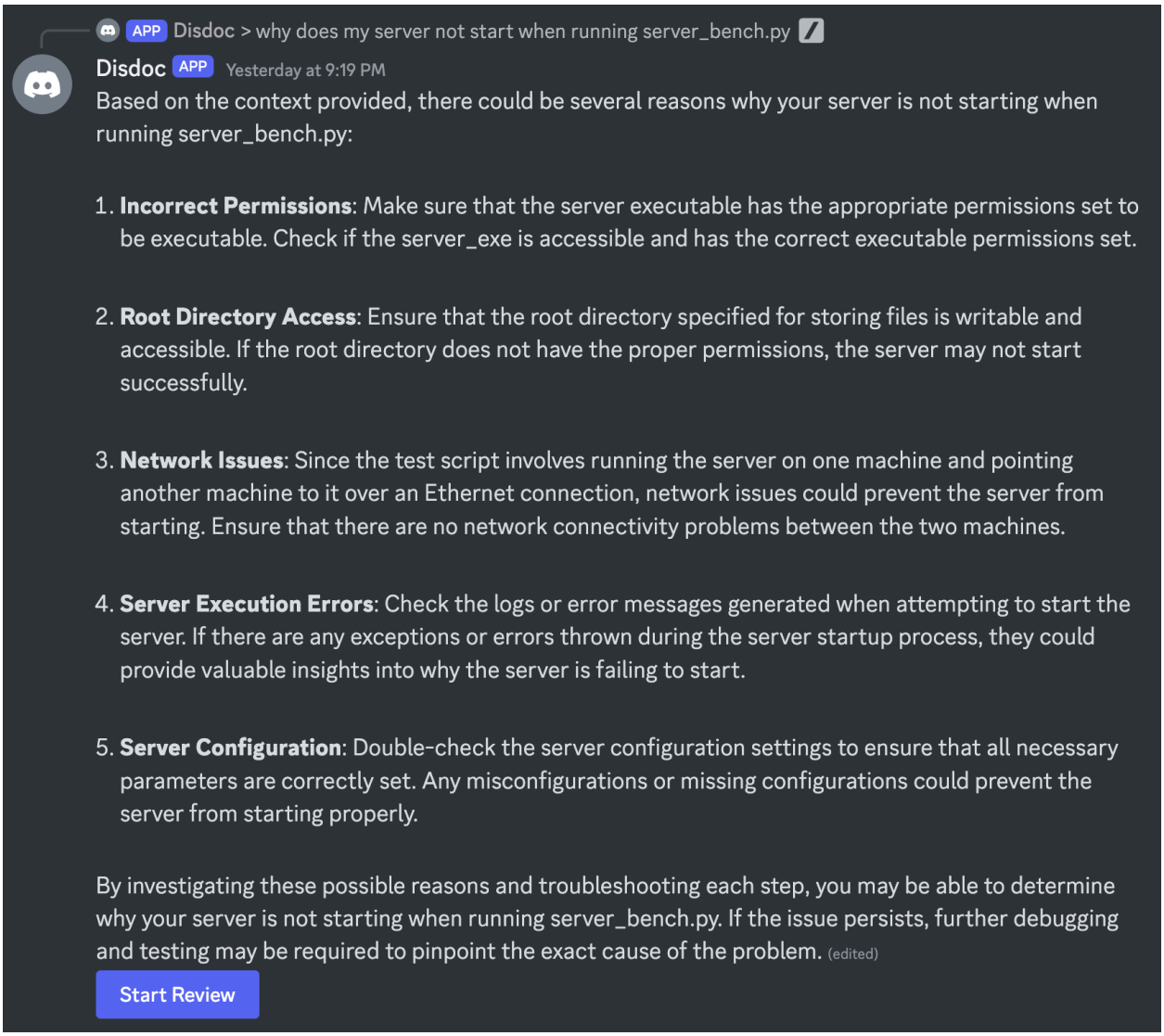


Figure 3.7: An example LLM post generated by the Discord bot in response to `/ask True p4 why does my server not start when running server_bench.py`. To generate this response, the RAG pipeline used the relevant chunks listed in Figure 3.6.

past answers to their problem. Additionally, all posts include a button to start a review for the post. Reviews are discussed further in Section 3.5.2.

In more detail, Handling the `/ask` command is as follows, as displayed in Figure 3.8:

1. User sends a `/ask` command via their Discord client (app, website, etc.).

2. Discord sends a message to the Discord bot's WebSocket indicating that the `/ask` slash command has been invoked, along with its parameters and the Discord ID of the user who invoked the command.

3. The Discord bot receives the message, extracts the parameters, and queries the `users` table of the SQLite database (which is shared between the bot and the backend) for the consent status of the user who used the `/ask` command by their Discord ID. Note that we perform minimal validation of parameters (type checks everywhere and enumeration validation where appropriate), as the Discord client will not allow the user to send messages containing invalid parameters.

4. The SQLite database yields the consent status of the user. If the user has not indicated their consent status via the frontend, we instruct the user to use the `/consent` command via a Discord message and do not proceed any further. If the user has indicated consent, we proceed to the RAG subsystem for retrieval.

5. We calculate the embedding of the student's question, provided via the `question` parameter to the `/ask` command.

6. We query the Chroma vector database for the top four relevant text chunks in the student's requested category based on the calculated embedding. This yields a list of chunks, along with each chunk's calculated similarity score and metadata. Control transfers back to the Discord bot subsystem.

7. We prepare the message containing RAG's relevant chunks. In the resulting message, each chunk will display its source document's name, similarity score (as a percentage, since we use cosine), its link, and the chunk's contents along with a button to give feedback about the relevant sources. We obtain the chunk's name and link using a combination of the chunk's metadata and our map from file path to link, which is

generated during document ingestion. The chunk's content is displayed in a variety of ways, depending on the type of the chunk's source document:

- For PDFs, we generate an image preview of the page containing the chunk's text content, with the relevant text chunk highlighted in yellow. We also maintain a cache of image previews.
- For code snippets, we create a code block for the associated language so that the user's Discord client can display syntax highlighting for it.
- For all other sources, we display the text without any processing.

8. We send the full post to Discord to be displayed. If the user did not request an LLM-generated response via the `use_llm` parameter, we do not proceed. Otherwise, we continue.

9. Post a blank message, which we will progressively update with the tokens of the LLM's response. Control transfers back to the RAG subsystem for generation.

10. We replace the `{question}` and `{context}` placeholders of the prompt template with the student's question and the joined text content of all four relevant text chunks.

11. We feed the prompt to the LLM indicated by the LLM set by the `/llmtype` command. While the specifics differ by LLM type, this will result in a stream of tokens (represented as a Python generator).

12. As the Python generator yields each token, we build up the full content of its response. Every time a token is added and the time interval set by the `/editbuffer` command has elapsed, we send an edit request to the Discord API with the current contents of the LLM's response. This will then display the new tokens in the user's Discord client. We continue this process until all tokens are exhausted.

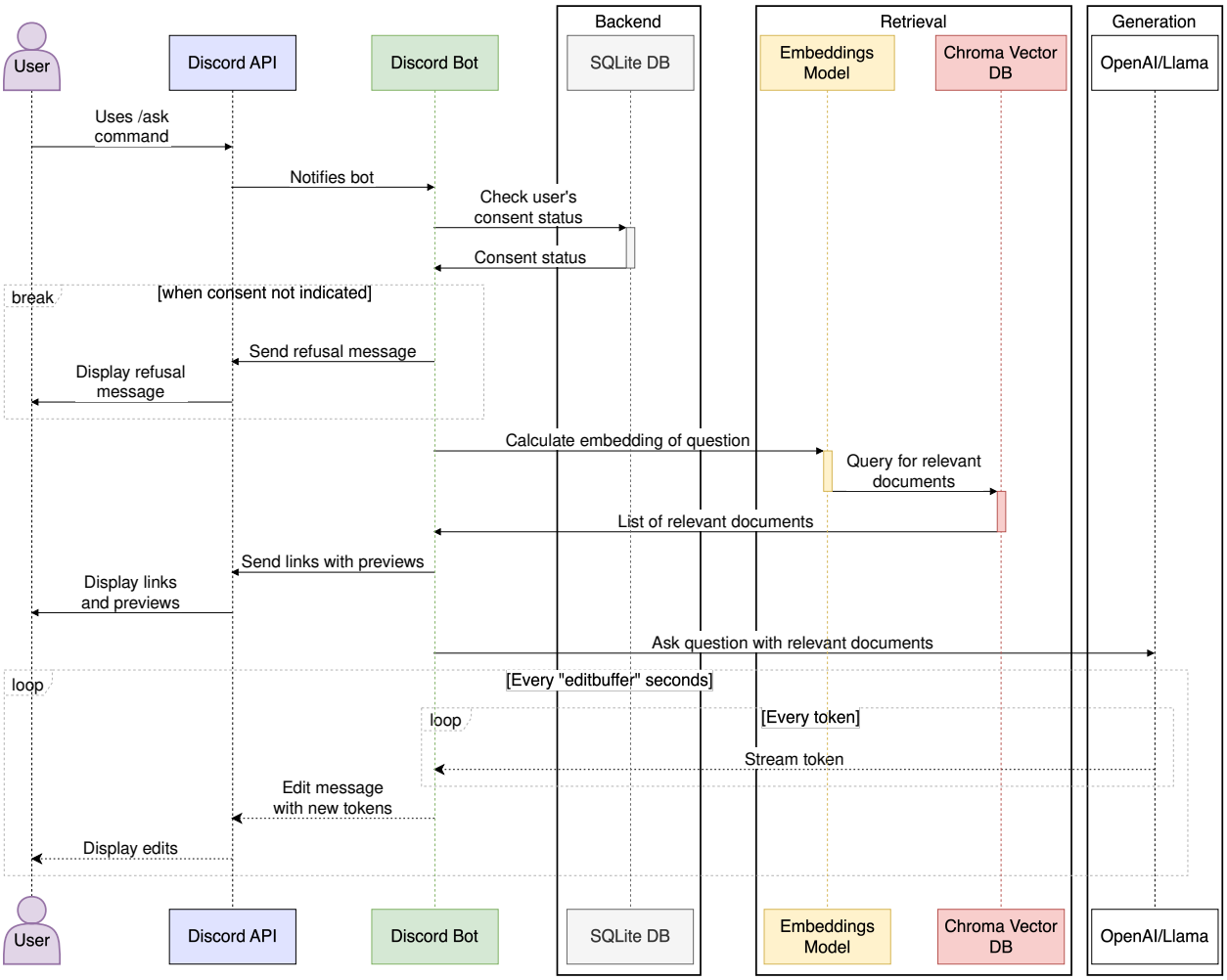


Figure 3.8: Sequence diagram of responding to a user question.

We need to direct users through our link forwarding service when they click on a link in one of the Discord bot's post, so we can track clicks. As such, we modify the original link stored in our map from resource path to link, as constructed during scraping, to point to our link forwarding service. To do this, we change the URL of the link to our link forwarding service's URL and add the required URL query parameters. Specifically, we add the original link from the map and the ID of the post we are linking from. We add additional information to certain destination links, such as page number for PDFs, as a convenience for students.

We also render PDF previews of chunks as a convenience for students. Using PyMuPDF [74],

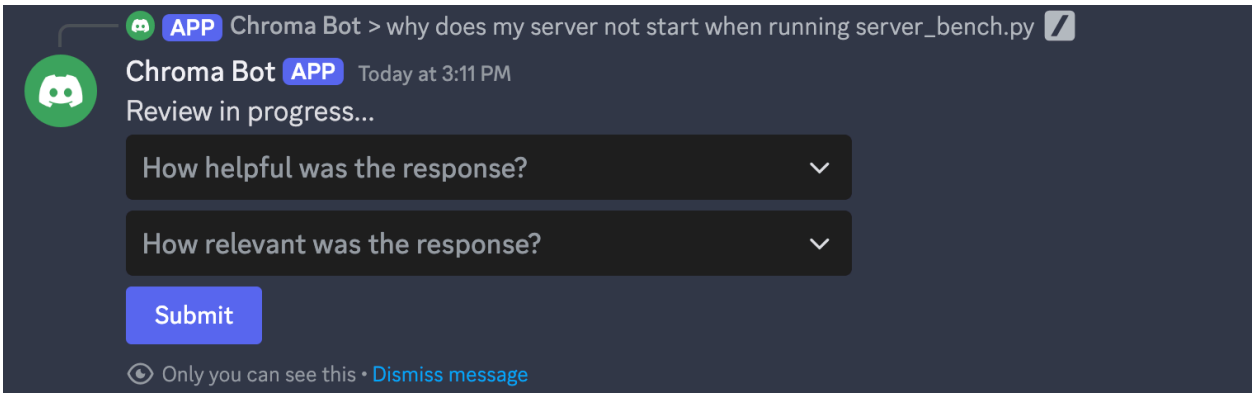
we render the chunk's source PDF document, search for the contents of the chunk in the document, draw a yellow highlight around it, then save the annotated render as an image to embed in the Discord bot's post. Because our PDF loader does not create chunks that span multiple pages, the PDF previews are guaranteed to be an image of a single page and can be easily included in a Discord post.

After responding to a `/ask` command, we save to the backend's database a unique post ID, the amount of time it took for RAG to retrieve the relevant chunks, and, if we created an LLM post, the amount of time it took for RAG to generate an answer and finish sending edits to Discord. We also save the content of generated posts to disk for moderation and future use.

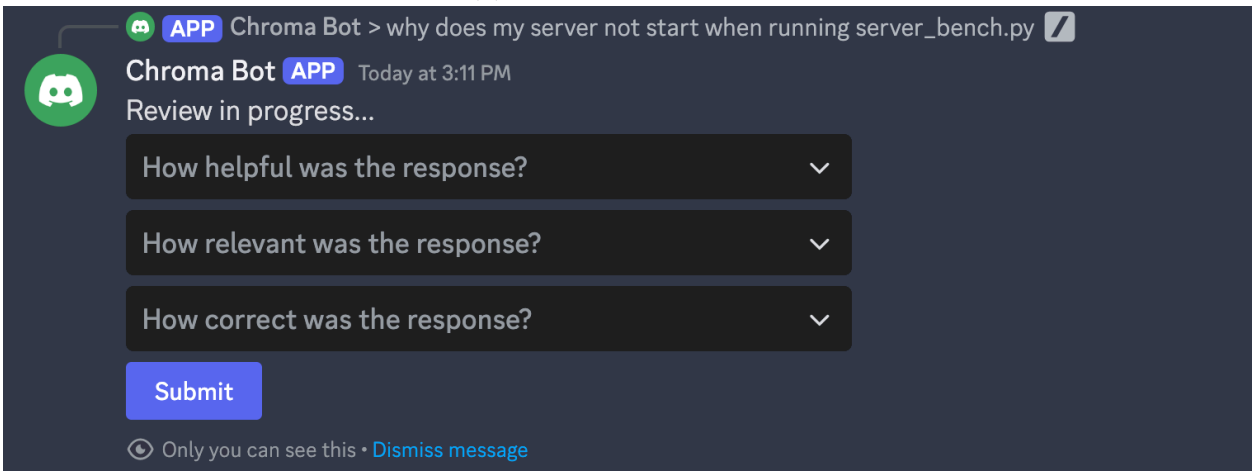
3.5.2 Reviews

At the bottom of each generated post is a "Start Review" button. Students who have given consent click this button to start a review or overwrite a past review for a particular post, depending on whether they have reviewed the post before. The Discord bot then replies with a post containing the UI used to give the review, which is only visible to the student who clicked the "Start Review" button. Because all posts generated by `/ask` are public, any student can click the "Start Review" button. This means multiple students can leave a review on a single post.

Reviews consist of several questions and Likert scale answers given via a set of dropdowns. In the current version of Disdoc, review responses are only used for data collection. Example retrieval and LLM post reviews are shown in Figure 3.9. For retrieval post reviews, we ask students to rank the post's helpfulness and relevance to their question on a 5-point Likert scale. For LLM post reviews, we ask the same questions and add a 7-point Likert scale rating



(a) Retrieval post review



(b) LLM post review

Figure 3.9: Example reviews for retrieval and LLM posts.

of the correctness of the LLM's response. Given the importance of correctness in student-facing material and concerns in the correctness of LLMs, we choose 7 points for correctness to improve the granularity of the metric.

The buttons and reviews use Discord's UI API features, which allow buttons and dropdowns to be embedded in Discord posts and trigger callbacks in the Discord bot's source code when interacted with. Specifically, code in the Discord bot is run when the "Start Review" button is clicked and when a review is submitted. We validate and report whether there were any missing fields by updating the post with a success or failure message. On success, we save the

contents of the review in the `retrieval_reviews` table or `llm_reviews` table, depending the type of the reviewed post.

Chapter 4

Evaluation

4.1 Goals and Methodology

The goals of evaluation are to determine:

- How did students use our tool? Sections 4.2.1 and 4.2.2.
- How did our tool perform during the study? Section 4.2.3.
- How did students rate the retrieval and LLM features? Sections 4.2.4 and 4.2.5.
- Did the reviews change over time, and was there a learning curve? Section 4.2.6.
- What links did our tool retrieve during the study? Section 4.2.7.
- What trends emerged in students' quantitative and qualitative survey responses? Sections 4.3.1 and 4.3.2, respectively.

To collect data for this evaluation, our tool was deployed in CS3214 Computer Systems from 4/23/2024 to 5/7/2024. During this time, the 340 enrolled students worked on a web server project in C, deployed their server using Docker as an exercise, and then studied for their final exam. To aid in studying for finals and to collect data on questions for past units, students were allowed to ask questions any topic discussed in the course. To encourage participation, students were offered extra credit for each review up to 5 reviews, regardless

of participation in the study. At the end of the study, students were asked to complete an exit survey with 21 multiple choice, ranking, and checkbox questions and one optional open response field for extra comments. No extra credit was offered for completing the survey. Our study was approved by Virginia Tech's IRB on April 5, 2024 and our protocol number is 24-275.

For the web server project, students were given a set of features to implement on top of a minimal web server implementation provided to them. These features included serving static files, an API for authentication, streaming MP4 files, an API endpoint for listing available video files, supporting HTML5 fallback, handling both HTTP/1.0 and HTTP/1.1 requests, supporting multiple clients at once (i.e. multi-threading), and other miscellaneous robustness features. Implementing these features also required code to parse HTTP request headers, such as the "Cookie" header for authentication with JSON Web Tokens.

Deploying their web server with Docker required minor modifications to the server, such as supporting CSS and SVG resources. Students were provided a walk-through on the course website to build and deploy their server in a Docker container, including a pre-written Dockerfile.

The final exam included content from the second half of the course. Specifically, students were tested on concepts from memory management, virtual memory, networking, and virtualization. Students also had access to final exams dating back to 2009.

At any time during the study period, students could use a dedicated channel in the course's Discord server to interact with Disdoc's Discord bot and see the questions asked by other students. All students could read posts generated by their peers, but they had to indicate their consent status (even if they opted out of the study) to ask questions or give reviews themselves. In response to a question, the Disdoc Discord bot would generate a post with

the four most relevant chunks from course material, referred to as a "retrieval post." Then, if the student set the `use_llm` option to `True`, the bot would additionally use an LLM to generate an answer to the question with the retrieved course material chunks and stream the result in another Discord post. We refer to these posts as "LLM posts." The students could click a button on any post - even one triggered by another student's question - to review the post. These reviews will be distinguished as "retrieval reviews" and "LLM reviews" to correspond with the type of post they review.

All of the information included in LLM and retrieval posts, most importantly the four most relevant chunks, the similarity between each chunk and the student's question, and the LLM-generated answer if requested, were copied to disk by the Discord bot for future analysis. Additionally, above each relevant chunk in a retrieval post, Disdoc shows a link to the chunk's full document. Any time a student clicked on such a link, their click was recorded by our link forwarding service.

Lastly, at the end of the study period, a survey was published for students to complete. This survey focused on overall student perceptions, as retrieval and LLM review data already indicates student perceptions at the post level.

For the study, we configured the program to use GPT 3.5 (`gpt-3.5-turbo-0125`) with a maximum token length of 500 tokens. All other values were left at OpenAI's API defaults. The edit buffer for LLM posts (`/editbuffer`) was set to 0.3 seconds between edits.

We hosted Disdoc's Discord bot and RAG pipeline on a virtual machine with the following specifications:

- CPU: 2x Intel Xeon E5-2630 2.4GHz processors, limited to 4 cores
- GPU: NVIDIA Tesla T4 (16GB VRAM) with CUDA version 12.3

- Memory: 32 GB @ 2933 MT/s
- Storage: 100 GB virtual drive
- Ethernet: 10 GB interconnect

We hosted Disdoc's backend and frontend on another physical machine on the same network:

- Machine: Dell R640 1U server
- CPU: 2x Intel Xeon Gold 5218 2.3GHz 16-core processors
- GPU: None
- Memory: 384 GB @ 2933 MT/s
- Storage: 1 TB system drive
- Ethernet: 10 GB interconnect

The two machines share a network CephFS filesystem so the backend and Discord bot can share the same SQLite database. Due to the use of a network filesystem, write-ahead logging was disabled in all SQLite clients.

4.2 User Interaction Data

The following subsections analyze the data collected from students' interactions with the tool during the research study. Specifically, this data includes the reviews users submitted, the links that they clicked, the resources we retrieved, and tool performance during the study.

4.2.1 Study Participation

Of the 340 students who were enrolled in CS3214 at the time of the study, 227 participated in the study. They asked 1099 questions, where each student asked 4.84 questions on average with a standard deviation of 5.24 questions. While this may seem to suggest a connection to the 5 reviews extra credit incentive, this is likely coincidental. In reality, 81.50% of students asked at least 1 question and 47.58% asked at least 5 questions, so the majority of students were in the 0-5 question range. Rather, the average was raised by outlier students with extremely high usage, like one student who asked 53 questions. This is shown graphically in Figure 4.1, where the distribution of questions per student is skewed towards lower question counts.

When asking their questions, students overwhelmingly preferred to have an LLM-generated answer: of the 1099 questions asked, 1067 of them had the `use_llm` option enabled. It is possible that students overwhelmingly enabled the LLM-generated answer feature because disabling it would lower the amount of information available to them or because of the novelty of LLMs. Regardless, this 33x higher preference for LLM-generated answers is extremely strong.

On these posts, students submitted 1318 reviews. Similar to questions, there was a skew towards reviews of LLM posts: 72.76% (959) of the reviews were on LLM posts while and 27.24% (359) of them were on retrieval posts. Notably, Disdoc always shows a retrieval post regardless of `use_llm`'s value, so this skew is purely driven by student bias.

In the end, 75.16% of all questions were reviewed at least once disregarding review type. Further, 69.73% of all LLM posts were reviewed at least once and 28.03% of all retrieval posts were reviewed at least once. Engagement on retrieval posts could be lower because there is no way to opt out of them.

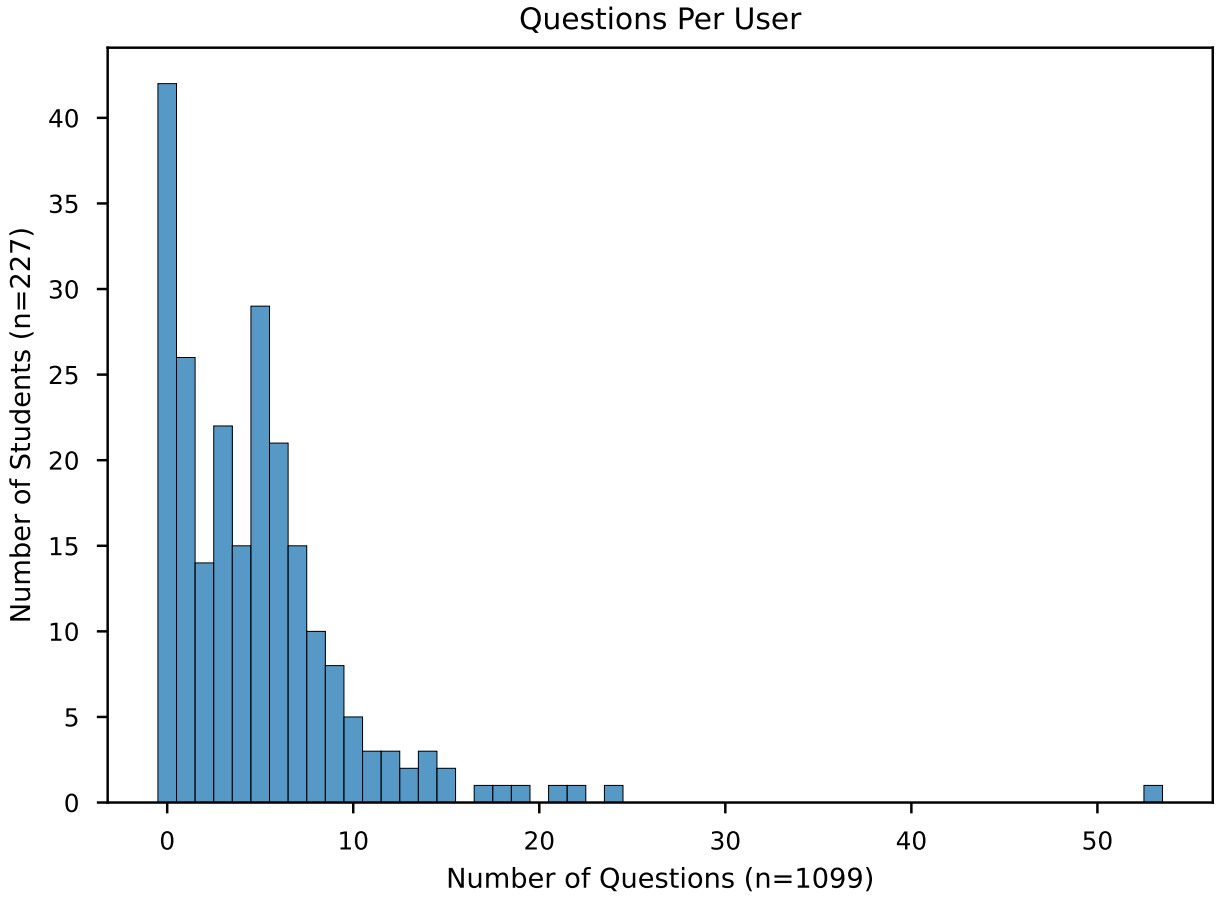


Figure 4.1: Plot of the number of students who asked each number of questions.

Unsurprisingly, most posts were reviewed at least by their own authors: 86.83% of reviewed LLM posts and 68.51% of reviewed retrieval posts were reviewed by their authors. Additionally, 64.92% of reviewed LLM posts and 56.17% of reviewed retrieval posts are only reviewed by their author. Indeed, the relationship between the number of questions asked and the number of reviews left by a student is statistically significant ($p = 6.37 \times 10^{-11}$). However, while reviewing the posts of peers was unpopular, we do not have the data to conclude that students did not read posts generated by their peers altogether; Discord does not provide such data through their API.

All in all, 97.80% of students left at least 1 review and 80.62% of students left at least

5 reviews. Interestingly, 43.18% of students left more than 5 reviews, meaning that many students were engaging with the tool beyond the extra credit incentive. In fact, when looking at the distribution of total reviews shown in Figure 4.2, 5 reviews appears to be an inflection point and we see that 43.18% of users left more than 5 reviews. So, while students were likely motivated by the extra credit incentive, a great number students also willingly used with the tool beyond the 5 reviews needed to maximize extra credit.

When comparing the number of retrieval and LLM reviews each student gave, shown in Figure 4.3, we again see student bias towards the LLM component. Specifically, the majority of students gave 3-5 LLM reviews, while the majority of students gave 0-1 retrieval reviews.

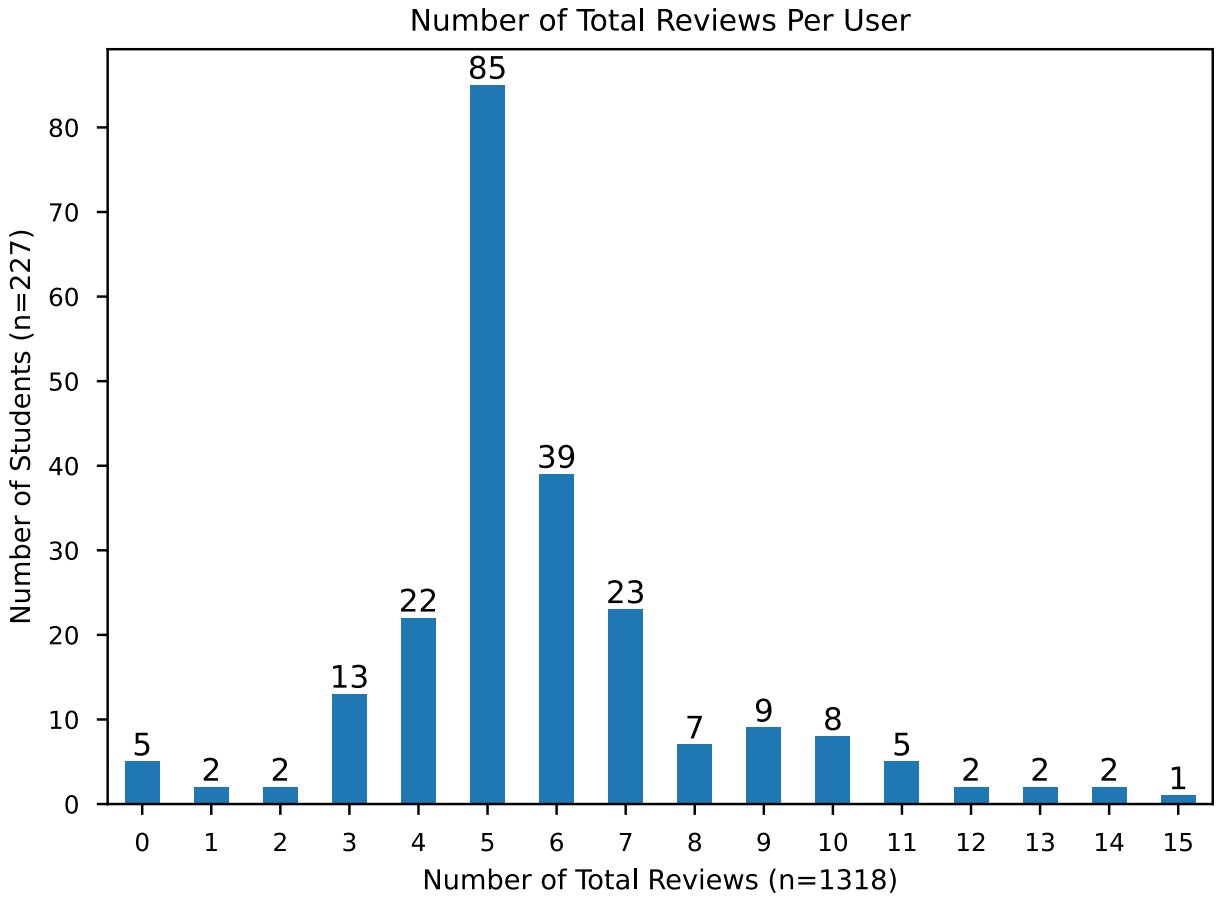
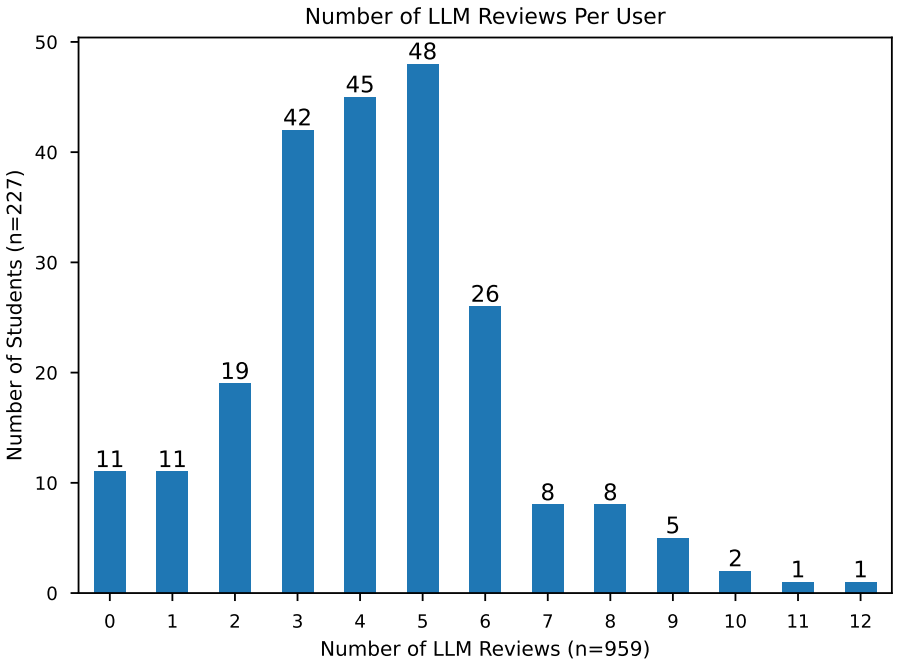


Figure 4.2: Number of students who left each number of reviews.

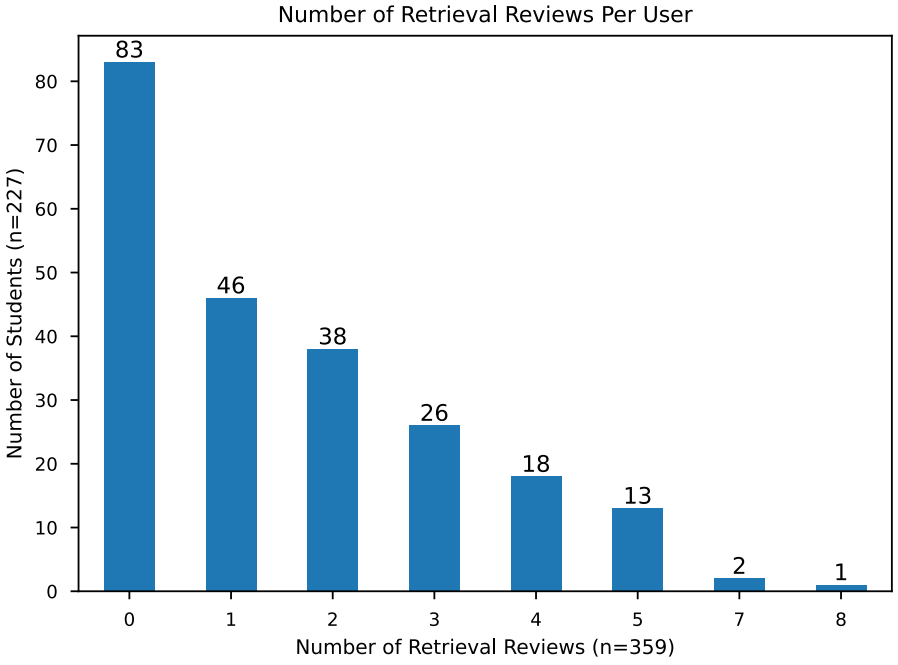
However, the number of students who gave 0-4 retrieval reviews or 0-4 LLM reviews is far larger than the number of students who gave 0-4 total reviews. This indicates that students in the 0-4 retrieval or LLM review range were mixing review types to achieve a higher total review count.

On the other hand, gauging how users engaged with our tool over the course of our study is difficult because we did not record timestamps on students' questions and reviews. Instead, we use the auto-incrementing row IDs from our SQLite database, which show relative order, as a proxy.

Thus, we can assign each student a color from a gradient based on when they asked their first question (their first question's ID), then draw a vertical bar in that color on a timeline whenever they asked a question. An even gradient across such a timeline would show highly clustered, short-term usage; a chaotic mix of colors would show repeated, longer-term usage. Our timeline, presented in Figure 4.4, has a mix of these two characteristics. In other words, some students used the tool for a short period of time and never again, while others used the tool repeatedly over the course of the study. Overall, there is an average of 270.00 and standard deviation of 298.19 posts between a student's first and last post.



(a) LLM reviews



(b) Retrieval reviews

Figure 4.3: Number of students who left each number of reviews, separated by review type.

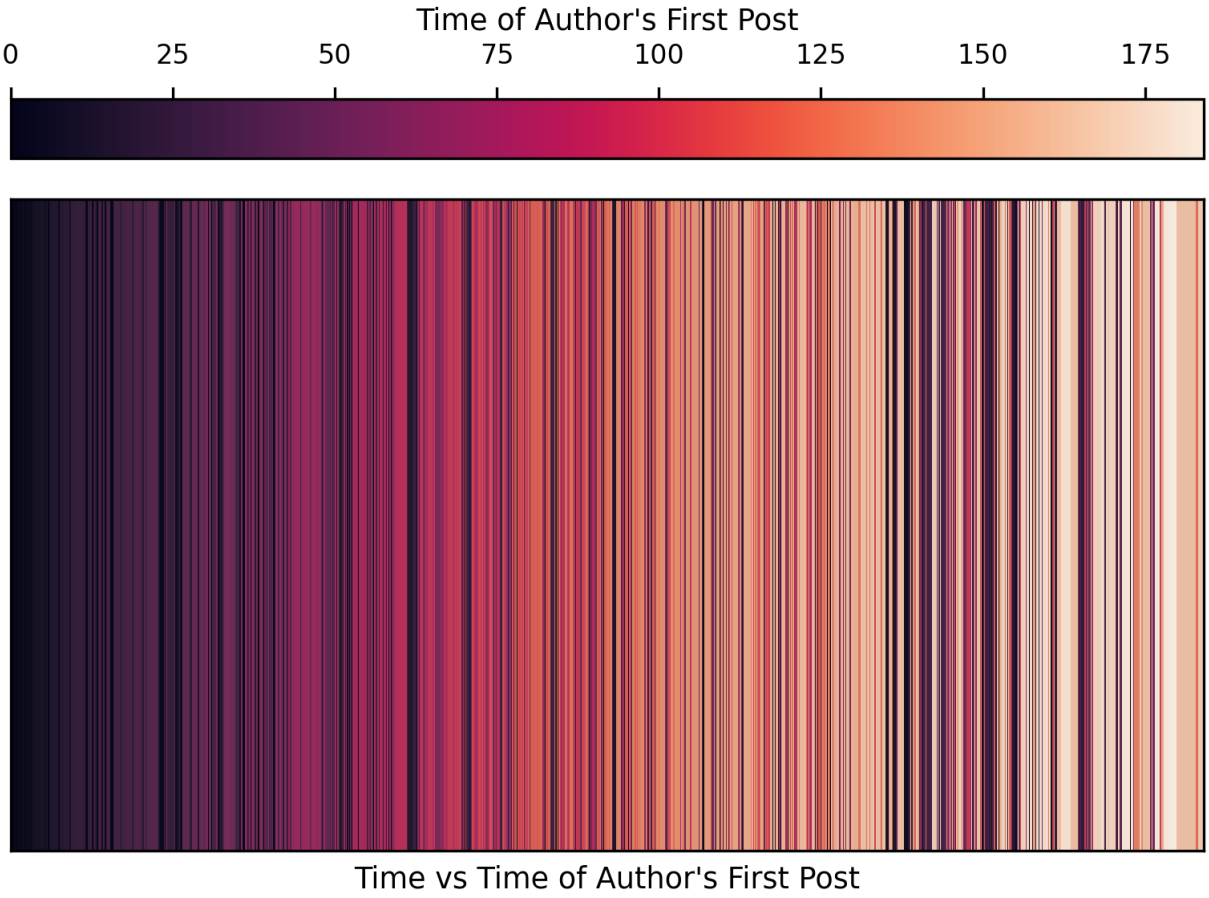


Figure 4.4: Timeline of when students asked questions. Vertical lines represent a student's question, uniquely colored based on when that student asked their first question. A uniform gradient would indicate clustered use, while a chaotic mix of colors would indicate repeated use during the study.

4.2.2 Link Click Data

Overall, students rarely clicked the links to each chunk's full document in retrieval posts. Of the 4396 links generated during the study, only 67 unique links were clicked for a total of 112 link clicks. Additionally, only 56 of the 227 students (32.75%) ever clicked a link. This translates to an overall clickthrough rate of 2.55%. We propose the following possible explanations for this behavior:

1. Students are satisfied by the previews of retrieved chunks embedded in retrieval posts. This is believable given that the majority of previews are well formatted: we syntax highlight code snippets, annotate PDFs, and display text content. However, some chunks need more context than the 500-character limit to be helpful and others (like those from webpages) are poorly formatted and easier to read online. Unfortunately, Discord does not indicate when a user reads parts of a post, which is necessary to validate this theory.

2. Students prefer the LLM-generated answer over the list of relevant chunks. This is supported by other areas of the data, particularly student's overwhelming preference to enable `use_llm` when asking questions. If this were the case, it would raise concerns that students may not be investigating the relevant chunks to verify the LLM's answer, blindly trusting answers instead.

3. Students prefer opening the resource themselves. Because students are building on the code that we occasionally link to and typically have the instructions for the assignment open as they work, they might not want to click on a link to a new webpage; they could open the file in their own code editor or look at the already open PDF rather than in a new tab. The list of clicked links indirectly supports this hypothesis: most of

the clicked links are resources like lecture presentations, FAQ pages, and past exams which a student is unlikely to have open or quick access to.

While each student only clicked 0.493 links on average, there is a fair amount of variance at a standard deviation of 2.425 link clicks. In fact, if we omit the users who never clicked a link, the average raises to 2.21 clicks per user. Looking at the distribution of the number of link clicks in Figure 4.5, we can see that the majority of students that clicked a link only clicked a few of them. However, a subset of students did engage with the relevant links more heavily.

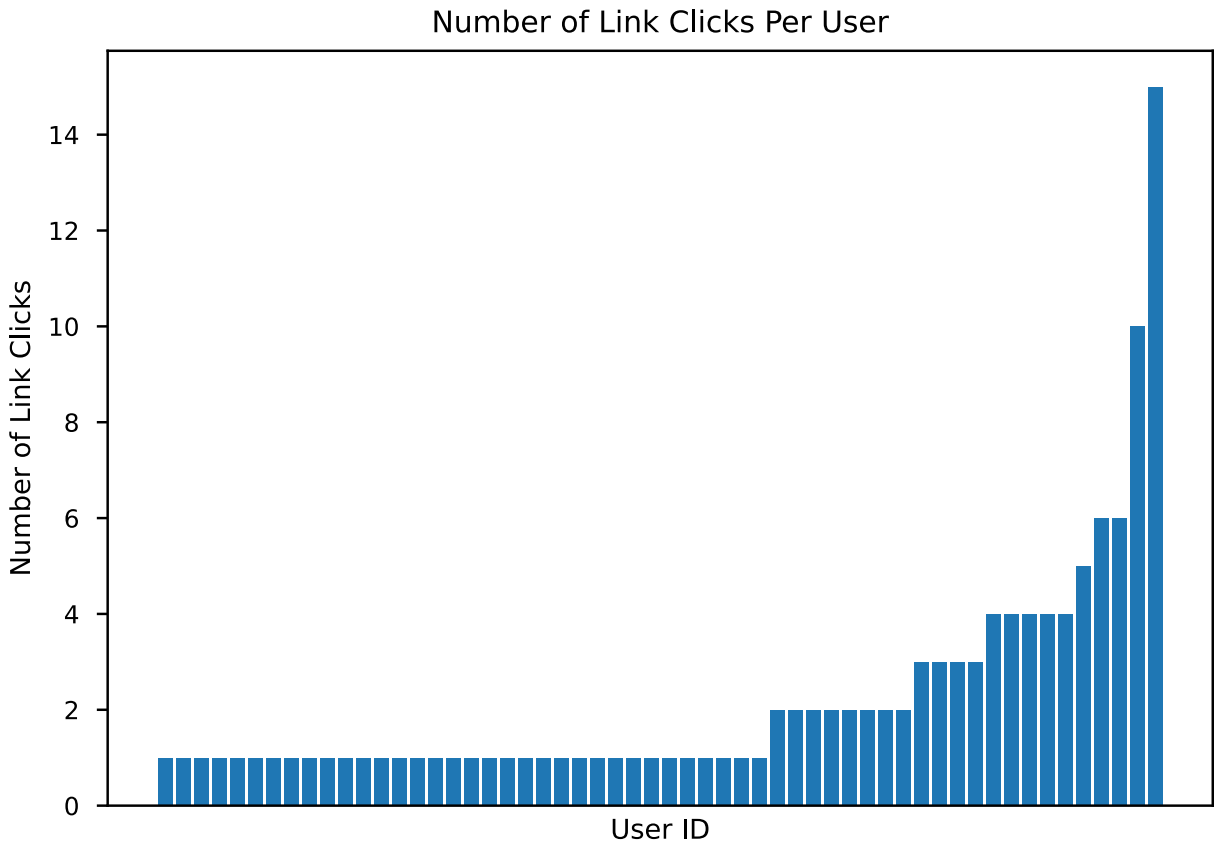


Figure 4.5: The number of links clicked by each user. Users with no link clicks are omitted for brevity.

Manual inspection of the activity of the top 3 students reveals that they primarily clicked links to past final exams, likely while reviewing for the final exam. While the top 10 students varied in usage more, half clicked the FAQ web page for the web server project at least once. Based on office hours, this is a resource that students often do not know about, suggesting that the relevant links were helpful for exposing some students to hard to find resources.

Lastly, there is a difference between link click activity on LLM and retrieval posts. The click-through rate for questions with an LLM-generated answer is 2.53% while the clickthrough rate for questions without one is 3.13%. However, note the significant disparity in the data: there are 108 clicks on the 1067 posts with an LLM-generated answer but only 4 clicks on the 32 posts without. As a result, while questions with only a retrieval post have a clickthrough rate that is 1.235x higher than those with an LLM post, there is not enough data to say that this relation is meaningful.

4.2.3 Performance

The average time to retrieve the top 4 most relevant chunks for a question was 366ms and the amount of time to post the full LLM's response was 199ms. Most values were below the 400ms Doherty threshold [19], meaning the delay should have little impact on student perceptions of Disdoc; only 9.54% of retrieval posts and 0.09% LLM posts took longer than 400ms. Notably, these values are upper bounds, as they do not factor in the nearly instantaneous progress indicators Disdoc shows or that Disdoc streams the LLM's output. Altogether, Disdoc's performance appear to be sufficient for students: there is no statistically significant connection between any of the timings and student's review values.

The distributions for retrieval, generation, and total times are visible in Figure 4.6. It is clear that the retrieval step takes longer than the generation step in most posts. In our

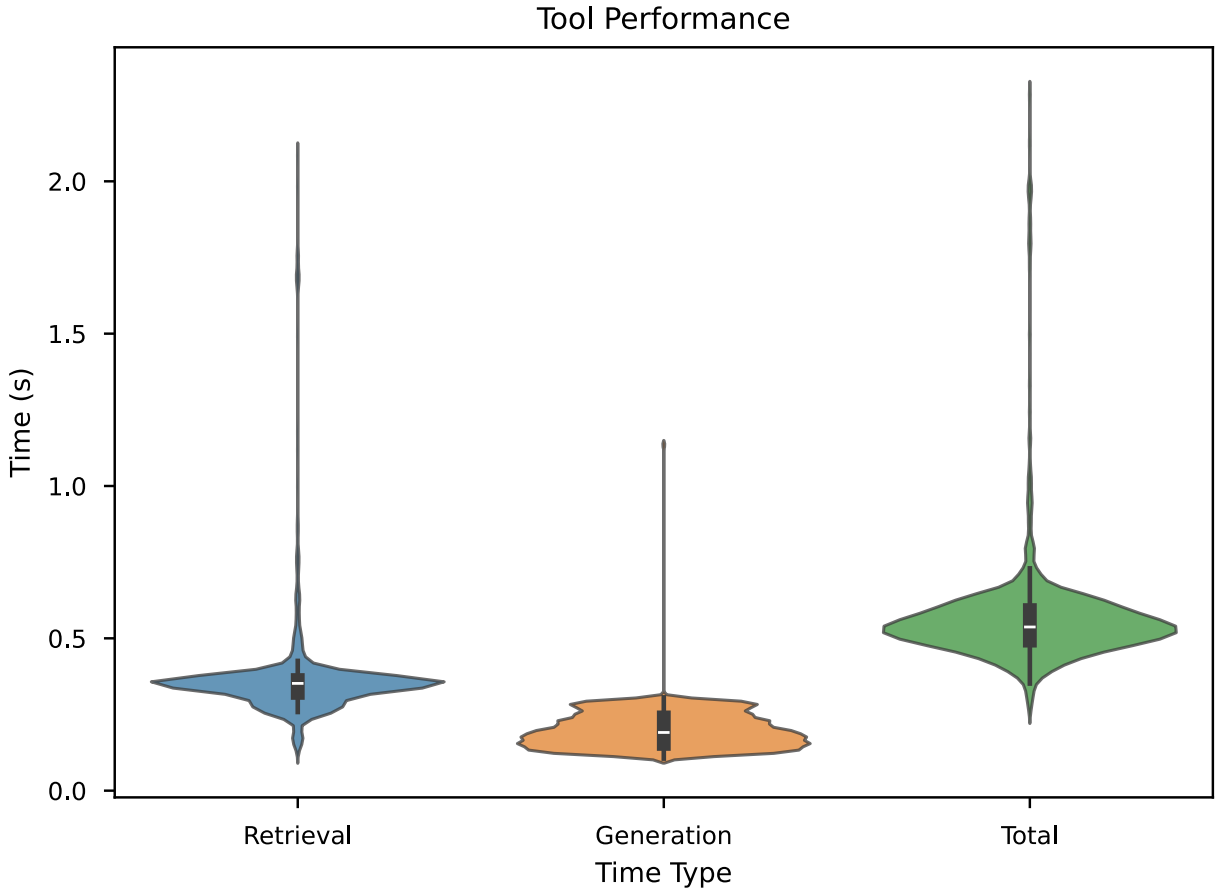


Figure 4.6: Distributions for the amount of time it took for Disdoc to retrieve the top 4 most relevant chunks for a question (but not post them), start streaming and finish posting the last token of the LLM's answer to the question, and to do both.

use case, this is largely due to the metadata filtering performance bottleneck in the Chroma database. Additionally, while all three times exhibit tight clustering around the median, there are several outliers in the data. Almost all outliers in the total time data are caused by poor retrieval performance, with only one caused by the generation step. More work needs to be done to diagnose these performance regressions.

4.2.4 Review Data

We plot the distribution of the relevance, helpfulness, and correctness metrics in Figure 4.7. Relevance and helpfulness range from -2 to 2, while helpfulness ranges from -3 to 3. Given that the spikes at high values are larger than those at lower values, we see that students gave notably more positive reviews than negative. And indeed, when looking at the averages, we see they are all above 0: relevance is 1.22, helpfulness is 0.66, and correctness is 1.40. However, the increase in ratings is not strictly linear; there is an isolated spike at 0 correctness among an otherwise monotonic increase in number of ratings correctness.

In order to understand why there is a spike at zero correctness, we must look for trends in the combinations of ratings shown in Figure 4.8 instead of isolated distributions. In these graphs, we see a disproportionate number of reviews with 0 for correctness and 0 for relevance as well as -2 to 1 for helpfulness. In most reviews in this range, the LLM gives a partial answer or refuses to give one at all because the information needed to answer it is not included in the context. For example, when asked about the location, date, and time of the final exam in Figure 4.9, the LLM responds that it was not provided that information. In this case, the LLM is correct because the information had not been published. But when the information does exist but is not retrieved and included in the LLM's prompt, this represents a failure in the retrieval step.

Looking back at Figure 4.8, we can extract more trends. Namely, while relevance and correctness seem to roughly correlate, the same is not true of relevance and helpfulness or helpfulness and correctness. Rather, there are a number of reviews with high relevance but low helpfulness and reviews with low helpfulness but high correctness. Interestingly, the opposite relations are not true.

We see trends similar to the LLM ratings in the distribution of retrieval ratings, shown in

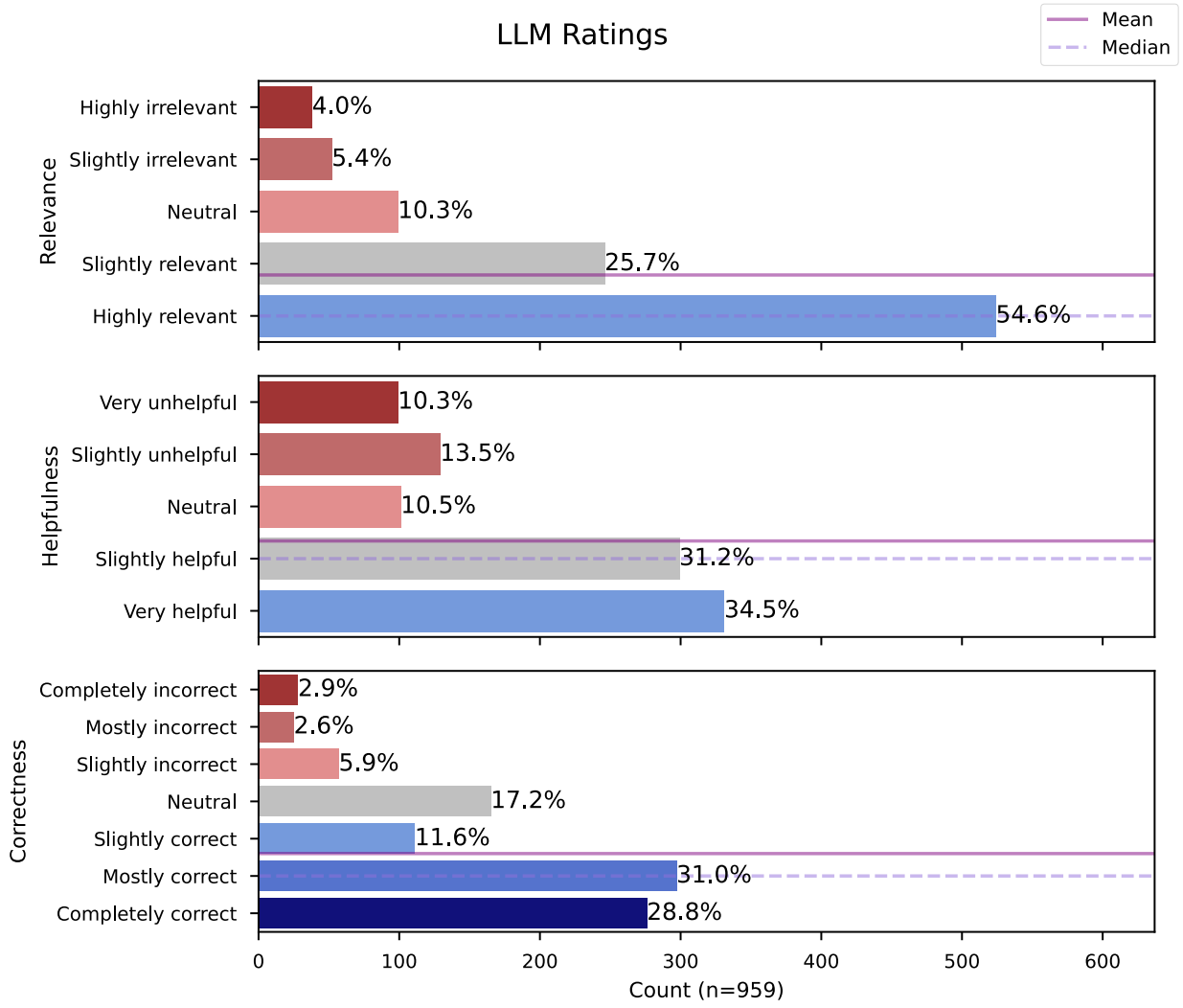
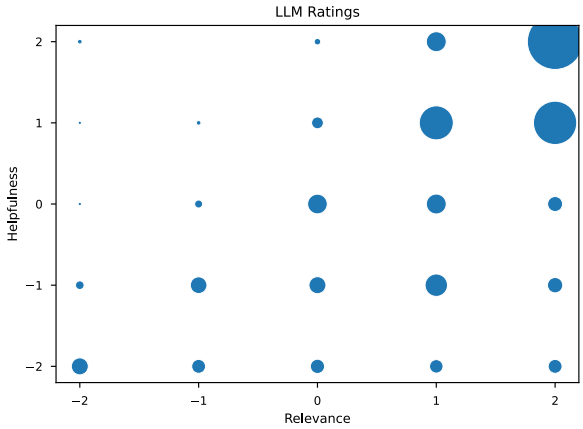
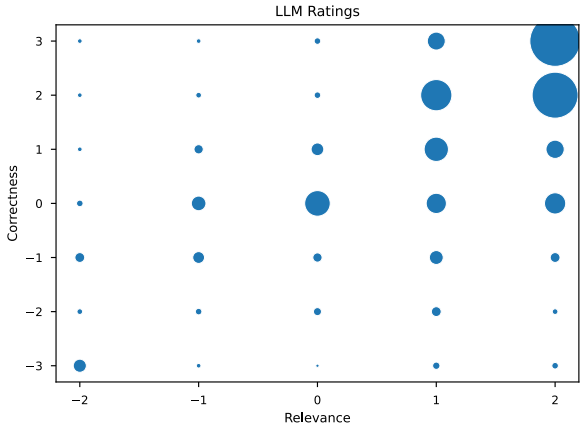


Figure 4.7: Plot of the student perceived relevance, helpfulness, and correctness of LLM posts, along with the mean and median values for each.

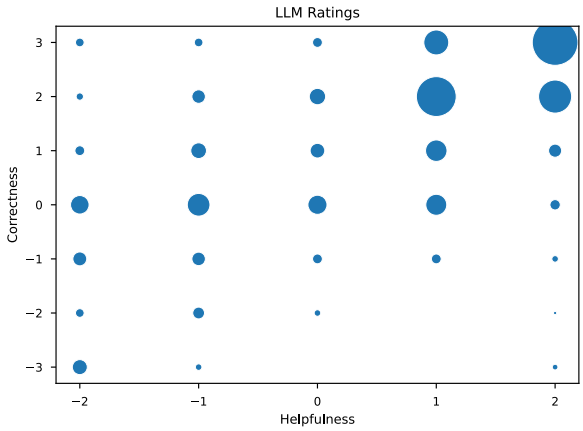
Figure 4.10. First, there is also a larger number of positive ratings, indicated by the spikes at higher values. Again, the average rating values are positive, but they are also lower than the average LLM values: 0.79 for relevance and 0.58 for helpfulness. While retrieval's average helpfulness is nearly the same as LLM reviews' average helpfulness, retrieval's average relevance is notably lower because the reviews are more evenly distributed among the different relevance values (i.e. there is no longer a spike at 2 for relevance).



(a) Relevance vs. Helpfulness



(b) Relevance vs. Correctness



(c) Helpfulness vs. Correctness

Figure 4.8: The number of reviews on LLM posts with different combinations of values. The size of a bubble represents the number of times students left a review with the corresponding combination of values.

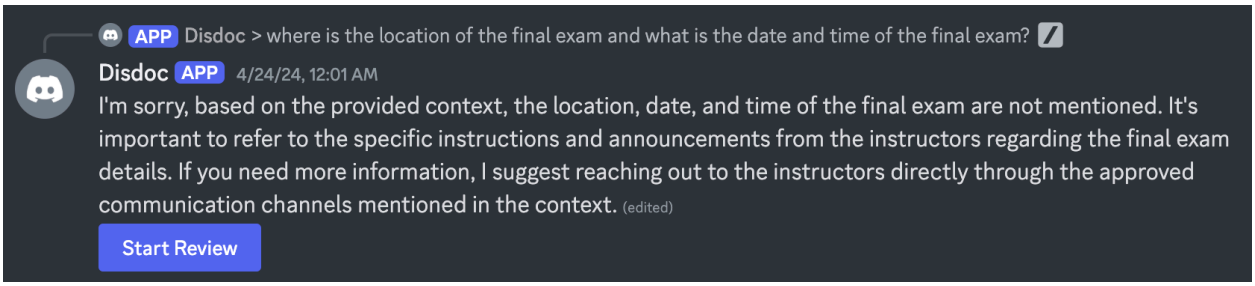


Figure 4.9: Example response which was reviewed with relevance=2, helpfulness=-2, and correctness=0.

There is also a small spike at 0 for helpfulness in the retrieval ratings. When looking at the number of times each combination of helpfulness and relevance values occurs in the dataset, as depicted by Figure 4.11, we see that low helpfulness often coincides with low relevance. While this is understandable, upon manual inspection of questions with helpfulness=0, some patterns emerge. These patterns are listed in descending order of frequency:

1. Questions about resources not included in the corpus: "where is the location of the final exam and what is the date and time of the final exam?," "How to find a good project partner?"

2. Incorrect ratings: the most relevant resource was correctly included in the context but the student still rated the helpfulness poorly

3. Retrieval failures: chunks from a relevant resource were not retrieved, despite being in the course's corpus

4. Questions that are too vague: "In the 4th question about Linking, there are Symbol Types, read that question and give me all the types."

5. Incorrect category: "How is our grade calculated?" with `lecture` instead of `admin`

Of these, only retrieval failures represent a real issue with our tool. Regardless, relevance

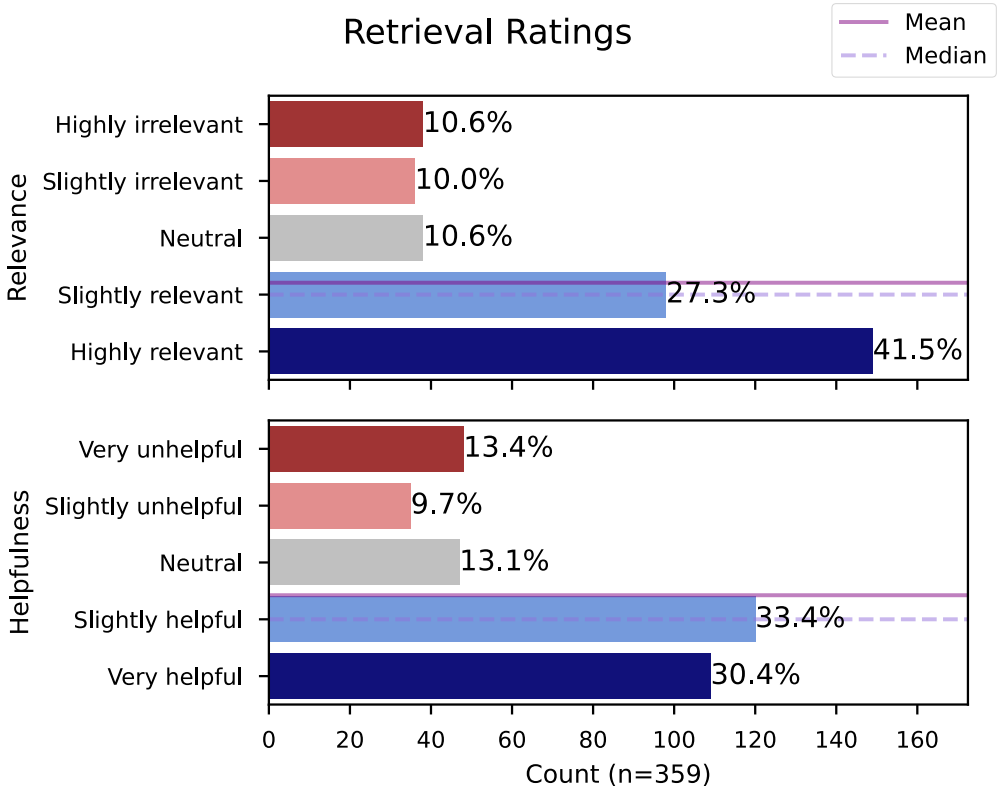


Figure 4.10: Plot of the student perceived relevance and helpfulness of retrieval posts, along with the mean and median values for each.

and retrieval exhibit a roughly linear relationship in Figure 4.11.

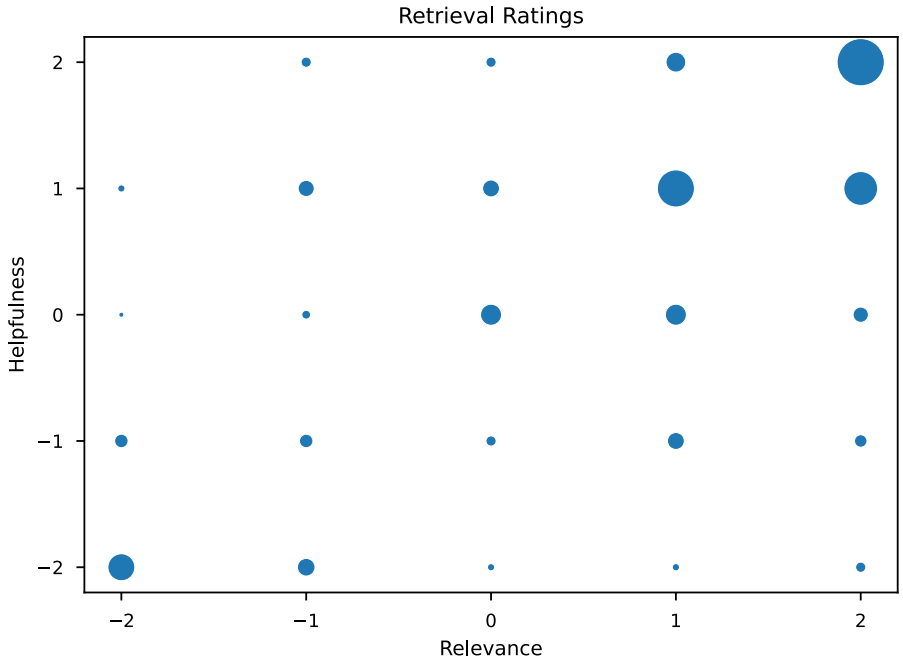
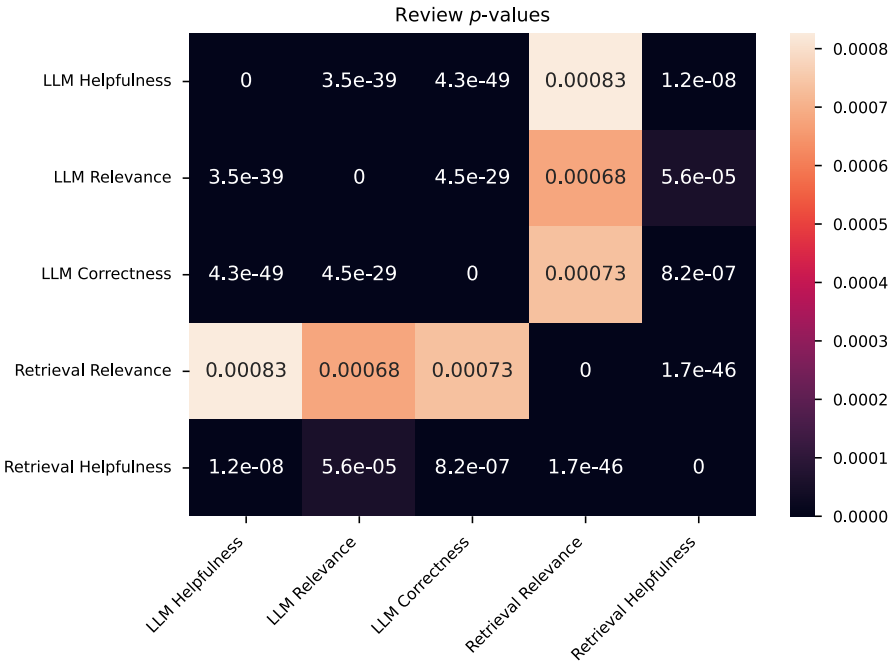


Figure 4.11: The number of reviews on retrieval posts with different combinations of values. The size of a bubble represents the number of times students left a review with the corresponding combination of values.

4.2.5 Cross-Metric Relationships

We can also look at the correlation between different metrics. This allows us to examine whether ratings for metrics within a group (e.g. retrieval relevance and helpfulness) correlate, but also whether metrics across groups (e.g. retrieval relevance and LLM helpfulness) correlate. We expect retrieval and LLM metrics to be highly correlated, as a successful retrieval step will provide the LLM course-specific information it can use to generate a high-quality answer. Because each metric has a discrete distribution, we use Spearman rank correlations to measure significance and correlation based on monotonicity rather than linearity.

A heatmap of the p-values and Spearman rank correlation coefficients for each metric are displayed in Figure 4.12. All p-values are below the significance threshold of 0.05, so all cross-metric correlations are statistically significant.



(a) P-Values

(b) Spearman rank correlation coefficients

Figure 4.12: The p-values and Spearman rank correlation coefficients between each metric of the review data. All p-values are below the significance threshold of 0.05.

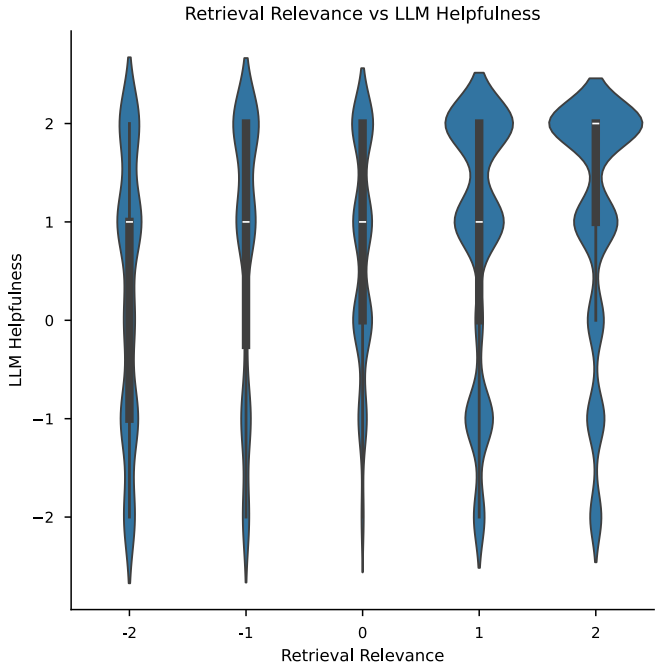
Evidently, each metric is highly, positively correlated to other metrics within their group with coefficients between 0.54 to 0.67. The order imposed by the coefficients is also notable, as it suggests that an LLM's correctness is a slightly better predictor of helpfulness than relevance. There is only one cross-metric relation for retrieval (i.e. retrieval relevance and helpfulness), but it is almost as highly correlated at 0.66 as the highest LLM cross-metric correlation, LLM correctness and helpfulness, at 0.67.

However, there is a far lower correlation between metrics of different groups. All intergroup correlation coefficients fall in the range of 0.17-0.29, which is far lower than the range of 0.54-0.67 for intragroup relations. This is the opposite of our hypothesis that retrieval metrics strongly correlate with LLM metrics. But indeed, visually inspecting plots of intergroup relations shows that the range of values is more evenly distributed and therefore less monotonic than intragroup relations.

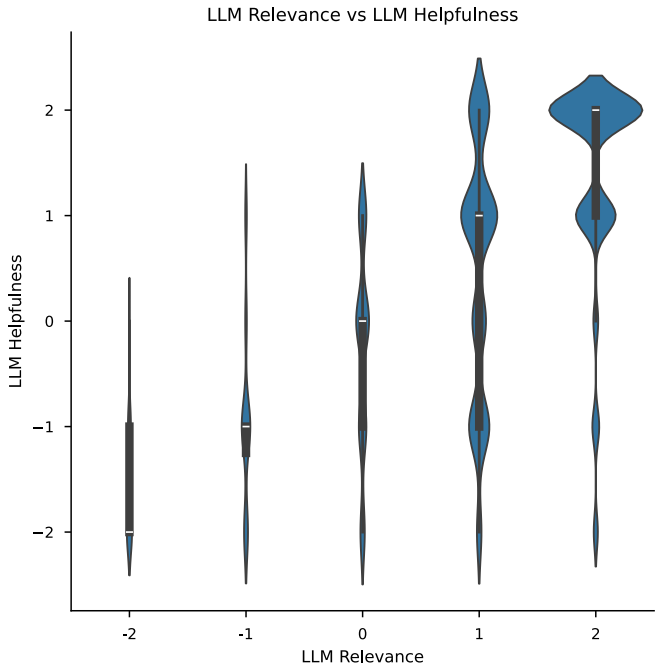
A graph of an intergroup relation, retrieval relevance to LLM helpfulness, is shown in Figure 4.13a and exhibits these characteristics: there are a great number of extremely helpful reviews, even on posts rated low in relevance. On the other hand, graphs of intragroup relations exhibit a clear visual monotonicity, sometimes to the point of linearity, with one example shown in Figure 4.13b for LLM relevance vs. helpfulness.

It is unclear what causes the evidently lower correlation between intergroup relations. We suggest two possibilities:

- Despite a poor retrieval step, the LLM is occasionally able to generate a high-quality response. This is likely dependent on whether the LLM is able to answer the question using solely information is learned from training, or whether it needs more context to understand the question or solution.
- Students were giving falsely poor ratings on retrieval, either because they did not



(a) Intergroup: Retrieval Relevance and LLM Helpfulness



(b) Intragroup: LLM Relevance and LLM Helpfulness

Figure 4.13: Violin plots of the distribution of values for example inter and intragroup relations. The area of the plots scale with the number of observations.

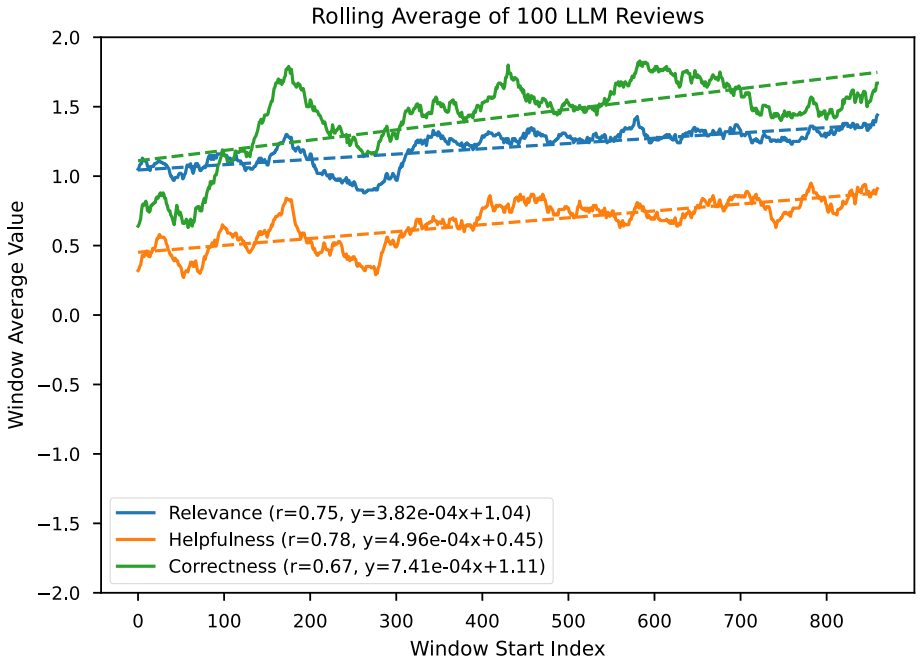
understand the retrieved chunks or because there were simply no relevant documents in the course to pick chunks from. This possibility could be confirmed or denied by comparing to an expert's reviews, an area we highlight for future work.

4.2.6 Changes to Ratings Over Time

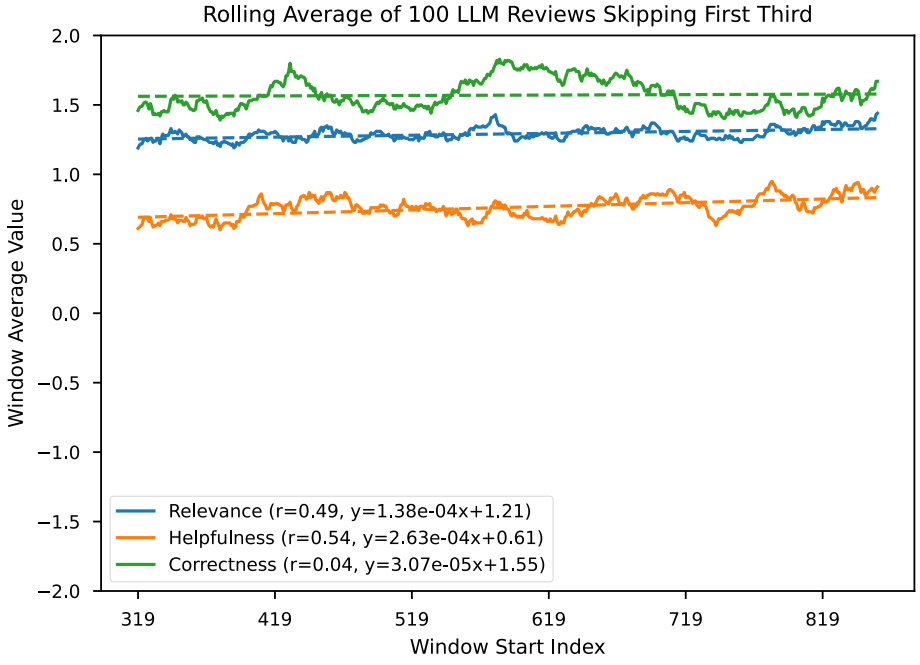
Because we did not record timestamps, it is difficult to accurately measure any connection between ratings and time. Our best approximation is using a rolling average with a window of 100 reviews at a time. In our analysis, we separately examine ratings over time with and then without the first third of the reviews, in an attempt to isolate the effect of an initial, course-wide learning curve. While an individual student's learning curve will still be present if they leave their first review after this last third, it will be mixed with experienced students' reviews, creating a less statistically notable impact.

We plot the rolling averages for the LLM metrics in Figure 4.14a and Figure 4.14b, respectively. When looking at the full data, all trend lines are statistically significant and positive, indicating an improvement in skill, question types, or perceived tool performance. However, when we omit the first third of data, the correctness line is no longer statistically significant and there is only a very slightly positive trend for relevance and helpfulness. Clearly, the change in students' ratings during the first third of data contributed greatly to the overall trends, suggesting a bias in early raters, learning curve, or change in question types over time.

We also plot the rolling averages for the retrieval metrics with and without the first third of data in Figure 4.15a and Figure 4.15b, respectively. Note that we have considerably fewer retrieval reviews (359) than LLM reviews, so the rolling window size of 100 is a greater portion of the data. Also unlike the LLM reviews, all trends in the rolling average data are



(a) All

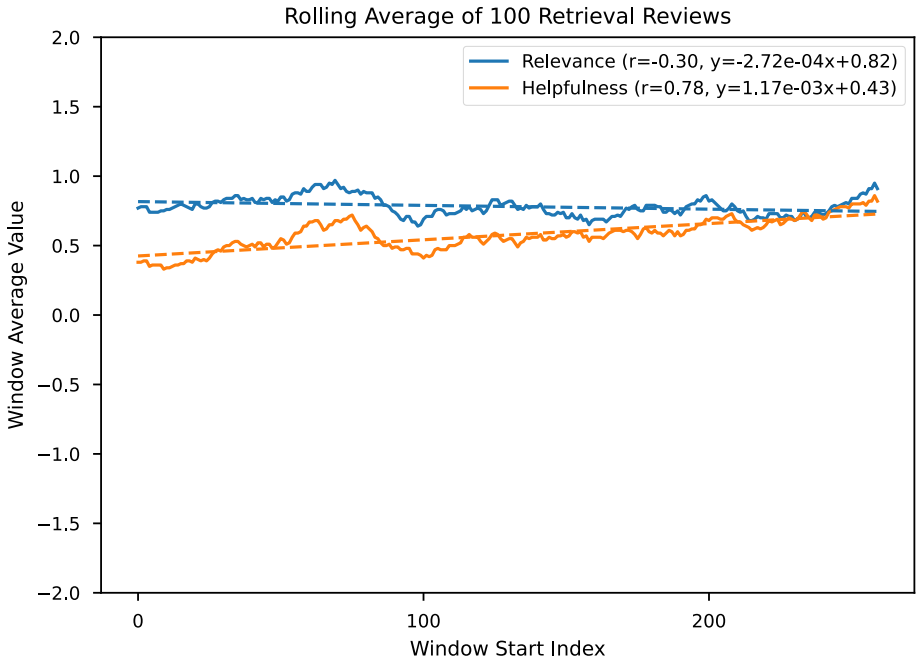


(b) Skip first third

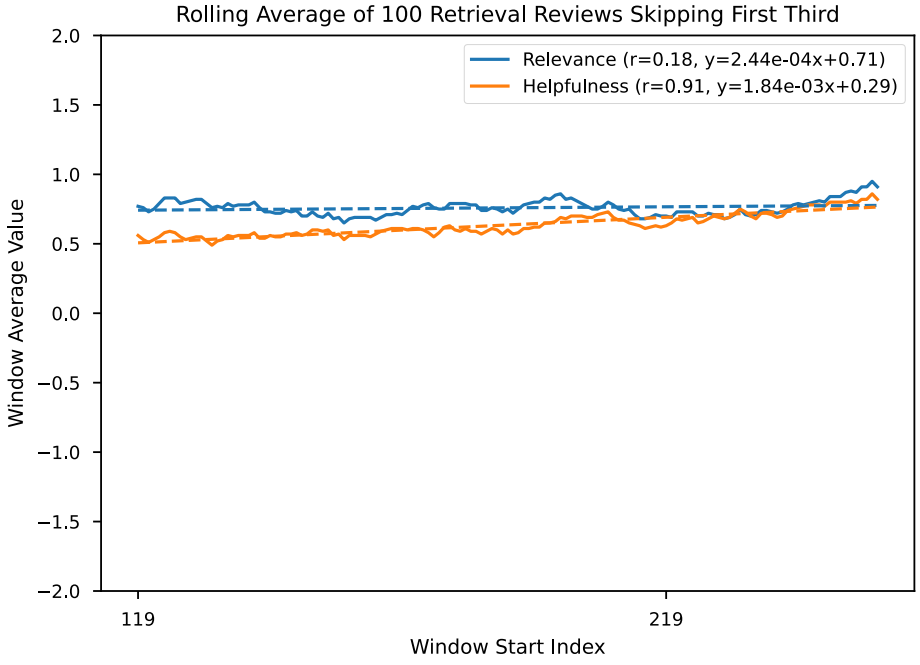
Figure 4.14: Rolling average of LLM metrics with a window size of 100 reviews using either all of the data or without the first third.

statistically significant, regardless of whether we include the first third. Additionally, when looking at the full data, relevance is negatively correlated with time, while helpfulness is positively correlated. There is not enough data to indicate why the relevance is negatively correlated with time in the full data, but it is possible that it lowered as student questions about the web server project became more specific or difficult, then spiked up as they started asking easier questions about final exams at the end of the study. On the other hand, helpfulness always has a strong positive correlation with time. Similar to relevance, we do not have enough data to explain this correlation: users could perceive answers to their questions as more helpful as their questions got trickier, or the later ratings were inflated by the tool's good performance on questions related to final exams.

Unfortunately, while we observe some spikes in the data, we are unable to attribute them to specific events (e.g. the start of a project or assignment) without exact timestamps. Additionally, without collecting data during more assignments, we cannot determine whether there is a common pattern to the changes in metrics as students progress through assignments.



(a) All



(b) Skip first third

Figure 4.15: Rolling average of retrieval metrics with a window size of 100 reviews using either all of the data or without the first third.

4.2.7 Retrieved Links

As Disdoc responds to each student question, it saves a record of the question, the relevant chunks it retrieves, and an answer to the question if one was generated. Using these records, we can look at what resources Disdoc retrieved for RAG and for students during the study. Additionally, we can look for trends at the level of chunks (the document snippets that the students see), documents (each file comprised of one or more chunks), types (file extensions), and categories (groups of documents, like lecture material or project documentation).

When looking across the retrieved chunks, we see that the average cosine similarity is 0.462. When averaging the similarity across all chunks in a document, we find the average document similarity is a bit lower at 0.411. The exact distribution of chunk and document similarities can be seen in Figure 4.16. Interestingly, 48.83% and 51.17% of chunks are above and below the average cosine similarity of their document. In other words, only chunks from around half of each document are likely to be retrieved in response to a question.

When looking at the amount of times each chunk was retrieved in Figure 4.17, we see that there were indeed chunks of high value and those of little value. But, while there are chunks that are retrieved often, the distribution is not highly skewed. Upon manual inspection, the top chunks are mostly from the rules sections of final exams, web server project instructions, and web server project unit tests. It is not entirely clear why the rules sections from various final exams are retrieved so often. One possibility is that they contain many phrases about final exams and are falsely identified as similar to any questions about the final exam as a result.

By comparison, the disparity in the recommendation count for each document is much higher. In fact, the recommendation counts for documents fit a Zipf distribution. The distribution of the counts and the fitted Zipf distribution are displayed in Figure 4.18. On average, a

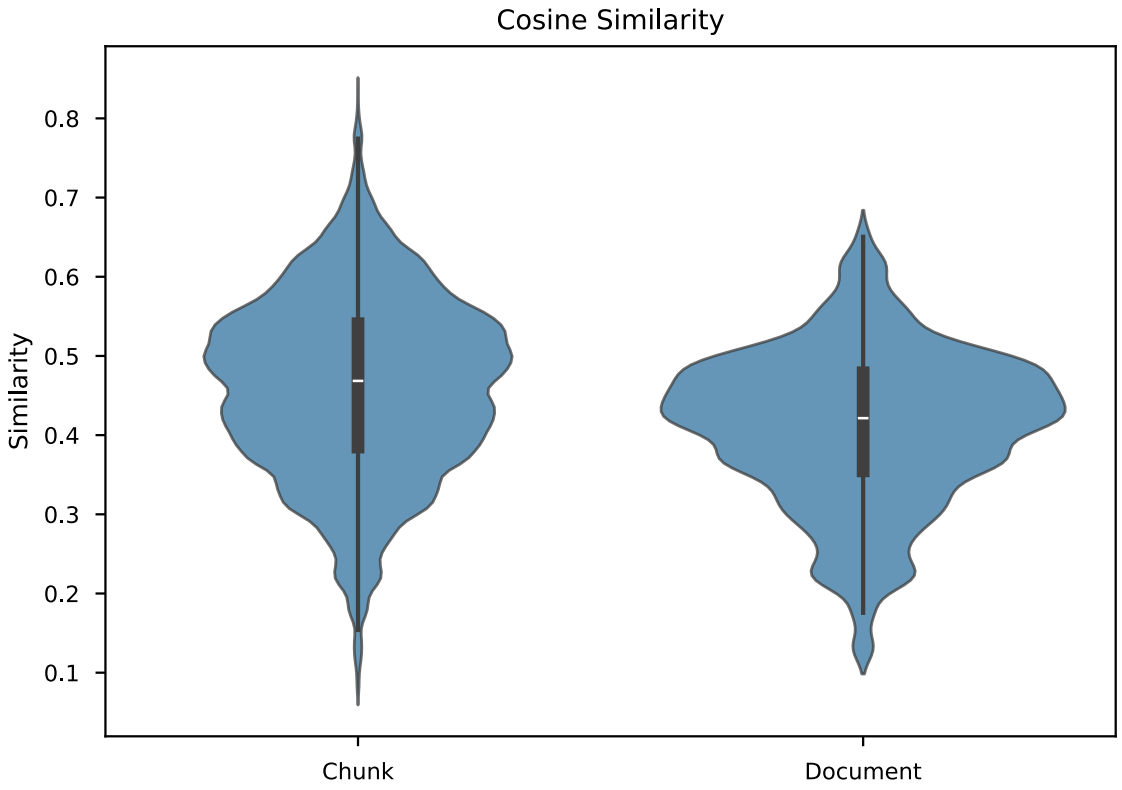


Figure 4.16: The average cosine similarity of each chunk, according to scores when each chunk was used in a response, and the average cosine similarity of each document, based on the similarities of its comprising chunks.

document was included in 1.19% of posts. When a document was included, Disdoc typically retrieved multiple of its chunks: when a document was included in a response, an average of 2.40 chunks was included with a standard deviation of 0.97. This results in chunks from an average of 2.49 documents in each response.

Surprisingly, the top 10 most retrieved chunks are largely missing from the top 10 most retrieved documents. Rather, the top 10 documents have their recommendation count spread across multiple of their chunks, as shown by the distribution of colors in Figure 4.17. The top 10 documents are fully listed in Table 4.1, where the only top 10 documents that also have a top 10 chunk are the web server project instructions and web server unit tests.

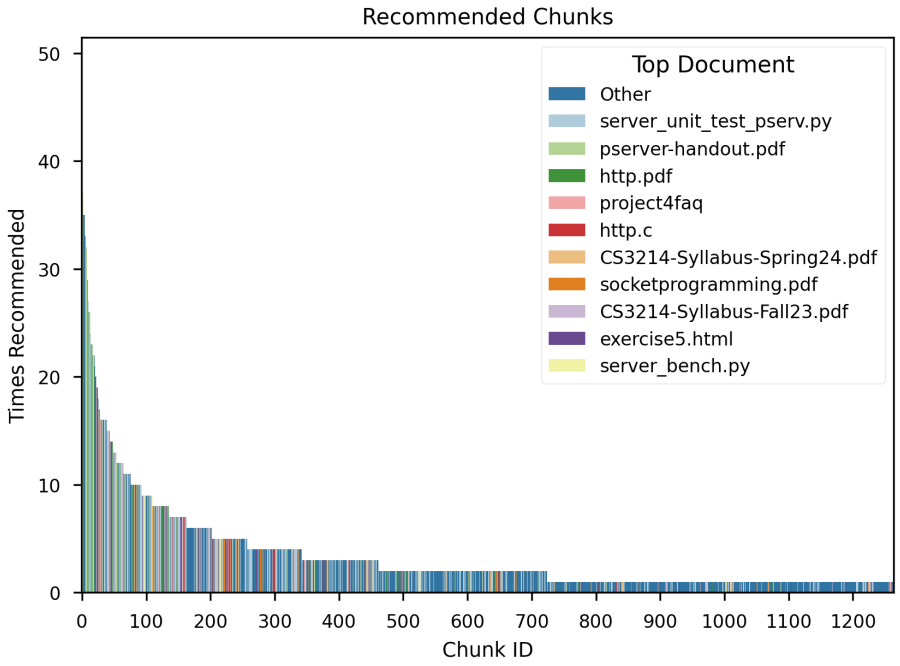


Figure 4.17: Number of times each chunk was retrieved. Chunks from the top 10 most retrieved documents are specially colored, with chunks from all other documents colored blue.

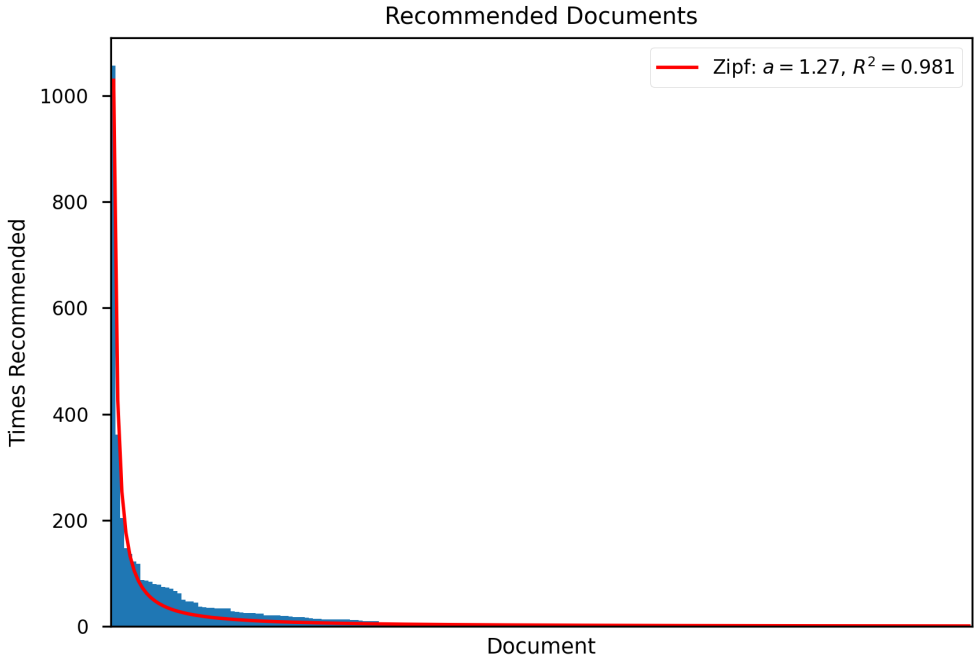


Figure 4.18: Number of times each document was retrieved.

Document	Description	Times Retrieved
`server_unit_test_pserv.py`	project unit tests	1056
`pserver-handout.pdf`	project instructions	361
`http.pdf`	lecture slides	204
`project4faq`	project FAQ	148
`http.c`	project starter code	137
`CS3214-Syllabus-Spring24.pdf`	syllabus	123
`socketprogramming.pdf`	lecture slides	118
`CS3214-Syllabus-Fall23.pdf`	syllabus	88
`exercise5.html`	exercise instructions	87
`server_bench.py`	project unit tests	84

Table 4.1: List of the top 10 most retrieved documents.

When looking at the top documents overall, particularly at the inclusion of the web server project FAQ and several lecture slides which students are often unaware of when they attend office hours, the tool appears to perform well at exposing students to high-value documents that they might not have read. Also, `server_unit_test_pserv.py`, which contained the web server project tests, was retrieved so many times because it was validly included whenever a student asked about how to fix an error or failed unit test: one or more chunks for the unit test causing them to fail was typically included. This type of question was extremely common and one or more chunks from this unit test file were included in 36.78% of all responses. On the other hand, a past semester's syllabus was mistakenly included in the CS3214 course corpus and recommended enough times to be the eighth most recommended document.

We can also see how often particular resource types were retrieved for students, as shown in Figure 4.19. PDFs (assignment specifications, lectures slides, help session slides, and more) were the most popular resource type, followed by Python (unit tests), HTML (CS3214 course website pages), and C (starter code). The rest of the document types were increasingly niche and not retrieved much. For most document types, the total recommenda-

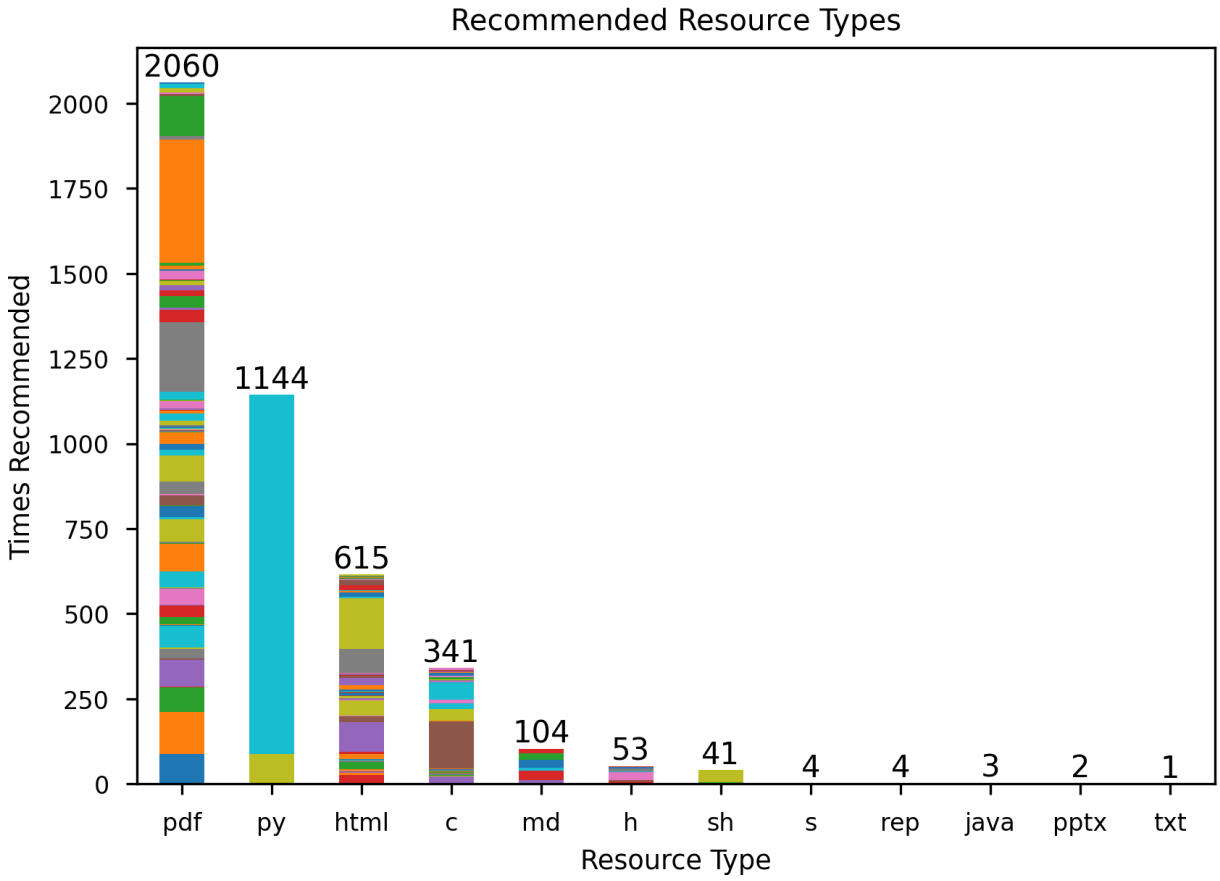
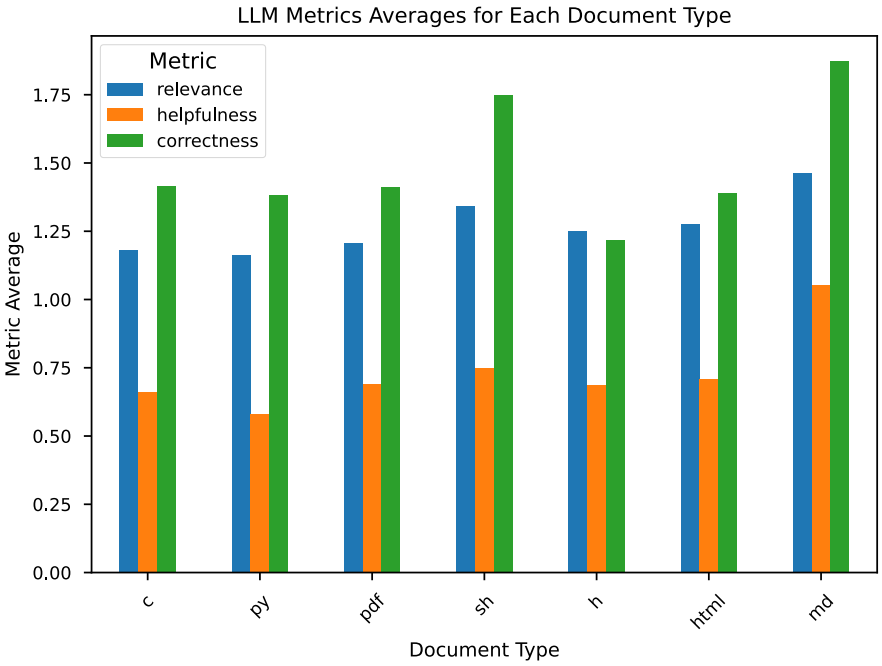


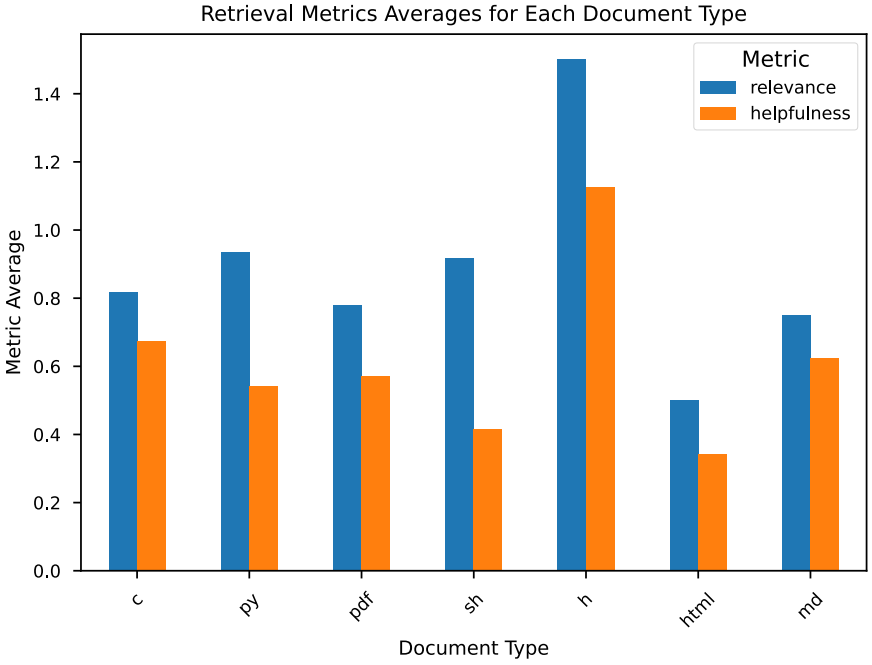
Figure 4.19: The number of times each document type was retrieved, with each contiguous color within a bar representing a document. Document types are shown according to their file extension, where rep was a custom file type for the second CS3214 project.

tions are spread among many documents, but for Python it is dominated by one document, `server_unit_test_pserv.py` (the web server project unit tests), due to its high recommendation count and because there are only two Python files in the course's corpus relevant to student's work during the study. Specifically, the other Python files are unit tests for the course's first and second projects, not the web server project, which is the fourth project.

While different document types were retrieved for different amounts, there are no overarching trends that emerge when inspecting the average LLM and retrieval metrics per document type as shown in Figure 4.20a and Figure 4.20b, respectively. However, we offer theories



(a) LLM reviews



(b) Retrieval reviews

Figure 4.20: Average review metrics per document type.

about several outliers:

- Shell scripts (`.sh`): the only shell script in this course's corpus is `testloginapi.sh`, which demonstrates how students can write `curl` [76] commands to interact with their web server in the web server project. As a result, questions that included this document type primarily asked how to write commands using the `curl` command. The LLM already knows how to write scripts using this tool, likely inflating the correctness metric for this document type.

- Markdown (`.md`): the only markdown documents in this course's corpus discuss fuzzing students' web servers for extra credit. As a result, most questions that include this document type ask about fuzzing. We were unable to find a clear answer for why correctness is higher for this subset of questions given the subset's small size (only 29 questions), but it is possible that the markdown files about fuzzing service the LLM better because they are terse and give clear examples of how to use the provided fuzzing tool or that questions in the subset are more clearly fully correct to students (resulting in students rating correctness at a full 3, rather than 1-2).

- C header files (`.h`): While there are a variety of header files in the course's corpus, they are all from projects' starter code given to students. Importantly, header files contain documentation for functions that students use from the starter code. When a student asks a retrieval-only question about these functions, it is likely that they are seeking this documentation, resulting in much higher average retrieval relevance and helpfulness scores than for other document types.

- HTML (`.html`): as part of the ingestion process, documents of this type were converted to text by extracting the text from the webpage without formatting or layout information. As a result, previews of these documents is an unformatted text dump.

> We believe that this text dump is difficult for humans to read, resulting in low retrieval metrics. On the other hand, we believe the LLM metrics were unaffected by the poor formatting because the LLM can still generate a high-quality response with poor formatting in its prompt.

Lastly, we assign a category to each document based on the document's original web URL. These categories are coarser than the RAG pipeline's categories. Specifically, they are project unit tests, project documentation, project code, lecture material, lecture code examples, and past exams. When looking at the percentage of posts that included one or more document from each category, displayed in Figure 4.21, we see that each category was not retrieved evenly. Project unit tests were predictably the most retrieved category due to the aforementioned high retrieval count of the web server unit tests, followed by lecture material. Surprisingly, lecture material was higher than project documentation, despite the prevalence of questions about projects. This is likely because documents from the lecture material category, due to their inclusion in the `material` or `admin` question categories, are not filtered out for assignment-specific questions. On the lower end, lecture code examples and project code were not retrieved often, suggesting either a trend in the type of question asked or in our retrieval system.

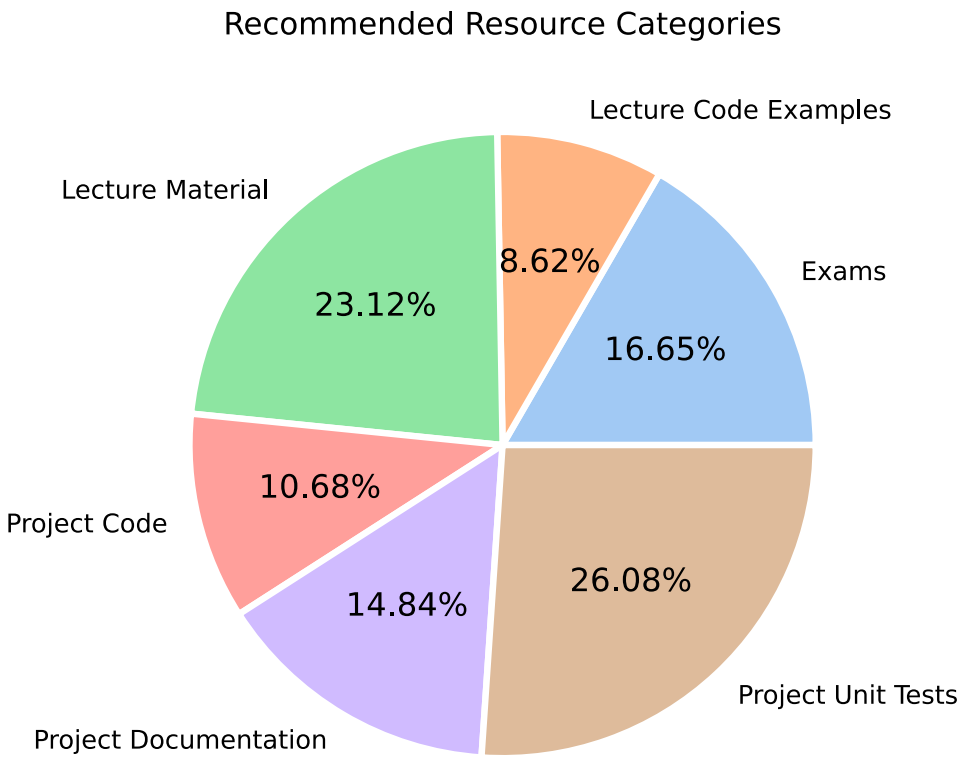


Figure 4.21: Percent of retrieval posts with one or more chunks from each resource category.

4.3 Survey Results

The exit survey was made available to students at the end of the research study. We received 33 responses of which 20 responses were complete. The survey consisted of 21 multiple choice, ranking, and checkbox questions and one optional text field for extra comments. Unlike the research study, no extra credit incentive was given for completing the survey. We break down our analysis into trends from the 21 structured questions and trends from the open response field.

4.3.1 Structured Questions

The survey shows that students used the tool for questions on both course assignments (96.88% of students) and lecture material (56.25% of students) but that they asked primarily about course assignments (81.25% of students). This may be influenced by the study period, in which students had two assignments to work on. Also, only one respondent indicated that they had asked a question in another category, namely an administrative question about the location of the final exam.

Reassuringly, the majority of students found Disdoc helpful, as shown in Figure 4.22. Specifically, 40.62% and 31.25% of students found the tool slightly helpful and helpful, respectfully, and only 3.12% found it slightly unhelpful or unhelpful. Additionally, a majority of students (56.25%) said they would like to use Disdoc more in the future.

When working on course assignments, students felt on average that the previews of relevant chunks embedded in each post were more helpful than the LLM-generated answers or links to the chunks' full documents. It is surprising that LLM-generated answers had the lowest average helpfulness rating, given students' overwhelming preference for seeing an LLM-generated

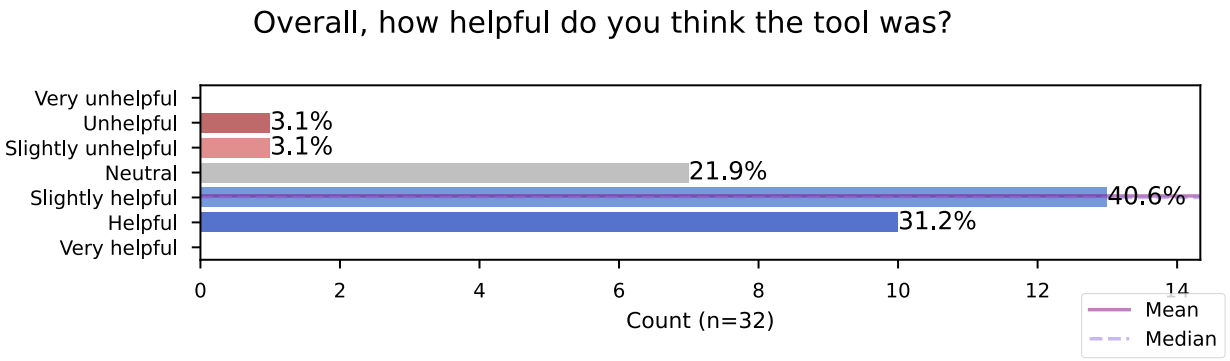


Figure 4.22: Survey responses on Disdoc's helpfulness.

answer to their question. But this appears to be due to a wider spread of answers, leaning more negative than for other features, around the most common response of "Slightly helpful." It is not at all surprising, given the low number of link clicks recorded during the research study, that the largest number of respondents reported they never used related links for questions related to course assignments. For questions about lecture material, students reported similar results.

When looking at student perceptions of relevance, there is no clear order among chunks previews, LLM-generated answers, and links to relevant documents. While students rated the relevance of all features as slightly positive, the relevance ratings are all within the margin of error of each other.

On the other hand, it is clear that students preferred code resources in responses over other types of resources, as shown in Table 4.2. Namely, students rated assignment code and lecture code examples highest with average ratings of 2.11 and 2.78, respectively. Next is assignment instructions, with an average rating of 2.99. It is likely that these resources are highly rated because they provide information that is the most directly relevant to course assignments. The embedded previews of code and PDF resources, being syntax highlighted code snippets and annotated PDFs respectively, are also well-formatted for users. It is surprising that

Document Type	Average Rating
Assignment code	2.11
Lecture code	2.78
Assignment PDFs	2.89
Powerpoints	3.44
Course website pages	4.28
Other	5.44

Table 4.2: Survey respondent's average rating for each of Disdoc's document types.

course web pages were ranked low, below even PowerPoint slides, which were rarely served in practice (most presentations in CS3214 were converted to PDFs before being published). In fact, web pages included many helpful resources, like FAQ pages for projects and the walk-through for the exercise students worked on during the research study. Unfortunately, we did not differentiate between questions about course assignments and lecture material for this ranking question, so it is unclear if these rankings differ for questions about course assignments and lecture material the way helpfulness ratings do.

Interestingly, the survey responses do not fully agree with the collected per-document type metrics from Figures 4.20a and 4.20b on which document types are the best. First, the survey responses indicate a strong preference for assignment code and lecture code examples, but the metrics for C source files (`.c`) are not higher than for other document types. Similarly, survey responses indicate an affinity for assignment instructions, but the metrics for PDFs are not significantly higher than those of other document types.

The majority (63.16%) of students reported that the LLM occasionally generated incorrect answers, shown in Figure 4.23. 31.58% of responses said that it generated incorrect responses rarely or very rarely. It is difficult to say whether these are accurate without comparing to expert ratings. Regardless, it is clear that students must be reminded not to blindly trust the output of LLMs. On the other hand, as shown in Figure 4.24, only 5.26% of students said that

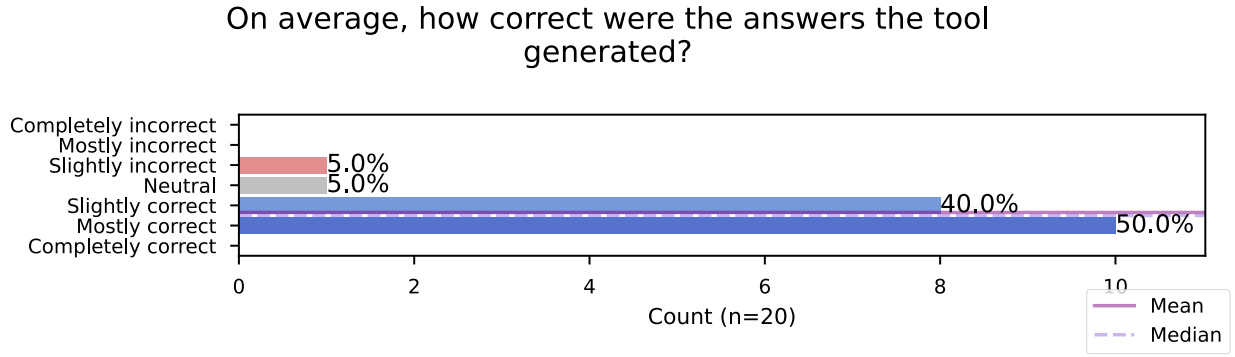


Figure 4.23: Survey responses on Disdoc's correctness.

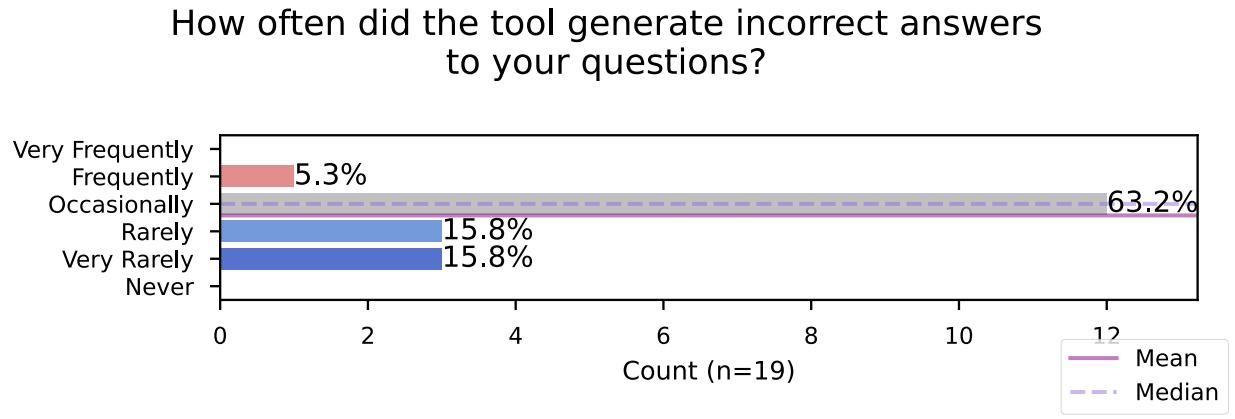


Figure 4.24: Survey responses on the frequency of incorrect answers.

Disdoc "frequently" reported incorrect answers and no students reported that Disdoc "very frequently" generated incorrect answers. Further investigation is required to see if there are trends in the types of questions Disdoc was frequently getting incorrect. Reassuringly, 50% and 40% of students said the LLM generated responses were mostly and slightly correct on average, respectively. So, according to survey respondents, Disdoc's responses are generally correct, but not perfect.

We also asked students about their use of AI tools in other classes, in an effort to identify trends or bias. 50% of respondents reported that they use AI tools several times per week, with the other responses spread evenly among "several times per day," "once a week," "several

times per month," and "never." Interestingly, no students reported that they use AI tools once per day. Thus, it appears that while the majority of students exhibit more reserved usage of AI, a subset of students depend on it heavily.

Students also indicated that their most popular uses for AI are for questions about assignments and lecture material (80% and 70% of students, respectively). Writing code (45%) and writing text (35%) were less popular. Remember that CS3214 students are typically towards the end of their degree program. In CS3214, they write highly technical, low-level code that LLMs may have trouble generating and likely write less prose than students new to Computer Science who often take courses outside of the department.

When asked to check all AI products they are familiar with, ChatGPT was far ahead of others: 95% of students were familiar with it, while Microsoft Copilot was in second with 30% and GitHub Copilot was in third with 20%. While ChatGPT's popularity is not shocking, the magnitude of the gap between ChatGPT and its competitors is. Notably, CS3214 students are encouraged to use Visual Studio Code, which GitHub Copilot integrates with.

Students reported that their experience with these other AI tools was helpful when using our tool, with 55% of respondents reporting that it was "slightly helpful." This is shown graphically in Figure 4.25. Only one student reported a negative sentiment, namely that their experience with other tools was "slightly unhelpful." The reported skill transfer is expected, given the popularity of ChatGPT and the use of GPT-3.5 Turbo for the LLM-generated answers.

Sentiment about AI tools was generally positive, shown in Figure 4.26, with only one respondent indicating a negative opinion with "dislike." In fact, 80% of respondents reported an opinion of "somewhat like" or better. Similarly, 95% of respondents reported that they think AI chatbots should be allowed in classes. Unfortunately, given the small sample size,

we can not discredit sample bias. However, when asked whether code-generating AI tools should be allowed in classes, there was a 70/30 for/against split. So while the vast majority of students like chatbots, some feel their code-generating capabilities should not be allowed.

Unfortunately, students are split between "somewhat worse" (45%) and "about the same" (40%) when asked how our tool compares to other AI tools that they have used in the past, as shown in Figure 4.27. Mistakenly, we did not ask about how our tools compares to others specifically for CS3214-related question and work. As a result, it is not clear whether

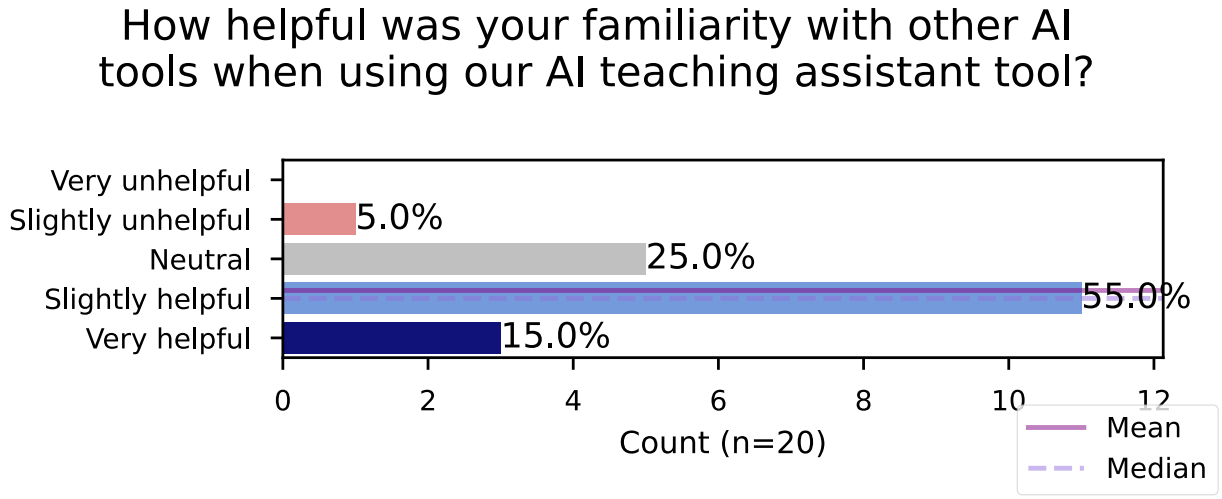


Figure 4.25: Survey responses on how helpful familiarity with other AI tools was when using Disdoc.

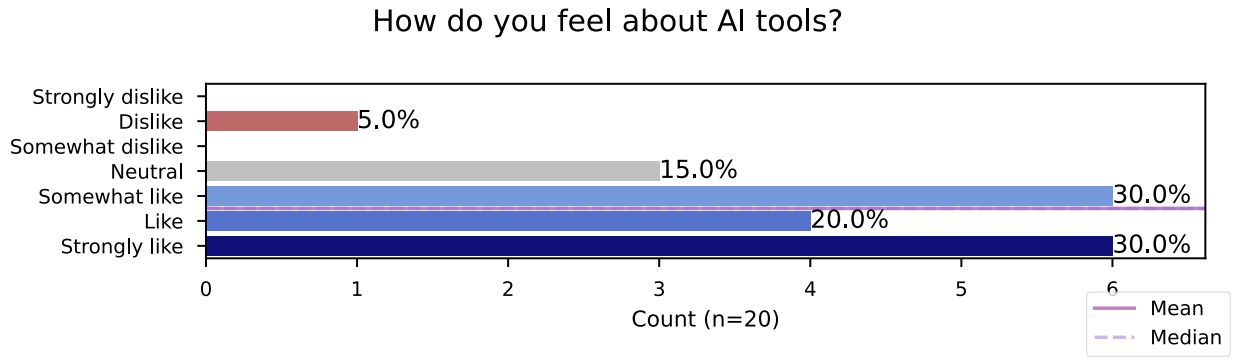


Figure 4.26: Survey responses on opinions about AI tools.

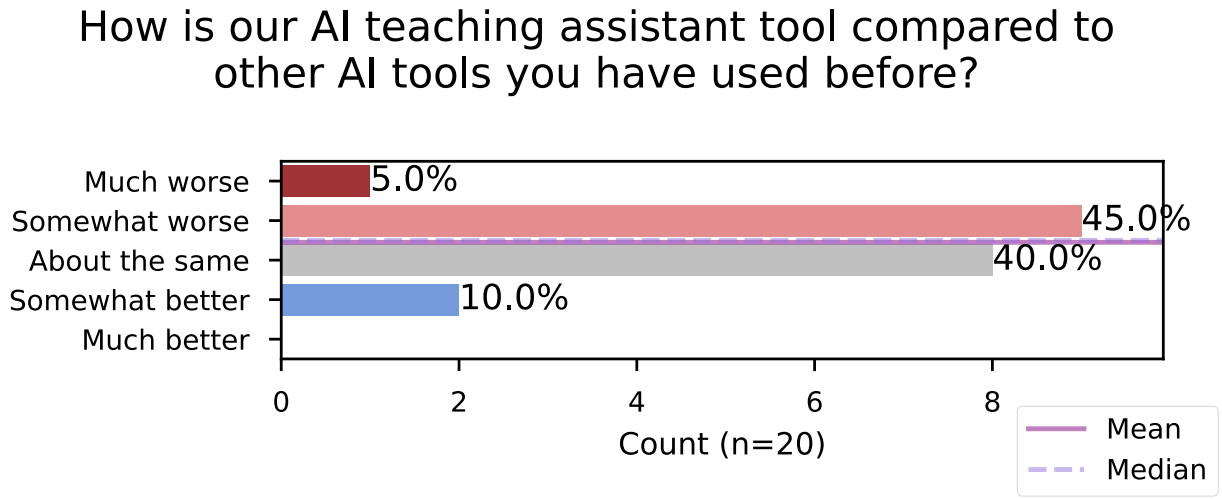


Figure 4.27: Survey responses comparing Disdoc to other AI tools.

students were rating our tool specifically on its target use case or on non-target, general use. Nevertheless, we make several possible explanations as to why it might be worse than the state of the art at the time:

- We did not use a state of the art model for LLM-generated answers for cost reasons. However, it is unknown whether students were measuring our tool against a state of the art model like GPT-4 or against more generally available free models like GPT-3.5, which was the only LLM available to users in ChatGPT's free plan at the time of the survey.

- We did not leverage the full context size of GPT-3.5 Turbo, instead limiting the context size to 4096 tokens for compatibility with Llama 2. Students could make use of the model's full context when using the chat bot directly through ChatGPT, for example including an entire copy of a C header file rather than just relevant chunks.

- We do not provide conversational functionality, since our use-case was a question and answer service. Unfortunately, this means that students can not easily ask follow-up

Category	Count
Want better retrieval results	4
Want project-specific metadata	2
Good at lecture material questions	1
Want more sources	1
Wasn't finding starter code	1

Table 4.3: Comments from the survey's open response section, grouped into categories.

questions or correct mistakes in the original response.

- A well-written prompt is key for quality LLM output, and a handwritten prompt can potentially be better than an automated one from our tool. However, our tool still has the potential to generate better prompts than students who do not know or recall the full breadth of resources available to them and can compile a prompt much quicker than a human. Additionally, suggestions for generating better prompts are given in Future Work.

4.3.2 Open Response

At the end of the survey, we asked students for any additional comments or feedback on the tool. While only 4 students wrote a response, 3 of them were quite detailed. The responses were broken up into separate comments and categorized in Table 4.3.

The top category, improving retrieval results, is one of the problems we identify to reduce the number of students who rated our tool "slightly worse" than other AI tools; we agree that improving the quality of retrieved chunks is a topic for future study. The identified issue of not finding starter code is related, but also suggests that we could improve the embeddings model. Further discussion is included in Future Work.

Project-specific metadata is an idea independently proposed by two students who noted that

they were occasionally getting answers that contained samples from the wrong programming language or missing basic details about a project. This is certainly a problem of having the wrong context. For example, asking how to fix a unit test might flood the context with samples from the Python unit tests, leading the LLM to generate Python code for a C-based project. Including project-specific metadata (at minimum, a summary of the assignment and language(s) used) in every prompt for a given category could lower our reliance on retrieval for this basic information, thereby alleviating such problems.

The student who reported that they want more sources also wrote that Disdoc failed as a replacement for Google when assignments asked students to research a topic to complete the assignment. However, in many cases this is desirable. For example, students were asked to research the binary layout of Linux executable files to write code that parses them. We intentionally do not include documents about this topic, such as the ELF specification, so that students have to research this topic themselves. Otherwise, the assignment would be trivial. However, as this student noted, Disdoc was incapable of answering most questions about such topics as a result, even innocent clarification questions students had after researching a topic. If answering such questions is desired, a better solution could be to include the omitted documents in the corpus, but include instructions in the project metadata to limit what the LLM can disclose from the documents.

Chapter 5

Related Work

5.1 Improving Traditional Help-Seeking Resources

Researchers have developed several electronic queue systems to improve the efficiency of office hours. My Digital Hand is a widely used open-source online tool for queuing and tracking student assistance [73]. Because queuing is managed by the tool, it is highly scalable and intended for use in large introductory computing courses. In using My Digital Hand at three institutions, Smith et al. discovered several trends: the top 5% of students used roughly 50% of total TA time, average wait time grows as deadlines approach, and long interactions contribute to long wait times. Similarly, CS50 Queue is a web- and iPad-based system designed to scale for large introductory courses, but relies on a designated "greeter" (typically a more senior TA) to manually assign students to on-duty TAs [55]. Notably, the greeter can send multiple students with the same question to a single TA using the tool, optimizing office hours. Unfortunately, this grouping functionality was limited in practice by students, who submitted vague or unhelpful questions.

Other research has examined the queuing strategy itself. Hott et al. investigated cosine similarity to group together similar questions in the queue [38]. Using office hours attendance from a past semester, they found that such an approach can reduce median wait times from 7 to 1 minutes on average and 23 to 6 minutes on the busiest days. Importantly, by automating the detection of similar problems, this approach eliminates the need for a greeter in CS50

queue while retaining the grouping functionality. Based on past data and estimations for how long students will wait in a queue before quitting, Gao et al. tested how four queuing strategies (first come first served, new student first, longest wait first, and very long wait first) impact the number of students TAs and instructors service [28]. They found that new student first was the best strategy for servicing the maximum number of students.

Some research focuses on alternative to course-specific, one-on-one office hours. In an effort to minimize the time course-specific TAs are idle, Campbell and Craig investigated drop-in help centers, where TAs are pooled across courses [9]. Specifically, they had one help center for first-year computing courses and another for all other computing classes. Shirazi et al. proposed undergraduate learning assistants (ULAs), undergraduate TAs whose only responsibility is to host drop-in office hours for their set of courses and attend weekly trainings [34]. Surveyed students who used ULAs reported higher course satisfaction than student who did not. Akhmetov et al. outlined a three-tiered TA structure for large courses, where instructors talk to lead TAs, who are each responsible for a team of TAs [2]. They settled on five teams: communication, content, "lost student" support, plagiarism, and scheduling.

We share a common goal with all of these works: improving the efficiency of office hours. However, to address this goal, we introduce an help resource that does not degrade under high use rather than modify office hours procedures. Note that our solution only affects office hours if it can help students well enough to eliminate their need for office hours at least some of the time. High student usage and the positive student perceptions of Disdoc's relevance, helpfulness, and correctness suggest we succeeded, but future work is necessary to fully establish a connection between use of Disdoc and use of office hours. Regardless, the minority of students who overuse office hours, as identified by My Digital Hand, are not a problem for Disdoc. Similarly, students with similar questions, as addressed by CS50 Queue and Hott et al., can ask Disdoc the same question repeatedly without creating inefficiencies.

These students can also search for the similar questions their peers have already asked, which is not possible with office hours.

Lastly, there is research on asynchronous question and answer systems that supplement office hours. Lenfant et al. reported that 81% in their 560 student CS2 course posted at least once to Piazza and over 99% of students viewed Piazza at least once [83]. Additionally, students who asked at least one question on Piazza had better grades in the course. Thinnyun et al. found that Piazza can also increase equity: women on Piazza "post more questions than men, spend more time on the discussion site, and achieve higher reputation scores" [79]. However, Mannekote et al. found that forums for three Computer Science courses contained relatively few conceptual questions and an order of magnitude fewer student responses than TA responses [58].

Discord, the platform students use to interact with Disdoc, is a communication platform that can be used as an asynchronous question and answer system. Bridson et al. dedicated TAs to 24/7 help through Discord, where 90% of students posted at least once, compared to less than half in past semesters with Piazza [5]. Similar to research on Piazza, they saw that usage by underrepresented groups in Computer Science matched the usage by overrepresented groups. While not the focus of their study, Akhmetov et al. reported that students praised the "positive energy, welcoming atmosphere, and learning-focused environment on the [course Discord] server" [2].

Like these works, we focus on an asynchronous question and answer system to supplement office hours. And similar to Lenfant et al. with Piazza and Bridson et al. with Discord, we see high student engagement with Disdoc. However, by leveraging LLMs and RAG, Disdoc does not rely on teaching assistants and instructors at all, unlike Piazza and Discord. As a result, Disdoc is significantly more scalable than Piazza and Discord. It is also possible that, as another online help resource like Piazza and Discord, Disdoc also sees higher engagement

by underrepresented groups than office hours. But, because this was not the focus of our work, we did not collect the data needed to evaluate this connection.

5.2 LLM-based Assistants

CodeHelp is an LLM-based assistant designed to help students in an introductory computer and data science course without generating solution code [50]. To accomplish this, CodeHelp provides students a simple web-based user interface and uses a multi-step prompting pipeline that generates answers to student questions and removes any generated code snippets from the answers. This focus on removing code from LLM-generated answers differs from our research, where we are primarily focused with connecting LLMs to course content and exposing course resources to students. As such, in place of CodeHelp's more robust prompting pipeline, we simply limit Disdoc's answer length such that generating large code solutions is infeasible. Unfortunately, CodeHelp's only connection to the course's content is a list of banned keywords and programming language default that the instructor configures for the course. As a result, CodeHelp suffers from more hallucinations when adapted for computing courses with content not adequately covered by the LLM's training corpus and relies heavily on students to provide any context behind their questions using the interface's optional source code and error message fields.

Sheese et. al analyzed how students use CodeHelp in an introductory computer and data science course [71]. They found that questions about debugging and implementation were far more popular than conceptual questions. Improving LLM-generated answers to such questions is a focus of Disdoc, motivating our connection between course content and the LLM using RAG. Interestingly, the authors found that 14% of all queries were substantially copied from course documents and 31 out of 49 total students submitted at least one query

like this. This process of copying snippets of relevant course documents into a prompt is exactly what RAG automates, meaning that Disdoc directly mirrors what students do for a substantial number of queries. Also mirroring our findings, the authors identified an outlier who submitted drastically more questions than the average student. Further research may give insight into why these students are heavily reliant on AI and what pedagogical impacts this reliance may have.

Also using CodeHelp, Denny et al. investigated what characteristics students in an introductory computing class desired from LLM-powered teaching assistants [17]. Notably, students desired guidance over generated solutions, correctness, 24/7 accessibility, and responses appropriate for the level of students and content of the course. Disdoc has most of these characteristics: large-scale code generation is infeasible due to the character limit, students rated correctness favorably, it is available 24/7, and we specifically use course content to generate level- and content-appropriate answers through RAG.

CodeAid is another LLM-based assistant developed for a course similar to Disdoc's target course, CS3214 [46]. CodeAid provides a rich user interface with five input templates and integrated documentation to guide students through a variety of predetermined issue types. This guidance differs from our research, where Disdoc provides an open-ended question text field. Interestingly, some users preferred the highly structured nature of CodeAid's user interface, while others preferred the more open-ended prompting interface of ChatGPT (which is closer to Disdoc's) when asking questions. But similar to CodeHelp, CodeAid does not explicitly integrate with course material, instead servicing student questions using its parameterized knowledge of C development and information included in its prompts. Again, this means CodeAid performs poorly on questions about content not included in the LLM's training corpus and relies on students to provide the missing context behind their questions.

In 2024, Liu et al. reported on their CS50.ai suite of AI tools in Harvard's introductory

programming course, CS50 [52]. Most relevant from the suite is Duck, a chatbot using a library of prompts and RAG over course lecture transcripts. Duck's RAG component only uses captions from lectures, which is considerably less material than we use with Disdoc. The authors note the lack of additional material contributed to inaccuracies and hallucinations in some queries, particularly for administrative questions. We did not separately analyze Disdoc's performance by question type, but we include administrative materials in Disdoc's RAG system and expect higher performance on administrative questions as a result. Additionally, while the CS50.ai suite was initially developed for CS50, the authors tweaked the suite's prompt library and RAG corpus to successfully deploy the suite to 10 other introductory courses. Because Disdoc shares this RAG-based architecture, this substantiates Disdoc's applicability and flexibility across other computing courses.

Hicke et al. evaluated many approaches to LLM-based question and answer tools using eight semesters of Piazza question and answer data from an introductory programming course [36]. Specifically, they tested combinations of models, fine-tuning on question-answer pairs, direct policy optimization on answers' edit history, and RAG on instructional material. Most notably, their RAG configurations included most of the same resources as Disdoc, the only omissions being administrative documents and past exams. Another commonality: the authors recognize that using solely embedding-based retrievers hinders questions about specific assignments. However, they address this issue using a hybrid embedding- and keyword-based retriever, while we have students manually label their questions with categories that narrow the set of text snippets eligible for our sole embedding-based retriever. Unfortunately, they did not deploy their solutions as we did; they use rubric-based human and GPT-4 evaluation of answers instead. We also expose Disdoc's retrieval results directly to students and evaluate these results, while they do not. Nevertheless, their evaluation shows that RAG was the most significant single modification of those tested for reducing hallucinations and

improving performance, reinforcing the value of Disdoc's RAG system.

In 2023, Kazemitabaar et al. integrated a LLM-based code generator into their self-paced, exercise-oriented, introductory Python platform Coding Steps [45]. They found that students submitted AI code suggestions without modification 49% of the time, causing concerns for over-reliance. Despite this, students did not perform worse than peers without access to the code generator on multiple choice questions or tasks where they had to manually modify code. Also, prompts to the code generator were 41% similar to task descriptions and 32% were exact duplicates, again suggesting significant opportunities for RAG. However, the course styles and student ages (10-17) in their research are quite different from ours. Particularly, students are required to progress through exercises to learn content in Coding Steps, whereas students are free to access any content in CS3214.

While Disdoc attempts to meet students where they are communicating, other research integrates LLMs with development tools students are using. Hou et al. integrated LLMs with a Python IDE for novice programmers, generating Parson's problems with four levels of explanations when students need assistance [39]. Taylor et al. integrate LLMs with the Debugging C Compiler to give students informative, natural language explanations for compile-time and run-time errors [77]. While there are merits to the broader, open-ended question and answer use case that Disdoc and other LLM assistants target, by narrowing their target type or format of student help, Ho et al. and Taylor et al. do not suffer from the missing connection between LLM and course-specific information. However, their prompts are designed for smaller projects, requiring the student's full source code to fit in their prompt to the LLM. Such an approach would not work for a course like CS3214, where projects span many thousands of lines.

Research has also focused on generating code explanations with LLMs. Leinonen et al. compared code explanations generated by GPT-3 to those written by students and found that

students rated the LLM-generated explanations as easier to understand and more accurate than those written by their peers [48]. MacNeil et al. used GPT-3 to generate three levels of code explanations for their course's interactive programming e-book and students reported that the explanations matched the code were useful for learning [54]. Low-rated explanations were typically overly detailed and verbose, mislabeled in level, or contained code despite being instructed not to. Disdoc can generate code explanations when asked, given its open-ended prompting interface. However, we did not evaluate Disdoc's code explanation capabilities specifically and there is no guarantee that Disdoc's retrieval system will pull the code chunks necessary for the student's requested code explanation. As a result, inspiration can be drawn from these studies to improve its code explanations.

Chapter 6

Future Work

As future work, we are focused on improving Disdoc's RAG pipeline, easing adoption of Disdoc by other courses, and evaluating Disdoc in additional areas.

6.1 Improving RAG

Since the start of our research, companies and researchers have published LLMs with better performance and significantly longer context sizes, at the same or lower cost than the GPT-3.5 model we used. For example, compared to GPT 3.5, GPT-4o has a 128 000 token context length (GPT-3.5's context length is 16 385 tokens) and exhibited 14.5% and 27.8% higher performance on the English versions of ARC-Easy and TruthfulQA, respectively [61]. As a result, newer models like GPT-4o could generate higher quality answers and enable longer chunks for retrieval.

Also, while we did not use fine-tuning or few-shot prompting in our RAG pipeline, these techniques could improve our RAG pipeline's retrieved chunks and generated answers. Both fine-tuning and few-shot prompting are possible with the collected student review data and past semesters' forum pages and Discord messages. In particular, fine-tuning the embeddings model could improve the relevance of retrieval results, an area that survey respondents noted for improvement, and better respect student preferences like a preference for code snippets.

Additionally, we can add new resource types to the RAG pipeline to improve results. Most enticing are student's own questions, which would allow students to search for questions similar to their own and instructors to investigate what questions students are asking. And although we intentionally did not integrate with student code due to prompt size limits and related issues, using student code as a source for RAG is also a potential area of research. In particular, having access to the student's code during debugging questions could help the LLM better address students errors. However, such a system would also require steps to prevent misuses like code generation, as in CodeHelp's prompting pipeline [50].

Next, survey respondents noted that the LLM occasionally misunderstood the basics of the assignments they asked about. For example, the LLM would suggest recommendations to improve the student's Python code for a project, when the project was really written in C. This is likely because the basics of the project were not retrieved and the LLM guessed the project's basic information based on what was retrieved instead. Including basic metadata for a category (at minimum, a summary of the assignment and language(s) used) in prompts could ensure sensible answers regardless of retrieval results.

Lastly, the chunks used in RAG can be improved. First, generating highlighted screenshots of relevant chunks from webpages, similar to PDFs, would be a significant improvement over the unstyled text dumps previews that the RAG engine currently produces and the Discord bot displays to students. Second, we could substitute our current character-based splitter for code, Langchain's `RecursiveCharacterTextSplitter` [35] for a custom splitter that leverages a parser like Tree-sitter[1]. This could prevent code chunks from ending arbitrarily, rather than at a semantic boundary (e.g. at the end of an if statement, loop, or method), which is a common cause of low quality chunks.

[1] https://github.com/tree-sitter/tree-sitter

6.2 Easing Adoption by Other Courses

While we deployed Disdoc to Virginia Tech's CS3214: Computer Systems, it is designed to be easily adapted to other courses. To support other courses, only Disdoc's categories and corpus need to be changed. However, this process remains untested, and deploying to other courses could reveal pain points. For instance, while Disdoc's scraper currently supports standalone course websites, integration with learning management systems (likely via the "learning tools integration" standard) would trivialize scraping for courses without a standalone website.

Another step required to use Disdoc with a new course is to assign categories to assignments. Although we assign categories by hand or with heuristics, it might be possible to fully automate the creation process by clustering related documents into communities, similar to [20]. This would eliminate a tedious process and lower the barrier for adoption by instructors uncomfortable with coding heuristics. The effectiveness of this automated system could then be compared to Disdoc's current approach and the hybrid embeddings- and keyword-based RAG pipeline used in [36].

6.3 Further Evaluation

While high student usage and positive student perceptions established Disdoc's value as a help resource, we would like to further investigate Disdoc's effectiveness in two areas. First, because we intend Disdoc to lower the demand on office hours by handling student questions without TA or instructor involvement, we would like to evaluate whether usage of Disdoc affects students' usage of office hours. In particular, if usage of Disdoc lowers office hours usage, then Disdoc was successful in lowering office hours demand. Second, use by

and perceptions from subject matter experts (i.e. instructors and TAs) could show whether student perceptions of Disdoc were accurate and whether Disdoc could be useful to more than students.

Chapter 7

Conclusion

As enrollment in Computer Science grows, traditional help-seeking opportunities for students degrade from a poor TA:student ratio. Current research on adapting LLMs for Computer Science education addresses this issue, but mostly target introductory computing courses and do not connect the LLM to all of a course's resources.

To address these issues, we developed Disdoc, an LLM-based question and answer tool for students in advanced computing courses. In Disdoc, we connected the LLM to all course materials through RAG and used question categories to improve retrieval quality under a potentially large amount of course material. Additionally, to connect students to materials relevant to their question, Disdoc exposes the relevant snippets from course documents retrieved through RAG. Disdoc does all of this without directly integrating with student code, avoiding potential issues and higher costs from large prompt lengths.

We deployed Disdoc in CS3214: Computer Systems, a 340-student advanced course at Virginia Tech. During the study, students could freely use Disdoc for help with projects, assignments, and final exam review. We tracked students reviews on the quality of responses and link click activity. In the end, students asked the tool 1099 questions and gave 1318 responses.

On average, students reviewed Disdoc's helpfulness, relevance, and correctness positively. We discovered that students strongly preferred to see an LLM-generated answer in addition to a

list of relevant resources. Also, very few students clicked links in the list of relevant resources, suggesting they were satisfied with LLM-generated answers and embedded snippet previews. While the number of responses to our survey was low, students indicated that Disdoc was helpful, particularly for questions about course assignments. Survey respondents also found code snippets the most useful type of resource.

In order to facilitate use in other courses, we also open-sourced each component of Disdoc and made recommendations for further improvements.

Bibliography

[1] AI@Meta. Llama 3 model card, 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL%5FCARD.md.

[2] Ildar Akhmetov, Sadaf Ahmed, and Kezziah Ayuno. How we manage an army of teaching assistants: Experience report on scaling a CS1 course. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 32–38, Portland, OR, USA, 2024. Association for Computing Machinery. doi: 10.1145/3626252.3630871. URL https://doi.org/10.1145/3626252.3630871.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In Christian Beecks, Felix Borutta, Peer Kröger, and Thomas Seidl, editors, *Similarity Search and Applications*, pages 34–49, Cham, 2017. Springer International Publishing. ISBN 978-3-319-68474-1.

[4] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. Leveraging community software in CS education to avoid reinventing the wheel. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*, ITiCSE '22, page 580–581, Dublin, Ireland, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3502717.3532169.

[5] Kathryn Bridson, Jeffrey Atkinson, and Scott D. Fleming. Delivering round-the-clock help to software engineering students using Discord: An experience report. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume*

1, SIGCSE 2022, page 759–765, Providence, RI, USA, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3478431.3499385.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Zylon by PrivateGPT. PrivateGPT, May 2023. URL https://github.com/zylon-ai/private-gpt.

[8] Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. Generation CS: the growth of computer science. *ACM Inroads*, 8(2):44–50, 2017.

[9] Jennifer Campbell and Michelle Craig. Drop-in help centres: An alternative to office hours. In *Proceedings of the 23rd Western Canadian Conference on Computing Education*, WCCCE '18, Victoria, BC, Canada, 2018. Association for Computing Machinery. URL https://doi.org/10.1145/3209635.3209642.

[10] Pit Ho Patrio Chiu and Paul Corrigan. A study of graduate teaching assistants' self-efficacy in teaching: Fits and starts in the first triennium of teaching. *Cogent Education*, 6(1):1579964, 2019. URL https://doi.org/10.1080/2331186X.2019.1579964.

[11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(1), January 2023. ISSN 1532-4435.

[12] Chroma. Chroma. https://github.com/chroma-core/chroma, October 2022. Accessed: 2024-04-23.

[13] James Clark. GNU roff. https://www.gnu.org/software/groff/, June 1990. Accessed: 2024-07-13.

[14] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1070. URL https://aclanthology.org/D17-1070/.

[15] Svelte Contributors. Svelte. https://github.com/sveltejs/svelte, November 2016. Accessed: 2024-04-23.

[16] Darla J. Twale David M. Shannon and Mathew S. Moore. TA teaching effectiveness. *The Journal of Higher Education*, 69(4):440–466, 1998. URL https://doi.org/10.1080/00221546.1998.11775144.

[17] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. Desirable characteristics for AI teaching assistants in programming education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 408–414, Milan, Italy, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3649217.3653574.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of*

the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423/.

[19] Walter J. Doherty. The economic value of rapid response time. Technical report, IBM Thomas J. Watson Research Center, 1982.

[20] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization, 2024. URL https://arxiv.org/abs/2404.16130.

[21] Bob Edmison, Stephen H. Edwards, Lujean Babb, Margaret Ellis, Chris Mayfield, Youna Jung, and Marthe Honts. Toward a new state-level framework for sharing computer science content. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 59–65, Toronto ON, Canada, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3545945.3569808.

[22] Stephen H. Edwards, Jason Snyder, Manuel A. Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, page 3–14, Berkeley, CA, USA, 2009. Association for Computing Machinery. URL https://doi.org/10.1145/1584322.1584325.

[23] Katrina Falkner, Rebecca Vivian, and Nickolas J.G. Falkner. Identifying computer science self-regulated learning strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, page 291–296,

Uppsala, Sweden, 2014. Association for Computing Machinery. URL https://doi.org/10.1145/2591708.2591715.

[24] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, February 1994. ISSN 0898-9788.

[25] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997.

[26] Zhikai Gao, Sarah Heckman, and Collin Lynch. Who uses office hours? a comparison of in-person and virtual office hours utilization. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 300–306, Providence, RI, USA, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3478431.3499334.

[27] Zhikai Gao, Collin Lynch, and Sarah Heckman. Too long to wait and not much to do: Modeling student behaviors while waiting for help in online office hours. *Proceedings of the 7th Educational Data Mining in Computer Science Education (CSEDM) Workshop*, 2023. URL https://par.nsf.gov/biblio/10392589.

[28] Zhikai Gao, Gabriel Silva de Oliveira, Damilola Babalola, Collin Lynch, and Sarah Heckman. Who should I help next? Simulation of office hours queue scheduling strategy in a CS2 course. In Benjamin PaaÃŸen and Carrie Demmans Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 484–490, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society. doi: 10.5281/zenodo.12729866.

[29] Zhikai Gao, Adam Gaweda, Collin Lynch, Sarah Heckman, Damilola Babalola, and Gabriel Silva de Oliveira. Using survival analysis to model students' patience in online

office hour queues. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, page 1646–1647, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626253.3635517.

[30] Dan Garcia, Kris Jordan, Colleen M. Lewis, and Ketan Mayer-Patel. Innovative approaches to managing scale. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2022, page 1037–1038, Providence, RI, USA, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3478432.3499220.

[31] Julie V. Gawenda, Adam T. Koehler, and Agne Nakvosaite. Augmenting computer science undergraduate TA training programs to benefit recruitment, refinement, and retention. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, page 1650–1651, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626253.3635625.

[32] Google. Puppeteer. https://github.com/puppeteer/puppeteer, May 2017. Accessed: 2024-07-12.

[33] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3929–3938. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/guu20a.html.

[34] Shirin Haji Amin Shirazi, Mariam Salloum, and Neftali Watkinson. A study of undergraduate learning assistants (ULAs) in computer science. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, page

1664–1665, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626253.3635630.

[35] Chase Harrison. Langchain. https://github.com/langchain-ai/langchain, 2022. Accessed: 2024-04-23.

[36] Yann Hicke, Anmol Agarwal, Qianou Ma, and Paul Denny. AI-TA: Towards an intelligent question-answer teaching assistant using open-source LLMs, 2023. URL https://arxiv.org/abs/2311.02775.

[37] Richard D Hipp. Sqlite. https://www.sqlite.org/index.html, 2000. Accessed: 2024-04-23.

[38] John R. Hott, Mark Floryan, and Nada Basit. Towards more efficient office hours for large courses: Using cosine similarity to efficiently construct student help groups. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, page 1684–1685, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626253.3635544.

[39] Xinying Hou, Barbara J. Ericson, and Xu Wang. Integrating personalized parsons problems with multi-level textual explanations to scaffold code writing. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, page 1686–1687, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626253.3635606.

[40] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, Dallas, Texas, USA, 1998. Association for Computing Machinery. URL https://doi.org/10.1145/276698.276876.

[41] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12), March 2023. URL https://doi.org/10.1145/3571730.

[42] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33 (1):117–128, 2011. doi: 10.1109/TPAMI.2010.57.

[43] Greg Kamradt. Needle in a haystack - pressure testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack, 2023. Accessed: 2024-10-18.

[44] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

[45] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi 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*, CHI '23, Hamburg, Germany, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3544548.3580919.

[46] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 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*, CHI '24, Honolulu, HI, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3613904.3642773.

[47] Shao-Heng Ko and Kristin Stephens-Martinez. What drives students to office hours: Individual differences and similarities. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 959–965, Toronto ON, Canada, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3545945.3569777.

[48] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 124–130, Turku, Finland, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3587102.3588785.

[49] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper%5Ffiles/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[50] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. CodeHelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli Calling '23, Koli, Finland, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3631802.3631830.

[51] Rachel S. Lim, Sophia Krause-Levy, Ismael Villegas Molina, and Leo Porter. Student

expectations of tutors in computing courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 437–443, Toronto ON, Canada, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3545945.3569766.

[52] Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. Teaching CS50 with AI: Leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 750–756, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626252.3630938.

[53] Julie A. Luft, Josepha P. Kurdziel, Gillian H. Roehrig, and Jessica Turner. Growing a garden without water: Graduate teaching assistants in introductory science laboratories at a doctoral/research university. *Journal of Research in Science Teaching*, 41(3):211–233, 2004. URL https://doi.org/10.1002/tea.20004.

[54] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 931–937, Toronto ON, Canada, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3545945.3569785.

[55] Tommy MacWilliam and David J. Malan. Scaling office hours: managing live Q&A in large courses. *Journal of Computing Sciences in Colleges*, 28(3):94–101, January 2013. ISSN 1937-4771.

[56] Varun Magesh, Faiz Surani, Matthew Dahl, Mirac Suzgun, Christopher D. Manning,

and Daniel E. Ho. Hallucination-free? assessing the reliability of leading AI legal research tools, 2024. URL https://arxiv.org/abs/2405.20362.

[57] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, April 2020. URL https://doi.org/10.1109/TPAMI.2018.2889473.

[58] Amogh Mannekote, Mehmet Celepkolu, Aisha Chung Galdo, Kristy Elizabeth Boyer, Maya Israel, Sarah Heckman, and Kristin Stephens-Martinez. Don't just paste your stacktrace: Shaping discussion forums in introductory CS courses. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2022, page 1164, Providence, RI, USA, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3478432.3499110.

[59] Samiha Marwan, Anay Dombe, and Thomas W. Price. Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 54–60, Trondheim, Norway, 2020. Association for Computing Machinery. URL https://doi.org/10.1145/3341525.3387394.

[60] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL https://arxiv.org/abs/1301.3781.

[61] OpenAI. GPT-4o system card, 2024. URL https://openai.com/index/gpt-4o-system-card/.

[62] Open.js Foundation. Node.js. https://github.com/nodejs/node, November 2014. URL https://github.com/nodejs/node. Accessed: 2024-04-23.

[63] Wisam A. Qader, Musa M. Ameen, and Bilal I. Ahmed. An overview of bag of words; importance, implementation, applications, and challenges. In *2019 International Engineering Conference (IEC)*, pages 200–204, 2019. doi: 10.1109/IEC47844.2019.8950616.

[64] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018. URL https://openai.com/index/language-unsupervised/.

[65] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019. URL https://openai.com/index/better-language-models/.

[66] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL https://arxiv.org/abs/1908.10084.

[67] Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for IDF. *Journal of Documentation*, 60:503–520, 10 2004. doi: 10.1108/00220410410560582.

[68] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, et al. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9Vrb9D0WI4.

[69] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the*

54th Annual Meeting of the Association for Computational Linguistics Volume 1, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

[70] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. OpenDSA: beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, page 112–117, Koli, Finland, 2011. Association for Computing Machinery. URL https://doi.org/10.1145/2094131.2094154.

[71] Brad Sheese, Mark Liffiton, Jaromir Savelka, and Paul Denny. Patterns of student help-seeking when using a large language model-powered programming assistant. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, page 49–57, Sydney, NSW, Australia, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3636243.3636249.

[72] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.320. URL https://aclanthology.org/2021.findings-emnlp.320/.

[73] Aaron J. Smith, Kristy Elizabeth Boyer, Jeffrey Forbes, Sarah Heckman, and Ketan 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*, SIGCSE '17, page 549–554,

Seattle, Washington, USA, 2017. Association for Computing Machinery. URL https://doi.org/10.1145/3017680.3017800.

[74] Artifex Software. Pymupdf. https://github.com/pymupdf/PyMuPDF, 2015. Accessed: 2024-07-14.

[75] Harald Steck, Chaitanya Ekanadham, and Nathan Kallus. Is cosine-similarity of embeddings really about similarity? In *Companion Proceedings of the ACM Web Conference 2024*, WWW '24, page 887–890. ACM, May 2024. URL http://doi.org/10.1145/3589335.3651526.

[76] Daniel Stenberg. curl, 1998. URL https://github.com/curl/curl.

[77] Andrew Taylor, Alexandra Vassar, Jake Renzella, and Hammond Pearce. dcc –help: Transforming the role of the compiler by generating context-aware error explanations with large language models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 1314–1320, Portland, OR, USA, 2024. Association for Computing Machinery. URL https://doi.org/10.1145/3626252.3630822.

[78] Unstructured Technologies. Unstructured. https://github.com/Unstructured-IO/unstructured, 2022. Accessed: 2024-07-13.

[79] Adrian Thinnyun, Ryan Lenfant, Raymond Pettit, and John R. Hott. Gender and engagement in CS courses on Piazza. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 438–444, Virtual Event, 2021. Association for Computing Machinery. URL https://doi.org/10.1145/3408877.3432395.

[80] S. M Towhidul Islam Tonmoy, S M Mehedi Zaman, Vinija Jain, Anku Rani, Vipula

Rawte, Aman Chadha, and Amitava Das. A comprehensive survey of hallucination mitigation techniques in large language models, 2024. URL https://arxiv.org/abs/2401.01313.

[81] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[83] Mickey Vellukunnel, Philip Buffum, Kristy Elizabeth Boyer, Jeffrey Forbes, Sarah Heckman, and Ketan Mayer-Patel. Deconstructing the discussion forum: Student questions and computer science learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 603–608, Seattle, Washington, USA, 2017. Association for Computing Machinery. URL https://doi.org/10.1145/3017680.3017745.

[84] Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. Pre-trained language models and their applications. *Engineering*, 25:51–65, 2023. URL https://doi.org/10.1016/j.eng.2022.04.024.

[85] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned language models are zero-shot

learners. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=gEZrGCozdqR.

[86] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL https://openreview.net/forum?id=yzkSU5zdwD.

[87] Dezhi Wu and Starr Roxanne Hiltz. Predicting learning from asynchronous online discussions. *Journal of Asynchronous Learning Networks*, 8(2):139–152, 2004.

[88] Tan Yu, Anbang Xu, and Rama Akkiraju. In defense of RAG in the era of long-context language models, 2024. URL https://arxiv.org/abs/2409.01666.

Appendices

Appendix A

Appendix

A.1 Prompt Template

```
<s>[INST] <<SYS>>
You are a helpful, respectful and honest assistant. Always answer as helpfully as possible, while being safe. Your answers should not include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal content. Please ensure that your responses are socially unbiased and positive in nature.

If a question does not make any sense, or is not factually coherent, explain why instead of answering something not correct. If you don't know the answer to a question, please don't share false information.
<</SYS>>

Use the following pieces of context to answer the question at the end. If you don't know the answer, just say that you don't know, don't try to make up an answer.

{context}
```

```
10
11 Question: {question} [/INST]
```