Using LLM-based Filtering to Develop Reliable Coding Schemes for Rare Debugging Strategies

Aysa Xuemo Fan[1 [0009−0002−6168−0460]], Qianhui Liu[1 [0000−0003−1765−1216]], Luc Paquette[1 [0000−0002−2738−3190]], Juan Pinto[1 [0000−0002−2972−485X]]

University of Illinois Urbana-Champaign, Champaign, IL, USA
{xuemof2, ql29, lpaq, jdpinto2}@illinois.edu

Abstract. Identifying and annotating student use of debugging strategies when solving computer programming problems can be a meaningful tool for studying and better understanding the development of debugging skills, which may lead to the design of effective pedagogical interventions. However, this process can be challenging when dealing with large datasets, especially when the strategies of interest are rare but important. This difficulty lies not only in the scale of the dataset but also in operationalizing these rare phenomena within the data. Operationalization requires annotators to first define how these rare phenomena manifest in the data and then obtain a sufficient number of positive examples to validate that this definition is reliable by accurately measuring Inter-Rater Reliability (IRR). This paper presents a method that leverages Large Language Models (LLMs) to efficiently exclude computer programming episodes that are unlikely to exhibit a specific debugging strategy. By using LLMs to filter out irrelevant programming episodes, this method focuses human annotation efforts on the most pertinent parts of the dataset, enabling experts to operationalize the coding scheme and reach IRR more efficiently.

Keywords: Inter-Rater Reliability · Large Language Models · Programming Education.

1 Introduction

Understanding why students succeed or fail in programming is fundamental to supporting their learning process in computer science education research. Online platforms have enabled the capture and analysis of student behavior and performance data on a large scale, helping to reveal factors that contribute to students' success [17]. Among various programming skills, debugging is a distinct and difficult task for novice programmers. They often do not yet possess a comprehensive knowledge of programming [6] and the strategic skills to control the programming process effectively [30]. Thus, understanding the strategies novices employ and the misconceptions they confront is useful for debugging instruction [28].

The study of debugging with log data often requires human interpretation or annotation of students' behaviors, such as in [14, 26]. However, the process of

defining codebook for phenomena of interest (debugging strategy in this study) can prove challenging in large datasets when the phenomena of interest are rare. An important difficulty lies in the need to be able to identify enough examples of the phenomena for annotators to engage in meaningful discussion about how a phenomenon should be operationalized in the data. Because of the scale of the dataset and the rarity of the phenomena annotators may spend a considerable amount of effort looking at irrelevant examples. Once an operational definition has been reached, the rarity of the phenomena increases the burden for validating the definitions through measures of Inter-Rater Reliability (IRR) as annotators will be required to annotate a large amount of examples to obtain a sufficient number of positive examples.

Recent developments in machine-assisted annotation tools have been leveraged to reduce the burden of human annotation when annotating textual data. Tools like nCoder [8] have been developed to help researchers define and automatically annotate phenomena of interest in textual data by searching for userdefined regular expressions aligning with the annotation's definition. nCoder uses such regular regulation to both inflate the number of positive examples of the phenomena when validating reliability of human annotations and to automatically annotate large scale datasets (once IRR has been achieved). While there has been effort to extend the use of nCoder beyond textual data [24], there is a need for similar approaches to annotating other types of data such as computer programming code, video recording or other mixed media.

The emergence of Large Language Models (LLMs) presents a promising avenue as a tool to assist with the annotation of student debugging strategies when solving computer programming problems [36, 35, 1, 4]. LLMs' ability to process and generate both natural language and computer programming code enables them to discern complex patterns and relationships within student programming problems. Such capabilities can be especially helpful when studying rare debugging strategies where LLMs can be used to process large datasets to identify and focus the attention of human annotators on examples of debugging behaviors that are most relevant to a rare strategy.

In this study, we introduce an methodology that leverages GPT-4 [1], a well known LLM, to assist human annotators in operationalizing definitions of rare debugging strategies, and validating these operationalizations through measurement of IRR. The proposed method leverages GPT-4 ability to process large amount of computer programming data to identify the problem-solving episodes that are most relevant to a given debugging strategy and to focus the attention of human annotators on these relevant examples. We demonstrate the effectiveness of our methodology through empirical evaluations and discuss the implications for using LLMs in supporting more targeted discussion of edge cases and facilitating validation of the annotations through measurement of IRR using test samples in which the frequency of occurrence of rare strategies has been inflated. In particular, we seek to answer the following research questions:

1. How well can LLM help focus the attention of human annotators on relevant examples of rare debugging strategies?

2. How can LLMs be used to iteratively refine the operational definition of rare debugging strategies?

3. How can LLMs assist human annotators in validating the operational definition of rare debugging strategies through measurement of IRR?

2 Related Work

2.1 Debugging Annotations

In computer science education, identification of debugging-related concepts in computer code has been an important step toward the interpretation of students' behaviors and effective intervention. Many researchers focused on the strategies employed in debugging, a systematic search for the source of a bug and its removal [20]. These strategies were mostly elicited based on the labeling of students' discourse, such as their think-aloud transcriptions [22], semi-structured interviews [13], or submissions [28]. Some studies focused more specifically on labeling debugging behaviors as components in a problem-solving process [25, 9, 26, 29, 18]. The commonly used annotation approaches are grounded theory [14] and a priori coding [21, 27].

One of the challenges in debugging annotation is that manual labeling is time-consuming. As such, studies that label data with debugging strategies tend to be on a smaller scale, such as in [26]. With the rapid popularization of online learning platforms for programming, the large amount of submission log data has been a rich resource for understanding students' learning process, and there is a need for automated annotations to be applied to larger-scale datasets. Existing work has investigated the use of logs of students' problem-solving traces to study debugging behaviors. However, these studies tend to apply more data-driven methods and look at specific components of debugging, rather than looking at strategies as a whole. For example, [19] calculated four fine-grained levels of code modifications (no, small, medium, and large change) to describe students' debugging techniques. Although code modification is indeed an important component of debugging, debugging strategies are usually a higher-level holistic behavior composed of many factors.

2.2 Rare Phenomena of Interest

Beyond the difficulty of annotating debugging strategies automatically, another challenge can emerge when phenomena of interest to be annotated that are rare. Once the codebook is created, researchers need to validate the coding schemes and also achieve inter-rater reliability (IRR) with a representative sample (usually random selection) of the whole dataset. However, for rare phenomena, their insufficient representation makes it difficult to refine and validate the definition in the codebook. Automated coding tools have been developed to help reduce the cost of this manual process, such as nCoder (https://app.n-coder.org/) for text data annotation. However, rare phenomena still pose challenges. As analyzed by [8], even when the automated coder achieves acceptable reliability with

the human annotators for rare phenomena, there can be a potentially high rate of false negatives and a low recall, which implies that the automated system fails to identify a substantial number of true occurrences of the phenomena or incorrectly picks out too many negative samples as phenomena, questioning the generalizability of the codebook to the whole dataset.

2.3 Annotations with LLMs

Recently, AI researchers have been exploring the potential of LLMs to emulate human annotators in various tasks. For example, [10] evaluated the performance of GPT-3 to annotate unlabeled data and to generate labeled data. [7] used GPT-3.5-turbo to annotate data for different sentiment analysis tasks and compared the accuracy with lexicon-based algorithms. [3] used GPT-3 to code teachers' textual transcripts in classrooms and found GPT outperformed Multinomial Naive Bayes and k-nearest neighbors. [15] found GPT-3.5-turbo outperformed crowd workers on four annotation tasks: relevance, stance, topics, and frame detection. ChatGPT also exceeded crowd workers and trained annotators on the inter-coder agreement in the same study [15]. These studies showed promising results for using GPT to produce comparable annotation results to humans at a lower cost. Besides accuracy and cost, GPT has also been used to provide explanations for implicit speech detection[16] as a way to enhance human understanding. [38] also found that GPT's explanations of its coding decisions can help improve the consistency and construct validity of human-generated codes.

Although LLMs exhibited strengths in automated annotations for general annotation tasks, such automated procedure often still require a pre-existing well-defined and reliable codebook. This is the case in the contexts such as computer science education where there is no unified codebook for debugging annotations. As such, there is a need to investigate ways in which LLMs can be used as tool to facilitate the creation and validation of reliable codebooks.

3 Methods

We first compiled a list of debugging strategies from prior computer science education research, then annotated episodes of student problem-solving behavior to define a codebook of each strategy in our dataset. The initial round of annotation revealed challenges in achieving IRR for rare debugging strategies. We then employed an LLM-based filtering mechanism to identify submission episodes likely to contain these rare strategies. This pre-selection was informed by developing and refining LLM prompts based on earlier expert-annotated examples. These prompts were then applied at a broader scale to identify examples of student behaviors that were closely aligned with the strategies of interest. The LLM's output was then used to 1) focus the annotators' attention on examples of debugging behaviors that could be discussed to refine the coding schemes, and 2) generate samples used to validate the codebook through measurement of IRR. The LLM used in this study is GPT-4.

3.1 Participants and Data Collection

The data was collected from 745 students enrolled in an undergraduate introductory computer science course (CS1) at a Midwestern public university. This course, primarily focusing on Java programming, drew a diverse cohort of students, predominantly first-year undergraduates, with an age range of 18 to 22 years (31% female). These participants represented a range of academic disciplines, though primarily from computer science and engineering fields, reflecting varied levels of prior programming experience (as measured through a self-reported pre-course survey).

Throughout the Fall 2019 semester, students were asked to solve 69 Java programming homework problems designed to progressively enhance their computer programming knowledge. Homework problems were typically assigned every day of the week, and students were given 24 hours to complete each question and as many submissions as they can. On average, students submitted 7.49 solution attempts per question. Upon submitting their code, students received immediate feedback from the auto-grading system. The system first checked each submission for syntax and code-style errors and reported every identified error. If no such issues were found, the student's code was then evaluated against a series of problem-specific test cases. If there was any test error, information about the first test error encountered was provided to the student. The auto-grading system logged each submission, including timestamps, submitted code, and the list of errors shared with the students. This rich dataset provides information about the students' learning process and about how students attempted to debug their programs throughout their multiple submissions. This data collection approach enables in-depth analysis of how students develop and apply debugging strategies as they progress through the course.

3.2 Annotation Process and Initial Challenges

The annotation team consisted of two graduate students proficient in Java programming and experienced in data annotation. They were also familiar with common student misconceptions and debugging strategies. Before annotating, the team reviewed literature to identify a broad range of debugging strategies used by novice and expert programmers, establishing a consistent understanding of observable behaviors indicative of these strategies within the study data.

To develop a comprehensive codebook for annotating debugging strategies, the team reviewed relevant literature, yielding a diverse set of strategies employed by programmers when debugging their code. Key strategies identified include working around the problem [28], gathering information [37], considering alternatives [28], iterative refinement [32], using intermediate results [32], tracing [28, 27], print statements [2, 12, 31, 27, 23], backtracking [12, 23], tinkering [28, 26], isolating the problem [28, 27], divide-and-conquer [27], fixing the first error and ignoring the rest [5], and starting over [14, 22]. An initial list of 22 strategies was collected and served as the foundation for the codebook. This

Table 1. Table of Debugging Strategies, Kappa Scores, and Occurrence Rates

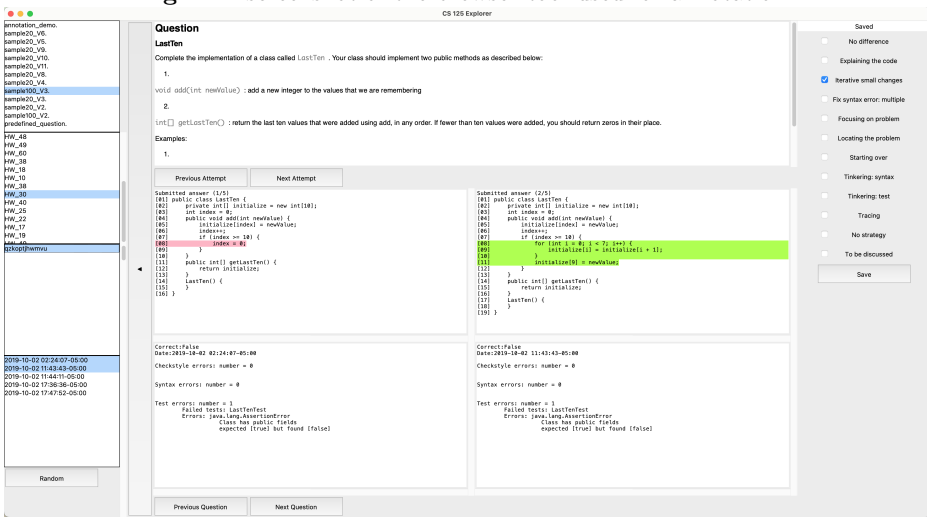
Strategy	Kappa	Occur	Description
Focusing on problem	0.76	52%	Concentrated efforts to rectify a singular syntax/checkstyle error across different code instances.
Iterative changes	0.82	29%	Step-by-step code corrections based on smaller, targeted modifications leveraging existing, functioning code segments.
No difference	0.50	1%	Changes that do not affect the output; include trivial modifications such as variable name changes and function equivalents.
Explaining the code	0.00	0%	Annotations or comments that elucidate code functionality; meaningful renaming of variables and addition of descriptive comments.
Fix multiple syntax errors	0.00	3%	Fixing multiple syntax/checkstyle errors at the same time.
Locating the problem	-0.01	0%	Employing strategies like Divide-and-conquer and Binary search to pinpoint the exact error location such as by commenting out a section of code.
Starting over	0.48	5%	Removing a substantial portion of the code to begin anew, often without a logical, iterative approach.
Syntax error tinkering	0.00	2%	Systematic yet unproductive code modifications aimed at syntax error resolution without a clear direction.
Test error tinkering	0.67	2%	Similar to Tinkering: Syntax, but with a focus on resolving test errors through aimless code alterations.
Tracing	0.39	4%	Utilizing print statements to observe code execution outcomes, with the aim of adjusting code based on the output insights.
No strategy	0.67	19%	(No specific description provided)

list was further refined to combine similar strategies and remove strategies that could not be observed in the study's data.

First, the annotation experts discussed the initial 22 strategies to identify cases where different terms were used for similar strategies, such as *printing and logging* and *print statement*, consolidating related strategies to create a more concise list. Second, they determined the relevance and observability of each strategy within the learning platform, refining the list to focus on those directly observable through students' code submissions and excluding the ones not observable. Overall, 10 strategies were retained. Table 1 presents the final list of debugging strategies and a short description for each one.

We developed a tool to facilitate the annotation of debugging strategies in the students' problem-solving data. The tool displays each student's submissions and errors for every homework question, allowing experts to browse submissions sequentially and compare pairs of submissions to identify debugging

Fig. 1. A screenshot of the browser tool used for annotation.



strategies. Checkboxes are provided for marking strategies, and navigation buttons enable efficient switching between questions and submissions. Randomly selected episodes of student problem-solving, consisting of three to five consecutive submissions, were chosen for annotation. If students required more than five submissions to solve a problem, a subset of five consecutive submissions was randomly selected from all submissions on that problem.

Annotators labeled strategies at the episode level, following the codebook to ensure consistency. They focused on changes between submissions rather than submissions themselves. Strategies like *tracing* and *starting over* were labeled if they occurred at least once in an episode, while others (*focusing on problem*, *iterative changes*, *tinkering: syntax*, *tinkering: test*, and *fix multiple syntax errors*), required at least two occurrences to be labeled. The *no strategy* label was mutually exclusive. The annotators first iteratively coded sets of 20 episodes at a time, comparing annotations, refining the codebook to ensure clear, comprehensive definitions for each strategy. This process continued until they achieved a satisfactory level of agreement, and 220 episodes were coded during this phase.

In the validation phase, the two annotators independently coded 253 episodes. IRR was measured using Cohen's Kappa, which accounts for chance agreement. Kappa values of 0.6-0.80 indicate substantial agreement, while values above 0.8 suggest almost perfect agreement. Only *focusing on problem* (Kappa = 0.76) and *iterative changes* (Kappa = 0.82) showed satisfactory Kappa values and were the most common strategies, occurring in 52% and 29% of episodes, respectively. The remaining strategies occurred less frequently (2-5% in all episodes) and did not achieve the same level of agreement, suggesting further refinement may be needed. Three strategies, *locating the problem*, *no difference*, and *explaining the code*, were almost never observed (occurrence of 1% or less) and were excluded from further analyses.

3.3 LLM-based Filtering Technique

We hypothesized that the low occurrence of relevant problem-solving episodes in the annotated data was a barrier to reaching reliable agreement on rare strategies. Exposing annotators to both positive and negative examples of a strategy during codebook refinement is crucial for discussing the boundaries of what constitutes a positive example. This motivated us to find automated ways to filter a larger amount of debugging episodes to identify those more likely to be relevant to rare strategies. To address this challenge, we employed an LLM-based mechanism to filter submissions, retaining only those likely to contain rare strategies. The goal was not to replace human annotation but to enable annotators to focus on episodes more likely to be relevant to the strategies of interest. This approach involves two steps:

1. Prompt Engineering: We develop prompts that guide the LLM to identify submissions more likely to contain rare debugging strategies, aligning with the characteristics and patterns defined by the annotators in their initial annotation.

2. Inflated sampling: The LLM filter processes the dataset and identifies episodes matching the criteria specified in the prompts, increasing the prevalence of rare strategies within the datasets for annotation.

3.4 Prompt Refinement

We first designed an initial LLM prompt using simple text-based descriptions of the strategy definitions used by human annotators, without explicit structure to guide the LLM's decision-making process. This setup yielded unsatisfactory results, with the LLM struggling to distinguish between rare and common debugging strategies, often overlooking episodes containing rare strategies and leading to misclassifications. For example, the LLM confused *fix multiple syntax errors* with *focusing on problem* and *iterative changes* with *starting over*.

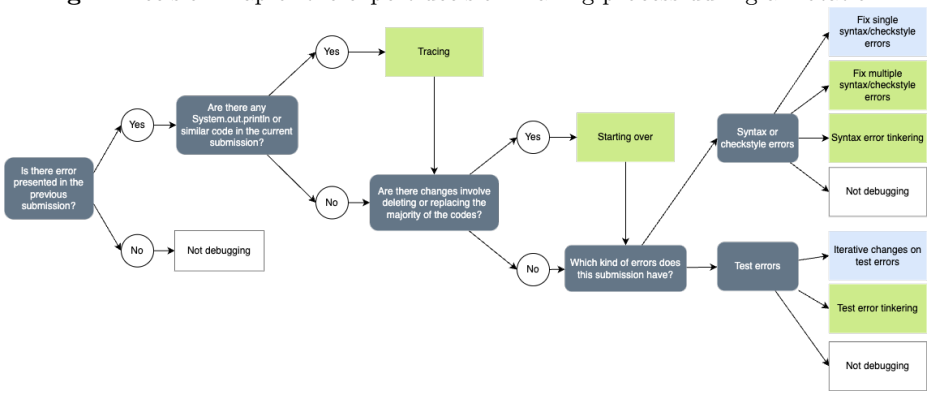
To evaluate the LLM's performance, we compared its annotations to a ground truth dataset of 253 expert-agreed annotations. Kappa scores between the LLM's labels and the ground truth labels revealed low agreement even for common strategies (Table 2). Minor adjustments to the wording and addition of detailed guidelines yielded minimal improvements, aligning with recent NLP research suggesting that prompt variations provide insignificant performance gains [33].

To improve the prompt, we sought to elicit the experts' domain knowledge. Discussions revealed that experts employ a hierarchical structure in their annotation process, ruling out certain strategies before examining others. We developed a decision map (Figure 2) capturing this hierarchical nature and restructured the prompts to emphasize the sequential filtering approach used by experts.

Additionally, we obfuscated strategy names to mitigate LLM bias based on the names, replacing them with generic labels (Strategy A to G, and N for *no strategy*; Table 2). This aimed to prevent the LLM from relying on semantic connotations, forcing it to focus on the underlying patterns and characteristics described in the prompt. The full prompts are available on GitHub[1].

[1] https://github.com/heds-lab/heds-lab-LLM-Filtering-Debugging-Strategies

Fig. 2. Decision map of the expert decision-making process during annotation.



3.5 Further Rounds of Annotations

The refined LLM filtering prompts (P2) were applied to a new set of 500 randomly selected problem-solving episodes to identify examples for refining the coding schemes of rare debugging strategies and assessing IRR. The filter identified 51 episodes for *tracing* and 70 for *starting over*. To inflate the frequency of these strategies, we randomly selected half of the identified episodes for each strategy (26 for *tracing* and 35 for *starting over*), mixed them with an equal number of randomly selected episodes from the remaining 388 episodes not flagged as relevant, and removed duplicates, resulting in 112 episodes for annotation.

Annotators worked through these episodes in batches of 30-40, focusing on the rare strategies and refining their codebook to achieve the desired IRR. This process allowed annotators to reach an agreement and acceptable IRR for *tracing*, but not for *starting over*.

Two additional rounds of annotation were conducted to refine the codebook for *starting over*. In the second round, the prompt was applied to 500 new episodes, identifying 58 relevant episodes. We selected 116 episodes, including the 58 relevant episodes and 58 randomly selected from the remaining 442 episodes. In the third round, the prompt was applied to 300 new episodes, identifying 41 relevant episodes. We selected 82 episodes, including the 41 relevant episodes and 41 randomly sampled from the remaining 259 non-relevant episodes. Acceptable IRR was reached for *starting over* in this third round.

4 Results

4.1 RQ1: Focusing Attention on Relevant Examples

The LLM filtering method, using a hierarchical prompt structure aligned with expert decision-making processes, improved the identification of episodes containing rare strategies compared to the initial simpler prompt.

Table 2. Debugging Strategies with Labels and Kappa Scores with Original Prompts and New Prompts.

Original Strategy	Label	Kappa_original	Kappa_new
Tracing	Strategy A*	0.38	0.71
Starting over	Strategy B*	0.14	0.15
Focusing on problem	Strategy C	0.33	0.44
Fix multiple syntax errors	Strategy D*	0.09	0.00
Syntax error tinkering	Strategy E*	0.43	-0.01
Iterative changes	Strategy F	0.32	0.71
Test error tinkering	Strategy G*	-0.02	0.14
No strategy	Strategy N	0.13	0.40

For the *tracing* strategy, when applied to 453 human-annotated episodes, the LLM filter correctly recalled 16/19 (84%) positive examples and excluded 420/434 (97%) irrelevant episodes. It selected 14/434 (3%) episodes as relevant when human annotation indicated otherwise and excluded 3/19 (16%) human-annotated *tracing* episodes.

For the *starting over* strategy, the LLM filter correctly recalled 20/21 (95%) positive examples and excluded 309/432 (72%) irrelevant episodes. In addition, it selected 123/432 (28%) episodes as relevant when human annotation indicated otherwise and excluded 1/21 (5%) human-annotated *starting over* episodes.

When applied to a dataset of 500 newly selected episodes, the refined P2 prompts identified 51 episodes as relevant to the *tracing* strategy and 70 episodes as relevant to the *starting over* strategy. This included 10 episodes identified as relevant for both strategies. From these, we randomly selected half of the episodes for each strategy (26 for *tracing* and 35 for *starting over*), which is to ensure a balanced examination of episodes that are relevant to the rare strategies. To maintain objectivity and reduce bias during annotation, an equal number of episodes (61) randomly selected from the 388 not relevant episodes were added to the set of episodes to be annotated by experts, and then eliminated the duplicates from the overlaps. This resulted in 52 episodes for *tracing*, 70 for *starting over*, and 10 overlapping episodes, totaling 112 episodes for annotation.

Human annotators identified 20 positive examples of *tracing* out of 52 episodes, and 19 of *starting over* out of 70 episodes, observed in 38% and 27% of episodes, respectively. This encounter rate was considerably higher than in randomly selected episodes without LLM (4% for *tracing* and 5% for *starting over*). This increased encounter rate of rare strategies when using an LLM filter allowed experts to focus on relevant examples manifesting the strategies, enabling them to better understand students' debugging behaviors. This focused analysis facilitated the refinement of the coding scheme for these rare strategies.

4.2 RQ2: Refinement of Coding Schemes

The use of an LLM filtering approach allowed us to create samples in which the occurrence of rare strategies was inflated. This increased occurrence rate allowed for more in-depth discussions and analysis of the ways in which the strategies

manifested in the data. For example, experts discovered that the *starting over* strategy could manifest in various ways, such as changing the code structure or changing the code style. These insights led to the refinement of the coding schemes to better capture the nuances of the strategy.

Moreover, the inflated sampling also identified relevant negative examples – episodes that were similar to those containing the target strategy but did not meet the criteria. Discussing these borderline cases helped experts clarify the boundaries of the strategy and refine the guidelines to minimize ambiguity. For instance, experts encountered episodes where students changed a big chunk of code by simplifying the code with the same logic, which prompted them to add specific criteria to the guidelines to differentiate between *starting over* and *not starting over*.

By focusing on both positive and relevant negative examples of rare strategies, the inflated sampling facilitated a more comprehensive understanding of the strategies and their variations. This understanding, in turn, enabled experts to refine the guidelines and make them more precise, comprehensive, and reliable. The refined guidelines not only improved the consistency of annotations but also enhanced the overall validity of the codebook.

4.3 RQ3: Validation of Coding Schemes

Despite the belief of the human annotators that the *tracing* strategy should have been relatively easy to identify in the data, they were not initially able to achieve a satisfactory IRR. This may have been because, out of 453 annotated episodes, only 19 contained positive examples of the *tracing* strategy and 21 contained positive examples of the *starting over* strategy. This limited their ability to define and validate their coding scheme. Validating coding schemes using IRR measures such as Cohen's Kappa requires annotators to code a sufficient number of samples. In particular, such a sample should include enough positive examples of the coded concepts. For this reason, validating annotations for rare debugging strategies, such as *tracing* and *starting over* can require an important amount of work. Using LLMs to filter and inflate the occurrence of rare strategies can have an important impact on the validation process.

This can be illustrated through the use of Shaffer's Rho [34, 11], a measure that can be used to estimate the number of episodes that should be annotated to achieve a desired IRR. Taking the *tracing* strategy as an example, this strategy had an occurrence rate of 4% in the sample initially coded by human annotators. According to Shaffer's Rho, achieving a desired Cohen's Kappa of 0.8 at a significance level of 0.05 would require a validation set of at least 280 episodes per validation attempt. In contrast, using the proposed LLM filtering approach allowed us to inflate the occurrence rate for the *tracing* strategy to 46% (annotators agreed on 21 tracing episodes out of a filtered set of 52). With this new occurrence rate, Shaffer's Rho indicated that 70 episodes would be sufficient to achieve a similar value of Cohen's Kappa, indicating an important reduction in required annotation efforts. A similar reduction can be observed for the *starting over* strategy where the use of LLM filtering increased the occurrence rate of

this strategy from 5% to 27% (annotators agreed on 19 out of a filtered set of 70) and reduced the number of required episodes (according to Shaffer's Rho) from 240 to 100.

Using the LLM filtering approach, human annotators were able to achieve a Cohen's Kappa of 0.88 for the *tracing* strategy and of 0.58 for the *starting over* strategy after labeling one additional set of 112 episodes with inflated sampling for both of these rare strategies. Having achieved acceptable IRR for the *tracing* strategy, further rounds of validation focused only on *starting over*. The next validation included 116 episodes, for which human annotators achieved a Cohen's Kappa of 0.73. Finally, a third round of validation was conducted using 82 episodes, for which a Kappa of 0.82 was achieved.

5 Discussions & Conclusions

The results of this study illustrate the potential of LLMs in assisting human annotators when defining and validating coding schemes for rare phenomena, such as debugging strategies that are rarely used by students when solving problems in an introductory computer science course. We proposed an LLM-based filtering approach that was used to focus the attention of human annotators on examples relevant to two rare strategies *tracing* and *starting over*, by inflating their occurrence rate in sets of randomly sampled episodes. This provided the annotators with more examples that could be used to refine their coding schemes, and validate these schemes more efficiently.

These findings align with recent literature that highlights the capabilities of LLMs in assisting with data annotation tasks. As observed by [38], LLMs' explanations of their coding decisions can help researchers validate and refine human-generated codes by identifying inconsistencies and prompting researchers to re-examine their rationales. Similarly, in this study, the LLM's ability to identify relevant examples facilitated more targeted discussions among experts, enabling them to refine their codebook and improve the overall reliability of the coding scheme.

Additionally, the LLM-based approach to inflate the occurrence rate of rare strategies reduced the annotation effort required to validate the codebook. This is consistent with the findings of [8], who reported a 50% to 63% reduction in the size of annotation sets when using an LSTM neural network to assist with the development of qualitative codes for text data using regular expressions. By increasing the prevalence of rare strategies in the annotation set, the proposed LLM filtering approach allowed annotators to achieve a reliable IRR with a smaller sample size, saving time and effort in the manual annotation process.

However, this study also revealed limitations of the LLM approach. In particular, it relies on being able to write prompts describing the annotation process in a way that the LLM can correctly interpret and that will filter out a large number of irrelevant examples while retaining both relevant positive and negative examples of the desired phenomena. For instance, the LLM struggled to understand the idea of "difference between submissions" in the codebook, which

is crucial for identifying debugging strategies that manifest across multiple submissions. Additionally, the LLM had difficulty distinguishing between printing statements used for tracing and those required by the homework questions. These limitations underscore the need for further refinement of the LLM prompts and codebook to improve the accuracy of identifying and annotating these specific debugging strategies.

Despite the promising results for the proposed approach, the designed LLM prompt performed poorly on three rare debugging strategies (*fix multiple syntax errors*, *syntax error tinkering* and *test error tinkering*) because the annotators have not even encountered enough samples to develop the filtering prompts. As a result, human annotators were not able to improve the coding schemes for these strategies. Further refinement of the LLM prompts would be required before our proposed method could be applied to these strategies. Future research should explore ways to enhance the LLM's understanding of complex coding patterns and context-specific nuances to better capture these strategies.

Acknowledgements. *This study is funded by National Science Foundation Award #1942962.*

Disclosure of Interests. *The authors have no competing interests to declare that are relevant to the content of this article.*

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Ahmadzadeh, M., Elliman, D., Higgins, C.: An analysis of patterns of debugging among novice computer science students. In: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education. pp. 84–88 (2005)
3. Amarasinghe, I., Marques, F., Ortiz-Beltrán, A., Hernández-Leo, D.: Generative pre-trained transformers for coding text data? an analysis with classroom orchestration data. In: European Conference on Technology Enhanced Learning. pp. 32–43. Springer (2023)
4. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
5. Becker, B.A., Murray, C., Tao, T., Song, C., McCartney, R., Sanders, K.: Fix the first, ignore the rest: Dealing with multiple compiler error messages. In: Proceedings of the 49th ACM technical symposium on computer science education. pp. 634–639 (2018)
6. Begum, M., Nørbjerg, J., Clemmensen, T.: Strategies of novice programmers (2018)
7. Belal, M., She, J., Wong, S.: Leveraging chatgpt as text annotation tool for sentiment analysis. arXiv preprint arXiv:2306.17177 (2023)
8. Cai, Z., Siebert-Evenstone, A., Eagan, B., Shaffer, D.W., Hu, X., Graesser, A.C.: ncoder+: a semantic tool for improving recall of ncoder coding. In: International Conference on Quantitative Ethnography. pp. 41–54. Springer (2019)

9. Chao, P.Y.: Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. Computers & Education **95**, 202–215 (2016)
10. Ding, B., Qin, C., Liu, L., Chia, Y.K., Joty, S., Li, B., Bing, L.: Is gpt-3 a good data annotator? arXiv preprint arXiv:2212.10450 (2022)
11. Eagan, B.R., Rogers, B., Serlin, R., Ruis, A.R., Arastoopour Irgens, G., Shaffer, D.W.: Can we rely on irr? testing the assumptions of inter-rater reliability. In: International Conference on Computer Supported Collaborative Learning (2017)
12. Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., Zander, C.: Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. Computer Science Education **18**(2), 93–116 (2008)
13. Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., Zander, C.: Debugging from the student perspective. IEEE Transactions on Education **53**(3), 390–396 (2009)
14. Fitzgerald, S., Simon, B., Thomas, L.: Strategies that students use to trace code: an analysis based in grounded theory. In: Proceedings of the first international workshop on Computing education research. pp. 69–80 (2005)
15. Gilardi, F., Alizadeh, M., Kubli, M.: Chatgpt outperforms crowd workers for text-annotation tasks. Proceedings of the National Academy of Sciences **120**(30), e2305016120 (2023)
16. Huang, F., Kwak, H., An, J.: Is chatgpt better than human annotators? potential and limitations of chatgpt in explaining implicit hate speech. In: Companion proceedings of the ACM web conference 2023. pp. 294–297 (2023)
17. Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S.H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K., et al.: Educational data mining and learning analytics in programming: Literature review and case studies. Proceedings of the 2015 ITiCSE on working group reports pp. 41–63 (2015)
18. Jayathirtha, G., Fields, D., Kafai, Y.: Pair debugging of electronic textiles projects: Analyzing think-aloud protocols for high school students' strategies and practices while problem solving (2020)
19. Jemmali, C., Kleinman, E., Bunian, S., Almeda, M.V., Rowe, E., Seif El-Nasr, M.: Maads: Mixed-methods approach for the analysis of debugging sequences of beginner programmers. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. pp. 86–92 (2020)
20. Katz, I.R., Anderson, J.R.: Debugging: An analysis of bug-location strategies. Human-Computer Interaction **3**(4), 351–399 (1987)
21. Ko, A.J., LaToza, T.D., Hull, S., Ko, E.A., Kwok, W., Quichocho, J., Akkaraju, H., Pandit, R.: Teaching explicit programming strategies to adolescents. In: Proceedings of the 50th ACM technical symposium on computer science education. pp. 469–475 (2019)
22. Lee, M.J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M., et al.: Principles of a debugging-first puzzle game for computing education. In: 2014 IEEE symposium on visual languages and human-centric computing (VL/HCC). pp. 57–64. IEEE (2014)
23. Li, C., Chan, E., Denny, P., Luxton-Reilly, A., Tempero, E.: Towards a framework for teaching debugging. In: Proceedings of the Twenty-First Australasian Computing Education Conference. pp. 79–86 (2019)
24. Li, Z., Paquette, L.: Automating the engineering of expert-coded features from student interaction log data. In: Conference Proceedings Supplement to the First International Conference on Quantitative Ethnography. pp. 28–29 (2019)

25. Liu, C.C., Cheng, Y.B., Huang, C.W.: The effect of simulation games on the learning of computational problem solving. Computers & Education **57**(3), 1907–1918 (2011)
26. Liu, Z., Zhi, R., Hicks, A., Barnes, T.: Understanding problem solving behavior of 6–8 graders in a debugging game. Computer Science Education **27**(1), 1–29 (2017)
27. Miljanovic, M.A., Bradbury, J.S.: Robobug: a serious game for learning debugging techniques. In: Proceedings of the 2017 acm conference on international computing education research. pp. 93–100 (2017)
28. Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., Zander, C.: Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. ACM SIGCSE Bulletin **40**(1), 163–167 (2008)
29. Pellas, N., Vosinakis, S.: The effect of simulation games on learning computer programming: A comparative study on high school students' learning performance by assessing computational problem-solving strategies. Education and Information Technologies **23**(6), 2423–2452 (2018)
30. Perkins, D.N., Martin, F.: Fragile knowledge and neglected strategies in novice programmers. In: Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers. pp. 213–229 (1986)
31. Perscheid, M., Siegmund, B., Taeumel, M., Hirschfeld, R.: Studying the advancement in debugging practice of professional software developers. Software Quality Journal **25**, 83–110 (2017)
32. Rich, K.M., Strickland, C., Binkowski, T.A., Franklin, D.: A k-8 debugging learning trajectory derived from research literature. In: Proceedings of the 50th ACM technical symposium on computer science education. pp. 745–751 (2019)
33. Salinas, A., Morstatter, F.: The butterfly effect of altering prompts: How small changes and jailbreaks affect large language model performance. arXiv preprint arXiv:2401.03729 (2024)
34. Shaffer, D.W., Ruis, A.R.: How we code. In: Advances in Quantitative Ethnography: Second International Conference, ICQE 2020, Malibu, CA, USA, February 1-3, 2021, Proceedings 2. pp. 62–77. Springer (2021)
35. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.H.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint (2023)
36. Wang, Y., Wang, W., Joty, S., Hoi, S.C.H.H.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: EMNLP (2021)
37. Whalley, J., Settle, A., Luxton-Reilly, A.: Novice reflections on debugging. In: Proceedings of the 52nd ACM technical symposium on computer science education. pp. 73–79 (2021)
38. Zambrano, A.F., Liu, X., Barany, A., Baker, R.S., Kim, J., Nasiar, N.: From ncoder to chatgpt: From automated coding to refining human coding.(2023). Publisher Full Text (2023)