FaithfulPersona: Balancing Faithfulness and Personalization in Code Explanations through Self-Critique

Anonymous NAACL submission

Abstract

Code explanations are crucial in real-world life, from educating students to aligning technical projects with business goals. However, existing approaches face challenges balancing faithfulness to the original code and personalization for diverse user needs. This paper addresses these challenges by introducing a novel benchmark and method for generating faithful personalized code explanations. Our benchmark, `FaithfulPersonaCodeX`, incorporates code samples and user profiles, employing various evaluation metrics to evaluate both faithfulness and personalization. We propose `DISCO`, a new method that uses a self-critique mechanism and two-stage optimization to balance faithfulness and personalization in code explanations, addressing the limitations of current large language model approaches. Our proposed model, `DISCO`, achieves a notable 3.7% improvement in Pass@5 compared to the strong baseline method, Self-Consistency, while maintaining high personalization with a 61.08% win rate in the LLM-as-a-Judge evaluation, effectively balancing faithfulness and user-specific needs in code explanations.

1 Introduction

Code explanations are crucial in the digital landscape, serving as essential learning tools for tech professionals and aligning technical projects with business goals for stakeholders (Feng et al., 2020; Husain et al., 2019; Guo et al., 2022). Beyond industry professionals, code explanations are vital in educating students, helping them gain knowledge and improve understanding of complex systems. These explanations also enhance the transparency of software systems, making them more accessible and comprehensible to a broader audience.

Two primary objectives emerge in code explanation: *faithfulness* and *personalization*. *Faithfulness* ensures that the explanation accurately represents the code's functionality and behavior (Schwettmann et al., 2024; Li et al., 2023b), while *personalization* tailors the explanation to the user's background, expertise, and specific needs (Chen et al., 2023a). However, two significant challenges impede the improvement of faithful and personalized code explanations: the absence of suitable benchmarks to evaluate the quality of different code explanation approaches and the poor performance of existing code explanation methods in handling both faithfulness and personalization.

Regarding the first challenge, most code explanation benchmarks (Yan et al., 2023; Bhattacharya et al., 2023; Tasnim Preoty, 2024) treat code comments or summaries as ground truth. However, this approach is inadequate for comprehensive code explanation (Jiang et al., 2024). Code comments or summaries often focus on specific implementation details or function purposes, failing to capture the broader context, algorithmic choices, or potential alternative approaches. Moreover, these benchmarks typically neglect personalized requests for code explanations, which are crucial for addressing diverse user needs. Some benchmarks (Sarsa et al., 2022; Oli et al., 2023) highly rely on human annotations through questionnaires, which, although they provide ground evaluation, is missing the extensibility to testing new code explanation methods.

The second challenge arises from the limitations of existing code explanation methods, including those based on recent advancements in large language models (LLMs) (Brown, 2020; Lewkowycz et al., 2022). While LLMs have shown promise in various language tasks, they still fail to generate faithful and personalized code explanations (Li et al., 2023b; Jiang et al., 2024). This dual objective presents a significant challenge: Increasing personalization often risks deviating from the code's functionality. At the same time, strict adherence to the code may result in too technical explanations for some users. Current LLM-based methods (Li et al., 2023b), which often rely on single-turn approaches,

are insufficient for addressing this complex balance, especially for intricate programming problems that require iterative analysis and refinement.

To address these challenges, we propose a comprehensive solution that encompasses both a novel benchmark and an innovative method for faithful and personalized code explanation. Firstly, we introduce a new faithful and personalized code explanation benchmark named `FaithfulPersonaCodeX`. That includes code samples, problem descriptions, test cases, and user profiles. This benchmark is designed to evaluate both the faithfulness and personalization aspects of code explanations. We employ a multifaceted evaluation approach, developing two new metrics: This reference-free approach offers several advantages over traditional reference-based metrics: it eliminates the need for costly and time-consuming human-written ground truth explanations, provides a more objective and scalable evaluation method, and allows for dynamic evaluation of explanations based on varying user profiles and preferences.

Building upon this benchmark, we propose **DISCO**, Dual-objective Iterative Self-Critiquing Optimization, a novel personalized code explanation method. Our approach leverages and extends recent advancements in iterative self-critiquing techniques (Gou et al., 2023; Shinn et al., 2024; Welleck et al., 2022). **DISCO** incorporates two key innovations: A customized self-critique mechanism that generates feedback through code regeneration, ensuring the faithfulness of explanations to the original implementation. A two-stage optimization process that balances the competing requirements of personalization and faithfulness. Experimental results demonstrate **DISCO**'s effectiveness over baselines. In summary, our work has the following contributions:

- Introduced `FaithfulPersonaCodeX`, a novel benchmark for evaluating faithful and personalized code explanations.
- Proposed `DISCO` (Dual-objective Iterative Self-Critiquing Optimization), an innovative method for generating faithful and personalized code explanations.
- `DISCO` significantly and consistently outperforms many LLM-based code explanation methods and commercial code explanation tools on `FaithfulPersonaCodeX`.

2 Background and Related Work

2.1 Code Explanation Generation

LLMs like GPT-4 (Achiam et al., 2023) and Llama-3 (Dubey et al., 2024) excel in language generation (Yang et al., 2024) and reasoning (Zhang et al., 2024), making them ideal for the code explanation task crucial in software engineering (Rai et al., 2022) and education (Sarsa et al., 2022). Recent studies (Sarsa et al., 2022; Oli et al., 2023; Bhattacharya et al., 2023; MacNeil et al., 2023; Li et al., 2023b; Nam et al., 2024; Yan et al., 2024b,a; Tasnim Preoty, 2024; Richards and Wessel, 2024; Luo et al., 2024; Jiang et al., 2024) demonstrate LLMs' proficiency in code explanation through few-shot, zero-shot capabilities, and task-specific fine-tuning. (Oli et al., 2023) examines LLMs' code explanation generation, emphasizing variations due to factors like few-shot prompting and evaluating the readability and accuracy of outputs. (Jiang et al., 2024) proposes a framework combining supervised fine-tuning and reinforcement learning to enhance LLMs' self-debugging and explanation abilities. Addressing the lack of ground truths, (Li et al., 2023b) introduces a method for automatically generating explanations for <problem, solution> pairs in competitive programming, evaluating their role in assisting LLMs in problem-solving. However, these approaches often rely on single-turn generation without self-correction or iterative refinement and fail to address personalized code explanations, which are crucial (Ullah et al., 2018).

2.2 Self-Critique of LLMs

LLMs have demonstrated strong performance in various NLP tasks (Qin et al., 2023a; Guo et al., 2023; Suzgun et al., 2022), but they still suffer from issues like hallucination (Zhang et al., 2023c), unfaithful reasoning (Lyu et al., 2023), and toxicity (Shaikh et al., 2023). One promising method to address these challenges is self-correction through feedback, where the LLM adjusts its own output based on feedback (Pan et al., 2023b). Self-correction can be applied at different stages of the process. During training, feedback is used to optimize the model's parameters (Huang et al., 2022; Zelikman et al., 2022). During generation, feedback guides the LLM to adjust its output as it is being produced (Yang et al., 2022; Lightman et al., 2023). However, both approaches can be resource-intensive or challenging to implement reliably. In this paper, we focus on post-hoc self-correction (Madaan et al.,

2024; Gou et al., 2023; Chen et al., 2023b; Pan et al., 2023a; Zhang et al., 2023a; Jiang et al., 2023; Zhang et al., 2023b), which refines the output after it is generated, without modifying model parameters. This approach allows for diverse feedback from self-feedback (generated by the LLM itself) or external feedback (trained models, code interpreters, or search engines). While previous methods have been effective, none have combined feedback from both external tools and LLMs nor addressed two tasks simultaneously in an iterative framework.

2.3 Role-Playing Language Agents

Recent advancements in LLMs have significantly boosted the rise of role-playing language agents, i.e., specialized AI systems designed to simulate assigned personas (Chen et al., 2024). The methodologies for constructing role-playing language agents generally involve either parametric training (Wang et al., 2023; Shao et al., 2023; Qin et al., 2023b) or non-parametric prompting (Li et al., 2023a; Zhou et al., 2023; Gupta et al., 2023; Ma et al., 2024; Chen et al., 2024), with both approaches potentially contributing to the development process. In parametric training, role-playing language agents are pre-trained using extensive raw text (Xu et al., 2023; Gupta et al., 2023; Wang et al., 2023; Shao et al., 2023). Conversely, non-parametric prompting involves presenting role-playing instructions and examples (Zhou et al., 2023; Li et al., 2023a; Shao et al., 2023; Deshpande et al., 2023). Currently, there is no established practice of employing self-correction mechanisms to refine personalized outputs produced by these agents.

3 `FaithfulPersonaCodeX`: Faithful and Personalized Code Explanation Benchmark

Our benchmark, `FaithfulPersonaCodeX`, focuses on generating code explanations that are both faithful to the original code and personalized to the user's background. This dual emphasis addresses limitations in existing benchmarks (Tasnim Preoty, 2024; Bhattacharya et al., 2023; Sarsa et al., 2022; Oli et al., 2023; Yan et al., 2023), which typically focus on generic explanations without considering user-specific contexts. To achieve this goal, we have carefully designed our dataset collection process and evaluation metrics. In the following subsections, we detail our approach to task description, data collection, and evaluation design and compare our benchmark with existing works.

3.1 Task Description

Our benchmark focuses on generating explanations for code problem solutions that are both faithful to the original code and personalized to the user's background. Given the code s, code context p, and user profiles h, the task is to produce an explanation e that accurately reflects the code's logic while tailored to the user's preferences and knowledge level. This process can be modeled as:

$$e \sim \text{ExplainModel}(\cdot|p, s, h). \quad (1)$$

An input example is shown in Fig. 1.

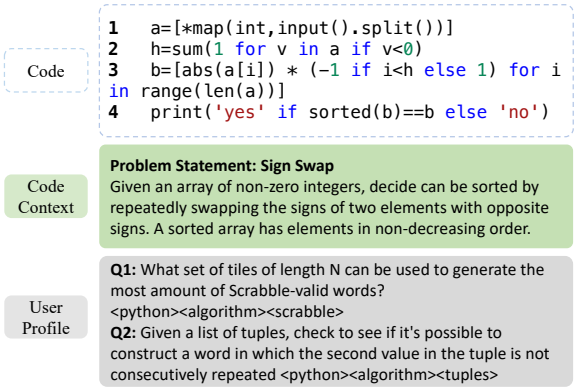

Figure 1: A code sample, code context, as well as a user profile represented by the user question history. The content is simplified.

3.2 Comparison with Existing Code Explanation Benchmarks

Existing code explanation benchmarks face significant limitations in evaluating faithful and personalized explanations. As shown in Tab. 1, many current benchmarks (e.g., Tasnim Preoty (2024), Bhattacharya et al. (2023), Yan et al. (2023)) rely on reference-based metrics like *BLEU*, *BERTScore*, and *ROUGE-L*, treating code comments or summaries as ground truth. This approach fails to capture broader context or alternative approaches (Jiang et al., 2024) and neglects personalized explanations. Some benchmarks (e.g., Sarsa et al. (2022), Oli et al. (2023)) use human evaluation, which provides ground truth but lacks scalability and extensibility for testing new methods. `FaithfulPersonaCodeX` addresses these limitations by providing a comprehensive dataset of 169 code samples with problem descriptions, test cases, and 10 distinct user profiles. It introduces novel evaluation metrics: *Pass@k* for faithfulness, and *ROUGE-L*, *Word Overlap*, and *LLM-as-a-Judge* for personalization. This approach

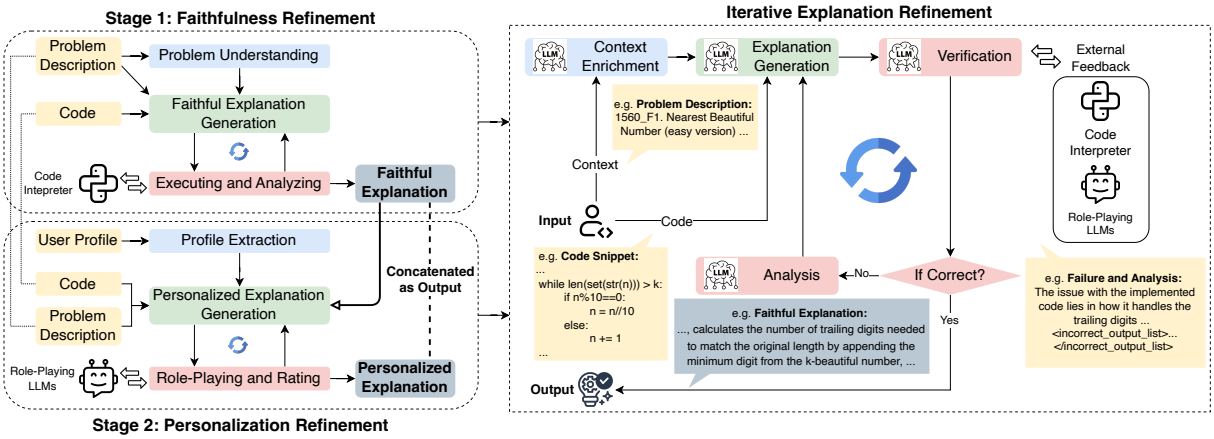

Figure 2: Illustration of DISCO. It generates code explanations through two iterative refinements: a faithfulness refinement to ensure technical accuracy, and a personalization refinement to tailor the explanation to the user's profile. The refinements operate independently to optimize their respective objectives and navigate the potential trade-off between personalization and faithfulness. The final explanation is the concatenation of faithful and personalized explanations.

eliminates the need for human-written ground truth, offers scalable evaluation, and allows dynamic assessment based on user profiles. By simultaneously evaluating faithfulness and personalization, FaithfulPersonaCodeX provides a more comprehensive benchmark for assessing code explanations, pushing the field towards more faithful and personalized explanations that cater to diverse user needs in various contexts.

Dataset	Codes	Users	Faith. Metric	Persona. Metric
(Tasnim Preoty, 2024)	50	-	BLEU; BERTScore	-
(Bhattacharya et al., 2023)	100	-	BLEU; BERTScore	-
(Sarsa et al., 2022)	4	-	Human Eval.	-
(Oli et al., 2023)	5	-	Human Eval.	-
(Yan et al., 2023)	4838	-	BERTScore; BLEU; Rouge-L	-
FaithfulPersonaCodeX	169	10	Pass@k	ROUGE-L; Word Overlap; LLM-as-a-Judge

Table 1: Faithful and personalized code explanation benchmark comparison.

3.3 Dataset Collection

The benchmark comprises two key components: code solutions and user profiles, each crucial to our evaluation framework.

Code Solutions. We collected diverse Python solutions from the CodeContests dataset (Li et al., 2022), focusing on validation (67 problems) and test (102 problems) sets to prevent data leakage. For each problem, we selected the shortest solution to accommodate LLM context limitations. Each solution includes public and private test cases for verification and evaluation and the corresponding problem description for context. Therefore, we have 169 <problem, solution> pairs.

User Profiles. To create diverse user profiles, we utilized the Stack Overflow question history dataset[1], which offers richer insights into users' programming knowledge and interests compared to traditional demographic data. While demographic data can provide some context, a user's history of programming questions more directly reflects their technical knowledge, areas of interest, and skill levels. Moreover, the question data is readily available and provides a rich, nuanced picture of a user's evolving expertise and challenges in programming. We selected 10 users who had asked at least 5 questions among the top 1000 most-viewed questions, ensuring a range of expertise levels and interests. The wordcloud of sampled user profiles are shown in Fig. 3.

Figure 3: Three user samples of different profiles. User 1 focuses on solving algorithms for programming games, such as dynamic programming; User 2 is interested in image processing; and User 3 focuses on data analysis.

[1] https://data.stackexchange.com/

3.4 Evaluation Design

Our evaluation metrics address two critical aspects: faithfulness and personalization. These metrics are designed to evaluate the quality and relevance of generated code explanations comprehensively.

Faithfulness Evaluation. We propose a novel approach using LLMs to reconstruct code based solely on the generated explanation. The reconstructed code is evaluated using the *Pass@k* metric on private test cases, effectively evaluating whether the explanation conveys accurate functional information. *Pass@k* (Qiu et al., 2024; Li et al., 2023b; Ridnik et al., 2024) measures the probability that at least one of k generated code samples passes all test cases, providing a robust measure of the explanation's ability to capture the code's functionality.

Personalization Evaluation. We employ a multi-faceted approach to capture various aspects of personalization: *ROUGE-L*, *Word Overlap* and *LLM-as-a-Judge* (Lin et al., 2023; Lin, 2004; Zheng et al., 2023). Both *ROUGE-L*, *Word Overlap* capture the overlap between the generated explanation and a user's questions on Stack Overflow, as well as the answers the user has selected as most helpful. *LLM-as-a-Judge* is to pairwisely compare the code explanation with our proposed method: `DISCO`[2].

4 Method

We present `DISCO`[3], a novel approach for generating high-quality, faithful, and personalized code explanations. The architecture of `DISCO` is illustrated in Fig. 2. To balance the potentially conflicting goals of faithfulness and personalization, `DISCO` employs a sequential approach, addressing each objective in turn. Both faithfulness and personalization refinement are based on an iterative explanation refinement process that leverages the LLMs' capability to get better results through self-correction based on external feedback.

4.1 Iterative Explanation Refinement

Inspired by the LLMs' ability to refine complex tasks iteratively (Shinn et al., 2024; Gou et al., 2023) and incorporate external feedback, we propose a multi-step process for generating high-quality code explanations as detailed in the right part of Fig. 2.

This process begins with *Context Enrichment*, where we augment the <problem, solution> pair with crucial contextual information such as the problem goal and the user knowledge domain extracted from the user profile. This enriched context forms the foundation for the subsequent explanation generation phase. In the *Explanation Generation* stage, we employ Chain-of-Thought (CoT) (Wei et al., 2022), providing the LLM with the <problem, solution> pair, enriched context, and any feedback from previous iterations. The LLM generates a code explanation and refines iteratively. To ensure continuously improving the quality of the generated explanation, we introduce novel feedback mechanisms in *Verification* and *Analysis*, which involves the LLM interacting with external tools, such as Python executors or other role-play LLMs, to evaluate the explanation against specific criteria for faithfulness and personalization. Our method identifies errors and provides revision suggestions for the explanation that does not meet the required standards. These insights feed into the iterative improvement phase, where the explanation undergoes continuous refinement. This cyclical process continues until the explanation satisfies predefined quality criteria for both faithfulness to the original code and personalization to the user's profile.

4.2 Faithful and Personalized Explanation Refinement

Our approach employs two sequential iterative refinements for faithfulness and personalization.

Faithful Refinement. This ensures that the generated explanation accurately represents the code's functionality. It begins with context enrichment, extracting problem goals, inputs, outputs, and conditions. The verification stage leverages the LLMs' code generation capability (Li et al., 2022; Ridnik et al., 2024; Ni et al., 2023) to test if the code generated from the explanation passes public test cases. Interestingly, we found that LLMs excel more in identifying code-related issues than textual problems. Consequently, our error analysis focuses on the generated verification code, using these insights to refine the explanation. This iterative process ensures high fidelity to the original code.

Personalization Refinement. This tailors the generated explanation to the individual user profile iteratively. Unlike existing role-playing LLM studies (Chen et al., 2024; Wang et al., 2023; Shao et al., 2023; Li et al., 2023a) that focus on demographic tags and conversation records, we extract user profiles from their question histories on Stack

[2]The prompt for *LLM-as-a-Judge* is in Appendix C

[3]All the prompt samples can be found in Appendix D

Overflow. This novel approach considers aspects such as programming languages, skill levels, and knowledge domains, allowing for more accurate persona representation. The verification stage employs a role-playing judging LLM to evaluate the explanation's alignment with the user's profile, guiding subsequent refinements.

Final Output. We prioritize the faithfulness refinement before the personalization refinement, ensuring that a technically accurate base explanation is then tailored to an individual user profile. This sequential approach aligns with natural cognitive processes and can adapt to scenarios where personal information is unavailable. The final output combines the faithful and personalized explanations, producing results that are both technically accurate and accessible to users with varying levels of programming expertise.

5 Experiments

5.1 Experimental Setup

Baseline Methods. To evaluate the faithfulness and personalization ability of DISCO, we employ two code explanation methods based on LLMs. **S2G** (Li et al., 2023b): The LLM is required to think step-by-step to generate explanations progressively, with detailed requirements provided for each step. **Self-Consistency** (Wang et al., 2022): The LLM generates n explanations and then ranks them based on predefined criteria, simulating a decision-making process to select the most suitable explanation. Ridnik et al. (2024) suggests that LLMs are more effective at ranking multiple options than making a single choice.

To further validate the effectiveness of our approach, we compare it with two commercial tools designed for code explanation. **ZZZ Code AI**[4]: An online AI-powered programming code explain tool. **AI Code Mentor**[5]: An explainer tool based on AI for optimizing, refactoring, and reviewing code. It is also worth mentioning that, due to cost considerations, we only evaluated these tools on the test dataset. Given the lack of personalized code explanation features in these online products, our evaluation mainly focuses on code faithfulness.

Implementation Details. We employed GPT-3.5-turbo (OpenAI, 2023) and Claude 3 Sonnet (Anthropic, 2024). To avoid generation uncertainty of LLMs (Lin et al., 2023), we sample 4 times for each problem and each chosen method. To simplify the naming, in the following content, we will refer to GPT-3.5 instead of GPT-3.5-turbo, and Claude 3 instead of Claude 3 Sonnet. It is worth mentioning that due to the cost of API calls, we only conducted experiments with Claude 3 on the test dataset.

[4] zzzcode.ai/code-explain
[5] code-mentor.ai

5.2 Automatic Evaluation

		Valid		Test	
		Pass@1	*Pass@5*	*Pass@1*	*Pass@5*
	ZZZ Code AI	-	-	16.67%	17.65%
	AI Code Mentor	-	-	29.41%	29.41%
GPT-3.5	S2G	25.11%	29.29%	21.57%	25.49%
GPT-3.5	Self-Consistency	26.08%	30.34%	22.60%	26.45%
GPT-3.5	DISCO	**30.11%**	**34.89%**	**26.52%**	**30.15%**
Claude 3	S2G	-	-	22.15%	25.77%
Claude 3	Self-Consistency	-	-	22.89%	26.61%
Claude 3	DISCO	-	-	**26.81%**	**30.53%**

Table 2: The table shows the *Pass@k* results for various methods in the validation and test sets. The values are presented as average percentages.

Faithfulness. As mentioned in § 3.4, faithfulness is evaluated by *Pass@k* (k = {1, 5}), which measures the success rate of the generated explanations in solving coding problems. As demonstrated by Tab. 2, our approach consistently achieves the highest performance across both the validation and test datasets, surpassing even specialized online products like ZZZ Code AI and AI Code Mentor. These improvements demonstrate the effectiveness of our iterative refinement process, which yields more reliable and effective code explanations. Specifically, when using GPT-3.5, our method outperforms Self-Consistency, a strong baseline, with a 3.7% absolute gain (13.99% relative) in *Pass@5* on the test dataset. Additionally, it achieves a 0.74% (2.51% relative) improvement over AI Code Mentor, a paid online code explanation tool. Similarly, Claude 3 shows comparable performance.

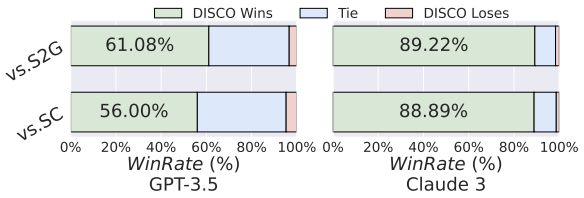

Figure 4: Results of *LLM-as-a-Judge* for DISCO against each baseline in the test set. SC is short for Self-Consistency.

Model	Set	Method	*ROUGH-L*	*Word Overlap*
GPT-3.5	Validation	S2G	0.0230	4.99%
		Self-Consistency	0.0232	5.04%
		DISCO	**0.0363**	**7.44%**
	Test	S2G	0.0227	4.86%
		Self-Consistency	0.0230	4.90%
		DISCO	**0.0361**	**7.35%**
Claude 3	Test	S2G	0.0439	11.08%
		Self-Consistency	0.0440	11.10%
		DISCO	**0.0704**	**17.94%**

Table 3: Comparison of *ROUGH-L* and *Word Overlap* for different methods in the validation and test sets. *ROUGH-L* is reported with mean values. *Word Overlap* are reported as average percentages.

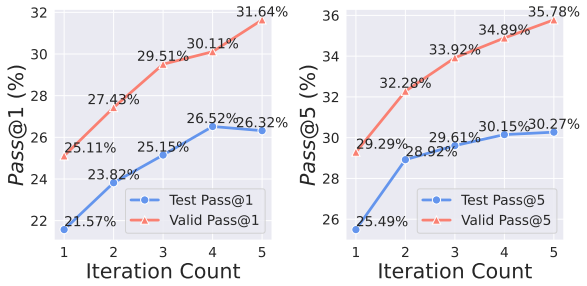

Figure 5: Hyperparameter analysis of the number of iterative refinement steps.

Personalization. As discussed in § 3.4, personalization is evaluated using *ROUGE-L*, *Word Overlap*, and *LLM-as-a-Judge*, with a focus on the alignment of generated explanations with individual user profiles. **1).** As shown in Tab. 3, our method consistently outperforms others in generating explanations that closely align with users' profiles, as measured by *ROUGE-L* and *Word Overlap*, regardless of the underlying LLM. Specifically, when using GPT-3.5, our approach achieves a significant absolute gain of 0.0131 (56.96% relative) in *ROUGE-L* and an absolute gain of 2.45% (50.00% relative) in *Word Overlap* compared to the Self-Consistency method. Even greater improvements are seen with Claude 3, where *ROUGE-L* increases by 0.0264 (60% relative) and *Word Overlap* improves by 6.84% (61.62% relative). **2).** As illustrated in Fig. 4, our method demonstrates significant superiority over competing approaches in *LLM-as-a-Judge*. Notably, with GPT-3.5, our method achieves a win rate of 61.08% on the test dataset against the robust baseline, Self-Consistency, with only a slight loss of approximately 3%. Even better results are observed with Claude 3, further reinforcing the effectiveness of our approach. Similar trends can be seen in the validation set, as detailed in Appendix B.1.

The results demonstrate that DISCO is superior in both faithfulness and personalization. The iterative refinement process not only improves the accuracy and effectiveness of the generated explanations but also ensures they are tailored to the individual user's needs. This method's dual focus on quality and personalization makes it robust to generating faithful and personalized code explanations.

5.3 Effects of Iterative Refinement

To evaluate the model's convergence speed and incremental improvements in generating high-quality explanations, we perform experiments to evaluate the effectiveness of our method by analyzing the number of iterative refinements with GPT-3.5. As illustrated in Fig. 5, increasing the number of iterative refinements from 1 to 5 enhances performance at both *Pass@1* and *Pass@5*, demonstrating that additional iterations facilitate refinement and error correction in the explanations generating process using our proposed method. Specifically, performance at *Pass@5* in the test dataset has improved from 25.49% to 30.27% from the 1st iteration to the 5th iteration. However, we found that the performance improvements become less significant between iterations 4 and 5. Therefore, considering the trade-off between cost and effectiveness, we typically select 4 iterations.

5.4 Human Evaluation

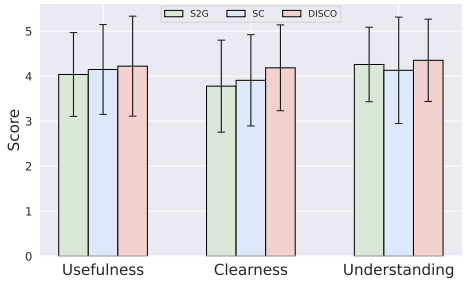

Figure 6: Human evaluation results. SC is short for Self-Consistency.

We conduct human evaluation to ensure that the generated explanations are not only accurate but also meaningful and comprehensible to end users. Specifically, we randomly sample 20 code explanation problems from the test dataset, accompanied by the corresponding <problem, code> pairs. We engage the services of 4 professional software development engineers, who are considered qualified for this task. Evaluators evaluate explanations generated by GPT 3.5 on *Usefulness*, *Clearness*,

and *Understanding*, assigning Likert scores from 1 (very poor) to 5 (excellent) (Li et al., 2023b)[6].

The results of our human evaluation demonstrate the efficacy of our proposed method, as illustrated in Fig. 6. Our findings reveal that DISCO consistently outperforms the other two approaches across all three aspects. This indicates that our method **1).** enhances the practical utility of explanations for solving code problems, **2).** effectively clarifies explanations, making them less ambiguous and easier to comprehend, and **3).** helps in conveying the core concepts more accurately.

5.5 Case Study

	Faithfulness
Initial Output	**Step-by-Step Description:** 5. Calculate the number of missing digits at the end. **High-Level Explanation:** The solution involves iterating over test cases. ... It ensures that the resulting number has no more than k different digits in its decimal representation.
Failure Analysis	The issue with the implemented code lies in how it handles the trailing digits. Instead of appending '9' to ensure at most k different digits, the correct approach is to append the minimum digit from the k-beautiful number. ...
Corrected Output	**Step-by-Step Description:** ... 5. Determine the number of trailing digits to add to n by finding the minimum digit from the k-beautiful number. **High-Level Explanation:** The solution iterates through each test case, finding the minimum k-beautiful integer x greater than or equal to a given integer n…
	Personalization
Initial Output	… This aligns with your interest in complex algorithms and data processing, showcasing how algorithms can be applied to solve specific number-related problems effectively
Failure Analysis	The explanation provided aligns well with the programmer's skills and background in Python and their interest in data analysis, image processing, and task scheduling. ... However, to improve the rating to a 5, specific examples or insights on real-world applications of the solution could be included for better understanding.
Corrected Output	Given your intermediate to advanced level in Python and your interest in data analysis, image processing, and task scheduling, ... For example, in data analysis, the concept of 'k-beautiful' numbers could be applied to ensure that certain numerical data meets specific digit constraints, ...

Figure 7: Case study of the iterative refinement for faithful and personalized code explanation. This case is generated by DISCO based on GPT-3.5.

To illustrate the practical application of our approach and demonstrate the effectiveness of DISCO in refining code explanations, we present a detailed case study that highlights the iterative improvements in faithfulness and personalization, as indicated in Fig. 7. We also observe the token count for each input and output step, and the detailed results can be seen in Appendix B.2.

Faithfulness. The corrected explanation offers a more detailed and faithful description of the code logic, particularly in the step-by-step breakdown. Taking the failure analysis into consideration, it clearly explains the process of checking and adjusting the number n based on its last digit, the calculation of the trailing digits to match the original length, and the concatenation and final output steps. This detailed breakdown helps users understand each part of the code more comprehensively, ensuring they grasp the nuances of the logic and how each condition affects the outcome. In contrast, the initial generated explanation provides a more general overview without delving deeply into the specifics of each step, which can leave gaps in understanding, particularly for users trying to follow the logic of adjusting n based on its digits.

Personalization. Following the LLM's rating, the corrected explanation excels in personalization by connecting the code's purpose to practical applications relevant to the user's interests. It illustrates how the concept of "k-beautiful" numbers can be applied in real-world scenarios, like data analysis and image processing, making the explanation more relatable and valuable. On the other hand, the initial generated explanation lacks the depth of connection to the user's specific areas of interest. While it mentions complex algorithms and data processing, it does not provide tangible examples or applications that resonate with the user's background and skills.

In conclusion, DISCO improves code explanations through iterative cycles. Each iteration makes the explanation more accurate, detailed, and aligned with the user's needs. This process helps address errors, enhances clarity, and ensures both faithfulness to the code and relevance to the user's interests.

6 Conclusion

This paper presents FaithfulPersonaCodeX, a novel benchmark for evaluating code explanations, and DISCO, an innovative method for generating personalized and faithful explanations. FaithfulPersonaCodeX addresses limitations in existing benchmarks by incorporating diverse code samples, user profiles, and multifaceted metrics to assess both faithfulness and personalization. DISCO leverages a self-critique mechanism and a two-stage optimization process to balance these competing objectives. Our experimental results demonstrate the effectiveness of this approach, outperforming existing baselines and advancing the field toward more comprehensive, user-tailored code explanations. By addressing the dual challenges of faithfulness and personalization, this work improves code comprehension across various contexts, from education to professional software development.

[6]Detailed questions can be found in Appendix A

7 Limitations

One primary limitation of this work is that code problems in our benchmark were mainly selected from only one source dataset - CodeContest. So, it may be unclear whether our method can be further generalized well to other problem sources, which may contain different levels of code problems. However, we assume that competitive-level programming problems in our benchmark are well-defined so that the distribution shift will not be significant between sources. The good news is that we tested a couple of LLM backbones in this paper, so we already mitigated the risk of potential performance deviation by different LLM backbones.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family.

Paheli Bhattacharya, Manojit Chakraborty, Kartheek NSN Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. 2023. Exploring large language models for code explanation. *arXiv preprint arXiv:2310.16673*.

Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023a. Gptutor: a chatgpt-powered programming tool for code explanation. In *International Conference on Artificial Intelligence in Education*, pages 321–327. Springer.

Jiangjie Chen, Xintao Wang, Rui Xu, Siyu Yuan, Yikai Zhang, Wei Shi, Jian Xie, Shuang Li, Ruihan Yang, Tinghui Zhu, et al. 2024. From persona to personalization: A survey on role-playing language agents. *arXiv preprint arXiv:2404.18231*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Ameet Deshpande, Vishvak Murahari, Tanmay Rajpurohit, Ashwin Kalyan, and Karthik Narasimhan. 2023. Toxicity in chatgpt: Analyzing persona-assigned language models. *arXiv preprint arXiv:2304.05335*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arXiv:2301.07597*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, and Ming Zhou. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.01679*.

Shashank Gupta, Vaishnavi Shrivastava, Ameet Deshpande, Ashwin Kalyan, Peter Clark, Ashish Sabharwal, and Tushar Khot. 2023. Bias runs deep: Implicit reasoning biases in persona-assigned llms. *arXiv preprint arXiv:2311.04892*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Hamel Husain, Ho-Hsiang Siddiqui, Huy Feng, Usama Chowdhury, Eric Hammond, Boris Tran, Vinod Mangal, Dima Kang, and Ankur Taly. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. In *arXiv preprint arXiv:1909.09436*.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Training llms to better self-debug and explain code. *arXiv preprint arXiv:2405.18649*.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857.

Cheng Li, Ziang Leng, Chenxi Yan, Junyi Shen, Hao Wang, Weishi Mi, Yaying Fei, Xiaoyang Feng, Song Yan, HaoSheng Wang, et al. 2023a. Chatharuhi: Reviving anime character in reality via large language model. *arXiv preprint arXiv:2308.09597*.

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023b. Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Zhen Lin, Shubhendu Trivedi, and Jimeng Sun. 2023. Generating with confidence: Uncertainty quantification for black-box large language models. *arXiv preprint arXiv:2305.19187*.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*.

Xiao Ma, Swaroop Mishra, Ariel Liu, Sophie Ying Su, Jilin Chen, Chinmay Kulkarni, Heng-Tze Cheng, Quoc Le, and Ed Chi. 2024. Beyond chatbots: Explorellm for structured thoughts and personalized model responses. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pages 1–12.

Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. 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*, pages 931–937.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Priti Oli, Rabin Banjade, Jeevan Chapagain, and Vasile Rus. 2023. The behavior of large language models when prompted to generate code explanations. *arXiv preprint arXiv:2311.01490*.

2023 OpenAI. 2023. Introducing ChatGPT. https://openai.com/index/chatgpt/.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023a. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023b. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*.

Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023a. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476*.

Ruiyang Qin, Jun Xia, Zhenge Jia, Meng Jiang, Ahmed Abbasi, Peipei Zhou, Jingtong Hu, and Yiyu Shi. 2023b. Enabling on-device large language model personalization with self-supervised data selection and synthesis. *arXiv preprint arXiv:2311.12275*.

Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*.

Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 13(5):1–44.

Jonan Richards and Mairieli Wessel. 2024. What you need is what you get: Theory of mind for an llm-based code understanding assistant. *arXiv preprint arXiv:2408.04477*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 27–43.

Sarah Schwettmann, Tamar Shaham, Joanna Materzynska, Neil Chowdhury, Shuang Li, Jacob Andreas, David Bau, and Antonio Torralba. 2024. Find: A function description benchmark for evaluating interpretability methods. *Advances in Neural Information Processing Systems*, 36.

Omar Shaikh, Hongxin Zhang, William Held, Michael Bernstein, and Diyi Yang. 2023. On second thought, let's not think step by step! bias and toxicity in zero-shot reasoning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4454–4470.

Yunfan Shao, Linyang Li, Junqi Dai, and Xipeng Qiu. 2023. Character-llm: A trainable agent for role-playing. *arXiv preprint arXiv:2310.10158*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Anika Tasnim Preoty. 2024. Implementing zero-shot learning in code-llm for effective code explanation generation. Master's thesis, A. Tasnim Preoty.

Zahid Ullah, Adidah Lajis, Mona Jamjoom, Abdulrahman Altalhi, Abdullah Al-Ghamdi, and Farrukh Saleem. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, 26(6):2328–2341.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Man Zhang, et al. 2023. Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. *arXiv preprint arXiv:2310.00746*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*.

Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. 2023. Expertprompting: Instructing large language models to be distinguished experts. *arXiv preprint arXiv:2305.14688*.

Hao Yan, Thomas D Latoza, and Ziyu Yao. 2024a. Intelliexplain: Enhancing interactive code generation through natural language explanations for non-professional programmers. *arXiv preprint arXiv:2405.10250*.

Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024b. Ivie: Lightweight anchored explanations of just-generated code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–15.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*.

Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. *arXiv preprint arXiv:2205.12443*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023b. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784.

Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A Smith. 2023c. How language model hallucinations can snowball. *arXiv preprint arXiv:2305.13534*.

Yadong Zhang, Shaoguang Mao, Tao Ge, Xun Wang, Adrian de Wynter, Yan Xia, Wenshan Wu, Ting Song, Man Lan, and Furu Wei. 2024. Llm as a mastermind: A survey of strategic reasoning with large language models. *arXiv preprint arXiv:2404.01230*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023.

Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

Jinfeng Zhou, Zhuang Chen, Dazhen Wan, Bosi Wen, Yi Song, Jifan Yu, Yongkang Huang, Libiao Peng, Jiaming Yang, Xiyao Xiao, et al. 2023. Characterglm: Customizing chinese conversational ai characters with large language models. *arXiv preprint arXiv:2311.16832*.

A Human Evaluation Questions

To comprehensively evaluate the quality of the explanation, we ask the following three questions: *Usefulness*, *Clearness*, and *Understanding*, to ensure that the explanation effectively solves the problem, is clear and unambiguous, and accurately captures the key idea behind the solution.

- *Usefulness* - How useful is the explanation for solving the problem?
- *Clearness* - How clear and unambiguous is the explanation?
- *Understanding* - How well does the explanation capture the key idea behind the solution?

B Experiment Results

B.1 Win Rate in the Validation Set

Fig. 8 indicates the results of LLM-as-a-Judge for various methods in the validation set using GPT-3.5.

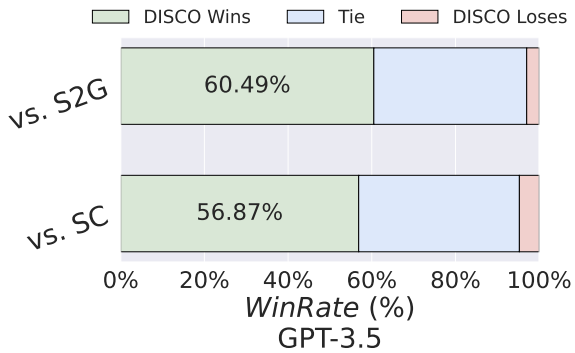

Figure 8: Results of *LLM-as-a-Judge* for DISCO against each baseline in the validation set using GPT-3.5. SC is short for Self-Consistency.

B.2 Token Count

Tab. 4 shows the average token count for each step in the process.

Type	Stage	Avg Token #
Context Enrichment	Problem Understanding	142.93
Input	Faithfulness Input	1128.75
Verification and Analysis	Failure Analysis	126.09
Explanation Generation	Faithful Explanation	202.00
Context Enrichment	Profile Extraction	145.51
Input	Personalization Input	1119.70
Verification and Analysis	Role-Playing and Rating	92.33
Explanation Generation	Personalized Explanation	182.92

Table 4: Average token count of each step in the process.

C LLM-as-a-Judge Prompt

LLM-as-a-Judge Prompt

Imagine you are a programmer with the following inquiry history:
{User Question History}
You are given a code contest problem and an accepted correct solution code:
{Problem Description}
{Code Solution}
You are trying to understand the code.
You have two code explanations to choose from:
{Personalized Explanation A}
{Personalized Explanation B}
Now create a leaderboard by ranking the two code explanations based on your skill level, background and preferences inferred from the content in the inquiry history, to determine which explanation is more helpful and informative to you.

D Explanation Generation Prompt

Problem Understanding Prompt

You are given a code contest problem: {Problem Description}
Given the code contest problem, you should reflect on the problem and describe it in your own words. Pay attention to small details, nuances, notes, and examples in the problem description.

Faithful Explanation Prompt

Your task is to comprehend a competitive programming problem and interpret its solution.
You are given a code contest problem, and a self-reflection on the problem:
{Problem Description}
{Problem Understanding}
Additionally, you are given an accepted correct solution:
{Code Solution}

Let's think step-by-step.
- First, provide a step-by-step description of the solution.
- Next, give a high-level explanation of the solution.

Verification Prompt

You are given a code contest problem:
{Problem Description} The following is a hint that can lead to one correct solution of the problem:
{Imperfect Faithful Explanation}

Your task is to read and understand the problem, analyze the hint and how to use it to solve the problem, think of a solution accordingly and complete the python code of the solution.

Failure Analysis Prompt

You are given a code contest problem and a self-reflection on the problem:
{Problem Description}
{Problem Understanding}
Additionnally, you are given an accepted correct solution:
{Code Solution}
A Python code solution was generated for the problem:
{Reconstructed Code}
However, when running on the given input, the code solution above failed to produce the expected output:
{Incorrect Output}
Your task is to analyze the failure.
Let's think step by step.

Faithfulness Refinement Prompt

You are given a code contest problem, a self-reflection on the problem, an input list of test cases and an expected output list:
{Problem Description}
{Problem Understanding}
You are also given one correct solution to the problem and its explanation:
{Code Solution}
{Imperfect Faithful Explanation}
Your task is to revise the provided explanation based on the following feedback.
A coder attempted to implement a solution with the hint of the explanation above. However, the code did not yield the correct output:
{Reconstructed Code}
{Incorrect Output}
Here's also a failure analysis of the coder's solution:
{Failure Analysis}
Please revise the provided explanation, considering the failure analysis of the coder's solution.
When other coders read your revised explanation, they should avoid the same mistakes made by the coder who failed to produce the correct solution.

Profile Extraction Prompt

Given the user's Stack Overflow question history provided below, analyze and infer the user's programming skills and background.
{User Question History}
Consider the following aspects in your analysis: Programming Languages, Skill Level, Topics of Interest, Problem-Solving Approach, Experience.

Personalized Explanation Prompt

Your task is to personalize the explanation of the solution based on the following user's programming skills and background:
{Extracted Profile}
You are given a code contest problem:
{Problem Description}
You are also given one correct solution to the problem and its explanation:
{Code Solution}
{Faithful Explanation}
You should provide a personalized explanation of the solution based on the user's programming skills and background and the explanation provided above.

Personalization Refinement Prompt

Your are given a code contest problem:
{Problem Description}
You are also given one correct solution to the problem and its explanation:
{Code Solution}
{Faithful Explanation}
A personalized explanation of the solution was generated for a user with the following programming skills and background:
{Extracted Profile}
This is the generated personalized explanation:
{Imperfect Personalized Explanation}
The user rated the personalized explanation and provided some revision suggestions:
{Rating}
Please revise the personalized explanation based on the user's feedback so that the user will rate the revised personalized explanation higher.

Role-Playing and Rating Prompt

Imagine you are a programmer with the following individual programming skills and background:
{Extracted Profile}
Your are given a code contest problem:
{Problem Description}
You are also given an accepted correct solution:
{Code Solution}
Your task is to rate the following personalized explanation of the solution based on how well it aligns with your programming skills and background:
{Imperfect Personalized Explanation}
The score should be in the range of 1 to 5. If the rating is under 5, please provide some revision suggestions in the reasoning section.