Uncovering Effective Steering Strategies in Code-Generating LLMs with Socratic Feedback

ZHENG ZHANG[*], University of Notre Dame, USA
ALEX C. WILLIAMS, AWS AI, Amazon, USA
JONATHAN BUCK, AWS AI, Amazon, USA
XIAOPENG LI, AWS AI, Amazon, USA
MATTHEW LEASE, AWS AI, Amazon, USA
LI ERRAN LI, AWS AI, Amazon, USA

Large Language Models (LLMs) are rapidly becoming commonplace tools for generating code solutions for complex programming problems. Alongside their widespread availability, the adoption of code-generating LLMs has been driven by a myriad of code-related features (e.g., self-debugging) that empower their ability to generate functionally correct code. Despite these recent advances, LLMs remain largely incapable of solving many programming problems of significant complexity unless otherwise steered through human feedback. In this paper, we uncover the strategies that professional programmers employ in steering code-generating LLMs toward successful outcomes. We present findings from a study in which 38 professional programmers used natural language to steer GPT-4 in generating solutions for programming problems of varying difficulty. We motivate our study design and analysis through the lens of *Socratic Feedback*, an inversion of the popular *Socratic Questioning* paradigm that frames professional programmers as "model teachers" to code-generating LLMs. Through an analysis of 80 conversational transcripts from interactions with GPT-4, we map observed error-feedback pairs to four conceptual stages of interactive steering in the context of code generation: *Understanding*, *Planning*, *Implementation*, and *Testing*. We ground our data analysis in these four stages and conclude by discussing their implications for improving the performance of code-generating LLMs through combined lens of user experience and functional utility.

CCS Concepts: • **Human-centered computing → Empirical studies in HCI**.

Additional Key Words and Phrases: Large language models, code generation, steering.

ACM Reference Format:
Zheng Zhang, Alex C. Williams, Jonathan Buck, Xiaopeng Li, Matthew Lease, and Li Erran Li. 2024. Uncovering Effective Steering Strategies in Code-Generating LLMs with Socratic Feedback. In *CSCW'24 Workshop on Supporting User Engagement in Testing, Auditing, and Contesting AI, October 15, 2023, Minneapolis, MN, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Recent advances in natural language processing (NLP) and automated code generation have combined to drive the development of Large Language Models (LLMs) that can generate code via natural language prompts. Empirical studies suggest that LLMs can provide significant gains to productivity and has transformed practices of daily programming. Programmers have frequently relied on those publicly available LLMs (e.g. ChatGPT [21]) and coding assistants (e,g. Copilot [7]) for code suggestion and brainstorming in their work and study.

Nonetheless, code-generating LLMs reportedly still struggle with intricate coding challenges that require a deep understanding of the problem, task breakdown, and the nuanced application of algorithms and libraries within given specification [3, 9, 23]. Recently, some advancements have revealed that newer LLMs possess a *self-debugging* ability [8,

[*]Work completed during an internship at Amazon.

23]. This lets them enhance code generation by iteratively analyzing errors from previous program attempts based on unit test outcomes, akin to common trial-and-error strategy used by human programmers. Yet, entirely relying on self-debugging capability comes with two major vulnerabilities. First, the model might erroneously pinpoint underlying reason for code failure. Moreover, even if the model correctly identifies the failure, it may struggle with generating contingent feedback to effectively steer the following code refinement. These obstacles account for the modest performance improvements achieved by self-debugging frameworks in the context of complex programming tasks, such as LeetCode's medium and hard-level problems [8, 23].

In this paper, we conducted an empirical study that understands how professional programmers could steer a self-debug capable model, GPT-4[1], to generate functionally correct code for programming problems that it failed to solve alone. Specifically, we encouraged programmers to employ the Socratic feedback approach that is commonly used in argumentation and tutoring. Instead of directly providing answers, this approach leverages targeted questions or prompts to ignite critical thinking and empower learners to formulate their own solutions. This is akin to the dynamic in college programming tutoring sessions, where the instructor offers incremental feedback corresponding to the learner's most recent attempt. Meanwhile, the learner is required to undergo several rounds of debugging efforts before seeking additional feedback. Our study with 38 professional programmers found that GPT-4 is able to solve originally failed competition-level programming problems with a few rounds of human Socratic feedback. Moreover, through programmers' self-annotations on the conversation log data, we uncovered a variety of error types that GPT-4 committed during both the instruction-following and self-debugging phases. We also identified a group of Socratic feedback techniques that programmers employed to guide the Language Learning Model (LLM). Furthermore, we analyzed programmers' challenges in steering GPT-4 for coding tasks.

2 RELATED WORK

2.1 LLM-enabled Code Generation

Recent studies [7, 15, 20, 22] suggest that incorporating code into training data enables general-purpose LLMs to generate programs from natural language prompts or to complete incomplete code snippets. Alternatively, specialized models like Codex [7], AlphaCode [**?**], StarCoder [15], and Code LLAMA [22] have also been developed or fine-tuned specifically for coding tasks. Though they have achieved SOTA performance on code generation benchmarks [7, 30], LLMs still exhibit limited performance on medium and hard competition-level programming problems [20]. These complex problems typically require a programmer's adept skills in understanding, planning, and implementing sophisticated reasoning tasks.

To address this limitation, some works used prompt-based techniques to boost LLMs' reasoning for correct code. For example, strategies like the chain-of-thought and tree-of-thought [**?**] were employed to prompt models to break down the planning process into manageable intermediate subproblems. Additionally, self-debugging or reflection techniques [8, 23**?**] encouraged models to analyze their own outputs and divide the debugging process into stages of code explanation and self-feedback generation. Then LLMs refined their planning and execution grounded on the insights obtained from their self-generated feedback. Besides stimulating models' self-reflection, some works used human prompts to support the code refinement process. For example, Austin et al. [3] explored human-model collaborative coding on MBPP dataset. They found that LLMs can improve or correct code based on human feedback, benefiting from human clarification of under-specified prompts and correction of small context errors. Apart from

[1]We used Advanced Data Analysis plugin of GPT-4

incorporating human feedback as prompts, Chen et al. [6] improved CodeGen using imitation learning from human language feedback, where human feedback is used to learn a refinement model that generates modification from human feedback and previous incorrect code.

Our study setting echoes Austin's work, where programmers engage in a dialogue, offering iterative natural language feedback to LLMs. However, our focus diverges as we concentrate on a self-debug capable model and tackle competition-level problems, which are notably more complex than those found in the MBPP dataset. Besides, we systematically identified the error and human feedback strategy taxonomy and analyzed GPT-4's steerability in the context of code problem-solving.

2.2 Socratic Questioning

Different from conventional prompting strategies which 'teach' LLMs to interpolate existing knowledge, Socratic reasoning 'coaches' LLMs and stimulates them to extrapolate new knowledge. Chang et al. [5] investigated the application of diverse Socratic methods to craft effective prompt templates for interacting with LLMs. By integrating techniques like *definition, elenchus, dialectic, maieutics, and counterfactual reasoning*, they showcased their application in tasks related to inductive, deductive, and abductive reasoning. Besides, several self-directed Socratic questioning frameworks [**? ?**] were proposed to encourage LLMs to recursively break down complex reasoning problems into solvable sub-problems.

On the other hand, several works attempted to use LLMs to generate Socratic questions in dialogue that guides human learners to solve challenging problems such as math word and code problems [1, 2, 24]. For example, AI-Hossami et al. [1] developed a benchmark dataset that tests LLM ability to help a novice programmer fix buggy solutions to simple coding problems via Socratic questioning. They also found that while GPT-4 performed much better than GPT-3.5, its precision and recall still fall short of human programmer's abilities.

To the best of our knowledge, our research is pioneering in investigating the application of human Socratic feedback to improve the code generation capabilities of LLMs. We discern the various forms of Socratic feedback employed by human programmers during guiding conversations and evaluate their efficacy in addressing existing errors.

3 A STUDY OF STEERING BEHAVIOR WITH CODE-GENERATING LANGUAGE MODELS

Code-generating LLMs can be prompted through natural language to generate solution implementations for complex programming problems. Recent feature extensions to commercial LLMs reportedly improve their ability to generate code with processes, such as *self-debugging*, that allow these models to iteratively debug and correct its generated solutions. However, even in the presence of these features, studies have continued to find that the code-generating LLMs typically achieve successful outcomes primarily through multi-turn conversation rather than single-turn conversation. Research has specifically indicated that the multi-turn nature of these conversations is driven by code-generating LLMs being unable to "self-correct" its errors without some form of human intervention, direction, or feedback [12].

3.1 Research Questions

Code-generating LLMs must be *steered* with natural language in order to generate solutions that adequately solve the problem. Within this use-case, "steering" can be characterized as a dyadic sequence of *generative errors* and *human feedback*. We hypothesize that there exists common sequences of steering behavior *steering strategies* used by programmers

Our study aims to uncover the steering strategies employed by professional programmers generative errors, Our study aims to address the following four research questions:

(1) What types of generative errors are observed by programmers when using code-generating LLMs?
(2) What types of feedback are used by programmers to resolve errors produced by code-generating LLMs?
(3) What types of steering strategies do programmers employ?
(4) What empirical lessons and user challenges we can learn?

3.2 Study Design

We designed and conducted an empirical study in which professional programmers used the Generative Pre-trained Transformer 4 (GPT-4) model to generate functionally correct Python code for coding problems related to algorithms and data structures. We chose GPT-4 on the basis of its ease-of-use, availability, and technical maturity in comparison to other alternatives that were available at the time of our study (e.g., Meta's Llama [26]). Furthermore, the "*Code Interpreter*" plugin[2] [18] allows end-users to instruct the model to execute and evaluate code via a Python interpreter that runs in a sandboxed execution environment. Through this expanded infrastructure, GPT-4 can handle prompts that instruct it to *self-debug* its code (i.e., modifying its generated code on the basis of errors that occurred during the program's execution or testing) [8]. Along these functionalities necessary for studying code generation, our model choice is driven by prior work that studied self-debugging with earlier models (e.g., GPT 3.5 Turbo) and acknowledged GPT-4 as an important area of study for future research in self-debugging [14].

3.2.1 A Curated Dataset of Unseen Programming Problems. By design, our study aims to study aspects of multi-turn conversation that only take place when a code-generating LLM fails to produce a satisfactory solution in its first response to the user. Benchmarking datasets are recognized as the gold standard for evaluating code generation functionality [3, 7, 16, 31]. However, the widespread availability of these datasets allows them to be easily consumed as potential use-cases for fine-tuning. In order to maximize GPT-4's inability to produce solutions in a single-turn, we curated a dataset of 110 recently created programming problem descriptions and solutions from LeetCode and CodeForce. Dataset creation was initiated by sorting LeetCode's available problems by their creation date and sampling a total of 100 problems across its difficulty category tags (i.e., 30 "Easy" problems, 40 "Medium" problems, and 30 "Hard" problems). Through a pilot study, we observed that GPT-4 was noticeably performant in solving LeetCode's "Easy" problems through single-turn conversation. We therefore chose to augment our dataset with an additional set of 10 problems from CodeForce that were analogous to LeetCode's "Easy" problems and that we verified could not be solved in a single turn. At the time of study, the dataset included a total of 110 unique problems spanning 39 algorithmic topics.

3.2.2 A One-Shot Prompt Template for Generating Code. Prior studies have illustrated that people can encounter a myriad of challenges related to prompt engineering with code-generating LLMs [13]. This is particularly true for "non-AI experts" who may fail to understand how to engage in prompt engineering [29]. In order to help participants initiate conversations with GPT-4, we provided participants with a prompt template with predetermined placeholders for various aspects of the coding problem, including its description, function signature, test cases, and constraints. The template also included an "instructions" section that described how the LLM should outline the step-by-step plan of its planned solution, provide the implementation of this solution in Python code, evaluate the success of provided test cases, and engage in an iterative process of self-debugging if any of the test cases fail. In order to avoid issues

[2]The "Code Interpreter" plugin was renamed to "Advanced Data Analysis" with the ChatGPT Enterprise announcement on August 28, 2023. [19]

related to streaming context length, we designed the prompt to engage in no more than five iterations of self-debugging. The template concluded with a statement that asked the model to report the five types of human feedback that would be most beneficial for improving the generated solution, regardless of whether the solution successfully passed all provided unit tests.

3.3 Aligning the Socratic Method to Human Feedback

In order to ease the burden of communicating with GPT-4, we designed four unique types of human feedback that draw inspiration from the Socratic method. For brevity, we refer to these types of feedback simply as *"Socratic feedback"*. Below, we describe how each type of feedback draws inspiration from a particular methodological abstraction of the Socratic method [5] while simultaneously mapping to an existing prompting technique that has already been studied in the context of code-generating LLMs. All templates were presented to participants being "optional", and they were encouraged to communicate with GPT-4 however they best see fit. Furthermore, templates were framed as "examples" of communication while simultaneously illustrating the ways in which GPT-4 might be prompted.

- **Definitive Feedback**: Definitive feedback is a reflection of the *Method of Definition* abstraction that focuses on understanding *"What is the object?"* [4]. In our scenario, we prepared a template that prompts participants to give feedback on problem requirements that GPT-4 does not satisfy.
- **Elenctic Feedback**: Elenctic feedback draws inspiration from the Socratic method's *Method of Elenchus* and aims to help learner "develop a deeper understanding of the validity of the reasons and conclusion, and to identify any potential weaknesses or flaws in the arguments" [5]. We prompted our participants to offer Elenctic feedback by identifying flaws in the GPT-4's program.
- **Dialectic Feedback**: Dialectic feedback involves "exploring opposing viewpoints to arrive at a deeper understanding of a subject" [11]. A fitting comparison for opposing viewpoints in coding tasks is the occurrence of failed tests. Therefore, we provide templates that prompt participants to reiterate or explain failed tests.
- **Maieutic Feedback**: Maieutic feedback aims to clue students in bringing forth their understanding [5]. As such, we provided templates that encouraged participants to give hints to GPT-4.

3.4 Study Platform

OpenAI allows users to interface with GPT-4 either through the ChatGPT web interface or through a standard web API. While each type of interface provides the same set of general functionality, the GPT-4 plugin can only be used via the ChatGPT web interface. This limitation drove us to develop a web application that navigated participants to complete task (Section 3.5) remotely. The front-end was developed using React, while the back-end utilized the Flask framework. We used DynamoDB to for data logging.

This web application comprises five distinct pages dedicated to background surveying, problem exploration and selection, searching for ground truth solutions, interacting with the model, and annotating conversations. Participants were instructed to view a tutorial video to familiarize themselves with the website's usage.

3.5 Task Procedure

Study participation was initiated by providing participants with a URL to our study platform, which allowed them to asynchronously participate at their own leisure. Upon entering the study website, participants were asked to complete a pre-study survey. After completing the survey, participants were instructed to watch a brief 5-minute tutorial video that

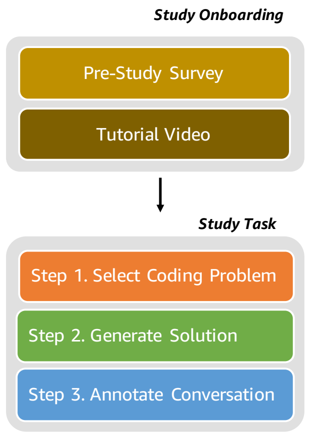
Study Onboarding
Pre-Study Survey
Tutorial Video
Study Task
Step 1. Select Coding Problem
Step 2. Generate Solution
Step 3. Annotate Conversation

Fig. 1. An overview of the study's procedure.

described the study's steps. Participants were then asked to identify a problem to begin solving with OpenAI's GPT-4. Upon problem selection, participants were provided with the problem's source webpage (e.g., on LeetCode) in a separate browser tab, asked to find and review the problem's known solution, and confirm their understanding of the solution's implementation, which they should steer GPT-4 toward solving. After providing confirmation, participants were asked to open the ChatGPT interface and initiate a conversation using the platform's provided self-debugging prompt template for starting conversations. Participants were asked to iteratively steer the GPT-4 model as they see fit. Participants were instructed to stop conversing with GPT-4 if the system generated a correct solution or if the system had failed to generate a solution after 10 iterations of conversation. After reaching either of these exit conditions, participants were asked to conclude the study by collecting qualitative and quantitative annotation data about conversation that they had with GPT-4.

3.6 Recruitment and Participants

We employed snowball sampling [17] to facilitate study recruitment. Our recruitment campaign was initiated by posting study advertisements in a team-based instant messaging application at a large technology corporation. Advertisements for participation were specifically targeted at company-wide communication channels that focused explicitly on practical applications of LLMs. The study platform facilitated participation was permitted for a duration of four weeks during August 2023.

A total of 38 individuals were recruited through our recruitment campaign. The participants held various roles such as software development engineer, software architect, research scientist, and software development engineer intern, with the majority boasting substantial programming expertise. Furthermore, a significant portion of the participants had a familiarity level of intermediate or higher with both algorithms and LLM.

3.7 Study Data

We collected the following types of data in our study:

- **Demographic Data**. We employed a pre-study survey that inquired about their job title, programming background, proficiency with algorithms and data structures, and familiarity with large language models.

- **Conversation Log Data**. Each study task concluded by asking participants to provide the *Share Chat* URL that allows others to view their conversation in the web browser. From this rendered webpage, we extract the conversational transcripts between each participant and ChatGPT.
- **Message-Level Annotation Data**. For We collect message-level annotations For each human turn in the conversation, participants were asked to describe their steering strategy used in the turn and rated the perceived effectiveness of their feedback in aiding the model to rectify errors from the preceding AI turn. For every AI turn, participants were prompted to detail the primary errors leading to program failure during both the instruction-following and self-debug phases. They also evaluated the accuracy level of the final program showcased in the AI's response. On a broader dialog scale, participants rated the general easiness of the entire steering process. Furthermore, they provided insights into their overarching methodology for choosing strategies during the conversation and shared their personal perceptions of instances where their strategies either aided or hindered the model's modifications.
- **Conversation-Level Annotation Data**. We also asked our participants to describe their overall steering strategy used in each conversation with GPT-4. Plus, we also asked them to rate the perceived easiness of steering for the conversation, and their perception of when their feedback started helping the model get closer to the right answer or even make a worse generation. Lastly, we asked participants to describe their challenges encountered in steering GPT-4.

3.8 Analysis Method

3.8.1 Thematic analysis of error type. To address RQ1, two researchers carried out a thematic analysis of the annotations made by participants about the 'instruction-following errors' and 'self-debug errors' that arose during each unsuccessful AI turn. Specifically, we drew on the axial and iterative coding methodology as seen in previous turn-level conversation analyses [10, 25]. In the initial phase, each annotator reviewed all turn-level error annotation data and perform open coding upon them. For each unsuccessful AI turn that received error annotation, annotators also revisited the corresponding AI message within conversation log data and examine if the participant annotation appropriately captured all the errors present. If not, the code was adjusted to better align with the actual AI message errors. After developing the initial codebook, annotators sat together to discuss the codes of instruction-following error and self-debug error and reached a consensus in the preliminary version of the codebook. Then, they reannotated the errors in each unsuccessful AI turn with the refined codebook.

3.8.2 Thematic analysis of steering strategy type. In addressing RQ2, we undertook a similar iterative thematic analysis focused on annotations detailing the human-steering strategy during each human turn. Annotators were tasked with coding based on the subsequent queries: *(1) What type of information did the human-steering feedback aim to provide to model? (2) Which Socratic questioning approach did the human-steering feedback resonate with?*. Each annotator first developed their own codebook independently and then collaborated to refine the codebook. They then reannotated each human-steering message with the refined codebook.

3.8.3 Qualitative analysis of steering strategy effectiveness. We conducted qualitative analysis on participants' open-ended feedback regarding their selection of steering strategies during each conversation, as well as their individual perspectives on the instances where their steering feedback either assisted or failed to benefit the model. While analyzing, we also revisited the conversation log data to understand the specific context of each interaction. Our qualitative examination sought to address these key questions : *(1) What distinguishes successful conversations from the unsuccessful*

Table 1. Themes and categories of code generation issues reported by participants.

Theme	Code	Category	"The model ..."
Understand	IU1	Consideration	... fails to consider an explicit solution requirement.
	IU2	Interpretation	... fails to interpret an explicit solution requirement.
Plan	IP1	Approach Optimization	... fails to produce a solution that is efficient.
	IP2	Approach Application	... fails to produce a solution that applies a suitable approach.
	IP3	Approach Reversion	... fails to produce a solution that uses hasn't already been suggested.
	IP4	Approach Retention	... fails to produce a solution that uses an alternative approach.
Implement	II1	Error Location	... generates a solution that has errors in a particular code region.
	II2	Error Edge-Case	... generates a solution that fails with a particular edge case.
Evaluate	IE1	Test Generation	... evaluates its solution with incorrect self-generated test case requirements.
	IE2	Test Contradiction	... evaluates its solution with contradictory test case requirements.

ones? (2) How do participants select their steering strategies during the dialogue? (3) What practical insights can we glean about effective Socratic steering?

4 PRELIMINARY RESULTS

The study's 38 participants generated a total of 80 conversations with OpenAI's Code Interpreter through the study's online platform. On average, each conversation included 2.4 messages (SD=1.8) sent by the user. When broken down by problem difficulty, participants provided 1.33 (SD=1.63) feedback for easy problems, 2.8 (SD=1.91) for medium problems, and 2.0 (SD=1.52) for hard problems.

On average, the duration of each conversational phase was 26 minutes. Alongside the conversational period, participants spent an average of 7 minutes completing the pre-conversation problem selection phase and an average of 10 minutes completing the post-conversation annotation phase The total duration for all three stages was, on average, 42.4 minutes. On average, participants gave 2.35 (SD=1.83) pieces of feedback per conversation. When broken down by problem difficulty, participants provided 1.33 (SD=1.63) feedback for easy problems, 2.8 (SD=1.91) for medium problems, and 2.0 (SD=1.52) for hard problems.

The distribution of results of steering conversation is shown in Figure 2. Participants collectively assisted the Code Interpreter in resolving 65% (33 out of 51) coding challenges it couldn't tackle on its own. On average, the discussions spanned 2.78 (SD=2.36) turns, with successful interactions averaging 1.92 (SD=1.38) turns. This suggests that even a slight human intervention using the Socratic feedback method can guide the Code Interpreter toward accurate solutions for problems it initially found challenging.

Among the 80 tasks completed by our participants, we observe that GPT-4 generated a correct solution for 29 problems (36%). Therefore, for this section, we limit our analysis to the remaining 51 tasks that include examples of steering behavior.

4.1 Error Type Taxonomy

Our thematic analysis identified the following error types at the instruction-following and self-debugging stages (Table 1 and 2).

4.2 Socratic feedback Taxonomy

Our thematic analysis identified the following steering strategy (Table 3) types that human participants used in our study.

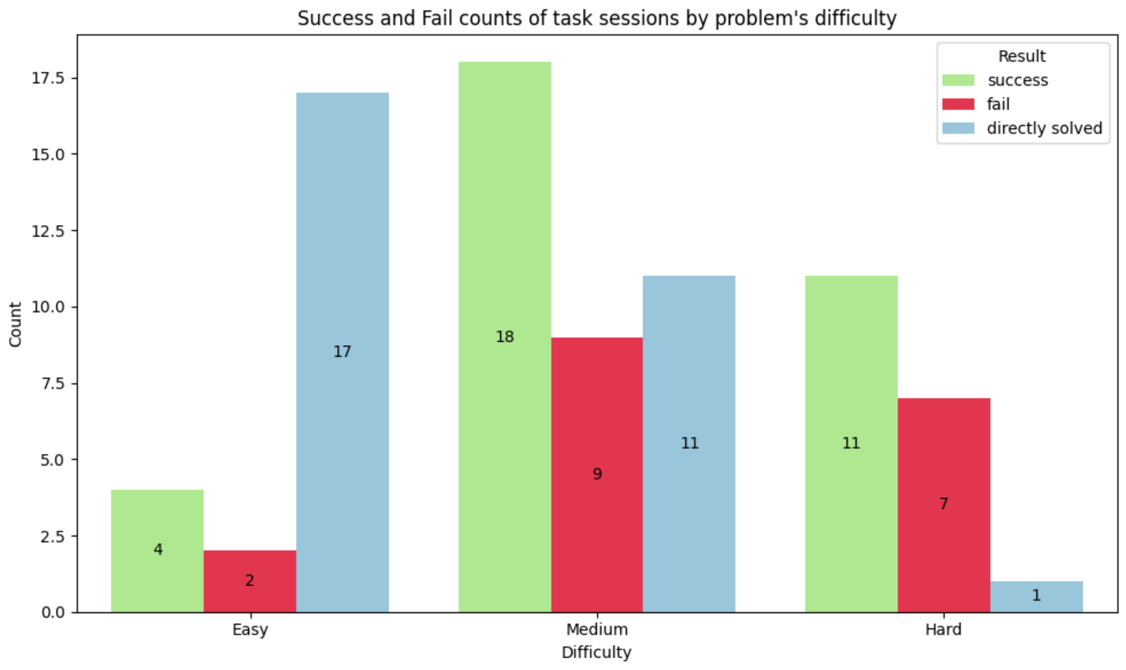

Fig. 2. Distribution of results of steering conversation

Table 2. Categories of self-debugging errors.

Code	Category	"The model's self-debugging procedure fails to ..."
[SD1]	Understanding	... understand the requirements for its generated code.
[SD2]	Identification	... identify an issue in its generated code.
[SD3]	Correction	... correct an identified issue in its generated code.
[SD4]	Repetition	... execute without repetitively debugging its generated code.
[SD5]	Evaluation	... evaluate correctness with appropriate unit tests.

Table 3. Categories of human feedback.

Theme	Category	"The programmer provided feedback that ..."	Socratic method
Evaluate	Test Reiteration (A1)	... reiterates an existing test case requirement.	Elenchus
	Test Definition (A2)	... provides a new test case requirement.	Dialectic
	Test Revision (A3)	... revises an existing test case requirement.	Elenchus
Implement	Programmatic Error (B1)	... resolves programmatic issues in a specific region of the generated code.	Elenchus
	Efficiency Error (B2)	... resolves efficiency issues in a specific region of the generated code.	Elenchus
Understand	Requirement Reiteration (C1)	... reiterates existing solution requirements.	Definition
	Requirement Clarification (C2)	... clarifies existing solution requirements.	Definition
Plan	Approach Orientation (D1)	... reframes the approach for solving the problem.	Maieutics
	Approach Instruction (D2)	... outlines a series of steps for solving the problem.	Maieutics
	Approach Reversion (D3)	... reverts the generated solution to a previous version.	Recollection

4.3 Preliminary lessons about effective steering

LLMs are recognized for their enhanced performance on complicated reasoning tasks when utilizing reasoning decomposition methods such as chain-of-thought [27] and tree-of-thought [28]. Yet, a significant hurdle for these automated decomposition techniques is their potential inability to accurately decompose a problem, often due to challenges in establishing a proper understanding or inductive bias. In our study, we observed that human-guided decomposition that converts the model into their familiar problem could effectively guide the model toward the correct plan, especially when it grappled with formulating an appropriate algorithmic strategy (#17, #26, #27), though the model may still encounter specific implementation issue (#8, #26, #27). For instance, in example #27, the model struggled to discern the dynamic programming aspect of the problem. Participant P18 advised the model to "*consider backtracking-style dynamic programming and apply the memorization technique for algorithm implementation*". This guidance led the model to produce a reasonably structured program. While there were some implementation errors, further hints regarding rectifying these issues enabled the model to derive the correct solution.

Identifying the programming stage that the model struggles with is essential for successful steering. Writing an accurate code needs programmers to go through multiple stages correctly, including understanding, planning, implementation, and testing. We found that our participants are more likely to make a successful steering if they give relevant feedback that helps the model conquer the hurdles faced at the struggling stage. For example, in #2, the model initially misunderstood a problem condition (P6: "*Problem means we can keep deleting the closest occurrence of c to the left of* i *if there exists one, while the model thinks it can only delete once*"). P8's initial feedback for cluing the model about the correct algorithm type did not help the model identify the misunderstanding and it carried the misunderstanding onto the new algorithmic plan. P8 then clarified the condition that the model misunderstood which effectively steered the model to come up with a program with minimal implementation issues. As an opposite example, P27 at example #22 kept cluing the model with the bug location, while not explicitly pointing out that the model overlooked the absolute sign in the condition "*the value of the partition is |max(nums1) - min(nums2)|*", which made the discourse finally failed.

4.4 User challenges

4.4.1 Challenges and Barriers to Effective Steering. Our qualitative analysis suggests that participants generated steering feedback based on present error, personal experience and problem nature. Participants described several key factors that impact their steering strategy choice during conversation. 18 participants mentioned that they used a "greedy" approach to determine what feedback to give mainly based on errors in model's last attempt and the effectiveness of prior steering attempts. For example, P8 said:

> "*My approach was entirely driven by how I see improvements with the error at hand. I'm not at able to predict what new errors might surface with my new suggestions, so this is the only way that I can imagine doing it*".

Similarly, P12 said he had to do a few back-and-forth trials to find out what model actually struggled with:

> "*My natural approach is one that centers around examples that say 'Here's what's right' or 'Here's what's wrong' in the last answer. I tried that for a few back-and-forth instances, and if I hit a roadblock, I assumed that Code Interpreter and I must be on two different pages. Ultimately, this was what drove me to provide a more thorough description of the problem after having no success in getting Code Interpreter to identify the issues*".

Only one participant (P19) clearly said that he "*planned how to steer at the beginning of conversation*". In addition, 7 participants mentioned that they relied on prior programming experience to choose steering strategy. For example, P24 said "*It's very similar to my everyday rubber duck debugging. I gave model feedback that I think would guide myself best when I faced the same error*". Besides, 3 participants drew parallels between the steering process and their experiences with programming tutoring in college. For example, P16 said "*It (steering) just like talking to a student in* *your programming class. Confirm the correct part that the model does, points out the failure part, asks it to fix the failure part with some suggestion on how to fix it*". Lastly, 8 participants thought problem nature affected how they steered the model. In particular, 4 participants emphasized they intentionally gave clear step-by-step instruction because solving those problems needs "*clever tricks*" (P20) or "*complicated implementation*" (P27), and they assumed GPT-4 cannot sort them out independently. On the other hand, 4 participants conveyed they primarily depended on the self-debug capability of GPT-4 since "*the solution looks straightforward*" (P2).

REFERENCES

[1] Erfan Al-Hossami, Razvan Bunescu, Ryan Teehan, Laurel Powell, Khyati Mahajan, and Mohsen Dorodchi. 2023. Socratic questioning of novice debuggers: A benchmark dataset and preliminary evaluations. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*. 709–726.

[2] Beng Heng Ang, Sujatha Das Gollapalli, and See Kiong Ng. 2023. Socratic Question Generation: A Novel Dataset, Models, and Evaluation. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. 147–165.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] John Beversluis. 1974. Socratic definition. *American Philosophical Quarterly* 11, 4 (1974), 331–336.

[5] Edward Y Chang. 2023. Prompting large language models with the socratic method. In *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 0351–0360.

[6] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

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

[9] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, et al. 2023. A static evaluation of code completion by large language models. *arXiv preprint arXiv:2306.03203* (2023).

[10] Paul Drew. 2004. Conversation Analysis. *Handbook of language and social interaction* (2004), 71–102.

[11] James Fieser and Bradley Dowden. 2011. Internet encyclopedia of philosophy. (2011).

[12] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large Language Models Cannot Self-Correct Reasoning Yet. arXiv:2310.01798 [cs.CL]

[13] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. Promptmaker: Prompt-based prototyping with large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–8.

[14] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. arXiv:2306.02907 [cs.CL]

[15] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[16] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]

[17] Mahin Naderifar, Hamideh Goli, and Fereshteh Ghaljaie. 2017. Snowball sampling: A purposeful method of sampling in qualitative research. *Strides in development of medical education* 14, 3 (2017).

[18] OpenAI. 2023. *ChatGPT Plugins: Code Interpreter.* https://openai.com/blog/chatgpt-plugins#code-interpreter

[19] OpenAI. 2023. *ChatGPT Release Notes.* https://help.openai.com/en/articles/6825453-chatgpt-release-notes Accessed on 2023-09-13.

[20] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://api.semanticscholar.org/CorpusID:257532815

[21] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[23] Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366* (2023).

[24] Kumar Shridhar, Jakub Macina, Mennatallah El-Assady, Tanmay Sinha, Manu Kapur, and Mrinmaya Sachan. 2022. Automatic Generation of Socratic Subquestions for Teaching Math Word Problems. *arXiv preprint arXiv:2211.12835* (2022).

[25] Clemencia Siro, Mohammad Aliannejadi, and Maarten de Rijke. 2023. Understanding and Predicting User Satisfaction with Conversational Recommender Systems. *ACM Transactions on Information Systems* (2023).

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[27] 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 (2022), 24824–24837.

[28] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).

[29] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 437, 21 pages. https://doi.org/10.1145/3544548.3581388

[30] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5673–5684. https://doi.org/10.1145/3580305.3599790

[31] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).