When Benchmarks Talk: Re-Evaluating Code LLMs with Interactive Feedback

Jane Pan[1]* Ryan Shar[2]* Jacob Pfau[1] Ameet Talwalkar [2] He He [1]† Valerie Chen [2]†
[1]New York University [2]Carnegie Mellon University
jane.pan@nyu.edu

Abstract

Programming is a fundamentally interactive process, yet coding assistants are often evaluated using static benchmarks that fail to measure how well models collaborate with users. We introduce an interactive evaluation pipeline to examine how LLMs incorporate different types of feedback in a collaborative setting. Specifically, we perturb static coding benchmarks so that the code model must interact with a simulated user to retrieve key information about the problem. We find that interaction significantly affects model performance, as the relative rankings of 10 models across 3 datasets often vary between static and interactive settings, despite models being fairly robust to feedback that contains errors. We also observe that even when different feedback types are equally effective with respect to performance, they can impact model behaviors such as (1) how models respond to higher- vs. lower-quality feedback and (2) whether models prioritize aesthetic vs. functional edits. Our work aims to "re-evaluate" model coding capabilities through an interactive lens toward bridging the gap between existing evaluations and real-world usage.

1 Introduction

Programming with a language model is a highly collaborative process, where developers interact with code models to provide updated information about initially underspecified requests or critique the output of the code model. Thus, giving and receiving feedback are critical elements of the process in which programmers use code models (Chidambaram et al., 2024). For example, chat interfaces like ChatGPT (OpenAI, 2022) or the chat panel of Github Copilot (Github, 2022) facilitate multi-turn conversations in which programmers can iteratively refine a piece of code by providing additional context and details to the LLM (Kalla et al., 2023; Xiao et al., 2023).

*Equal Contribution.
†Co-senior Authors.

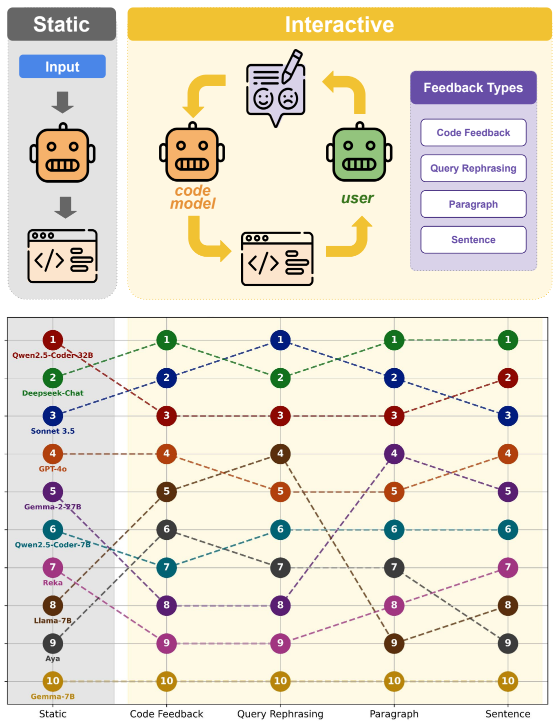

Figure 1: While most existing benchmarks statically evaluate LLM coding capabilities, code LLMs are used interactively in practice. We introduce an evaluation pipeline that evaluates code models in an interactive setting (top). Across three datasets, such as LiveCodeBench (bottom), we find that interactively evaluating models with different feedback types (CODE FEEDBACK, QUERY REPHRASING, PARAGRAPH, and SENTENCE) leads to different rankings when compared to static evaluation.

Despite the popularity of these tools, existing static benchmarks that measure task performance often rely on a simple input-output configuration, where the question is well defined and the model is asked to generate the whole completion in one shot (Chen et al., 2021; Austin et al., 2021; Jain et al., 2024; White et al., 2024). While this relatively simplistic setting is scalable and enables effi-

Findings of the Association for Computational Linguistics: ACL 2025, pages 24672–24700
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

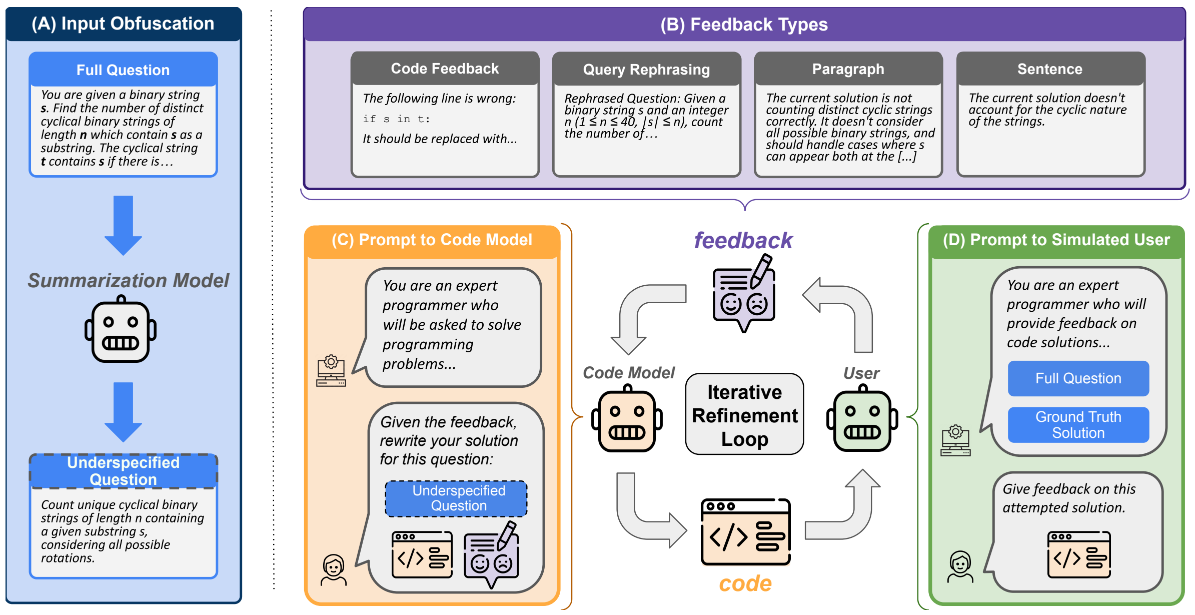
(A) Input Obfuscation
Full Question
You are given a binary string s. Find the number of distinct cyclical binary strings of length n which contain s as a substring. The cyclical string t contains s if there is…
Summarization Model
Underspecified Question
Count unique cyclical binary strings of length n containing a given substring s, considering all possible rotations.
(B) Feedback Types
Code Feedback
The following line is wrong:
if s in t:
It should be replaced with...
Query Rephrasing
Rephrased Question: Given a binary string s and an integer n (1 ≤ n ≤ 40, |s| ≤ n), count the number of…
Paragraph
The current solution is not counting distinct cyclic strings correctly. It doesn't consider all possible binary strings, and should handle cases where s can appear both at the [...]
Sentence
The current solution doesn't account for the cyclic nature of the strings.
(C) Prompt to Code Model
You are an expert programmer who will be asked to solve programming problems...
Given the feedback, rewrite your solution for this question:
Underspecified Question
feedback
Code Model
Iterative Refinement Loop
User
code
(D) Prompt to Simulated User
You are an expert programmer who will provide feedback on code solutions...
Full Question
Ground Truth Solution
Give feedback on this attempted solution.

Figure 2: Overview of our interactive pipeline for coding evaluation. (A) We obfuscate the input of existing fully specified datasets to reflect how programmers tend to underspecify requests to LLMs (e.g., via docstrings or comments) in practice. (B) As developers may interact with models in a variety of ways, we explore 4 different feedback types and introduce a pipeline that mimics the iterative refinement loop that programmers often use with chat models, (C) where the code model generates a solution using feedback on its previous solution, (D) and the user provides updated feedback to the code model.

cient evaluation, it does not capture how developers realistically use models to write code (Mozannar et al., 2024). While recent benchmarks have begun to explore how interactive settings can lead to performance gains in coding applications (Wang et al., 2023b), they assume a single form of natural language feedback. In practice, developers provide many forms of feedback when implementing code (Chidambaram et al., 2024), which can range from binary feedback on the correctness of the code to suggesting direct changes to the code.

Building evaluations of programming assistants that more closely mimic this setting enables a better understanding of model behavior and potential pitfalls in the interactive setting, such as model capability with respect to processing feedback, the effects of different feedback types, or model robustness. We bridge the gap between existing evaluations and real-world use cases by benchmarking how different feedback types impact model behavior in a simulated interactive setting (Figure 1, top). We propose an evaluation method that transforms a static coding benchmark into an interactive, collaborative one (Figure 2). The pipeline components include input obfuscation to create underspecified problems to induce collaboration, a simulated user for scalability, and multiple types of feedback (CODE FEEDBACK, QUERY REPHRASING, PARAGRAPH, and SENTENCE).

Across 10 coding models (6 open and 4 closed) and 3 coding benchmarks, we find that relative performance between models often changes between static and interactive settings (Figure 1, bottom), suggesting that models perform differently in the static vs. interactive settings. Beyond performance-based metrics, we analyze important components of model-user interactions, including feedback quality and code model steerability. We use these insights to investigate the effects of different feedback types. For example, we find that PARAGRAPH and CODE FEEDBACK tend to lead to the highest performance boost compared to other feedback types (e.g., SENTENCE or QUERY REPHRASING). However, when considering the effect of feedback quality on performance, we find that unlike PARAGRAPH, CODE FEEDBACK's higher-quality feedback makes output worse more frequently than its lower-quality counterpart for stronger models. Furthermore, PARAGRAPH leads to more surface-level edits than CODE FEEDBACK, whereas users may prefer variation in model behavior to be robust to changes in feedback type.

Our work provides a new approach to investigating the downstream ramifications of different interactive programming settings and their effects on model behavior. We open-source our evaluation pipeline[1], which makes it easy to add static benchmarks and turn them into interactive ones, to facilitate the evaluation of more models and datasets.

2 Methodology

Figure 2 provides an overview of our pipeline for interactive evalution, which is driven by an iterative refinement loop in which the user model and code model interact. At each step of the loop, the code model is given the obfuscated programming question in natural language, the code from the previous attempt, and the user model's feedback on the previous attempt. Each iterative refinement loop lasts 5 steps for each question, terminating early if a correct solution is reached in less than 5 steps.

2.1 Dataset Transformation

Our pipeline is designed to transform static coding questions into interactive ones. We describe our selection criteria for the datasets used in this work, as well as the input obfuscation protocol that enables collaboration between the user and code model.

Input obfuscation. To ensure that the user and code model collaborate, we obfuscate the input given to the code model to remove critical information from the input (Figure 2A). This induces an information asymmetry, akin to that which might exist in practice, that forces the code model to rely on feedback from the user to recover key details about the full problem specification. We underspecify the input to the code model by using `Sonnet-3.5` to summarize the original questions, which often contain significant detail about desired behaviors and potential edge cases, into one-sentence summaries. Appendix A.4 includes additional details on the design of our input obfuscation method, and Appendix A.9 compares the effects of input obfuscation on the SELF-CRITIQUE BASELINE setting.

To illustrate input obfuscation, consider a question from APPS (Interview) which asks how long it will take for two flight attendants to serve lunch to a customer in a given seat. The original question includes details about the flight attendant's serving speed, the order they traverse the rows of the plane, and the serving order of seats in the row. The summarized form of this question might be *"Calculate the time it takes for a passenger in a specific seat to receive their lunch on an airplane with an infinite number of rows, given the serving pattern of two flight attendants moving from front to back,"* omitting some of the key details required to fully solve the question (e.g., row and seat order). We use this as a running example in the following section.

Datasets. We select challenging datasets with lengthy problem descriptions and available ground-truth solutions. We use the Interview and Introductory levels of APPS (Hendrycks et al., 2021), ClassEval (Du et al., 2023), and the Easy, Medium, and Hard levels of LiveCodeBench (Jain et al., 2024). We randomly sample 200 examples from APPS Interview, 200 examples from APPS Introductory, and 75 examples from ClassEval. We use 70 examples from LiveCodeBench (across all three difficulty levels). More details on the datasets can be found in Appendix A.1.

2.2 Feedback Types

Existing work on developer-code model interactions shows that programmer feedback is diverse (Chidambaram et al., 2024). Following their results, we explore multiple variants of feedback types outside of generic natural language feedback and investigate four categories[2] of feedback: PARAGRAPH, SENTENCE, QUERY REPHRASING, and CODE FEEDBACK (Figure 2B). For each interaction, we fix the feedback type so that the user model always responds with the same kind of feedback[3].

The SENTENCE and PARAGRAPH feedback types are ones where the user model provides feedback using natural language. These feedback styles mimic the inputs used in chat-based interfaces, where users respond to the model via a chat window. In our setting, the user is prompted to only use a sentence or paragraph for their response. An example of SENTENCE feedback for the airplane question might be *"The current solution doesn't follow the seat serving order f-e-d-a-b-c,"* while PARAGRAPH feedback tends to have specific critiques of the algorithm with sentences like *"It doesn't account for the two flight attendants serving simultaneously. It should first calculate the number of complete 4-row blocks served, then handle the remainder."*

[1]Our code is publicly available at `https://github.com/janepan9917/WhenBenchmarksTalk/`.

[2]Appendix A.3 provides additional details on how we selected these four categories.

[3]For additional experiments where interactions use multiple kinds of feedback, see Appendix A.10.

QUERY REPHRASING and CODE FEEDBACK feedback aim to replicate common feedback styles from developers (Chidambaram et al., 2024). In QUERY REPHRASING, the user is prompted to rewrite a similar-length version of the underspecified question with additional details required for the code model to find a solution. We constrain the length to mimic how real users rephrase their inputs when providing feedback and to prevent the user from simply copying in the full question. An example of QUERY REPHRASING in the airplane question might be *"Question: Calculate the time for a passenger to receive lunch on a plane where two flight attendants serve food. Attendants start at rows 1 and 3, move forward by 2 rows after serving. They serve right side (f to d) then left side (c to a) of each row. Output the waiting time in seconds."* CODE FEEDBACK prompts the user to directly indicate which lines of code are incorrect and suggest alternate code snippets. An example of CODE FEEDBACK in the airplane question might be *"The function get_time_to_lunch(seat, num_attendants) should be get_time_to_lunch(seat) as the number of attendants is always 2."* Appendix A.2 provides additional examples of each type of feedback.

2.3 Code Models

We select a total of 10 code models, spanning both open-source and closed models and a wide range of capabilities and parameter sizes. We selected the following open-source models for their ability to follow user instructions, their range of parameter sizes, and overall coding capability: `Deepseek-V3` (DeepSeek-AI et al., 2024), `Gemma-7B-it` (Team et al., 2024a), `Gemma-2-27B-it` (Team et al., 2024b), `Llama-3.1-8B-Instruct` (Grattafiori et al., 2024), `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024), and `Qwen2.5-Coder-32B-Instruct` (Hui et al., 2024). We select the following closed models for their commercial adoption and performance on existing static benchmarks: `Aya` (Üstün et al., 2024),, `GPT-4o` (OpenAI et al., 2024), `Reka` (Team et al., 2024c), and `Sonnet-3.5` (Anthropic, 2023). The parameters used to query each model are provided in Appendix A.5.

Code model prompts. We prompt the code models with their previous solution and the user feedback on that solution. We do not provide a history of all code model interactions with the user, only the most recent code attempt and user feedback on the most recent attempt. Specific instructions are given for each dataset for varying input and output formats. All prompts are in Appendix A.6.

2.4 User Models

Following prior work (Dubois et al., 2023; Zheng et al., 2023; Mozannar et al., 2023), we use LLMs to scalably simulate feedback given by users when interacting with code models. To help close the capability gap between real-world expert users and LLMs, we give the user model access to the original fully-specified question, as well as a ground-truth solution. This allows the simulated user to produce higher-quality feedback more often. The user model prompt also includes instructions to avoid leaking the exact solution in its responses to the code model. We only constrain the formatting style of the feedback, allowing the user to choose the content of criticism (e.g. input/output formatting, algorithmic correctness, code style). We choose `Sonnet-3.5` due to its high performance on static coding benchmarks and strong reasoning capabilities, but verify that other user models (e.g., `GPT-4o-mini`) are also able to improve performance over SELF-CRITIQUE BASELINE (see Section 3 for details). These results, along with prompts for the user can be found in Appendix A.6.

3 Static vs. Interactive Performance

We compare the performance of models in static and interactive evaluation settings. To measure the effectiveness of different feedback types, we compare against the STATIC and the SELF-CRITIQUE BASELINE settings. The STATIC setting of each dataset evaluates the code model on the original, fully specified questions. In this setting, the code model is not given any feedback and the first attempt is used as the final output. To match the test-time compute of the interactive settings, the SELF-CRITIQUE BASELINE setting uses five iterations of self-critique to generate feedback using the underspecified question and the output from the previous step (Madaan et al., 2023). For this setting, no additional information of the original question or solution is given and no user is involved.

3.1 Performance Metrics

We use **test case accuracy (TCA)** to evaluate model performance. In ClassEval, we combine the set of function tests and class tests to measure

Dataset	STATIC	SELF-CRITIQUE BASELINE	PARAGRAPH	SENTENCE	CODE FEEDBACK	QUERY REPHRASING
APPS	0.335 (0.003)	0.034 (0.001)	0.381 (0.004)	0.271 (0.003)	0.428 (0.004)	0.289 (0.003)
LCB	0.699 (0.008)	0.274 (0.008)	0.655 (0.009)	0.611 (0.009)	0.631 (0.009)	0.183 (0.008)
ClassEval	0.714 (0.002)	0.483 (0.006)	0.679 (0.006)	0.642 (0.006)	0.759 (0.006)	0.693 (0.005)

Table 1: Test case accuracy and standard error of each setting in APPS, LiveCodeBench (LCB), and ClassEval, averaged across all code models. We find that feedback can recover performance comparable to or even exceeding the STATIC setting.

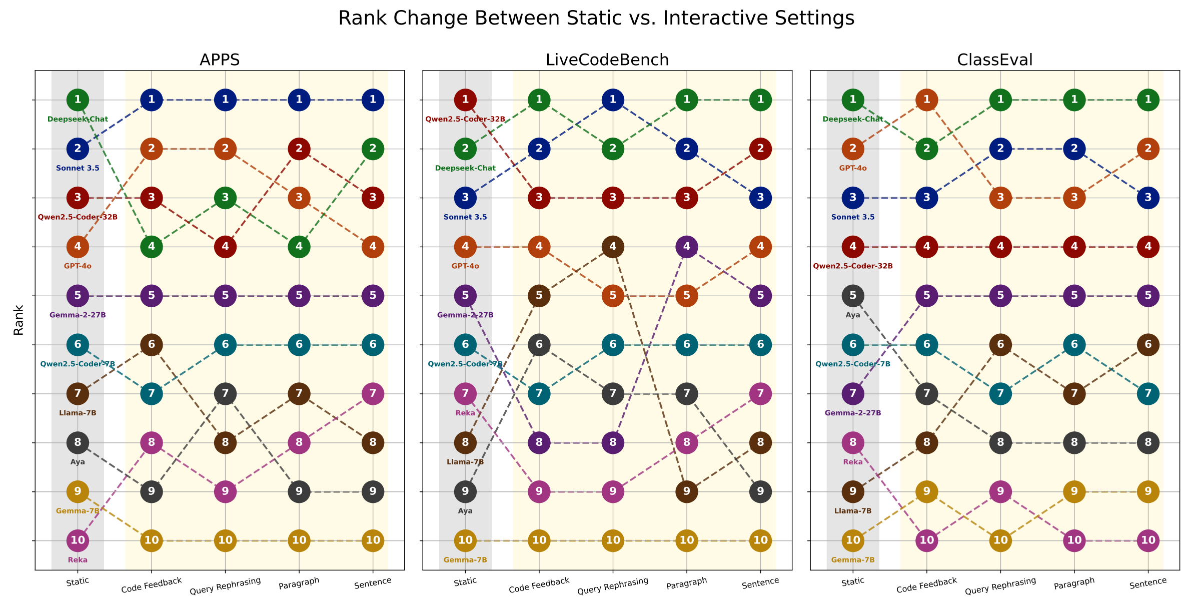

Figure 3: Rank changes between static and interactive settings across 3 datasets— APPS, LiveCodeBench, and ClassEval. We stratify interactive settings by feedback type (CODE FEEDBACK, QUERY REPHRASING, PARAGRAPH, and SENTENCE), and observe changes in rankings across all datasets and interactive settings.

TCA. To measure the distances between two rankings σ_A and σ_B of length n, we use a normalized variant of **Spearman's Footrule** ($\tilde{F} : \to [0, 1]$):

$$\tilde{F}(\sigma_A, \sigma_B) = \frac{\sum_{i=1}^{n} |\sigma_A(i) - \sigma_B(i)|}{\max_{\sigma,\sigma'} \sum_{i=1}^{n} |\sigma(i) - \sigma'(i)|}$$

where σ, σ' are any ranking of length n. For a perfectly correlated pair of rankings, $\tilde{F} = 0$; for uncorrelated rankings, $\tilde{F} = 0.73$; for perfectly anti-correlated rankings, $\tilde{F} = 1$. Appendix A.7 contains details on how we derive these metrics and thresholds for correlation.

3.2 Results

Feedback can recover performance comparable to or even exceeding the STATIC setting. Table 1 shows the performances averaged across all models for each dataset and feedback type. Comparing STATIC to SELF-CRITIQUE BASELINE shows that our input perturbation often obfuscates the problem, as SELF-CRITIQUE BASELINE usually underperforms the STATIC setting, and feedback is often required to achieve performance comparable with the STATIC setting (as with LiveCodeBench). Moreover, interacting with feedback may also allow code models to surpass STATIC performance (as with APPS and ClassEval). This may be because the user may supply not only additional specifications about the problem, but also guidance with respect to general problem-solving or programming capabilities.

CODE FEEDBACK and PARAGRAPH improve performance the most. CODE FEEDBACK and/or PARAGRAPH are consistently the most effective at improving model performance in the underspecified setting (Table 1). Compared to others, these feedback types tend to be longer and thus may encapsulate more helpful information to the code model. The weaker performing feedback types are SENTENCE and QUERY REPHRASING, the latter of which struggles the most on LiveCodeBench. This may stem from the fact that LiveCodeBench is not in the training sets of most models (due to its problem cut-off date); for popular datasets, the code

Dataset	PARAGRAPH	SENTENCE	CODE FEEDBACK
APPS	0.91 (0.01)	0.89 (0.01)	0.94 (0.01)
LCB	0.92 (0.01)	0.88 (0.02)	0.79 (0.03)
ClassEval	0.96 (0.01)	0.91 (0.01)	0.77 (0.02)

Table 2: Average rate of directional correctness with standard error for each dataset and setting.

model may sometimes recognize the full question using the user's QUERY REPHRASING, leading to improved performance.

Models perform differently in static vs. interactive settings. Figure 3 plots the relative rankings (measured by TCA) across static and interactive settings. While we generally observe permutations in rankings when comparing STATIC and interactive, LiveCodeBench demonstrates the most variance, with some models changing 4 ranks between STATIC and interactive settings.

To understand how rank changes vary between static vs. interactive settings, we calculate the normalized Spearman's Footrule distances between the STATIC and interactive settings (Table 16). All three datasets demonstrate relatively weak positive correlation in many interactive settings; for instance, for CODE FEEDBACK, $\tilde{F}$ ranges from 0.222 to 0.346 across the three datasets. Generally, top models tend to be consistently high across feedback types, whereas weaker models tend to demonstrate more variance in rank.

4 Feedback Quality

To understand whether the ranking changes in Section 3 are due to variations in feedback quality across models, we develop a proxy for feedback quality and examine its effect on how code models interact with feedback.

4.1 Quality Metrics

Previous works infer feedback quality by the feedback's effect on performance (Zhang and Choi, 2023). Instead of relying only on performance, we classify feedback by **directional correctness**, a binary value of whether it accurately claims that the code solution was correct or incorrect. For instance, if the solution is incorrect (i.e. has TCA < 1), but the feedback claims that the solution is correct, then we consider the feedback directionally incorrect. However, if the solution is correct (i.e. has TCA $= 1$) and the feedback claims that it is correct, we consider it directionally correct.

We automate the classification via `GPT-4o` on SENTENCE, PARAGRAPH, and CODE FEEDBACK.[4] Appendix A.8 discusses other feedback quality metrics we considered, as well as additional information on the classification protocol.

4.2 Results

Directional correctness is consistently high across models and feedback types. Table 2 compares average directional correctness by feedback type, which does not vary greatly across models or feedback types and often reaches above 0.8. This suggests that the feedback is high enough quality to compare across models and feedback types. Although PARAGRAPH and CODE FEEDBACK feedback induce the highest performances, PARAGRAPH feedback tends to have the highest directional correctness, whereas CODE FEEDBACK tends to have the lowest.

Code models are generally robust to directionally incorrect feedback. Figure 4 shows the distribution of solutions whose performances improve versus decrease when comparing directionally correct feedback to directionally incorrect feedback. Although directionally correct feedback has a higher rate of solutions whose performances improve, the rate of directionally *incorrect* feedback that results in improved performance is still substantial. For instance, `Sonnet-3.5` has equal rates of improved performance after either directionally correct or directionally incorrect PARAGRAPH and SENTENCE feedback.

For stronger models, directionally correct CODE FEEDBACK tend to worsen post-feedback solutions than directionally incorrect CODE FEEDBACK. While all directionally correct feedback have roughly similar effects on the rate of improved post-feedback solutions, PARAGRAPH and SENTENCE also decrease the proportion of worse post-feedback solutions (Figure 4, center and right). CODE FEEDBACK is the only feedback type where stronger models (e.g., `Sonnet-3.5`, `GPT-4o`, `Qwen2.5-Coder-32B-Instruct`) are more likely to generate a worse solution when given directionally correct feedback rather than directionally incorrect (Figure 4, left).

[4]QUERY REPHRASING does not provide direct feedback on the solution, so it is not eligible for our quality metric.

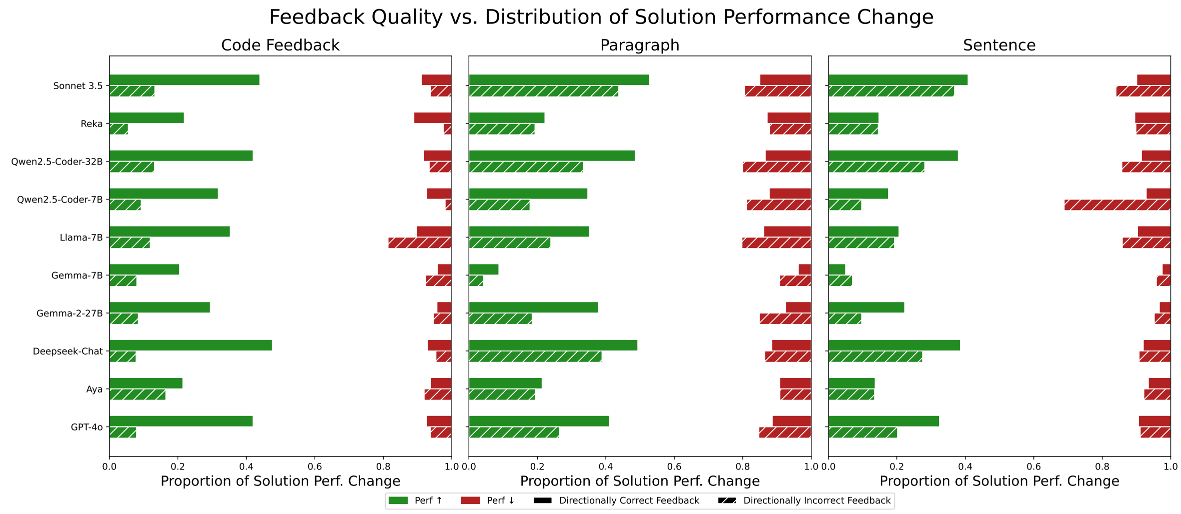

Figure 4: Distribution of performance change across feedback types and directional correctness. We split solutions into post-feedback performance gains (green) or losses (red) and observe that models can still benefit from directionally incorrect feedback, and that directionally correct CODE FEEDBACK sometimes increases the rate of post-feedback performance loss.

5 Model Steerability

We extend our analysis beyond performance to investigate **steerability**, or how much a code model adjusts its previous solution in response to feedback. We show that drops in performance in interactive settings are likely due to ineffectively incorporating feedback, rather than outright ignoring feedback in the next iteration of the solution.

5.1 Steerability Metrics

We evaluate model steerability on APPS and LiveCodeBench[5] and consider two metrics of change between solution iterations. Firstly, we use **Levenshtein edit distance** to evaluate surface-level changes between consecutive versions of code. Secondly, we count the number of **changes in test-case behavior** — with respect to whether incorrect test cases flip to correct or vice-versa — to evaluate behavioral-level adjustments to the code. We refer to the former as *surface-level steerability* and the latter as *behavioral steerability*.

5.2 Results

PARAGRAPH feedback is associated with higher behavioral-level and surface-level steerability across all models. Figure 5 plots each feedback type by the behavioral (top) or surface-level (bottom) steerability it induces. PARAGRAPH and CODE FEEDBACK score the highest in behavioral steerability (changing 21.8% and 19.3% of test cases on average), while PARAGRAPH score the highest in surface-level steerability (with an average edit distance of 445.6 characters). Notably, PARAGRAPH is also one of the highest-performing feedback styles, suggesting that it induces effective changes in the code solution on both the behavioral and surface levels.

Weaker models tend to make surface-level rather than effective behavioral-level changes. Figure 6 shows how code models change their previous solutions on all datasets across all feedback types. Weaker models (e.g. `Gemma-7B-it` and `Llama-3.1-8B-Instruct`) tend to make many surface-level changes that do not greatly change the behavior of the code. However, stronger models (e.g. `GPT-4o`, `Sonnet-3.5`, and `Qwen2.5-Coder-32B-Instruct`) may make relatively small edits that highly affect code behavior.

6 Related Work

Code benchmarks. Static benchmarks, e.g., HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), largely focusing on interview-style programming problems, have been the most commonly used to evaluate coding capabilities (Lu et al., 2021; Nijkamp et al., 2023; Zhu et al., 2022; Wang et al., 2023a; Liu et al., 2023; Jimenez et al., 2023; Khan et al., 2023; Yan et al., 2023; Cassano et al., 2023; Muennighoff et al., 2023; Dinh et al., 2023; Yang et al., 2023; Du et al., 2023).

[5]We do not evaluate on ClassEval as their evaluation utilities do not report exactly which test cases pass or fail.

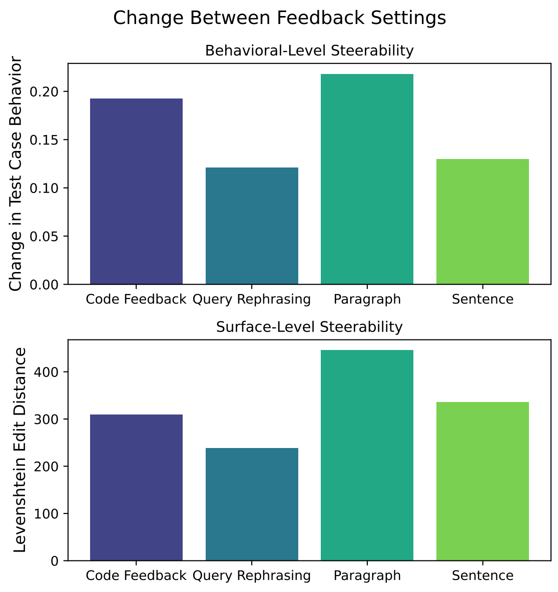

Figure 5: Behavioral-level (top) and surface-level (bottom) steerability by feedback type, averaged across all models for APPS and LiveCodeBench. PARAGRAPH feedback induces the most changes at both levels, while CODE FEEDBACK leads to more behavioral changes with less aesthetic changes.

Recent live benchmarks aim to reduce contamination risks (Jain et al., 2024; White et al., 2024). Our evaluation pipeline can convert many of these static benchmarks into an interactive one, evaluating model abilities to incorporate different types of feedback; we demonstrate this with 3 datasets.

Interactive evaluation. As programmers are increasingly writing code collaboratively with AI chat assistants like ChatGPT (OpenAI, 2022) or Claude (Anthropic, 2023), many user studies have evaluated how programmers use chat assistants to write code (Ross et al., 2023; Chopra et al., 2023; Kazemitabaar et al., 2023; Xiao et al., 2023; Nam et al., 2024; Mozannar et al., 2024; Chidambaram et al., 2024), typically employing only a few models in the study (≤ 3). While new platforms evaluate model coding capabilities at scale by collecting human preferences (Chiang et al., 2024; Chi et al., 2024), it remains challenging to understand the fine-grained effects of feedback. We create a benchmark with *simulated* users to enable scalable evaluation of the nuances of feedback in interactive coding settings, while drawing from insights of existing *human* studies (e.g., common types of feedback (Chidambaram et al., 2024) and tendencies to underspecify inputs (Xiao et al., 2023)).

Prior work has explored interactive benchmarks with simulated users for various applications, such as tool use (Yao et al., 2024), creative tasks (Jia et al., 2024), coding (Wang et al., 2023c; Shao et al., 2024), and other collaborative contexts (Wu et al., 2023). Our benchmark extends them by introducing diverse forms of user feedback and investigating their effect on feedback quality and model steerability. We also enforce collaboration between the simulated user and code model via input obfuscation, aligning with real-world use cases where the user's input to the model may be underspecified.

7 Conclusion

We propose a new approach to evaluating code models by introducing an interactive pipeline where the code model must collaborate with a simulated user to solve underspecified coding problems with different feedback types. We find that the relative performances of models change radically between static and interactive settings. We analyze key elements of model-user interactions, such as feedback quality and model steerability, to provide insights into the downstream effects of feedback type on model behavior and feedback effectiveness. Our work bridges the gap between existing static benchmarks and real-world usage, and we hope to inspire future work on scalable methods for evaluating models in a collaborative setting.

8 Limitations

In this work, the focus of our experiments is on three diverse code benchmarks to demonstrate the generality of our pipeline. However, given the expansive set of static benchmarks, our results may not encompass the full set of observations one might obtain from considering more varied datasets (e.g., non-Python coding questions). On the evaluation front, since we do not explicitly compare LLM responses to human-generated feedback on the extensive set of modified questions, we focus on trends of model behavior as a response to different feedback types, rather than specific degrees of change. We also focus the majority of our evaluation on performance, but our benchmark is easily extendable to other measures of code quality; Appendix A.11 provides sample experiments using code readability instead of performance as the evaluation metric.

While we performed qualitative analysis on real human interactions with LLMs (Appendix A.3), the feedback types studied in this work are not

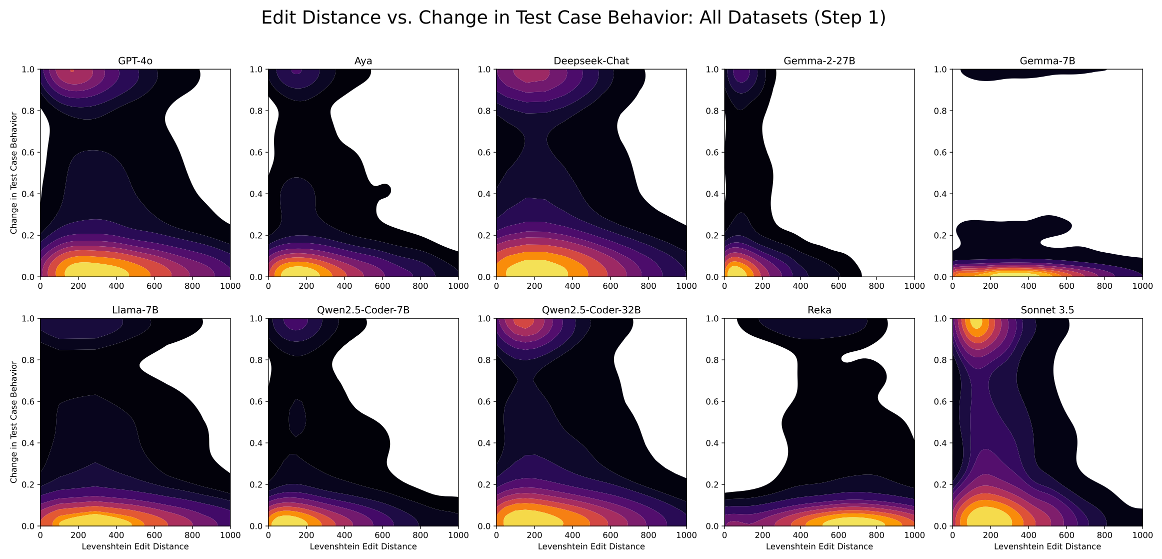

Figure 6: Distribution of surface-level steerability (x-axis) vs. behavioral steerability (y-axis) for all models during the first step of iterative refinement, averaged across all datasets. While some models make only surface-level changes that do not induce much behavioral change in code (e.g. `Gemma-7B-it`), others are also able to make highly effective edits that induce large changes in the behavior of the solution (e.g., `Sonnet-3.5`, `Qwen2.5-Coder-32B-Instruct`, `GPT-4o`).

fully comprehensive, and we do not claim that the user model's feedback is necessarily representative of actual human users. Rather, we use a simulated user to scalably examine how LLMs react to feedback in a collaborative setting, not to realistically imitate how expert humans use LLMs to program. For instance, users may mix feedback types within a single interaction, whereas we fix the feedback type across rounds of iterative refinement to isolate the effects of an individual feedback type. Appendix A.10 provides additional experiments where the user provides multiple types of feedback in the same interaction. Recent works also study whether LLMs can proactively seek user feedback via clarification questions or other interactions (Zhang and Choi, 2023; Zhang et al., 2024), whereas we only consider settings where the user initiates the feedback-giving process. This work was not intended to exhaustively test feedback types but to highlight a new approach to understanding the downstream effects of interactive programming settings.

8.1 Potential Risks

Our pipeline is intended to be used as a testbed to examine model behavior in response to feedback, rather than to realistically mimic actual human usage with models. As such, it should not be used as an approximation of actual human users. Other risks include overexposure to certain programming languages and natural languages, as we only include Python programming questions and English feedback.

9 Acknowledgements

We thank the members of the NYU ML^2 group for their valuable advice, thoughts, and discussions. We also thank Vishakh Padmakumar, Austin Wang, Jens Tuyls, and Naman Jain for their feedback and comments. This project has benefited from financial support from Eric and Wendy Schmidt (made by recommendation of the Schmidt Futures program) and Open Philanthropy, and from in-kind support by the NYU High-Performance Computing Center. This material is also partially supported by the National Science Foundation under Award IIS-2340345. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Anthropic. 2023. Meet claude.

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*, abs/2108.07732.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

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*, abs/2107.03374.

Wayne Chi, Valerie Chen, Anastasios N. Angelopoulos, Wei-Lin Chiang, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. 2024. Copilot arena: A platform for code llm evaluation in the wild.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot arena: An open platform for evaluating llms by human preference. *ArXiv preprint*, abs/2403.04132.

Subramanian Chidambaram, Li Erran Li, Min Bai, Xiaopeng Li, Kaixiang Lin, Xiong Zhou, and Alex C Williams. 2024. Socratic human feedback (sohf): Expert steering strategies for llm code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15491–15502.

Bhavya Chopra, Ananya Singha, Anna Fariha, Sumit Gulwani, Chris Parnin, Ashish Tiwari, and Austin Z Henley. 2023. Conversational challenges in ai-powered data science: Obstacles, needs, and design opportunities. *ArXiv preprint*, abs/2310.16164.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, and Chengda Lu et al. 2024. Deepseek-v3 technical report. *ArXiv preprint*, abs/2412.19437.

Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *ArXiv preprint*, abs/2308.01861.

Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Alpacafarm: A simulation framework for methods that learn from human feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Github. 2022. Github copilot - your ai pair programmer.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and Ahmad Al-Dahle et al. 2024. The llama 3 herd of models. *ArXiv preprint*, abs/2407.21783.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *ArXiv preprint*, abs/2105.09938.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *ArXiv preprint*, abs/2409.12186.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *ArXiv preprint*, abs/2403.07974.

Qi Jia, Xiang Yue, Tianyu Zheng, Jie Huang, and Bill Yuchen Lin. 2024. Simulbench: Evaluating language models with creative simulation tasks. *ArXiv preprint*, abs/2409.07641.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Dinesh Kalla, Nathan Smith, Fnu Samaah, and Sivaraju Kuraku. 2023. Study and analysis of chat gpt and its impact on different fields of study. *International journal of innovative science and research technology*, 8(3).

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*, pages 455:1–455:23. ACM.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *ArXiv preprint*, abs/2303.03004.

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. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The realhumaneval: Evaluating large language models' abilities to support programmers. *ArXiv preprint*, abs/2404.02806.

Hussein Mozannar, Valerie Chen, Dennis Wei, Prasanna Sattigeri, Manish Nagireddy, Subhro Das, Ameet Talwalkar, and David Sontag. 2023. Simulating iterative human-ai interaction in programming with llms. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*.

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.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, and Aidan Clark et al. 2024. Gpt-4o system card. *ArXiv preprint*, abs/2410.21276.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue.

Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, pages 491–514.

Yijia Shao, Vinay Samuel, Yucheng Jiang, John Yang, and Diyi Yang. 2024. Collaborative gym: A framework for enabling and evaluating human-agent collaboration. *ArXiv preprint*, abs/2412.15701.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, and Shreya Pathak et al. 2024a. Gemma: Open models based on gemini research and technology. *ArXiv preprint*, abs/2403.08295.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, and Bobak Shahriari et al. 2024b. Gemma 2: Improving open language models at a practical size. *ArXiv preprint*, abs/2408.00118.

Reka Team, Aitor Ormazabal, Che Zheng, Cyprien de Masson d'Autume, Dani Yogatama, Deyu Fu, Donovan Ong, Eric Chen, Eugenie Lamprecht, Hai Pham, Isaac Ong, Kaloyan Aleksiev, Lei Li, Matthew Henderson, Max Bain, Mikel Artetxe, Nishant Relan, Piotr Padlewski, Qi Liu, Ren Chen, Samuel Phua, Yazheng Yang, Yi Tay, Yuqi Wang, Zhongkai Zhu, and Zhihui Xie. 2024c. Reka core, flash, and edge: A series of powerful multimodal language models. *ArXiv preprint*, abs/2404.12387.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023a. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023b. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *ArXiv preprint*, abs/2309.10691.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023c. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *ArXiv preprint*, abs/2309.10691.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *ArXiv preprint*, abs/2406.19314.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *ArXiv preprint*, abs/2308.08155.

Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2023. Devgpt: Studying developer-chatgpt conversations. *ArXiv preprint*, abs/2309.03914.

Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. Devgpt: Studying developer-chatgpt conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 227–230. ACM.

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*, abs/2311.08588.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *ArXiv preprint*, abs/2406.12045.

Michael J. Q. Zhang and Eunsol Choi. 2023. Clarify when necessary: Resolving ambiguity through interaction with lms. *ArXiv preprint*, abs/2311.09469.

Michael J. Q. Zhang, W. Bradley Knox, and Eunsol Choi. 2024. Modeling future conversation turns to teach llms to ask clarifying questions. *ArXiv preprint*, abs/2410.13788.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *ArXiv preprint*, abs/2206.08474.

Ahmet Üstün, Viraat Aryabumi, Zheng-Xin Yong, Wei-Yin Ko, Daniel D'souza, Gbemileke Onilude, Neel Bhandari, Shivalika Singh, Hui-Lee Ooi, Amr Kayid, Freddie Vargus, Phil Blunsom, Shayne Longpre, Niklas Muennighoff, Marzieh Fadaee, Julia Kreutzer, and Sara Hooker. 2024. Aya model: An instruction finetuned open-access multilingual language model. *ArXiv preprint*, abs/2402.07827.

A Appendix

A.1 Additional Details on Datasets

Our pipeline is designed to accommodate generic static benchmarks with some modifications. For instance, because ClassEval requires the model to fill in the skeleton code of a class (rather than providing explicit programming questions), we underspecify its problems by summarizing the docstrings for each method. Likewise, LiveCodeBench does not provide ground-truth solutions, so we generate solutions for LiveCodeBench by sampling twice from `Sonnet-3.5`; if a correct solution is generated, we use it as the ground-truth solution for the question. (If not, we do not use the question as our pipeline requires the presence of a ground-truth solution in the user prompt.)

For evaluation of APPS and LiveCodeBench, Table 1, Table 2, and Figure 3 average across difficulty levels for brevity.

A.2 Example Feedback

We provide sample feedback from `Sonnet-3.5` in response to a proposed solution. All feedback types are in response to the same question (Table 3).

A.3 Additional Details on the Design of Feedback Types

We designed our four feedback types to reflect these different forms of real-world feedback. To further verify our choices of feedback types, we manually inspected 100 conversations from DevGPT (Xiao et al., 2024), an open-source dataset of ChatGPT interactions with developers. After filtering for multi-turn interactions where the user first asks the model to output code and then provides feedback on a previous generation, we found that almost all of the user feedback could be classified as PARAGRAPH, SENTENCE, CODE FEEDBACK, or QUERY REPHRASING. Our manual inspection of the feedback shows that our chosen feedback types are indeed representative of many real-world interactions.

A.4 Designing the Input Obfuscation Protocol

When designing the input obfuscation protocol, we tried to capture general features of underspecification by looking at real-world examples. Specifically, we analyzed how users document code in the wild by manually inspecting randomly sampled Github repos written in Python, primarily focusing on how developers write docstrings and inline comments. We found that users tended to write short docstrings (65 characters on average) and comments (55 characters on average). We also found that users tended to write self-contained docstrings that do not reference external classes or variables. To align our input obfuscation with these insights, we manually created short and self-contained question summaries for our 11-shot prompt (Figure **??**). The handcrafted summaries were 85 characters on average to reflect the terse user docstrings. In contrast, static benchmarks have significantly longer input specification lengths (e.g., 1561 characters on average for APPS and 1048 characters on average for LiveCodeBench).

A.5 Additional Details on Models

We obtained the weights for google/gemma-7b-it (`Gemma-7B-it`) from Hugging Face at `https://huggingface.co/google/gemma-7b-it`, meta-llama/Meta-Llama-3.1-8B-Instruct (`Llama-3.1-8B-Instruct`) from huggingface at `https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct`, and Qwen/Qwen2.5-Coder-7b-Instruct (`Qwen2.5-Coder-7B-Instruct`) from Hugging Face at `https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct`. We run each of the models on a single L40S GPU. We use a temperature setting of 0.9, 4096 max tokens, and the "do_sample" setting enabled. We use ⟨end_of_turn⟩, ⟨|eot_id|⟩, ⟨|im_end|⟩ as the EOS token for `Gemma-7B-it`, `Llama-3.1-8B-Instruct`, and `Qwen2.5-Coder-7B-Instruct` respectively.

We use Together AI (`https://api.together.xyz/`) to run the large open-weight models google/gemma-2-27b-it (`Gemma-2-27B-it`) and Qwen/Qwen2.5-Coder-32B-Instruct (`Qwen2.5-Coder-32B-Instruct`). We set "n_sample" to 1 when generating code solutions and limit the max number of tokens to 4096. The weights for these models can be found at `https://huggingface.co/google/gemma-2-27b-it` and `https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct` respectively.

We access `c4ai-aya-expanse-32b` (`Aya`) through the Cohere API at `https://cohere.com/research/aya`, reka-core-20240501 (`Reka`) through the Reka API at `https://www.reka.ai/reka-api`, and deepseek-chat (`Deepseek-V3`) through the Deepseek API at `https://api-docs.deepseek.com/`. We use the

Feedback Style	Generated Feedback on APPS (Interview)
SENTENCE	The current solution doesn't account for the cyclic nature of the strings and may count duplicates, so we need to implement a proper cyclic string comparison and ensure uniqueness in the count.
PARAGRAPH	The current solution misunderstands the problem. It's not counting distinct cyclic strings correctly, as it doesn't consider all possible binary strings of length n that contain s as a substring. The solution should use a more sophisticated approach, possibly involving dynamic programming or KMP algorithm, to generate and count all valid cyclic strings. It needs to handle cases where s can appear at any position in the cyclic string, not just at the beginning. The current implementation also doesn't account for the cyclical nature of the strings properly. A complete overhaul of the algorithm is needed to correctly solve this problem.

CODE FEEDBACK	The main issue is that this solution doesn't correctly handle cyclic strings. The line:
	
	 "'python
	 if s in t or s in t+t:
	 "'
	
	 should be replaced with:
	
	 "'python
	 if s in (t * 2):
	 "'
	
	 This change ensures that all possible cyclic shifts are checked. Additionally, the solution doesn't account for the fact that cyclic shifts of the same string should be counted as one distinct string. A more fundamental restructuring of the algorithm is needed to correctly solve the problem as described in the full problem statement.

QUERY REPHRASING	Question: Calculate the number of unique cyclical binary strings of length n that include a given binary string s as a substring. A cyclical string contains s if any rotation of the string includes s. Two cyclical strings are different if they're not rotations of each other. Return the count modulo $10^9 + 7$

Table 3: Example feedback from `Sonnet-3.5`, given in response to a proposed solution for APPS Interview (Question #42)

default API settings for inference, limiting the max number of tokens to 4096.

`GPT-4o` inference is done through the OpenAI API `https://platform.openai.com/docs/overview` and `Sonnet-3.5` inference through the Anthropic API `https://www.anthropic.com/api`. We use the default API settings for inference, limiting the max number of tokens to 4096.

A.6 Prompts

We use `Sonnet-3.5` to generate user feedback for out experiments, but we show that other models can produce comparable feedback (Figure 7)

We provide all of the prompts for the user model and code model. All user model prompts were provided with the same system prompt with the original question and code solution (Figure 8). The PARAGRAPH prompt (Figure 9) and SENTENCE prompt (Figure 10) are given the current code model solution and generate feedback constrained by output length. The CODE FEEDBACK prompt is given the current code model solution and provides a correction to specific lines of code in the solution (Figure 11). The QUERY REPHRASING feedback prompt is given the current code model solution and underspecified question and generates an updated version of the question with missing details (Figure 12).

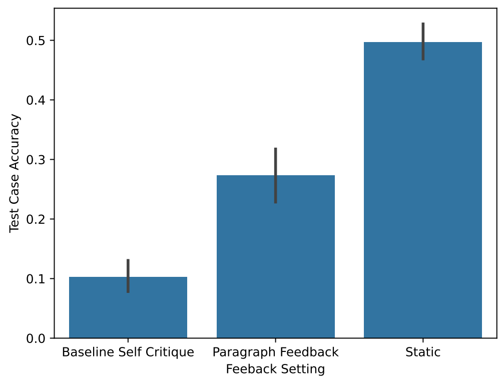

Figure 7: APPS (Interview) test case accuracy with `Sonnet-3.5` as the coding model and `GPT-4o-mini` as the user model providing feedback.

The code model prompts for APPS and LiveCodeBench are given in Figure 13 and Figure 14. The code model prompts for ClassEval is given in Figure 15 and Figure 16.

The 11-shot prompt used to summarize APPS and LiveCodeBench questions is given in Figure 17. The prompt used to summarize each docstring in ClassEval is given in Figure 18.

You are an expert human programmer who is using a coding assistant to write code in order to solve some programming puzzles. The coding assistant has completed a potential solution to the problem, but needs your help to make adjustments to the code.

You have access to the full question, including formatting instructions and some test cases. You also have access to a natural language description of the correct solution. The coding assistant has access to a summarized, less detailed version of the problem, but only you have access to the full problem. This means that the code assistant may need additional information on how the code should work or how its output should be formatted.

Here is the description of the programming problem:

{*full question*}

Here is a description of the correct solution:

{*solution info*}

Figure 8: System prompt given to user model. Blue text indicates that the relevant text would be inserted at that location in the prompt.

A.7 Performance Metrics

Test case accuracy. Test case accuracy can be defined as below:

$$\text{TCA} = \frac{\text{\# Test Cases Passed}}{\text{Total \# Test Cases}}$$

Normalized Spearman's Footrule distance. The normalized Spearman's Footrule distance is:

$$\tilde{F}(\sigma_A, \sigma_B) = \frac{\sum_{i=1}^{n} |\sigma_A(i) - \sigma_B(i)|}{\max_{\sigma,\sigma'} \sum_{i=1}^{n} |\sigma(i) - \sigma'(i)|}$$

Consider two rankings σ_A and σ_B over items $\{1, 2, \ldots, n\}$. To measure the distance between them, we use Spearman's Footrule Distance, which can be thought of as the Manhattan distance between two rankings:

$$F(\sigma_A, \sigma_B) = \sum_{i=1}^{n} |\sigma_A(i) - \sigma_B(i)|$$

We normalize F by its maximum possible value, $\frac{n^2}{2}$ for even n, to get the **Normalized Spearman's Footrule Distance**.

$$\tilde{F}(\sigma_A, \sigma_B) = \frac{F(\sigma_A, \sigma_B)}{\frac{n^2}{2}} = \frac{2F(\sigma_A, \sigma_B)}{n^2}$$

where $\tilde{F} :\rightarrow [0, 1]$. In other words, $\tilde{F} = 0$ indicates $\sigma_1 = \sigma_2$, whereas $\tilde{F} = 1$ indicates the maximum possible distance between σ_1 and σ_2.

Now, we would like to derive the expected $\tilde{F}$ between two rankings which are completely uncorrelated. Let us randomly sample σ_A, σ_B uniformly at random. Then the expected Spearman's Footrule Distance (F) is:

$$\begin{aligned}\mathbb{E}[F(\sigma_A, \sigma_B)] &= \mathbb{E}[\sum_{i=1}^{n} |\sigma_A(i) - \sigma_B(i)|] \\ &= \sum_{i=1}^{n} \mathbb{E}[|\sigma_A(i) - \sigma_B(i)|] \\ &= \sum_{i=1}^{n} \frac{n+1}{3} \\ &= \frac{n(n+1)}{3}\end{aligned}$$

Normalizing this by the maximum possible F gives:

$$\frac{\frac{n(n+1)}{3}}{\frac{n^2}{2}} = \frac{2(n+1)}{3n}$$

Thus, for uncorrelated rankings of length 10, $\tilde{F} \simeq 0.73$; for a perfectly correlated pair of rankings, $\tilde{F} = 0$; and for perfectly anti-correlated rankings, $\tilde{F} = 1$.

A.8 Additional Details on Measuring Feedback Quality

Automatic classification of directional correctness. We use `GPT-4o` to classify the feedback into

Your goal is to provide feedback about the solution that you think would help the assistant fix or adjust the code. This feedback should be purely about the function of the code, not its aesthetics or nonessential structure (e.g. do not make a suggestion regarding optimization or other choices that would not change how the code behaves). The coding assistant will use this feedback to help generate the next version of the code. Write one paragraph of feedback that you think would be most helpful to the coding assistant. Do not write more than 100 words.

Here is the code assistant's solution.

```python
{full solution}
```

Please provide one paragraph of feedback that would best help the coding assistant write a better version of the solution. Don't copy the code. Just write your reply.

Figure 9: Prompt given to user model to get PARAGRAPH feedback. Blue text indicates that the relevant text would be inserted at that location in the prompt.

two classes: (1) the feedback claimed that the solution was correct or (2) the feedback claimed that the solution was incorrect. As some feedback claims that the "logic" of the solution is correct, but then states that it is missing critical edge cases or input/output formatting, we also apply rule-based string matching to re-classify such feedback as incorrect. We then compare the feedback to the actual TCA performance to classify it into directionally correct vs. directionally incorrect feedback.

Directionally incorrect feedback inaccurately claims that correct solutions are incorrect or an incorrect solution is correct. We find that, in some cases, feedback in the latter case may still contain low-level suggestions that the model can use in the next round of iteration (e.g., an edge case). While our metric is a rough proxy of feedback accuracy, we expect that our conclusions should still hold because we apply it equally to all feedback types. The key takeaway remains that the effect of directional correctness varies by feedback type, even when feedback types perform similarly with respect to test case accuracy (e.g., CODE FEEDBACK and PARAGRAPH both induce the best performance, but PARAGRAPH feedback tends to have higher directional correctness compared to CODE FEEDBACK's lower directional correctness).

Aggregating directional correctness for Table 2 and Figure 4. For Table 2, we average across all steps of each interaction and then average across all interactions. For Figure 4, we plot on a step-wise basis, where "Solution Performance Change" refers to whether the ith step solution performed better or worse than the i-1th step solution. We bucket all feedback steps into "Performance [of Next Iteration of Solution] Increased" and "Performance Decreased", and then further divide each bucket into "Directionally Correct" and "Directionally Correct".

Other metrics of feedback quality. We considered two other metrics of feedback quality. First, we attempted to consider the increase in probability over either the ground-truth solution or the full question, comparing the code model's solution with and without feedback. However, we found that this measure was too noisy to impart any meaningful value.

We also attempted to prompt `GPT-4o` to classify feedback relevance (to either the ground-truth solution or full question) on a scale of 1-5. However, this measure was also noisy, not to mention hard to define in the prompt, as even humans would struggle to distinguish between, for example, a "2" vs. a "3" in relevance.

A.9 Additional SELF-CRITIQUE BASELINE Settings

The standard SELF-CRITIQUE BASELINE uses the underspecified question. We add an additional version of SELF-CRITIQUE BASELINE which provides the code model with the fully specified question rather than the underspecified question. We run this baseline on a subset of models for LiveCodeBench and show results in Table 4.

Your goal is to provide feedback about the solution that you think would help the assistant fix or adjust the code. This feedback should be purely about the function of the code, not its aesthetics or nonessential structure (e.g. do not make a suggestion regarding optimization or other choices that would not change how the code behaves). The coding assistant will use this feedback to help generate the next version of the code. Write one sentence of feedback that you think would be most helpful to the coding assistant. Do not write more than 50 words.

Here is the code assistant's solution.

```python
{full solution}
```

Please provide one sentence of feedback that would best help the coding assistant write a better version of the solution. Don't copy the code. Just write your reply.

Figure 10: Prompt given to user model to get SENTENCE feedback. Blue text indicates that the relevant text would be inserted at that location in the prompt.

Code Model	SELF-CRITIQUE BASELINE	SELF-CRITIQUE BASELINE w/ Fully Specified Queries
GPT-4o	0.323 (0.026)	0.767 (0.023)
Sonnet-3.5	0.387 (0.026)	0.891 (0.016)
Qwen2.5-Coder-32B-Instruct	0.309 (0.025)	0.869 (0.017)
Gemma-2-27B-it	0.282 (0.025)	0.796 (0.020)
Aya	0.235 (0.022)	0.488 (0.028)

Table 4: Performance of five models on SELF-CRITIQUE BASELINE with underspecified queries (via input obfuscation) and SELF-CRITIQUE BASELINE with fully specified queries on LiveCodeBench.

Unsurprisingly, SELF-CRITIQUE BASELINE with Fully Specified Queries improves greatly over the standard SELF-CRITIQUE BASELINE. We expect a strong model given full question specifications to improve via iterative refinement, which we can think of as an upper bound on performance in the interactive setting. In contrast, we expect our SELF-CRITIQUE BASELINE, which reflects more realistic input specifications, to perform poorly, demonstrating that model must collaborate with the user. We are happy to extend this to more datasets and models and add this to the next revision of the paper.

A.10 Multiple Feedback Types in the Same Interaction

We have implemented two versions of interactions where the user model can provide multiple feedback types in the same interaction. In RANDOM FEEDBACK, we randomly sample feedback types at each iteration of the interaction. In MIXED FEEDBACK, we list all feedback types in the prompt and allow the model to choose one to use at each step. To avoid ordering biases, we randomly shuffle the order in which the feedback types are listed each time the user model is queried.

We show results in these two settings for two datasets (ClassEval and LiveCodeBench) over 8 models in Tables 17 and 18. For both datasets, we continue to observe ranking changes in these settings, with ClassEval changing more dramatically than LiveCodeBench, as we also find in Figure 3.

A.11 Ranking Changes Across Code Readability

To demonstrate the versatility of our evaluation protocol, we extend our evaluation to include another axis of the coding experience: code readability. Specifically, we ran a PEP8 style checker (`pycodestyle`) to find the average number of violations in the final output for each question in every dataset and setting. For all 3 datasets, the Spearman's footrule ranking correlations between interactive settings and the static settings are in Table 5.

We note that our instructions to the user explicitly notes that feedback should be focused on functionality rather than aesthetics (e.g., code readabil-

Your goal is to provide feedback about the solution that you think would help the assistant fix or adjust the code. This feedback should be purely about the function of the code, not its aesthetics or nonessential structure (e.g. do not make a suggestion regarding optimization or other choices that would not change how the code behaves). The coding assistant will use this feedback to help generate the next version of the code. Point out specific lines of the code that are incorrect and explain why. Do not write more than 100 words.

Here is the code assistant's solution.

```python
{full solution}
```

Please point out specific lines of the code that are incorrect and give the corrected version. Make sure you copy paste the specific line in the solution which is incorrect! You should write both the original line (exactly as found in the solution) and also write what line it should be replaced with.

Figure 11: Prompt given to user model to get CODE FEEDBACK. Blue text indicates that the relevant text would be inserted at that location in the prompt.

Dataset	CODE FEEDBACK	QUERY REPHRASING	PARAGRAPH	SENTENCE
LiveCodeBench	0.222	0.4888	0.2222	0.1777
ClassEval	0.6667	0.7556	0.8444	0.7556
APPS	0.7555	0.8444	30.8	0.8

Table 5: Spearman's footrule ranking correlations between interactive settings and static settings using a code readability metric.

ity). Nonetheless, we still observe ranking changes, especially in LiveCodeBench. In other words, evaluating models in an interactive setting also affects code readability, even when the user is prompted to avoid giving feedback about readability! Future work could extend this to additional elements of the broader coding experience (e.g., maintainability or usability).

A.12 Performance Tables for Static vs. Interactive Settings

In this section, we provide tables for the performance of models across static and interactive settings, including all feedback types and baselines. Table 6 gives the TCA of APPS (Interview), Table 7 gives the TCA of APPS (Introductory). Table 8 gives the TCA of LiveCodeBench and Table 9.

A.13 Additional Tables

All additional tables – including information about feedback quality by dataset, steerability metrics, and ranking distance metrics — can be found in this section.

Table 10, Table 11, Table 12 have the average directional correctness of each setting and the number of steps it takes to reach a solution with 100% TCA. We partition the analysis by model and by feedback setting.

Table 14 measures the average number of edits made by each model for each feedback. Table 15 measures the average number of test cases flipped by each feedback setting.

Tables 16 gives the normalized Spearman's Footrule distance of each setting's ranking compared to the STATIC setting.

Model	Static	Self-Critique Baseline	Sentence	Paragraph	Code Feedback	Query Rephrasing
GPT-4o	0.498 (0.016)	0.068 (0.007)	0.422 (0.016)	0.544 (0.016)	**0.598 (0.016)**	0.488 (0.016)
Aya	**0.261 (0.014)**	0.009 (0.002)	0.131 (0.011)	0.259 (0.015)	0.235 (0.014)	0.214 (0.013)
Deepseek-V3	**0.616 (0.015)**	0.048 (0.007)	0.449 (0.016)	0.512 (0.023)	0.521 (0.026)	0.442 (0.019)
Gemma-2-27B-it	0.409 (0.013)	0.007 (0.002)	0.351 (0.016)	**0.556 (0.016)**	0.51 (0.017)	0.403 (0.015)
Gemma-7B-it	0.177 (0.01)	0.009 (0.002)	0.039 (0.006)	0.084 (0.009)	**0.299 (0.016)**	0.029 (0.004)
Llama-3.1-8B-Instruct	0.253 (0.012)	0.025 (0.003)	0.236 (0.014)	0.402 (0.015)	**0.453 (0.016)**	0.215 (0.012)
Qwen2.5-Coder-7B-Instruct	0.369 (0.014)	0.026 (0.004)	0.283 (0.014)	0.423 (0.016)	**0.495 (0.016)**	0.336 (0.015)
Qwen2.5-Coder-32B-Instruct	0.542 (0.015)	0.039 (0.004)	0.5 (0.016)	0.582 (0.015)	**0.605 (0.015)**	0.503 (0.015)
Reka	0.164 (0.011)	0.018 (0.003)	0.224 (0.013)	0.303 (0.015)	**0.4 (0.016)**	0.157 (0.011)
Sonnet-3.5	0.59 (0.014)	0.114 (0.009)	0.571 (0.015)	**0.654 (0.014)**	0.627 (0.015)	0.62 (0.014)

Table 6: Average TCA of each model with standard error on APPS Interview questions.

Model	Static	Self-Critique Baseline	Sentence	Paragraph	Code Feedback	Query Rephrasing
GPT-4o	0.412 (0.015)	0.071 (0.007)	0.392 (0.016)	0.512 (0.016)	**0.539 (0.016)**	0.409 (0.016)
Aya	0.182 (0.008)	0.004 (0.001)	0.069 (0.006)	**0.179 (0.009)**	0.177 (0.009)	0.142 (0.007)
Deepseek-V3	-	-	-	-	-	-
Gemma-2-27B-it	0.322 (0.012)	0.018 (0.003)	0.228 (0.013)	**0.454 (0.016)**	0.41 (0.016)	0.299 (0.013)
Gemma-7B-it	0.131 (0.008)	0.013 (0.003)	0.023 (0.004)	0.045 (0.006)	**0.213 (0.014)**	0.014 (0.003)
Llama-3.1-8B-Instruct	0.205 (0.01)	0.028 (0.004)	0.12 (0.009)	0.292 (0.014)	**0.402 (0.015)**	0.153 (0.01)
Qwen2.5-Coder-7B-Instruct	0.28 (0.012)	0.034 (0.004)	0.186 (0.012)	0.27 (0.014)	**0.392 (0.016)**	0.2 (0.011)
Qwen2.5-Coder-32B-Instruct	0.348 (0.024)	0.038 (0.007)	0.376 (0.023)	0.479 (0.025)	**0.486 (0.025)**	0.37 (0.023)
Reka	0.124 (0.009)	0.018 (0.003)	0.141 (0.01)	0.275 (0.014)	**0.34 (0.015)**	0.11 (0.009)
Sonnet-3.5	0.497 (0.014)	0.073 (0.007)	0.468 (0.015)	**0.556 (0.015)**	0.53 (0.015)	0.492 (0.015)

Table 7: Average TCA of each model with standard error in each setting for APPS introductory. `Deepseek-V3` is missing for this setting due to rate limits on the API that impeded evaluation.

Model	Static	Self-Critique Baseline	Sentence	Paragraph	Code Feedback	Query Rephrasing
GPT-4o	**0.8 (0.023)**	0.323 (0.026)	0.767 (0.024)	0.745 (0.024)	0.702 (0.026)	0.23 (0.025)
Aya	0.522 (0.027)	0.235 (0.022)	0.482 (0.027)	**0.632 (0.026)**	0.483 (0.028)	0.134 (0.02)
Deepseek-V3	**0.944 (0.013)**	0.332 (0.026)	0.841 (0.02)	0.849 (0.019)	0.756 (0.024)	0.345 (0.028)
Gemma-2-27B-it	**0.766 (0.021)**	0.282 (0.025)	0.67 (0.026)	**0.766 (0.023)**	0.62 (0.026)	0.119 (0.019)
Gemma-7B-it	0.347 (0.023)	0.173 (0.019)	0.196 (0.021)	0.285 (0.024)	**0.524 (0.027)**	0.028 (0.008)
Llama-3.1-8B-Instruct	0.587 (0.025)	0.237 (0.022)	0.538 (0.027)	0.588 (0.026)	**0.647 (0.026)**	0.203 (0.023)
Qwen2.5-Coder-7B-Instruct	**0.755 (0.021)**	0.27 (0.024)	0.621 (0.027)	0.678 (0.026)	0.629 (0.027)	0.189 (0.023)
Qwen2.5-Coder-32B-Instruct	**0.961 (0.007)**	0.309 (0.025)	0.809 (0.021)	0.747 (0.024)	0.712 (0.025)	0.237 (0.025)
Reka	0.617 (0.025)	0.246 (0.023)	0.583 (0.026)	**0.636 (0.026)**	0.629 (0.027)	0.111 (0.018)
Sonnet-3.5	**0.937 (0.012)**	0.387 (0.026)	0.832 (0.02)	0.821 (0.02)	0.735 (0.025)	0.395 (0.029)

Table 8: Average TCA of each model with standard error on a subset LiveCodeBench questions. To provide code solutions to the user model, we select questions which `Sonnet-3.5` solves perfectly within two attempts, using the generated solution as ground truth.

Model	Static	Self-Critique Baseline	Sentence	Paragraph	Code Feedback	Query Rephrasing
GPT-4o	0.839 (0.005)	0.561 (0.018)	0.836 (0.014)	0.848 (0.014)	**0.895 (0.011)**	0.789 (0.014)
Aya	0.718 (0.007)	0.305 (0.019)	0.596 (0.021)	0.597 (0.022)	**0.781 (0.018)**	0.675 (0.016)
Deepseek-V3	0.849 (0.005)	0.559 (0.018)	0.837 (0.013)	0.867 (0.012)	**0.889 (0.012)**	0.806 (0.013)
Gemma-2-27B-it	0.704 (0.008)	0.563 (0.018)	0.776 (0.015)	0.815 (0.015)	**0.867 (0.014)**	0.735 (0.015)
Gemma-7B-it	0.350 (0.008)	0.303 (0.017)	0.375 (0.018)	0.390 (0.018)	**0.687 (0.019)**	0.340 (0.017)
Llama-3.1-8B-Instruct	0.636 (0.007)	0.443 (0.019)	0.653 (0.019)	0.710 (0.019)	**0.773 (0.017)**	0.701 (0.018)
Qwen2.5-Coder-7B-Instruct	0.714 (0.006)	0.502 (0.019)	0.605 (0.018)	0.757 (0.017)	**0.819 (0.014)**	0.697 (0.016)
Qwen2.5-Coder-32B-Instruct	0.816 (0.006)	0.542 (0.018)	0.815 (0.014)	0.832 (0.015)	**0.876 (0.012)**	0.778 (0.014)
Reka	0.670 (0.007)	0.471 (0.027)	0.096 (0.015)	0.109 (0.016)	0.112 (0.016)	**0.600 (0.019)**
Sonnet-3.5	0.833 (0.006)	0.564 (0.019)	0.821 (0.014)	0.865 (0.013)	**0.881 (0.013)**	0.803 (0.014)

Table 9: Average TCA with standard error in each setting for ClassEval.

Model	Feedback Type	Average Steps to Correct Solution	Average Directional Correctness
GPT-4o	Code Feedback	2.981	0.937
	Paragraph	3.003	0.896
	Sentence	3.436	0.850
Aya	Code Feedback	3.586	0.928
	Paragraph	3.687	0.926
	Sentence	3.886	0.901
Deepseek-V3	Code Feedback	3.207	0.928
	Paragraph	3.132	0.886
	Sentence	3.484	0.873
Gemma-2-27B-it	Code Feedback	3.296	0.948
	Paragraph	3.259	0.920
	Sentence	3.630	0.895
Gemma-7B-it	Code Feedback	3.661	0.978
	Paragraph	3.947	0.982
	Sentence	3.977	0.950
Llama-3.1-8B-Instruct	Code Feedback	3.349	0.963
	Paragraph	3.567	0.923
	Sentence	3.850	0.897
Qwen2.5-Coder-7B-Instruct	Code Feedback	3.328	0.948
	Paragraph	3.453	0.909
	Sentence	3.724	0.899
Qwen2.5-Coder-32B-Instruct	Code Feedback	3.082	0.952
	Paragraph	3.058	0.900
	Sentence	3.411	0.876
Reka	Code Feedback	3.520	0.935
	Paragraph	3.634	0.927
	Sentence	3.854	0.879
Sonnet-3.5	Code Feedback	3.035	0.914
	Paragraph	2.882	0.865
	Sentence	3.299	0.834

Table 10: Average directional correctness of feedback and the average number of steps required to reach 100% TCA on the APPS dataset.

Model	Feedback Type	Average Steps to Correct Solution	Average Directional Correctness
GPT-4o	Code Feedback	2.697	0.736
	Paragraph	2.337	0.902
	Sentence	2.416	0.821
Aya	Code Feedback	2.856	0.852
	Paragraph	2.553	0.945
	Sentence	3.097	0.898
Deepseek-V3	Code Feedback	2.487	0.785
	Paragraph	1.808	0.891
	Sentence	2.063	0.806
Gemma-2-27B-it	Code Feedback	3.089	0.809
	Paragraph	2.246	0.912
	Sentence	2.685	0.896
Gemma-7B-it	Code Feedback	3.229	0.927
	Paragraph	3.640	0.986
	Sentence	3.818	0.963
Llama-3.1-8B-Instruct	Code Feedback	2.773	0.888
	Paragraph	2.672	0.957
	Sentence	3.169	0.890
Qwen2.5-Coder-7B-Instruct	Code Feedback	2.876	0.786
	Paragraph	2.314	0.958
	Sentence	2.847	0.909
Qwen2.5-Coder-32B-Instruct	Code Feedback	2.811	0.771
	Paragraph	1.949	0.871
	Sentence	2.219	0.817
Reka	Code Feedback	2.801	0.668
	Paragraph	2.575	0.883
	Sentence	2.871	0.807
Sonnet-3.5	Code Feedback	2.663	0.701
	Paragraph	2.017	0.885
	Sentence	2.247	0.859

Table 11: Average directional correctness of feedback and the average number of steps required to reach 100% TCA on the LiveCodeBench dataset.

Model	Feedback Type	Average Steps to Correct Solution	Average Directional Correctness
GPT-4o	Code Feedback	3.436	0.743
	Paragraph	3.546	0.963
	Sentence	3.153	0.899
Aya	Code Feedback	3.977	0.749
	Paragraph	3.987	0.953
	Sentence	3.840	0.898
Deepseek-V3	Code Feedback	4.000	0.698
	Paragraph	4.000	0.969
	Sentence	4.000	0.910
Gemma-2-27B-it	Code Feedback	3.076	0.789
	Paragraph	3.334	0.977
	Sentence	3.661	0.917
Gemma-7B-it	Code Feedback	3.880	0.902
	Paragraph	3.874	0.979
	Sentence	3.940	0.944
Llama-3.1-8B-Instruct	Code Feedback	3.389	0.872
	Paragraph	3.806	0.959
	Sentence	3.699	0.899
Qwen2.5-Coder-7B-Instruct	Code Feedback	3.684	0.782
	Paragraph	3.724	0.967
	Sentence	3.322	0.929
Qwen2.5-Coder-32B-Instruct	Code Feedback	3.536	0.744
	Paragraph	4.000	0.977
	Sentence	3.585	0.889
Reka	Code Feedback	3.978	0.769
	Paragraph	4.000	0.942
	Sentence	4.000	0.901
Sonnet-3.5	Code Feedback	3.664	0.659
	Paragraph	3.675	0.947
	Sentence	4.000	0.913

Table 12: Average directional correctness of feedback and the average number of steps required to reach 100% TCA on the ClassEval dataset

Model	Feedback Type	Average Steps to Correct Solution	Average Directional Correctness
GPT-4o	Code Feedback	2.925	0.904
	Paragraph	2.916	0.906
	Sentence	3.293	0.858
Aya	Code Feedback	3.400	0.918
	Paragraph	3.516	0.934
	Sentence	3.738	0.905
Deepseek-V3	Code Feedback	2.814	0.831
	Paragraph	2.721	0.929
	Sentence	3.171	0.886
Gemma-2-27B-it	Code Feedback	3.218	0.915
	Paragraph	3.120	0.925
	Sentence	3.479	0.903
Gemma-7B-it	Code Feedback	3.545	0.969
	Paragraph	3.888	0.982
	Sentence	3.936	0.952
Llama-3.1-8B-Instruct	Code Feedback	3.250	0.945
	Paragraph	3.427	0.930
	Sentence	3.708	0.900
Qwen2.5-Coder-7B-Instruct	Code Feedback	3.237	0.921
	Paragraph	3.287	0.917
	Sentence	3.663	0.874
Qwen2.5-Coder-32B-Instruct	Code Feedback	3.002	0.919
	Paragraph	2.931	0.908
	Sentence	3.265	0.883
Reka	Code Feedback	3.508	0.889
	Paragraph	3.560	0.929
	Sentence	3.761	0.878
Sonnet-3.5	Code Feedback	2.983	0.877
	Paragraph	2.803	0.880
	Sentence	3.193	0.849

Table 13: Average directional correctness of feedback and the average number of steps required to reach 100% TCA across all datasets.

Model	SENTENCE	PARAGRAPH	CODE FEEDBACK	QUERY REPHRASING
GPT-4o	265.029	308.837	451.749	325.441
Aya	333.695	274.630	503.740	351.736
Deepseek-V3	143.413	190.044	301.157	264.532
Gemma-2-27B-it	116.412	108.401	182.592	108.516
Gemma-7B-it	286.987	211.680	299.998	261.134
Llama-3.1-8B-Instruct	361.366	400.246	511.731	385.156
Qwen2.5-Coder-7B-Instruct	214.421	178.882	363.474	211.171
Qwen2.5-Coder-32B-Instruct	270.493	230.114	496.959	328.425
Reka	541.689	292.388	768.962	638.245
Sonnet-3.5	155.923	204.138	320.164	215.693

Table 14: Surface-level steerability (as measured by edit distance) vs. feedback type across each model.

Model	SENTENCE	PARAGRAPH	CODE FEEDBACK	QUERY REPHRASING
GPT-4o	0.225	0.164	0.240	0.152
Aya	0.169	0.090	0.208	0.108
Deepseek-V3	0.159	0.138	0.208	0.161
Gemma-2-27B-it	0.149	0.113	0.199	0.095
Gemma-7B-it	0.094	0.007	0.040	0.015
Llama-3.1-8B-Instruct	0.2	0.102	0.201	0.107
Qwen2.5-Coder-7B-Instruct	0.157	0.095	0.184	0.101
Qwen2.5-Coder-32B-Instruct	0.228	0.142	0.263	0.192
Reka	0.154	0.059	0.168	0.095
Sonnet-3.5	0.239	0.225	0.311	0.212

Table 15: Behavioral-level steerability (as measured by number of test cases changed from correct to incorrect or vice-versa) vs. feedback type across each model.

Dataset	Feedback Type	Normalized Spearman's Footrule Distance
Apps	Code Feedback	0.267
	Input Refinement	0.222
	Paragraph	0.222
	Sentence	0.178
ClassEval	Code Feedback	0.222
	Input Refinement	0.267
	Paragraph	0.267
	Sentence	0.267
LiveCodeBench	Code Feedback	0.356
	Input Refinement	0.356
	Paragraph	0.222
	Sentence	0.044

Table 16: Normalized Spearman's Footrule Distance when comparing each feedback setting's ranking order to the ranking order on static benchmark.

Here is the code assistant's solution.

```python
{full solution}
```

Here is the previous version of the question.

{underspecified question}

Please rewrite the question so as to provide an updated question of similar length which would help the model generate a better version of the code. Make sure you don't make the new question longer than the older version! Begin your response with "Question:" and don't add any extra text to the end.

Figure 12: Prompt given to user model to get QUERY REPHRASING feedback. Blue text indicates that the relevant text would be inserted at that location in the prompt.

You are a coding assistant who is writing code in order to solve some programming puzzles.

You will be provided with a summary of the problem. You may also be provided with some starter code that you need to complete.

Your goal is to complete the code so as to solve the problem. Do not add anything else to the code, including natural language that is not part of the code or comments. Your generation should be ready to run without needing any modifications.

APPS: Keep in mind that for this dataset, the results should be printed to stdout, so don't just write a function without calling it or printing something to stdout.

Here is the programming problem description:

{underspec question}

Enclose your solution in a markdown block beginning with ```python. When you are ready to submit your response, please end the markdown block with ``` on a new line.

```python
{partial solution}
```

Figure 13: Prompt given to code model in the STATIC, SELF-CRITIQUE BASELINE, and QUERY REPHRASING settings for APPS and LiveCodeBench. Blue text indicates that the relevant text would be inserted at that location in the prompt. Orange text is the APPS-specific formatting instructions. Red text is prefill for the code model.

Code Model	STATIC	MIXED FEEDBACK	RANDOM FEEDBACK
GPT-4o	1	2	2
Sonnet-3.5	2	1	1
Qwen2.5-Coder-32B-Instruct	3	4	3
Gemma-2-27B-it	4	3	4
Aya	5	7	7
Llama-3.1-8B-Instruct	6	5	6
Qwen2.5-Coder-7B-Instruct	7	6	5
Gemma-7B-it	8	8	8

Table 17: Performance of 8 models on STATIC, MIXED FEEDBACK, and RANDOM FEEDBACK on ClassEval.

You are a coding assistant who is writing code in order to solve some programming puzzles.

You will be provided with a summary of the problem. You may also be provided with some starter code that you need to complete.

Your goal is to complete the code so as to solve the problem. Do not add anything else to the code, including natural language that is not part of the code or comments. Your generation should be ready to run without needing any modifications.

APPS: Keep in mind that for this dataset, the results should be printed to stdout, so don't just write a function without calling it or printing something to stdout.

Here is the programming problem description:

{underspec question}

Here is your last version of the code:

```python
*{prev solution}*
```

The user provided the following feedback on the code:

{user response}

You can choose to use this response, or you can choose to ignore it. Only incorporate the information that you think is relevant and helpful into the code.
Enclose your solution in a markdown block beginning with ```python. When you are ready to submit your response, please end the markdown block with ``` on a new line.

```python

Figure 14: Prompt given to code model in the PARAGRAPH, SENTENCE, CODE FEEDBACK setting for APPS and LiveCodeBench. Blue text indicates that the relevant text would be inserted at that location in the prompt. Orange text is the APPS specific formatting instructions. Red text is prefill for the code model.

| Code Model | STATIC | MIXED FEEDBACK | RANDOM FEEDBACK |
|---|---|---|---|
| Qwen2.5-Coder-32B-Instruct | 1 | 2 | 2 |
| Sonnet-3.5 | 2 | 1 | 1 |
| GPT-4o | 3 | 3 | 3 |
| Gemma-2-27B-it | 4 | 4 | 4 |
| Qwen2.5-Coder-7B-Instruct | 5 | 5 | 5 |
| Aya | 6 | 6 | 7 |
| Gemma-7B-it | 7 | 7 | 6 |
| Llama-3.1-8B-Instruct | 8 | 8 | 8 |

Table 18: Performance of 8 models on STATIC, MIXED FEEDBACK, and RANDOM FEEDBACK on LiveCodeBench.
```

You are a coding assistant who is writing code in order to fill in the skeleton of a python class.

You will be provided with the skeleton of the class with function names and docstrings. You may also be provided with some functions already completed.

Your goal is to complete the code according to the docstrings. Do not add anything else to the code, including natural language that is not part of the code or comments. Your generation should be ready to run without needing any modifications.

Here is the skeleton:

{underspec question}

Enclose your solution in a markdown block beginning with ```python. When you are ready to submit your response, please end the markdown block with ``` on a new line.

~~~python

*{partial solution}*

Figure 15: Prompt given to code model in the STATIC, SELF-CRITIQUE BASELINE, and QUERY REPHRASING settings for ClassEval. Blue text indicates that the relevant text would be inserted at that location in the prompt. Red text is prefill for the code model.

You are a coding assistant who is writing code to fill in some incomplete classes.

You will be provided with the skeleton of the class with function names and docstrings. You may also be provided with some functions already completed.

Your goal is to complete the code according to the docstrings. Do not add anything else to the code, including natural language that is not part of the code or comments. Your generation should be ready to run without needing any modifications.

Here is the skeleton:

*{underspec question}*

Here is your last version of the skeleton:

```python

*{prev solution}*

``` The user provided the following feedback on the code:

*{user response}* You can choose to use this response, or you can choose to ignore it. Only incorporate the

information that you think is relevant and helpful into the code.
Enclose your solution in a markdown block beginning with ```python. When you are ready to submit your response, please end the markdown block with ``` on a new line.

```python

Figure 16: Prompt given to code model in the PARAGRAPH, SENTENCE, CODE FEEDBACK setting for Classeval. Blue text indicates that the relevant text would be inserted at that location in the prompt. Red text is prefill for the code model.
~~~

```
Summarize the question with header 'FORMAL QUESTION' using only natural language. Write your
summary under 'SUMMARY'
Do not use any variable or function names from the question. Do not write any code.
"You are given a coding/algorithmic question below. Your goal is to come up with a {sent length}
sentence summary using natural language to describe the problem.
"Here are examples of a question and a summary labeled 'EX QUESTION' and 'EX SUMMARY'.
Format your summary of 'FORMAL QUESTION' in a similar way to these summaries using {sent
length} sentence(s).

###EX QUESTION
Example question 1

###EX SUMMARY
Example summary 1

###EX QUESTION
Example question 2

###EX SUMMARY
Example summary 2

###EX QUESTION
Example question 3

###EX SUMMARY
Example summary 3
⋮

###EX QUESTION
Example question 11

###EX SUMMARY
Example summary 11

###FORMAL QUESTION
{question}

###SUMMARY
<YOUR SUMMARY HERE>
```

Figure 17: Format for the 11 shot prompt we use to generate summaries in APPS and LiveCodeBench problems. Blue text indicates that the relevant text would be inserted at that location in the prompt.

Do not reference any variables or function names. Do not write any code or examples of behavior.

You are given a method signature and a docstring below. Write a short, one sentence summary of the docstring. Do not use code or examples in your summary. Retain only the key information. Do not use more than 15 words.

SIGNATURE and DOCSTRING *{function}*
SUMMARY <YOUR SUMMARY HERE>

Figure 18: Prompt we use to generate summarized docstring for each function in ClassEval skeletons. Blue text indicates that the relevant text would be inserted at that location in the prompt.