WHEN DOES FEEDBACK HELP? EVALUATING FEEDBACK FOR ITERATIVE REFINEMENT IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in AI increasingly rely on agents that iteratively refine their responses at test time using *feedback* from an environment. Yet it remains unclear what type of feedback most effectively enables an agent to make discoveries beyond its existing capabilities. We investigate this question using a controlled testbed of coding problems spanning different difficulty levels. We compare six forms of feedback most commonly used in practice: detailed execution reports, aggregate performance scores, self-reflection, external LLM code review, failures on LLM-generated tests, and ground-truth hints written by human experts. Using this framework, we show that feedback effectiveness depends strongly on problem difficulty: On easy problems, nearly all feedback types enable the agent to reach full accuracy when the agent was unable to solve them in the initial attempt. On medium problems, feedback derived from ground-truth solutions yields the largest improvement (40.4%), followed by feedback from LLM-generated test cases (24.1%). On hard problems, however, even ground-truth hints provide only a marginal improvement (5.3%). In this case, feedback based on LLM-generated test cases remains the most effective (20.4%). Nevertheless, under the same inference budget, best-of-N parallel sampling outperforms every iterative-feedback method in hard problems. In fact, we show that the gains from refinement are proportional to the gap between the initial response and the best solution obtainable through parallel sampling. These findings reveal an *illusion of sequential refinement*: on difficult problems, iterative refinement primarily helps the agent recover from poor initial generations and approach, but not surpass, the best solutions it could independently generate.

1 INTRODUCTION

Recent research in AI aims to tackle frontier problems that extend beyond its current capabilities, including scientific discovery and autonomous research (Lu et al., 2026; Novikov et al., 2025; Si et al., 2026; Romera-Paredes et al., 2024; Zhang et al., 2026a). An increasingly common approach in agentic systems is to build agents that refine their responses by repeatedly interacting with an environment at test time (see Figure 1) (Zheng et al., 2025a; Li et al., 2025; Wu et al., 2025; Si et al., 2026). At each iteration, the agent takes an action, observes feedback on the outcome, and uses that feedback to revise its strategy (Shinn et al., 2023; Gou et al., 2024; Yang et al., 2024).

These systems rely on the premise that sufficiently informative feedback enables the agent to improve beyond its initial one-shot performance (Gou et al., 2024; Kamoi et al., 2024; Shinn et al., 2023). However, feedback in real-world environments may be noisy, sparse, or delayed (Dulac-Arnold et al., 2021; Pignatelli et al., 2023). We therefore ask a fundamental question about the role of feedback: ***What type of feedback most effectively enables an agent to make discoveries beyond its existing capabilities?***

To study this question, we propose a controlled testbed for systematically ablating different types of feedback. Our framework uses coding problems curated from Codeforces (Zheng et al., 2025b), spanning three difficulty levels. Each problem includes a natural-language description and a collection of predefined test cases.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

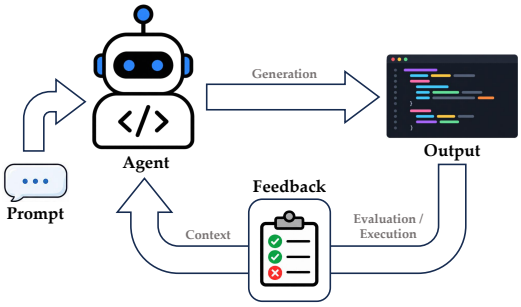


Figure 1: Overview of our experimental setup. Given an input prompt, the agent generates a response, which is evaluated to produce feedback. This feedback is added to the agent's context and used to guide its next response. This process is repeated iteratively. Our study investigates which types of feedback most effectively support improvement and whether feedback can enable the agent to discover solutions beyond its initial capabilities.

We particularly compare six types of feedback: (1) a *detailed execution report*, which provides the agent with the test cases that its solution failed on from the predefined test set; (2) an *aggregate performance score*, which merely reports the total number of passed test cases; (3) *self-reflection*, in which the agent critiques and revises its own code; (4) *external code review*, which provides prescriptive feedback generated by an independent LLM judge; (5) *LLM-generated tests*, which reports failures on test cases generated by an external LLM; and (6) *ground-truth hints*, which provide guidance using a human-written reference strategy.

To evaluate the effectiveness of feedback, the agent first makes an initial attempt to solve the problem. Then, we measure the improvement over the initial attempt produced by each feedback type. Our experiments reveal that the effectiveness of feedback strongly depends on question difficulty: On easy questions, nearly all feedback types enable the agent to reach full accuracy when the agent was unable to solve them in the initial attempt. On problems with medium difficulty, ground-truth hints demonstrate a 40.4% average improvement over the agents' initial attempts, while feedback based on LLM-generated test cases shows 24.1% improvement. The remaining feedback types provide smaller but still meaningful gains.

Hard problems show a surprisingly different pattern: even ground-truth hints provide only a marginal 5.3% improvement. Our results suggest that, **on sufficiently difficult problems, agents may be unable to effectively translate even highly informative feedback into better solutions**. We additionally show that failure reports from LLM-generated test cases remain the most effective type of feedback on hard questions, with a 20.4% improvement. At the same time, we show that all types of feedback are outperformed by best-of-N parallel sampling with the same inference budget. In fact, we observe that **iterative refinement can only improve the initial response up to the models' best attempt through parallel sampling.**

Together, these findings reveal an *illusion of sequential refinement*: when facing sufficiently hard problems, agents cannot make novel advancements through interaction with the environment. Instead, iterative refinement primarily acts as a form of conditional resampling, helping the model recover from a poor initial generation and approach, but not surpass, the best solution it could have produced independently through parallel sampling.

In summary, our main contributions are:

- We introduce a controlled testbed for comparing different forms of iterative feedback.
- We show that feedback effectiveness depends on question difficulty: On hard questions, feedback from LLM-generated test cases is the most effective, whereas ground-truth hints or external LLM code reviews offer no meaningful advantage.
- We identify an *illusion of refinement*: under the same inference budget, best-of-N parallel sampling outperforms all feedback-based methods in hard problems, suggesting that iterative refinement primarily acts as conditional resampling rather than enabling genuine discovery.

2 Related Work

LLM-based coding agents. LLM-based coding agents extend one-shot code generation by using language models as central controllers that can plan, invoke development tools, observe execution outcomes, and iteratively revise their solutions. Systems such as MetaGPT and ChatDev organize multiple specialized agents to emulate collaborative software-development workflows span-

ning requirement analysis, system design, implementation, testing, and debugging (Hong et al., 2024; Qian et al., 2024). SWE-agent instead equips an LLM with an agent–computer interface for navigating repositories, editing files, executing programs, and resolving software-engineering issues (Yang et al., 2024). More specialized frameworks, such as AgentCoder, divide code generation, test generation, and test execution among different agents and refine programs using execution feedback (Huang et al., 2023), while LDB incorporates fine-grained runtime traces to support error localization and program repair (Zhong et al., 2024). These systems demonstrate that tool use, role specialization, and iterative interaction can improve the reliability of generated software. However, because they jointly vary agent architecture, tool access, search strategy, and feedback source, it remains difficult to isolate which type of feedback is primarily responsible for their performance gains.

Sequential and parallel test-time inference. From the perspective of inference-time computation, existing approaches can be broadly divided into sequential and parallel paradigms. *Sequential* methods condition each new attempt on previous generations or feedback, allowing the model to iteratively revise an existing solution trajectory (Madaan et al., 2023; Shinn et al., 2023; Muennighoff et al., 2025). For example, s1 (Muennighoff et al., 2025) uses budget forcing to encourage the model to continue reasoning for longer by appending "Wait" when it attempts to stop. *Parallel* methods instead generate multiple independent trajectories and select or aggregate their outputs using mechanisms such as self-consistency, reward models, or executable tests (Wang et al., 2022; Li et al., 2022; Snell et al., 2024). For example, Brown et al. (2024) show that repeatedly sampling independent solutions substantially increases the probability of finding a solution that the model can solve.

Several studies have directly compared sequential and parallel test-time computation. Olausson et al. (2024) find that the relative effectiveness of sequential self-repair and parallel sampling depends on the available inference budget. Wu et al. (2025) find that increasing the diversity of initial reasoning paths can outperform deeper iterative reasoning. Concurrently, Gu et al. (2026) show that parallel sampling can outperform sequential refinement on mathematical reasoning and coding tasks. However, these studies do not systematically examine how the *type of feedback* or *problem difficulty* affects the effectiveness of sequential refinement. Our work addresses this gap by comparing different feedback types within a controlled refinement framework and examining how their effectiveness changes with problem difficulty. This allows us to distinguish the benefits of feedback from those of simply allocating more inference-time computation.

Beyond these approaches, recent discovery systems combine sequential improvement with population-based parallel exploration (Novikov et al., 2025; Zhang et al., 2026a). Our findings suggest that, particularly on difficult coding problems, sequential feedback alone may be insufficient to discover solutions beyond those accessible through parallel sampling, highlighting the complementary role of diversity and parallel exploration in test-time search.

3 PRELIMINARIES

3.1 PROBLEM SETUP

We study iterative test-time refinement for code generation. This task is particularly suitable for evaluating feedback signals because objective test cases are abundant, allowing generated programs to be automatically executed against them. The execution environment can then provide feedback at different levels of granularity, ranging from an aggregate test-passing rate to detailed information about failed test cases.

Let x denote a code-generation problem and f denote the target large language model whose code-generation and refinement abilities we intend to evaluate. For each problem, the target model first generates an initial program $c_1 \sim f(\cdot \mid x)$. The generated program is then evaluated and transformed into a particular form of feedback, denoted by z_1. In each subsequent round, the target model revises its program based on the problem statement, the program generated in the previous round, and the provided feedback. The overall refinement process is expressed as $c_n \sim f(\cdot \mid x, c_{n-1}, z_{n-1})$, where $n \in \{2, \ldots, N\}$ denotes the refinement round and z_n is the feedback extracted from c_n. After each round, the revised program is evaluated again to obtain updated feedback for the next round. This process is repeated for up to N rounds.

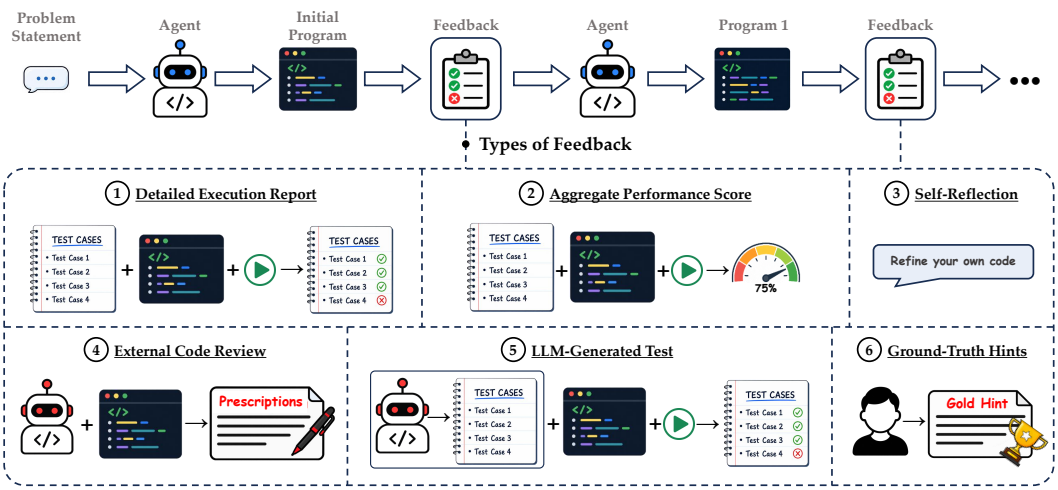


Figure 2: The six feedback types considered in our study. The agent first makes an initial attempt to solve the problem. Then, it iteratively uses the provided feedback to improve its solution.

In our setup, the parameters of the target model remain fixed throughout this procedure. Thus, the proposed setting evaluates test-time refinement rather than additional model training. To ensure a controlled comparison, all feedback types begin with the same initial program for each problem and use the same decoding configuration.

3.2 Feedback Types

We investigate six types of feedback, shown in Figure 2, that represent common scenarios encountered in practical agentic systems:

Detailed Execution Report. The generated program is executed against the available test cases, and the target model receives detailed information about each failed test case. Depending on the execution environment, this information may include the test input, expected output, actual output, compilation error, runtime exception, or timeout message. This feedback represents the most informative form of execution-based feedback considered in our study.

Aggregate Performance Score. The target model receives only the proportion of test cases passed by its current program, such as "The program passed 90% of the test cases". It is not informed of which test cases failed or why they failed. This feedback allows us to examine whether a coarse performance signal alone is sufficient to facilitate effective refinement.

Self-Reflection. The model receives no feedback from any external sources. Instead, it is explicitly instructed to inspect its previous program and perform self-refinement to produce an improved implementation. Therefore, this feedback measures the extent to which improvement can arise from repeated inference and explicit self-reflection alone. The complete self-reflection prompt is provided in Appendix A.1.

External Code Review. A separate reviewer LLM examines the problem and the program generated by the target model. The reviewer provides a natural-language critique that may identify logical errors, incorrect assumptions, missing edge cases, efficiency issues, or implementation defects. The reviewer does not receive execution results and instead evaluates the solution using only the problem specification and generated code. The prescription is then passed to the primary agent as an actionable recommendation for the next iteration. This feedback type, therefore, isolates the effect of semantic, prescriptive feedback produced by an independent LLM.

LLM-Generated Test. A separate LLM generates a new set of test cases and their expected outputs based on the programming problem once, and uses them instead of the predefined test suite. The target model's program is then executed only against these generated tests, and detailed execution results are provided as feedback. The subsequent procedure is the same as that used for the detailed execution report.

Ground-Truth Hints. The target model receives human-written natural-language guidance collected from the official editorial solutions. These hints describe the key algorithmic insight or strategy required to solve the problem without directly providing the complete implementation. This condition provides the richest solution-specific guidance and allows us to test whether direct access to the correct underlying strategy enables successful refinement.

Each of these conditions represents a stylized version of common agentic harnesses that are used in practice, and we are particularly interested in isolating the effect of feedback from other design choices in these systems.

3.3 Parallel Sampling with Best-of-N

In addition to the aforementioned methods that rely on feedback to perform iterative refinement, we evaluate a simple baseline: best-of-N parallel sampling. Given a problem x, we independently sample N candidate programs from the target model as $c_n \sim f(\cdot \mid x)$, where $n \in \{1, \ldots, N\}$. Unlike iterative refinement, each candidate is generated directly from the original problem and does not observe previous candidates or feedback from the environment.

Crucially, we fix the total generation budget to be identical across iterative refinement and parallel sampling. For iterative refinement, this budget includes the initial program and all subsequent revisions. Both methods, therefore, use the same number of target-model calls and the same decoding configuration.

For analysis, we define best-of-N performance using the candidate that achieves the highest score on the test suite.

4 Empirical Analysis

We evaluate feedback strategies on competitive-programming problems drawn from LiveCodeBench-Pro (Zheng et al., 2025b). We sample problems whose test cases fit within the model's context-window limit. This includes 40 problems stratified by difficulty (15 easy, 15 medium, 10 hard). We use `gpt-oss-120b` as our main model, and defer the results for `gpt-5.4-nano`, `gpt-oss-20b`, `gemma-4-E4B-it`, and `gemma-4-31B` to Appendix B.1. For experiments that require an external LLM for feedback, we use a separate instance of the same model, initialized with a fresh context. Appendix A provides additional experimental details, including the sampling configurations.

Regardless of the feedback type, all generated programs are evaluated against the predefined test cases and scored based on the proportion of test cases they pass. Compilation errors, runtime errors, incorrect outputs, and time-limit violations are treated as failures. We deliberately do not reserve a separate held-out test set, as our objective is not to evaluate generalization to unseen test cases. Instead, we investigate how effectively agents can use feedback to improve their solutions, including in the favorable setting where the predefined test cases are directly exposed to the model.

We repeat each experiment 10 times for every problem and feedback type to account for sampling variance. Because our objective is to measure improvement through iterative refinement, we terminate the refinement process early once a program passes the complete test suite.

Unless stated otherwise, we allocate the same budget of $N = 10$ model calls to both parallel sampling and iterative refinement. For each problem, we report the score achieved in the last refinement round and measure improvement relative to the shared initial attempt. Appendix B.2 reports complementary results using the maximum score achieved across all refinement rounds rather than the final-round score.

4.1 Effectiveness of the feedback depends on the problem difficulty

Figure 3 summarizes our main experiments. We initialize every refinement loop with the same starting program, so differences in the final pass rate are only a consequence of varying feedback. We observe that the effectiveness of refinement depends on the question's difficulty. We summarize our findings below.

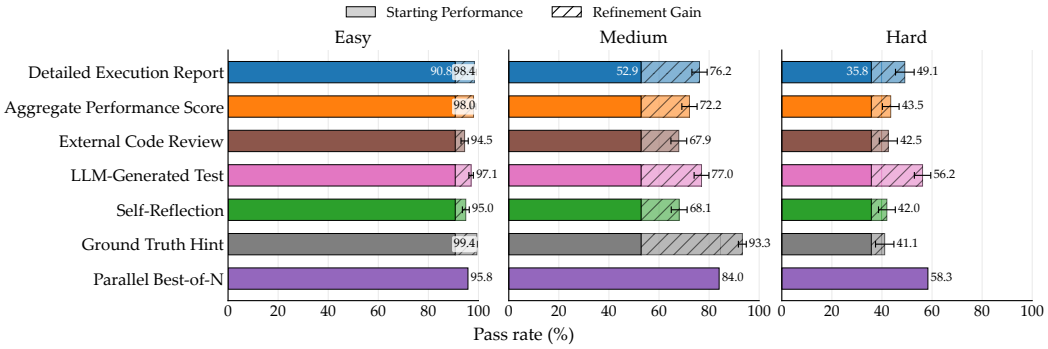


Figure 3: Average pass rate of each feedback type and best-of-N parallel sampling. Results are from `gpt-oss-120b` and are grouped by problem difficulty. The error bars indicate standard error of the mean (SEM), computed across questions and 10 repetitions per question. Each bar is decomposed into the shared initial performance and the improvement achieved through refinement. Although iterative refinement provides meaningful gains on easy and medium problems, its benefits diminish on hard problems. Importantly, best-of-N parallel sampling outperforms every feedback-based refinement method under the same inference budget on hard problems.

Ground-truth hints are ineffective on hard problems. When problems are relatively simple, explicit guidance about the correct strategy allows the model to recover from incorrect initial attempts more effectively than any other feedback type. On easy problems, ground-truth hints achieve almost full performance. On medium problems, it improves the performance from 52.9% to 93.3%, gaining a 40.4% average improvement and exceeding all other feedback types. This result shows that informative guidance can be highly effective when the question is easy relative to the model's capability.

The benefit of ground-truth hints nearly disappears on hard problems, where it produces an average improvement of only 5.3%. Thus, the usefulness of feedback is not only constrained by the information it contains but also depends on the model's ability to interpret that information. In our case, even when the gold algorithmic insight is supplied, the model may lack the capabilities to realize it.

Feedback from LLM-generated test cases performs best on hard problems. Among different types of feedback, failures on LLM-generated test cases perform best on hard problems, improving over the initial attempts by 20.4%. Concrete failure cases can therefore help the model identify and repair some local errors even when richer semantic guidance is ineffective.

Additionally, the results show that across all difficulties, LLM-generated tests achieve performance comparable to that of the predefined test cases in the detailed execution report. This finding suggests that LLM-generated tests can provide useful diagnostic signals even without external human input. More broadly, these results suggest that the gains from execution feedback stem from access to an environment in which the model can test and refine its programs, rather than merely from exploiting access to the predefined evaluation cases.

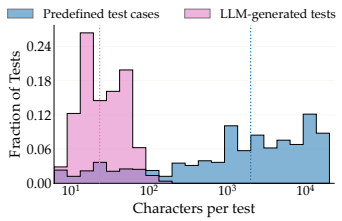


Figure 4: Comparing the length of test cases written by LLM with those included in the predefined test set. LLM-generated test cases are notably shorter.

To provide further insights, we visualize the lengths of test cases generated by the LLM alongside those in the predefined set. Figure 4 shows that test cases generated by LLMs are much shorter than the pre-defined test cases, not exceeding 100 characters in most cases. One explanation is that these shorter test cases use up less context, and hence are easier to interpret. This suggests that the model itself has the potential to identify more effective diagnostics as compared to human-defined signals. We leave further investigation of this hypothesis to future work.

Aggregate scores, external reviews, and self-reflection add little value. While aggregate performance scores, external code reviews, and self-reflection show moderate performance on medium

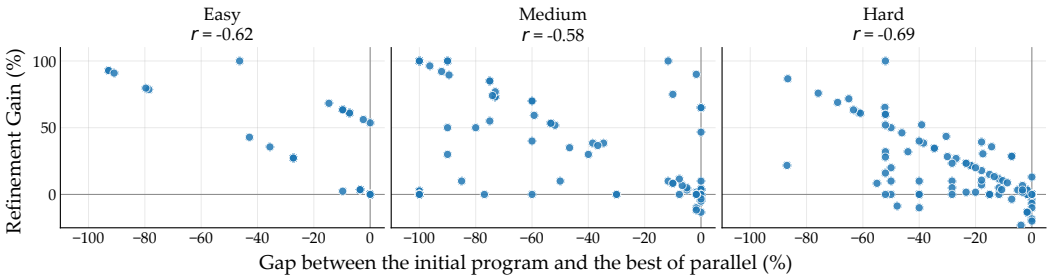


Figure 5: Relationship between refinement gain and the performance gap between the initial program and best-of-N parallel sampling, when the detailed execution report is used as feedback. Each point corresponds to a problem instance and a repetition trial. The downward trend shows that as the quality of the initial program gets closer to the best of parallel, the refinement gain possible through iterative refinement shrinks. Pearson's r is reported in the figure.

or easy questions, they yield only modest improvements on hard problems. These feedback signals provide little reliable information that is not already available to the original model. This questions the effectiveness of these feedback types for discovery on hard problems (Zhang et al., 2026b; Madaan et al., 2023; Shinn et al., 2023; Hong et al., 2024; Qian et al., 2024).

4.2 Parallel sampling outperforms any iterative refinement on hard problems

Figure 3 shows that multiple feedback types in easy and medium questions can outperform best-of-N parallel sampling under the same inference budget. For instance, ground-truth hints in both cases outperform parallel sampling. However, best-of-N parallel sampling achieves the highest performance in hard problems. This suggests that a diverse set of feedback commonly used in practice fails to help the model move beyond the strongest solutions found through independent sampling.

To better understand this behavior, we investigate the relation between improvement through detailed execution feedback and the gap between the initial program and the best-of-N parallel sample in Figure 5. The strong negative relationship shows that refinement produces its largest gains when the initial attempt is far below the model's best parallel sample. As the initial solution approaches the best parallel sample, the achievable refinement gain rapidly diminishes. This indicates that sequential refinement primarily recovers performance already accessible through independent sampling on difficult questions. Therefore, harnesses commonly designed for routine coding tasks based on iterative refinement may not automatically generalize to making discoveries in harder problems beyond the model's existing capabilities.

4.3 Models are not making effective progress through feedback

Does the iterative refinement process show meaningful progress? Figure 6 shows the number of iterations needed until the model fully solves a problem from each difficulty using detailed execution report. Notably, almost all easy questions are solved within the first two iterations. For medium questions, however, the model continues to gain from iterative refinement also in later iterations. For hard questions, the model can hardly solve them within the given budget, with 77% of trajectories remaining unsolved after 10 rounds of refinement. A natural question that arises here is whether models are progressing and need more time to fully solve the problem, or are just performing random explorations in the space. We address this question in multiple ways.

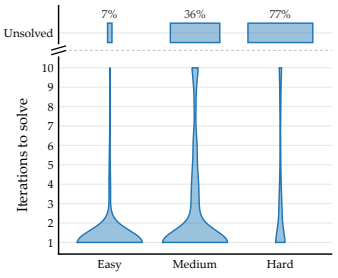


Figure 6: Iterations needed for the model to pass all test cases for the problem with the detailed execution report for each level of difficulty.

Figure 7 examines two representative hard problems over an extended horizon of 50 refinement rounds. Rather than improving monotonically as feedback accumulates, performance fluctuates

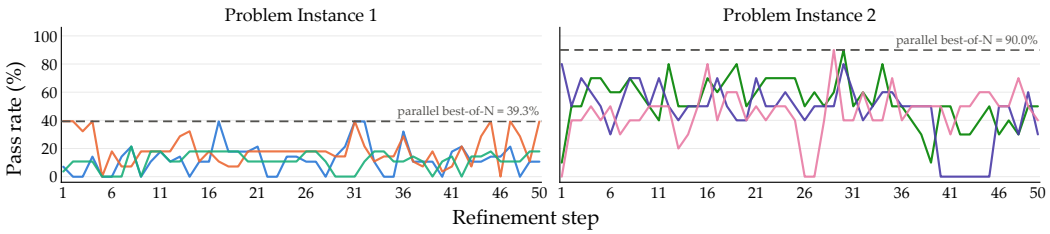


Figure 7: Demonstrating the performance of three randomly sampled trajectories with detailed execution feedback for two hard problems. The loop is extended to $N = 50$ rounds. We observe that performance fluctuates substantially and only occasionally reaches the best score found through parallel sampling. The trajectories resemble stochastic retries bounded by the parallel-sampling frontier rather than steady, cumulative improvement.

across rounds: revisions frequently discard previous progress, revisit earlier errors, or replace one failure with another. The trajectories occasionally reach the best performance observed through parallel sampling but do not reliably exceed it. In fact, the behavior resembles stochastic resampling rather than a cumulative improvement, in which successive feedback would be expected to produce sustained progress toward a solution.

To complement the performance trajectories, we examine the generated programs in a shared embedding space. We use `Qwen3-Embedding-8B`, an embedding model designed for semantic similarity tasks, commonly used in code retrieval (Zhang et al., 2025). For each problem, we embed all programs generated by the evaluated methods, together with the human-written reference solution.

Figure 8 presents a t-SNE projection for a hard problem, allowing us to qualitatively compare the refinement trajectories with independently sampled programs and the reference solution. Across rounds, the trajectories frequently shift between regions near different parallel samples, with little consistent movement toward the reference solution. This pattern complements the fluctuating pass rates in Figure 7 and supports the claim that refinement explores alternative solutions rather than accumulating improvements.

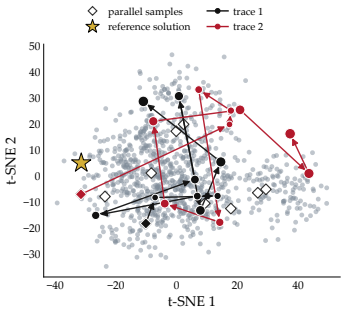


Figure 8: A t-SNE visualization of program embeddings for a hard problem, showing refinement trajectories with detailed execution report alongside independently sampled programs and the human-written reference solution. The trajectories frequently jump between regions near different parallel samples, with no consistent movement toward the reference solution.

To go beyond qualitative demonstrations, we finally present the quantitative measurements in the embedding space. In particular, we measure the cosine distance between the generated program at every iterative step and the nearest solution in our inventory to measure if the model is making meaningful progress toward a solution. Figure 9 compares averaged trajectories for unsolved runs on hard problems and solved runs on medium problems. For unsolved hard-problem trajectories, distances fluctuate around an approximately constant mean, indicating no consistent movement toward the solutions. In solved trajectories, however, we observe a constant decay toward a solution. Together, these results suggest that, on the unsolved hard problems examined, the agent explores alternative programs without consistently accumulating progress toward a correct solution.

5 Conclusion

We presented a controlled study of how different feedback signals affect iterative test-time refinement in code generation. Our results show that the value of feedback depends strongly on problem difficulty. We additionally showed that for hard problems, best-of-N parallel sampling outperforms all feedback-based refinements under the same generation budget. In fact, iterative interaction can

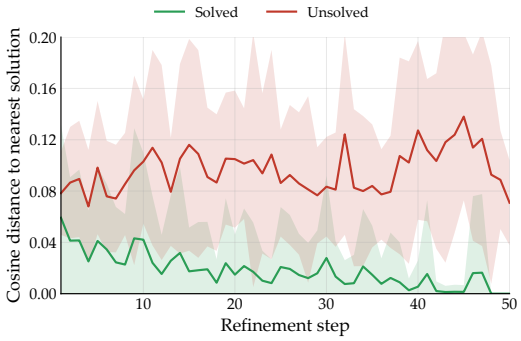


Figure 9: Average cosine distance between the program generated at each refinement step and the nearest solution in our inventory. Shades present the standard deviations. Distances fluctuate around an approximately constant mean for unsolved runs on hard problems but decrease steadily for solved runs on medium problems, suggesting that the former lack consistent progress toward known solutions in the embedding space.

help a model recover solutions already accessible within its existing capabilities, but does not necessarily enable it to discover solutions beyond them.

Future work should investigate how to train models that can effectively interpret, retain, and act on feedback across multiple iterations. In particular, models may require mechanisms for accumulating insights, preserving progress, and revising their underlying strategies rather than merely resampling solutions conditioned on recent failures. Developing such capabilities may allow iterative refinement to move beyond recovering existing knowledge and become a genuine mechanism for discovery.

6 AI Disclosure

In this work, we used generative AI tools for polishing writing, implementing methods, and visualization. We have not used generative AI tools for designing or providing feedback on research methodology or experiments, supporting qualitative and thematic data analysis, and interpreting results. Generating synthetic data sets, developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, assisting with translation, cleaning, and reformatting the dataset are not applicable to this work. Additionally, we used generative AI tools to polish writing, to help with data visualization, but not for ideation or structural framing. We have reviewed all AI-assisted work. We verified the generated AI code and results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

References

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Xiangming Gu, Soham De, Larisa Markeeva, Petar Veličković, and Razvan Pascanu. Understanding performance gap between parallel and sequential sampling in large reasoning models. *arXiv preprint arXiv:2604.05868*, 2026.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, zili wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Transactions of the Association for Computational Linguistics*, 12:1417–1440, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In *EMNLP (Findings)*, pp. 15964–15978, 2025.

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

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914–919, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20275–20321, November 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*, volume 2024, pp. 36545–36593, 2024.

Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, Olivier Pietquin, and Laura Toni. A survey of temporal credit assignment in deep reinforcement learning. *arXiv preprint arXiv:2312.01072*, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

Chenglei Si, Zitong Yang, Yejin Choi, Emmanuel Candès, Diyi Yang, and Tatsunori Hashimoto. Towards execution-grounded automated ai research. *arXiv preprint arXiv:2601.14525*, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

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

Zongqian Wu, Tianyu Li, Baoduo Xu, Jiaying Yang, Mengmeng Zhan, Xiaofeng Zhu, and Lei Feng. Is depth all you need? an exploration of iterative reasoning in llms. *arXiv preprint arXiv:2502.10858*, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. What makes large language models reason in (multi-turn) code generation? In *International Conference on Learning Representations*, volume 2025, pp. 40144–40181, 2025a.

Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, Peiyao Sheng, Zixuan Wang, Wenhao Chai, Aleksandra Korolova, Peter Henderson, Sanjeev Arora, Pramod Viswanath, Jingbo Shang, and Saining Xie. Livecodebench pro: How do olympiad medalists judge llms in competitive programming? In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, 2025b.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870, 2024.

A Further Experimental Details

We used the default model-specific sampling configuration for each model. For `gpt-oss` models, we used the `medium` reasoning budget, since the model will never stop thinking on hard questions if extended thinking is used. Except `gpt-5.4-nano`, all models were deployed via vLLM (Kwon et al., 2023) without quantization.

Codeforces problems in our dataset were originally specified for C++. However, we instructed the models to produce solutions in Python 3 for consistency and ease of execution, while relaxing time and space constraints. We evaluated each generated program in a sandboxed environment that provided access only to the problem input through standard input and required the program to return its answer through standard output.

A.1 Prompts

In this section, we present the prompts used in our experiments. The following two prompt templates are used primarily for the main agent. The {Problem Block} placeholder denotes the complete problem statement. Each problem also includes a set of sample input–output pairs, which are provided alongside the problem statement in the {Problem Block} to clarify the expected input and output formats. These examples are included only as part of the prompt and are distinct from the test cases used for evaluation.

Prompt used for the main agent: first iteration

```
You are solving a coding problem.

=== PROBLEM ===
{Problem Block}
===============

CONTRACT:
- Your output is executed AS-IS as a Python 3 script.
- The script MUST read all input from ``sys.stdin`` (or ``input()``)
and write its answer to ``sys.stdout`` (or via ``print``).
- The script must be a complete, runnable program (no unresolved
imports). Use only the Python standard library.
- Output must exactly match the expected output up to trailing
whitespace per line (the grader normalizes line endings). Do not print
 any extra prompts, banners, or debug commentary.

OUTPUT: emit the Python 3 source of a complete stdin/stdout program
that solves the problem. Wrap the code in a single ``python`` fenced
block; the grader extracts the last fenced block as the candidate. Do
NOT print any commentary outside the fenced block.
```

Prompt used for the main agent: subsequent iterations

```
You are solving a coding problem.

=== PROBLEM ===
{Problem Block}
===============

CONTRACT:
- Your output is executed AS-IS as a Python 3 script.
- The script MUST read all input from ``sys.stdin`` (or ``input()``)
and write its answer to ``sys.stdout`` (or via ``print``).
- The script must be a complete, runnable program (no unresolved
imports). Use only the Python standard library.
```

```
- Output must exactly match the expected output up to trailing
whitespace per line (the grader normalizes line endings). Do not print
 any extra prompts, banners, or debug commentary.

In your previous attempt, you have produced the following Python
program:

=== PRIOR ATTEMPT ===
{Code Block}
====================

{Feedback Block}

OUTPUT: emit the Python 3 source of a complete stdin/stdout program
that solves the problem. Wrap the code in a single ``python`` fenced
block; the grader extracts the last fenced block as the candidate. Do
NOT print any commentary outside the fenced block.
```

The {Code Block} holds a Python 3 program that the model has generated in the previous round. The {Feedback Block} contains the feedback produced by evaluating that program under the corresponding feedback condition. For example, under the detailed execution report feedback, the model sees:

Feedback block: detailed execution report

```
The prior attempt above was tested on the example inputs. It failed on
 the following cases -- use them to guide the next attempt:

=== FAILED CASE ===
INPUT: {Input}

EXPECTED OUTPUT: {Expected Output}

ACTUAL OUTPUT: {Actual Output}

ERROR: {Error Type}
============================
```

In the self-reflection feedback condition, we simply use "Write a new program that you believe will improve performance." as the {Feedback Block}, noting that it is a fixed instruction independent of the previous program.

A.2 Prompt for LLM Reviewer

We use the following prompt in a fresh context as the external LLM code reviewer:

Prompt used for the external LLM reviewer

```
Consider the problem below:

=== PROBLEM ===
{Problem Block}
===============

Now consider the following attempt to solve the problem:

=== CANDIDATE SOLUTION ===
{Code Block}
```

```
==========================

Your job is to provide a JSON output covering the following items:

- correctness: Whether the solution's core algorithm actually solves
the stated problem on the intended input space.
- prescription: A piece of actionable advice for improving the code.

For each item above, write one short paragraph (1-3 sentences) of
concrete, actionable advice grounded in this specific problem and this
 specific solution.

Return ONLY a single JSON object matching this schema -- no markdown
fences, no commentary outside the object:

{
  "Output": {
    "correctness": "<one short paragraph of prescription for
correctness>",
    "prescription": "<one short paragraph of prescription for
prescription>"
  },
  "reasoning": "<2-4 sentence justification>"
}
```

A.3 Prompt for Generating Synthetic Test Cases

We use the following prompt in a fresh context to derive LLM-generated test cases. We sample from the model 20 times to generate 20 synthetic test cases.

Prompt used for test case generation

```
Consider the problem below:

=== PROBLEM ===
{Problem Block}
===============

Produce one valid test case for this problem. Compute the expected
stdout yourself with care. Return only a single JSON object of the
shape below (no markdown fences, no commentary outside the object):

{{"input": "<stdin text>", "expected_output": "<stdout text>"}}

Both ``input`` and ``expected_output`` MUST be plain strings (embed
line breaks as ``\\n``).
```

B Further Experiments

In this section, we provide additional experiments.

B.1 Additional Models

Figures 10, 11, 12, and 13 present experiments with additional models at different scales. While base performance across models varies by family and scale, the observed phenomena confirm our conclusions in the main text.

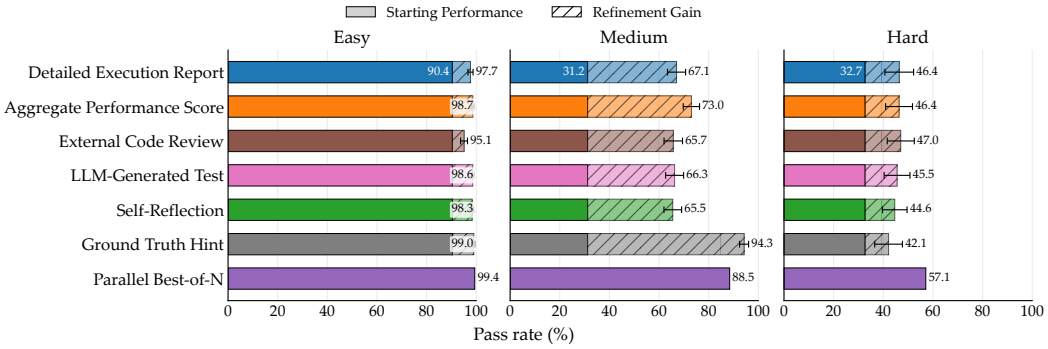


Figure 10: Pass rates for `gpt-oss-20b` with different feedback types. Bars show the mean, and error bars indicate ± 1 standard error of the mean (SEM).

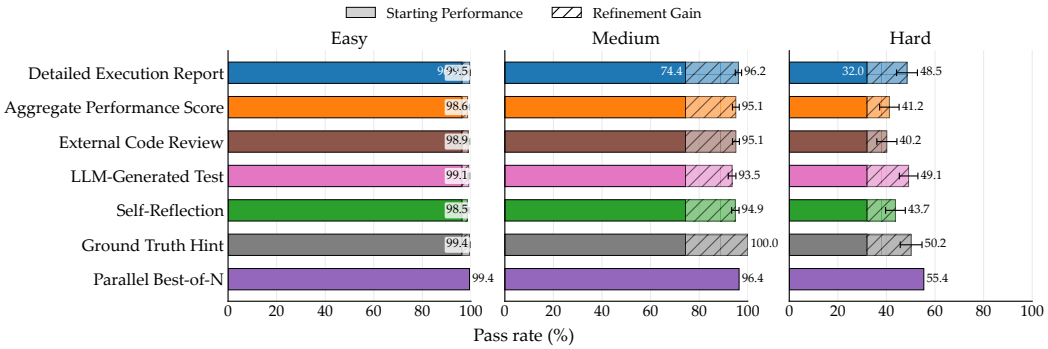


Figure 11: Pass rates for `gpt-5.4-nano` with different feedback types. Bars show the mean, and error bars indicate ± 1 standard error of the mean (SEM).

B.2 Maximum Performance Across Refinement Rounds

In the main experiments, we report the performance of the program generated in the final refinement round. This choice evaluates whether improvements are retained throughout the refinement process and reflects the program that an agent would return after exhausting its inference budget. However, refinement trajectories are not necessarily monotonic: a model may produce a strong solution in an intermediate round and subsequently degrade it.

We therefore conduct a complementary analysis in which each run is scored using the best program generated across all refinement rounds. This evaluation favors iterative refinement by assuming access to an oracle capable of selecting the best intermediate program using the complete evaluation test suite. Best-of-N parallel sampling is evaluated under the same selection rule.

Figure 14 presents the resulting performance. Selecting the best intermediate program improves the measured performance of feedback-based methods because it avoids penalizing regressions in later rounds. Nevertheless, the overall conclusions remain unchanged. The relative effectiveness of feedback continues to depend strongly on problem difficulty, detailed execution reports remain the most useful feedback on hard problems, and best-of-N parallel sampling continues to outperform feedback-based refinement under the same generation budget.

These results further demonstrate that the limitation of sequential refinement cannot be explained solely by the model failing to preserve its best solution. Even when an oracle selects the strongest program encountered anywhere along the trajectory, iterative refinement does not reliably surpass the solution frontier reached through independent parallel sampling.

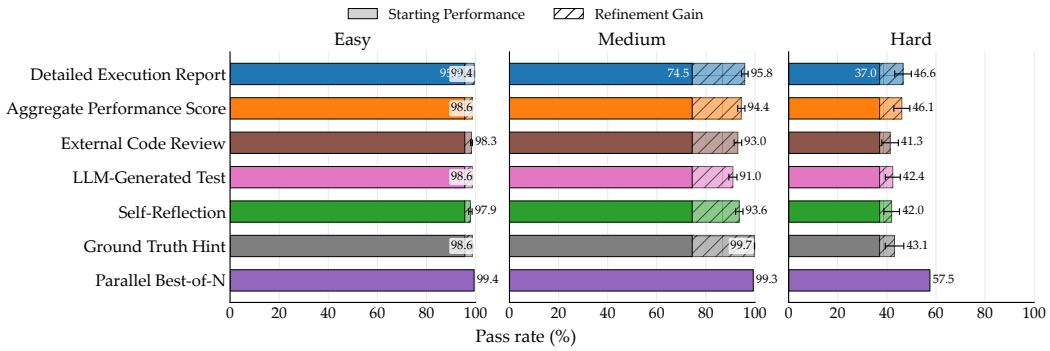


Figure 12: Pass rates for `gemma-4-31B` with different feedback types. Bars show the mean, and error bars indicate ± 1 standard error of the mean (SEM).

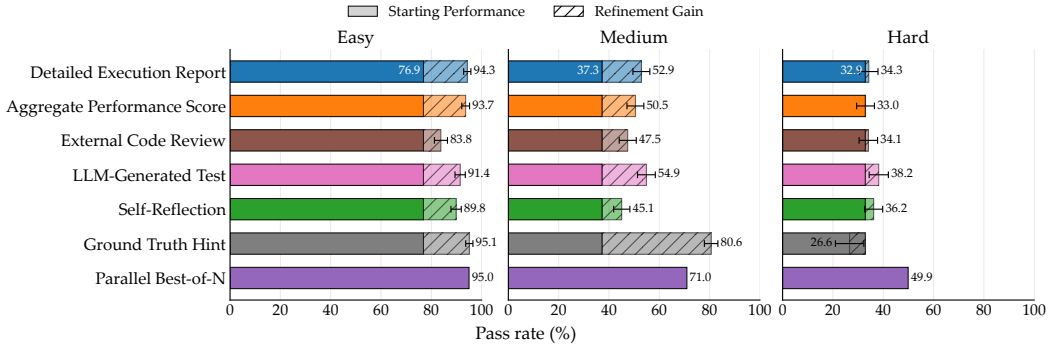


Figure 13: Pass rates for `gemma-4-E4B-it` with different feedback types. Bars show the mean, and error bars indicate ± 1 standard error of the mean (SEM).

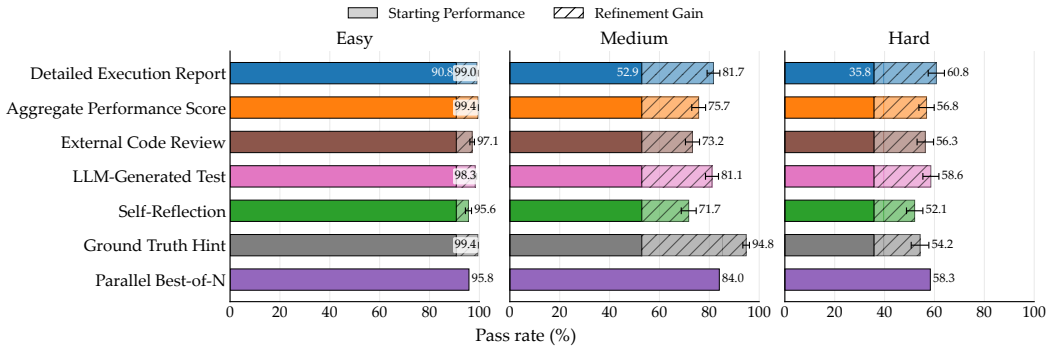


Figure 14: Average maximum pass rate for `gpt-oss-120b` achieved across all refinement rounds under each feedback condition, compared with best-of-N parallel sampling. Unlike the main analysis, which evaluates the final program, this analysis selects the highest-scoring program encountered during each refinement trajectory.