GENERATE-FEEDBACK-REFINE: HOW MUCH DOES MODEL QUALITY IN EACH ROLE MATTER?

Xiang Pan[*†], **Jason Phang**[*†], **Guy Davidson**[*†], **Ethan Perez**[‡]

ABSTRACT

From early in grade school, people learn from explicit feedback provided in response to assignments or other interactions. In this work, we explore how effectively language models incorporate textual feedback, focusing on exploring the utility of having weaker models feedback stronger ones, a potential pathway to scalable oversight. Using code generation as a test domain, we experimentally investigate a generate-feedback-refine process, varying model strengths for generation, feedback, and refinement across the MBPP, APPS, and DS-1000 datasets. We find that weaker models can provide feedback as effectively as stronger models in some cases. Feedback-and-refinement consistently improves performance on APPS and DS-1000, while on MBPP, feedback mainly benefits weaker generation models, underscoring differences across tasks.

1 INTRODUCTION

As models continue to improve in capability, they approach or match human performance in many domains. In order for models to keep improving, we require methods that can allow models to learn from human data or inputs but ultimately surpass the ability of their human trainers. One proposed approach is scalable oversight (Bowman et al., 2022), where humans continue to provide a helpful learning or supervision signal even to models with superhuman capabilities. This allows humans to continue to play a crucial role in providing feedback and steering for systems more capable than themselves, either to improve performance or to avoid dangerous and undesirable model behavior.

To study scalable oversight in a tightly controlled experimental setup, we consider a generate-feedback-refine setup, where we can simulate a more capable model and a less capable human by having a weaker model serve as a proxy for the human providing feedback. In this work, we study the generate-feedback-refine loop already established in the literature for code generation, where we have one model generate an initial code solution, a second model provide feedback on the solution, and return to the first model to refine the original solution based on the feedback. To investigate the impact of model capability in each of the three steps on the overall performance, we run a large set of empirical experiments varying the model used in each of the "roles". We focus on varying the relative strength of the feedback and generation models, where weaker feedback models serve as proxy for humans in the scalable oversight setting, while stronger feedback models offer potential cost savings (compared to using a stronger model for all three steps). We explore this setup across three diverse code generation datasets–MBPP, APPS, and DS-1000.

We find that weaker models can provide effective feedback, at times matching or exceeding more performant models. Our experiments demonstrate that incorporating a feedback-and-refinement loop consistently enhances code generation performance. In the APPS and DS-1000 datasets, even weaker feedback models provide significant improvements when paired with stronger generation models. On the MBPP dataset, where models' initial generations are much more often correct, feedback seems less valuable and more beneficial to weaker generation models. While preliminary, this evidence suggests that the benefits of language feedback are seen primarily when there's more

[*]Equal contribution
[†]New York University
[‡]Anthropic

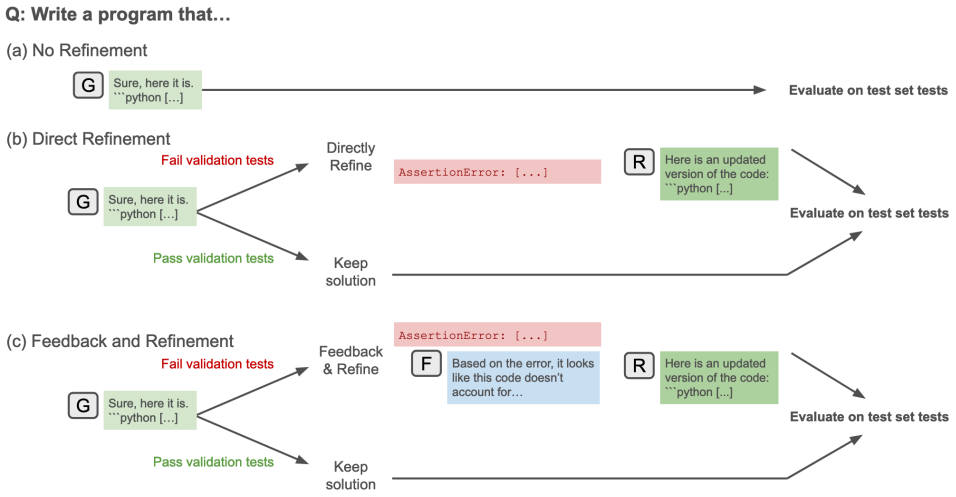


Figure 1: Experimental setup overview. Given a programming task, we prompt a generation model (G) to generate an initial solution. In the *No Refinement* setup, we evaluate the solution directly on a test set of held-out code tests (not presented to the model). In the *Direct Refinement* setup, we run the initial solution on a validation set of held-out tests. If any of these tests fail, we prompt the refinement (R) model to generate a new solution. In the *Feedback and Refinement* setup, we first prompt a feedback model (F) to provide feedback on the initial solution, which we include in the prompt for refinement.

room for improvement—which, if the results hold, offers encouraging potential to scalable oversight in difficult task settings.

2 METHODOLOGY

Figure 1 summarizes our setup. We first present a language model (denoted the *generation model*, (G)) with a coding problem and obtain an **initial solution**. We then evaluate the solution on a set of preliminary test cases. If the initial solution passes all preliminary tests, we take it as the model's solution for the problem; if it fails one or more of the preliminary test cases, we proceed to the feedback and refinement steps. To obtain model **feedback**, we prompt a language model (denoted the *feedback model* (F)) with the problem, the initial solution, and one of the failed test cases and resulting error. We ask the feedback model to provide 1-3 sentences of feedback to correct the code. We then prompt a language model (the *refinement model* (R)) with the problem, the initial solution, and the model-generated feedback to sample a **refined** solution. We then evaluate the solution (either the refined one or, if all preliminary tests passed, the initial one) on a held-out set of tests that are distinct from the preliminary tests. We also compare to a control case where the generation model is prompted to refine its solution after failing a test, but without feedback from another model.

Crucially, we vary which models fulfill the roles of the generation, feedback, and refinement models. We focus our analysis on the case where the same model is used for generation and refinement (as both generate code) and vary which model provides feedback. This allows us to explore feedback from stronger to weaker models, and vice versa, and to evaluate the impact of using models from the same family compared to different model families.

2.1 DATASETS

APPS (Automated Programming Progress Standard) APPS (Hendrycks et al., 2021) is a dataset of Python coding problems. The inputs and outputs for each problem are generally presented in plain text. Each problem comes with a set of test cases, with a subset also having accompanying ground-truth solutions. The problems are grouped into *introductory*, *interview*, and *competition* categories of increasing difficulty.

DS-1000 (Data Science 1000) (Lai et al., 2022) is a code generation benchmark related to data science tasks, with a focus on using Python libraries to manipulate and analyze data. The inputs are StackOverflow questions and intended results, and the outputs are code snippets that answer the question but could be generalized to other similar queries. The libraries are `NumPy`, `Scipy`, `Pandas`, `TensorFlow`, `PyTorch`, `Scikit-learn`, and `Matplotlib`

MBPP (Mostly Basic Python Problems) (Austin et al., 2021) is a benchmark dataset designed for evaluating the performance of code-generation models on simple Python programming tasks. The dataset consists of 974 problems, each consisting of a problem statement, input-output examples, and reference solutions. We use the EvalPlus (Liu et al., 2023) version of MBPP, which provides additional tests beyond the original dataset.

2.2 MODELS

We evaluate our methods with Claude-1-instant (C1i) Anthropic (2024), Claude-2 (C2) Anthropic (2024), GPT 3.5 Turbo (G3.5t) OpenAI et al. (2024), GPT 4 (G4) OpenAI et al. (2024). As discussed above, we fix a single generation and refinement model and vary the feedback models.[1]

2.3 SYNTHETIC TEST GENERATION FOR APPS

Prior work on using automated coding feedback or refinement has highlighted a common methodological issue with using the same test for generating model feedback and evaluating the final model solution (Pan et al., 2023; Huang et al., 2024). Specifically, this introduces a form of selection bias that implicitly raises the scores of refined solutions relative to initial solutions, as refined solutions are conditioned on the initial solutions failing the tests.

One approach to avoiding this issue is having separate sets of preliminary and held-out tests, as described in Section 2. However, the APPS dataset does not include sufficient tests per problem to allow for preliminary/held-out splits. To address this issue, we propose using synthetically generated test cases as the preliminary tests and using the original test cases provided in each dataset as the held-out tests.

To synthetically sample the preliminary test cases for each problem in the dataset, we provide GPT-4 with the problem specifications and few-shot examples of the ground-truth tests. We then prompt it to generate five more examples of similar tests. We repeat this process 20 times, resulting in 100 candidate tests per problem. Next, we filter out tests that exactly match any of the ground-truth tests. We also filter out tests that the ground-truth solutions provided in each dataset fail to pass. We now have an additional set of valid tests for each problem. We exclude problems with fewer than five valid generated tests or without ground-truth solutions from our experiments.

We wish to emphasize two points: (1) This synthetic test generation should not be treated as a part of the code refinement process since it uses the ground truth tests as inputs. Instead, it serves as a data-augmentation method, providing a second set of tests to conduct our code refinement experiments. (2) There may still be indirect information leakage between the ground truth tests and the synthetically generated tests (e.g., if the tests are very similar and probe the same edge cases of the coding problem). Our results on APPS should be interpreted with this in mind, but we find that this approach is preferable to not having held-out tests or to excluding problems without sufficient tests to hold some out.

3 RESULTS

APPS In Figure 2, we show the scores on the held-out test cases on the APPS dataset, where we use the same model for both generation and refinement and vary the feedback model across the columns. We offer the complete set of results in Table 3 to Table 6.

We find that for code generation, performance generally follows the order of GPT-4 $>$ GPT-3.5 Turbo $>$ Claude-2 $>$ Claude-Instant-V1. Scoring the initial solutions without any refinement consistently

[1]We show the complete list and examples with different generation and refinement models in the Appendix A and Appendix B

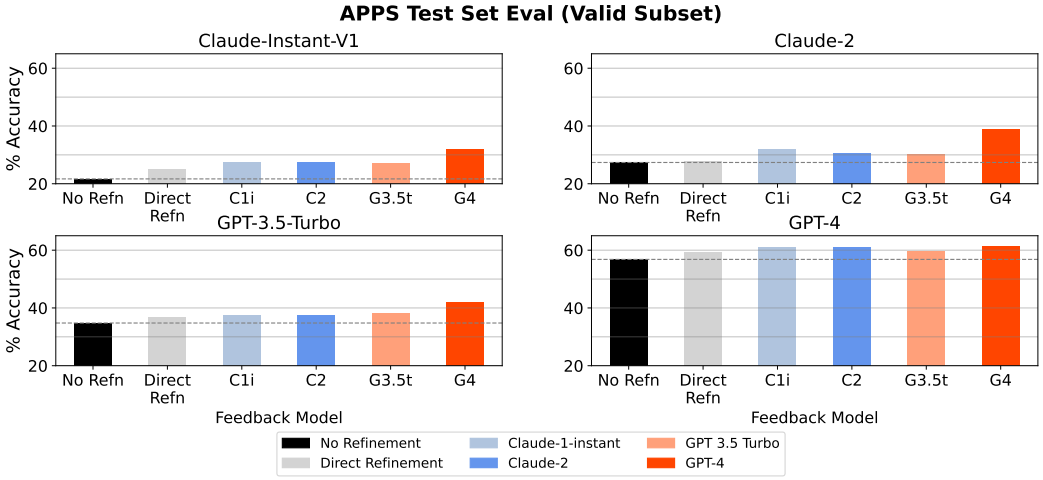


Figure 2: **APPS dataset results**, averaging across problems in all three subsets. Each plot shows the results on APPS using each of the four models as for both code generation and refinement, while the columns within each plot vary in the feedback model used (or lack thereof). We include a no refinement baseline (*No Refn*), where the initial generated solution is scored directly. Across all four generation models, model-generated feedback leads to improved solutions, with stronger feedback models leading to greater improvement.

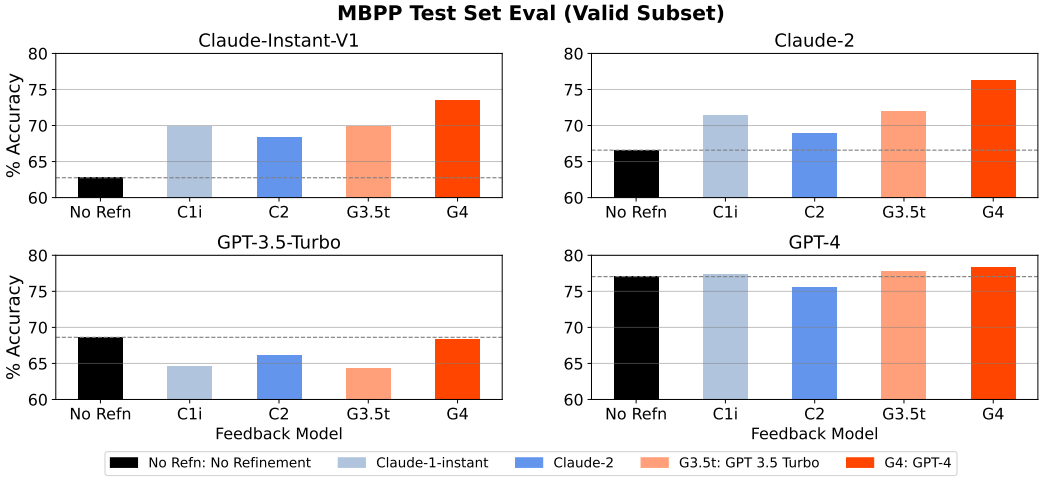


Figure 3: **MBPP dataset results**. Weak feedback models provide equally effective feedback to the stronger models, especially when the generation model is strong. Note that the Y-axis begins at 60%.

leads to the worst performance. We find that incorporating a feedback-and-refinement step leads to improved performance across all generation and feedback models. Broadly, using stronger models, either for generation or for feedback, leads to better performance. Notably, even feedback from a weaker model for a strong model (e.g., generating a solution with GPT-4, using Claude-2 to generate feedback and GPT-4 to generate a refinement) leads to improved performance over the initial solution. We also include an additional "direct refinement" baseline, where we task the refinement model to directly refine the original solution, using the test errors but without an additional feedback step. This baseline underperforms an explicit feedback step, even when a weaker model is used to provide feedback.

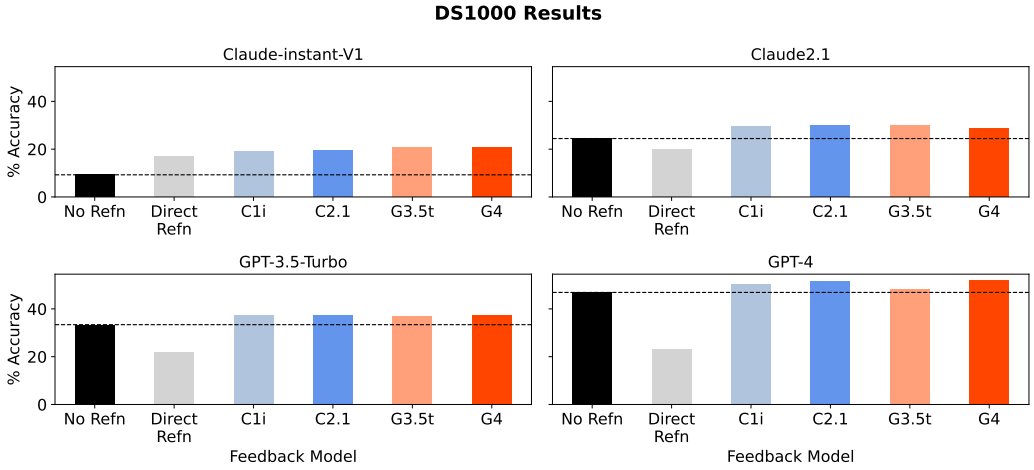


Figure 4: **DS-1000 dataset results**. Each plot shows the results on DS-1000 using each of the four models for code generation and refinement, while the columns within each plot vary in the feedback model used. *Init* refers to the initial solution generated by the generation model, *Direct Refinement* refers to the refinement model directly refining the initial solution without feedback, and *C2.1*, *G3.5t*, *G4*, and *C1.2i* refer to the feedback models generating feedback for the refinement model. The feedback-and-refinement process leads to improved performance over the baselines.

MBPP Figure 3 depicts the EvalPlus MBPP dataset results using the same experimental setup as above in (and see the result full in Table 7).

Compared to APPS, the baseline performance of all models is substantially higher. This provides less room for improvement. Unsurprisingly, we observe that weaker models benefit more from feedback and that our most capable model (GPT-4) offers the most useful feedback. We otherwise find the weaker models similarly capable at providing feedback, though we also observe a case win which feedback is harmful (compared to the feedback-less baseline), rather than helpful (GPT-3.5-Turbo).

DS-1000 We present the results on the DS-1000 dataset in Figure 4.

Our results on DS-1000 resemble our results on APPS. Across all cases, providing feedback to the models helps improve performance over the no-refinement baseline. Furthermore, all models provide similarly useful feedback, where the model with the weakest overall accuracy (Claude-1-instant) providing feedback that in some cases is similarly helpful to the strongest model (GPT-4).

A potential pitfall is that the wrong feedback may lead to worse performance rather than better results. To evaluate the quality of the feedback generated by the feedback models, we compare the *feedback direction accuracy* of the feedback models to the evaluation results of the refinement model. Feedback Direction Accuracy reflects how accurately the feedback model assesses the correctness of the generated code. Feedback direction is correct if the feedback predicts the solution is accurate and the solution passes all the tests. Otherwise, the feedback model should point out that the code is incorrect and describe possible errors. We calculate the feedback direction accuracy as the percentage of the feedback instances that agree with the evaluation results of the refinement model in the full evaluation.

$$\text{Feedback Direction Accuracy} = \frac{\text{\# of Correct Feedback Directions}}{\text{\# of Feedback}} \tag{1}$$

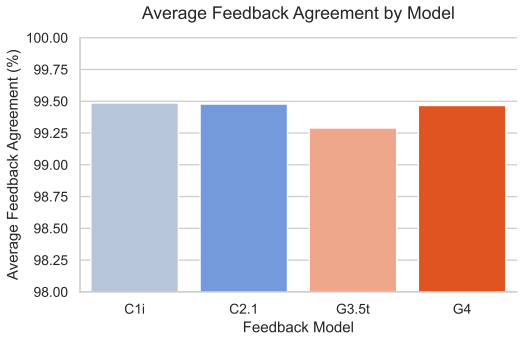


Figure 5: Averaged Feedback Direction Accuracy on the DS-1000 dataset.

We show the feedback direction accuracy in Equation 3. We find that the feedback direction accuracy is generally high, with the feedback models generally providing feedback that agrees with the evaluation results of the refinement model.

4 Related Work

4.1 Code Generation

Code has been a popular target for language models, with early work focusing on generating code snippets from natural language descriptions (Feng et al., 2020; Chen et al., 2021). Recent studies have shown that large language models can generate high-quality code for a wide range of programming tasks, including coding challenges, data science tasks, and software engineering problems. These models have demonstrated the ability to produce code that is both syntactically correct and semantically meaningful, often outperforming traditional methods across various benchmarks. However, LLMs are still imperfect in real-world tasks (Jimenez et al., 2023) and may fail on the first attempt in those code tasks.

4.2 Model Feedback for Code Refinement

Consequently, Madaan et al. (2024) propose multi-step generation and self-refinement approaches to enhance performance by leveraging signals from humans, the compiler, or the models themselves. In recent research, Kim et al. (2024) addresses the challenge of code generation by using large language models (LLMs) to create a plan composed of actions for interacting with a computer interface, applying Recursive-Critique-and-Improvement (RCI) to both the plan and individual steps. Jiang et al. (2023) propose generating snippets of background knowledge relevant to a given task and incorporating this knowledge into the model's feedback and refinement processes. Kim et al. (2023) utilize both LLM-generated explanations and execution traces as signals for refinement. Additionally, Olausson et al. (2024) identify that refinement performance can be constrained by the capabilities of the feedback model. Ni et al. (2024) focus on training models to reason about execution traces and states to enhance code repair capabilities. In a similar vein, McAleese et al. (2024) train LLMs to write test code for a given problem, while Tian & Chen (2023) incorporate test-case analysis into the feedback-and-refinement process. Furthermore, Ding et al. (2024) highlight the potential of using smaller models in the self-refinement process by learning. In our work, we demonstrate that a weaker model can provide feedback as effectively as a stronger feedback model, particularly when the generation model is robust. This suggests that a weaker model can serve as an effective supervisor in the feedback-and-refinement process.

4.3 Automated Supervision

For real-world complex tasks, feedback from humans is not always feasible. We aim to develop scalable oversight and provide automated signals to help the model Bai et al. (2022). Using LLMs as graders to provide feedback for tuning data Ramji et al. (2024) involves external metrics in the

self-refinement process. Pan et al. (2023) and Bowman et al. (2022) survey the landscape of diverse self-correction strategies and scalable oversight methods.

In this work, we evaluate a feedback-and-refinement methodology that leverages weaker models to provide feedback to stronger models, which could be more scalable and efficient than self-refinement by learning.

4.4 Inference-Time Compute

The feedback-and-refinement procedure for generating code solutions from models can also be seen as a form of test-time compute, which has gained traction as an alternative means of improving model performance besides modifying or scaling up the underlying model. Methods such as best-of-N, rejection sampling, and beam search that use trade off more computation for better performance have a long history of use in the field, while newer approaches focus on scaling inference-time compute beyond sampling more. OpenAI (2024) and DeepSeek-AI et al. (2025) demonstrated that teaching models to reason over chains of thought can lead to significant improvements in the reasoning capabilities of models, setting the state of the art in many current model intelligence benchmarks. Snell et al. (2024) explored the scaling up of inference-time compute using multiple samples and a verifier model, finding that scaling inference-time compute to be more economical than scaling up model sizes.

5 Conclusion

In this work, we evaluate using the feedback-and-refinement methodology for code generation, using a spectrum of weak of models in feedback, refinement and solution generation roles. While we find that stronger models in any of the three roles generally leads to better performance, our results also show that even feedback from weaker models can still improve upon the performance of a stronger generation model. By incorporating weaker models into the feedback loop, we can significantly reduce the computational cost and latency of generating valid solutions, making the feedback-and-refinement process more scalable and accessible for real-world applications.

REFERENCES

Anthropic. Claude 2 model card, 2024. URL https://www-cdn.anthropic.com/bd2a28d2535bfb0494cc8e2a3bf135d2e7523226/Model-Card-Claude-2.pdf. Accessed: 2024-08-08.

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

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Samuel R Bowman, Jeeyoon Hyun, Ethan Perez, Edwin Chen, Craig Pettit, Scott Heiner, Kamilė Lukošiūtė, Amanda Askell, Andy Jones, Anna Chen, et al. Measuring progress on scalable oversight for large language models. *arXiv preprint arXiv:2211.03540*, 2022.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):392–418, 2024.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2024. URL https://arxiv.org/abs/2310.01798.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks, 2023. URL https://arxiv.org/abs/2303.17491.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

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, 2024.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. Llm critics help catch llm bugs. *arXiv preprint arXiv:2407.00215*, 2024.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2024. URL https://arxiv.org/abs/2306.09896.

OpenAI. Learning to reason with llms, 2024.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne

Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. GPT-4 Technical Report, March 2024.

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

Keshav Ramji, Young-Suk Lee, Ramón Fernandez Astudillo, Md Arafat Sultan, Tahira Naseem, Asim Munawar, Radu Florian, and Salim Roukos. Self-refinement of language models from external proxy metrics feedback. *arXiv preprint arXiv:2403.00827*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Zhao Tian and Junjie Chen. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120*, 2023.

A APPS RESULTS

A.1 APPS EXAMPLE

Listing 1: Initial Solution Prompt

```
{Apps Question}

ANSWER:
```

Listing 2: Feedback Prompt

```
{Apps Question}

----Incorrect Python Code----

{Generated Initial Solution}

----Instruction for Feedback----

{Error Message from Initial Solution}

Please helping me debug the above program, which has some errors and is not passing the tests. Please give a
      concise (at most 2-3 sentences) textual explanation of what is wrong with the code. Do *not* generate
      any code, because I want to fix the code myself.
```

Listing 3: Feedback Prompt

```
{Apps Question}

----Incorrect Python Code----

{Generated Initial Solution}

----Feedback----

{Generated Feedback}

----Instruction for Revision----

Based on the above specifications, please revise the incorrect Python code to a correct implementation. Please
       take the above feedback into consideration.
```

G Model	F Model	R Model	Introductory	Interview	Competition	Avg
C1i	-	-	0.389 ± 0.024	0.232 ± 0.003	0.029 ± 0.017	0.217 ± 0.011
C1i	C1i	C1i	0.489 ± 0.013	0.303 ± 0.004	0.029 ± 0.017	0.274 ± 0.007
C1i	C1i	C2	0.532 ± 0.013	0.315 ± 0.008	0.029 ± 0.017	0.292 ± 0.008
C1i	C1i	G3.5t	0.568 ± 0.013	0.340 ± 0.011	0.029 ± 0.017	0.312 ± 0.009
C1i	C1i	G4	0.705 ± 0.023	0.552 ± 0.013	0.386 ± 0.058	0.548 ± 0.026
C1i	C2	C1i	0.505 ± 0.010	0.292 ± 0.006	0.029 ± 0.017	0.275 ± 0.009
C1i	C2	C2	0.547 ± 0.010	0.300 ± 0.005	0.029 ± 0.017	0.292 ± 0.006
C1i	C2	G3.5t	0.526 ± 0.019	0.335 ± 0.005	0.043 ± 0.017	0.302 ± 0.011
C1i	C2	G4	0.711 ± 0.012	0.503 ± 0.009	0.243 ± 0.017	0.485 ± 0.010
C1i	G3.5t	C1i	0.479 ± 0.010	0.308 ± 0.009	0.029 ± 0.017	0.272 ± 0.010
C1i	G3.5t	C2	0.495 ± 0.015	0.314 ± 0.010	0.029 ± 0.017	0.279 ± 0.013
C1i	G3.5t	G3.5t	0.526 ± 0.012	0.331 ± 0.005	0.043 ± 0.029	0.300 ± 0.010
C1i	G3.5t	G4	0.695 ± 0.006	0.505 ± 0.007	0.243 ± 0.029	0.481 ± 0.010
C1i	G4	C1i	0.563 ± 0.011	0.325 ± 0.006	0.071 ± 0.023	0.320 ± 0.007
C1i	G4	C2	0.595 ± 0.016	0.358 ± 0.007	0.114 ± 0.036	0.356 ± 0.013
C1i	G4	G3.5t	0.595 ± 0.021	0.389 ± 0.010	0.100 ± 0.029	0.361 ± 0.013
C1i	G4	G4	0.700 ± 0.018	0.508 ± 0.008	0.214 ± 0.039	0.474 ± 0.014

Table 3: Full APPS results, with Claude-1-Instant as the Generator Model

G Model	F Model	R Model	Introductory	Interview	Competition	Avg
C2	-	-	0.558 ± 0.005	0.263 ± 0.006	0.000 ± 0.000	0.274 ± 0.003
C2	C1i	C1i	0.584 ± 0.010	0.328 ± 0.008	0.000 ± 0.000	0.304 ± 0.005
C2	C1i	C2	0.611 ± 0.019	0.326 ± 0.012	0.029 ± 0.017	0.322 ± 0.011
C2	C1i	G3.5t	0.642 ± 0.013	0.357 ± 0.006	0.029 ± 0.017	0.343 ± 0.008
C2	C1i	G4	0.716 ± 0.015	0.529 ± 0.015	0.286 ± 0.023	0.510 ± 0.014
C2	C2	C1i	0.584 ± 0.005	0.314 ± 0.013	0.000 ± 0.000	0.299 ± 0.006
C2	C2	C2	0.595 ± 0.006	0.309 ± 0.013	0.014 ± 0.014	0.306 ± 0.009
C2	C2	G3.5t	0.611 ± 0.010	0.349 ± 0.012	0.029 ± 0.029	0.329 ± 0.011
C2	C2	G4	0.711 ± 0.017	0.518 ± 0.010	0.243 ± 0.029	0.491 ± 0.015
C2	G3.5t	C1i	0.574 ± 0.010	0.325 ± 0.010	0.000 ± 0.000	0.299 ± 0.006
C2	G3.5t	C2	0.595 ± 0.013	0.314 ± 0.014	0.000 ± 0.000	0.303 ± 0.008
C2	G3.5t	G3.5t	0.600 ± 0.005	0.334 ± 0.012	0.014 ± 0.014	0.316 ± 0.005
C2	G3.5t	G4	0.716 ± 0.021	0.518 ± 0.005	0.257 ± 0.036	0.497 ± 0.017
C2	G4	C1i	0.616 ± 0.006	0.380 ± 0.008	0.100 ± 0.029	0.365 ± 0.011
C2	G4	C2	0.637 ± 0.013	0.389 ± 0.011	0.143 ± 0.032	0.390 ± 0.013
C2	G4	G3.5t	0.658 ± 0.008	0.392 ± 0.012	0.157 ± 0.014	0.402 ± 0.006
C2	G4	G4	0.695 ± 0.024	0.538 ± 0.014	0.286 ± 0.032	0.506 ± 0.019

Table 4: Full APPS results, with Claude-2 as the Generator Model

G Model	F Model	R Model	Introductory	Interview	Competition	Avg
G3.5t	-	-	0.642 ± 0.011	0.358 ± 0.015	0.043 ± 0.017	0.348 ± 0.011
G3.5t	C1i	C1i	0.674 ± 0.018	0.391 ± 0.015	0.043 ± 0.017	0.369 ± 0.012
G3.5t	C1i	C2	0.695 ± 0.013	0.402 ± 0.016	0.057 ± 0.027	0.384 ± 0.013
G3.5t	C1i	G3.5t	0.689 ± 0.021	0.392 ± 0.019	0.043 ± 0.017	0.375 ± 0.015
G3.5t	C1i	G4	0.758 ± 0.010	0.528 ± 0.017	0.300 ± 0.027	0.529 ± 0.012
G3.5t	C2	C1i	0.684 ± 0.012	0.388 ± 0.012	0.057 ± 0.027	0.376 ± 0.012
G3.5t	C2	C2	0.689 ± 0.010	0.402 ± 0.015	0.043 ± 0.017	0.378 ± 0.012
G3.5t	C2	G3.5t	0.695 ± 0.013	0.391 ± 0.015	0.043 ± 0.017	0.376 ± 0.014
G3.5t	C2	G4	0.789 ± 0.019	0.529 ± 0.006	0.329 ± 0.029	0.549 ± 0.014
G3.5t	G3.5t	C1i	0.684 ± 0.019	0.378 ± 0.018	0.043 ± 0.017	0.369 ± 0.014
G3.5t	G3.5t	C2	0.689 ± 0.010	0.385 ± 0.014	0.043 ± 0.017	0.372 ± 0.011
G3.5t	G3.5t	G3.5t	0.695 ± 0.013	0.377 ± 0.020	0.071 ± 0.039	0.381 ± 0.021
G3.5t	G3.5t	G4	0.768 ± 0.010	0.538 ± 0.019	0.214 ± 0.045	0.507 ± 0.020
G3.5t	G4	C1i	0.695 ± 0.011	0.435 ± 0.014	0.071 ± 0.023	0.401 ± 0.012
G3.5t	G4	C2	0.700 ± 0.023	0.443 ± 0.013	0.114 ± 0.036	0.419 ± 0.020
G3.5t	G4	G3.5t	0.716 ± 0.015	0.432 ± 0.011	0.114 ± 0.036	0.421 ± 0.018
G3.5t	G4	G4	0.795 ± 0.010	0.558 ± 0.012	0.300 ± 0.052	0.551 ± 0.020

Table 5: Full APPS results, with GPT-3.5 Turbo as the Generator Model

G Model	F Model	R Model	Introductory	Interview	Competition	Avg
G4	-	-	0.837 ± 0.015	0.525 ± 0.013	0.343 ± 0.027	0.568 ± 0.010
G4	C1i	C1i	0.853 ± 0.013	0.555 ± 0.014	0.343 ± 0.027	0.584 ± 0.008
G4	C1i	C2	0.847 ± 0.015	0.540 ± 0.011	0.343 ± 0.027	0.577 ± 0.010
G4	C1i	G3.5t	0.858 ± 0.013	0.540 ± 0.015	0.357 ± 0.023	0.585 ± 0.005
G4	C1i	G4	0.868 ± 0.014	0.566 ± 0.013	0.400 ± 0.017	0.612 ± 0.010
G4	C2	C1i	0.847 ± 0.019	0.549 ± 0.013	0.357 ± 0.023	0.585 ± 0.012
G4	C2	C2	0.847 ± 0.013	0.545 ± 0.014	0.371 ± 0.014	0.588 ± 0.006
G4	C2	G3.5t	0.863 ± 0.015	0.542 ± 0.015	0.343 ± 0.027	0.583 ± 0.011
G4	C2	G4	0.895 ± 0.017	0.562 ± 0.010	0.371 ± 0.027	0.609 ± 0.011
G4	G3.5t	C1i	0.853 ± 0.013	0.534 ± 0.010	0.357 ± 0.023	0.581 ± 0.008
G4	G3.5t	C2	0.858 ± 0.013	0.532 ± 0.012	0.343 ± 0.027	0.578 ± 0.009
G4	G3.5t	G3.5t	0.858 ± 0.013	0.534 ± 0.010	0.357 ± 0.023	0.583 ± 0.008
G4	G3.5t	G4	0.874 ± 0.013	0.549 ± 0.010	0.371 ± 0.027	0.598 ± 0.011
G4	G4	C1i	0.874 ± 0.010	0.542 ± 0.015	0.357 ± 0.039	0.591 ± 0.010
G4	G4	C2	0.858 ± 0.006	0.546 ± 0.011	0.357 ± 0.039	0.587 ± 0.011
G4	G4	G3.5t	0.863 ± 0.015	0.537 ± 0.015	0.357 ± 0.023	0.586 ± 0.010
G4	G4	G4	0.874 ± 0.010	0.566 ± 0.016	0.400 ± 0.043	0.613 ± 0.007

Table 6: Full APPS results, with GPT-4 as the Generator Model

G Model	F Model	R Model	Score
C2			0.666
C1i			0.628
G3.5t			0.686
G4			0.770
C2	C2	C2	0.689
C2	C1i	C2	0.714
C2	G3.5t	C2	0.719
C2	G4	C2	0.763
C1i	C2	C1i	0.684
C1i	C1i	C1i	0.699
C1i	G3.5t	C1i	0.699
C1i	G4	C1i	0.735
G3.5t	C2	G3.5t	0.661
G3.5t	C1i	G3.5t	0.645
G3.5t	G3.5t	G3.5t	0.643
G3.5t	G4	G3.5t	0.684
G4	C2	G4	0.755
G4	C1i	G4	0.773
G4	G3.5t	G4	0.778
G4	G4	G4	0.783

Table 7: MBPP results

B DS-1000

B.1 DS-1000 PROMPT

Listing 4: Initial Solution Prompt

```
{System:  You are a Python programmer, write code begin from BEGIN SOLUTION, do not include appeared example
      in your output.  Output in raw Python code only, do not include any other text.  Wrap the code with
      <code> and </code>.}
{Question}

    ANSWER:
```

Listing 5: Feedback Prompt

```
{DS1000 Question}

---Question---

\{QUESTION\}

---Initial Code---

{INITIAL_CODE}

---Test Code---

{TEST_CODE}


Please helping me debug the above program, which has some errors and is not passing the tests. Please give a
      concise (at most 2-3 sentences) textual explanation of what is wrong with the code. Do *not* generate
      any code, because I want to fix the code myself.
```

Listing 6: Refinement Prompt

```
{DS1000 Question}

---Question---

\{QUESTION\}

---Initial Code---

{INITIAL_CODE}

---Feedback---

{FEEDBACK}

---Instruction for Revision---

Based on the above specifications, please revise the incorrect Python code to a correct implementation. Please
       take the above feedback into consideration.

You are a refining code solution provider, please provide a refined code solution based on the Initial code
      solution and the Feedback provided. Output the reason and the refined code solution. Please do not
      include the Question in your response.
```

B.2 DS-1000 EXAMPLE

Init Stage

SYSTEM:

You are a Python programmer, write code begin from BEGIN SOLUTION, do not include appeared example in your output. Output in raw Python code only, do not include any other text. Wrap the code with `<code>` and `</code>`.

SAMPLE PROBLEM TEXT: How do I get the dimensions of an array? `import numpy as np a = np.array([[1,2],[3,4]])` BEGIN SOLUTION `[insert]` END SOLUTION `print(result)`

SAMPLE ANSWER TEXT: `print(a.shape)`

USER:

Problem:

Is it possible in PyTorch to change the learning rate of the optimizer in the middle of training dynamically (I don't want to define a learning rate schedule beforehand)?

So let's say I have an optimizer: `optim = torch.optim.SGD(..., lr=0.01)` Now due to some tests which I perform during training, I realize my learning rate is too high so I want to change it to say 0.001. There doesn't seem to be a method `optim.set_lr(0.001)` but is there some way to do this?

A:

`<code>`

import numpy as np

import pandas as pd

import torch

optim = load_data()

`</code>`

BEGIN SOLUTION

`<code>`

INIT MODEL: `for g in optim.param_groups:  g['lr'] = 0.001`

Feedback Stage

SYSTEM: You are a feedback provider, please provide feedback based on the initial code solution. Output the reason and the feedback. Please do not include the Question in your response.

USER:

—Question—

{QUESTION}

—Initial Code—

{INITIAL_CODE}

—Test Code—

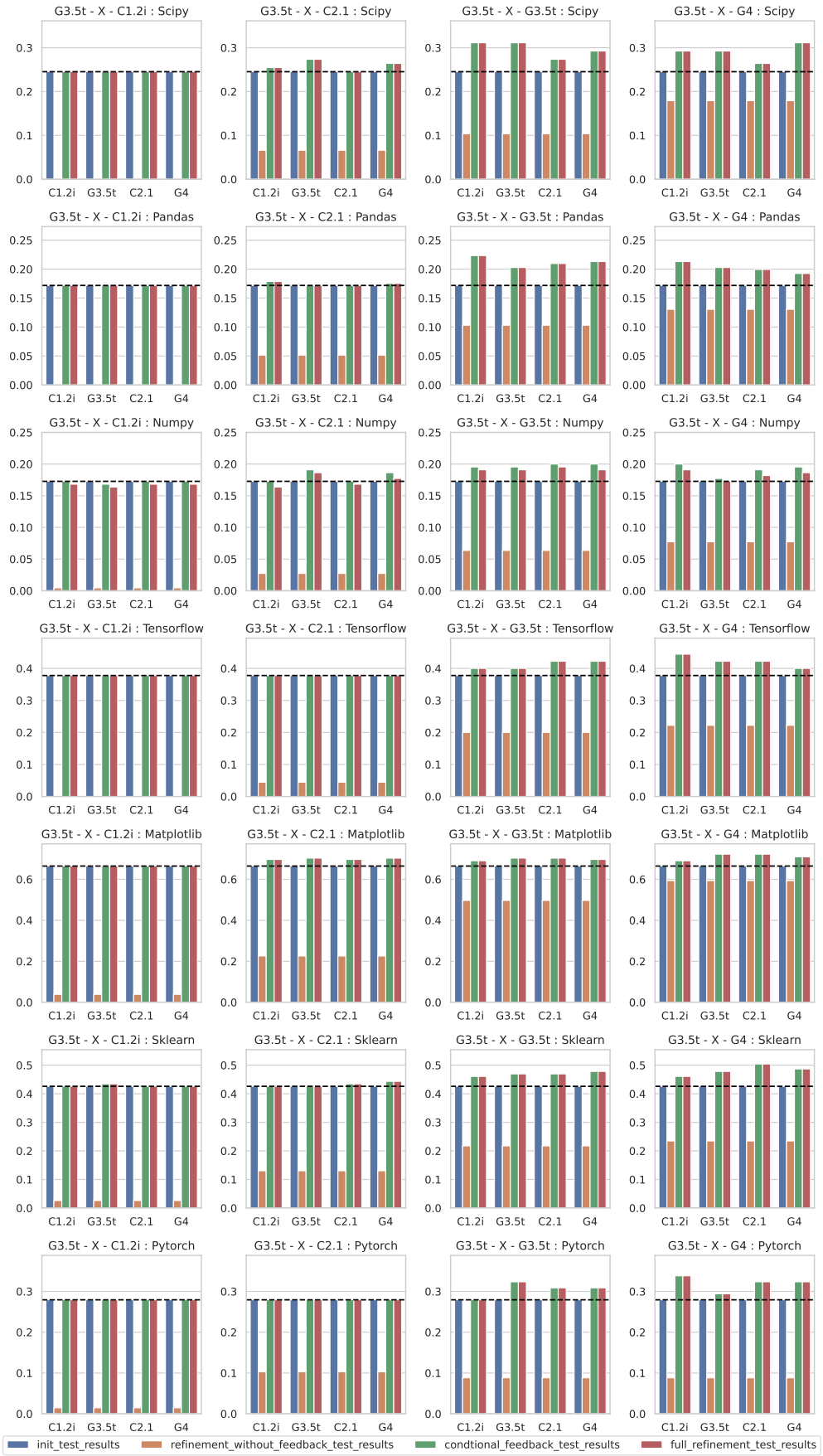
{TEST_CODE}

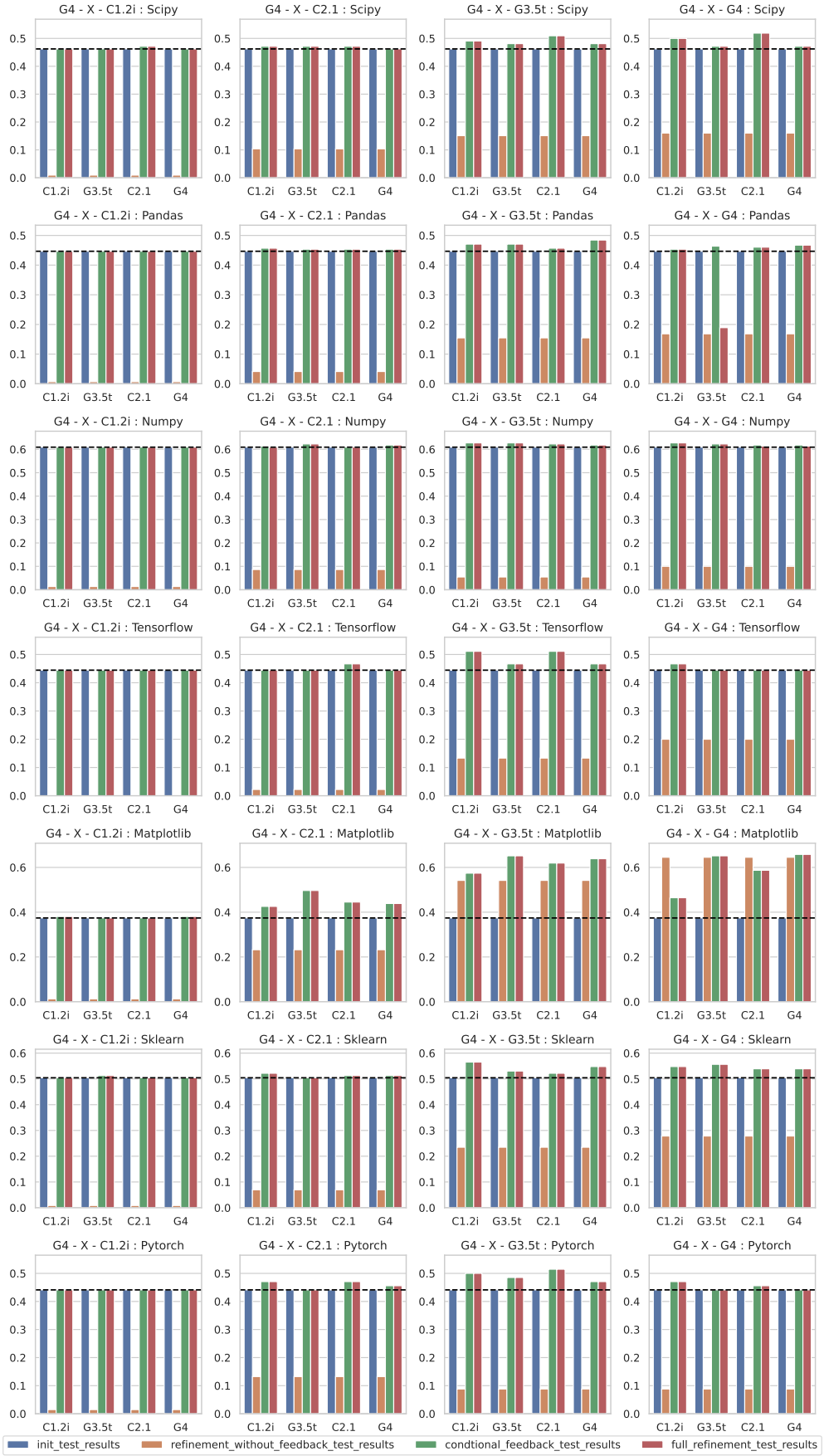
—Instruction for Feedback— Please helping me debug the above program, which has some errors and is not passing the tests. Please give a concise (at most 2-3 sentences) textual explanation of what is wrong with the code. Do *not* generate any code as solution, because I want to fix the code myself.

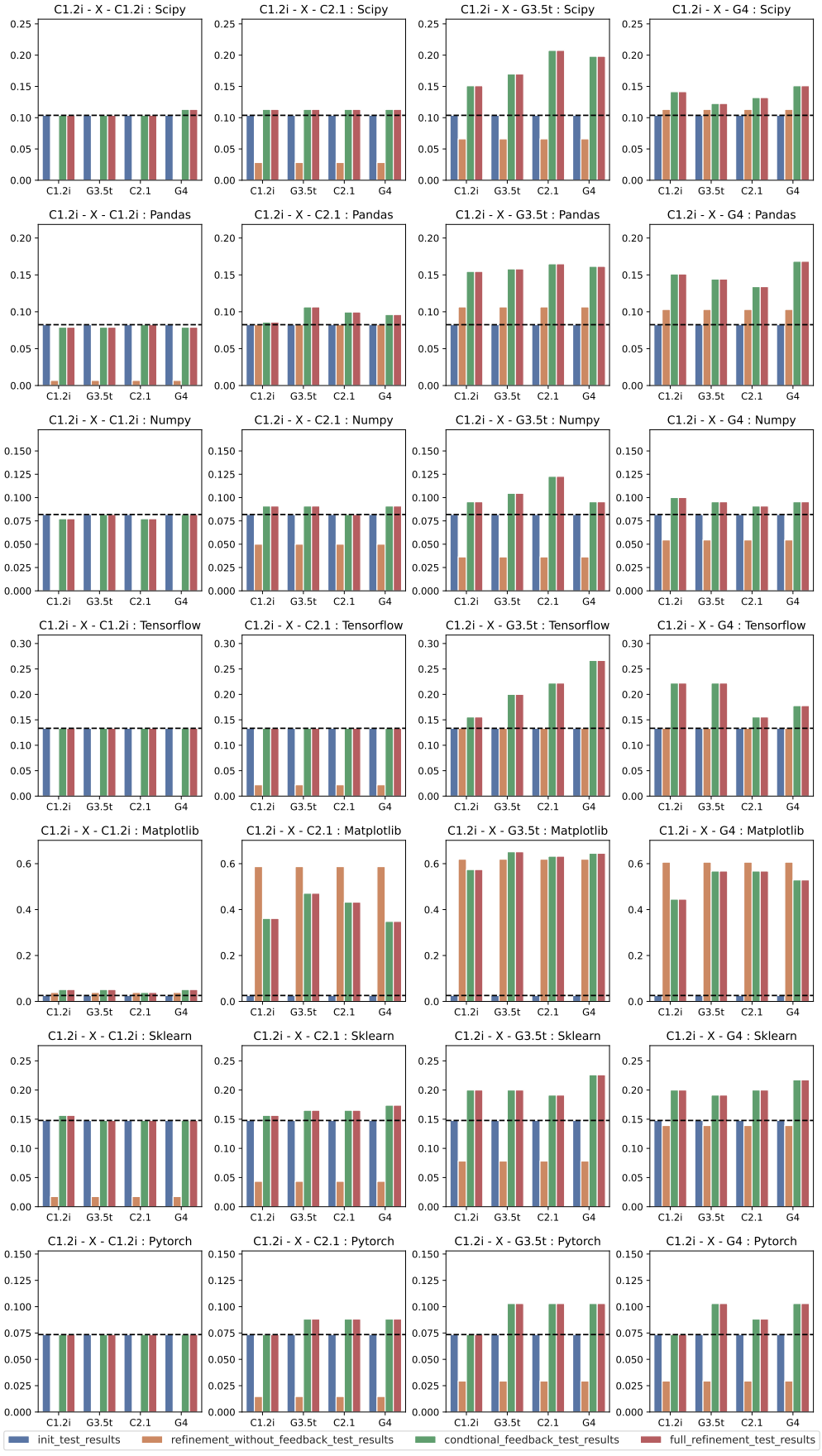
FEEDBACK:

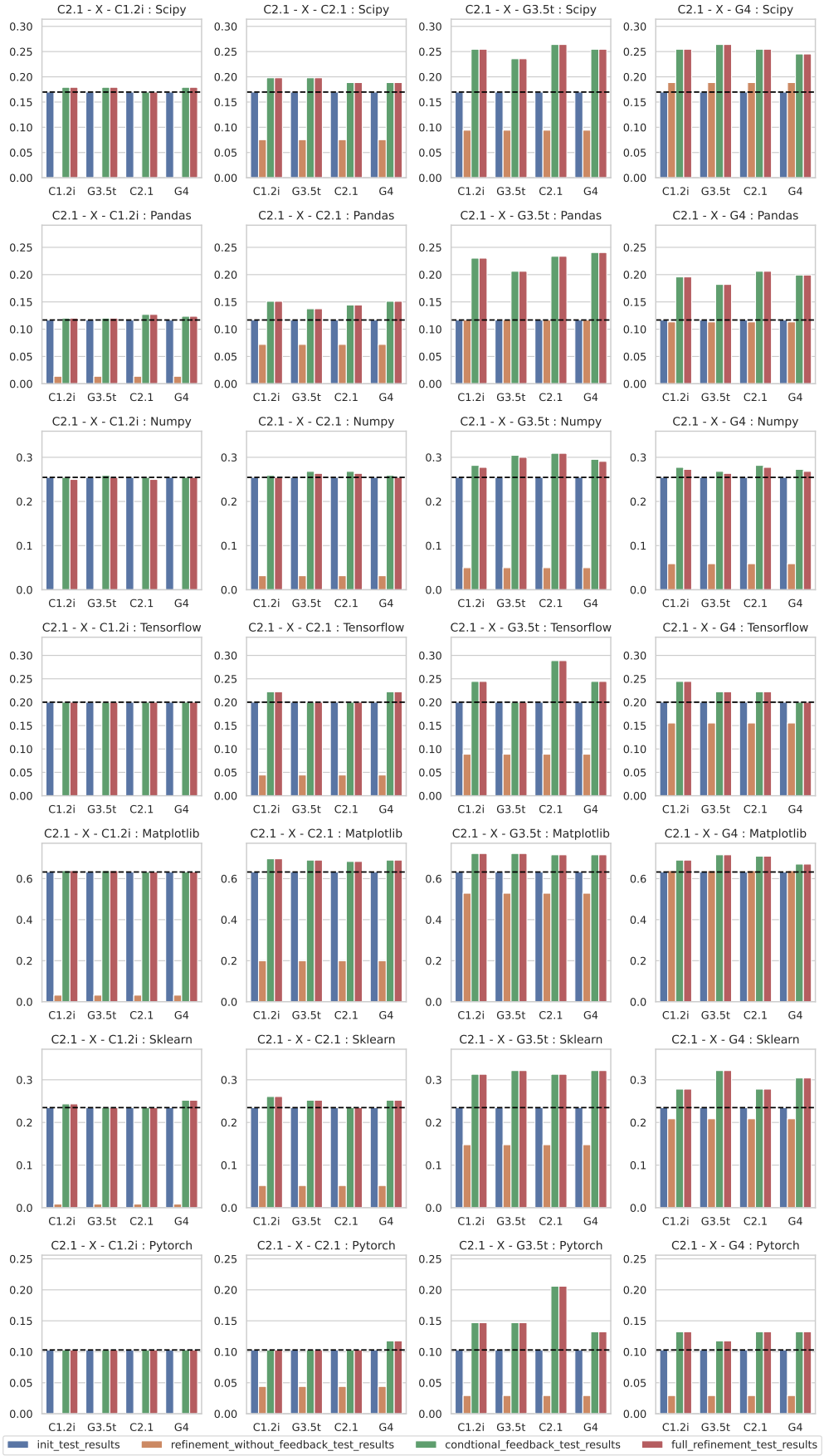
B.3 DS-1000 RESULTS

To investigate how initial model and the refinement model affects the pipeline performance, we vary the initial model and feedback in the DS-1000 dataset. The results are shown in the following figures.


G3.5t - X - C1.2i : Scipy
G3.5t - X - C2.1 : Scipy
G3.5t - X - G3.5t : Scipy
G3.5t - X - G4 : Scipy
G3.5t - X - C1.2i : Pandas
G3.5t - X - C2.1 : Pandas
G3.5t - X - G3.5t : Pandas
G3.5t - X - G4 : Pandas
G3.5t - X - C1.2i : Numpy
G3.5t - X - C2.1 : Numpy
G3.5t - X - G3.5t : Numpy
G3.5t - X - G4 : Numpy
G3.5t - X - C1.2i : Tensorflow
G3.5t - X - C2.1 : Tensorflow
G3.5t - X - G3.5t : Tensorflow
G3.5t - X - G4 : Tensorflow
G3.5t - X - C1.2i : Matplotlib
G3.5t - X - C2.1 : Matplotlib
G3.5t - X - G3.5t : Matplotlib
G3.5t - X - G4 : Matplotlib
G3.5t - X - C1.2i : Sklearn
G3.5t - X - C2.1 : Sklearn
G3.5t - X - G3.5t : Sklearn
G3.5t - X - G4 : Sklearn
G3.5t - X - C1.2i : Pytorch
G3.5t - X - C2.1 : Pytorch
G3.5t - X - G3.5t : Pytorch
G3.5t - X - G4 : Pytorch
C1.2i G3.5t C2.1 G4
init_test_results
refinement_without_feedback_test_results
condtional_feedback_test_results
full_refinement_test_results


G4 - X - C1.2i : Scipy
G4 - X - C2.1 : Scipy
G4 - X - G3.5t : Scipy
G4 - X - G4 : Scipy
G4 - X - C1.2i : Pandas
G4 - X - C2.1 : Pandas
G4 - X - G3.5t : Pandas
G4 - X - G4 : Pandas
G4 - X - C1.2i : Numpy
G4 - X - C2.1 : Numpy
G4 - X - G3.5t : Numpy
G4 - X - G4 : Numpy
G4 - X - C1.2i : Tensorflow
G4 - X - C2.1 : Tensorflow
G4 - X - G3.5t : Tensorflow
G4 - X - G4 : Tensorflow
G4 - X - C1.2i : Matplotlib
G4 - X - C2.1 : Matplotlib
G4 - X - G3.5t : Matplotlib
G4 - X - G4 : Matplotlib
G4 - X - C1.2i : Sklearn
G4 - X - C2.1 : Sklearn
G4 - X - G3.5t : Sklearn
G4 - X - G4 : Sklearn
G4 - X - C1.2i : Pytorch
G4 - X - C2.1 : Pytorch
G4 - X - G3.5t : Pytorch
G4 - X - G4 : Pytorch
C1.2i
G3.5t
C2.1
G4
init_test_results
refinement_without_feedback_test_results
condtional_feedback_test_results
full_refinement_test_results


C1.2i - X - C1.2i : Scipy
C1.2i - X - C2.1 : Scipy
C1.2i - X - G3.5t : Scipy
C1.2i - X - G4 : Scipy
C1.2i - X - C1.2i : Pandas
C1.2i - X - C2.1 : Pandas
C1.2i - X - G3.5t : Pandas
C1.2i - X - G4 : Pandas
C1.2i - X - C1.2i : Numpy
C1.2i - X - C2.1 : Numpy
C1.2i - X - G3.5t : Numpy
C1.2i - X - G4 : Numpy
C1.2i - X - C1.2i : Tensorflow
C1.2i - X - C2.1 : Tensorflow
C1.2i - X - G3.5t : Tensorflow
C1.2i - X - G4 : Tensorflow
C1.2i - X - C1.2i : Matplotlib
C1.2i - X - C2.1 : Matplotlib
C1.2i - X - G3.5t : Matplotlib
C1.2i - X - G4 : Matplotlib
C1.2i - X - C1.2i : Sklearn
C1.2i - X - C2.1 : Sklearn
C1.2i - X - G3.5t : Sklearn
C1.2i - X - G4 : Sklearn
C1.2i - X - C1.2i : Pytorch
C1.2i - X - C2.1 : Pytorch
C1.2i - X - G3.5t : Pytorch
C1.2i - X - G4 : Pytorch
C1.2i
G3.5t
C2.1
G4
init_test_results
refinement_without_feedback_test_results
condtional_feedback_test_results
full_refinement_test_results


C2.1 - X - C1.2i : Scipy
C2.1 - X - C2.1 : Scipy
C2.1 - X - G3.5t : Scipy
C2.1 - X - G4 : Scipy
C2.1 - X - C1.2i : Pandas
C2.1 - X - C2.1 : Pandas
C2.1 - X - G3.5t : Pandas
C2.1 - X - G4 : Pandas
C2.1 - X - C1.2i : Numpy
C2.1 - X - C2.1 : Numpy
C2.1 - X - G3.5t : Numpy
C2.1 - X - G4 : Numpy
C2.1 - X - C1.2i : Tensorflow
C2.1 - X - C2.1 : Tensorflow
C2.1 - X - G3.5t : Tensorflow
C2.1 - X - G4 : Tensorflow
C2.1 - X - C1.2i : Matplotlib
C2.1 - X - C2.1 : Matplotlib
C2.1 - X - G3.5t : Matplotlib
C2.1 - X - G4 : Matplotlib
C2.1 - X - C1.2i : Sklearn
C2.1 - X - C2.1 : Sklearn
C2.1 - X - G3.5t : Sklearn
C2.1 - X - G4 : Sklearn
C2.1 - X - C1.2i : Pytorch
C2.1 - X - C2.1 : Pytorch
C2.1 - X - G3.5t : Pytorch
C2.1 - X - G4 : Pytorch
C1.2i
G3.5t
C2.1
G4
init_test_results
refinement_without_feedback_test_results
condtional_feedback_test_results
full_refinement_test_results